ADSP-21161 SHARC® Processor Hardware Reference

Revision 4.0, February 2005

Part Number
82-001944-01

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

CONTENTS

MEMORY

EXTERNAL PORT

1 INTRODUCTION

Thank you for purchasing the Analog Devices SHARC® digital signal processor (DSP).

Design Advantages

The ADSP-21161 processor is a high-performance 32-bit processor used for medical imaging, communications, military, audio, test equipment, 3D graphics, speech recognition, motor control, imaging, and other applications. This processor builds on the ADSP-21000 Family processor core to form a complete system-on-a-chip, adding a dual-ported on-chip SRAM, integrated I/O peripherals, and an additional processing element for Single-Instruction-Multiple-Data (SIMD) support.

The SHARC architecture balances a high performance processor core with high performance buses (PM, DM, IO). In the core, every instruction can execute in a single cycle. The buses and instruction cache provide rapid, unimpeded data flow to the core to maintain the execution rate.

Figure 1-1 shows a detailed block diagram of the processor, which illustrates the following architectural features.

- Two processing elements (PEx and PEy), each containing 32-Bit IEEE floating-point computation unit—multiplier, ALU, Shifter, and data register file
- Program sequencer with related instruction cache, interval timer, and Data Address Generators (DAG1 and DAG2)

- Dual-ported SRAM
- External port for interfacing to off-chip memory such as SDRAM, peripherals, hosts, and multiprocessor systems
- Input/Output (IO) processor with integrated DMA controller, SPI-compatible port, serial ports, and link ports for point-to-point multiprocessor communications
- JTAG Test Access Port for emulation

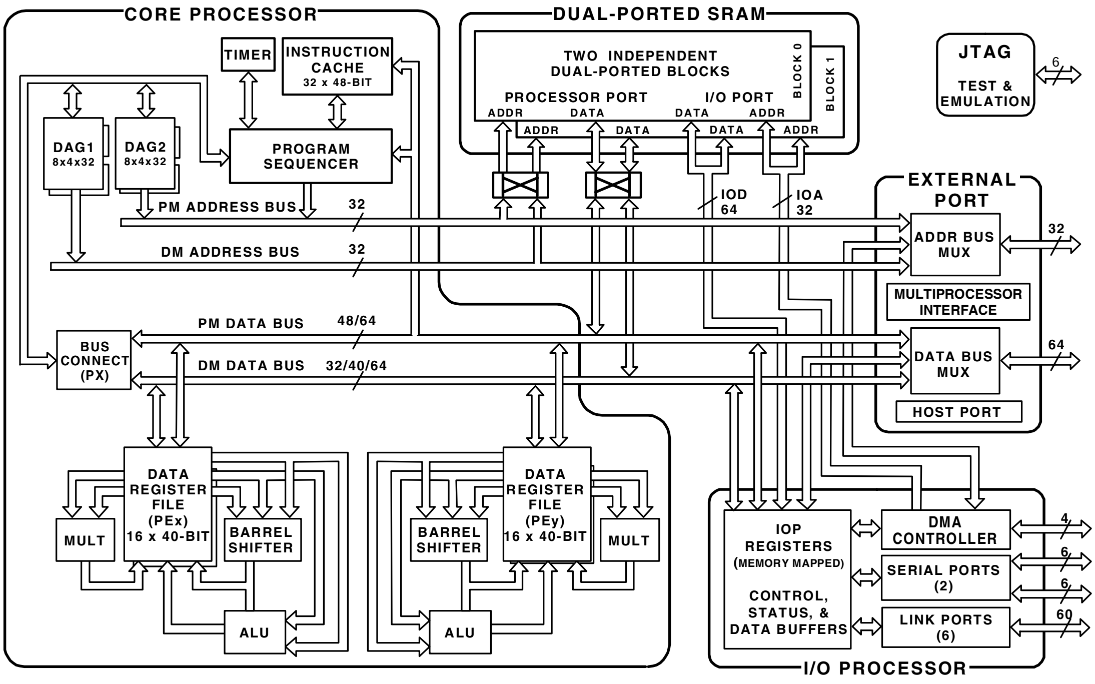


Figure 1-1. ADSP-21161 SHARC Block Diagram

Figure 1-1 also shows the three on-chip buses of the ADSP-21161 processor: the Program Memory (PM) bus, Data Memory (DM) bus, and Input/Output (IO) bus. The PM bus provides access to either instructions or data. During a single cycle, these buses let the processor access two data operands from memory, access an instruction (from the cache), and perform a DMA transfer.

The buses connect to the ADSP-21161 processor external port, which provides the processor interface to external memory, memory-mapped I/O, a host processor, and additional multiprocessing ADSP-21161 processors. The external port performs bus arbitration and supplies control signals to shared, global memory and I/O devices.

Figure 1-2 illustrates a typical single-processor system.

The ADSP-21161 processor includes extensive support for multiprocessor systems as well. For more information, see “Multiprocessor (MP) Interface” on page 7-87.

Further, the ADSP-21161 processor addresses the five central requirements for DSPs:

- Fast, flexible arithmetic computation units
- Unconstrained data flow to and from the computation units
- Extended precision and dynamic range in the computation units
- Dual address generators with circular buffering support
- Efficient program sequencing

Fast, Flexible Arithmetic. The ADSP-21000 Family processors execute all instructions in a single cycle. They provide fast cycle times and a complete set of arithmetic operations. The processor is IEEE floating-point compatible and allows either interrupt on arithmetic exception or latched status exception handling.

Unconstrained Data Flow. The ADSP-21161 processor has a Super Harvard Architecture combined with a 10-port data register file. In every cycle, the processor can write or read two operands to or from the register

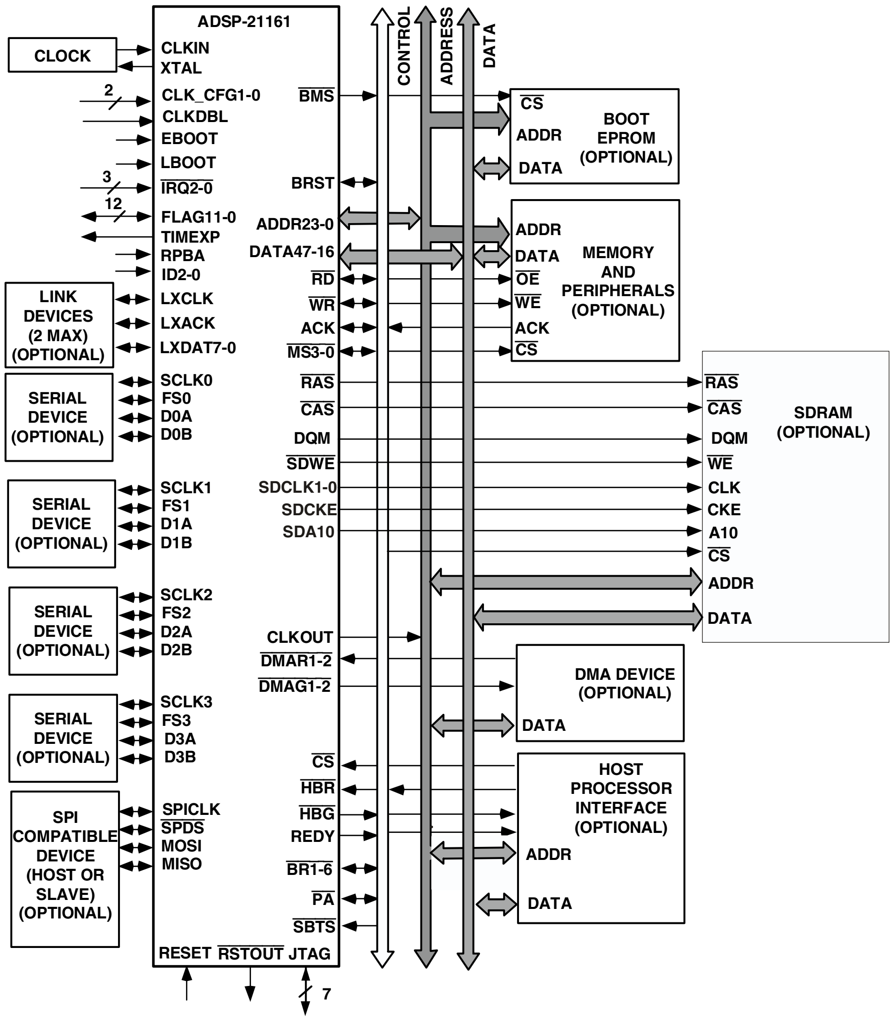


Figure 1-2. Typical Single Processor System

file, supply two operands to the ALU, supply two operands to the multi-plier, and receive three results from the ALU and multiplier. The processor's 48-bit orthogonal instruction word supports parallel data transfers and arithmetic operations in the same instruction.

40-Bit Extended Precision. The processor handles 32-bit IEEE floating-point format, 32-bit integer and fractional formats (twos-complement and unsigned), and extended-precision 40-bit floating-point format. The processors carry extended precision throughout their computation units, limiting intermediate data truncation errors.

Dual Address Generators. The processor has two Data Address Generators (DAGs) that provide immediate or indirect (pre- and post-modify) addressing. Modulus, bit-reverse, and broadcast operations are supported with no constraints on data buffer placement.

Efficient Program Sequencing. In addition to zero-overhead loops, the processor supports single-cycle setup and exit for loops. Loops are both nestable (six levels in hardware) and interruptable. The processors support both delayed and non-delayed branches.

Architecture Overview

The ADSP-21161 processor forms a complete system-on-a-chip, integrating a large, high-speed SRAM and I/O peripherals supported by a dedicated I/O bus. The following sections summarize the features of each functional block in the ADSP-21161 processor SHARC architecture, which appears in Figure 1-1 on page 1-2. With each summary, a cross reference points to the sections where the features are described in greater detail.

Processor Core

The processor core of the ADSP-21161 processor consists of two processing elements (each with three computation units and data register file), a program sequencer, two data address generators, a timer, and an instruction cache. All digital signal processing occurs in the processor core.

Processing Elements

The processor core contains two processing elements (PEx and PEy). Each element contains a data register file and three independent computation units: an ALU, a multiplier with a fixed-point accumulator, and a shifter. For meeting a wide variety of processing needs, the computation units process data in three formats: 32-bit fixed-point, 32-bit floating-point and 40-bit floating-point.

The floating-point operations are single-precision IEEE-compatible. The 32-bit floating-point format is the standard IEEE format, whereas the 40-bit extended-precision format has eight additional Least Significant Bits (LSBs) of mantissa for greater accuracy.

The ALU performs a set of arithmetic and logic operations on both fixed-point and floating-point formats. The multiplier performs floating-point or fixed-point multiplication and fixed-point multiply/add or multiply/subtract operations. The shifter performs logical and arithmetic shifts, bit manipulation, field deposit and extraction, and exponent derivation operations on 32-bit operands. These computation units perform single-cycle operations; there is no computation pipeline. All units are connected in parallel, rather than serially. The output of any unit may serve as the input of any unit on the next cycle. In a multifunction computation, the ALU and multiplier perform independent, simultaneous operations.

Each processing element has a general-purpose data register file that transfers data between the computation units and the data buses and stores intermediate results. A register file has two sets (primary and secondary) of sixteen registers each, for fast context switching. All of the registers are 40 bits wide. The register file, combined with the core processor's Super Harvard architecture, allows unconstrained data flow between computation units and internal memory.

Primary Processing Element (PEx). PEx processes all computational instructions whether the processor is in Single-Instruction, Single-Data (SISD) or Single-Instruction, Multiple-Data (SIMD) mode. This element corresponds to the computational units and register file in previous ADSP-21000 family DSPs.

Secondary Processing Element (PEy). PEy processes each computational instruction in lock-step with PEx, but only processes these instructions when the processor is in SIMD mode. Because many operations are influenced by this mode, more information on SIMD is available in multiple locations:

- For information on PEy operations, see "Processing Elements" on page 2-1
- For information on data addressing in SIMD mode, see "Addressing in SISD and SIMD Modes" on page 4-18
- For information on data accesses in SIMD mode, see "SISD, SIMD, and Broadcast Load Modes" on page 5-51
- For information on multiprocessing in SIMD mode, see "Multiprocessor (MP) Interface" on page 7-87
- For information on SIMD programming, see the *ADSP-21160 SHARC DSP Instruction Set Reference*

Program Sequence Control

Internal controls for ADSP-21161 processor program execution come from four functional blocks: program sequencer, data address generators, timer, and instruction cache. Two dedicated address generators and a program sequencer supply addresses for memory accesses. Together the sequencer and data address generators allow computational operations to execute with maximum efficiency since the computation units can be devoted exclusively to processing data. With its instruction cache, the

ADSP-21161 processor can simultaneously fetch an instruction from the cache and access two data operands from memory. The data address generators implement circular data buffers in hardware.

Program Sequencer. The program sequencer supplies instruction addresses to program memory. It controls loop iterations and evaluates conditional instructions. With an internal loop counter and loop stack, the ADSP-21161 processor executes looped code with zero overhead. No explicit jump instructions are required to loop or to decrement and test the counter.

The ADSP-21161 processor achieves its fast execution rate by means of pipelined fetch, decode, and execute cycles. If external memories are used, they are allowed more time to complete an access than if there were no decode cycle.

Data Address Generators. The Data Address Generators (DAGs) provide memory addresses when data is transferred between memory and registers. Dual data address generators enable the processor to output simultaneous addresses for two operand reads or writes. DAG1 supplies 32-bit addresses to data memory. DAG2 supplies 32-bit addresses to program memory for program memory data accesses.

Each DAG keeps track of up to eight address pointers, eight modifiers and eight length values. A pointer used for indirect addressing can be modified by a value in a specified register, either before (pre-modify) or after (post-modify) the access. A length value may be associated with each pointer to perform automatic modulo addressing for circular data buffers; the circular buffers can be located at arbitrary boundaries in memory. Each DAG register has a secondary register that can be activated for fast context switching.

Circular buffers allow efficient implementation of delay lines and other data structures required in digital signal processing, and are commonly used in digital filters and Fourier transforms. The DAGs automatically handle address pointer wraparound, reducing overhead, increasing performance, and simplifying implementation.

Interrupts. The ADSP-21161 processor has four external hardware interrupts: three general-purpose interrupts, $\overline{\texttt{IRQ}}\texttt{2-0}$, and a special interrupt for reset. The processor also has internally generated interrupts for the timer, DMA controller operations, circular buffer overflow, stack overflows, arithmetic exceptions, multiprocessor vector interrupts, and user-defined software interrupts.

For the general-purpose external interrupts and the internal timer interrupt, the ADSP-21161 processor automatically stacks the arithmetic status and mode (`MODE1`) registers in parallel with the interrupt servicing, allowing fifteen nesting levels of very fast service for these interrupts.

Context Switch. Many of the processor's registers have secondary registers that can be activated during interrupt servicing for a fast context switch. The data registers in the register file, the DAG registers, and the multiplier result register all have secondary registers. The primary registers are active at reset, while the secondary registers are activated by control bits in a mode control register.

Timer. The programmable interval timer provides periodic interrupt generation. When enabled, the timer decrements a 32-bit count register every cycle. When this count register reaches zero, the ADSP-21161 processor generates an interrupt and asserts its timer expired output. The count register is automatically reloaded from a 32-bit period register and the count resumes immediately.

Instruction Cache. The program sequencer includes a 32-word instruction cache that enables three-bus operation for fetching an instruction and two data values. The cache is selective; only instructions whose fetches conflict with program memory data accesses are cached. This caching allows full-speed execution of core, looped operations such as digital filter multiply-accumulates and FFT butterfly processing.

Processor Internal Buses

The processor core has six buses: PM address, PM data, DM address, DM data, IO address, and IO data. Due to processor's Super Harvard Architecture, data memory stores data operands, while program memory can store both instructions and data. This architecture allows dual data fetches, when the instruction is supplied by the cache.

Bus Capacities. The PM address bus and DM address bus transfer the addresses for instructions and data. The PM data bus and DM data bus transfer the data or instructions from each type of memory. the PM address bus is 32 bits wide, allowing access of up to 62 Mwords for non-SDRAM and 254 Mwords for SDRAM banks of mixed instructions and data. The PM data bus is 64 bits wide from (8-, 16-, and 32-bits) to accommodate the 48-bit instructions and 32-bit data.

The DM address bus is 32 bits wide allowing direct access of up to 4G words of data. The DM data bus is 64 bits wide. The DM data bus provides a path for the contents of any register in the processor to be transferred to any other register or to any data memory location in a single cycle. The data memory address comes from one of two sources: an absolute value specified in the instruction code (direct addressing) or the output of a data address generator (indirect addressing).

The IO address and IO data buses let the IO processor access internal memory for DMA without delaying the processor core. The IO address bus is 18 bits wide, and the IO data bus is 64 bits wide.

Data Transfers. Nearly every register in the processor core is classified as a Universal Register (UREG). Instructions allow transferring data between any two universal registers or between a universal register and memory. This support includes transfers between control registers, status registers, and data registers in the register file. The PM bus connect (`PX`) registers permit data to be passed between the 64-bit PM data bus and the 64-bit DM data bus, or between the 40-bit register file and the PM data bus. These registers contain hardware to handle the data width difference. For more information, see For more information, see "Processing Element Registers" on page A-23.

Processor Peripherals

The term *processor peripherals* refers to everything outside the processor core. The ADSP-21161 processor peripherals include internal memory, external port, I/O processor, JTAG port, and any external devices that connect to the processor.

Dual-Ported Internal Memory (SRAM)

The ADSP-21161 processor contains 1 megabit of on-chip SRAM, organized as two blocks of 0.5 Mbits. Each block can be configured for different combinations of code and data storage. Each memory block is dual-ported for single-cycle, independent accesses by the core processor and I/O processor or DMA controller. The dual-ported memory and separate on-chip buses allow two data transfers from the core and one from I/O, all in a single cycle.

All of the memory can be accessed as 16-, 32-, 48-, or 64-bit words. On the ADSP-21161 processor, the memory can be configured as a maximum of 32K words of 32-bit data, 64K words of 16-bit data, 21.25K words of 48-bit instructions (and 40-bit data), or combinations of different word sizes up to 1.0 Mbit.

The processor supports a 16-bit floating-point storage format, which effectively doubles the amount of data that may be stored on chip. Conversion between the 32-bit floating-point and 16-bit floating-point formats completes in a single instruction.

While each memory block can store combinations of code and data, accesses are most efficient when one block stores data, using the DM bus for transfers, and the other block stores instructions and data, using the PM bus for transfers. Using the DM bus and PM bus in this way, with one dedicated to each memory block, assures single-cycle execution with two data transfers. In this case, the instruction must be available in the cache. The processor uses its external port to maintain single-cycle execution when one of the data operands is transferred to or from off-chip.

External Port

The ADSP-21161 processor external port provides the processor interface to off-chip memory and peripherals. The 254 Mword off-chip address space is included in the unified address space of the ADSP-21161 processor. The separate on-chip buses—for PM address, PM data, DM address, DM data, IO address, and IO data—multiplex at the external port to create an external system bus with a single 24-bit address bus and a single 32-bit data bus. The ADSP-21161 processor on-chip DMA controller automatically packs external data into the appropriate word width during transfers.

The ADSP-21161 processor supports instruction packing modes to execute from 48-, 32-, 16-, and 8-bit wide memories. With the link ports disabled, the additional link port pins can be used to execute 48-bit wide instructions. The ADSP-21161 processor also includes 32- to 48-bit, 16- to 48-bit, 8- to 48-bit execution packing for executing instruction directly from 32-bit, 16-bit, or 8-bit wide external memories. External SDRAM, SRAM, or SBSRAM can be 8-, 16-, or 32-bits wide for DMA transfers to or from external memory.

On-chip decoding of high-order address lines generates memory bank select signals for addressing external memory devices. The ADSP-21161 processor provides programmable memory waitstates and external memory acknowledge controls for interfacing to peripherals with variable access, hold, and disable time requirements.

SDRAM Interface. The ADSP-21161 processor integrated on-chip SDRAM controller transfers data to and from synchronous DRAM (SDRAM) at the core clock frequency or one-half the core clock frequency. The synchronous approach, coupled with the core clock frequency, supports data transfer at a high throughput—up to 400 Mbytes/second for 32-bit transfers and 600 Mbytes/second for 48-bit transfers.

The SDRAM interface provides a glueless interface with standard SDRAMs—16 Mbits, 64 Mbits, 128 Mbits, and 256 Mbits—and includes options to support additional buffers between the ADSP-21161 processor and SDRAM. The SDRAM interface is extremely flexible and provides capability for connecting SDRAMs to any one of the ADSP-21161 processor four external memory banks, with up to all four banks mapped to SDRAM.

Systems with several SDRAM devices connected in parallel may require buffering to meet overall system timing requirements. The ADSP-21161 processor supports pipelining of the address and control signals to enable such buffering between itself and multiple SDRAM devices.

Host Processor Interface. The ADSP-21161 processor host interface allows easy connection to standard microprocessor buses, 8-bit, 16-bit and 32-bit, with little additional hardware required. The interface supports asynchronous and synchronous transfers at speeds up to the half the internal core clock rate of the ADSP-21161 processor. The host interface operates through the ADSP-21161 processor external port and maps into the unified address space. Four channels of DMA are available for the host interface; code and data transfers occur with low software overhead. The host can directly read and write the IOP register space of the ADSP-21161 processor and can access the DMA channel setup and mailbox registers. The host can also perform DMA transfers to and from the internal memory of the processor. Vector interrupt support provides for efficient execution of host commands.

Multiprocessor System Interface. The ADSP-21161 processor offers powerful features tailored to multiprocessing systems. The unified address space allows direct interprocessor accesses of each ADSP-21161 processor internal IOP registers. Distributed bus arbitration logic on the processor allows simple, glueless connection of systems containing up to six ADSP-21161 processor and a host processor. Master processor changeover incurs only one cycle of overhead. Bus arbitration handles either fixed or rotating priority. Processor bus lock allows indivisible read-modify-write sequences for semaphores. A vector interrupt capability is provided for interprocessor commands.

I/O Processor

The ADSP-21161 processor Input/Output Processor (IOP) includes four serial ports, two link ports, a SPI-compatible port, and a DMA controller. One of the processes that the IO processor automates is booting. The processor can boot from the external port (with data from an 8-bit EPROM or a host processor) or a link port. Alternatively, a no-boot mode lets the processor start by executing instructions from external memory without booting.

Serial Ports. The ADSP-21161 processor features four synchronous serial ports that provide an inexpensive interface to a wide variety of digital and mixed-signal peripheral devices. The serial ports can operate at up to half the processor core clock rate. Programmable data direction provides greater flexibility for serial communications. Serial port data can automatically transfer to and from on-chip memory using DMA. Each of the serial ports offers a TDM multichannel mode (up to 128 channels) and supports m-law or A-law companding. I^2S support is also provided with the ADSP-21161 processor.

The serial ports can operate with little-endian or big-endian transmission formats, with word lengths from 3 to 32 bits. The serial ports offer selectable synchronization and transmit modes. Serial port clocks and frame syncs can be internally or externally generated.

Link Ports. The ADSP-21161 processor features two 8-bit link ports that provide additional I/O capabilities. Link port I/O is especially useful for point-to-point interprocessor communication in multiprocessing systems. The link ports can operate independently and simultaneously. The data packs into 32-bit or 48-bit words, which the processor core can directly read or the IO processor can DMA-transfer to on-chip memory. Clock and acknowledge handshaking signals control link port transfers. Transfers are programmable as either transmit or receive.

Serial Peripheral (Compatible) Interface. The ADSP-21161 processor Serial Peripheral Interface (SPI) is an industry standard synchronous serial link that enables the ADSP-21161 processor SPI-compatible port to communicate with other SPI-compatible devices. SPI is a 4-wire interface consisting of two data pins, one device select pin, and one clock pin. It is a full-duplex synchronous serial interface, supporting both master and slave modes. It can operate in a multi-master environment by interfacing with up to four other SPI-compatible devices, either acting as a master or slave device. The ADSP-21161 processor SPI-compatible peripheral implementation also supports programmable baud rate and clock phase/polarities, and the use of open drain drivers to support the multi-master scenario to avoid data contention.

DMA Controller. The ADSP-21161 processor on-chip DMA controller allows zero-overhead data transfers without processor intervention. The DMA controller operates independently and invisibly to the processor core, allowing DMA operations to occur while the core is simultaneously executing its program. Both code and data can be downloaded to the ADSP-21161 processor using DMA transfers.

DMA transfers can occur between the ADSP-21161 processor internal memory and external memory, external peripherals, or a host processor. DMA transfers between external memory and external peripheral devices are another option. External bus packing to 8-, 16-, 32-, 48-, or 64-bit words is automatically performed during DMA transfers.

Fourteen channels of DMA are available on the ADSP-21161 processor—two over the link ports (shared with SPI), eight over the serial ports, and four over the processor's external port. The external port DMA channels serve for host processor, other ADSP-21161 processor DSPs, memory, or I/O transfers.

JTAG Port

The JTAG port on the ADSP-21161 processor supports the IEEE standard 1149.1 Joint Test Action Group (JTAG) standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system. Emulators use the JTAG port to monitor and control the processor during emulation. Emulators using this port provide full-speed emulation with access to inspect and modify memory, registers, and processor stacks. JTAG-based emulation is non-intrusive and does not effect target system loading or timing.

Differences From Previous SHARC Processors

This section identifies differences between the ADSP-21161 processor and previous SHARC processors: ADSP-21160, ADSP-21060, ADSP-21061, ADSP-21062, and ADSP-21065. The ADSP-21161 processor preserves much of the ADSP-2106x architecture and is comparable to the ADSP-21160 with extended performance and functionality. For background information on SHARC and the ADSP-2106x Family processors, see the *ADSP-2106x SHARC User's Manual* or the *ADSP-21065L SHARC Technical Reference*.

Processor Core Enhancements

Computational bandwidth on the ADSP-21161 processor is significantly greater than that on the ADSP-2106x DSPs. The increase comes from raising the operational frequency and adding another processing element: ALU, shifter, multiplier, and register file. The new processing element lets the processor to process multiple data streams in parallel (SIMD mode).

Like the ADSP-21160, the program sequencer on the ADSP-21161 processor differs from the ADSP-2106x family, having several enhancements: new interrupt vector table definitions, SIMD mode stack and conditional execution model, and instruction decodes associated with new instructions. Interrupt vectors have been added that detect illegal memory accesses. Link port interrupt control has moved to a new register to support the additional DMA channels. Also, mode stack and mode mask support has been added to improve context switch time.

As with the ADSP-21160, the data address generators on the ADSP-21161 processor differ from the ADSP-2106x in that DAG2 (for the PM bus) has the same addressing capability as DAG1 (for the DM bus). The DAG registers move 64-bits per cycle. Additionally, the DAGs support the new memory map and Long Word transfer capability. Circular buffering on the ADSP-21161 processor can be quickly disabled on interrupts and restored on the return. Data “broadcast”, from one memory location to both data register files, is determined by appropriate index register usage.

Processor Internal Bus Enhancements

The PM, DM, and IO data buses on the ADSP-21161 processor have increased on the ADSP-2106x processors to 64 bits. Additional multiplexing and control logic on the ADSP-21161 processor enable 16-, 32-, or 64-bit wide moves between both register files and memory. The ADSP-21161 processor is capable of broadcasting a single memory location to each of the register files in parallel. Also, the ADSP-21161 processor permits register contents to be exchanged between the two processing elements' register files in a single cycle.

Memory Organization Enhancements

The ADSP-21161 processor memory map differs from the ADSP-2106x's and is similar in organization to the ADSP-21160. The system memory map on the ADSP-21161 processor supports double-word transfers each cycle, reflects extended internal memory capacity for derivative designs, and works with updated control register for SIMD support.

External Port Enhancements

The ADSP-21161 processor external port differs from the ADSP-2106x DSPs. The data bus on the ADSP-21160 is 32-bits wide. A new packing mode permits DMA for instructions and data to and from 8-bit external memory. The ADSP-21161 processor has a new synchronous interface that improves local bus switching frequency. Also, burst support on the ADSP-21161 processor improves bus usage.

Host Interface Enhancements

The ADSP-21161 processor host interface differs from the ADSP-2106x DSPs. It is 32-bit wide and supports 8-bit, 16- and 32-bit hosts. Although the ADSP-21161 processor supports the ADSP-2106x asynchronous host interface protocols, the ADSP-21161 processor also provides new synchronous interface protocols for maximum throughput.

The host/local bus deadlock resolution function on the ADSP-21161 processor is extended to the DMA controller. With this function the host (or bridge) logic forces the local bus to wait until the host completes it's operation.

Multiprocessor Interface Enhancements

The ADSP-21161 processor multiprocessor system interface supports greater throughput than the ADSP-2106x DSPs. The throughput between ADSP-21161 processors in a multiprocessing application increases due to new shared bus transfer protocols, shared bus cycle time improvements due to synchronous interface, and improvements in link port throughput. The external port supports glueless multiprocessing, with distributed arbitration for up to six ADSP-21161 processors.

IO Architecture Enhancements

The IO processor on the ADSP-21161 processor provides much greater throughput than the ADSP-2106x DSPs. This section describes how the link ports and DMA controller differ on the ADSP-21161 processor.

DMA Controller Enhancements

The ADSP-21161 processor DMA controller supports 14 channels compared to 10 on the ADSP-2106x DSPs. New packing modes support the 64-bit internal busing. To resolve potential deadlock scenarios, the ADSP-21161 processor DMA controller relinquishes the local bus in a similar fashion to the processor core when host logic asserts both $\overline{\texttt{HBR}}$ and $\overline{\texttt{SBTS}}$.

Link Port Enhancements

The ADSP-21161 processor two link ports provide greater throughput than the ADSP-2106x DSPs. The link port data bus width on the ADSP-21161 processor is 8 bits wide (versus 4 bits on the ADSP-2106x DSPs). Link port clock control on the ADSP-21161 processor supports a wider frequency range.

Instruction Set Enhancements

ADSP-21161 processor provides source code compatibility with the previous SHARC family members, to the application assembly source code level. All instructions, control registers, and system resources available in the ADSP-2106x core programming model are available in ADSP-21161 processor. Instructions, control registers, or other facilities, required to support the new feature set of ADSP-2116x core include the following.

- Code compatible to the ADSP-21160 SIMD core
- Supersets of the ADSP-2106x programming model
- Reserved facilities in the ADSP-2106x programming model
- Symbol name changes from the ADSP-2106x and ADSP-21161 processor programming models

These name changes can be managed through re-assembly using the ADSP-21161 processor development tools to apply the ADSP-21161 processor symbol definitions header file and linker description file. While these changes have no direct impact on existing core applications, system and I/O processor initialization code and control code do require modifications.

Although the porting of source code written for the ADSP-2106x family to ADSP-21161 processor has been simplified, code changes are required to take full advantage of the new ADSP-21161 processor features. For more information, see the *ADSP-21160 SHARC DSP Instruction Set Reference*.

For More Information About Analog Products

Analog Devices is online on the internet at http://www.analog.com. Our web pages provide information on the company and products, including access to technical information and documentation, product overviews, and product announcements.

Additional information may be obtained about Analog Devices and its products in any of the following ways:

- Visit our World Wide Web site at www.analog.com
- FAX questions or requests for information to 1(781)461-3010.
- Access the Computer Products Division File Transfer Protocol (FTP) site at ftp ftp.analog.com or ftp 137.71.23.21 or ftp://ftp.analog.com

For Technical or Customer Support

Our Customer Support group can be reached in the following ways:

- E-mail questions to dsp.support@analog.com (hardware support), dsptools.support@analog.com (software support) or dsp.europe@analog.com (European customer support).
- Contact your local ADI sales office or an authorized ADI distributor
- Send questions by mail to:

```
Analog Devices, Inc.
DSP Division
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA
```

What's New in This Manual

The fourth edition of the *ADSP-21161 SHARC Processor Hardware Reference* is revised based on the published document errata.

Related Documents

For more information about Analog Devices DSPs and development products, see the following documents:

- *ADSP-21161 SHARC DSP Microcomputer Data Sheet*
- *ADSP-21160 SHARC DSP Instruction Set Reference*
- *Getting Started Guide for VisualDSP++ & ADSP-21xxx Family DSPs*
- *VisualDSP++ User's Guide for ADSP-21xxx Family DSPs*
- *C/C++ Compiler & Library Manual for ADSP-21xxx Family DSPs*
- *Assembler Manual for ADSP-21xxx Family DSPs*
- *Linker & Utilities Manual for ADSP-21xxx Family DSPs*

All the manuals are included in the software distribution CD-ROM. To access these manuals, use the Help Topics command in the VisualDSP++ environment's Help menu and select the Online Manuals book. From this Help topic, you can open any of the manuals, which are in Adobe Acrobat PDF format.

Conventions

The following are conventions that apply to all chapters. Note that additional conventions, which apply only to specific chapters, appear throughout this document.

Table 1-1. Notation Conventions

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
filename	Non-keyword placeholders appear in text with italic style format.
ⓘ	**Note:** For correct operation, ... A Note: provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
🛇	**Warning:** Injury to device users may result if ... A Warning: identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

2 PROCESSING ELEMENTS

The processor's Processing Elements (PEx and PEy) perform numeric processing for digital signal processing algorithms. Each processing element contains a data register file and three computation units: an arithmetic/logic unit (ALU), a multiplier, and a shifter. Computational instructions for these elements include both fixed-point and floating-point operations, and each computational instruction can execute in a single cycle.

The computational units in a processing element handle different types of operations. The ALU performs arithmetic and logic operations on fixed-point and floating-point data. The multiplier does floating-point and fixed-point multiplication and executes fixed-point multiply/add and multiply/subtract operations. The shifter completes logical shifts, arithmetic shifts, bit manipulation, field deposit, and field extraction operations on 32-bit operands. Also, the Shifter can derive exponents.

Data flow paths through the computational units are arranged in parallel, as shown in Figure 2-1. The output of any computation unit may serve as the input of any computation unit on the next instruction cycle. Data moving in and out of the computational units goes through a 10-port register file, consisting of sixteen primary registers and sixteen alternate registers. Two ports on the register file connect to the PM and DM data buses, allowing data transfer between the computational units and memory (and anything else) connected to these buses.

The processor's assembly language provides access to the data register files in both processing elements. The syntax lets programs move data to and from these registers and specify a computation's data format at the same time with naming conventions for the registers. For information on the data register names, see "Data Register File" on page 2-30.

Figure 2-1 provides a graphical guide to the other topics in this chapter. First, a description of the `MODE1` register shows how to set rounding, data format, and other modes for the processing elements. Next, an examination of each computational unit provides details on operation and a summary of computational instructions. Outside the computational units, details on register files and data buses identify how to flow data for computations. Finally, details on the processor's advanced parallelism reveal how to take advantage of multifunction instructions and SIMD mode.

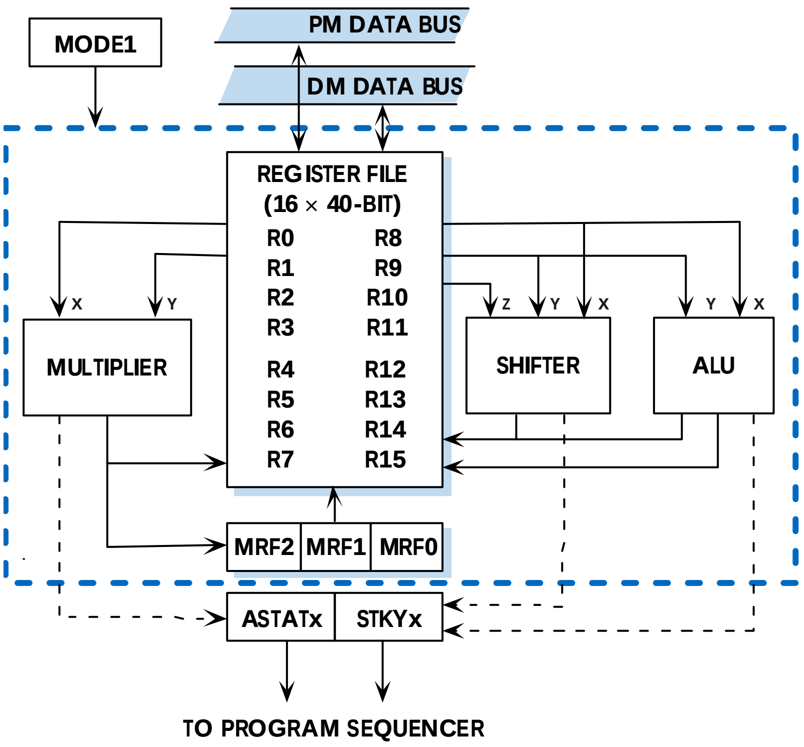


Figure 2-1. Computation Units

Setting Computational Modes

The MODE1 register controls the operating mode of the processing elements. Table A-2 on page A-3 lists all the bits in MODE1. The following bits in MODE1 control computational modes.

- **Floating-point data format.** Bit 16 (RND32) directs the computational units to round floating-point data to 32 bits (if 1) or round to 40 bits (if 0).
- **Rounding mode.** Bit 15 (TRUNC) directs the computational units to round results with round-to-zero (if 1) or round-to-nearest (if 0).
- **ALU saturation.** Bit 13 (ALUSAT) directs the computational units to saturate results on positive or negative fixed-point overflows (if 1) or return unsaturated results (if 0).
- **Short word sign extension.** Bit 14 (SSE) directs the computational units to sign extend short-word, 16-bit data (if 1) or zero-fill the upper 16 bits (if 0).
- **Secondary processor element (PEy).** Bit 21 (PEYEN) enables computations in PEy—SIMD mode—(if 1) or disables PEy—SISD mode—(if 0).

32-Bit (Normal Word) Floating-Point Format

In the default mode of the processor (RND32 bit=1), the multiplier and ALU support a single-precision floating-point format, which is specified in the IEEE 754/854 standard. For more information on this standard, see

For more information, see "Numeric Formats" on page C-1. This format is IEEE 754/854 compatible for single-precision floating-point operations in all respects except that:

- The processor does not provide inexact flags.
- NAN ("Not-A-Number") inputs generate an invalid exception and return a quiet NAN (all 1s).
- Denormal operands flush to zero when input to a computation unit and do not generate an underflow exception. Any denormal or underflow result from an arithmetic operation flushes to zero and generates an underflow exception.
- The processor supports round to nearest and round toward zero modes, but does not support round to +Infinity and round to -Infinity.

IEEE Single-precision floating-point data uses a 23-bit mantissa with an 8-bit exponent plus sign bit. In this case, the computation unit sets the eight LSBs of floating-point inputs to zeros before performing the operation. The mantissa of a result rounds to 23 bits (not including the hidden bit), and the 8 LSBs of the 40-bit result clear to zeros to form a 32-bit number, which is equivalent to the IEEE standard result.

In fixed-point to floating-point conversion, the rounding boundary is always 40 bits even if the `RND32` bit is set.

40-Bit Floating-Point Format

When in extended precision mode (`RND32` bit=0), the processor supports a 40-bit extended precision floating-point mode, which has eight additional LSBs of the mantissa and is compliant with the 754/854 standards; however, results in this format are more precise than the IEEE single-precision standard specifies. Extended-precision floating-point data uses a 31-bit mantissa with a 8-bit exponent plus sign bit.

16-Bit (Short Word) Floating-Point Format

The processor supports a 16-bit floating-point storage format and provides instructions that convert the data for 40-bit computations. The 16-bit floating-point format uses an 11-bit mantissa with a 4-bit exponent plus sign bit. The 16-bit data goes into bits 23 through 8 of a data register. Two shifter instructions, Fpack and Funpack, perform the packing and unpacking conversions between 32-bit floating-point words and 16-bit floating-point words. The Fpack instruction converts a 32-bit IEEE floating-point number in a data register into a 16-bit floating-point number. Funpack converts a 16-bit floating-point number in a data register into a 32-bit IEEE floating-point number. Each instruction executes in a single cycle.

When 16-bit data is written to bits 23 through 8 of a data register, the processor automatically extends the data into a 32-bit integer (bits 39 through 8). If the SSE bit in MODE1 is set (1), the processor sign extends the upper 16 bits. If the SSE bit is cleared (0), the processor zeros the upper 16 bits.

The 16-bit floating-point format supports gradual underflow. This method sacrifices precision for dynamic range. When packing a number that would have underflowed, the exponent clears to zero and the mantissa (including "hidden" 1) right-shifts the appropriate amount. The packed result is a denormal, which can be unpacked into a normal IEEE floating-point number.

32-Bit Fixed-Point Format

The processor always represents fixed-point numbers in 32 bits, occupying the 32 MSBs in 40-bit data registers. Fixed-point data may be fractional or integer numbers and unsigned or twos-complement. Each computational unit has its own limitations on how these formats may be mixed for a given operation. All computational units read the upper 32 bits of data (inputs, operands) from the 40-bit registers (ignoring the 8 LSBs) and write results to the upper 32 bits (zeroing the 8 LSBs).

Rounding Mode

The `TRUNC` bit in the `MODE1` register determines the rounding mode for all ALU operations, all floating-point multiplies, and fixed-point multiplies of fractional data. The processor supports two modes of rounding: round-toward-zero and round-toward-nearest. The rounding modes comply with the IEEE 754 standard and have the following definitions.

- **Round-Toward-Zero** (`TRUNC` **bit=1).** If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to zero. This definition is equivalent to truncation.

- **Round-Toward-Nearest** (`TRUNC` **bit=0).** If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to the result before rounding. If the result before rounding is exactly halfway between two numbers in the destination format (differing by an LSB), the rounded result is the number that has an LSB equal to zero.

Statistically, rounding up occurs as often as rounding down, so there is no large sample bias. Because the maximum floating-point value is one LSB less than the value that represents Infinity, a result that is halfway between the maximum floating-point value and Infinity rounds to Infinity in this mode.

Though these rounding modes comply with standards set for floating-point data, they also apply for fixed-point multiplier operations on fractional data. The same two rounding modes are supported, but only the round-to-nearest operation is actually performed by the multiplier. Using its local result register for fixed-point operations, the multiplier rounds-to-zero by reading only the upper bits of the result and discarding the lower bits.

Using Computational Status

The multiplier and ALU each provide exception information when executing floating-point operations. Each unit updates overflow, underflow, and invalid operation flags in the processing element's arithmetic status (ASTATx and ASTATy) register and sticky status (STKYx and STKYy) register. An underflow, overflow, or invalid operation from any unit also generates a maskable interrupt. There are three ways to use floating-point exceptions from computations in program sequencing:

- **Interrupts.** Enable interrupts and use an interrupt service routine to handle the exception condition immediately. This method is appropriate if it is important to correct all exceptions as they occur.
- ASTATx **and** ASTATy **registers.** Use conditional instructions to test the exception flags in the ASTATx or ASTATy register after the instruction executes. This method permits monitoring each instruction's outcome.
- STKYx **and** STKYy **registers.** Use the Bit Tst instruction to examine exception flags in the STKY register after a series of operations. If any flags are set, some of the results are incorrect. This method is useful when exception handling is not critical.

More information on ASTAT and STKY status appears in the sections that describe the computational units. For summaries relating instructions and status bits, see Table 2-1, Table 2-2, Table 2-4, Table 2-6, and Table 2-7.

Arithmetic Logic Unit (ALU)

The ALU performs arithmetic operations on fixed-point or floating-point data and logical operations on fixed-point data. ALU fixed-point instructions operate on 32-bit fixed-point operands and output 32-bit fixed-point results. ALU floating-point instructions operate on 32-bit or 40-bit floating-point operands and output 32-bit or 40-bit floating-point results. ALU instructions include:

- Floating-point addition, subtraction, add/subtract, average
- Fixed-point addition, subtraction, add/subtract, average
- Floating-point manipulation: binary log, scale, mantissa
- Fixed-point add with carry, subtract with borrow, increment, decrement
- Logical And, Or, Xor, Not
- Functions: Abs, pass, min, max, clip, compare
- Format conversion
- Reciprocal and reciprocal square root primitives

ALU Operation

ALU instructions take one or two inputs: X input and Y input. These inputs (also known as operands) can be any data registers in the register file. Most ALU operations return one result; in add/subtract operations, the ALU operation returns two results, and in compare operations, the ALU operation returns no result (only flags are updated). ALU results can be returned to any location in the register file.

The processor transfers input operands from the register file during the first half of the cycle and transfers results to the register file during the second half of the cycle. With this arrangement, the ALU can read and write the same register file location in a single cycle. If the ALU operation is fixed-point, the inputs are treated as 32-bit fixed-point operands. The ALU transfers the upper 32 bits from the source location in the register file. For fixed-point operations, the result(s) are always 32-bit fixed-point values. Some floating-point operations (Logb, Mant and Fix) can also yield fixed-point results.

The processor transfers fixed-point results to the upper 32 bits of the data register and clears the lower eight bits of the register. The format of fixed-point operands and results depends on the operation. In most arithmetic operations, there is no need to distinguish between integer and fractional formats. Fixed-point inputs to operations such as scaling a floating-point value are treated as integers. For purposes of determining status such as overflow, fixed-point arithmetic operands and results are treated as twos-complement numbers.

ALU Saturation

When the `ALUSAT` bit is set (1) in the `MODE1` register, the ALU is in saturation mode. In this mode, all positive fixed-point overflows return the maximum positive fixed-point number (0x7FFF FFFF), and all negative overflows return the maximum negative number (0x8000 0000).

When the `ALUSAT` bit is cleared (0) in the `MODE1` register, fixed-point results that overflow are not saturated; the upper 32 bits of the result are returned unaltered.

The ALU overflow flag reflects the ALU result before saturation.

ALU Status Flags

ALU operations update seven status flags in the processing element's Arithmetic Status (ASTATx and ASTATy) register. Table A-4 on page A-18 lists all the bits in these registers. The following bits in ASTATx or ASTATy flag ALU status (a 1 indicates the condition) for the most recent ALU operation:

- **ALU result zero or floating-point underflow.** Bit 0 (AZ)
- **ALU overflow.** Bit 1 (AV)
- **ALU result negative.** Bit 2 (AN)
- **ALU fixed-point carry.** Bit 3 (AC)
- **ALU X input sign** for Abs, Mant operations. Bit 4 (AS)
- **ALU floating-point invalid operation.** Bit 5 (AI)
- **Last ALU operation was a floating-point operation.** Bit 10 (AF)
- **Compare Accumulation register results** of last 8 compare operations. Bits 31-24 (CACC)

ALU operations also update four "sticky" status flags in the processing element's Sticky status (STKYx and STKYy) register. "Sticky Status Registers (STKYx and STKYy)" on page A-18 lists all the bits in these registers. The following bits in STKYx or STKYy flag ALU status (a 1 indicates the condition). Once set, a sticky flag remains high until explicitly cleared:

- **ALU floating-point underflow.** Bit 0 (AUS)
- **ALU floating-point overflow.** Bit 1 (AVS)
- **ALU fixed-point overflow.** Bit 2 (AOS)
- **ALU floating-point invalid operation.** Bit 5 (AIS)

Flag update occurs at the end of the cycle in which the status is generated and is available on the next cycle. If a program writes the arithmetic status register or sticky status register explicitly in the same cycle that the ALU is performing an operation, the explicit write to the status register supersedes any flag update from the ALU operation.

ALU Instruction Summary

Table 2-1 and Table 2-2 list the ALU instructions and how they relate to `ASTATx,y` and `STKYx,y` flags. For more information on assembly language syntax, see the *ADSP-21160 SHARC DSP Instruction Set Reference*. In these tables, note the meaning of the following symbols.

- **Rn, Rx, Ry** indicate any register file location; treated as fixed-point
- **Fn, Fx, Fy** indicate any register file location; treated as floating-point
- * indicates the flag may be set or cleared, depending on results of instruction
- ** indicates the flag may be set (but not cleared), depending on results of instruction
- – indicates no effect

Table 2-1. Fixed-Point ALU Instruction Summary

Instruction	ASTATx,y Status Flags							STKYx,y Status Flags				
Fixed-Point:	**AZ**	**AV**	**AN**	**AC**	**AS**	**AI**	**AF**	**CACC**	**AUS**	**AVS**	**AOS**	**AIS**
Rn = Rx + Ry	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx – Ry	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx + Ry + CI	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx – Ry + CI – 1	*	*	*	*	0	0	0	–	–	–	**	–
Rn = (Rx + Ry)/2	*	0	*	*	0	0	0	–	–	–	–	–
COMP(Rx, Ry)	*	0	*	0	0	0	0	*	–	–	–	–
COMPU(Rx,Ry)	*	0	*	0	0	0	0	*	--	--	--	--
Rn = Rx + CI	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx + CI – 1	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx + 1	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx – 1	*	*	*	*	0	0	0	–	–	–	**	–
Rn = –Rx	*	*	*	*	0	0	0	–	–	–	**	–
Rn = ABS Rx	*	*	0	0	*	0	0	–	–	–	**	–
Rn = PASS Rx	*	0	*	0	0	0	0	–	–	–	–	–
Rn = Rx AND Ry	*	0	*	0	0	0	0	–	–	–	–	–
Rn = Rx OR Ry	*	0	*	0	0	0	0	–	–	–	–	–
Rn = Rx XOR Ry	*	0	*	0	0	0	0	–	–	–	–	–
Rn = NOT Rx	*	0	*	0	0	0	0	–	–	–	–	–
Rn = MIN(Rx, Ry)	*	0	*	0	0	0	0	–	–	–	–	–
Rn = MAX(Rx, Ry)	*	0	*	0	0	0	0	–	–	–	–	–
Rn = CLIP Rx BY Ry	*	0	*	0	0	0	0	–	–	–	–	–

Table 2-2. Floating-Point ALU Instruction Summary

Instruction	ASTATx,y Status Flags							STKYx,y Status Flags				
Floating–Point:	AZ	AV	AN	AC	AS	AI	AF	CACC	AUS	AVS	AOS	AIS
Fn = Fx + Fy	*	*	*	0	0	*	1	–	**	**	–	**
Fn = Fx – Fy	*	*	*	0	0	*	1	–	**	**	–	**
Fn = ABS (Fx + Fy)	*	*	0	0	0	*	1	–	**	**	–	**
Fn = ABS (Fx – Fy)	*	*	0	0	0	*	1	–	**	**	–	**
Fn = (Fx + Fy)/2	*	0	*	0	0	*	1	–	**	–	–	**
COMP(Fx, Fy)	*	0	*	0	0	*	1	*	–	–	–	**
Fn = –Fx	*	*	*	0	0	*	1	–	–	**	–	**
Fn = ABS Fx	*	*	0	0	*	*	1	–	–	**	–	**
Fn = PASS Fx	*	0	*	0	0	*	1	–	–	–	–	**
Fn = RND Fx	*	*	*	0	0	*	1	–	–	**	–	**
Fn = SCALB Fx BY Ry	*	*	*	0	0	*	1	–	**	**	–	**
Rn = MANT Fx	*	*	0	0	*	*	1	–	–	**	–	**
Rn = LOGB Fx	*	*	*	0	0	*	1	–	–	**	–	**
Rn = FIX Fx BY Ry	*	*	*	0	0	*	1	–	**	**	–	**
Rn = FIX Fx	*	*	*	0	0	*	1	–	**	**	–	**
Fn = FLOAT Rx BY Ry	*	*	*	0	0	0	1	–	**	**	–	–
Fn = FLOAT Rx	*	0	*	0	0	0	1	–	–	–	–	–
Fn = RECIPS Fx	*	*	*	0	0	*	1	–	**	**	–	**
Fn = RSQRTS Fx	*	*	*	0	0	*	1	–	–	**	–	**
Fn = Fx COPYSIGN Fy	*	0	*	0	0	*	1	–	–	–	–	**
Fn = MIN(Fx, Fy)	*	0	*	0	0	*	1	–	–	–	–	**
Fn = MAX(Fx, Fy)	*	0	*	0	0	*	1	–	–	–	–	**
Fn = CLIP Fx BY Fy	*	0	*	0	0	*	1	–	–	–	–	**

Multiply—Accumulator (Multiplier)

The multiplier performs fixed-point or floating-point multiplication and fixed-point multiply/accumulate operations. Fixed-point multiply/accumulates are available with either cumulative addition or cumulative subtraction. Multiplier floating-point instructions operate on 32-bit or 40-bit floating-point operands and output 32-bit or 40-bit floating-point results. Multiplier fixed-point instructions operate on 32-bit fixed-point data and produce 80-bit results. Inputs are treated as fractional or integer, unsigned or twos-complement. Multiplier instructions include:

- Floating-point multiplication
- Fixed-point multiplication
- Fixed-point multiply/accumulate with addition, rounding optional
- Fixed-point multiply/accumulate with subtraction, rounding optional
- Rounding result register
- Saturating result register
- Clearing result register

Multiplier Operation

The multiplier takes two inputs: X input and Y input. These inputs (also known as operands) can be any data registers in the register file. The multiplier can accumulate fixed-point results in the local Multiplier Result (`MRF`) registers or write results back to the register file. The results in `MRF` can also be rounded or saturated in separate operations. Floating-point multiplies yield floating-point results, which the multiplier always writes directly to the register file.

The multiplier transfers input operands during the first half of the cycle and transfers results during the second half of the cycle. With this arrangement, the multiplier can read and write the same register file location in a single cycle.

For fixed-point multiplies, the multiplier reads the inputs from the upper 32 bits of the data registers. Fixed-point operands may be either both in integer format or both in fractional format. The format of the result matches the format of the inputs. Each fixed-point operand may be either an unsigned or a twos-complement number. If both inputs are fractional and signed, the multiplier automatically shifts the result left one bit to remove the redundant sign bit. The register name(s) within the multiplier instruction specify input data type(s)—Fx for floating-point and Rx for fixed-point.

Multiplier (Fixed-Point) Result Register

Fixed-point operations place 80-bit results in the multiplier's foreground `MRF` register or background `MRB` register, depending on which is active. For more information on selecting the result register, see "Alternate (Secondary) Data Registers" on page 2-32.

The location of a result in the `MRF` register's 80-bit field depends on whether the result is in fractional or integer format, as shown in Figure 2-2. If the result is sent directly to a data register, the 32-bit result with the same format as the input data is transferred, using bits 63-32 for a fractional result or bits 31-0 for an integer result. The eight LSBs of the 40-bit register file location are zero-filled.

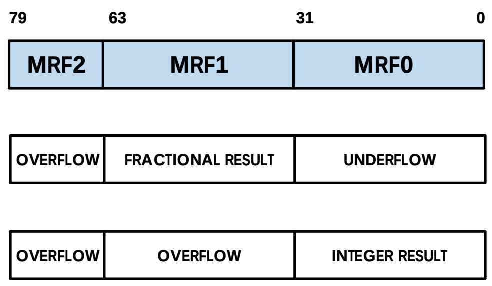


Figure 2-2. Multiplier Fixed-Point Result Placement

Fractional results can be rounded-to-nearest before being sent to the register file. If rounding is not specified, discarding bits 31-0 effectively truncates a fractional result (rounds to zero). For more information on rounding, see "Rounding Mode" on page 2-7.

The `MRF` register is divided into `MRF2`, `MRF1`, and `MRF0` registers, which can be individually read from or written to the register file. Each of these registers has the same format. When data is read from `MRF2`, it is sign-extended to 32 bits as shown in Figure 2-3. The processor zero fills the eight LSBs of the 40-bit register file location when data is read from `MRF2`, `MRF1`, or `MRF0` to the register file. When the processor writes data into `MRF2`, `MRF1`, or `MRF0` from the 32 MSBs of a register file location, the eight LSBs are ignored. Data written to `MRF1` is sign-extended to `MRF2`, repeating the MSB of `MRF1` in the 16 bits of `MRF2`. Data written to `MRF0` is not sign-extended.

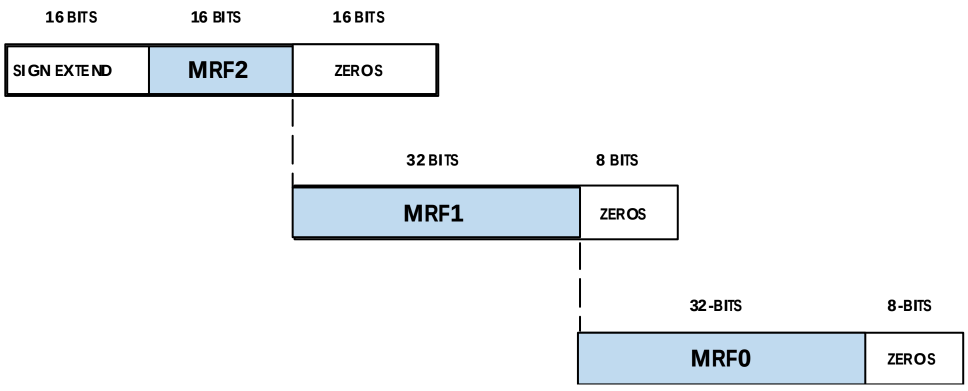


Figure 2-3. MR Transfer Formats

In addition to multiplication, fixed-point operations include accumulation, rounding and saturation of fixed-point data. There are three `MRF` register operations: Clear, Round, and Saturate.

The clear operation—`MRF`=0—resets the specified `MRF` register to zero. Often, it is best to perform this operation at the start of a multiply/accumulate operation to remove results left over from the previous operation.

The rounding operation—`MRF`=Rnd `MRF`—applies only to fractional results, so integer results are not effected. This operation rounds the 80-bit `MRF` value to nearest at bit 32; for example, the `MRF1`-`MRF0` boundary. Rounding of a fixed-point result occurs either as part of a multiply or multiply/accumulate operation or as an explicit operation on the `MRF` register. The rounded result in `MRF1` can be sent either to the register file or back to the same `MRF` register. To round a fractional result to zero (truncation) instead of to nearest, a program would transfer the unrounded result from `MRF1`, discarding the lower 32 bits in `MRF0`.

The saturate operation—`MRF`=Sat `MRF`—sets `MRF` to a maximum value if the `MRF` value has overflowed. Overflow occurs when the `MRF` value is greater than the maximum value for the data format—unsigned or twos-complement and integer or fractional—as specified in the saturate instruction.

The six possible maximum values appear in Table 2-3. The result from `MRF` saturation can be sent either to the register file or back to the same `MRF` register.

Table 2-3. Fixed-Point Format Maximum Values (For Saturation)

Maximum Number	**(Hexadecimal)**		
	MRF2	**MRF1**	**MRF0**
2’s complement fractional (positive)	0000	7FFF FFFF	FFFF FFFF
2’s complement fractional (negative)	FFFF	8000 0000	0000 0000
2’s complement integer (positive)	0000	0000 0000	7FFF FFFF
2’s complement integer (negative)	FFFF	FFFF FFFF	8000 0000
Unsigned fractional number	0000	FFFF FFFF	FFFF FFFF
Unsigned integer number	0000	0000 0000	FFFF FFFF

Multiplier Status Flags

Multiplier operations update four status flags in the processing element’s arithmetic status register (`ASTATx` and `ASTATy`). Table A-5 on page A-19 lists all the bits in these registers. The following bits in `ASTATx` or `ASTATy` flag multiplier status (a 1 indicates the condition) for the most recent multiplier operation.

- **Multiplier result negative.** Bit 6 (`MN`)
- **Multiplier overflow.** Bit 7 (`MV`)
- **Multiplier underflow.** Bit 8 (`MU`)
- **Multiplier floating-point invalid operation.** Bit 9 (`MI`)

Multiplier operations also update four "sticky" status flags in the processing element's Sticky status (STKYx and STKYy) register. Table A-5 on page A-19 lists all the bits in these registers. The following bits in STKYx or STKYy flag multiplier status (a 1 indicates the condition). Once set, a sticky flag remains high until explicitly cleared:

- **Multiplier fixed-point overflow.** Bit 6 (MOS)
- **Multiplier floating-point overflow.** Bit 7 (MVS)
- **Multiplier underflow.** Bit 8 (MUS)
- **Multiplier floating-point invalid operation.** Bit 9 (MIS)

Flag update occurs at the end of the cycle in which the status is generated and is available on the next cycle. If a program writes the arithmetic status register or sticky register explicitly in the same cycle that the multiplier is performing an operation, the explicit write to ASTAT or STKY supersedes any flag update from the multiplier operation.

Multiplier Instruction Summary

Table 2-4 and Table 2-6 list the Multiplier instructions and how they relate to ASTATx,y and STKYx,y flags. For more information on assembly language syntax, see the *ADSP-21160 SHARC DSP Instruction Set Reference*. In these tables, note the meaning of the following symbols.

- **Rn, Rx, Ry** indicate any register file location; treated as fixed-point
- **Fn, Fx, Fy** indicate any register file location; treated as floating-point
- * indicates the flag may be set or cleared, depending on results of instruction
- ** indicates the flag may be set (but not cleared), depending on results of instruction

- – indicates no effect
- The **Input Mods** column indicates the types of optional modifiers that you can apply to the instructions inputs. For a list of modifiers, see Table 2-5.

Table 2-4. Fixed-Point Multiplier Instruction Summary

Instruction	Input Mods	ASTATx,y Flags				STKYx,y Flags			
Fixed-Point: For Input Mods, see Table 2-5		**MU**	**MN**	**MV**	**MI**	**MUS**	**MOS**	**MVS**	**MIS**
Rn = Rx * Ry	1	*	*	*	0	–	**	–	–
MRF = Rx * Ry	1	*	*	*	0	–	**	–	–
MRB = Rx * Ry	1	*	*	*	0	–	**	–	–
Rn = MRF + Rx * Ry	1	*	*	*	0	–	**	–	–
Rn = MRB + Rx * Ry	1	*	*	*	0	–	**	–	–
MRF = MRF + Rx * Ry	1	*	*	*	0	–	**	–	–
MRB = MRB + Rx * Ry	1	*	*	*	0	–	**	–	–
Rn = MRF – Rx * Ry	1	*	*	*	0	–	**	–	–
Rn = MRB – Rx * Ry	1	*	*	*	0	–	**	–	–
MRF = MRF – Rx * Ry	1	*	*	*	0	–	**	–	–
MRB = MRB – Rx * Ry	1	*	*	*	0	–	**	–	–
Rn = SAT MRF	2	*	*	*	0	–	**	–	–
Rn = SAT MRB	2	*	*	*	0	–	**	–	–
MRF = SAT MRF	2	*	*	*	0	–	**	–	–
MRB = SAT MRB	2	*	*	*	0	–	**	–	–
Rn = RND MRF	3	*	*	*	0	–	**	–	–
Rn = RND MRB	3	*	*	*	0	–	**	–	–
MRF = RND MRF	3	*	*	*	0	–	**	–	–
MRB = RND MRB	3	*	*	*	0	–	**	–	–
MRF = 0	–	–	–	–	–	–	–	–	–

Table 2-4. Fixed-Point Multiplier Instruction Summary (Cont'd)

Instruction	Input Mods	ASTATx,y Flags				STKYx,y Flags			
Fixed-Point: For Input Mods, see Table 2-5		MU	MN	MV	MI	MUS	MOS	MVS	MIS
MRB = 0	–	–	–	–	–	–	–	–	–
MRxF = Rn	–	–	–	–	–	–	–	–	–
MRxB = Rn	–	–	–	–	–	–	–	–	–
Rn = MRxF	–	–	–	–	–	–	–	–	–
Rn = MRxB	–	–	–	–	–	–	–	–	–

Table 2-5. Input Modifiers For Fixed-Point Multiplier Instruction

Input Mods from Table 2-4	Input Mods—Options For Fixed-point Multiplier Instructions
	Note the meaning of the following symbols in this table: SSigned input UUnsigned input IInteger input(s) FFractional input(s) FRFractional inputs, Rounded output Note that (SF) is the default format for 1-input operations, and (SSF) is the default format for 2-input operations
1	(SSF), (SSI), (SSFR), (SUF), (SUI), (SUFR), (USF), (USI), (USFR), (UUF), (UUI), or (UUFR)
2	(SF), (SI), (UF), or (UI)
3	(SF) or (UF)

Table 2-6. Floating-Point Multiplier Instruction Summary

Instruction	ASTATx,y Flags				STKYx,y Flags			
Floating-Point:	MU	MN	MV	MI	MUS	MOS	MVS	MIS
Fn = Fx * Fy	*	*	*	*	**	–	**	**

Barrel-Shifter (Shifter)

The shifter performs bit-wise operations on 32-bit fixed-point operands. Shifter operations include:

- Shifts and rotates from off-scale left to off-scale right
- Bit manipulation operations, including bit set, clear, toggle, and test
- Bit field manipulation operations, including extract and deposit
- Fixed-point/floating-point conversion operations, including exponent extract, number of leading 1s or 0s

Shifter Operation

The shifter takes from one to three inputs: X-input, Y-input, and Z-input. The inputs (also known as operands) can be any register in the register file. Within a shifter instruction, the inputs serve as follows.

- The X-input provides data that is operated on
- The Y-input specifies shift magnitudes, bit field lengths or bit positions
- The Z-input provides data that is operated on and updated

In the following example, Rx is the X-input, Ry is the Y-input, and Rn is the Z-input. The shifter returns one output (Rn) to the register file.

```
Rn = Rn OR LSHIFT Rx BY Ry;
```

As shown in Figure 2-4, the shifter fetches input operands from the upper 32 bits of a register file location (bits 39-8) or from an immediate value in the instruction. The shifter transfers operands during the first half of the cycle and transfers the result to the upper 32 bits of a register (with the eight LSBs zero-filled) during the second half of the cycle. With this arrangement, the shifter can read and write the same register file location in a single cycle.

The X-input and Z-input are always 32-bit fixed-point values. The Y-input is a 32-bit fixed-point value or an 8-bit field (shf8), positioned in the register file. These inputs appear in Figure 2-4.

Some shifter operations produce 8-bit or 6-bit results. As shown in Figure 2-5, the shifter places these results in either the shf8 field or the bit6 field and sign-extends the results to 32 bits. The shifter always returns a 32-bit result.

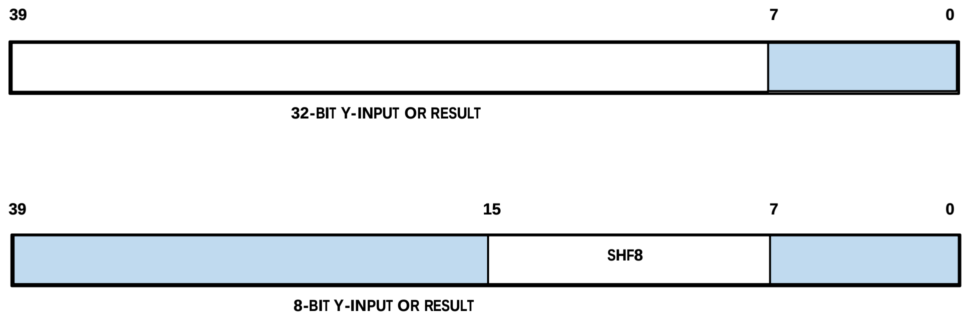


Figure 2-4. Register File Fields for Shifter Instructions

The shifter supports bit field deposit and bit field extract instructions for manipulating groups of bits within an input. The Y-input for bit field instructions specifies two 6-bit values: bit6 and len6, which are positioned in the Ry register as shown in Figure 2-5. The shifter interprets bit6 and len6 as positive integers. Bit6 is the starting bit position for the deposit or extract, and len6 is the bit field length, which specifies how many bits are deposited or extracted.

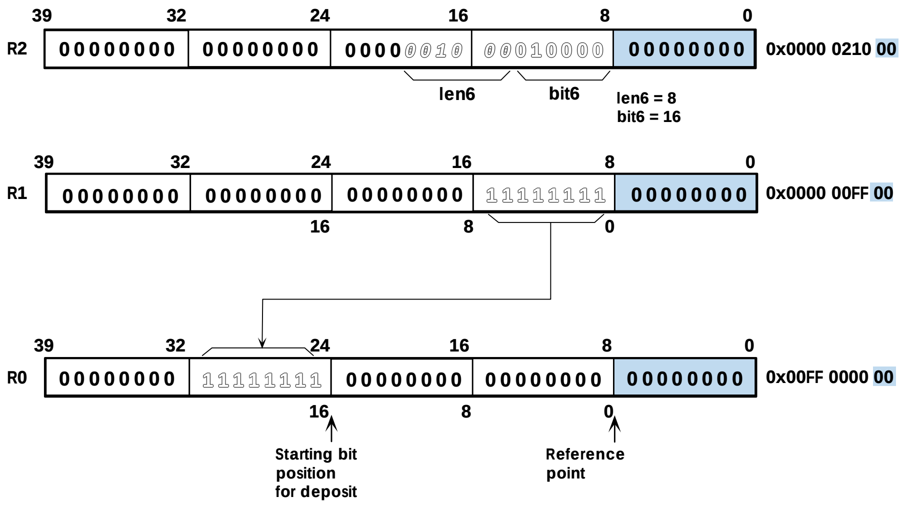


Figure 2-5. Register File Fields for FDEP, FEXT Instructions

Field deposit (Fdep) instructions take a group of bits from the input register (starting at the LSB of the 32-bit integer field) and deposit the bits as directed anywhere within the result register. The bit6 value specifies the starting bit position for the deposit. Figure 2-7 shows how the inputs, bit6 and len6, work in an field deposit instruction (`Rn=Fdep Rx By Ry`). Figure 2-8 shows bit placement for the field deposit instruction `R0 = FDEP R1 BY R2;`.

Field extract (Fext) instructions extract a group of bits as directed from anywhere within the input register and place them in the result register (aligned with the LSB of the 32-bit integer field). The bit6 value specifies the starting bit position for the extract. Figure 2-8 shows bit placement for the following field extract instruction `R3 = FEXT R4 BY R5;`

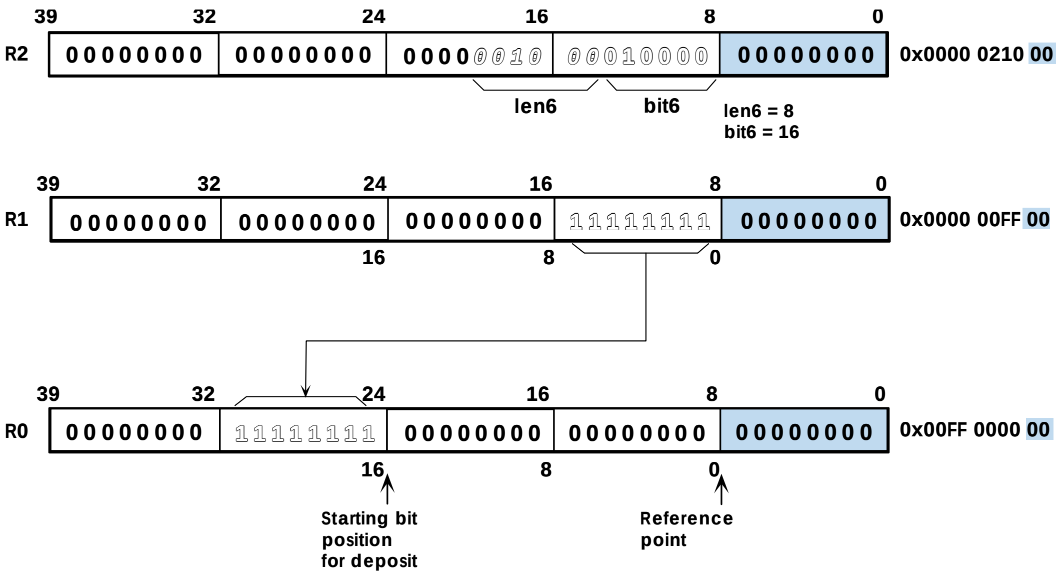
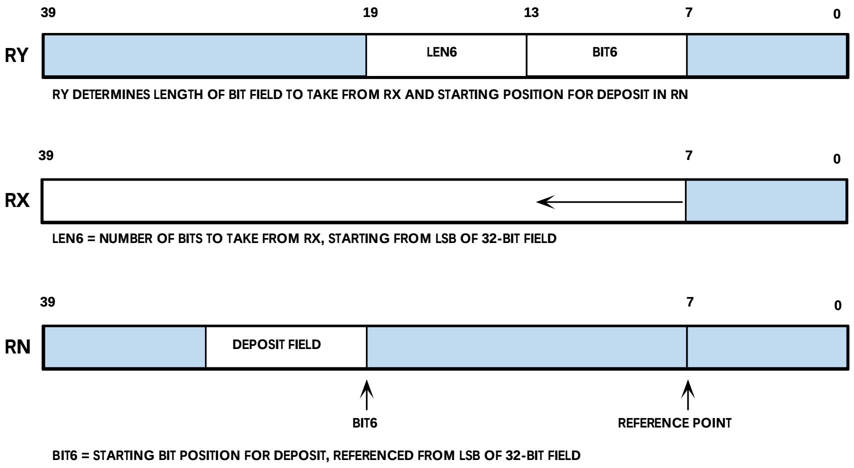


Figure 2-6. Bit Field Deposit Example



39 19 13 7 0
RY
LEN6
BIT6
RY DETERMINES LENGTH OF BIT FIELD TO TAKE FROM RX AND STARTING POSITION FOR DEPOSIT IN RN
RX
LEN6 = NUMBER OF BITS TO TAKE FROM RX, STARTING FROM LSB OF 32-BIT FIELD
RN
DEPOSIT FIELD
BIT6
REFERENCE POINT
BIT6 = STARTING BIT POSITION FOR DEPOSIT, REFERENCED FROM LSB OF 32-BIT FIELD

Figure 2-7. Bit Field Deposit Instruction

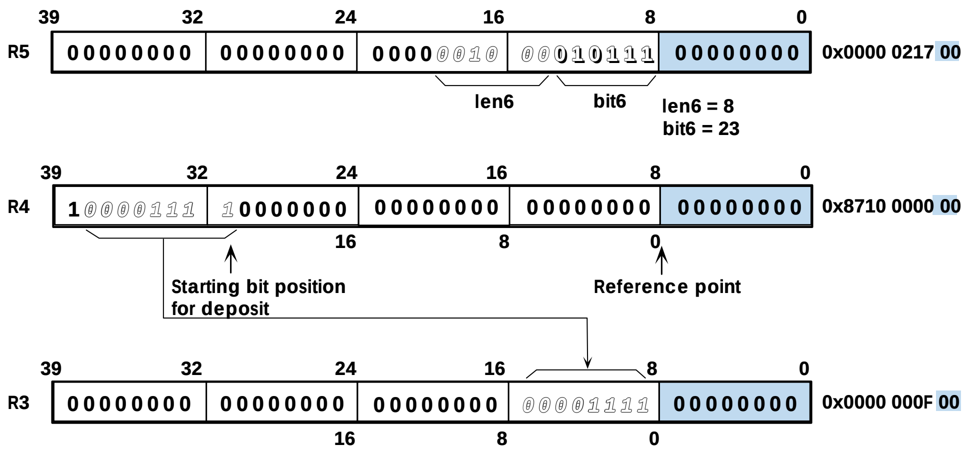


Figure 2-8. Bit Field Extract Example

Shifter Status Flags

Shifter operations update three status flags in the processing element's arithmetic status register (ASTATx and ASTATy). Table A-4 on page A-13 lists all the bits in these registers. The following bits in ASTATx or ASTATy indicate shifter status (a 1 indicates the condition) for the most recent ALU operation:

- **Shifter overflow of bits to left of MSB.** Bit 11 (SV)
- **Shifter result zero.** Bit 12 (SZ)
- **Shifter input sign for exponent extract only.** Bit 13 (SS)

Flag update occurs at the end of the cycle in which the status is generated and is available on the next cycle. If a program writes the arithmetic status register explicitly in the same cycle that the shifter is performing an operation, the explicit write to ASTAT supersedes any flag update caused by the shift operation.

Shifter Instruction Summary

Table 2-7 lists the Shifter instructions and how they relate to ASTATx,y flags. For more information on assembly language syntax, see the *ADSP-21160 SHARC DSP Instruction Set Reference*. In these tables, note the meaning of the following symbols:

- **Rn, Rx, Ry** indicate any register file location; bit fields used depend on instruction
- **Fn, Fx** indicate any register file location; floating-point word
- * indicates the flag may set or cleared, depending on data

Table 2-7. Shifter Instruction Summary

Instruction	**ASTATx,y Flags**		
	SZ	**SV**	**SS**
Rn = LSHIFT Rx BY Ry	*	*	0
Rn = LSHIFT Rx BY <data8>	*	*	0
Rn = Rn OR LSHIFT Rx BY Ry	*	*	0
Rn = Rn OR LSHIFT Rx BY <data8>	*	*	0
Rn = ASHIFT Rx BY Ry	*	*	0
Rn = ASHIFT Rx BY<data8>	*	*	0
Rn = Rn OR ASHIFT Rx BY Ry	*	*	0
Rn = Rn OR ASHIFT Rx BY <data8>	*	*	0
Rn = ROT Rx BY Ry	*	0	0
Rn = ROT Rx BY <data8>	*	0	0
Rn = BCLR Rx BY Ry	*	*	0
Rn = BCLR Rx BY <data8>	*	*	0
Rn = BSET Rx BY Ry	*	*	0
Rn = BSET Rx BY <data8>	*	*	0
Rn = BTGL Rx BY Ry	*	*	0

Table 2-7. Shifter Instruction Summary (Cont'd)

Instruction	ASTATx,y Flags		
	SZ	SV	SS
Rn = BTGL Rx BY <data8>	*	*	0
BTST Rx BY Ry	*	*	0
BTST Rx BY <data8>	*	*	0
Rn = FDEP Rx BY Ry	*	*	0
Rn = FDEP Rx BY <bit6>:<len6>	*	*	0
Rn = Rn OR FDEP Rx BY Ry	*	*	0
Rn = Rn OR FDEP Rx BY <bit6>:<len6>	*	*	0
Rn = FDEP Rx BY Ry (SE)	*	*	0
Rn = FDEP Rx BY <bit6>:<len6> (SE)	*	*	0
Rn = Rn OR FDEP Rx BY Ry (SE)	*	*	0
Rn = Rn OR FDEP Rx BY <bit6>:<len6> (SE)	*	*	0
Rn = FEXT Rx BY Ry	*	*	0
Rn = FEXT Rx BY <bit6>:<len6>	*	*	0
Rn = FEXT Rx BY Ry (SE)	*	*	0
Rn = FEXT Rx BY <bit6>:<len6> (SE)	*	*	0
Rn = EXP Rx (EX)	*	0	*
Rn = EXP Rx	*	0	*
Rn = LEFTZ Rx	*	*	0
Rn = LEFTO Rx	*	*	0
Rn = FPACK Fx	0	*	0
Fn = FUNPACK Rx	0	0	0

Data Register File

Each of the processor's processing elements has a data register file: a set of data registers that transfer data between the data buses and the computation units. These registers also provide local storage for operands and results.

The two register files each consist of 16 primary registers and 16 alternate (secondary) registers. All of the data registers are 40 bits wide. Within these registers, 32-bit data is always left-justified. If an operation specifies a 32-bit data transfer to these 40-bit registers, the eight LSBs are ignored on register reads, and the eight LSBs are cleared to zeros on writes.

Program memory data accesses and data memory accesses to/from the register file(s) occur on the PM data bus and DM data bus, respectively. One PM data bus access for each processing element and/or one DM data bus access for each processing element can occur in one cycle. Transfers between the register files and the DM or PM data buses can move up to 64-bits of valid data on each bus.

If an operation specifies the same register file location as both an input and output, the read occurs in the first half of the cycle and the write in the second half. With this arrangement, the processor uses the old data as the operand, before updating the location with the new result data. If writes to the same location take place in the same cycle, only the write with higher precedence actually occurs. The processor determines precedence for the write operation from the source of the data; from highest to lowest, the precedence is:

1. Data memory or universal register
2. Program memory
3. PEx ALU
4. PEy ALU

5. PEx Multiplier
6. PEy Multiplier
7. PEx Shifter
8. PEy Shifter

The data register file in Figure 2-1 on page 2-3 lists register names of R0 through R15 within PEx's register file. When a program refers to these registers as R0 through R15, the computational units treat the registers' contents as fixed-point data. To perform floating point computations, refer to these registers as F0 through F15. For example, the following instructions refer to the same registers, but direct the computational units to perform different operations:

```
F0=F1 * F2; /*floating-point multiply*/

R0=R1 * R2; /*fixed-point multiply*/
```

The F and R prefixes on register names do not effect the 32-bit or 40-bit data transfer; the naming convention only determines how the ALU, multiplier, and shifter treat the data.

ⓘ To maintain compatibility with code written for previous SHARC DSPs, the assembly syntax accommodates references to PEx data registers and PEy data registers.

Code may only refer to the PEy data registers (`S0` through `S15`) for data move instructions. The rules for using register names are as follows.

- `R0` through `R15` and `F0` through `F15` always refer to PEx registers for data move and computational instructions, whether the processor is in SISD or SIMD mode

- R0 through R15 and F0 through F15 refer to both PEx and PEy register for computational instructions in SIMD mode
- S0 through S15 always refer to PEy registers for data move instructions, whether the processor is in SISD or SIMD mode

For more information on SISD and SIMD computational operations, see "Alternate (Secondary) Data Registers" on page 2-32. For more information on ADSP-21161 processor assembly language, see the *ADSP-21160 SHARC DSP Instruction Set Reference*.

Alternate (Secondary) Data Registers

Each register file has an alternate register set. To facilitate fast context switching, the processor includes alternate register sets for data, results, and data address generator registers. Bits in the MODE1 register control when alternate registers become accessible. While inaccessible, the contents of alternate registers are not effected by processor operations. Note that there is a one cycle latency between writing to MODE1 and being able to access an alternate register set. The alternate register sets for data and results are described in this section. For more information on alternate data address generator registers, see the DAG "Alternate (Secondary) Data Registers" on page 2-32.

Bits in the MODE1 register can activate independent-alternate-data-register sets: the lower half (R0-R7 and S0-S7) and the upper half (R8-R15 and S8-S15). To share data between contexts, a program places the data to be shared in one half of either the current processing element's register file or the opposite processing element's register file and activates the alternate register set of the other half. For information on how to activate alternate data registers, see the description on page 2-33.

Each multiplier has a primary or foreground (MRF) register and alternate or background (MRB) results register. A bit in the MODE1 register selects which result register receives the result from the multiplier operation, swapping which register is the current MRF or MRB. This swapping facilitates context switching. Unlike other registers that have alternates, both MRF and MRB are accessible at the same time. All fixed-point multiplies can accumulate results in either MRF or MRB, without regard to the state of the MODE1 register. With this arrangement, code can use the result registers as primary and alternate accumulators, or code can use these registers as two parallel accumulators. This feature facilitates complex math.

The MODE1 register controls the access to alternate registers. Table A-2 on page A-3 lists all the bits in MODE1. The following bits in MODE1 control alternate registers (a 1 enables the alternate set).

- **Secondary registers for computation unit results.** Bit 2 (SRCU)
- **Secondary registers for hi register file, R8-R15 and S8-15.** Bit 7 (SRRFH)
- **Secondary registers for lo register file, R0-R7 and S0-S7.** Bit 10 (SRRFL)

The following example demonstrates how code should handle the one cycle of latency from the instruction setting the bit in MODE1 to when the alternate registers may be accessed. Note that it is possible to use any instruction that does not access the switching register file instead of an NOP instruction.

```
BIT SET MODE1 SRRFL;     /* activate alternate reg. file */
NOP;                     /* wait for access to alternates */
R0=7;
```

Multifunction Computations

Using the many parallel data paths within its computational units, the processor supports multiple-parallel (multifunction) computations. These instructions complete in a single cycle, and they combine parallel operation of the multiplier and the ALU or dual ALU functions. The multiple operations perform the same as if they were in corresponding single-function computations. Multifunction computations also handle flags in the same way as the single-function computations, except that in the dual add/subtract computation the ALU flags from the two operations are Or'ed together.

To work with the available data paths, the computation units constrain which data registers may hold the four input operands for multifunction computations. These constraints limit which registers may hold the X-input and Y-input for the ALU and multiplier.

Figure 2-9 shows a computational unit and indicates which registers may serve as X-inputs and Y-inputs for the ALU and multiplier. For example, the X-input to the ALU can only be R8, R9, R10 or R11. Note that the shifter is gray in Figure 2-7 to indicate that there are no shifter multifunction operations.

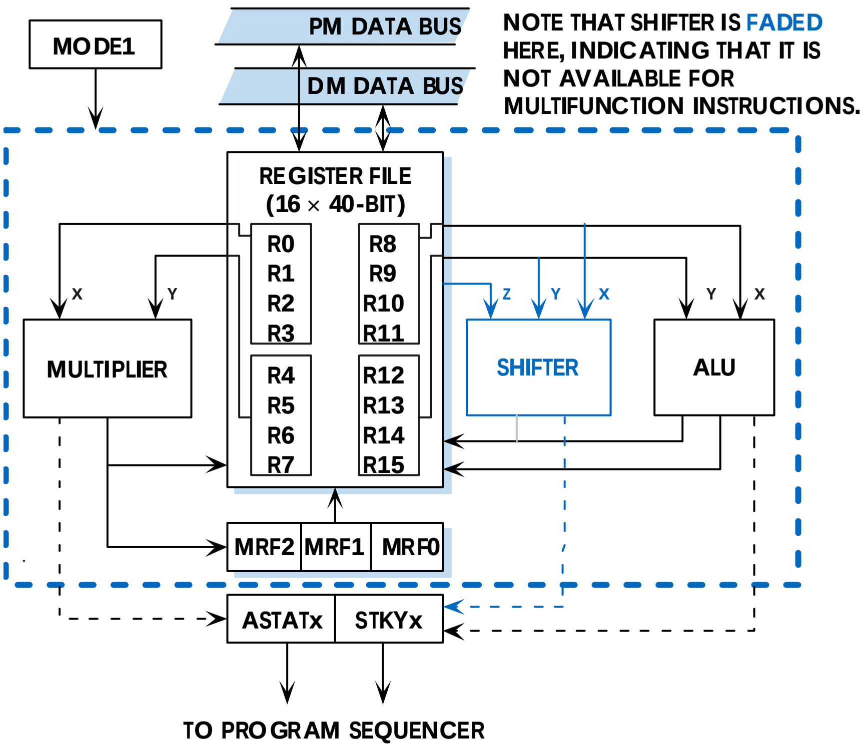


Figure 2-9. Input Registers for Multifunction Computations (ALU and Multiplier)

Table 2-8, through Table 2-11 list the multifunction computations. For more information on assembly language syntax, see the *ADSP-21160 SHARC DSP Instruction Set Reference*. In these tables, note the meaning of the following symbols.

- **Rm, Ra, Rs, Rx, Ry** indicate any register file location; fixed-point
- **Fm, Fa, Fs, Fx, Fy** indicate any register file location; floating-point
- **R3-0** indicates data file registers R3, R2, R1, or R0, and **F3-0** indicates data file registers F3, F2, F1, or F0

- **R7-4** indicates data file registers `R7`, `R6`, `R5`, or `R4`, and **F7-4** indicates data file registers `F7`, `F6`, `F5`, or `F4`
- **R11-8** indicates data file registers `R11`, `R10`, `R9`, or `R8`, and **F11-8** indicates data file registers `F11`, `F10`, `F9`, or `F8`
- **R15-12** indicates data file registers `R15`, `R14`, `R13`, or `R12`, and **F15-12** indicates data file registers `F15`, `F14`, `F13`, or `F12`
- **SSFR** indicates the X-input is signed, Y-input is signed, use Fractional inputs, and Rounded-to-nearest output
- **SSF** indicates the X-input is signed, Y-input is signed, use Fractional input

Table 2-8. Dual Add And Subtract

Ra = Rx + Ry, Rs = Rx – Ry
Fa = Fx + Fy, Fs = Fx – Fy

Table 2-9. Fixed-Point Multiply and Add, Subtract, Or Average

(Any combination of left and right column)		
Rm=R3-0 * R7-4 (SSFR),		Ra=R11-8 + R15-12
MRF=MRF + R3-0 * R7-4 (SSF),		Ra=R11-8 – R15-12
Rm=MRF + R3-0 * R7-4 (SSFR),		Ra=(R11-8 + R15-12)/2
MRF=MRF – R3-0 * R7-4 (SSF),		
Rm=MRF – R3-0 * R7-4 (SSFR),		

Table 2-10. Floating-Point Multiply And ALU Operation

Fm=F3-0 * F7-4, Fa=F11-8 + F15-12
Fm=F3-0 * F7-4, Fa=F11-8 – F15-12
Fm=F3-0 * F7-4, Fa=FLOAT R11-8 by R15-12
Fm=F3-0 * F7-4, Ra=FIX F11-8 by R15-12

Table 2-10. Floating-Point Multiply And ALU Operation (Cont'd)

Fm=F3-0 * F7-4, Fa=(F11-8 + F15-12)/2
Fm=F3-0 * F7-4, Fa=ABS F11-8
Fm=F3-0 * F7-4, Fa=MAX (F11-8, F15-12)
Fm=F3-0 * F7-4, Fa=MIN (F11-8, F15-12)

Table 2-11. Multiply With Dual Add and Subtract

Rm = R3-0 * R7-4 (SSFR), Ra = R11-8 + R15-12, Rs = R11-8 – R15-12
Fm = F3-0 * F7-4, Fa = F11-8 + F15-12, Fs = F11-8 – F15-12

Another type of multifunction operation is also available on the processor, combining transfers between the results and data registers and transfers between memory and data registers. Like other multifunction instructions, these parallel operations complete in a single cycle. For example, the processor can perform the following multiply and parallel read of data memory:

```
MRF=MRF-R5*R0, R6=DM(I1,M2);
```

Or, the processor can perform the following result register transfer and parallel read:

```
R5=MR1F, R6=DM(I1,M2);
```

Secondary Processing Element (PEy)

The ADSP-21161 processor contains two sets of computation units and associated register files. As shown in Figure 2-10, these two Processing Elements (PEx and PEy) support Single Instruction, Multiple Data (SIMD) operation.

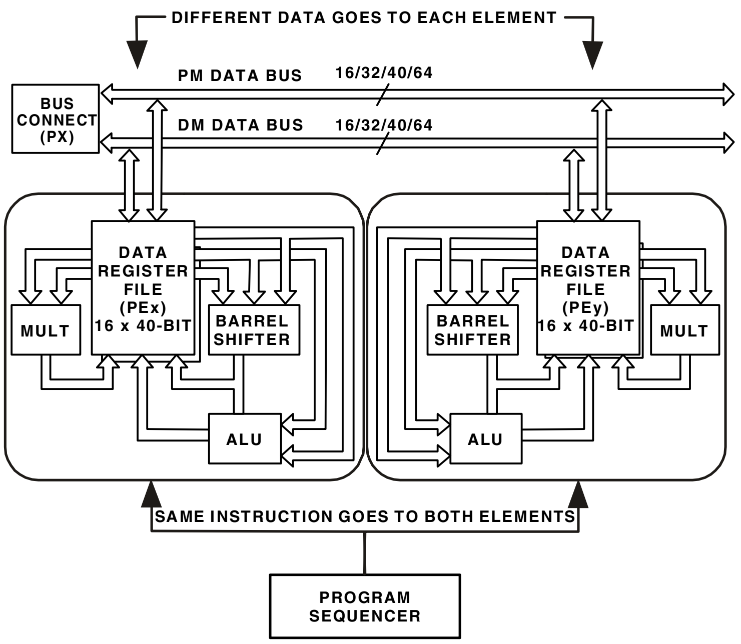

DIFFERENT DATA GOES TO EACH ELEMENT
PM DATA BUS
16/32/40/64
DM DATA BUS
16/32/40/64
BUS CONNECT (PX)
DATA REGISTER FILE (PEx) 16 x 40-BIT
MULT
BARREL SHIFTER
ALU
DATA REGISTER FILE (PEy) 16 x 40-BIT
BARREL SHIFTER
MULT
ALU
SAME INSTRUCTION GOES TO BOTH ELEMENTS
PROGRAM SEQUENCER

Figure 2-10. Block Diagram Showing Secondary Execution Complex

The MODE1 register controls the operating mode of the processing elements. Table A-2 on page A-3 lists all the bits in MODE1. The PEYEN bit (bit 21) in the MODE1 register enables or disables the PEy processing element. When PEYEN is cleared (0), the ADSP-21161 processor operates in Single-Instruction-Single-Data (SISD) mode, using only PEx; this is the mode in which ADSP-2106x family DSPs operate. When the PEYEN bit is set (1), the ADSP-21161 processor operates in SIMD mode, using the PEx and PEy processing elements. There is a one cycle delay after PEYEN is set or cleared, before the change to or from SIMD mode takes effect.

To support SIMD, the processor performs the following parallel operations.

- Dispatches a single instruction to both processing element's computation units
- Loads two sets of data from memory, one for each processing element
- Executes the same instruction simultaneously in both processing elements
- Stores data results from the dual executions to memory

Using the information here and in the *ADSP-21160 SHARC DSP Instruction Set Reference*, it is possible through SIMD mode's parallelism to double performance over similar algorithms running in SISD (ADSP-2106x processor compatible) mode.

The two processing elements are symmetrical, and each contains the following functional blocks.

- ALU
- Multiplier primary and alternate result registers
- Shifter
- Data register file and alternate register file

Dual Compute Units Sets

The computation units (ALU, Multiplier, and Shifter) in PEx and PEy are identical. The data bus connections for the dual computation units permit asymmetric data moves to, from, and between the two processing elements. Identical instructions execute on the PEx and PEy computational units; the difference is the data. The data registers for PEy operations are identified (implicitly) from the PEx registers in the instruction. This implicit relation between PEx and PEy data registers corresponds to complementary register pairs in Table 2-12. Any universal registers that don't appear in Table 2-12 have the same identities in both PEx and PEy. When a computation in SIMD mode refers to a register in the PEx column, the corresponding computation in PEy refers to the complimentary register in the PEy column.

Table 2-12. SIMD Mode Complementary Register Pairs

PEx	PEy
R0	S0
R1	S1
R2	S2
R3	S3
R4	S4
R5	S5
R6	S6
R7	S7
R8	S8
R9	S9
R10	S10
R11	S11
R12	S12
R13	S13
R14	S14
ASTATx	ASTATy
STKYx	STKYy

Table 2-13 lists the multiplier result SIMD mode complementary register pairs. These multiplier result registers are not universal (UREGs) registers and cannot be accessed directly. These registers can be read with the following multiplier operations:

```
MRxF/B = Rn;
Rn = MRxF/B;
```

Table 2-13. Multiplier Result SIMD Mode Complementary Register Pairs

PEx	PEy
MRF0	MSF0
MRF1	MSF1
MRF2	MSF2
MRB0	MSB0
MRB1	MSB1
MRB2	MSB2

Table 2-14. Other Complementary Register Pairs

USTAT1	USTAT2
USTAT3	USTAT4
PX1	PX2

Dual Register Files

The two 16 entry data register files (one in each PE) and their operand and result busing and porting are identical. The same is true for each 16 entry alternate register files. The transfer direction, source and destination registers, and data bus usage depend on the following conditions:

- **Computational mode:**
  – Is PEy enabled—`PEYEN` bit=1 in `MODE1` register
  – Is the data register file in PEx (`R0-R15`, `F0-F15`) or PEy (`S0-S15`)
  – Is the instruction a data register swap between the processing elements

- **Data addressing mode:**
  – What is the state of the Internal Memory Data Width (`IMDW`) bits in the System Configuration (`SYSCON`) register
  – Is Broadcast write enabled—`BDCST1,9` bits in `MODE1` register
  – What is the type of address—long, normal, or short word
  – Is Long Word override (`LW`) specified in the instruction
  – What are the states of instruction fields for DAG1 or DAG2

- **Program sequencing (conditional logic):**
  –What is the outcome of the instruction's condition comparison on each processing element

For information on SIMD issues that relate to computational modes, see "SIMD (Computational) Operations" on page 2-43. For information on SIMD issues relating to data addressing, see "SIMD Mode and Sequencing" on page 3-57. For information on SIMD issues relating to program sequencing, see "Addressing in SISD and SIMD Modes" on page 4-18.

Dual Alternate Registers

Both register files consist of a primary set of 16 by 40-bit registers and an alternate set of 16 by 40-bit registers. Context switching between the two sets of registers occur in parallel between the two processing elements. "Alternate (Secondary) Data Registers" on page 2-32.

SIMD (Computational) Operations

In SIMD mode, the dual processing elements execute the same instruction, but operate on different data. To support SIMD operation, the elements support a variety of dual data move features.

The processor supports unidirectional and bidirectional register-to-register transfers with the conditional compute and move instruction. All four combinations of inter-register file and intra-register file transfers (PEx ↔ PEx, PEx ↔ PEy, PEy ↔ PEx, and PEy ↔ PEy) are possible in both SISD (unidirectional) and SIMD (bidirectional) modes.

In SISD mode (PEYEN bit=0), the register-to-register transfers are unidirectional, meaning that an operation performed on one processing element is not duplicated on the other processing element. The SISD transfer uses a source register and a destination register, and either register can be in either element's data register file. For a summary of unidirectional transfers, see the upper half of Table 2-15. Note that in SISD mode a condition for an instruction only tests in the PEx element and applies to the entire instruction.

In SIMD mode (PEYEN bit=1), the register-to-register transfers are bidirectional, meaning that an operation performed on one element is duplicated in parallel on the other element. The instruction uses two source registers (one from each element's register file) and two destination registers (one from each element's register file). For a summary of bidirectional transfers, see the lower half of Table 2-15. Note that in SIMD mode a conditional for an instruction test in both the PEx and PEy elements, dividing control of the explicit and implicit transfers as detailed in Table 2-15.

Bidirectional register-to-register transfers in SIMD mode are allowed between a data register and DAG, control, or status registers. When the DAG, control, or status register is a source of the transfer, the destination can be a data register. This SIMD transfer duplicates the contents of the source register in a data register in both processing elements.

Careful programming is required when a DAG, control, or status register is a destination of a transfer from a data register. If the destination register has a complement (for example `ASTATx` and `ASTATy`), the SIMD transfer moves the contents of the explicit data register into the explicit destination and moves the contents of the implicit data register into the implicit destination (the complement). If the destination register has no complement (for example, I0), only the explicit transfer occurs.

Even if the code uses a conditional operation to select whether the transfer occurs, only the explicit transfer can take place if the destination register has no complement.

In the case where a DAG, control, or status register is both source and destination, the data move operation executes the same as if SIMD mode were disabled.

In both SISD and SIMD modes, the processor supports bidirectional register-to-register swaps. The swap always occurs between one register in each processing element's data register file.

Registers swaps use the special swap operator, <->. A register-to-register swap occurs when registers in different processing elements exchange values; for example `R0` <-> `S1`. Only single, 40-bit register to register swaps are supported—no double register operations.

When they are unconditional, register-to-register swaps operate the same in SISD mode and SIMD mode. If a condition is added to the instruction in SISD mode, the condition tests only in the PEx element and controls the entire operation. If a condition is added in SIMD mode, the condition tests in both the PEx and PEy elements and controls the halves of the operation as detailed in Table 2-15.

Table 2-15. Register-To-Register Move Summary (SISD Versus SIMD)

Mode	Instruction	Explicit Transfer	Implicit Transfer
SISD[1]	IF condition compute, Rx = Ry;	Rx loaded from Ry	None
	IF condition compute, Rx = Sy;	Rx loaded from Sy	None
	IF condition compute, Sx = Ry;	Sx loaded from Ry	None
	IF condition compute, Sx = Sy;	Sx loaded from Sy	None
	IF condition compute, Rx <-> Sy;	Rx swaps to Sy Sy swaps to Rx	None
SIMD[2]	IF condition compute, Rx = Ry;	Rx loaded from Ry	Sx loaded from Sy
	IF condition compute, Rx = Sy;	Rx loaded from Sy	Sx loaded from Ry
	IF condition compute, Sx = Ry;	Sx loaded from Ry	Rx loaded from Sy
	IF condition compute, Sx = Sy;	Sx loaded from Sy	Rx loaded from Ry
	IF condition compute, Rx <-> Sy;[3]	Rx swaps to Sy Sy swaps to Rx	None

1 In SISD mode, the conditional applies only to the entire operation and is only tested against PEx's flags. When the condition tests true, the entire operation occurs.

2 In SIMD mode, the conditional applies separately to the explicit and implicit transfers. Where the condition tests true (PEx for the explicit and PEy for the implicit), the operation occurs in that processing element.

3 Register to register transfers (R0=S0) and register swaps (R0<->S0) do not cause a PMD bus conflict. These operations use only the DMD bus and a hidden 16-bit bus to do the two register moves.

SIMD conditional instructions with the same destination registers do not produce predictable transfers. For example, the instruction `IF EQ R4 = R14 - R15, S4 = R6;` may not work as expected. This kind of usage is prohibited, as it is not logical to use it this way.

SIMD And Status Flags

When the processor is in SIMD mode (PEYEN bit=1), computations on both processing elements generate status flags, producing a logical Oring of the exception status test on each processing element. If one of the four fixed-point or floating-point exceptions is enabled, an exception condition on either or both processing elements generates an exception interrupt. Interrupt service routines must determine which of the processing elements encountered the exception. Note that returning from a floating point interrupt does not automatically clear the STKY state. Code must clear the STKY bits in both processing element's sticky status (STKYx and STKYy) registers as part of the exception service routine. For more information, see "Interrupts and Sequencing" on page 3-34.

3 PROGRAM SEQUENCER

The processor's program sequencer implements program flow which constantly provides the address of the next instruction to be executed by other parts of the processor. Program flow in the processor is mostly linear, with the processor executing program instructions sequentially. This linear flow varies occasionally when the program uses non-sequential program structures, such as those illustrated in Figure 3-1. Non-sequential structures direct the processor to execute an instruction that is not at the next sequential address following the current instruction. These structures include:

- **Loops.** One sequence of instructions executes several times with zero overhead.
- **Subroutines.** The processor temporarily interrupts sequential flow to execute instructions from another part of program memory.
- **Jumps.** Program flow transfers permanently to another part of program memory.
- **Interrupts.** Subroutines in which a runtime event (not an instruction) triggers the execution of the routine.
- **Idle.** An instruction that causes the processor to cease operations and hold its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

The sequencer manages execution of these program structures by selecting the address of the next instruction to execute. As part of its process, the sequencer handles the following tasks:

- Increments the fetch address
- Maintains stacks
- Evaluates conditions
- Decrements the loop counter
- Calculates new addresses
- Maintains an instruction cache
- Handles interrupts

To accomplish these tasks, the sequencer uses the blocks shown in Figure 3-2. The sequencer's address multiplexer selects the value of the next fetch address from several possible sources. The fetched address enters the instruction pipeline, made up of the fetch address register, decode address register, and program counter (`PC`). These contain the 24-bit addresses of the instructions currently being fetched, decoded, and executed. The `PC` couples with the `PC` stack, which stores return addresses and top-of-loop addresses. All addresses generated by the sequencer are 24-bit program memory instruction addresses.

To manage events, the sequencer's interrupt controller handles interrupt processing, determines whether an interrupt is masked, and generates the appropriate interrupt vector address.

With selective caching, the instruction cache lets the processor access data in program memory and fetch an instruction (from the cache) in the same cycle. The DAG2 data address generator outputs program memory data addresses.

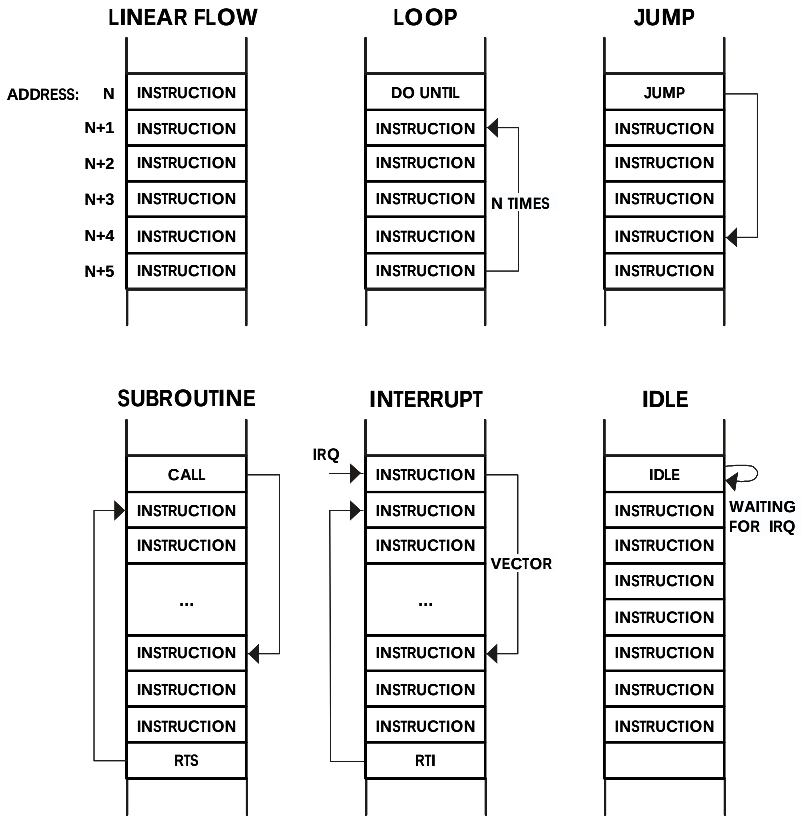


Figure 3-1. Program Flow Variations

The sequencer evaluates conditional instructions and loop termination conditions by using information from the status registers. The loop address stack and loop counter stack support nested loops. The status stack stores status registers for implementing nested interrupt routines.

Table 3-1 and Table 3-2 list the registers within and related to the program sequencer. All registers in the program sequencer are universal registers, so they are accessible to other universal registers and to data memory. All the sequencer's registers and the tops of stacks are readable,

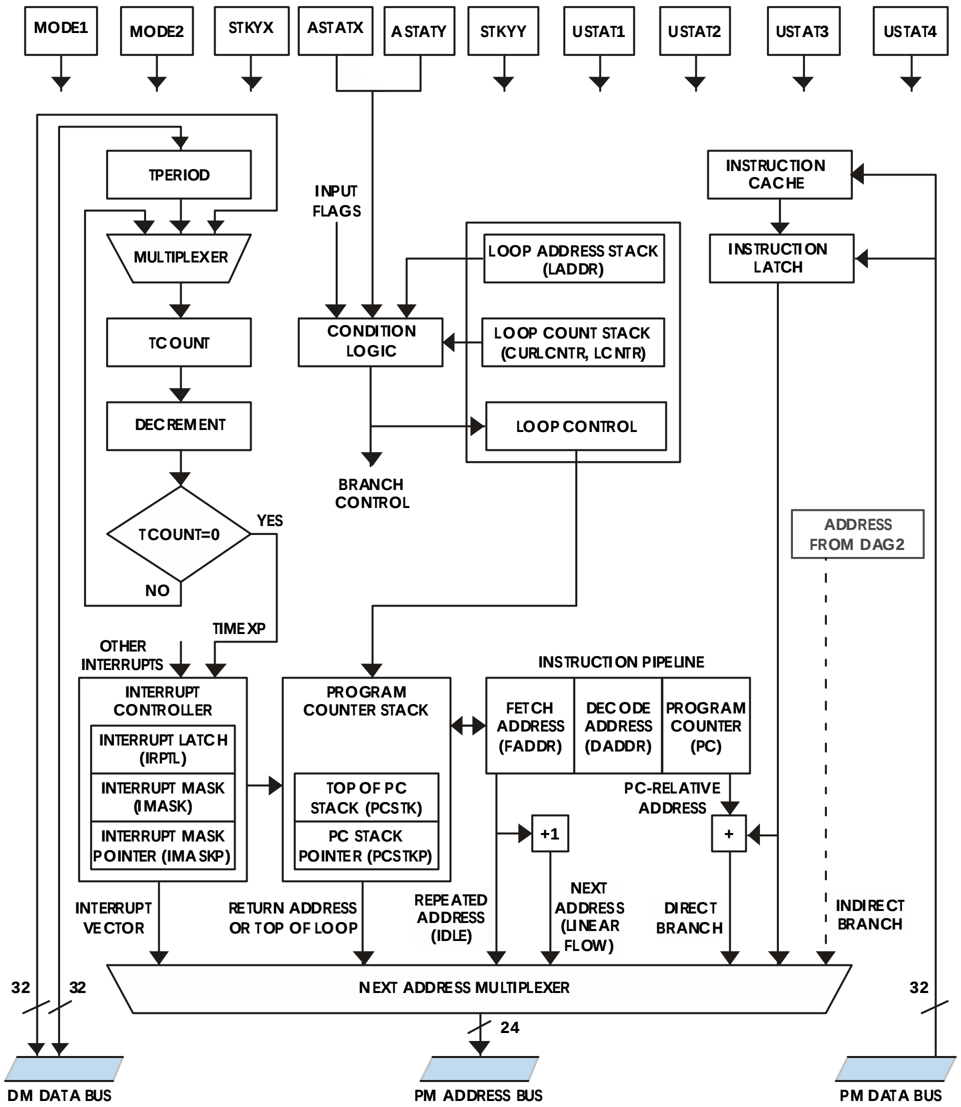


Figure 3-2. Sequencer Block Diagram

and all these registers are writable, except for the fetch address, decode address, and PC. Pushing or popping the PC stack is done with a write to the PC stack pointer, which is readable and writable. Pushing or popping the loop address stack requires explicit instructions.

A set of system control registers configures or provides input to the sequencer. These registers appear across the top and within the interrupt controller shown in Figure 3-2. A bit manipulation instruction permits setting, clearing, toggling, or testing specific bits in the system registers. For information on this instruction (Bit), see the *ADSP-21160 SHARC DSP Instruction Set Reference*. Writes to some of these registers do not take effect on the next cycle. For example, after a write to the `MODE1` register to enable ALU saturation mode, the change does not take effect until two cycles after the write. Also, some of these registers do not update on the cycle immediately following a write. An extra cycle is required before a read of the register returns the new value. With the lists of sequencer and system registers, Table 3-1 and Table 3-2 summarize the number of extra cycles (latency) for a write to take effect (effect latency) and for a new value to appear in the register (read latency). A "0" indicates that the write takes effect or appears in the register on the next cycle after the write instruction is executed, and a "1" indicates one extra cycle.

Table 3-1. Program Sequencer Registers Read and Effect Latencies

Register	Contents	Bits	Read Latency	Effect Latency
FADDR	fetch address	24	—	—
DADDR	decode address	24	—	—
PC	execute address	24	—	—
PCSTK	top of PC stack	24	0	0
PCSTKP	PC stack pointer	5	1	1
LADDER	top of loop address stack	32	0	0

Table 3-1. Program Sequencer Registers Read and Effect Latencies (Cont'd)

Register	Contents	Bits	Read Latency	Effect Latency
CURLCNTR	top of loop count stack (current loop count)	32	0	0
LCNTR	loop count for next DO UNTIL loop	32	0	0

Table 3-2. System Registers Read and Effect Latencies

Register	Contents	Bits	Read Latency	Maximum Effect Latency
MODE1	mode control bits	32	0	1
MODE2	mode control bits	32	0	1
IRPTL	interrupt latch	32	0	1
IMASK	interrupt mask	32	0	1
IMASKP	interrupt mask pointer (for nesting)	32	1	1
MMASK	mode mask	32	0	1
FLAGS	flag inputs	32	0	1
LIRPTL	link port interrupt latch/mask	32	0	1
ASTATX	arithmetic status flags	32	0	1
ASTATY	arithmetic status flags	32	0	1
STKYX	sticky status flags	32	0	1
STKYY	sticky status flags	32	0	1
USTAT1	user-defined status flags	32	0	0
USTAT2	user-defined status	32	0	0
USTAT3	user-defined status	32	0	0
USTAT4	user-defined status	32	0	0

The following sections in this chapter explain how to use each of the functional blocks in Figure 3-2:

- "Instruction Pipeline" on page 3-7
- "Instruction Cache" on page 3-8
- "Branches and Sequencing" on page 3-13
- "Loops and Sequencing" on page 3-22
- "Interrupts and Sequencing" on page 3-34
- "Timer and Sequencing" on page 3-50
- "Stacks and Sequencing" on page 3-52
- "Conditional Sequencing" on page 3-53
- "SIMD Mode and Sequencing" on page 3-57

Instruction Pipeline

The program sequencer determines the next instruction address by examining both the current instruction being executed and the current state of the processor. If no conditions require otherwise, the processor executes instructions from program memory in sequential order by incrementing the fetch address. Using its instruction pipeline, the processor processes instructions in three clock cycles:

- **Fetch cycle.** The processor reads the instruction from either the on-chip instruction cache or from program memory.
- **Decode cycle.** The processor decodes the instruction, generating conditions that control instruction execution.
- **Execute cycle.** The processor executes the instruction; the operations specified by the instruction complete in a single cycle.

These cycles overlap in the pipeline, as shown in Table 3-3. In sequential program flow, when one instruction is being fetched, the instruction fetched in the previous cycle is being decoded, and the instruction fetched two cycles before is being executed. Sequential program flow always has a throughput of one instruction per cycle.

Table 3-3. Pipelined Execution Cycles

Cycles	Fetch	Decode	Execute
1	0x08		
2	0x09	0x08	
3	0x0A	0x09	0x08
4	0x0B	0x0A	0x09
5	0x0C	0x0B	0x0A

Any non-sequential program flow can potentially decrease the processor's instruction throughput. Non-sequential program operations include:

- Program memory data accesses that conflict with instruction fetches
- Jumps
- Subroutine calls and returns
- Interrupts and return
- Loops

Instruction Cache

Usually, the sequencer fetches an instruction from memory on each cycle. Occasionally, bus constraints prevent some of the data and instructions from being fetched in a single cycle. To alleviate these data flow constraints, the processor has an instruction cache, which appears in Figure 3-2. When the processor executes an instruction that requires data access over the PM data bus, a bus conflict occurs because the sequencer uses the PM data bus for fetching instructions. To avoid these conflicts, the processor caches these instructions, reducing delays. Except for enabling or disabling the cache, its operation requires no user intervention. For more information, see "Using the Cache" on page 3-11.

When the processor first encounters a fetch conflict, the processor must wait to fetch the instruction on the following cycle, causing a delay. The processor automatically writes the fetched instruction to the cache to prevent the same delay from happening again. The sequencer checks the instruction cache on every program memory data access. If the instruction needed is in the cache, the instruction fetch from the cache happens in parallel with the program memory data access, without incurring a delay.

Because of the three-stage instruction pipeline, as the processor executes an instruction (at address n) that requires a program memory data access, this execution creates a conflict with the instruction fetch (at address n+2), assuming sequential execution. The cache stores the fetched instruction (n+2), not the instruction requiring the program memory data access.

If the instruction needed to avoid a conflict is in the cache, the cache provides the instruction while the program memory data access is performed. If the needed instruction is not in the cache, the instruction fetch from memory takes place in the cycle following the program memory data access, incurring one cycle of overhead. The fetched instruction is loaded into the cache, if the cache is enabled and not frozen, so that it is available the next time the same conflict occurs.

Figure 3-3 shows a block diagram of the instruction cache. The cache holds 32 instuction-address pairs. These pairs (or cache entries) are arranged into 16 (15-0) cache sets according to their address' 4 least significant bits (3-0). The two entries in each set (entry 0 and entry 1) have a valid bit, indicating whether the entry contains a valid instruction. The least recently used (LRU) bit for each set indicates which entry was not used last (0=entry 0 and 1=entry 1).

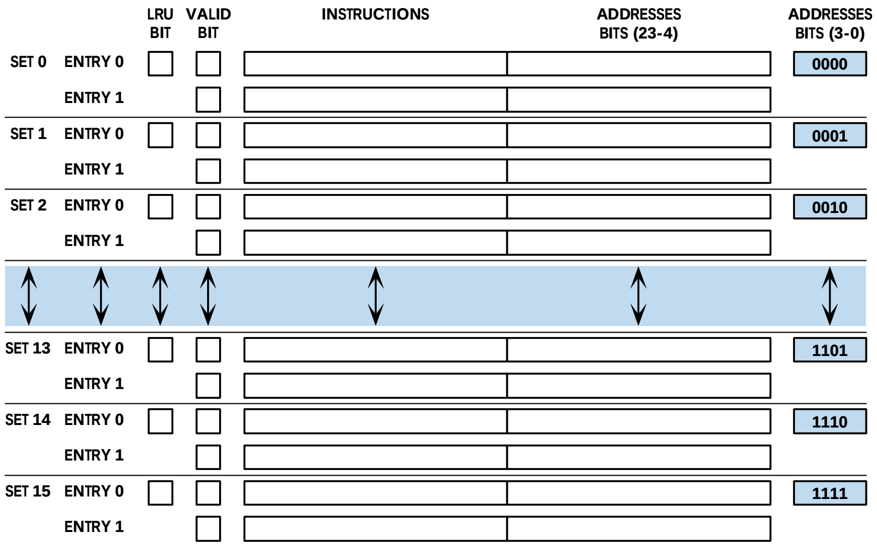


Figure 3-3. Instruction Cache Architecture

The cache places instructions in entries according to the 4 LSBs of the instruction's address. When the sequencer checks for an instruction to fetch from the cache, it uses the 4 address LSBs as an index to a cache set. Within that set, the sequencer checks the addresses of the two entries, looking for the needed instruction. If the cache contains the instruction, the sequencer uses the entry and updates the LRU bit (if necessary) to indicate the entry did not contain the needed instruction.

When the cache does not contain a needed instruction, the cache loads a new instruction and its address, placing these in the least recently used entry of the appropriate cache set and toggling the LRU bit (if necessary).

Using the Cache

After a processor reset, the cache starts cleared (containing no instructions), unfrozen, and enabled. From then on, the `MODE2` register controls the operating mode of the instruction cache. Table A-3 on page A-10 lists all the bits in `MODE2`. The following bits in `MODE2` control cache modes:

- **Cache Disable.** Bit 4 (`CADIS`) directs the sequencer to disable the cache (if 1) or enable the cache (if 0). Disabling the cache does not mark the current content of the cache as invalid. If the cache is enabled again, the existing content is used again. To clear the cache, use the `FLUSH CACHE` instruction.

- **Cache Freeze.** Bit 19 (`CAFRZ`) directs the sequencer to freeze the contents of the cache (if 1) or let new entries displace the entries in the cache (if 0).

If self-modifying code (for example, software loader kernel) or software overlays are used, execute a `FLUSH CACHE` instruction followed by a `NOP` before executing the new code. Otherwise, old content from the cache could still be used, although the code has changed.

When changing the cache's mode, note that an instruction containing a program memory data access must not be placed directly after a cache enable or cache disable instruction, because the processor must wait at least one cycle before executing the PM data access. A program should have an `NOP` inserted after the cache enable instruction.

Optimizing Cache Usage

Cache operation is usually efficient and requires no intervention. However, certain ordering of instructions can work against the cache's architecture and degrade cache efficiency. When the order of PM data accesses and instruction fetches continuously displaces cache entries and loads new entries, the cache is not operating efficiently. Rearranging the order of these instructions remedies this inefficiency.

An example of inefficient cache code appears in Table 3-4. The program memory data access at address 0x101 in the loop, Outer, causes the cache to load the instruction at 0x103 (into set 3). Each time the program calls the subroutine, Inner, the program memory data accesses at 0x201 and 0x211 displace the instruction at 0x103 by loading the instructions at 0x203 and 0x213 (also into set 3). If the program only calls the Inner subroutine rarely during the Outer loop execution, the repeated cache loads do not greatly influence performance. If the program frequently calls the subroutine while in the loop, the cache inefficiency has a noticeable effect on performance. To improve cache efficiency on this code (if for instance, execution of the Outer loop is time-critical), rearrange the order of some instructions. Moving the subroutine call up one location (starting at 0x201) would work here, because with that order the two cached instructions end up in cache set 4 instead of set 3.

Table 3-4. Cache-Inefficient Code

Address	Instruction
0x0100	lcntr=1024, do Outer until LCE;
0x0101	r0=dm(i0,m0), pm(i8,m8)=f3;
0x0102	r1=r0-r15;
0x0103	if eq call (Inner);
0x0104	f2=float r1;
0x0105	f3=f2*f2;
0x0106	Outer: f3=f3+f4;
0x0107	pm(i8,m8)=f3;
…	
0x0200	Inner: r1=R13;
0x0201	r14=pm(i9,m9);
…	
0x0211	pm(i9,m9)=r12;

Table 3-4. Cache-Inefficient Code (Cont'd)

Address	Instruction
...	
0x021F	rts;

Branches and Sequencing

One of type of non-sequential program flow that the sequencer supports is branching. A branch occurs when a `JUMP` or `CALL`/return instruction begins execution at a new location, other than the next sequential address. For descriptions on how to use `JUMP` and `CALL`/return instructions, see the *ADSP-21160 SHARC DSP Instruction Set Reference*. Briefly, these instructions operate as follows:

- A `JUMP` or a `CALL` instruction transfers program flow to another memory location. The difference between a `JUMP` and a `CALL` is that a `CALL` automatically pushes the return address (the next sequential address after the `CALL` instruction) onto the PC stack. This push makes the address available for the `CALL` instruction's matching return instruction, allowing easy return from the subroutine.

- A return instruction causes the sequencer to fetch the instruction at the return address, which is stored at the top of the PC stack. The two types of return instructions are return from subroutine (`RTS`) and return from interrupt (`RTI`). While the return from subroutine (`RTS`) only pops the return address off the PC stack, the return from interrupt (`RTI`) pops the return address and:

    1. Pops the status stack if the `ASTATx,y` and `MODE1` status registers have been pushed for any of the following interrupts: $\overline{\texttt{IRQ}}$`2-0`, timer, or `VIRPT`.

    2. Clears the interrupt's bit in the interrupt latch register (`IRPTL`) and the interrupt mask pointer (`IMASKP`).

There are a number of parameters that can be specified for branches:

- JUMP and CALL/return instructions can be conditional. The program sequencer can evaluate status conditions to decide whether to execute a branch. If no condition is specified, the branch is always taken. For more information on these conditions, see "Conditional Sequencing" on page 3-53.

- JUMP and CALL/return instructions can be immediate or delayed. Because of the instructions pipeline, an immediate branch incurs two lost (overhead) cycles. A delayed branch has no overhead. For more information, see "Delayed Branches" on page 3-15.

- JUMP instructions that appear within a loop or within an interrupt service routine have additional options. For information on the loop abort (LA) option, see "Loops and Sequencing" on page 3-22. For information on the loop re-entry (LR) option, see "Restrictions on Ending Loops" on page 3-25.For information on the clear interrupt (CI) option, see "Interrupts and Sequencing" on page 3-34.

The sequencer block diagram in Figure 3-2 on page 3-4 shows that branches can be direct or indirect. The difference is that the sequencer generates the address for a direct branch, and the PM data address generator (DAG2) produces the address for an indirect branch.

Direct branches are JUMP or CALL/return instructions that use an absolute—not changing at runtime—address (such as a program label) or use a PC-relative address. Some instruction examples that cause a direct branch are:

```
JUMP fft1024; /*Where fft1024 is an address label*/
CALL (pc,10); /*Where (pc,10) a PC-relative address*/
```

Indirect branches are JUMP or CALL/return instructions that use a dynamic address that comes from the PM data address generator. For more information on the data address generator, see Chapter 4, Data Address Generator. Some instruction examples that cause an indirect branch are:

```
JUMP (m8,i12); /*where (m8,i12) are DAG2 registers*/
CALL (m9,i13); /*where (m9,i13) are DAG2 registers*/
```

Conditional Branches

The sequencer supports conditional branches. These are JUMP or CALL/return instructions whose execution is based on testing an IF condition. For more information on condition types in IF condition instructions, see "Conditional Sequencing" on page 3-53. Note that the processor's Single-Instruction, Multiple-Data mode influences the execution of conditional branches. For more information, see "SIMD Mode and Sequencing" on page 3-57.

Delayed Branches

The instruction pipeline influences how the sequencer handles branches. For immediate branches in which JUMPs and CALL/return instructions are not specified as delayed branches (DB), two instruction cycles are lost (NOPs) as the pipeline empties and refills with instructions from the new branch.

As shown in Table 3-5 and Table 3-6, the processor does not execute the two instructions after the branch, which are in the fetch and decode stages. For a CALL, the decode address (the address of the instruction after the CALL) is the return address. During the two lost (no-operation) cycles, the pipeline fetches and decodes the first instruction at the branch address.

Table 3-5. Pipelined Execution Cycles for Immediate Branch (JUMP/Call)

Cycles	Fetch	Decode	Execute
1	n+2	n+1→nop[1]	n
2	j[2]	n+2→nop[3]	NOP
3	j+1	j	NOP
4	j+2	j+1	j
Note that n is the branching instruction, and j is the instruction branch address **1. n+1 suppressed** **2. For call, n+1 pushed on PC stack** **3. n+2 suppressed**			

Table 3-6. Pipelined Execution Cycles for Immediate Branch (Return)

Cycles	Fetch	Decode	Execute
1	n+2	n+1→nop[1]	n[2]
2	r	n+2→nop[3]	NOP
3	r+1	r	NOP
4	r+2	r+1	r
Note that n is the branching instruction, and r is the instruction branch address **1. n+1 suppressed** **2. r (n+1 in Table 3-5) popped from PC stack** **3. n+2 suppressed**			

For delayed branches, `JUMP`s and `CALL`/return instructions with the delayed branches `(DB)` modifier, no instruction cycles are lost in the pipeline, because the processor executes the two instructions after the branch while the pipeline fills with instructions from the new branch.

As shown in Table 3-7 and Table 3-8, the processor executes the two instructions after the branch, while the instruction at the branch address is fetched and decoded. In the case of a `CALL`, the return address is the third address after the branch instruction. While delayed branches use the instruction pipeline more efficiently than immediate branches, note that delayed branch code can be harder to understand because of the instructions between the branch instruction and the actual branch.

Table 3-7. Pipelined Execution Cycles for Delayed Branch (JUMP or CALL)

Cycles	Fetch	Decode	Execute
1	n+2	n+1	n
2	j[1]	n+2	n+1
3	j+1	j	n+2
4	j+2	j+1	j

Note that n is the branching instruction, and j is the instruction branch address
1. For call, n+3 pushed on PC stack

Table 3-8. Pipelined Execution Cycles For Delayed Branch (return)

Cycles	Fetch	Decode	Execute
1	n+2	n+1	n[1]
2	r	n+2	n+1
3	r+1	r	n+2
4	r+2	r+1	r

n is the branching instruction, and r is the instruction branch address
1. r (n+3 in Table 3-7) popped from PC stack

Besides being more challenging to code, delayed branches impose some limitations that stem from the instruction pipeline architecture. Because the delayed branch instruction and the two instructions that follow it must execute sequentially, the instructions in the two locations that follow a delayed branch instruction cannot be any of the following:

- Other branches (no `JUMP`, `CALL`, or return instructions)
- Any stack manipulations (no `PUSH` or `POP` instructions or writes to the PC stack or PC stack pointer)
- Any loops or other breaks in sequential operation (no `DO/UNTIL` or `IDLE` instructions)

ⓘ Development software for the processor should always flag these types of instructions as code errors in the two locations after a delayed branch instruction.

It is possible to follow a delayed branch instruction with a JUMP, CALL, or return instruction in one special case. If the sequential branch instructions use mutually exclusive conditions, one branch may following another. The following example is valid.

```
if gt jump (PC, 7) (db); // if greater than...
if le jump (PC,11) (db);  // if less than or equal...
```

Interrupt processing is also influenced by delayed branches and the instruction pipeline. Because the delayed branch instruction and the two instructions that follow it must execute sequentially, the processor does not immediately process an interrupt that occurs in between a delayed branch instruction and either of the two instructions that follow. Any interrupt that occurs during these instructions is latched, but is not processed until the branch is complete.

During a delayed branch, a program can read the PC stack or PC stack pointer immediately after a delayed call or return. This read shows that the return address on the PC stack has already been pushed or popped, even though the branch has not occurred yet.

Restrictions and Limitations When Using Delayed Branches

Besides being more challenging to code, delayed branches impose some limitations that stem from the instruction pipeline architecture. Because the delayed branch instruction and the two instructions that follow it must execute sequentially, the instructions in the two locations that follow a delayed branch instruction cannot be any of those described in the following five sections.

Development software for the ADSP-21161 processor should always flag the operations described in the next five sections as code errors in the two locations after a delayed branch instruction.

Normally it is not valid to use two conditional instructions using the (DB) option following each other. But the execution is allowed when these instructions are mutually exclusive as shown below.

```
If gt jump (PC, 7) (db);
If le jump (pc, 11) (db);
```

Other Jumps, or Calls with RTI, RTS

These instructions cannot be used when they follow a delayed branch instruction. This is shown in the following code that uses the `JUMP` instruction.

```
jump foo(db);
jump my(db);
r0=r0+r1;
r1=r1+r2;
```

In this case, the delayed branch instruction `r1=r1+r2`, is not executed. Further, the control jumps to `my` instead of `foo`, where the delayed branch instruction is the execution of `foo`.

The exception is for the `JUMP` instruction, which applies for the mutually exclusive conditions EQ (equal), and NE (not equal). If the first EQ condition evaluates true, then the NE conditional jump has no meaning and is the same as a `NOP` instruction. Code samples for these conditions are shown below.

```
if eq jump label1 (db);
if ne jump label1 (db);
nop;
nop;
```

Pushes or Pops of the PC Stack

In this case a push of the PC stack in a delayed branch is followed by a pop. If a value is pushed in the delayed branch of a call, it is first popped in the called subroutine. This is followed by an `RTS` instruction.

```
call foo (db);
push PCSTK;
nop;  /* second push due to PCSTK */
foo;  /* first push because of call */
```

This example shows that when a program pushes the `PCSTK` during a delayed slot, the PC stack pointer is pushed onto the `PCSTK`.

The following instructions are executed prior to executing the `RTS`.

```
pop PCSTK;
RTS (db);
nop;
nop;
```

If pushing the PC stack, a stack pop must be performed first. This is followed by an RTS instruction. If a value is popped inside a delayed branch, whatever subroutine return address is pushed is popped back, which is not allowed.

Writes to the PC Stack or PC Stack Pointer

The following two situations may arise when programs attempt to write to the PC stack inside a delayed branch.

1. If programs write into the PC stack inside a jump, one of the following situations can occur.

   a. The PC stack cannot hold a value that has already been pushed onto the PC stack.

      When the PC stack contains a value and a program writes that same value onto the stack, the original value is overwritten by the new value and the original value becomes corrupted.

   b. The PC stack is empty.

      Programs cannot write to the PC stack when they are inside a jump. In this case the PC stack remains empty.

2. Write to the PC stack inside a call.

   If a program writes to the PC stack inside of a call, the value that is pushed onto the PC stack because that call is overwritten by the value written onto the PC stack. Therefore, when a program performs an RTS, the program returns to the address pushed onto the PC stack and not to the address pushed while branching to the subroutine. For example:

```
call foo3 (db);
PCSTK=0x9011C;
nop;
```

The value 90114 is pushed onto the PC stack, while the value 9011C is written to the PC stack. Accordingly, the value 90114 is overwritten by the value 9011C in the PC stack because values that are pushed onto the stack have precedence over values written to the stack. Therefore, when the program comes back by executing an RTS, the return is to address 9011C and not to 90114.

IDLE Instruction

An interrupt is needed to come out of the IDLE instruction. If a program places an IDLE instruction inside the delayed branch the processor remains in the idled state because interrupts are latched but not serviced until the program exits a delayed branch.

Loops and Sequencing

Another type of non-sequential program flow that the sequencer supports is looping. A loop occurs when a DO/UNTIL instruction causes the processor to repeat a sequence of instructions until a condition tests true.

A special condition for terminating a loop is Loop Counter Expired (LCE). This condition tests whether the loop has completed the number of iterations in the LCNTR register. Loops that terminate with conditions other than LCE have some additional restrictions. For more information, see "Restrictions on Ending Loops" on page 3-25 and "Restrictions on Short Loops" on page 3-26. For more information on condition types in DO/UNTIL instructions, see "Conditional Sequencing" on page 3-53.

The processor's Single-Instruction, Multiple-Data mode influences the execution of loops. For more information, see "SIMD Mode and Sequencing" on page 3-57.

The DO/UNTIL instruction uses the sequencer's loop and condition features, which appear in Figure 3-2 on page 3-4. These features provide efficient software loops without the overhead of additional instructions to branch, test a condition, or decrement a counter. The following code example shows a DO/UNTIL loop that contains three instructions and iterates 30 times.

```
LCNTR=30, DO the_end UNTIL LCE; /*Loop iterates 30 times*/
R0=DM(I0,M0), F2=PM(I8,M8);
R1=R0-R15;
the_end: F4=F2+F3;             /*Last instruction in loop*/
```

When executing a DO/UNTIL instruction, the program sequencer pushes the address of the loop's last instruction and loop's termination condition onto the loop address stack. The sequencer also pushes the top-of-loop address—address of the instruction following the DO/UNTIL instruction—onto the PC stack.

The sequencer's instruction pipeline architecture influences loop termination. Because instructions are pipelined, the sequencer must test the termination condition (and, if the loop is counter-based, decrement the counter) before the end of the loop. Based on the test's outcome, the next fetch either exits the loop or returns to the top-of-loop.

The condition test occurs when the processor is executing the instruction two locations before the last instruction in the loop (at location e – 2, where e is the end-of-loop address). If the condition tests false, the sequencer repeats the loop, fetching the instruction from the top-of-loop address, which is stored on the top of the PC stack. If the condition tests true, the sequencer terminates the loop, fetching the next instruction after the end of the loop and popping the loop and PC stacks.

A special case of loop termination is the loop abort instruction, `JUMP (LA)`. This instruction causes an automatic loop abort when it occurs inside a loop. When the loop aborts, the sequencer pops the PC and loop address stacks once. If the aborted loop was nested, the single pop of the stacks leaves the correct values in place for the outer loop.

Table 3-9 and Table 3-10 show the pipeline states for loop iteration and termination.

Table 3-9. Pipelined Execution Cycles for Loop Back (Iteration)

Cycles	Fetch	Decode	Execute
1	e	e –1	e –2[1]
2	b[2]	e	e –1
3	b+1	b	e
4	b+2	b+1	b

Note that e is the loop end instruction, and b is the loop start instruction.
1. Termination condition tests false
2. Loop start address is top of PC stack

Table 3-10. Pipelined Execution Cycles for Loop Termination

Cycles	Fetch	Decode	Execute
1	e	e –1	e –2[1]
2	e+1[2]	e	e –1
3	e+2	e+1	e
4	e+3	e+2	e+1

Note that e is the loop end instruction.
1. Termination condition tests true
2. Loop aborts and loop stacks pop

Restrictions on Ending Loops

The sequencer's loop features (which optimize performance in many ways) limit the types of instructions that may appear at or near the end of the loop. These restrictions include:

- Nested loops cannot use the same end-of-loop instruction address.
- Nested loops with a non-counter-based loop as the outer loop must place the end address of the outer loop at least two addresses after the end address of the inner loop.
- Nested loops with a non-counter-based loop as the outer loop that use the loop abort instruction, `JUMP (LA)`, to abort the inner loop may not `JUMP (LA)` to the last instruction of the outer loop.
- An instruction that writes to the loop counter from memory cannot be used as the third-to-last instruction of a counter-based loop (at e–2, where e is the end-of-loop address).
- An `IF NOT LCE` instruction cannot be used as the instruction that follows a write to `CURLCNTR` from memory.
- Branch (`JUMP` or `CALL`/return) instructions may not be used as any of the last three instructions of a loop. This no end-of-loop branches rule also applies to single-instruction and two-instruction loops with only one iteration.

There is one exception to the no end-of-loop branches rule. The last three instructions of a loop may contain an immediate `CALL` —a `CALL` without a `DB` modifier—that is paired with a loop re-entry return—a return (`RTS`) with loop re-entry modifier (`LR`). The immediate `CALL` may be one of the last three instructions of a loop, but not in a one-instruction loop or a two-instruction, single-iteration loop.

Restrictions on Short Loops

The sequencer's pipeline features (which optimize performance in many ways) restrict how short loops iterate and terminate. Short loops (1- or 2-instruction loops) terminate in a special way because they are shorter than the instruction pipeline. Counter-based loops (`DO/UNTIL LCE`) of one or two instructions are not long enough for the sequencer to check the termination condition two instructions from the end of the loop. In these short loops, the sequencer has already looped back when the termination condition is tested. The sequencer provides special handling to prevent overhead (`NOP`) cycles if the loop is iterated a minimum number of times.

Table 3-11 and Table 3-12 show the pipeline execution for counter-based single-instruction loops. Table 3-13 and Table 3-14 show the pipeline execution for counter-based two-instruction loops. For no overhead, a loop of length one must be executed at least three times and a loop of length two must be executed at least twice. Loops of length one that iterate only once or twice and loops of length two that iterate only once incur two cycles of overhead, because two aborted instructions after the last iteration clear the instruction pipeline.

Table 3-11. Pipelined Execution Cycles for Single Instruction Counter-Based Loop With Three Iterations

Cycles	Fetch	Decode	Execute
1	n+2	n+1	n[1]
2	n+1[2]	n+1	n+1 (pass 1)
3	n+2[3]	n+1	n+1 (pass 2)
4	n+3	n+2	n+1 (pass 3)
5	n+4	n+3	n+2

Note: n is the loop start instruction, and n+2 is the instruction after the loop.
1. Loop count (LCNTR) equals 3
2. No opcode latch or fetch address update; count expired tests true
3. Loop iteration aborts; PC and loop stacks pop

Table 3-12. Pipelined Execution Cycles for Single Instruction Counter-Based Loop With Two Iterations (Two Overhead Cycles)

Cycles	Fetch	Decode	Execute
1	n+2	n+1	n[1]
2	n+1[2]	n+1	n+1 (pass 1)
3	n+1[3]	n+1→nop[4]	n+1 (pass 2)
4	n+2	n+1→nop[5]	NOP
5	n+3	n+2	NOP
6	n+4	n+3	n+2

Note: n is the loop start instruction, and n+2 is the instruction after the loop.
1. Loop count (LCNTR) equals 2
2. No opcode latch or fetch address update
3. Count expired tests true
4. Loop iteration aborts; PC and loop stacks pop; n+1 suppressed
5. n+1 suppressed

Table 3-13. Pipelined Execution Cycles for Two Instruction Counter-Based Loop With Two Iterations

Cycles	Fetch	Decode	Execute
1	n+2	n+1	n[1]
2	n+1[2]	n+2	n+1 (pass 1)
3	n+2[3]	n+1	n+2 (pass 1)
4	n+3[4]	n+2	n+1 (pass 2)
5	n+4	n+3	n+2 (pass 2)
6	n+5	n+4	n+3

Note: n is the loop start instruction, and n+3 is the instruction after the loop.
1. Loop count (LCNTR) equals 2
2. PC stack supplies loop start address
3. Count expired tests true
4. Loop iteration aborts; PC and loop stacks pop

Table 3-14. Pipelined Execution Cycles for Two Instruction Counter-Based Loop With One Iteration (Two Overhead Cycles)

Cycles	Fetch	Decode	Execute
1	n+2	n+1	n[1]
2	n+1[2]	n+2	n+1 (pass 1)
3	n+2[3]	n+1→nop[4]	n+2 (pass 1)
4	n+3	n+2→nop[5]	NOP
5	n+4	n+3	NOP
6	n+5	n+4	n+3

Note: n is the loop start instruction, and n+3 is the instruction after the loop.
1. Loop count (LCNTR) equals 1
2. PC stack supplies loop start address
3. Count expired tests true
4. Loop iteration aborts; PC and loop stacks pop; n+1 suppressed
5. n+2 suppressed

Processing of an interrupt that occurs during the last iteration of a one-instruction loop is delayed by one cycle in the following cases:

- the loop executes once or twice
- a two-instruction loop executes once
- a cycle follows one of these loops (which is an `NOP`)

Similarly, in a one-instruction loop that iterates at least three times, processing is delayed by one cycle if the interrupt occurs during the third-to-last iteration. For more information on pipeline execution during interrupts, see "Interrupts and Sequencing" on page 3-34.

Short non-counter-based loops terminate differently from short counter-based loops. These differences stem from the architecture of the pipeline and conditional logic:

- In a three-instruction non-counter-based loop, the sequencer tests the termination condition when the processor executes the top of loop instruction. When the condition tests true, the sequencer completes the iteration of the loop and terminates.
- In a two-instruction non-counter-based loop, the sequencer tests the termination condition when the processor executes the last (second) instruction. If the condition becomes true when the first instruction is executed, the condition tests true during the second instruction, and the sequencer completes one more iteration of the loop before exiting. If the condition becomes true during the second instruction, the sequencer completes two more iterations of the loop before exiting.
- In a one-instruction non-counter-based loop, the sequencer tests the termination condition every cycle. After the cycle when the condition becomes true, the sequencer completes three more iterations of the loop before exiting.

Loop Address Stack

The sequencer's loop support, which appears in Figure 3-2 on page 3-4, includes a loop address stack. The loop address stack is six levels deep by 32 bits wide.

The `LADDR` register contains the top entry on the loop address stack. This register is readable and writable over the DM Data bus. Reading and writing `LADDR` does not move the loop address stack pointer; only a stack push or pop performed with explicit instructions moves the stack pointer. `LADDR` contains the value 0xFFFF FFFF when the loop address stack is empty. Table A-14 on page A-45 lists all the bits in `LADDR`.

The sequencer pushes an entry onto the loop address stack when executing a DO/UNTIL or PUSH loop instruction. The stack entry pops off the stack two instructions before the end of its loop's last iteration or on a POP loop instruction. A stack overflow occurs if a seventh entry (one more than full) is pushed onto the loop stack. The stack is empty when no entries are occupied.

The loop stacks' overflow or empty status is available. Because the sequencer keeps the loop stack and loop counter stack synchronized, the same overflow and empty flags apply to both stacks. These flags are in the sticky status register (STKYx). For more information on STKYx, see Table A-5 on page A-19. For more information on how these flags work with the loop stacks, see "Loop Counter Stack" on page 3-30. Note that a loop stack overflow causes a maskable interrupt.

Because the sequencer tests the termination condition two instructions before the end of the loop, the loop stack pops before the end of the loop's final iteration. If a program reads LADDR at either of these instructions, the value is already the termination address for the next loop stack entry.

Loop Counter Stack

The sequencer's loop support, which appears in Figure 3-2 on page 3-4, includes a loop counter stack. The sequencer keeps the loop counter stack synchronized with the loop address stack. Both stacks always have the same number of locations occupied. Because these stacks are synchronized, the same empty and overflow status flags from the STKYx register apply to both stacks.

The loop counter stack is six locations deep. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is already full. Bits in the STKYx register indicate the loop counter stack full and empty states. Table A-5 on page A-19 lists the bits in the STYKx register. The STKYx bits that indicate loop counter stack status are:

- **Loop stacks overflowed.** Bit 25 (LSOV) indicates that the loop counter stack and loop stack are overflowed (if 1) or not overflowed (if 0)—A sticky bit.
- **Loop stacks empty.** Bit 26 (LSEM) indicates that the loop counter stack and loop stack are empty (if 1) or not empty (if 0)—Not sticky, cleared by a PUSH.

Within the sequencer, the current loop counter (CURLCNTR) and loop counter (LCNTR) registers allow access to the loop counter stack. CURLCNTR tracks iterations for a loop being executed, and LCNTR holds the count value before the loop is executed. The two counters let the processor maintain the count for an outer loop, while a program is setting up the count for an inner loop.

The top entry in the loop counter stack (CURLCNTR) always contains the current loop count. This register is readable and writable over the DM Data bus. Reading CURLCNTR when the loop counter stack is empty returns the value 0xFFFF FFFF.

The sequencer decrements the value of CURLCNTR for each loop iteration. Because the sequencer tests the termination condition two instruction cycles before the end of the loop, the loop counter also decrements before the end of the loop. If a program reads CURLCNTR at either of the last two loop instructions, the value is already the count for the next iteration.

The loop counter stack pops two instructions before the end of the last loop iteration. When the loop counter stack pops, the new top entry of the stack becomes the CURLCNTR value—the count in effect for the executing loop. If there is no executing loop, the value of CURLCNTR is 0xFFFF FFFF after the pop.

Writing CURLCNTR does not cause a stack push. If a program writes a new value to CURLCNTR, the program changes the count value of the loop currently executing. When no DO/UNTIL LCE loop is executing, writing to CURLCNTR has no effect. Because the processor must use CURLCNTR to perform counter-based loops, some restrictions apply to how a program can write CURLCNTR. For more information, see "Restrictions on Ending Loops" on page 3-25.

The next-to-top entry in the loop counter stack (LCNTR) is the location on the stack that takes effect on the next loop stack push. To set up a count value for a nested loop without changing the count for the currently executing loop, a program writes the count value to LCNTR.

ⓘ A value of zero in LCNTR causes a loop to execute 2^{32} times.

A DO/UNTIL LCE instruction pushes the value of LCNTR onto the loop count stack, making that value the new CURLCNTR value. Figure 3-4 demonstrates this process for a set of nested loops. The previous CURLCNTR value is preserved one location down in the stack. If a program reads LCNTR when the loop counter stack is full, the stack returns invalid data. When the loop counter stack is full, the stack discards any data written to LCNTR. If a program reads LCNTR during the last two instructions of a terminating loop, the value of LCNTR is the last CURLCNTR value for the loop.

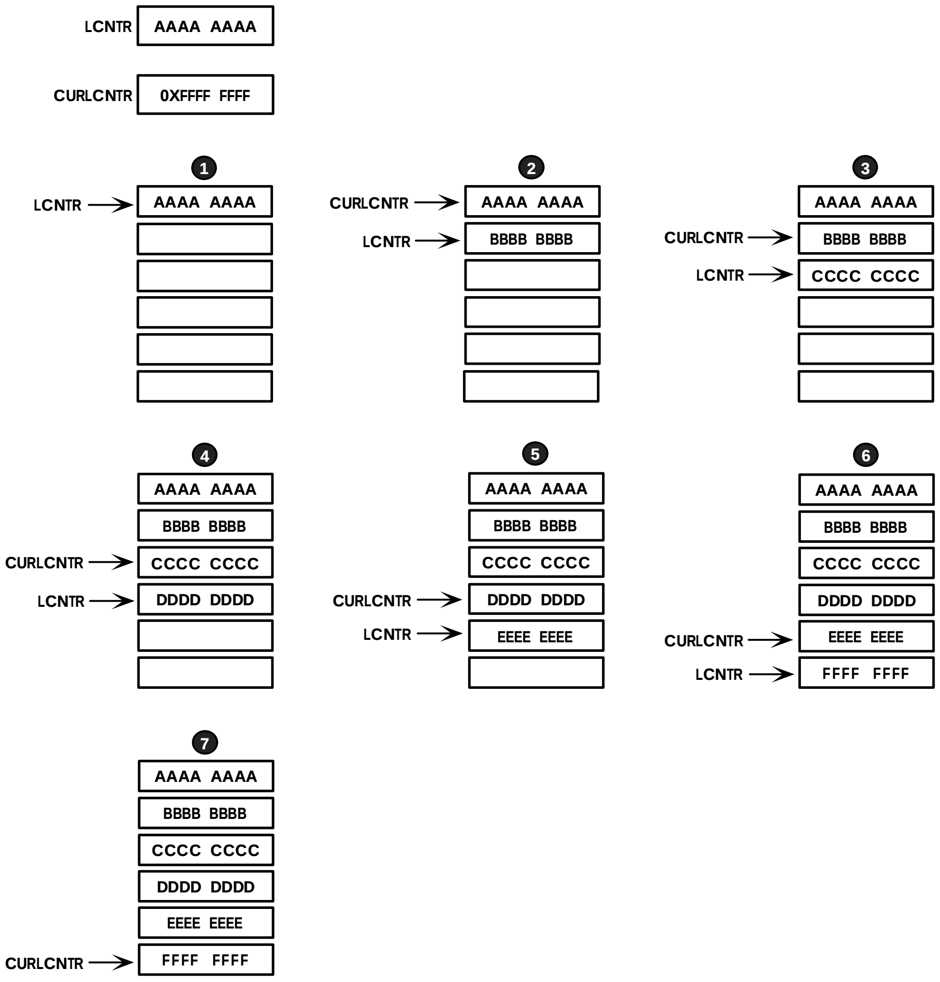


Figure 3-4. Pushing the Loop Counter Stack for Nested Loops

Interrupts and Sequencing

Another type of non-sequential program flow that the sequencer supports is interrupt processing. Interrupts may stem from a variety of conditions, both internal and external to the processor. In response to an interrupt, the sequencer processes a subroutine call to a predefined address, the interrupt vector. The processor assigns a unique vector to each type of interrupt.

The processor supports three prioritized, individually-maskable external interrupts, each of which can be either level- or edge-sensitive. External interrupts occur when another device asserts one of the processor's interrupt inputs ($\overline{\text{IRQ}}$2-0). The processor also supports internal interrupts. An internal interrupt can stem from arithmetic exceptions, stack overflows, or circular data buffer overflows. Several factors control the processor's response to an interrupt request. The processor responds to an interrupt request if:

- The processor is executing instructions or is in an idle state
- The interrupt is not masked
- Interrupts are globally enabled
- A higher priority request is not pending

When the processor responds to an interrupt, the sequencer branches program execution with a call to the corresponding interrupt vector address. Within the processor's program memory, the interrupt vectors are grouped in an area called the interrupt vector table. The interrupt vectors in this table are spaced at 4-instruction intervals. For a list of interrupt vector addresses and their associated latch and mask bits, see Table B-1 on page B-1. Each interrupt vector has associated latch and mask bits. Table A-9 on page A-27 lists the latch and mask bits.

To process an interrupt, the processor's program sequencer does the following:

1. Outputs the appropriate interrupt vector address
2. Pushes the current PC value (the return address) onto the PC stack
3. Pushes the current value of the ASTATx,y and MODE1 registers onto the status stack (if the interrupt is $\overline{\text{IRQ}}$2-0, timer, or VIRPT)
4. Sets the appropriate bit in the interrupt latch register (IRPTL)
5. Alters the interrupt mask pointer (IMASKP) to reflect the current interrupt nesting state, depending on the nesting mode

At the end of the interrupt service routine, the sequencer processes the return from interrupt (RTI) instruction and does following:

1. Returns to the address stored at the top of the PC stack
2. Pops this value off of the PC stack
3. Pops the status stack (if the ASTATx,y and MODE1 status registers were pushed for the $\overline{\text{IRQ}}$2-0, timer, or VIRPT interrupt)
4. Clears the appropriate bit in the interrupt latch register (IRPTL) and interrupt mask pointer (IMASKP)

Except for reset, all interrupt service routines should end with a return-from-interrupt (RTI) instruction. After reset, the PC stack is empty, so there is no return address. The last instruction of the reset service routine should be a jump to the start of your program.

If software writes to a bit in IRPTL forcing an interrupt, the processor recognizes the interrupt in the following cycle, and two cycles of branching to the interrupt vector follow the recognition cycle.

The processor responds to interrupts in three stages: synchronization and latching (1 cycle), recognition (1 cycle), and branching to the interrupt vector (2 cycles). Table 3-15, Table 3-16, and Table 3-17 show the pipelined execution cycles for interrupt processing.

Table 3-15. Pipelined Execution Cycles for Interrupt During Single-Cycle Instruction

Cycles	Fetch	Decode	Execute
1	n+1	n	n–1[1]
2	n+2[2]	n+1→nop[3]	n
3	v[4]	n+2→nop[5]	nop
4	v+1	v	nop
5	v+2	v+1	v

Note that n is the single-cycle instruction, and v is the interrupt vector instruction
1. Interrupt occurs
2. Interrupt recognized
3. n+1 pushed on PC stack; n+1 suppressed
4. Interrupt vector output
5. n+2 suppressed

Table 3-16. Pipelined Execution Cycles for Interrupt During Instruction With Conflicting PM Data Access (Instruction Not Cached)

Cycles	Fetch	Decode	Execute
1	n+1	n	n–1[1]
2	—[2]	$n+1 \rightarrow nop$[3]	n
3	n+2[4]	$n+1 \rightarrow nop$[5]	nop
4	v[6]	$n+2 \rightarrow nop$[7]	nop
5	v+1	v	nop
6	v+2	v+1	v

Note that n is the conflicting instruction, and v is the interrupt vector instruction
1. Interrupt occurs
2. Interrupt recognized, but not processed; PM data access
3. n+1 suppressed
4. Interrupt processed
5. n+1 suppressed
6. Interrupt vector output
7. n+1 pushed on PC stack; n+2 suppressed

Table 3-17. Pipelined Execution Cycles for Interrupt During Delayed Branch Instruction

Cycles	Fetch	Decode	Execute
1	n+1	n	n–1[1]
2	n+2[2]	n+1	n
3	j	n+2	n+1
4	j+1[3]	j→nop[4]	n+2
5	v[5]	j+1→nop[6]	nop
6	v+1	v	nop
7	v+2	v+1	v

Note that n is the delayed branch instruction, j is the instruction at the branch address, and v is the interrupt vector instruction
1. Interrupt occurs
2. Interrupt recognized, but not processed
3. Interrupt processed
4. For a Call, n+3 (return address) is pushed onto the PC stack; j suppressed
5. Interrupt vector output
6. j pushed on PC stack; j+1 suppressed

For most interrupts, internal and external, only one instruction is executed after the interrupt occurs (and before the two instructions aborted) while the processor fetches and decodes the first instruction of the service routine. Because of the one-cycle delay between an arithmetic exception and the `STKYx,y` register update, interrupt processing starts two cycles after an arithmetic exception occurs. Table 3-18 lists the latency associated with the $\overline{\text{IRQ}}$`2-0` interrupts and the multiprocessor vector interrupt.

Table 3-18. Minimum Latency of the $\overline{\text{IRQ}}$2-0 and VIRPT Interrupts

Interrupt	Minimum Latency
$\overline{\text{IRQ}}$2-0	3 cycles
VIRPT	6 cycles

If nesting is enabled and a higher priority interrupt occurs immediately after a lower priority interrupt, the service routine of the higher priority interrupt is delayed by one additional cycle. This delay allows the first instruction of the lower priority interrupt routine to be executed before it is interrupted. For more information, see "Nesting Interrupts" on page 3-45.

Certain processor operations that span more than one cycle hold off interrupt processing. If an interrupt occurs during one of these operations, the processor latches the interrupt, but delays its processing. The operations that have delayed interrupt processing are as follows:

- A branch (`JUMP` or `CALL`/return) instruction and the following cycle, whether it is an instruction (in a delayed branch) or an `NOP` (in a non-delayed branch)
- The first of the two cycles used to perform a program memory data access and an instruction fetch when the instruction is not cached
- The third-to-last iteration of a one-instruction loop
- The last iteration of either a one-instruction loop executed once or twice or a two-instruction loop executed once, and the following cycle (which is an `NOP`)
- The first of the two cycles used to fetch and decode the first instruction of an interrupt service routine
- Any waitstates for external memory accesses
- Any external memory access required when the processor does not have control of the external bus, during a host bus grant or when the processor is a bus slave in a multiprocessing system

Sensing Interrupts

The processor supports two types of interrupt sensitivity—the signal shape that triggers the interrupt. On interrupt pins ($\overline{\text{IRQ2-0}}$), either the input signal's edge or level can trigger an external interrupt.

The processor detects a level-sensitive interrupt if the signal input is low (active) when sampled on the rising edge of CLKIN. A level-sensitive interrupt must go high (inactive) before the processor returns from the interrupt service routine. If a level-sensitive interrupt is still active when the processor samples it after returning from its service routine, the processor treats the signal as a new request. The processor repeats the same interrupt routine without returning to the main program, assuming no higher priority interrupts are active.

The processor detects an edge-sensitive interrupt if the input signal is high (inactive) on one cycle and low (active) on the next cycle when sampled on the rising edge of CLKIN. An edge-sensitive interrupt signal can stay active indefinitely without triggering additional interrupts. To request another interrupt, the signal must go high, then low again.

Edge-sensitive interrupts require less external hardware compared to level-sensitive requests, because negating the request is unnecessary. An advantage of level-sensitive interrupts is that multiple interrupting devices may share a single level-sensitive request line on a wired-OR basis, allowing easy system expansion.

The MODE2 register controls external interrupt sensitivity. Table A-3 on page A-10 lists all bits in the MODE2 register. The following bits in MODE2 control interrupt sensitivity:

- **Interrupt 0 Sensitivity.** Bit 0 (IRQ0E), directs the processor to detect $\overline{\text{IRQ0}}$ as edge-sensitive (if 1) or level-sensitive (if 0).

- **Interrupt 1 Sensitivity.** Bit 1 (IRQ1E), directs the processor to detect $\overline{\text{IRQ1}}$ as edge-sensitive (if 1) or level-sensitive (if 0).
- **Interrupt 2 Sensitivity.** Bit 2 (IRQ2E), directs the processor to detect $\overline{\text{IRQ2}}$ as edge-sensitive (if 1) or level-sensitive (if 0).

The processor accepts external interrupts that are asynchronous to the processor's clock (CLKIN), allowing external interrupt signals to change at any time. An external interrupt must be held low at least one CLKIN cycle to guarantee that the processor samples the signal.

External interrupts must meet the setup and hold time requirements relative to the rising edge of CLKIN. For information on interrupt signal timing requirements, see the processor's Data Sheet.

Masking Interrupts

The sequencer supports interrupt masking—latching an interrupt, but not responding to it. Except for the $\overline{\text{RESET}}$ and $\overline{\text{EMU}}$ interrupts, all interrupts are maskable. If a masked interrupt is latched, the processor responds to the latched interrupt if it is later unmasked.

Interrupts can be masked globally or selectively. Bits in the MODE1, IMASK, and LIRPTL registers control interrupt masking. Table A-2 on page A-3 lists the bits in MODE1, Table A-9 on page A-27 lists the bits in IMASK, and Table A-10 on page A-34 lists the bits in LIRPTL. These bits control interrupt masking as follows:

- **Global interrupt enable.** MODE1, Bit 12 (IRPTEN) directs the processor to enable (if 1) or disable (if 0) all interrupts.
- **Selective interrupt enable.** IMASK, Bits 30-10 and 8-0, direct the processor to enable (if 1) or disable/mask (if 0) the corresponding interrupt.

- **Selective link port interrupt enable.** LIRPTL, Bits 17-16 (LPxMSK) direct the processor to enable (if 1) or disable/mask (if 0) the corresponding link port interrupt.
- **SPI port interrupt enable.** LIRPTL, Bit 18 (SPIRMSK) and Bit 19 (SPITMSK) direct the processor to enable (if 1) or disable/mask (if 0) the SPI port receive interrupt or transmit interrupt, respectively.

Except for the non-maskable interrupts and boot interrupts, all interrupts are masked at reset. For booting, the processor automatically unmasks and uses the external port (EP0I), link port (LP0I) or SPI port (SPIRI) interrupt after reset. Usage depends on whether the ADSP-21161 processor is booting from EPROM, host, SPI or link ports.

Latching Interrupts

When the processor recognizes an interrupt, the processor's interrupt latch (IRPTL and LIRPTL) registers latch the interrupts—set a bit to record that the interrupt occurred. The bits in these registers indicate all interrupts that are currently being serviced or are pending. Because these registers are readable and writable, any interrupt except reset can be set or cleared in software. Note that writing to the reset bit (bit 1) in IRPTL puts the processor into an illegal state.

When an interrupt occurs, the sequencer sets the corresponding bit in IRPTL or LIRPTL. During execution of the interrupt's service routine, the processor clears this bit during every cycle to prevent the same interrupt from being latched while its service routine is executing. After the return from interrupt (RTI), the sequencer stops clearing the latch bit.

If necessary, it is possible to re-use an interrupt while it is being serviced. For more information, see "Reusing Interrupts" on page 3-47.

The interrupt latch bits in IRPTL correspond to interrupt mask bits in the IMASK register. In both registers, the interrupt bits are arranged in order of priority. The interrupt priority is from 0 (highest) to 31 (lowest). Interrupt priority determines which interrupt is serviced first when more than one occurs in the same cycle. Priority also determines which interrupts are nested when the processor has interrupt nesting enabled. For more information, see "Nesting Interrupts" on page 3-45.

While IRPTL latches interrupts for a variety of events, the LIRPTL register contains latch and mask bits only for Link port and SPI DMA interrupts. A logical Or'ing of link port interrupts (masked-latch state) appears in the LPSUM bit in the IRPTL register. Because the LPSUM bit has a corresponding mask bit in the IMASK register, programs can use LPSUM for a second level of link port interrupt masking.

Multiple events can cause arithmetic interrupts—fixed-point overflow (FIXI) and floating-point overflow (FLTOI), underflow (FLTUI), and invalid operation (FLTII). To determine which event caused the interrupt, a program can read the arithmetic status flags in the STYKx or STKYy status registers. Table A-5 on page A-19 lists the bits in these registers. Service routines for arithmetic interrupts must clear the appropriate STKYx or STKYy bits to clear the interrupt. If the bits are not cleared, the interrupt is still active after the return from interrupt (RTI).

Status bits in STKYy apply only in SIMD mode. For more information, see "Secondary Processing Element (PEy)" on page 2-37.

One event can cause multiple interrupts. The timer decrementing to zero causes two timer expired interrupts, TMZHI (high priority) and TMZLI (low priority). This feature allows selection of the priority for the timer interrupt. Programs should unmask the timer interrupt with the desired priority and leave the other one masked. If both interrupts are unmasked, IRPTL latches both interrupts when the timer reaches zero, and the processor services the higher priority interrupt first, and then the lower priority interrupt.

The IRPTL also supports software interrupts. When a program sets the latch bit for one of these interrupts (SFT0I, SFT1I, SFT2I, or SFT3I), the sequencer services the interrupt, and the processor branches to the corresponding interrupt routine. Software interrupts have the same behavior as all other maskable interrupts.

Stacking Status During Interrupts

To run in an interrupt driven system, programs depend on the processor being restored to its pre-interrupt state after an interrupt is serviced. The sequencer's status stack eases the return from interrupt process by eliminating some interrupt service overhead—register saves and restores.

The status stack is fifteen locations deep. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is already full. Bits in the STKYx register indicate the status stack full and empty states. Table A-5 on page A-19 lists the bits in the STYKx register. The STKYx bits that indicate status stack status are:

- **Status stack overflow.** Bit 23 (SSOV) indicates that the status stack is overflowed (if 1) or not overflowed (if 0)—A sticky bit.
- **Status stack empty.** Bit 24, (SSEM) indicates that the status stack is empty (if 1) or not empty (if 0)—Not sticky, cleared by a PUSH.

For some interrupts ($\overline{\text{IRQ}}$2-0, timer expired, and VIRPT), the sequencer automatically pushes the ASTATx, ASTATy, and MODE1 registers onto the status stack. When the sequencer pushes an entry onto the status stack, the processor uses the MMASK register to clear the corresponding bits in the MODE1 register. All other bit settings remain the same. For more information and an example of how the MMASK and MODE1 registers work together, see the section "Mode Mask Register (MMASK)" on page A-8.

The sequencer automatically pops the ASTATx, ASTATY, and MODE1 registers from the status stack during the return from interrupt instruction (RTI). In one other case, JUMP (CI), the sequencer pops the stack. For more information, see "Reusing Interrupts" on page 3-47.

Only the $\overline{\text{IRQ}}$2-0, timer expired, and VIRPT interrupts cause the sequencer to push an entry onto the status stack. All other interrupts require either explicit saves and restores of effected registers or an explicit push or pop of the stack (PUSH/POP STS).

Pushing ASTATx, ASTATy, and MODE1 preserves the status and control bit settings. This feature allows a service routine to alter these bits with the knowledge that the original settings are automatically restored upon the return from the interrupt.

The top of the status stack contains the current values of ASTATx, ASTATy, and MODE1. Reading and writing these registers does not move the stack pointer. Explicit PUSH or POP instructions do move the status stack pointer.

Nesting Interrupts

The sequencer supports interrupt nesting—responding to another interrupt while a previous interrupt is being serviced. Bits in the MODE1, IMASKP, and LIRPTL registers control interrupt nesting. Table A-2 on page A-3 lists the bits in MODE1, Table A-9 on page A-27 lists the bits in IMASKP, and Table A-10 on page A-34 lists the bits in LIRPTL. These bits control interrupt nesting as follows:

- **Interrupt nesting enable.** MODE1 Bit 11 (NESTM). This bit directs the processor to enable (if 1) or disable (if 0) interrupt nesting.
- **Interrupt Mask Pointer.** IMASKP Bits 30- 15, 13-10 and 8-0. These bits list the interrupts in priority order and provide a temporary interrupt mask for each nesting level.

- **Link Port DMA Interrupt Mask Pointer.** LIRPTL Bits 25-24, (LPxMSKP). These bits are the link port DMA interrupts in priority order. They provide a temporary interrupt mask for each nesting level.
- **SPI Port DMA Interrupt Mask Pointer.** LIRPTL Bits 27-26, (SPITMSKP and SPIRMSKP). These bits are the SPI port transmit and receive DMA interrupts respectively. They provide a temporary interrupt mask.

When interrupt nesting is disabled, a higher priority interrupt can not interrupt a lower priority interrupt's service routine. Other interrupts are latched as they occur, but the processor processes them after the active routine finishes.

When interrupt nesting is enabled, a higher priority interrupt can interrupt a lower priority interrupt's service routine. Lower interrupts are latched as they occur, but the processor process them after the nested routines finish.

Programs should change the interrupt nesting enable (NESTM) bit only while outside of an interrupt service routine or during the reset service routine.

If nesting is enabled and a higher priority interrupt occurs immediately after a lower priority interrupt, the service routine of the higher priority interrupt is delayed by one cycle. This delay allows the first instruction of the lower priority interrupt routine to be executed, before it is interrupted.

When servicing nested interrupts, the processor uses the interrupt mask pointer (IMASKP) to create a temporary interrupt mask for each level of interrupt nesting; the IMASK value is not effected. The processor changes IMASKP each time a higher priority interrupt interrupts a lower priority service routine.

The bits in IMASKP correspond to the interrupts in order of priority. When an interrupt occurs, the processor sets its bit in IMASKP. If nesting is enabled, the processor uses IMASKP to generate a new temporary interrupt mask, masking all interrupts of equal or lower priority to the highest priority bit set in IMASKP and keeping higher priority interrupts the same as in IMASK. When a return from an interrupt service routine (RTI) is executed, the processor clears the highest priority bit set in IMASKP and generates a new temporary interrupt mask. The processor masks all interrupts of equal or lower priority to the highest priority bit set in IMASKP. The bit set in IMASKP that has the highest priority always corresponds to the priority of the interrupt being serviced.

If an interrupt recurs while its service routine is running and nesting is enabled, the processor updates IRPTL, but does not service the interrupt. The processor waits until the return from interrupt (RTI) completes before vectoring to the service routine again.

If nesting is not enabled, the processor masks out all interrupts and IMASKP is not used, but the processor still updates IMASKP to create a temporary interrupt mask.

The interrupt controller uses the IMASKP register and the LPxMSKP, SPITMSKP, and SPIRMSKP bits of the LIRPTL register. These bits should not be modified.

Reusing Interrupts

When an interrupt occurs the sequencer sets the corresponding bit in IRPTL. During execution of the service routine, the sequencer keeps this bit cleared—the processor clears the bit during every cycle, preventing the same interrupt from being latched while its service routine is already executing.

If necessary, it is possible to re-use an interrupt while it is being serviced. Using a JUMP clear interrupt, JUMP (CI), instruction in the interrupt service routine clears the interrupt, allowing its reuse while the service routing is executing.

The JUMP (CI) instruction reduces an interrupt service routine to a normal subroutine, clearing the appropriate bit in the interrupt latch and interrupt mask pointer and popping the status stack. After the JUMP (CI) instruction, the processor stops automatically clearing the interrupt's latch bit, allowing the interrupt to latch again.

When returning from a subroutine entered with a JUMP (CI) instruction, a program must use a return loop reentry, RTS(LR), instruction. For more information, see "Restrictions on Ending Loops" on page 3-25.

The following example shows an interrupt service routine that is reduced to a subroutine with the (CI) modifier:

```
instr1; /*Interrupt entry from main program*/
JUMP(PC,3) (DB,CI); /*Clear interrupt status*/
instr3;
instr4;
instr5;
RTS (LR); /*Use LR modifier with return from subroutine*/
```

The JUMP (PC,3)(DB,CI) instruction actually only continues linear execution flow by jumping to the location PC + 3 (instr5). The two intervening instructions (instr3, instr4) are executed because of the delayed branch (DB). This JUMP instruction is only an example—a JUMP (CI) can be to any location.

Interrupting IDLE

The sequencer supports placing the processor in IDLE—a special instruction that halts the processor core in a low-power state. The halt occurs until an external interrupt ($\overline{\text{IRQ}}$2-0), timer interrupt, DMA interrupt, or

VIRPT vector interrupt occurs. When executing an IDLE instruction, the sequencer fetches one more instruction at the current fetch address and then suspends operation. The processor's I/O processor is not effected by the IDLE instruction—DMA transfers to or from internal memory continues uninterrupted.

The processor's internal clock and timer (if enabled) continue to run during IDLE. When an external interrupt ($\overline{\text{IRQ}}$2-0), timer interrupt, DMA interrupt, or VIRPT vector interrupt occurs, the processor responds normally. After two cycles used to fetch and decode the first instruction of the interrupt service routine, the processor continues executing instructions normally.

Multiprocessing Interrupts

The sequencer supports a multiprocessor vector interrupt. The vector interrupt (VIRPT) permits passing interprocessor commands in multiple-processor systems. This interrupt occurs when an external processor (a host or another processor) writes an address to the VIRPT register, inserting a new vector address for VIRPT.

The VIRPT register has space for the vector address and data for the service routine. Table A-19 on page A-64 lists the bits in the VIRPT registers.

When servicing a VIRPT interrupt, the processor automatically pushes the status stack and executes the service routine located at the address specified in VIRPT. During the return from interrupt (RTI), the processor automatically pops the status stack.

To flag that a VIRPT interrupt is pending, the processor sets the VIPD bit in the SYSTAT register when the external processor writes to the VIRPT register. Programs passing interprocessor commands must monitor VIPD to check if the processor can receive a new VIRPT address, because:

- If an external processor writes VIRPT while a previous vector is pending, the new VIRPT address replaces the previous pending one.
- If an external processor writes VIRPT while a previous vector is executing, the new VIRPT address does not execute (no new interrupt is triggered).

When returning from a VIRPT interrupt, the processor clears the VIPD bit. Note that if a processor writes to its own VIRPT register, the write is ignored.

Timer and Sequencing

The sequencer includes a programmable interval timer, which appears in Figure 3-2 on page 3-4. Bits in the MODE2, TCOUNT, and TPERIOD registers control timer operations. Table A-3 on page A-10 lists the bits in the MODE2 register. The bits that control the timer are given as follows:

- **Timer enable.** MODE2 Bit 5 (TIMEN). This bit directs the processor to enable (if 1) or disable (if 0) the timer.
- **Timer count.** (TCOUNT) This register contains the decrementing timer count value, counting down the cycles between timer interrupts.
- **Timer period.** (TPERIOD) This register contains the timer period, indicating the number of cycles between timer interrupts.

The TCOUNT register contains the timer counter. The timer decrements the TCOUNT register during each clock cycle. When the TCOUNT value reaches zero, the timer generates an interrupt and asserts the TIMEXP output high for 4 cycles (when the timer is enabled), as shown in Figure 3-5. On the clock cycle after TCOUNT reaches zero, the timer automatically reloads TCOUNT from the TPERIOD register.

The TPERIOD value specifies the frequency of timer interrupts. The number of cycles between interrupts is TPERIOD + 1. The maximum value of TPERIOD is $2^{32} - 1$.

To start and stop the timer, programs use the MODE2 register's TIMEN bit. With the timer disabled (TIMEN=0), the program loads TCOUNT with an initial count value and loads TPERIOD with the number of cycles for the desired interval. Then, the program enables the timer (TIMEN=1) to begin the count.

When a program enables the timer, the timer starts decrementing the TCOUNT register at the end of the next clock cycle. If the timer is subsequently disabled, the timer stops decrementing TCOUNT after the next clock cycle as shown in Figure 3-5.

The timer expired event (TCOUNT decrements to zero) generates two interrupts, TMZHI and TMZLI. For information on latching and masking these interrupts to select timer expired priority, see "Latching Interrupts" on page 3-42.

As with other interrupts, the sequencer needs two cycles to fetch and decode the first instruction of the timer expired service routine before executing the routine. The pipeline execution for the timer interrupt appears in Table 3-15 on page 3-36.

Programs can read and write the TPERIOD and TCOUNT registers by using universal register transfers. Reading the registers does not effect the timer. Note that an explicit write to TCOUNT takes priority over the sequencer's loading TCOUNT from TPERIOD and the timer's decrementing of TCOUNT. Also note that TCOUNT and TPERIOD are not initialized at reset. Programs should initialize these registers before enabling the timer.

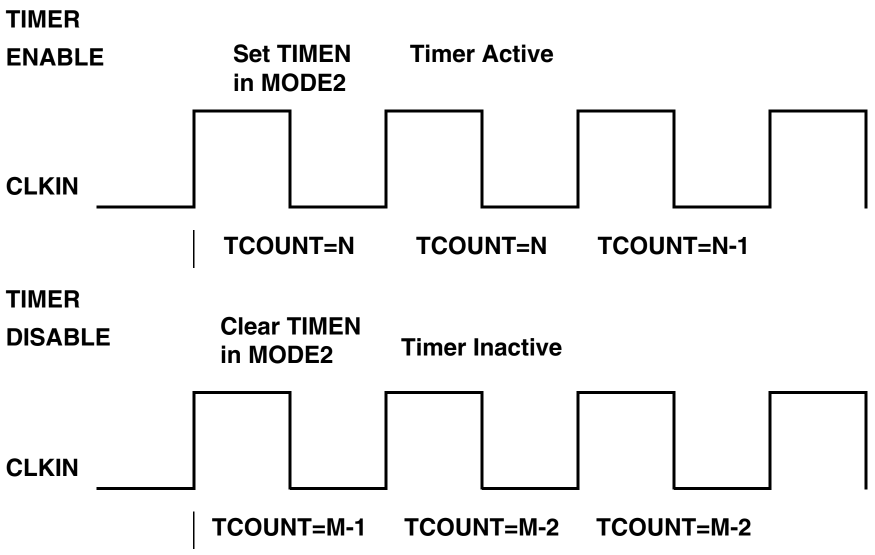


Figure 3-5. Timer Enable and Disable

Stacks and Sequencing

The sequencer includes a Program Counter (PC) stack, which appears in Figure 3-2 on page 3-4. At the start of a subroutine or loop, the sequencer pushes return addresses for subroutines (CALL/return instructions) and top-of-loop addresses for loops (DO/UNTIL) instructions onto the PC stack. The sequencer pops the PC stack during a return from interrupt (RTI), returns from subroutine (RTS), and loop termination.

The PC stack is 30 locations deep. The stack is full when all entries are occupied, is empty when no entries are occupied, and is overflowed if a push occurs when the stack is already full. Bits in the STKYx register indicate the PC stack full and empty states. Table A-5 on page A-19 lists the bits in the STYKx register. The STKYx bits that indicate PC stack status are:

- **PC stack full.** Bit 21 (PCFL) indicates that the PC stack is full (if 1) or not full (if 0)—Not a sticky bit, cleared by a POP.
- **PC stack empty.** Bit 22 (PCEM) indicates that the PC stack is empty (if 1) or not empty (if 0)—Not sticky, cleared by a PUSH.

The PC stack full condition causes a maskable interrupt (SOVFI). This interrupt occurs when the PC stack has 29 locations filled (the almost full state). The PC stack full interrupt occurs when one location is left, because the PC stack full service routine needs that last location for its return address.

The address of the top of the PC stack is available in the PC stack pointer (PCSTKP) register. The value of PCSTKP is zero when the PC stack is empty, is 1...30 when the stack contains data, and is 31 when the stack overflows. This register is a readable and writable register. A write to PCSTKP takes effect after a one-cycle delay. If the PC stack is overflowed, a write to PCSTKP has no effect.

The overflow and full flags provide diagnostic aid only. Programs should not use these flags for runtime recovery from overflow. Note that the status stack, loop stack overflow, and PC stack full conditions trigger a maskable interrupt.

The empty flags can ease stack saves to memory. Programs can monitor the empty flag when saving a stack to memory to determine when the processor has transferred all values.

Conditional Sequencing

The sequencer supports conditional execution with conditional logic that appears in Figure 3-2 on page 3-4. This logic evaluates conditions for conditional (IF) instructions and loop (DO/UNTIL) terminations. The conditions are based on information from the arithmetic status registers (ASTATx and ASTATy), the mode control 1 register (MODE1), the flag inputs, and the loop counter. For more information on arithmetic status, see "Using Computational Status" on page 2-8. When in SIMD mode, conditional execution is effected by the arithmetic status of both processing elements. For information on conditional sequencing in SIMD mode, see "SIMD Mode and Sequencing" on page 3-57.

Each condition that the processor evaluates has an assembler mnemonic. The condition mnemonics for conditional instructions appear in Table 3-19. For most conditions, the sequencer can test both true and false states. For example, the sequencer can evaluate ALU equal-to-zero (`EQ`) and ALU not-equal-to-zero (`NZ`).

To test conditions that do not appear in Table 3-19, a program can use the Test Flag (`TF`) condition generated from a Bit Test Flag (`BTF`) instruction. The `TF` flag is set or cleared as a result of a `BIT TEST` or `BIT XOR` instruction, which can test the contents of any of the processor's system registers, including `STKYx` and `STKYy`.

Table 3-19. IF Condition and DO/UNTIL Termination Mnemonics

Condition From	Description	True if...	Mnemonic
ALU	ALU = 0	AZ = 1	EQ
	ALU ≠ 0	AZ = 0	NE
	ALU > 0	footnote[1]	GT
	ALU < zero	footnote[2]	LT
	ALU ≥ 0	footnote[3]	GE
	ALU ≤ 0	footnote[4]	LE
	ALU carry	AC = 1	AC
	ALU not carry	AC = 0	NOT AC
	ALU overflow	AV = 1	AV
	ALU not overflow	AV = 0	NOT AV

Table 3-19. IF Condition and DO/UNTIL Termination Mnemonics (Cont'd)

Condition From	Description	True if…	Mnemonic
Multiplier	Multiplier overflow	MV = 1	MV
	Multiplier not overflow	MV= 0	NOT MV
	Multiplier sign	MN = 1	MS
	Multiplier not sign	MN = 0	NOT MS
Shifter	Shifter overflow	SV = 1	SV
	Shifter not overflow	SV = 0	NOT SV
	Shifter zero	SZ = 1	SZ
	Shifter not zero	SZ = 0	NOT SZ
Bit Test	Bit test flag true	BTF = 1	TF
	Bit test flag false	BTF = 0	NOT TF
Flag Input	Flag0 asserted	FI0 = 1	FLAG0_IN
	Flag0 not asserted	FI0 = 0	NOT FLAG0_IN
	Flag1 asserted	FI1 = 1	FLAG1_IN
	Flag1 not asserted	FI1 = 0	NOT FLAG1_IN
	Flag2 asserted	FI2 = 1	FLAG2_IN
	Flag2 not asserted	FI2 = 0	NOT FLAG2_IN
	Flag3 asserted	FI3 = 1	FLAG3_IN
	Flag3 not asserted	FI3 = 0	NOT FLAG3_IN
Mode	Bus master true		BM
	Bus master false		NOT BM

Table 3-19. IF Condition and DO/UNTIL Termination Mnemonics (Cont'd)

Condition From	Description	True if…	Mnemonic
Sequencer	Loop counter expired (Do)	CURLCNTR = 1	LCE
	Loop counter not expired (IF)	CURLCNTR ≠ 1	NOT ICE
	Always false (Do)	Always	FOREVER
	Always true (IF)	Always	TRUE

1 ALU greater than (GT) is true if: [$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN)] or $\overline{\text{AZ}}$ = 0
2 ALU less than (LT) is true if: [$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN and $\overline{\text{AZ}}$)] = 1
3 ALU greater equal (GE) is true if: [$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN and $\overline{\text{AZ}}$)] = 0
4 ALU lesser or equal (LT) is true if: [$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN)] or $\overline{\text{AZ}}$ = 1

The two conditions that do not have complements are `LCE/NOT LCE` (loop counter expired/not expired) and `TRUE/FOREVER`. The context of these condition codes determines their interpretation. Programs should use `TRUE` and `NOT LCE` in conditional (`IF`) instructions. Programs should use `FOREVER` and `LCE` to specify loop (`DO/UNTIL`) termination. A `DO FOREVER` instruction executes a loop indefinitely, until an interrupt or reset intervenes.

There are some restrictions on how programs may use conditions in `DO/UNTIL` loops. For more information, see "Restrictions on Ending Loops" on page 3-25 and "Restrictions on Short Loops" on page 3-26.

The bus master (`BM`) condition indicates whether the processor is the current bus master in a multiprocessor system. To enable testing this condition, a program must clear the `MODE1` register's Condition Code Select (`CSEL`) bits. Otherwise, the bus master condition is always false.

SIMD Mode and Sequencing

The processor supports a Single-Instruction, Multiple-Data (SIMD) mode. In this mode, both of the processor's processing elements (PEx and PEy) execute instructions and generate status conditions. For more information on SIMD computations, see "Secondary Processing Element (PEy)" on page 2-37.

Because the two processing elements can generate different outcomes, the sequencers must evaluate conditions from both elements (in SIMD mode) for conditional (IF) instructions and loop (DO/UNTIL) terminations. The processor records status for the PEx element in the ASTATx and STKYx registers. The processor records status for the PEy element in the ASTATy and STKYy registers. Table A-4 on page A-13 lists the bits in ASTATx and ASTATy, and Table A-5 on page A-19 lists the bits in STKYx and STKYy.

Even though the processor has dual processing elements, the sequencer does not have dual sets of stacks. The sequencer has one PC stack, one loop address stack, and one loop counter stack. The status bits for stacks are in STKYx and are not duplicated in STKYy. In SIMD mode, the status stack stores both ASTATx and ASTATy. A status stack PUSH or POP instruction in SIMD mode affects both registers in parallel.

While in SIMD mode, the sequencer evaluates conditions from both PE's for conditional (IF) and loop (DO/UNTIL) instructions. Table 3-20 summarizes how the sequencer resolves each conditional test when SIMD mode is enabled.

Table 3-20. Conditional Execution Summary

Conditional Operation	Conditional Outcome Depends On …
Compute Operations	Executes in each processing element independently depending on condition test in each processing element
Branches and Loops	Executes in sequencer depending on ANDing condition test in each processing element
Data Moves (from complementary pair[1] to complementary pair)	Executes move in each processing element (and/or memory) independently depending on condition test in each processing element. The same uncomplimented universal register is the source for each move, including X<->Y swap.
Data Moves (from uncomplemented Ureg register to complementary pair)	Executes move in each processing element (and/or memory) independently depending on condition test in each processing element. The same uncomplimented universal register is the source for each move, including X<->Y swap.
Data Moves (from complementary pair to uncomplemented register[2])	Executes explicit move to uncomplemented universal register depending on condition test in PEx only; no implicit move occurs. The same uncomplimented universal register is the source for each move, including X<->Y swap.
DAG Operations	Executes modify[3] in DAG depending on ORing condition test in each processing element

1 Complementary pairs are registers with SIMD complements, include PEx/y data registers and USTAT1/2, USTAT3/4, ASTATx/y, STKYx/y, and PX1/2 universal registers.
2 Uncomplemented registers are universal registers that do not have SIMD complements.
3 Post-modify operations follow this rule, but pre-modify operations always occur despite outcome.

Conditional Compute Operations

While in SIMD mode, a conditional compute operation can execute on both PE's, either PE, or neither PE, depending on the outcome of the status flag test. Flag testing is independently performed on each PE.

Conditional Branches and Loops

The processor executes a conditional branch (`JUMP` or `CALL`/return) or loop (`DO/UNTIL`) based on the result of AND'ing the condition tests on both PEx and PEy. A conditional branch or loop in SIMD mode occurs only when the condition is true in PEx and PEy.

Using complementary conditions (for example `EQ` and `NE`), programs can produce an OR'ing of the condition tests for branches and loops in SIMD mode. A conditional branch or loop that uses this technique should consist of a series of conditional compute operations. These conditional computes generate `NOP`s on the processing element where a branch or loop does not execute. For more information on programming in SIMD mode, see the *ADSP-21160 SHARC DSP Instruction Set Reference*.

Conditional Data Moves

The execution of a conditional (IF) data move (register-to-register and register-to/from-memory) instruction depends on three factors:

- The explicit data move depends on the evaluation of the conditional test in the PEx processing element
- The implicit data move depends on the evaluation of the conditional test in the PEy processing element
- Both moves depend on the types of registers used in the move

There are four cases for SIMD conditional data moves:

Case 1: Complementary Register Pair Data Move

In this case data moves from a complementary register pair to a complementary register pair. The processor executes the explicit move depending on the evaluation of the conditional test in the PEx processing element and the implicit move depending on the evaluation of the conditional test in the PEy processing element.

Example: Register–to–Memory Move — PEx Explicit Register

```
IF EQ DM(I0,M0) = R2;
```

For this instruction the processor is operating in SIMD mode, a register in the PEx data register file is the explicit register and `I0` is pointing to an even address in internal memory. Indirect addressing is shown in the instructions shown in this example. However, the same results occur using direct addressing. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 3-21.

Table 3-21. Register–to–Memory Moves – Complementary Pairs

Condition in PEx	Condition in PEy	Result	
AZx	**AZy**	**Explicit**	**Implicit**
0	0	NO data move occurs	NO data move occurs
0	1	NO data move occurs from r2 to location I0	s2 transfers to location (I0+1)
1	0	r2 transfers to location I0	NO data move occurs from s2 to location (I0+1)
1	1	r2 transfers to location I0	s2 transfers to location (I0+1)

Example: Register–to–Memory Move — PEy Explicit Register

```
IF EQ DM(I0,M0) = S2;
```

For this instruction the processor is operating in SIMD mode, a register in the PEy data register file is the explicit register and I0 is pointing to an even address in internal memory. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 3-22.

Table 3-22. Register–to–Register Moves – Complementary Pairs

Condition in PEx	**Condition in PEy**	**Result**	
AZx	**AZy**	**Explicit**	**Implicit**
0	0	NO data move occurs	NO data move occurs
0	1	NO data move occurs from s2 to location I0	r2 transfers to location I0+1
1	0	s2 transfers to location I0	NO data move occurs from r2 to location I0+1
1	1	s2 transfers to location I0	r2 transfers to location I0+1

Examples: Register–to–Register Move Instructions

```
IF EQ R8  = R2;
IF EQ PX1 = R2;
IF EQ USTAT1 = R2;
```

For these instruction the processor is operating in SIMD mode and registers in the PEx data register file are used as the explicit registers. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 3-23.

Table 3-23. Register–to–Register Moves – Complementary Pairs

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
0	0	NO data move occurs	NO data move occurs
0	1	NO data move to registers r9,px1,ustat1 occurs	s2 transfers to registers s9,px2 and ustat2
1	0	r2 transfers to registers r9,px1 and ustat1	NO data move to s9, px2, or ustat2 occurs
1	1	r2 transfers to registers r9,px1, and ustat1	s2 transfers to registers s9,px2,and ustat2

Examples: Register–to–Register Move Instructions

```
IF EQ R8  = S2;
IF EQ PX1 = S2;
IF EQ USTAT1 = S2;
```

For these instructions the processor is operating in SIMD mode and registers in the PEy data register file are used as explicit registers. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 3-24.

Table 3-24. Register–to–Register Moves – Complementary Register Pairs

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
0	0	NO data move occurs	NO data move occurs
0	1	NO data move to registers s9,px and ustat1 occurs	r2 transfers to registers s9,px2, and ustat2

Table 3-24. Register–to–Register Moves – Complementary Register Pairs (Cont'd)

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
1	0	s2 transfers to registers r9,px1 and ustat1	NO data move to registers s9,px2, and ustat2 occurs
1	1	s2 transfers to registers r9,px1, and ustat1	r2 transfers to registers s9,px2, and ustat2

Case 2: Uncomplemented–to–Complementary Register Move

In this case data moves from an uncomplemented register (Ureg without a SIMD complement) to a complementary register pair. The processor executes the explicit move depending on the evaluation of the conditional test in the PEx processing element. The processor executes the implicit move depending on the evaluation of the conditional test in the PEy processing element. In each processing element where the move occurs, the content of the source register is duplicated in destination.

Example: Register–to–Register Move

```
IF EQ R1 = PX;
```

While PX1 and PX2 are complementary registers, the combined PX register has no complementary register. For more information, see "Internal Data Bus Exchange" on page 5-10.

For this instruction the processor is operating in SIMD mode. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 3-24.

Table 3-25. Complementary–to–Uncomplemented Register Move

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
0	0	r1 remains unchanged	s1 remains unchanged
0	1	r1 remains unchanged	s1 gets px value
1	0	r1 gets px value	s1 remains unchanged
1	1	r1 gets px value	s1 gets px value

Case 3: Complementary Register => Uncomplimentary Register

In this case data moves from a complementary register pair to an uncomplemented register. The processor executes the explicit move to the uncomplemented universal register, depending on the condition test in the PEx processing element only. The processor does not perform an implicit move.

Example: Register–to–Register Move

```
IF EQ PX = R1;
```

For this instruction the processor is operating in SIMD mode. The data movement resulting from the evaluation of the conditional test in the PEx and PEy processing elements is shown in Table 3-26.

Table 3-26. Complementary–to–Uncomplemented Move

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
0	0	px remains unchanged	no implicit move
0	1	px remains unchanged	no implicit move

Table 3-26. Complementary–to–Uncomplemented Move (Cont'd)

Condition in PEx	Condition in PEy	Result	
AZx	AZy	Explicit	Implicit
1	0	r1 40-bit explicit move to px	no implicit move
1	1	r1 40-bit explicit move to px	no implicit move

For more details on PX register transfers, refer to "Internal Data Bus Exchange" on page 5-10.

Case 4: Data Move Involves External Memory or IOP Memory Space

Conditional data moves from a complementary register pair to an uncomplemented register with an access to external memory space or IOP memory space. This results in unexpected behavior and should not be used.

```
IF EQ DM(I0,M0) = R2;
IF EQ DM(I0,M0) = S2;
```

For these instruction the processor is operating in SIMD mode and the explicit register is either a PEx register or PEy register. I0 points to either external memory space or IOP memory space.

Indirect addressing is shown in the instructions shown in this example. However, the same results occur using direct addressing.

Conditional DAG Operations

Conditional post-modify DAG operations update the DAG register based on OR'ing of the condition tests on both processing elements. Actual data movement involved in a conditional DAG operation is based on independent evaluation of condition tests in PEx and PEy. Only the post modify update is based on the OR'ing of the these conditional tests.

Conditional pre-modify DAG operations behave differently. The DAGs always pre-modify an index, independent of the outcome of the condition tests on each processing element.

4 DATA ADDRESS GENERATOR

The processor's Data Address Generators (DAGs) generate addresses for data moves to and from Data Memory (DM) and Program Memory (PM). By generating addresses, the DAGs let programs refer to addresses indirectly, using a DAG register instead of an absolute address. The DAGs architecture, which appears in Figure 4-1, supports several functions that minimize overhead in data access routines. These functions include:

- **Supply address and post-modify**—provides an address during a data move and auto-increments the stored address for the next move.
- **Supply pre-modified address**—provides a modified address during a data move without incrementing the stored address.
- **Modify address**—increments the stored address without performing a data move.
- **Bit-reverse address**—provides a bit-reversed address during a data move without reversing the stored address.
- **Broadcast data moves**—performs dual data moves to complementary registers in each processing element to support SIMD mode.

As shown in Figure 4-1, each DAG has four types of registers. These registers hold the values that the DAG uses for generating addresses. The four types of registers are:

- **Index registers (I0-I7 for DAG1 and I8-I15 for DAG2).** An index register holds an address and acts as a pointer to memory. For example, the DAG interprets `DM(I0,0)` and `PM(I8,0)` syntax in an instruction as addresses.
- **Modify registers (M0-M7 for DAG1 and M8-M15 for DAG2).** A modify register provides the increment or step size by which an index register is pre- or post-modified during a register move. For example, the `DM(I0, M1)` instruction directs the DAG to output the address in register `I0` then modify the contents of `I0` using the `M1` register.
- **Length and Base registers (L0-L7 and B0-B7 for DAG1 and L8-L15 and B8-B15 for DAG2).** Length and base registers setup the range of addresses and the starting address for a circular buffer. For more information on circular buffers, see "Addressing Circular Buffers" on page 4-12.

Setting DAG Modes

The `MODE1` register controls the operating mode of the DAGs. Table A-2 on page A-3 lists all the bits in `MODE1`. The following bits in `MODE1` control Data Address Generator modes:

- **Circular buffering enable.** Bit 24 (`CBUFEN`) enables circular buffering (if 1) or disables circular buffering (if 0).
- **Broadcast register loading enable, DAG1-I1.** Bit 23 (`BDCST1`) enables register broadcast loads to complementary registers from I1 indexed moves (if 1) or disables broadcast loads (if 0).

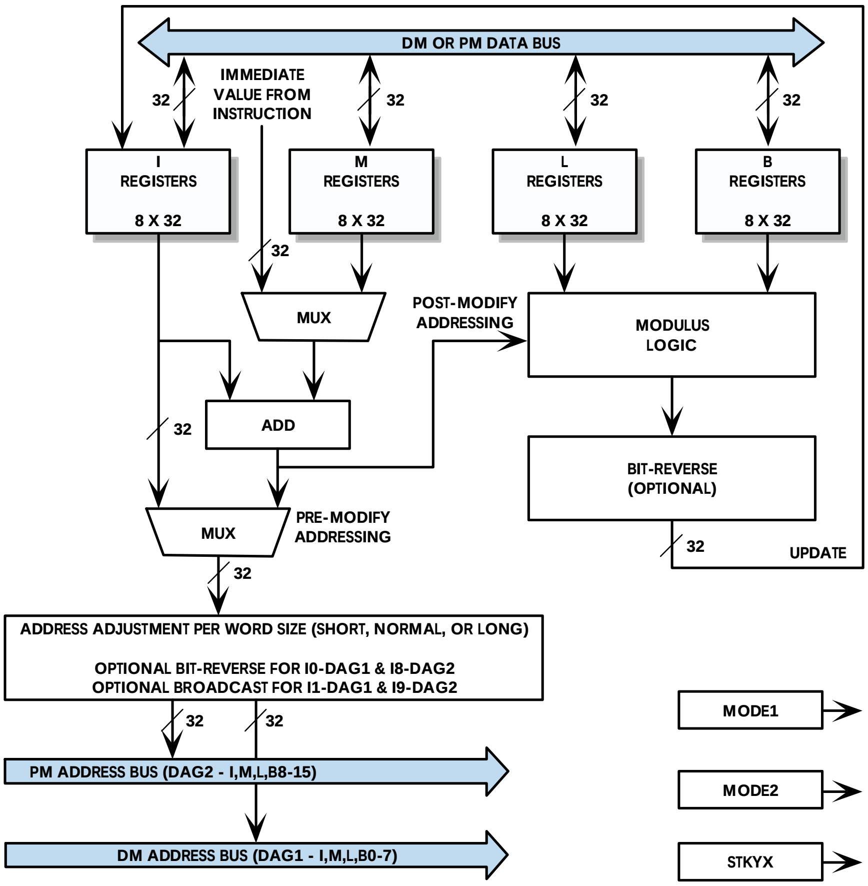

DM OR PM DATA BUS
IMMEDIATE VALUE FROM INSTRUCTION
32
I REGISTERS 8 X 32
M REGISTERS 8 X 32
L REGISTERS 8 X 32
B REGISTERS 8 X 32
MUX
POST-MODIFY ADDRESSING
MODULUS LOGIC
ADD
BIT-REVERSE (OPTIONAL)
MUX
PRE-MODIFY ADDRESSING
UPDATE
ADDRESS ADJUSTMENT PER WORD SIZE (SHORT, NORMAL, OR LONG)
OPTIONAL BIT-REVERSE FOR I0-DAG1 & I8-DAG2
OPTIONAL BROADCAST FOR I1-DAG1 & I9-DAG2
PM ADDRESS BUS (DAG2 - I,M,L,B8-15)
DM ADDRESS BUS (DAG1 - I,M,L,B0-7)
MODE1
MODE2
STKYX

Figure 4-1. Data Address Generator (DAG) Block Diagram

- **Broadcast register loading enable, DAG2-I9.** Bit 22 (`BDCST9`) enables register broadcast loads to complementary registers from I9 indexed moves (if 1) or disables broadcast loads (if 0).

- **SIMD mode enable.** Bit 21 (`PEYEN`) enables computations in PEy—SIMD mode—(if 1) or disables PEy—SISD mode—(if 0). For more information on SIMD mode, see "Secondary Processing Element (PEy)" on page 2-37.

- **Secondary registers for DAG2 lo, I,M,L,B8-11.** Bit 6 (SRD2L)
  Secondary registers for DAG2 hi, I,M,L,B12-15. Bit 5 (SRD2H)
  Secondary registers for DAG1 lo, I,M,L,B0-3. Bit 4 (SRD1L)
  Secondary registers for DAG1 hi, I,M,L,B4-7. Bit 3 (SRD1H)
  These bits select the corresponding secondary register set (if 1) or select the corresponding primary register set—the set that is available at reset—(if 0).

- **Bit-reverse addressing enable, DAG1-I0.** Bit 1 (BR0) enables bit-reversed addressing on I0 indexed moves (if 1) or disables bit-reversed addressing (if 0).

- **Bit-reverse addressing enable, DAG2-I8.** Bit 0 (BR8) enables bit-reversed addressing on I8 indexed moves (if 1) or disables bit-reversed addressing (if 0).

Circular Buffering Mode

The CBUFEN bit in the MODE1 register enables circular buffering—a mode in which the DAG supplies addresses ranging within a constrained buffer length (set with an L register), starting at a base address (set with a B register), and incrementing the addresses on each access by a modify value (set with an M register).

For revision 1.0 and greater of ADSP-21161 processor, the Circular Buffer Enable bit (CBUFEN) in SYSCON is set (=1) upon reset. For earlier silicon revisions 0.x, this bit is cleared (=0) upon reset. This change was made to ensure code compatibility with the ADSP-2106x SHARC family (ADSP-21060/1/2 and ADSP-21065L) where circular buffering is active upon reset.

However, circular buffering is disabled upon reset for the ADSP-21160. Make note of this when porting code from ADSP-21160 to ADSP-21161 processor.

For more information on setting up and using circular buffers, see "Addressing Circular Buffers" on page 4-12. When using circular buffers, the DAGs can generate an interrupt on buffer overflow (wrap around). For more information, see "Using DAG Status" on page 4-8.

Broadcast Loading Mode

The `BDCST1` and `BDCST9` bits in the `MODE1` register enable broadcast loading mode—multiple register loads from a single load command. When the `BDCST1` bit is set (1), the DAG performs a dual data register load on instructions that use the `I1` register for the address. The DAG loads both the named register (explicit register) in one processing element and loads that register's complementary register (implicit register) in the other processing element. The `BDCST9` bit in the `MODE1` register enables this feature for the `I9` register.

Enabling either DAG1 or DAG2 register load broadcasting has no effect on register stores or loads to universal registers other than the register file data registers. Table 4-1 demonstrates the effects of a register load operation on both processing elements with register load broadcasting enabled. In Table 4-1, note that Rx and Sx are complementary data registers.

Table 4-1. Dual Processing Element Register Load Broadcasts

Instruction syntax	Rx = DM(I1,Ma); {Syntax #1}
	Rx = PM(I9,Mb); {Syntax #2}
	Rx = DM(I1,Ma), Rx = PM(I9,Mb); {Syntax #3}

PEx explicit operations	Rx = DM(I1,Ma); {Explicit #1}
	Rx = PM(I9,Mb); {Explicit #2}
	Rx = DM(I1,Ma), Rx = PM(I9,Mb); {Explicit #3}

PEy implicit operations	Sx = DM(I1,Ma); {Implicit #1}
	Sx = PM(I9,Mb); {Implicit #2}
	Sx = DM(I1,Ma), Sx = PM(I9,Mb); {Implicit #3}

1. Note that the letters a and b (as in Ma or Mb) indicate numbers for modify registers in DAG1 and DAG2. The letter a indicates a DAG1 register and can be replaced with 0 through 7. The letter b indicates a DAG2 register and can be replaced with 8 through 15.

The PEYEN bit (SISD/SIMD mode select) does not influence broadcast operations. Broadcast loading is particularly useful in SIMD applications where the algorithm needs identical data loaded into each processing element. For more information on SIMD mode (in particular, a list of complementary data registers), see "Secondary Processing Element (PEy)" on page 2-37.

Alternate (Secondary) DAG Registers

Each DAG has an alternate register set. To facilitate fast context switching, the processor includes alternate register sets for data, results, and data address generator registers. Bits in the MODE1 register control when alternate registers become accessible. While inaccessible, the contents of alternate registers are not effected by processor operations. Note that there is a one cycle latency between writing to MODE1 and being able to access an alternate register set. The alternate register sets for the DAGs are described in this section. For more information on alternate data and results registers, see "Alternate (Secondary) Data Registers" on page 2-32.

Bits in the MODE1 register can activate alternate register sets within the DAGs: the lower half of DAG1 (I,M,L,B0-3), the upper half of DAG1 (I,M,L,B4-7), the lower half of DAG2 (I,M,L,B8-11), and the upper half of DAG2 (I,M,L,B12-15). Figure 4-1 shows the DAG's primary and alternate register sets.

To share data between contexts, a program places the data to be shared in one half of either the current DAG's registers or the other DAG's registers and activates the alternate register set of the other half. The following example demonstrates how code should handle the one cycle of latency

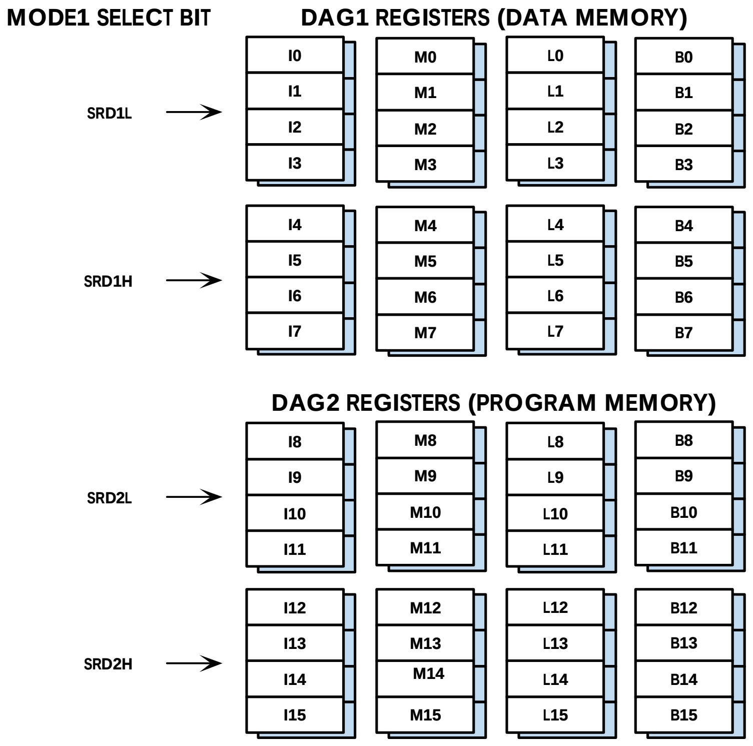


Figure 4-2. Data Address Generator Primary and Alternate Registers

from the instruction setting the bit in `MODE1` to when the alternate registers may be accessed. Note that it is possible to use any instruction that does not access the switching register file instead of an `NOP` instruction.

```
BIT SET MODE1 SRD1L;      /* Activate alternate dag1 lo regs */
NOP;                      /* Wait for access to alternates */
R0=DM(i0,m1);
```

Bit-reverse Addressing Mode

The BR0 and BR8 bits in the MODE1 register enable bit-reverse addressing mode—outputting addresses in reverse bit order. When BR0 is set (1), DAG1 bit-reverses 32-bit addresses output from I0. When BR8 is set (1), DAG2 bit-reverses 32-bit addresses output from I8. The DAGs only bit-reverse the address output from I0 or I8; the contents of these registers are not reversed. Bit-reverse addressing mode effects both pre-modify and post-modify operations. The following example demonstrates how bit-reverse mode effects address output:

```
BIT SET Mode1 BR0;      /* Enables bit-rev. addressing for DAG1 */
I0=0x8a000;             /* Loads I0 with the bit reverse of the
buffer’s base address, DM(0x51000) */
M0=0x4000000;           /* Loads M0 with value for post-modify */
R1=DM(I0,M0);           /* Loads r1 with contents of DM address
DM(0x51000), which is the bit-reverse of 0x8a000, then post modi-
fies I0 for the next access with (0x8a000 + 0x4000000)=0x408a000,
which is the bit-reverse of DM(0x51020) */
```

In addition to bit-reverse addressing mode, the processor supports a bit-reverse instruction (BITREV). This instruction bit-reverses the contents of the selected register. For more information on the BITREV instruction, see “Modifying DAG Registers” on page 4-17 or the *ADSP-21160 SHARC DSP Instruction Set Reference*.

Using DAG Status

As described in “Addressing Circular Buffers” on page 4-12, the DAGs can provide addressing for a constrained range of addresses, repeatedly cycling through this data (or buffer). A buffer overflow (or wrap around) occurs each time the DAG circles past the buffer’s base address.

The DAGs can provide buffer overflow information when executing circular buffer addressing for I7 or I15. When a buffer overflow occurs (a circular buffering operation increments the I register past the end of the buffer), the appropriate DAG updates a buffer overflow flag in a sticky status (STKYx) register. A buffer overflow can also generate a maskable interrupt. Two ways to use buffer overflows from circular buffering are:

- **Interrupts.** Enable interrupts and use an interrupt service routine to handle the overflow condition immediately. This method is appropriate if it is important to handle all overflows as they occur; for example in a "ping-pong" or swap I/O buffer pointers routine.
- **STKYx registers.** Use the BIT TST instruction to examine overflow flags in the STKY register after a series of operations. If an overflow flag is set, the buffer has overflowed—wrapped around—at least once. This method is useful when overflow handling is not critical.

DAG Operations

The processor's DAGs perform several types of operations to generate data addresses. As shown in Figure 4-1 on page 4-3, the DAG registers and the MODE1, MODE2, and STKYx registers all contribute to DAG operations. The following sections provide details on DAG operations:

- "Addressing With DAGs" on page 4-10
- "Addressing Circular Buffers" on page 4-12
- "Modifying DAG Registers" on page 4-17

An important item to note from Figure 4-1 on page 4-3 is that the DAG automatically adjusts the output address per the word size of the address location (short word, normal word, or long word). This address adjustment lets internal memory use the address directly.

SISD/SIMD mode, access word size, and data location (internal/external) all influence data access operations.

Addressing With DAGs

The DAGs support two types of modified addressing—generating an address that is incremented by a value or a register. In pre-modify addressing, the DAG adds an offset (modifier), either an M register or an immediate value, to an I register and outputs the resulting address. Pre-modify addressing does not change (or update) the I register. The other type of modified addressing is post-modify addressing. In post-modify addressing, the DAG outputs the I register value unchanged then adds an M register or immediate value, updating the I register value. Figure 4-3 compares pre- and post-modify addressing.

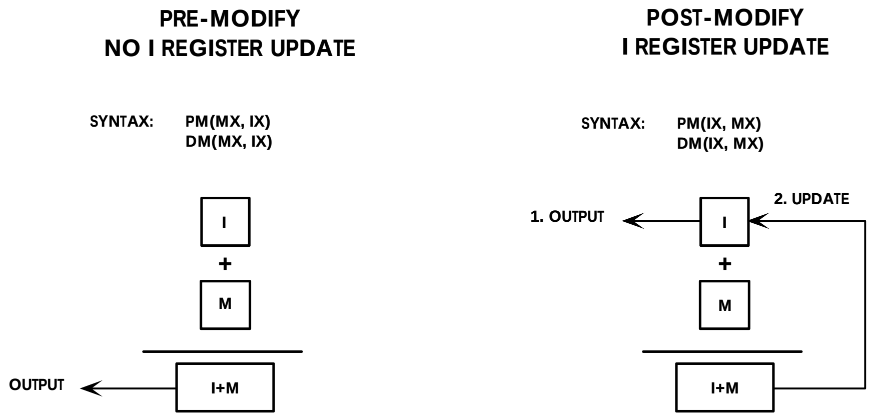


Figure 4-3. Pre-Modify and Post-Modify Operations

The difference between pre-modify and post-modify instructions in the processor's assembly syntax is the position of the index and modifier in the instruction. If the I register comes before the modifier, the instruction is a post-modify operation. If the modifier comes before the I register, the instruction is a pre-modify without update operation. The following instruction accesses the program memory location indicated by the value in I15 and writes the value I15 + M12 to the I15 register:

```
R6 = PM(I15,M12); /* Post-modify addressing with update */
```

By comparison, the following instruction accesses the program memory location indicated by the value I15 + M12 and does not change the value in I15:

```
R6 = PM(M12,I15); /* Pre-modify addressing without update */
```

Modify (M) registers can work with any index (I) register in the same DAG (DAG1 or DAG2). For a list of I and M registers and their DAGs, see Figure 4-2 on page 4-7.

Instructions can use a number (immediate value), instead of an M register, as the modifier. The size of an immediate value that can modify an I register depends on the instruction type. For all single data access operations, modify immediate values can be up to 32 bits wide. Instructions that combine DAG addressing with computations limit the size of the modify immediate value. In these instructions (multifunction computations), the modify immediate values can be up to 6 bits wide. The following example instruction accepts up to 32-bit modifiers:

```
R1=DM(0x40000000,I1);    /* DM address = I1+0x4000 0000 */
```

The following example instruction accepts up to 6-bit modifiers:

```
F6=F1+F2,PM(I8,0x0B)=ASTAT; /* PM address = I8, I8=I8+0x0B */
```

Note that pre-modify addressing operations must not change the memory space of the address. For example, pre-modifying an address in the processor's internal memory space should not generate an address in external memory space.

Addressing Circular Buffers

The DAGs support addressing circular buffers—a range of addresses containing data that the DAG steps through repeatedly, “wrapping around” to repeat stepping through the range of addresses in a circular pattern. To address a circular buffer, the DAG steps the index pointer (`I` register) through the buffer, post-modifying and updating the index on each access with a positive or negative modify value (`M` register or immediate value). If the index pointer falls outside the buffer, the DAG subtracts or adds the length of the buffer from or to the value, wrapping the index pointer back to the start of the buffer. The DAG’s support for circular buffer addressing appears in Figure 4-1 on page 4-3, and an example of circular buffer addressing appears in Figure 4-4.

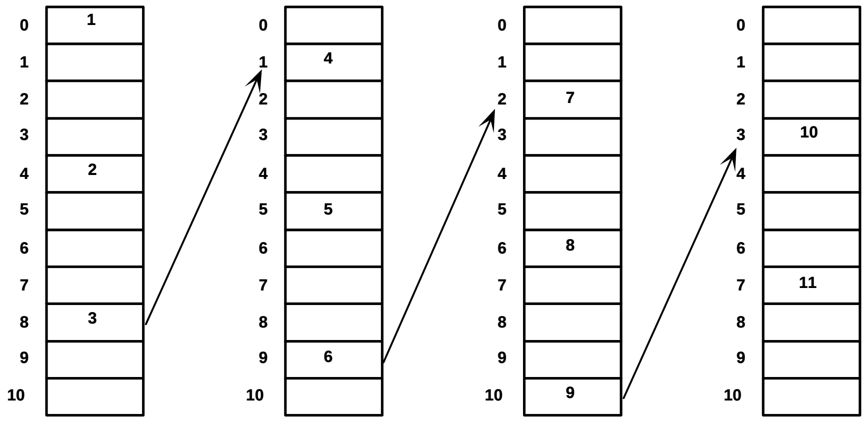
The starting address that the DAG wraps around is called the buffer’s base address (`B` register). There are no restrictions on the value of the base address for a circular buffer.

Circular buffering may only use post-modify addressing. The DAG’s architecture, as shown in Figure 4-1 on page 4-3, cannot support pre-modify addressing for circular buffering, because circular buffering requires that the index be updated on each access.

It is important to note that the DAGs do not detect memory map overflow or underflow. If the address post-modify produces I+M > 0xFFFF FFFF or I–M < 0, circular buffering may not function correctly. Also, the length of a circular buffer should not let the buffer straddle the top of the memory map. For more information on the processor’s memory map, see “Internal Address and Data Buses” on page 5-7.

```
THE FOLLOWING SYNTAX SETS UP AND ACCESSES A CIRCULAR BUFFER WITH:
     LENGTH = 11
     BASE ADDRESS = 0X55000
     MODIFIER = 4

     BIT SET MODE1 CBUFEN;          /* ENABLES CIRCULAR BUFFER ADDRESSING; JUST ONCE IN PROGRAM  */
     B0 = 0X55000;                  /* LOADS B0 AND L0 REGISTERS WITH BASE ADDRESS */
     L0 = 0XB;                      /* LOADS L0 REGISTER WITH LENGTH OF BUFFER */
     M1 = 0X4;                      /* LOADS M1 WITH MODIFIER OR STEP SIZE */
     LCNTR = 11, DO MY_CIR_BUFFER UNTIL LCE;   /* SETS UP A LOOP CONTAINING BUFFER ACCESSES */
     R0 = DM(I0,M1);                /* AN ACCESS WITHIN THE BUFFER USES POST MODIFY ADDRESSING */
     ...                            /* OTHER INSTRUCTIONS IN THE MY_CIR_BUFFER LOOP */
     MY_CIR_BUFFER: NOP;            /* END OF MY_CIR_BUFFER LOOP */
```



THE COLUMNS ABOVE SHOW THE SEQUENCE IN ORDER OF LOCATIONS ACCESSED IN ONE PASS.
NOTE THAT "0" ABOVE IS ADDRESS DM(0X55000). THE SEQUENCE REPEATS ON SUBSEQUENT PASSES.

Figure 4-4. Circular Data Buffers

As shown in Figure 4-4, programs use the following steps to set up a circular buffer:

1. Enable circular buffering (`BIT SET Mode1 CBUFEN;`). This operation is only needed once in a program.

2. Load the buffer's base address into the `B` register. This operation automatically loads the corresponding `I` register.

3. Load the buffer's length into the corresponding `L` register. For example, `L0` corresponds to `B0`.

4. Load the modify value (step size) into an `M` register in the corresponding DAG. For example, `M0` through `M7` correspond to `B0`. Alternatively, the program can use an immediate value for the modifier.

After this set up, the DAGs use the modulus logic in Figure 4-1 on page 4-3 to process circular buffer addressing.

On the ADSP-21161 processor, programs enable circular buffering by setting the `CBUFEN` bit in the `MODE1` register. This bit has a corresponding mask bit in the `MMASK` register. Setting the corresponding `MMASK` bit causes the `CBUFEN` bit to be cleared following a push status instruction (`PUSH STS`), the execution of an external interrupt, timer interrupt, or vectored interrupt. This feature lets programs disable circular buffering while in an interrupt service routine that does not use circular buffering. By disabling circular buffering, the routine does not need to save and restore the DAG's `B` and `L` registers.

Clearing the `CBUFEN` bit disables circular buffering for all data load and store operations. The DAGs perform normal post-modify load and store accesses instead, ignoring the `B` and `L` register values. Note that a write to a `B` register modifies the corresponding `I` register, independent of the state of the `CBUFEN` bit. The `MODIFY` instruction executes independent of the state of the `CBUFEN` bit. The `MODIFY` instruction always performs circular buffer modify of the index registers if the corresponding `B` and `L` registers are set up, independent of the state of the `CBUFEN` bit.

For revision 1.0 and greater of ADSP-21161 processor, the Circular Buffer Enable bit (`CBUFEN`) in `SYSCON` is set (=1) upon reset. For earlier silicon revisions 0.x, this bit is cleared (=0) upon reset. This change was made to ensure code compatibility with the ADSP-2106x SHARC family (ADSP-21060/1/2 and ADSP-21065L) where circular buffering is active upon reset.

However, circular buffering is disabled upon reset for the ADSP-21160. Make note of this when porting code from ADSP-21160 to ADSP-21161 processor.

On the first post-modify access to the buffer, the DAG outputs the `I` register value on the address bus then modifies the address by adding the modify value. If the updated index value is within the buffer length, the DAG writes the value to the `I` register. If the updated value is outside the buffer length, the DAG subtracts (positive) or adds (negative) the `L` register value before writing the updated index value to the `I` register. In equation form, these post-modify and wrap around operations work as follows:

- If M is positive:

  Inew = I_{old} + M if I_{old} + M < Buffer base + length (end of buffer)

  $I_{new} = I_{old} + M - L$ if $I_{old} + M \geq$ Buffer base + length (end of buffer)

- If M is negative:

  $I_{new} = I_{old} + M$ if $I_{old} + M \geq$ Buffer base (start of buffer)

  $I_{new} = I_{old} + M + L$ if $I_{old} + M <$ Buffer base (start of buffer)

The DAGs use all four types of DAG registers for addressing circular buffers. These registers operate as follows for circular buffering:

- The index (`I`) register contains the value that the DAG outputs on the address bus.

- The modify (M) register contains the post-modify amount (positive or negative) that the DAG adds to the `I` register at the end of each memory access. The `M` register can be any `M` register in the same DAG as the `I` register and does not have to have the same number. The modify value also can be an immediate value instead of an M register. The size of the modify value, whether from an M register or immediate, must be less than the length (L register) of the circular buffer.

- The length (L) register sets the size of the circular buffer and the address range that the DAG circulates the I register through. L must be positive and cannot have a value greater than $2^{31} - 1$. If an L register's value is zero, its circular buffer operation is disabled.

- The base (B) register, or the B register plus the L register, is the value that the DAG compares the modified I value with after each access. When the B register is loaded, the corresponding I register is simultaneously loaded with the same value. When I is loaded, B is not changed. Programs can read the B and I registers independently.

There is one set of registers (I7 and I15) in each DAG that can generate an interrupt on circular buffer overflow (address wraparound). For more information, see "Using DAG Status" on page 4-8.

When a program needs to use I7 or I15 without circular buffering and the processor has the circular buffer overflow interrupts unmasked, the program should disable the generation of these interrupts by setting the B7/B15 and L7/L15 registers to values that prevent the interrupts from occurring. If I7 were accessing the address range 0x1000–0x2000, the program could set B7=0x0000 and L7=0xFFFF. Because the processor generates the circular buffer interrupt based on the wrap around equations on page 4-15, setting the L register to zero does not necessarily achieve the desired results. If the program is using either of the circular buffer overflow interrupts, it should avoid using the corresponding I register(s) (I7 or I15) where interrupt branching is not needed.

In the case of circular buffer overflow interrupts, if `CBUFEN` = 1 and register `L7` = 0 (or `L15` = 0), then the `CB7I` (or `CB15I`) interrupt occurs at every change of `I7` (or `I15`), after the index register (`I7` or `I15`) crosses the base register (`B7` or `B15`) value. This behavior is independent of the context of the DAG registers, both primary and alternate.

When a Long word access, SIMD access, or Normal word access (with `LW` option) crosses the end of the circular buffer, the processor completes the access before responding to the end of buffer condition.

Modifying DAG Registers

The DAGs support two operations that modify an address value in an index register without outputting an address. These two operations, address bit-reversal and address modify, are useful for bit-reverse addressing and maintaining pointers.

The `MODIFY` instruction modifies addresses in any DAG index register (`I0-I15`) without accessing memory. If the `I` register's corresponding `B` and `L` registers are set up for circular buffering, a `MODIFY` instruction performs the specified buffer wrap around (if needed). The syntax for `MODIFY` is similar to post-modify addressing (index, then modifier). `MODIFY` accepts either a 32-bit immediate values or an `M` register as the modifier. The following example adds 4 to `I1` and updates `I1` with the new value:

```
MODIFY(I1,4);
```

The `BITREV` instruction modifies and bit-reverses addresses in any DAG index register (`I0-I15`) without accessing memory. This instruction is independent of the bit-reverse mode. The `BITREV` instruction adds a 32-bit immediate value to a DAG index register, bit-reverses the result, and writes the result back to the same index register. The following example adds 4 to `I1`, bit-reverses the result, and updates `I1` with the new value:

```
BITREV(I1,4);
```

Addressing in SISD and SIMD Modes

Single-Instruction, Multiple-Data (SIMD) mode (PEYEN bit=1) does not change the addressing operations in the DAGs, but it does change the amount of data that moves during each access. The DAGs put the same addresses on the address buses in SIMD and SISD modes. In SIMD mode, the processor's memory and processing elements get data from the locations named (explicit) in the instruction syntax and complementary (implicit) locations. For more information on data moves between registers, see "Secondary Processing Element (PEy)" on page 2-37.

DAGs, Registers, and Memory

DAG registers are part of the processor's universal register set. Programs may load the DAG registers from memory, from another universal register, or with an immediate value. Programs may store DAG registers' contents to memory or to another universal register.

The DAG's registers support the bidirectional register-to-register transfers that are described in "SIMD (Computational) Operations" on page 2-43. When the DAG register is a source of the transfer, the destination can be a register file data register. This transfer results in the contents of the single source register being duplicated in complementary data registers in each processing element.

Programs should use care in the case where the DAG register is a destination of a transfer from a register file data register source. Programs should use a conditional operation to select either one processing element or neither as the source. Having both processing elements contribute a source value results in the PEx element's write having precedence over the PEy element's write.

In the case where a DAG register is both source and destination, the data move operation executes the same as it would if SIMD mode were disabled (PEYEN cleared).

DAG Register-to-Bus Alignment

There are three word alignment cases for DAG registers and PM or DM data buses: Normal word, Extended-precision Normal word, and Long word.

The DAGs align normal word (32-bit) addressed transfers to the low order bits of the buses. These transfers between memory and 32-bit DAG1 or DAG2 registers use the 64-bit DM and PM data buses. Figure 4-5 illustrates these transfers.

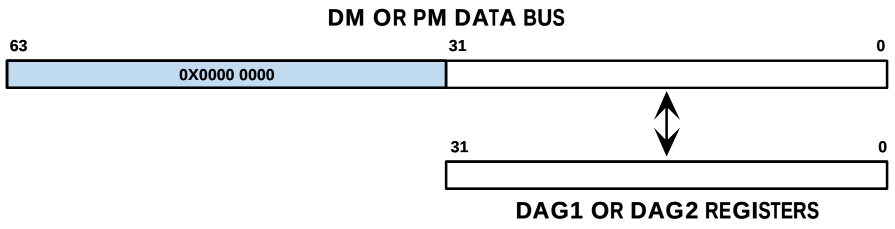


Figure 4-5. Normal Word (32-bit) DAG Register Memory Transfers

The DAGs align extended-precision normal word (40-bit) addressed transfers or register-to-register transfers to bits 39-8 of the buses. These transfers between a 40-bit data register and 32-bit DAG1 or DAG2 registers use the 64-bit DM and PM data buses. Figure 4-6 illustrates these transfers.

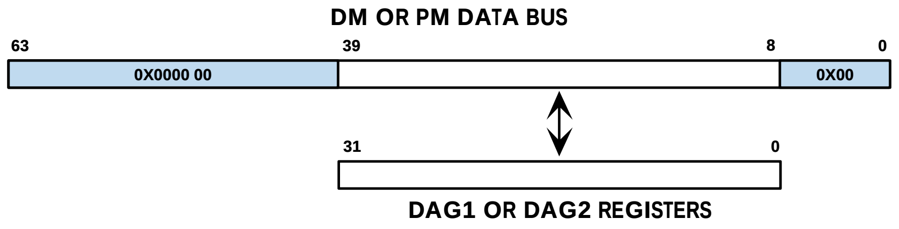


Figure 4-6. DAG Register to Data Register Transfers

Long word (64-bit) addressed transfers between memory and 32-bit DAG1 or DAG2 registers target double DAG registers and use the 64-bit DM and PM data buses. Figure 4-7 illustrates how the bus works in these transfers.

If the Long word transfer specifies an even-numbered DAG register (e.g., `I0` or `I2`), then the even numbered register value transfers on the lower half of the 64-bit bus, and the even numbered register + 1 value transfers on the upper half (bits 63-32) of the bus.

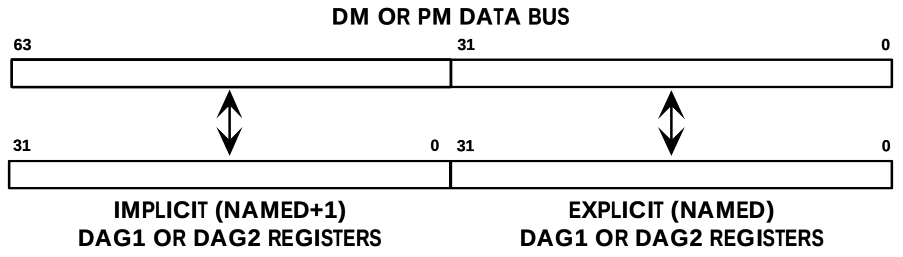


Figure 4-7. Long Word DAG Register to Data Register Transfers

If the Long word transfer specifies an odd numbered DAG register (e.g., `I1`, or `B3`), the odd numbered register value transfers on the lower half of the 64-bit bus, and the odd numbered register - 1 value (`I0` or `B2` in this example) transfers on the upper half (bits 63-32) of the bus.

In both the even- and odd-numbered cases, the explicitly specified DAG register sources or sinks bits 31-0 of the Long word addressed memory.

DAG Register Transfer Restrictions

The two types of transfer restrictions are hold-off conditions and illegal conditions that the processor does not detect.

For certain instruction sequences involving transfers to and from DAG registers, an extra (`NOP`) cycle is automatically inserted by the processor. When an instruction that loads a DAG register is followed by an instruction that uses any register in the same DAG register pair[1] for data addressing, modify instructions, or indirect jumps, the processor inserts an extra (`NOP`) cycle between the two instructions. This hold-off happens because the same bus is needed by both operations in the same cycle. So, the second operation must be delayed. The following case causes a delay because it exhibits a write/read dependency in which `I0` is written in one cycle. The results of that register write are not available to a register read for one cycle. Note that if either instruction had specified `I1`, the stall would still occur, because the processor's DAG register transfers can occur in pairs. The DAG detects write/read dependencies with a register pair granularity:

```
I0=8;
DM(I0,M1)=R1;
```

[1] DAG register are accessible in pair granularity for single-cycle access. The pairings are odd-even. For example I0 and I1 are a pair, and I2 and I3 are a pair.

Certain other sequences of instructions cause incorrect results on the processor and are flagged as errors by processor assembler software. These types of instructions can execute on the processor, but cause incorrect results:

- An instruction that stores a DAG register in memory using indirect addressing from the same DAG, with or without update of the index register. The instruction writes the wrong data to memory or updates the wrong index register.

  Do not try these: `DM(M2,I1)=I0;` or `DM(I1,M2)=I0;`
  These example instructions do not work because `I0` and `I1` are both DAG1 registers.

- An instruction that loads a DAG register from memory using indirect addressing from the same DAG, with update of the index register. The instruction either loads the DAG register or updates the index register, but not both.

  Do not try this: `L2=DM(I1,M0);`
  This example instruction does not work because `L2` and `I1` are both DAG1 registers.

DAG Instruction Summary

Table 4-3 through, Table 4-9 list the DAG instructions. For more information on assembly language syntax, see the *ADSP-21160 SHARC DSP Instruction Set Reference*. In these tables, note the meaning of the following symbols:

- **I15-8** indicates a DAG2 index register: `I15`, `I14`, `I13`, `I12`, `I11`, `I10`, `I9`, or `I8`, and **I7-0** indicates a DAG1 index register `I7`, `I6`, `I5`, `I4`, `I3`, `I2`, `I1`, or `I0`.
- **M15-8** indicates a DAG2 modify register: `M15`, `M14`, `M13`, `M12`, `M11`, `M10`, `M9`, or `M8`, and **M7-0** indicates a DAG1 modify register `M7`, `M6`, `M5`, `M4`, `M3`, `M2`, `M1`, or `M0`.
- **Ureg** indicates any universal register; For a list of the processor's universal registers, see Table A-1 on page A-2.
- **Dreg** indicates any data register; For a list of the processor's data registers, see the Data Register File registers that are listed in Table A-1 on page A-2.
- **Data32** indicates any 32-bit value, and **Data6** indicates any 6-bit value

Table 4-2. Post-Modify Addressing, Modified By M Register and Updating I Register

DM(I7-0,M7-0)=Ureg (LW); {DAG1}
PM(I15-8,M15-8)=Ureg (LW); {DAG2}
Ureg=DM(I7-0,M7-0) (LW); {DAG1}
Ureg=PM(I15-8,M15-8) (LW); {DAG2}
DM(I7-0,M7-0)=Data32; {DAG1}
PM(I15-8,M15-8)=Data32; {DAG2}

Table 4-3. Post-Modify Addressing, Modified By 6-Bit Data and Updating I Register

DM(I7-0,Data6)=Dreg; {DAG1}
PM(I15-8,Data6)=Dreg; {DAG2}
Dreg=DM(I7-0,Data6); {DAG1}
Dreg=PM(I15-8,Data6); {DAG2}

Table 4-4. Pre-Modify Addressing, Modified By M Register (No I Register Update)

DM(M7-0,I7-0)=Ureg (LW); {DAG1}
PM(M15-8,I15-8)=Ureg (LW); {DAG2}
Ureg=DM(M7-0,I7-0) (LW); {DAG1}
Ureg=PM(M15-8,I15-8) (LW); {DAG2}

Table 4-5. Pre-Modify Addressing, Modified By 6-Bit Data (No I Register Update)

DM(Data6,I7-0)=Dreg; {DAG1}
PM(Data6,I15-8)=Dreg; {DAG2}
Dreg=DM(Data6,I7-0); {DAG1}
Dreg=PM(Data6,I15-8); {DAG2}

Table 4-6. Pre-Modify Addressing, Modified By 32-Bit Data (No I Register Update)

Ureg=DM(Data32,I7-0) (LW); {DAG1}
Ureg=PM(Data32,I15-8) (LW); {DAG2}
DM(Data32,I7-0)=Ureg (LW); {DAG1}
PM(Data32,I15-8)=Ureg (LW); {DAG2}

Table 4-7. Update (Modify) I Register, Modified By M Register

Modify(I7-0,M7-0); {DAG1}
Modify(I15-8,M15-8); {DAG2}

Table 4-8. Update (Modify) I Register, Modified By 32-Bit Data

Modify(I7-0,Data32); {DAG1}
Modify(I15-8,Data32); {DAG2}

Table 4-9. Bit-Reverse and Update I Register, Modified By 32-Bit Data

Bitrev(I7-0,Data32); {DAG1}
Bitrev(I15-8,Data32); {DAG2}

5 MEMORY

The ADSP-21161 processor contains a large, dual-ported internal memory for single-cycle, simultaneous, independent accesses by the core processor and I/O processor. The dual-ported memory in combination with three separate on-chip buses allow two data transfers from the core and one transfer from the I/O processor in a single cycle. Using the IO bus, the I/O processor provides data transfers between internal memory and the processor's communication ports (link ports, serial ports, and external port) without hindering the processor core's access to memory. This chapter describes the processor's memory and how to use it. The processor provides access to external memory through the processor's external port. For information on connecting and timing accesses to external memory, see "External Memory Interface" on page 7-3.

The processor contains one megabit of on-chip SRAM, organized as two blocks of 0.5 Mbits. Each block can be configured for different combinations of code and data storage. All of the memory can be accessed as 16-bit, 32-bit, 48-bit, or 64-bit words. The memory can be configured in each block as a maximum of 16K words of 32-bit data, 8K words of 64-bit data, 32K words of 16-bit data, 10.67K words of 48-bit instructions (or 40-bit data), or combinations of different word sizes up to 0.5 Mbit. This gives a total for the complete internal memory: a maximum of 32K words of 32-bit data, 16K words of 64-bit data, 64K words of 16-bit data, and 21K words of 48-bit instructions (or 40-bit data). The processor features a 16-bit floating-point storage format that effectively doubles the amount of data that may be stored on-chip. A single instruction converts the format from 32-bit floating-point to 16-bit floating-point.

While each memory block can store combinations of code and data, accesses are most efficient when one block stores data using the DM bus, (typically block 1) for transfers, and the other block (typically block 0) stores instructions and data using the PM bus. Using the DM bus and PM bus with one dedicated to each memory block assures single-cycle execution with two data transfers. In this case, the instruction must be available in the cache.

Internal Memory

The ADSP-21161 has 2 MBits of internal memory space; 1 MBit is addressable. The 1 MBit of memory is divided into two 0.5 MBit blocks: Block 0 and Block 1. The additional 1MBit of the memory space is reserved on the ADSP-21161. Table 5-1 shows the maximum number of data or instruction words that can fit in each 0.5 MBit internal memory block.

Table 5-1. Words Per 0.5 MBit Internal Memory Block

Word Type	Bits Per Word	Maximum Number of Words Per 0.5 MBit block
Instruction	48-bits	10.67K Words
Long Word Data	64-bits	8K Words
Extended Precision Normal Word Data	40-bits	10.67K Words
Normal Word Data	32-bits	16K Words
Short Word Data	16-bits	32K Words

External Memory

While the processor's internal memory is divided into **blocks**, the processor's external memory spaces are divided into **banks**. The internal memory blocks and the external memory spaces may be addressed by either data address generator. External memory banks are fixed sizes that can be configured for various waitstate and access configurations. For more information, see "External Memory" on page 5-22.

There are 254 Mwords of external memory space that the processor can address. External memory connects to the processor's external port, which extends the processor's 24-bit address and 32-bit data buses off the processor. The processor can make 8, 16, 32, or 48-bit accesses to external memory for instructions and 8,16, or 32-bit accesses for data. Table 5-2 shows the access types and words for processor external memory accesses. The processor's DMA controller automatically packs external data into the appropriate word width during data transfer.

The external data bus can be expanded to 48-bits if the link ports are disabled and the corresponding full width instruction packing mode (`IPACK`) is enabled in the `SYSCON` register. Ensure that link ports are disabled when executing code from external 48-bit memory. For more information, see "Executing Instructions From External Memory" on page 5-101.

Table 5-2. Internal-to-External Memory Word Transfers[1]

Word Type	Transfer Type
Packed Instruction	32, 16, or 8- to 48-bit packing
Normal Word Data	32-bit word in 32-bit transfer
Short Word Data	Not supported

1 For external port word alignment, see Figure 7-1 on page 7-2.

The total addressable space for the fixed external memory bank sizes depends on whether SDRAM or Non-SDRAM (for example, SRAM, SBSRAM) is used. Each external memory bank for SDRAM can address 64M words. For Non-SDRAM memory, each bank can address up to

16M words. The remaining 48M words are reserved. These reserved addresses for non-SDRAM accesses are aliased to the first 16M spaces within the bank.

The total external memory available is given as follows:

3*(16M) + 14M = 62M (Non- SDRAM banks)

3*(64M) + 62M = 254M (SDRAM banks)

Banks 1, 2 and 3 have the same amount of external memory (16M for Non-SDRAM and 64M for SDRAM), while bank 0 is smaller (14M for Non-SDRAM and 62M for SDRAM).

The external memory address bus is 24-bits wide with four additional bank select $\overline{\texttt{MSx}}$ lines. For more information on the external memory, see the section "External Memory" on page 5-22.

Processor Architecture

Most microprocessors use a single address and single data bus for memory access. This type of memory architecture is called Von Neumann architecture. But, DSPs require greater data throughput than Von Neumann architecture provides, so many DSPs use memory architectures that have separate data and address buses for program and data storage. These two sets of buses let the processor retrieve a data word and an instruction simultaneously. This type of memory architecture is called Harvard architecture.

SHARC DSPs go a step further by using a Super Harvard architecture. This four bus architecture has two address buses and two data buses, but provides a single, unified address space for program and data storage. While the Data Memory (DM) bus only carries data, the Program Memory (PM) bus handles instructions and data, allowing dual-data accesses.

Processor core and I/O processor accesses to internal memory are completely independent and transparent to one another. Each block of memory can be accessed by the processor core and I/O processor in every cycle—no extra cycles are incurred if the processor core and the I/O processor access the same block.

A memory access conflict can occur when the processor core attempts two accesses to the same internal memory block in the same cycle. When this conflict, known as *block conflict* occurs, an extra cycle is incurred. The DM bus access completes first and the PM bus access completes in the following (extra) cycle.

During a single-cycle, dual-data access, the processor core uses the independent PM and DM buses to simultaneously access data from both memory blocks. Though dual-data accesses provide greater data throughput, it is important to note some limitations on how programs may use them. The limitations on single-cycle, dual-data accesses are:

- The two pieces of data must come from different memory blocks.

  If the core accesses two words from the same memory block over the same bus in a single instruction, an extra cycle is needed.

- The data access execution may not conflict with an instruction fetch operation. The PM data bus tries to fetch an instruction in every cycle. If a data fetch is also attempted over the PM bus, an extra cycle may be required depending on the cache.

  If the cache contains the conflicting instruction, the data access completes in a single-cycle and the sequencer uses the cached instruction. If the conflicting instruction is not in the cache, an extra cycle is needed to complete the data access and cache the conflicting instruction. For more information, see "Instruction Cache" on page 3-8.

For more information on how the buses access memory blocks, see "Internal Memory" on page 5-16.

Off-Chip Memory and Peripherals Interface

The ADSP-21161 processor's external port provides the processor's interface to off-chip memory and peripherals. Figure 5-9 on page 5-23 shows the external memory of ADSP-21161. The 62 Mword off-chip address space (254 Mword if all SDRAM) is included in the ADSP-21161's unified address space. The separate on-chip buses-for PM addresses, PM data, DM addresses, DM data, I/O addresses, and I/O data-are multiplexed at the external port to create an external system bus with a single 24-bit address bus and a single 32-bit data bus. Every access to external memory is based on an address that fetches a 32-bit word. When fetching instructions from external 32-bit memory, the program sequencer accesses two 32-bit data locations, four 16-bit locations or eight 8-bit locations. Unused link port lines can also be used as additional data lines `DATA15-0`, allowing single cycle execution of 48-bit instructions from external memory at up to 100 MHz.

The external port supports asynchronous, synchronous, and synchronous burst accesses. ZBT synchronous burst SRAM can be interfaced gluelessly. However, the zero bus turnaround feature is not supported by this processor; only the bursting protocol is supported. The ADSP-21161 processor also can interface gluelessly to SDRAM. Addressing of external memory devices is facilitated by on-chip decoding of high-order address lines to generate memory bank select signals. The ADSP-21161 processor provides programmable memory wait states and external memory acknowledge controls to allow interfacing to memory and peripherals with variable access, hold, and disable time requirements.

Efficient memory usage relies on how the program and data are arranged in memory and varies how the program accesses the data. For more information, see "Arranging Data in Memory" on page 5-100.

Buses

As shown in Figure 5-1 on page 5-9, the processor has three sets of internal buses connected to its dual-ported memory, the Program Memory (PM) bus, Data Memory (DM) bus, and I/O Processor (IO) bus. The PM bus and DM bus share one memory port and the IO bus connects to the other port. Memory accesses from the processor's core (computational units, data address generators, or program sequencer) use the PM or DM buses, while the I/O processor uses the IO bus for memory accesses.

The processor core's PM bus and DM bus and I/O processor's External Port (EP) bus can try to access multiprocessor memory space or external memory space in the same cycle. The processor has a two level arbitration system to handle this conflicting access. Arbitration stems from a priority convention and the state of the SYSCON register's EBPRx bits. When arbitrating between the processor core buses, the DM bus always has priority over the PM bus. Arbitration between the winning core bus and I/O processor EP bus depends on the priority set with the EBPRx bits. For more information on setting this priority, see "External Bus Priority" on page 5-39.

Internal Address and Data Buses

Figure 5-1 shows that the PM buses, DM buses, and I/O processor have access to the external bus (pins DATA47-16, ADDR23-0) through the processor's external port. The external port provides access to system (off-processor) memory and peripherals. This port also lets the processor access the IOP register space of other DSPs when connected in a multiprocessing system.

Almost without exception, the processor's three buses can access all memory spaces, supporting all data sizes. There are three restrictions on the access of buses to memory. The limitations on the PM, DM, and IO buses are as follows:

- The PM, DM, and IO buses make Normal Word addressing accesses to multiprocessor or external memory. These buses can make 40/48 bit data transfers by configuring the link data pins as additional data pins for external accesses. For more information, see "Multiprocessor Memory" on page 5-19.
- The IO bus may not access the I/O processor's memory mapped registers. For more information, see "I/O Processor" on page 6-1.
- The IO bus may not use short word addressing for DMA operation.

Addresses for the PM and DM buses come from the processor's program sequencer and Data Address Generators (DAGs). The program sequencer generates 24-bit program memory addresses while DAGs supply 32-bit addresses for locations throughout the processor's memory spaces. The DAGs supply addresses for data reads and writes on both the PM and DM address buses, while the program sequencer uses only the PM address bus for sequencing execution.

Each DAG is associated with a particular data bus. DAG1 supplies addresses over the DM bus and DAG2 supplies addresses over the PM bus. For more information on address generation, see "Program Sequencer" on page 3-1 or "Data Address Generator" on page 4-1.

Because the processor's internal memory is arranged in four 16-bit wide by 8K high columns, memory is addressable in widths that are multiples of columns up to 64 bits: 1 column = 16-bit words, 2 columns = 32-bit words, 3 columns = 48- or 40-bit words, and 4 columns = 64-bit words. For more information on the how the processor works with memory words, see "Memory Organization and Word Size" on page 5-25.

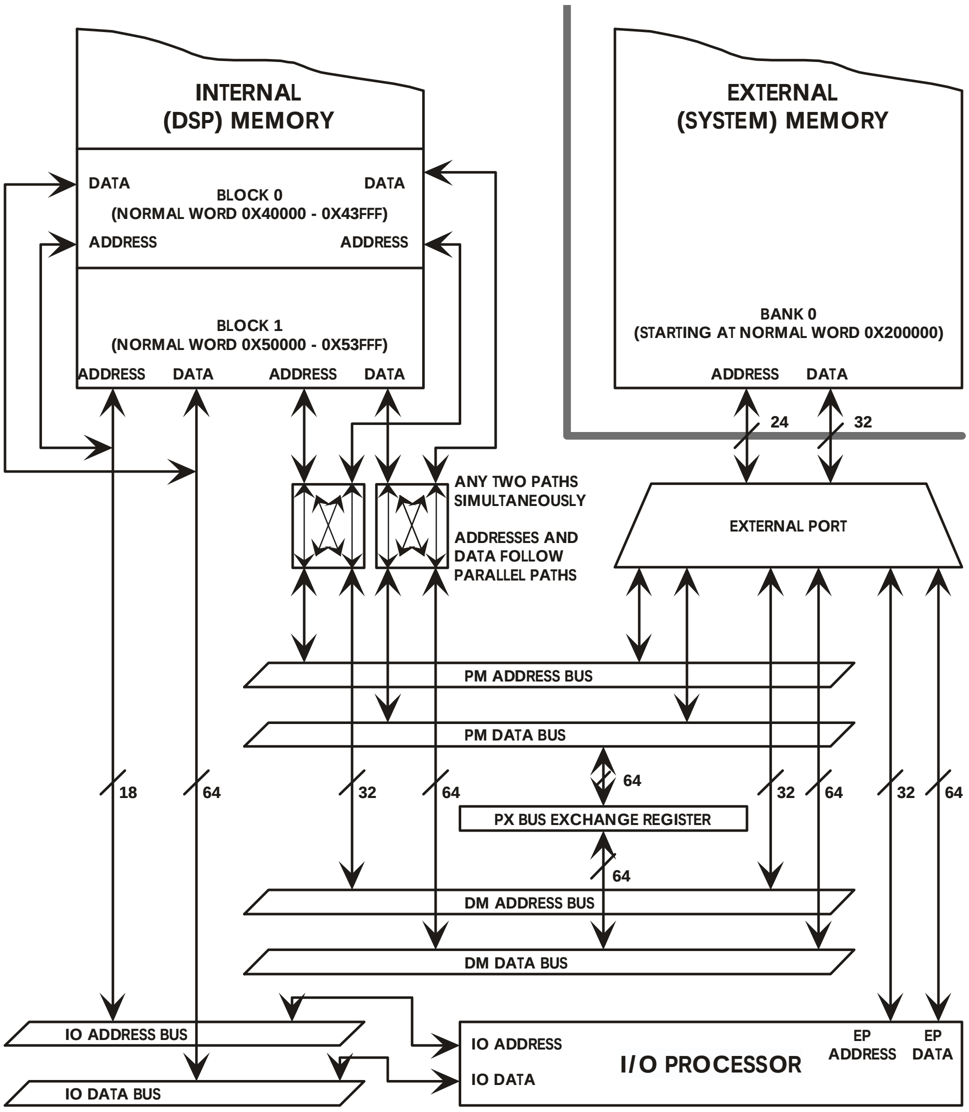


Figure 5-1. ADSP-21161 Memory and Internal Buses Block Diagram

The PM and DM data buses are 64 bits wide. Both data buses can handle long word (64-bit), normal word (32-bit), extended-precision normal word (40-bit), and short word (16-bit) data, but only the PM data bus carries Instruction words (48-bit).

Internal Data Bus Exchange

The data buses let programs transfer the contents of any register in the processor to any other register or to any internal memory location in a single cycle. As shown in Figure 5-1 on page 5-2, the PM Bus Exchange (`PX`) register permits data to flow between the PM and DM data buses. The `PX` register can work as one 64-bit register or as two 32-bit registers (`PX1` and `PX2`). The alignment of `PX1` and `PX2` within `PX` appears in Figure 5-2.

Instruction Examples

```
PX = DM(0x80000)(LW);
PX = DM(0x40000);
```

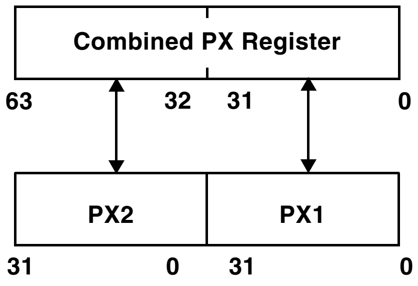


Figure 5-2. PM Bus Exchange (PX, PX1, and PX2) Registers

The `PX1`, `PX2`, and the combined `PX` register are Universal registers (UREG) that are accessible for register-to-register or memory-to-register transfers.

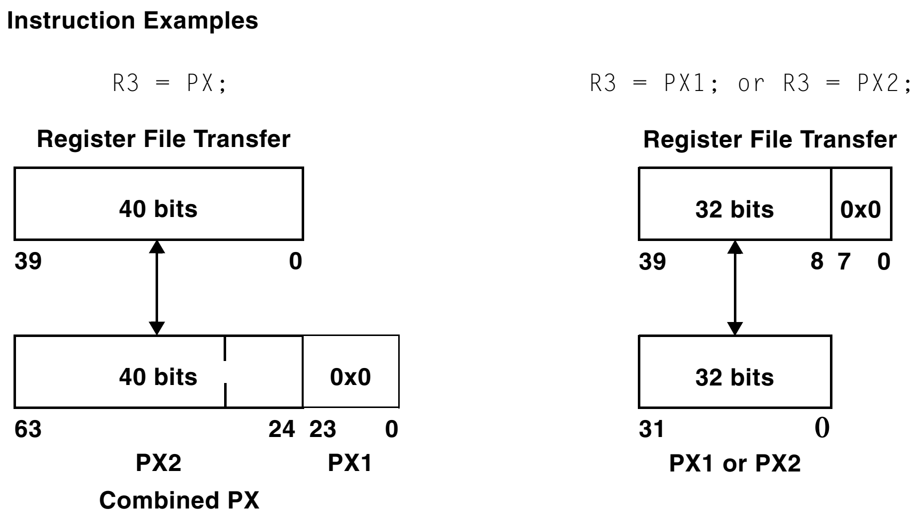


Figure 5-3. PX, PX1, and PX2 Register-to-Register Transfers

`PX` *register-to-register* transfers with data registers are either 40-bit transfers for the combined *PX* or 32-bit transfers for *PX1* or *PX2*. Figure 5-3 shows the bit alignment and gives an example of instructions for register-to-register transfers.

Figure 5-3 shows that during a transfer between PX1 or PX2 and a data register (DREG), the bus transfers the upper 32 bits of the register file and zero fills the eight LSBs.

During a transfer between the combined `PX` register and a register file, the bus transfers the upper 40 bits of `PX` and zero fills the lower 24 bits.

PX *register-to- internal memory* transfers over the DM or PM data bus are either 48-bit for the combined PX or 32-bit transfers (on bits 31-0 of the bus) for PX1 or PX2. Figure 5-4 shows these transfers.

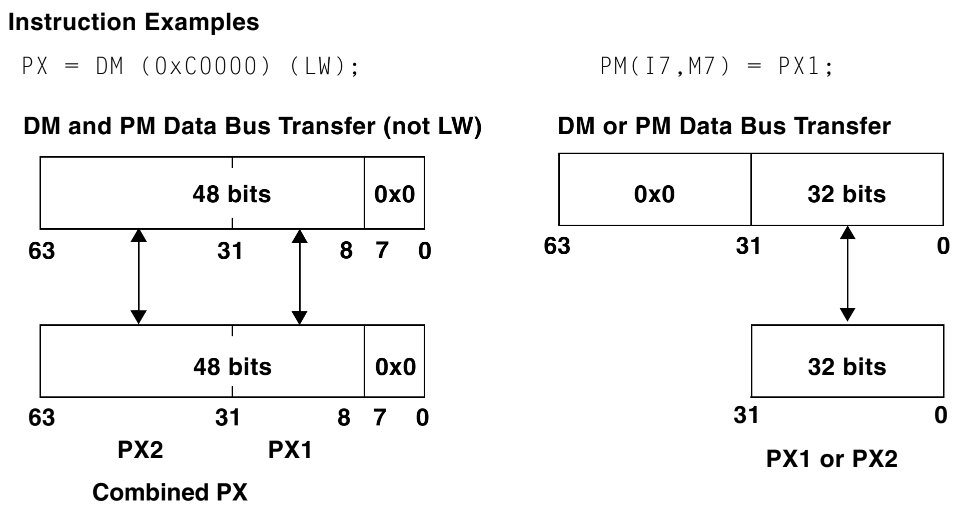


Figure 5-4. PX, PX1, PX2 Register-to-Memory Transfers on DM (LW) or PM (LW) Data Bus

Figure 5-4 shows that during a transfer between PX1 or PX2 and internal memory, the bus transfers the lower 32 bits of the register.

During a transfer between the combined PX register and internal memory, the bus transfers the upper 48 bits of PX and zero fills the lower 8 bits.

The status of the memory block's Internal Memory Data Width (IMDWx) setting does not effect this default transfer size for PX to internal memory.

Figure 5-5 shows a PX *register-to-external memory* transfer. The PX register transfers the upper 32 bits of the PM data bus into PX1 and the lower 16 bits to PX2, zero filling the remaining 16 bits.

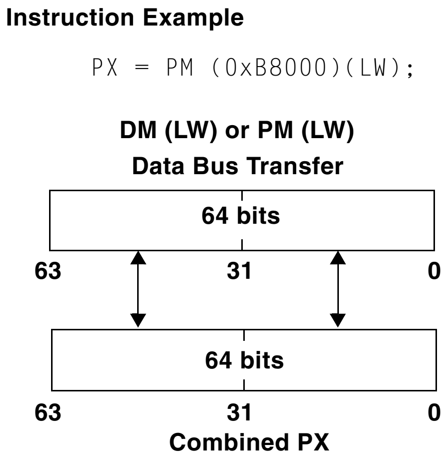


Figure 5-5. PX Register-to-External Memory Transfers

Since there are 32 DATA pins on the ADSP-21161 processor, 40/48 bit data transfers using register to register transfers are not directly supported. To accomplish 40/48 bit data transfers with the PX register, you must configure the link data pins as additional data pins for external accesses. Full width instruction mode (IPACK) must be enabled in the SYSCON register. The 16 link data pins are configured as DATA pins and the processor fetches the upper 32 bits of instruction on 32 DATA pins and lower 16 bits of instruction on the link data pins.

To transfer both 48-bit instructions and 40-bit double precision data to a register, you must swap the PX1 and PX2 registers. See the following code examples:

Example 1: To transfer 48-bits from external memory to internal memory, use the following code:

```
PX = DM(EXT_MEMORY_LOC);
R0 = PX1;
PX1 = PX2;
PX2 = R0;
DM(INT_MEMORY_LOC) = PX;
```

Example 2: To transfer a 40-bit data from external memory to a register, use the following code:

```
PX = DM(EXT_MEMORY_LOC);
R0 = PX1;
PX1 = PX2;
PX2 = R0;
R1 = PX;
```

All transfers between the PX register and the I/O processor LBUFx registers are 48-bit transfers (most significant 48-bits of PX).

All transfers between the PX register (or any other internal register/memory) and any I/O processor register (other than the EPBx or LBUFx) are 32-bit transfers (least significant 32-bits of PX).

All transfers between the PX register and data registers (R0-R15 or S0-S15) are 40-bit transfers. The most significant 40-bits are transferred as shown in Figure 5-3 on page 5-11.

Figure 5-6 shows the transfer size between PX and internal memory over the PM or DM data bus when using the long word (LW) option.

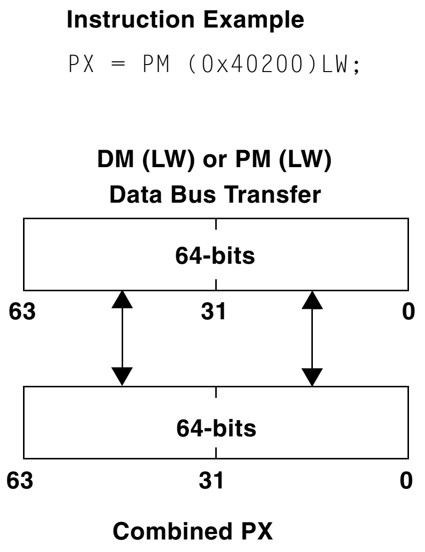


Figure 5-6. PX Register-to-Memory Transfers on PM Data Bus

The `LW` notation in Figure 5-6 draws attention to an important feature of `PX` *register-to-internal memory* transfers over the PM or DM data bus for the combined `PX` register. `PX` transfers to memory are 48-bit (3-column) transfers on bits 0-31 of the PM or DM data bus, unless forced to be 64-bit (4-column) transfers with the `LW` (Long Word) mnemonic.

There is no implicit move when the combined `PX` register is used in SIMD mode. For example, in SIMD mode, the following moves could occur:

```
PX1 = R0;  /* R0 32-bit explicit move to PX1,
            and R1 32-bit implicit move to PX2 */
PX = R0;  /* R0 40-bit explicit move to PX,
            but no implicit move for R1 */
```

ADSP-21161 Memory Map

The ADSP-21161's memory map appears in Figure 5-7 and has three memory spaces: internal memory space, multiprocessor memory space, and external memory space. These spaces have the following definitions:

- **Internal memory space.** This space ranges from address 0x0000 0000 through 0x0005 3FFF (Normal word). Internal memory space refers to the processor's on-chip SRAM and memory mapped registers.
- **Multiprocessor memory space.** This space ranges from address 0x0010 0000 through 0x001F FFFF (Normal word). Multiprocessor memory space refers to the internal memory space of other DSPs that are connected in a multiprocessor system.
- **External memory space.** This space ranges from address 0x0200 0000 to 0x0CFF FFFF for Non-SDRAM and 0x0020 0000 through 0x0FFF FFFF (Normal word) for SDRAM. External memory space refers to the off-chip memory or memory mapped peripherals that are attached to the processor's external address (ADDR23-0) and data (DATA47-16) buses.

Internal Memory

The ADSP-21161's internal memory space appears in Figure 5-7. This memory space has four address regions.

- **I/O processor memory mapped registers.** This region ranges from address 0x0000 0000 through 0x0000 01FF (Normal Word).
- **Reserved memory.** This region ranges from address 0x0000 0200 through 0x0001 FFFF. These addresses are not accessible.

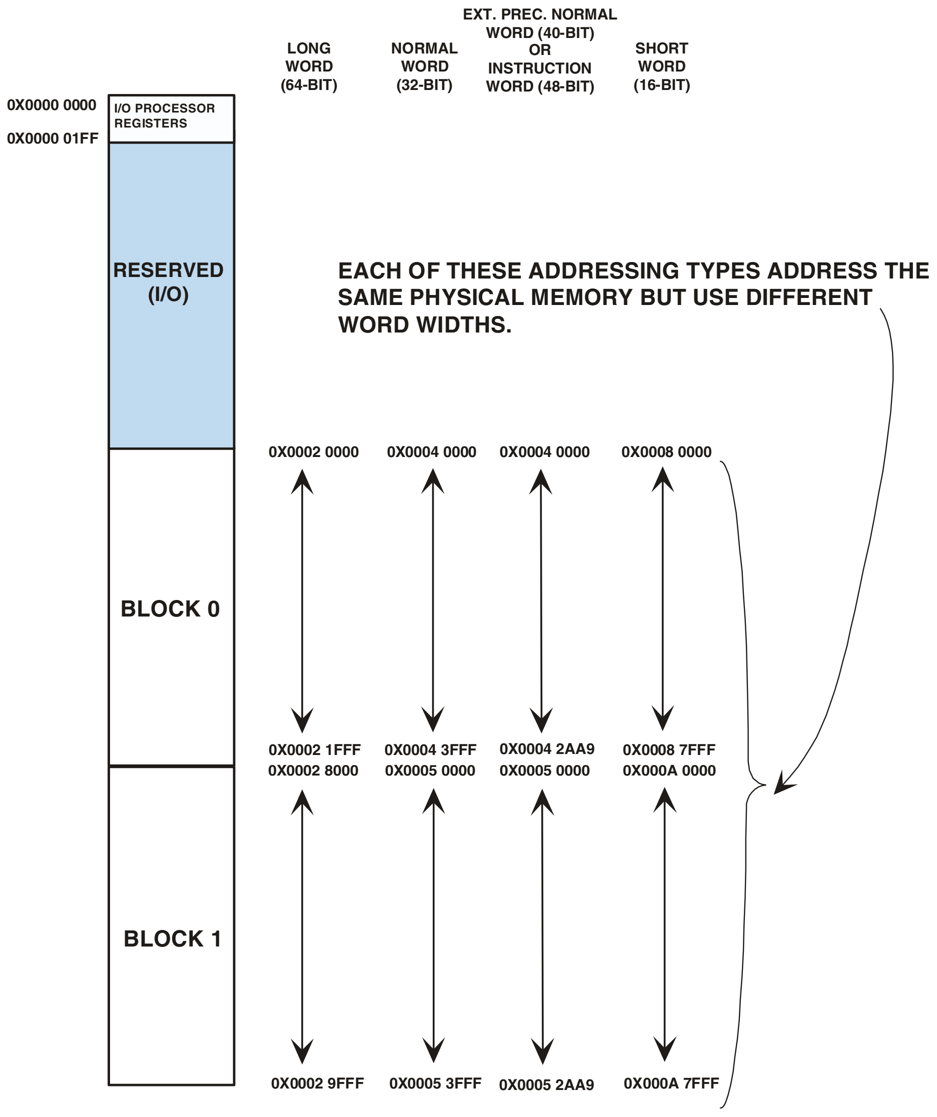


Figure 5-7. ADSP-21161 Internal Memory Space

- **Block 0 memory.** This region, typically PM, ranges from address 0x0004 0000 through 0x0004 3FFF (Normal Word). DAG2 generates PM data addresses.
- **Block 1 memory.** This region, typically DM, ranges from address 0x0005 0000 through 0x0005 3FFF (Normal Word). DAG1 generates DM data addresses.

The I/O processor's memory-mapped registers control the system configuration of the processor and I/O operations. For more information, see "I/O Processor" on page 6-1. These registers occupy consecutive 32-bit locations in this region.

If a program uses long word addressing (forced with the LW mnemonic) to accesses this region, the access is only to the addressed 32-bit register, rather than accessing two adjacent I/O processor registers. The register contents are transferred on bits 31-0 of the data bus. There are a couple of exceptions to this one-at-a-time I/O processor register access rule:

- Long word accesses to external port buffer (EPBx) or link port buffer (LBUFx) locations using the PX register access two adjacent 32-bit I/O registers.
- Long word accesses to the external port data buffer locations (EPBx) in SIMD mode access two adjacent 32-bit I/O registers.

As shown in Figure 5-7 on page 5-17, the processor can address memory in the Block 0 and Block 1 using long word, normal word, or short word addressing. The processor interprets the addressing mode from the address range for the access. Though there are multiple addressing modes for each memory region, these different modes are addressing the same physical memory. For example, the long word address 0x0002 0000 corresponds to the same locations as normal word addresses 0x0004 0000 and 0x0004 0001. This also corresponds to the same locations as short word addresses 0x0008 0000, 0x0008 0001, 0x0008 0002, and 0x0008 0003.

Figure 5-7 on page 5-17 also shows that there are gaps in the processor's memory map when using normal word addressing for 48-bit (instruction word) or 40-bit (extended-precision normal word) accesses. These gaps of missing addresses stem from the arrangement of this 3-column data in memory. For more information, see "Memory Organization and Word Size" on page 5-25.

Multiprocessor Memory

The ADSP-21161's multiprocessor memory space appears in Figure 5-8. This memory space has seven address regions that correspond to the IOP register space of the DSPs in a multiprocessing system. Each of the processors in such a system has a processor ID, which is set with the processor's `ID2-0` pins. The address regions by processor ID are:

- **Internal memory with ID=001.** This region ranges from address 0x0010 0000 through 0x0011 FFFF.
- **Internal memory with ID=010.** This region ranges from address 0x0012 0000 through 0x0013 FFFF.
- **Internal memory with ID=011.** This region ranges from address 0x0014 0000 through 0x0015 FFFF.
- **Internal memory with ID=100.** This region ranges from address 0x0016 0000 through 0x0017 FFFF.
- **Internal memory with ID=101.** This region ranges from address 0x0018 0000 through 0x0019 FFFF.
- **Internal memory with ID=110.** This region ranges from address 0x001A 0000 through 0x001B FFFF.

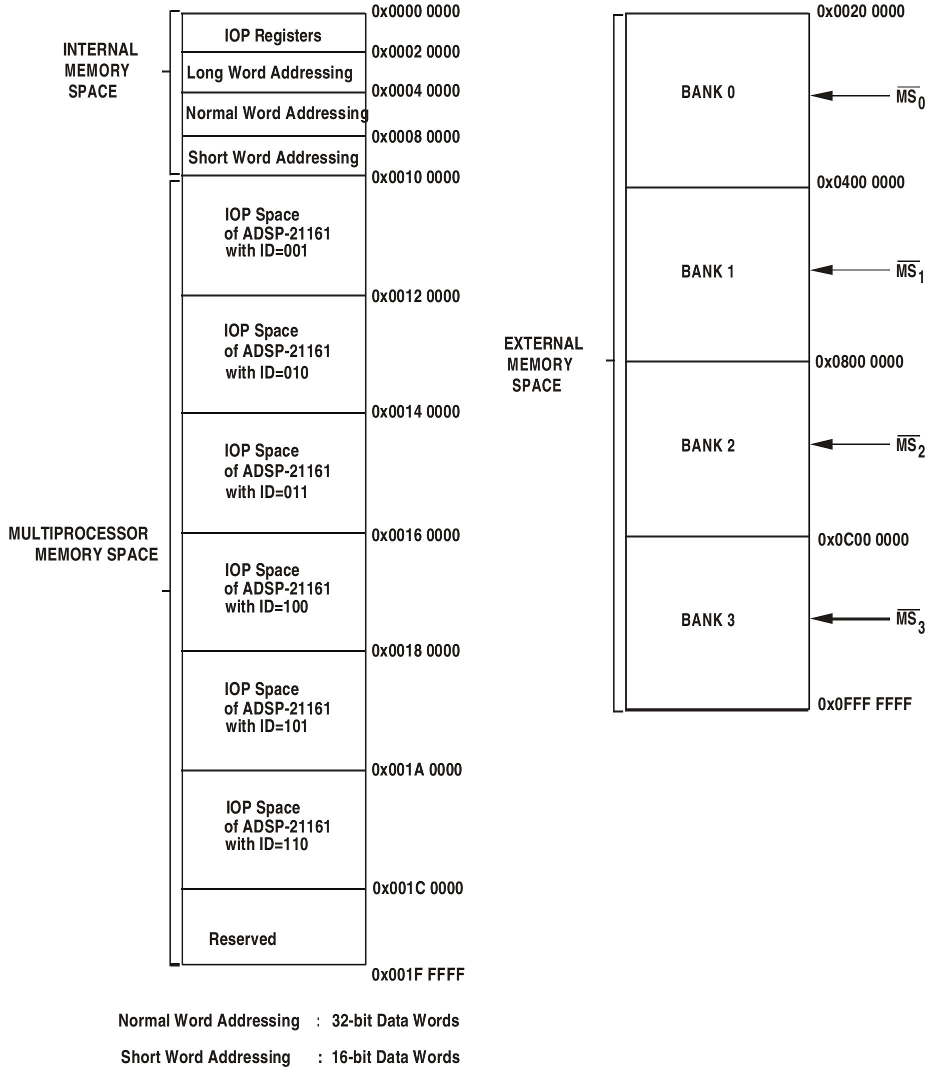


Figure 5-8. Multiprocessor Memory Map

It is important to note that programs may only use normal word addressing in multiprocessor memory space. Long or short word writes may corrupt valid data, and long or short word reads return invalid data.

The address range of the access determines which processor's internal memory is the multiprocessor memory access source or destination. Instead of using its own IOP register address range, a processor can access its IOP space through the corresponding address range in multiprocessor memory space. In this case, the processor reads or writes to its own IOP registers and does not make an access on the external system bus. Note that such self-accesses through multiprocessor memory space may only be accomplished with processor-core-generated addresses, not I/O processor-generated addresses.

For more information on memory accesses in multiprocessor systems, see "External Port" on page 7-1.

Table 5-3 shows how the processor decodes and routes memory addresses over the DM and PM buses.

Table 5-3. Address Decoding For Memory Accesses

Address Bits[1]	Field	Description
ADDR31-28	NA	Reserved
ADDR27-24	V	Virtual address. Drives MS3-0 as follows: 00 = Depends on E, S and M bits; address corresponds to local processor's internal or external memory bank 0 01 = External memory bank 1, local processor 10 = External memory bank 2, local processor 11 = External memory bank 3, local processor
ADDR23-21	E[2]	Memory address. 00000[00] = Address in local or remote processor's internal memory space. xxxxx[xx] = Based on V bits; address in one of local processor's four external memory banks.

Table 5-3. Address Decoding For Memory Accesses (Cont'd)

Address Bits[1]	Field	Description
ADDR20	M[2]	Multiprocessor memory. If this bit is 1, the address is in multiprocessor memory space. If this bit is 0, the address is in IOP register space.
ADDR19-17	S[2]	IOP MMS accesses. Depends on M bit. When bit 20 is set to 1, bits 19:17 indicate the following: 000 = Address is in IOP space of processor with ID1 001 = Address is in IOP space of processor with ID2 010 = Address is in IOP space of processor with ID3 100 = Address is in IOP space of processor with ID4 011 = Address is in IOP space of processor with ID5 101 = Address is in IOP space of processor with ID6
ADDR16-0	NA	Internal memory and IOP register space.

1 Setup and hold times for these address lines are specified in the processor Data Sheet.
2 For a description of these address fields, see "Multiprocessor Memory" on page 5-19.

External Memory

The ADSP-21161's external memory space appears in Figure 5-9. The processor accesses external memory space through the external port, which multiplexes the processor core's PM and DM buses and the I/O processor's EP bus. To address this space, the processor's DAG1, DAG2, and I/O processor generate 32-bit addresses over the DM, PM, and EP address buses, allowing the processor to access to the complete 254 Mword memory map.

ⓘ The program sequencer only generates 24-bit addresses over the PM bus, limiting sequencing to the low 62 Mwords (for SDRAM) or low 14 Mwords (for SRAM) of the memory map.

The external memory space has four banks (bank 0-3). The processor controls access to the banked regions with memory select lines ($\overline{\texttt{MS3-0}}$) in addition to the memory address. Each region of external memory may be configured for access modes and waitstates. For more information on configuring external memory banks, see "Setting Data Access Modes" on page 5-32. For more information on accessing external memory, see "External Port" on page 7-1.

The external memory space can also accommodate an optional boot memory EPROM or FLASH. For more information, see "Using Boot Memory" on page 5-35.

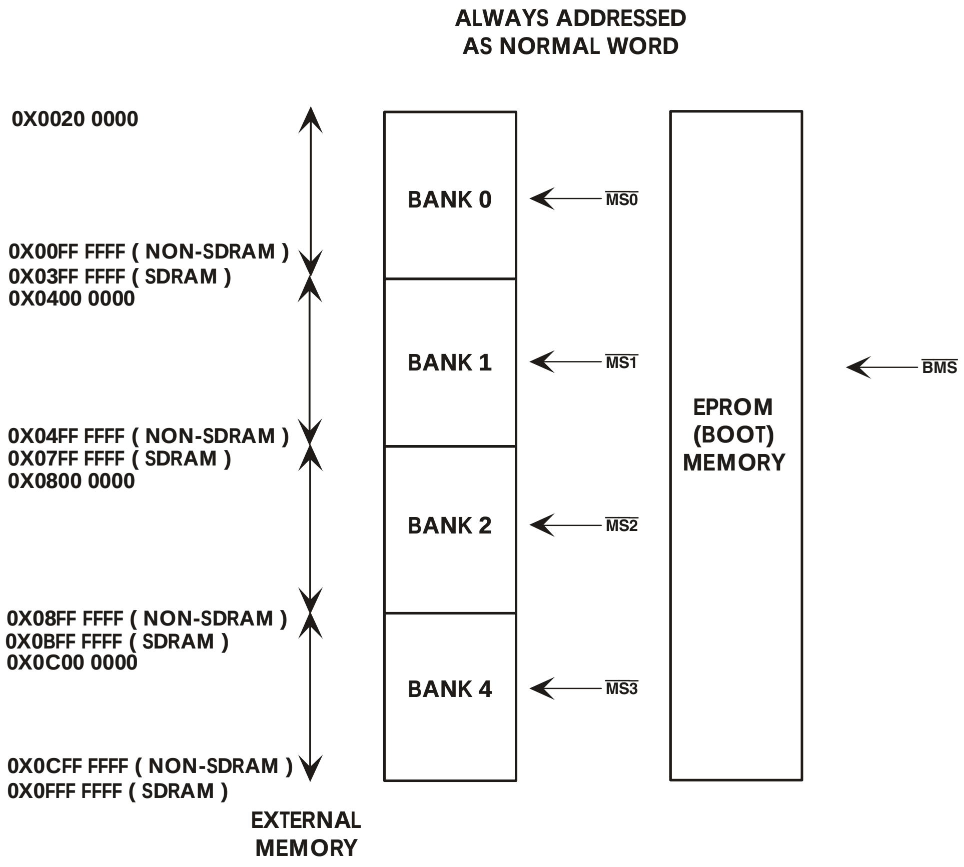


Figure 5-9. ADSP-21161 External Memory Space

Shadow Write FIFO

Because the processor's internal memory operates at high speeds, writes to the memory do not go directly into the memory array, but rather to a two-deep FIFO called the shadow write FIFO. This FIFO uses a non-read cycle (either a write cycle, or a cycle in which there is no access of internal memory) to load data from the FIFO into internal memory. When an internal memory write cycle occurs, the FIFO loads any data from a previous write into memory and accepts new data. FIFO operation is normally transparent, but there is one case in which programs need to intervene in the operation of the shadow write FIFO: mixing 48-bit and 32-bit word accesses to the same locations in memory.

The shadow FIFO cannot differentiate between the mapping of 48-bit words and the mapping of 32-bit words. Examples of these mappings appear in Figure 5-10 through Figure 5-13. If a program writes a 48-bit word to memory and then tries to read the data with a 16-, 32- or 64-bit word access or writes a 16-, 32- or 64-bit word to memory and tries to read the data with a 48-bit access, the shadow FIFO does not intercept the read. It returns incorrect data.

If a program must mix 48-bit or 40-bit accesses and 16-, 32-, or 64-bit accesses to the same locations, the program must ensure that the FIFO is flushed before attempting to read the data. The program flushes the FIFO by performing two dummy writes or executing two instructions that do not access the internal memory. These operations force the FIFO to automatically use the non-access cycles to push the write data.

Memory Organization and Word Size

The processor's internal memory is organized as four 16-bit wide by 8K high columns. These columns of memory are addressable as a variety of word sizes:

- 64-bit long word data (4-columns)
- 48-bit instruction words or 40-bit extended-precision normal word data (3-columns)
- 32-bit normal word data (2-columns)
- 16-bit short word data (1-column)

Extended precision normal word data is only accessible if the `IMDWx` bit is set in the `SYSCON` register. It is left-justified within a three column location, using bits 47-8 of the location.

Placing 32-Bit Words and 48-Bit Words

When the processor core or I/O processor addresses memory, the word width of the access determines which columns within the memory are accessed. For instruction words (48-bit) or extended-precision normal word data (40-bit), the word width is 48 bits, and the processor accesses from the memory's 16-bit columns in groups of three. Because these sets of three column accesses are packed into a four column matrix, there are four rotations of the columns for storing 40/48-bit data. The 3-column word rotations within the 4-column matrix appear in Figure 5-10.

For long word (64-bit), normal word (32-bit), and short word (16-bit) memory accesses, The processor selects from fixed columns in memory. No rotations of words within columns occur for these data types.

Figure 5-7 on page 5-17 shows the memory ranges for each data size in the processor's internal memory.

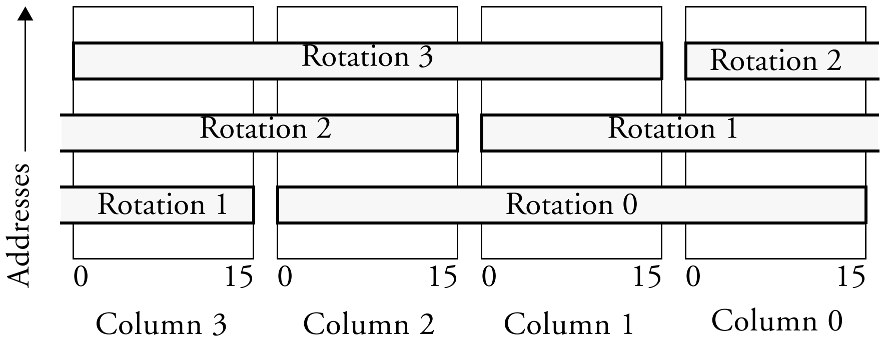


Figure 5-10. 48-bit Word Rotations

Mixing 32-Bit and 48-Bit Words

The processor's memory organization lets programs freely place memory words of all sizes (see "Memory Organization and Word Size" on page 5-25) with few restrictions (see "Restrictions on Mixing 32-Bit and 48-Bit Words" on page 5-28). This memory organization also lets programs mix (place in adjacent addresses) words of all sizes. This section discusses how to mix odd (3-column) and even (4-column) data words in the processor's memory.

Transition boundaries between 48-bit (3-column) data and any other data size, can only occur at any 64-bit address boundary within either internal memory block. Depending on the ending address of the 48-bit words, there are zero, one, or two empty locations at the transition between the 48-bit (3-column) words and the 64-bit (4-column) words. These empty locations result from the column rotation for storing 48-bit words. The three possible transition arrangements appear in Figure 5-11, Figure 5-12, and Figure 5-13.

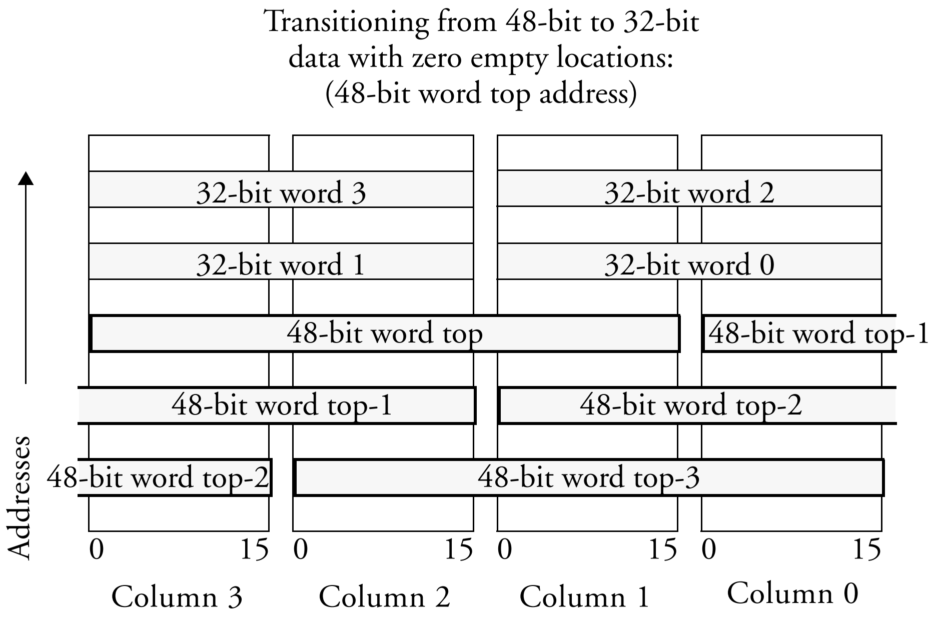


Figure 5-11. Mixed Instructions and Data With No Unused Locations

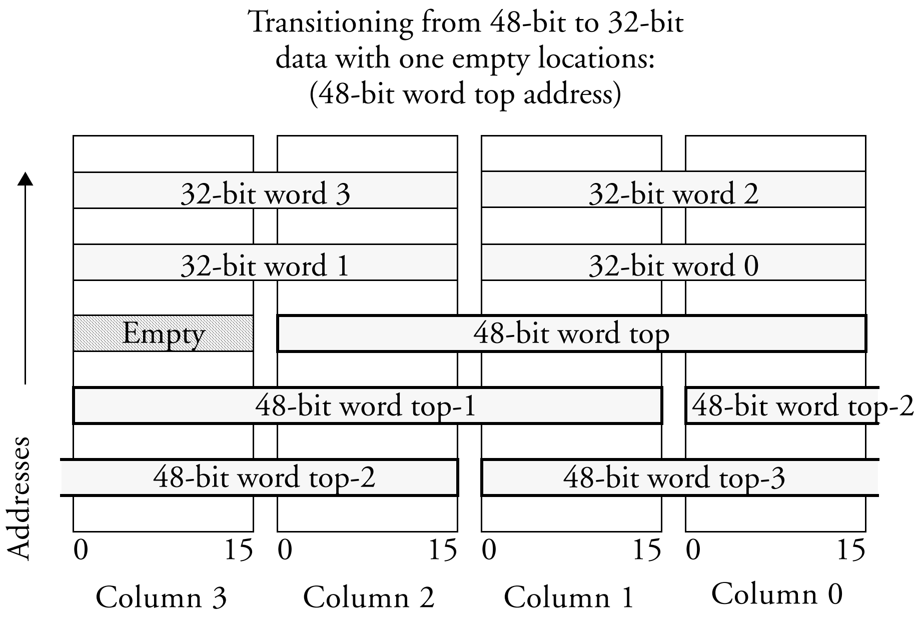


Figure 5-12. Mixed Instructions and Data With One Unused Location

Restrictions on Mixing 32-Bit and 48-Bit Words

There are some restrictions that stem from the memory column rotations for 3-column data (48- or 40-bit words) and relate to the way that 3-column data can mix with 4-column data (32-bit words) in memory. These restrictions apply to mixing 48- and 32-bit words, because the processor uses a normal word address to access both of these types of data even though 48-bit data maps onto 3-columns of memory and 32-bit data maps onto 2-columns of memory.

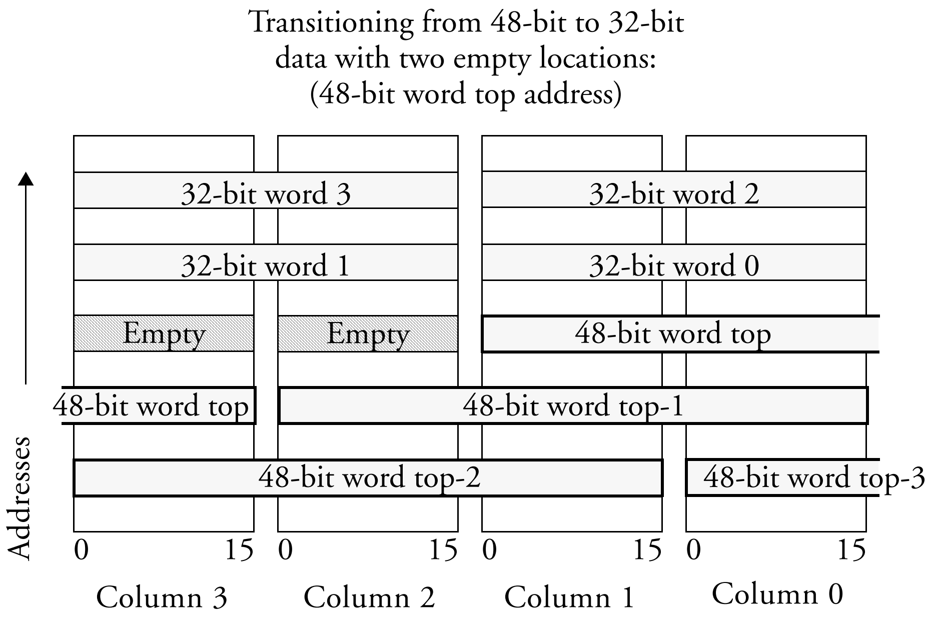


Figure 5-13. Mixed Instructions and Data With One Unused Location

When a system has a range of 3-column (48-bit) words followed by a range of 2-column (32-bit) words, there is often a gap of empty 16-bit locations between the two address ranges. The size of the address gap varies with the ending address of the range of 48-bit words. Because the addresses within the gap alias to both 48- and 32-bit words, a 48-bit write into the gap corrupts 32-bit locations, and a 32-bit write into the gap corrupts 48-bit locations. The locations within the gap are only accessible with short word (16-bit) accesses.

Calculating the starting address for 4-column data that minimizes the gap after 3-column data is a useful calculation for programs that are mixing 3- and 4-column data. Given the last address of the 3-column (48-bit) data, the starting address of the 32-bit range that most efficiently uses memory can be determined by the equation shown in Listing 5-1:

Listing 5-1. Starting Address

$$\mathbf{m} = \mathbf{B} + 2\,[(\mathbf{n}\ \mathrm{MOD}\ 10{,}922) - \mathrm{TRUNC}((\mathbf{n}\ \mathrm{MOD}\ 10{,}922)\ /\ 4)]$$

where:

- **n** is the number of contiguous 48-bit words allocated in the internal memory block (**n** < 21845)
- **B** is the base normal word address of the internal memory block; if {0 < **n** < **10,922**} then **B** = 0x40000 (Block 0) else **B** = 0x50000 (Block 1)
- **m** is the first 32-bit normal word address to use after the end of 48-bit words

Example 1: Calculating a starting address for a 32-bit addresses

The last valid address is 0x42694. The number of 48-bit words (n) is given as follows:

n = 0x42694 - 0x40000+1= 0x2695

When you convert 0x2695 to decimal representation, the result is 9877.

The base (B) Normal word address of the internal memory block is 0x40000 since the condition: 0 < 10922 is TRUE.

The first 32-bit Normal word address to use after the end of the 48-bit words is given by:

m = 0x40000 + 2 [(9877 MOD 10922)- TRUNC (9877 MOD 10922)/4]

m = 0x40000 + 14816decimal

Convert to a hexadecimal address:

$14816_{decimal}$ = 0x39E0

m = 0x40000 + 0x39E0 = 0x439E0

The first valid starting 32-bit address is 0x439E0. The starting address must begin on an even address.

48-Bit Word Allocation

Another useful calculation for programs that are mixing 3- and 4-column data is to calculate the amount of 3-column data that minimizes the gap before starting 4-column data. Given the starting address of the 4-column (32-bit) data, the number of 48-bit words to allocate that most efficiently uses memory can be determined as shown in Listing 5-2:

Listing 5-2. 48-bit Word Allocation

m = TRUNC{(4/3)[(1/2)(m-b)]} + W

where

- **m** is the first 32-bit normal word address after the end of 48-bit words (0x3FFFF < **m** < 0x44000 for block 1, 0x4FFFF < **m** < 0x54000 for block 2)
- **B** is the base normal word address of the internal memory block; if {0x3FFFF < **m** < 0x50000} then **B** = 0x40000 else **B** = 0x50000 (Block 1)

- W is the number of offset words; if {B = 0x50000} then W = 43,690 else W = 0
- **n** is the number of contiguous 48-bit words the system should allocate in the internal memory block

Setting Data Access Modes

The SYSCON, MODE1, MODE2, and WAIT registers control the operating mode of the processor's memory. Table A-18 on page A-60 lists all the bits in SYSCON, Table A-2 on page A-3 lists all the bits in MODE1, Table A-2 on page A-3 lists all the bits in MODE2, and Table A-20 on page A-66 lists all the bits in WAIT.

SYSCON Register Control Bits

Figure 5-14 shows the control bits for the SYSCON register. The following bits in the SYSCON register control memory access modes:

- **Boot Select Override.** SYSCON Bit 1 (BSO). This bit overrides normal usage of $\overline{\text{MSx}}$ chip select lines in favor of the $\overline{\text{BMS}}$ select line for access to boot memory instead of external memory (if 1) or allows normal access to external memory with the $\overline{\text{MSx}}$ chip select lines (if 0).
- **Internal Interrupt Vector Table.** SYSCON Bit 2 (IIVT). This bit forces placement of the interrupt vector table at address 0x0004 0000 regardless of booting mode (if 1) or allows placement of the interrupt vector table as selected by the booting mode (if 0).
- **Internal Memory Block Data Width.** SYSCON Bit 9 (IMDW0) and Bit 10 (IMDW1). These bits select the normal word data access size for internal memory Block 0 and Block1. A block's normal word access size is fixed as 32-bit (2-column, IMDWx=0) or 40-bit (3-column, IMDWx=1).

- **Instruction Packing Mode.** `SYSCON` Bits 30 and 31 (`IPACK1` and `IPACK0`). These bits select the external packing instruction execution as 8- to 48-bit, 16- to 48-bit, 32- to 48-bit or no pack mode.
- **External Bus Priority.** `SYSCON` Bits 18-17 (`EBPRx`). This bit field selects the priority for the I/O processor's EP bus when both the core and the IOP attempt to access external memory.

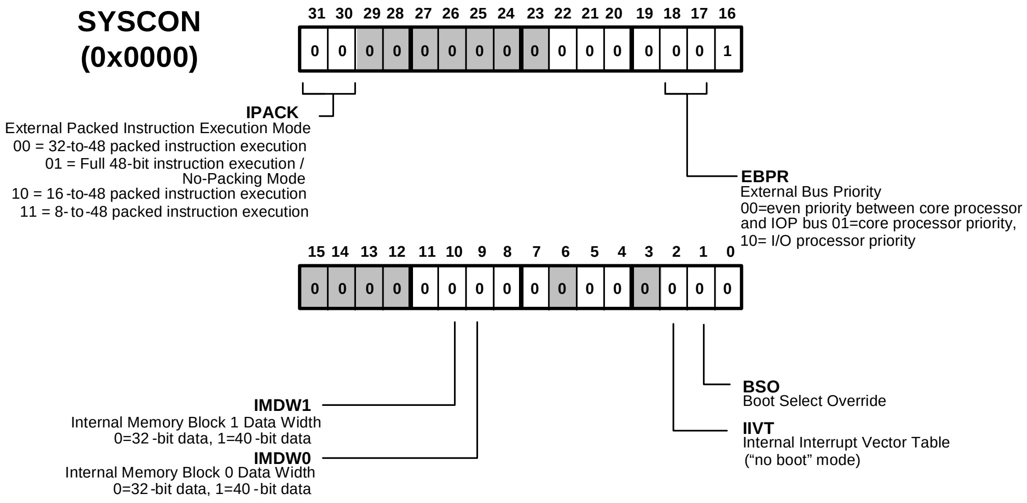

SYSCON
(0x0000)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
IPACK
External Packed Instruction Execution Mode
00 = 32-to-48 packed instruction execution
01 = Full 48-bit instruction execution / No-Packing Mode
10 = 16-to-48 packed instruction execution
11 = 8-to-48 packed instruction execution
EBPR
External Bus Priority
00=even priority between core processor and IOP bus 01=core processor priority, 10= I/O processor priority
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
IMDW1
Internal Memory Block 1 Data Width
0=32-bit data, 1=40-bit data
IMDW0
Internal Memory Block 0 Data Width
0=32-bit data, 1=40-bit data
BSO
Boot Select Override
IIVT
Internal Interrupt Vector Table
("no boot" mode)

Figure 5-14. Syscon Register – Control Bits Only

Mode 1 Register Control Bits

The following bits in the MODE1 register control memory access modes:

- **Secondary Processor Element (PEy).** MODE1 Bit 21 (PEYEN) enables computations in PEy—SIMD mode—(if 1) or disables PEy—SISD mode—(if 0).
- **Broadcast Register Loads.** Mode1 **Bit 22** (BDCST9) **and Bit 23** (BDCST1) **enable broadcast register loads for memory transfers indexed with I1 (if** BDCST1 **= 1) or indexed with I9 (if** BDCST9 **=1).**

Mode 2 Register Control Bits

The following bits in the MODE2 register control memory access modes:

- **Illegal IOP Register Access Enable.** MODE2 Bit 20 (IIRAE) enables detection of I/O processor register access (if 1) or disables detection (if 0).
- **Unaligned 64-bit Memory Access Enable.** MODE2 Bit 21 (U64MAE) enables detection of uneven address memory access (if 1) or disables detection (if 0).

Wait Register Control Bits

The following bits in the WAIT register control memory access modes:

- **External Bank X Access Mode.** WAIT Bits 1-0 (EB0AM), Bits 6-5 (EB1AM), Bits 11-10 (EB2AM), Bits 16-15 (EB3AM), and Bits 21-20 (RBAM). These bit fields select the access modes (synchronous, asynchronous, SDRAM, SBSRAM) for the external memory banks.

- **External Bank X Waitstates.** WAIT Bits 4-2 (EB0WS), Bits 9-7 (EB1WS), Bits 14-12 (EB2WS), Bits 19-17 (EB3WS) and Bits 24-22 (RBWS). These bit fields independently select the number of waitstates for each of the external memory banks. After reset, the default number of waitstates is seven.

Using Boot Memory

As shown in Figure 5-9 on page 5-23, the processor supports an external boot EPROM mapped to external memory and selected with the $\overline{\text{BMS}}$ pin. The boot EPROM provides one of the methods for automatically loading a program in to the internal memory of the processor after power-up or after a software reset. This process is called booting. For information on boot options and the booting process, see the following sections:

- "Bootloading Through The External Port" on page 6-70
- "Bootloading Through The Link Port" on page 6-88
- "Bootloading Through the SPI Port" on page 6-113

For information on systems with a boot EPROM, see "Booting Single and Multiple Processors" on page 13-71.

Reading From Boot Memory

When the processor boots from an EPROM, the processor's I/O processor is hard-wired to load 256 instructions automatically from EPROM (via DMA). Once the initial 256-word DMA is complete, the processor typically needs to maintain access to boot memory. The processor does this by setting the Boot Select Override (BSO) bit in the SYSCON register.

Setting (=1) the BSO bit overrides the external memory selects and asserts the processor's $\overline{\text{BMS}}$ pin for an external memory DMA transfer. For accessing boot memory, the program first sets the BSO bit in SYSCON then sets up an external port DMA channel to read the EPROM's contents. The program must unmask the DMA channel's interrupt in the IMASK register; if using external port DMA buffer zero (EP0I), the program could enable this interrupt by setting the EP0I bit to 1 in the IMASK register. For more information on external port DMA, see "External Port DMA" on page 6-29.

While a program may use any external port DMA channel for accessing boot memory, it is important to note that only DMA channel 10 has a fixed 8- to 48-bit packing mode for boot memory reads. By using DMA channel 10 to complete initial program loading, a program can take advantage of this special packing mode.

When a program sets BSO, the processor ignores the DMA channel's packing mode (PMODE) bits for DMA channel 10 and forces 8- to 48-bit packing for reads. This 8-bit packing mode is used on DMA channel 10 during EPROM booting or on DMA reads when BSO is set. While one of the external port DMA channels is making a DMA access to boot memory with the BSO bit set, none of the other three channels may make a DMA access to external (not boot) memory.

Only external port DMA transfers assert $\overline{\text{BMS}}$ when BSO is set; processor core accesses to external memory always use the $\overline{\text{MSx}}$ pins. Because the processor core only accesses external (not boot) memory, programs can access external memory in between DMA accesses to boot memory.

Writing to Boot Memory

In systems using write-able EEPROM or FLASH memory for boot memory, programs can write new data to the processor's boot memory using the boot select override (BSO) pin. As described in "Reading From Boot Memory" on page 5-35, setting (=1) the BSO bit overrides the external memory selects and asserts the processor's $\overline{\text{BMS}}$ pin for an external memory DMA transfer.

To write to boot memory using the $\overline{\text{BMS}}$ signal, programs must use DMA channels 11, 12 or 13, but not DMA channel 10. With the BSO bit set, programs should only use DMA channel 10 for reads.

When BSO is set, programs can use DMA channels 11-13 with any settings in channel's the DMACx register, any packing mode, and any data or instruction.

Internal Interrupt Vector Table

The default location of the ADSP-21161's interrupt vector table depends on the processor's booting mode. When the processor boots from an external source (EPROM, host port, SPI port or link port booting), the vector table starts at address 0x0004 0000 (normal word). When the processor is in "no boot" mode (runs from external memory location 0x0020 0000 without loading), the interrupt vector table starts at address 0x0020 0000.

The Internal Interrupt Vector Table (IIVT) bit in the SYSCON register overrides the default placement of the vector table. If IIVT is set (=1), the interrupt table starts at address 0x0004 0000 (internal memory) regardless of the booting mode.

Internal Memory Data Width

The processor's internal memory blocks use normal word addressing to access either single-precision 32-bit data or extended-precision 40-bit data. Programs select the data width independently for each internal memory block using the Internal Memory Data Width (IMDW0 and IMDW1) bits in the SYSCON register. If a block's IMDWx bit is cleared (=0), normal word addressed accesses to the block access 32-bit data. If a block's IMDWx bit is set (=1), normal word addressed accesses to the block access 40-bit extended-precision data. Reading or writing 40-bit data using a normal word access to a memory block whose IMDWx bit is cleared (=0) has the following results.

- If a program tries to write 40-bit data (for example, a data register-to-memory transfer), the transfer truncates the lower 8-bits from the register; only writing 32 most significant bits.
- If a program tries to read 40-bit data (for example, a memory-to-data register transfer), the transfer zero-fills the lower 8 bits of the register; only reading the 32 most significant bits.

The Program Memory Bus Exchange (`PX`) register is the only exception to these transfer rules—all loads/stores of the `PX` register are performed as 48-bit accesses unless forced to 64-bit access with the `LW` mnemonic. If any 40-bit data must be stored in a memory block configured for 32-bit words, the program should use the `PX` register to access the 40-bit data in 48-bit words. Programs should take care not to corrupt any 32-bit data with this type of access. For more information, see "Restrictions on Mixing 32-Bit and 48-Bit Words" on page 5-28.

The Long word (`LW`) mnemonic only effects normal word address accesses and overrides all other factors (SIMD, IMDWx).

Memory Bank Size

The processor's external memory space has four banks of equal, fixed size. Mapping peripherals into different banks lets systems accommodate I/O devices with different timing requirements, because the banked regions have associated waitstate and access mode settings. This processor permits a glueless interface to multiple devices because each bank has a independent memory select signal associated with it. For more information, see "External Bank X Access Mode" on page 5-42 and "External Bank X Waitstates" on page 5-45.

As shown in Figure 5-9 on page 5-23, bank 0 starts at address 0x0020 0000 in external memory, and the banks 1, 2, and 3 regions follow. Whenever the processor generates an address that is located within one of the four banks, the processor asserts the corresponding memory select line ($\overline{\text{MS3-0}}$).The size of the memory banks is 3.67 Mwords (SRAM) or 15.67 Mwords (SDRAM).

External Bus Priority

The processor's internal bus architecture lets the PM bus, DM bus, and IOP bus try to access multiprocessor memory space or external memory space in the same cycle. This contending access produces a conflict that the processor resolves with a two level arbitration policy. The processor core's DM bus always has priority over the PM bus. External Bus Priority (EBPRx) bits in the SYSCON register control the further arbitration between the winning core bus and the I/O processor. The EBPRx field assigns priority as follows:

- If EBPR is 00, priority rotates between core and I/O processor buses. Priority is evaluated and switched in each cycle in which the conflict exists. For example, if the IOP was transferring data to the external port and the core tried to read from the external memory four times consecutively, the core and IOP would take turns accessing external memory for eight cycles.
- If EBPR is 01, the winning core bus has priority over the I/O processor bus.
- If EBPR is 10, the I/O processor bus has priority over the winning core bus.

Secondary Processor Element (PEy)

When the PEYEN bit in the MODE1 register is set (=1), the processor is in Single-Instruction, Multiple-Data (SIMD) mode. In SIMD mode, many data access operations differ from the processor's default Single-Instruction, Single-Data (SISD) mode. These differences relate to doubling the amount of data transferred for each data access.

Accesses in SIMD mode transfer both an explicit (named) location and an implicit (un-named, complementary) location. The explicit transfers is a data transfers between the explicit register and the explicit address, and the implicit transfer is between the implicit register and the implicit address.

For information on complementary (implicit) registers in SIMD mode accesses, see "Secondary Processing Element (PEy)" on page 2-37. For more information on complementary (implicit) memory locations in SIMD mode accesses, see "Accessing Memory" on page 5-46.

Broadcast Register Loads

The processor's `BDCST1` and `BDCST9` bits in the `MODE1` register control broadcast register loading. When broadcast loading is enabled, the processor writes to complementary registers or complementary register pairs in each processing element on writes that are indexed with DAG1 register `I1` (if `BDCST1` =1) or DAG2 register `I9` (if `BDCST9` =1). Broadcast load accesses are similar to SIMD mode accesses in that the processor transfers both an explicit (named) location and an implicit (un-named, complementary) location, but broadcast loading only influences writes to registers and write identical data to these registers. Broadcast mode is independent of SIMD mode.

Table 5-4 shows examples of explicit and implicit effects of broadcast register loads to both processing elements. Note that broadcast loading only effects loads of data registers (register file); broadcast loading does not effect register stores or loads to other system registers. And, broadcast loads only work on register loads; broadcast loading cannot be used for memory writes. For more information on broadcast loading, see "Accessing Memory" on page 5-46.

Table 5-4. Register Load Dual PE Broadcast Operation

Instruction (Explicit, PEx Operation)[1]	(Implicit, PEy operation)

Rx = dm(i1,ma);
Rx = pm(i9,mb);
Rx = dm(i1,ma), Ry = pm(i9,mb);
	Sx = dm(i1,ma);
	Sx = pm(i9,mb);
	Sx = dm(i1,ma), Sy = pm(i9,mb);

1 The post increment in the explicit operation is performed before the implicit instructions are executed.

Illegal I/O Processor Register Access

The processor monitors I/O processor register access when the Illegal I/O processor Register Access (`IIRAE`) bit in the `MODE2` register is set (=1). If access to the IOP registers is detected, an Illegal Input Condition Detected (`IICDI`) interrupt occurs. The interrupt is enabled in the `IMASK` register in the following cases:

- A core access to an IOP register occurs.
- A host external port access to an IOP register occurs.

The I/O processor's DMA controller cannot generate the `IICDI` interrupt. For more information, see "Mode Control 2 Register (MODE2)" on page A-10.

Unaligned 64-Bit Memory Access

The processor monitors for unaligned 64-bit memory accesses if the Unaligned 64-bit Memory Accesses (`U64MAE`) bit in the `MODE2` register (bit 21) is set (=1). An unaligned access is an odd numbered address normal word access that is forced to 64-bit with the `LW` mnemonic. When detected, this condition is an input that can cause an Illegal Input Condition Detected (IICDI) interrupt if the interrupt is enabled in the IMASK register. For more information, see “Mode Control 2 Register (MODE2)” on page A-10.

The following code example shows the access for even and odd addresses. When accessing an odd address, the sticky bit is set to indicate the unaligned access.

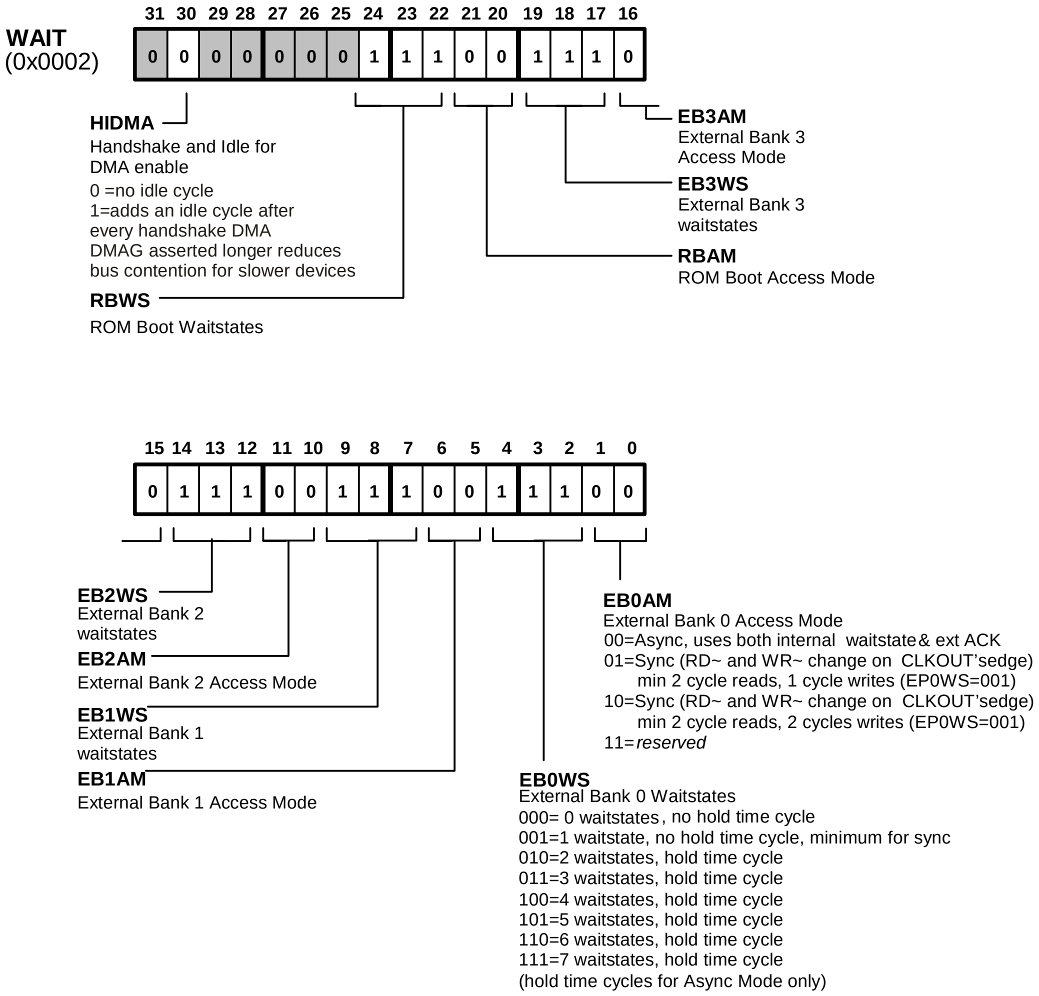
```
bit set mode2 U64MAE;  //set testbit for align or unaligned 64
bit access
r0=0x11111111;
r1=0x22222222;
pm(0x4e800)=r0(lw); //even address in 32 bit, access is aligned
pm(0x4e803)=r0(lw); //odd address in 32 bit, sticky bit is set
```

External Bank X Access Mode

The processor has four modes for accessing external memory space. The External Bank Access Mode (EBxAM) fields in the WAIT register select how the processor uses waitstates and the acknowledge (ACK) pin to access each external memory bank region. ACK has a 20 kΩ internal pull-up resistor that is enabled during reset or on DSPs with `ID2-0`=00x. The external bank access modes appear in Table 5-5. The `WAIT` register bit descriptions appear in Figure 5-15.

Table 5-5. External Bank Access Mode

EBxAM Field	External Bank Access Mode
00	Asynchronous $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes change before CLKOUT’s edge. Accesses use the waitstate count setting from EBxWS AND require external acknowledge (ACK), allowing a de-asserted ACK to extend the access time.
01	Synchronous $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes change on CLKOUT’s edge. Accesses use the waitstate count setting from EBxWS (minimum EBxWS=001) AND require external acknowledge (ACK), allowing a de-asserted ACK to extend the read access time. Writes are 0-wait state.
10	Synchronous $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes change on CLKOUT’s edge. Accesses use the waitstate count setting from EBxWS (minimum EBxWS=001) AND require external acknowledge (ACK), allowing a de-asserted ACK to extend the read access time. Writes are 1-wait state.
11	Reserved



31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
WAIT
(0x0002)
0 0 0 0 0 0 0 1 1 1 0 0 1 1 1 0
HIDMA
Handshake and Idle for DMA enable
0 =no idle cycle
1=adds an idle cycle after every handshake DMA
DMAG asserted longer reduces bus contention for slower devices
RBWS
ROM Boot Waitstates
EB3AM
External Bank 3 Access Mode
EB3WS
External Bank 3 waitstates
RBAM
ROM Boot Access Mode
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 0 0 1 1 1 0 0 1 1 1 0 0
EB2WS
External Bank 2 waitstates
EB2AM
External Bank 2 Access Mode
EB1WS
External Bank 1 waitstates
EB1AM
External Bank 1 Access Mode
EB0AM
External Bank 0 Access Mode
00=Async, uses both internal waitstate& ext ACK
01=Sync (RD~ and WR~ change on CLKOUT'sedge)
min 2 cycle reads, 1 cycle writes (EP0WS=001)
10=Sync (RD~ and WR~ change on CLKOUT'sedge)
min 2 cycle reads, 2 cycles writes (EP0WS=001)
11=reserved
EB0WS
External Bank 0 Waitstates
000= 0 waitstates, no hold time cycle
001=1 waitstate, no hold time cycle, minimum for sync
010=2 waitstates, hold time cycle
011=3 waitstates, hold time cycle
100=4 waitstates, hold time cycle
101=5 waitstates, hold time cycle
110=6 waitstates, hold time cycle
111=7 waitstates, hold time cycle
(hold time cycles for Async Mode only)

Figure 5-15. WAIT Register

External Bank X Waitstates

The processor applies waitstates to each external memory access depending on the bank's external memory access mode (`EBxAM`). The External Bank Waitstate (`EBxWS`) field in the `WAIT` register sets the number of waitstates for each bank as shown in Table 5-6.

Table 5-6 lists the hold time settings that EBxWS associates with external memory accesses. A hold time cycle is an inactive bus cycle that the processor inserts automatically at the end of a read or write, allowing a longer hold time for address and data. The address and data remain unchanged and are driven for one cycle after the processor deasserts the read or write strobes.

Table 5-6. External Bank Waitstates

EBxWS	# of Waitstates	Hold Time Cycle?[1]
000	0	no
001	1	no
010	2	yes
011	3	yes
100	4	yes
101	5	yes
110	6	yes
111	7	yes

1 Hold cycle applies to asynchronous mode only.

The processor applies hold time cycles regardless of the external bank access mode (`EBxAM`). For example, the asynchronous (`ACK` plus waitstate) mode could also have an associated hold time cycle.

Using Memory Access Status

As described in “Illegal I/O Processor Register Access” on page 5-41 and “Unaligned 64-Bit Memory Access” on page 5-41, the processor can provide illegal access information for long word or I/O register accesses. When these conditions occur, the processor updates an illegal condition flag in a sticky status (STKYx) register. Either of these two conditions can also generate a maskable interrupt. Two ways to use illegal access information are:

- **Interrupts.** Enable interrupts and use an interrupt service routine to handle the illegal access condition immediately. This method is appropriate if it is important to handle all illegal accesses as they occur.
- **STKYx registers.** Sticky registers hold a value that can be checked for a specific condition at a later time. Use the Bit Tst instruction to examine illegal condition flags in the STKY register after an interrupt to determine which illegal access condition occurred.

Accessing Memory

The word width of the processor core accesses to internal memory include the following:

48-bit access for instruction words, extended-precision normal word (40-bit) data, and PX register

- 64-bit access for long word data, and normal word (32-bit) or PX register data with the LW mnemonic
- 32-bit access for normal word (32-bit) data
- 16-bit access for short word data

The processor determines whether a normal word access is 32- or 40-bit from the internal memory block's `IMDWx` setting. For more information, see "Internal Memory Data Width" on page 5-37. While mixed accesses of 48-bit words and 16-, 32-, or 64-bit words at the same address are not allowed, mixed read/writes of 16-, 32-, and 64-bit words to the same address are allowed. For more information, see "Restrictions on Mixing 32-Bit and 48-Bit Words" on page 5-28.

The processor's DM and PM buses support 24 combinations of register-to-memory data access options. The following factors influence the data access type:

- Size of words: short word, normal word, extended-precision normal word, or long word
- Number of words: single- or dual-data move
- Mode of processor: SISD, SIMD, or broadcast load

Access Word Size

The processor's internal memory accommodates the following word sizes:

- 64-bit word data
- 48-bit instruction words
- 40-bit extended-precision normal word data
- 32-bit normal word data
- 16-bit short word data

The processor's external memory accommodates the following word sizes:

- 48-bit instruction words
- 40-bit extended-precision normal word data (accessed as 48-bit via PX)
- 32-bit normal word data

Long Word (64-Bit) Accesses

A program makes a long word (64-bit) access to internal memory, using an access to a long word address. Programs can also make a 64-bit access through normal word addressing with the LW mnemonic or through a PX register move with the LW mnemonic. Programs may not use long word addressing to access multiprocessor memory space or external memory. The address ranges for internal memory accesses appear in Figure 5-7 on page 5-17.

ⓘ Since the ADSP-21161 processor external port is 32 bits wide, the SIMD and long word accesses are not supported.

When data is accessed using long word addressing, the data is always long word aligned on 64-bit boundaries in internal memory space. When data is accessed using normal word addressing and the LW mnemonic, the program should maintain this alignment by using an even normal word address (least significant bit of address =0). This register selection aligns the normal word address with a 64-bit boundary (long word address).

All long word accesses load or store two consecutive 32-bit data values. The register file source or destination of a long word access is a set of two neighboring data registers in a processing element. In a forced long word access (uses the LW mnemonic), the even (normal word address) location moves to or from the explicit register in the neighbor-pair, and the odd (normal word address) location moves to or from the implicit register in the neighbor-pair. For example, the following long word moves could occur:

```
DM(0x40000) = R0 (LW);
/* The data in R0 moves to location DM(0x40000),
 and the data in R1 moves to location DM(0x40001) */
R0 (LW) = DM(0x40003)(LW);
/* The data at location DM(0x40002) moves to R0,
 and the data at location DM(0x40003) moves to R1 */
```

The example shows that `R0` and `R1` are a **neighbor** registers in the same processing element. Table 5-7 lists the other neighbor register assignments that apply to long word accesses.

In un-forced long word accesses (accesses to `LW` memory space), the processor places the lower 32-bits of the long word in the named (explicit) register and places the upper 32-bits of the long word in the neighbor (implicit) register.

Table 5-7. Neighbor Registers for Long Word Accesses

PEx neighbor registers
r0 neighbors r1
r2 neighbors r3
r4 neighbors r5
r6 neighbors r7
r8 neighbors r9
r10 neighbors r11
r12 neighbors r13
r14 neighbors r15

PEy neighbor registers
s0 neighbors s1
s2 neighbors s3
s4 neighbors s5
s6 neighbors s7
s8 neighbors s9
s10 neighbors s11
s12 neighbors s13
s14 neighbors s15

Programs can monitor for unaligned 64-bit accesses by enabling the U64MAE bit. For more information, see "Unaligned 64-Bit Memory Access" on page 5-41.

ⓘ The Long word (LW) mnemonic only effects normal word address accesses and overrides all other factors (PEYEN, IMDWx).

Instruction Word (48-Bit) and Extended-Precision Normal Word (40-Bit) Accesses

The sequencer uses 48-bit memory accesses for instruction fetches. Program can make 48-bit accesses with PX register moves, which default to 48-bit.

A program makes an extended-precision normal word (40-bit) access to internal memory using an access to a normal word address when that internal memory block's IMDWx bit is set (=1) for 40-bit words. Programs may not use extended-precision normal word addressing to access multiprocessor memory space or external memory. The address ranges for internal memory accesses appear in Figure 5-7 on page 5-17. For more information on configuring memory for extended-precision normal word accesses, see "Internal Memory Data Width" on page 5-37.

The processor transfers the 40-bit data to internal memory as a 48-bit value, zero-filling the least significant 8 bits on stores and truncating these 8 bits on loads. The register file source or destination of such an access is a single 40-bit data register.

Normal Word (32-Bit) Accesses

A program makes a normal word (32-bit) access to internal memory using an access to a normal word address when that internal memory block's IMDWx bit is cleared (=0) for 32-bit words. Programs use normal word addressing to access all processor memory spaces: internal, multiprocessor, and external memory space. The address ranges for memory accesses appear in Figure 5-7 on page 5-17, and Figure 5-9 on page 5-23.

The register file source or destination of a normal word access is a single 40-bit data register. The processor zero-fills the least significant 8 bits on loads and truncates these bits on stores.

External memory space accesses using normal word addressing and the LW mnemonic perform a 32-bit accesses, not a 64-bit access.

Short Word (16-Bit) Accesses

A program makes a short word (16-bit) access to internal memory, using an access to a short word address. Programs may not use short word addressing to access multiprocessor memory space or external memory. The address ranges for internal memory accesses appear in Figure 5-7 on page 5-17.

The register file source or destination of such an access is a single 40-bit data register. The processor zero-fills the least significant 8 bits on loads and truncates these bits on stores. Depending on the value of the SSE bit in the MODE1 system register, the processor loads the register's upper 16 bits by either:

- Zero-filling these bits if SSE=0
- Sign-extending these bits if SSE=1

SISD, SIMD, and Broadcast Load Modes

These three processing element modes influence memory accesses. For a comparison of their effects, see the examples in "Data Access Options" on page 5-52. For more information on SISD and SIMD modes, see "Secondary Processing Element (PEy)" on page 2-37.

Broadcast load mode is a hybrid between SISD and SIMD modes, transferring dual-data under special conditions. For examples of broadcast transfers, see "Data Access Options" on page 5-52. For more information on broadcast load mode, see "Broadcast Register Loads" on page 5-40.

Single and Dual Data Accesses

The number of transfers that occur in a cycle influences the data access operation. As described on page 5-5, the processor supports single-cycle, dual-data accesses to and from internal memory for register-to-memory and memory-to-register transfers. Dual-data accesses occur over the PM and DM bus and act independent of SIMD/SISD. Though only available for transfers between memory and data registers, dual-data transfers are extremely useful because they double the data throughput over single-data transfers.

Instruction Examples

```
R8 = DM (I4,M3), PM (I12,M13) = R0;    /* Dual access */
R0 = DM (I5,M5);                       / * Single access */
```

For examples of data flow paths for single- and dual-data transfers, see "Data Access Options" on page 5-52.

Data Access Options

Table 5-8 on page 5-53 lists the processor's possible memory transfer modes and provides a cross reference to examples of each memory access option that stems from the processor's data access options.

Table 5-8 shows the transfer modes that stem from the following data access options:

- The mode of the processor: SISD, SIMD, or Broadcast Load
- The size of access words: long, extended-precision normal word, normal word, or short word
- The number of transferred words: single- or dual-data

Note that long and short word addressing may not target multiprocessor memory space or external memory space.

Table 5-8. Memory Transfer Modes Cross Reference

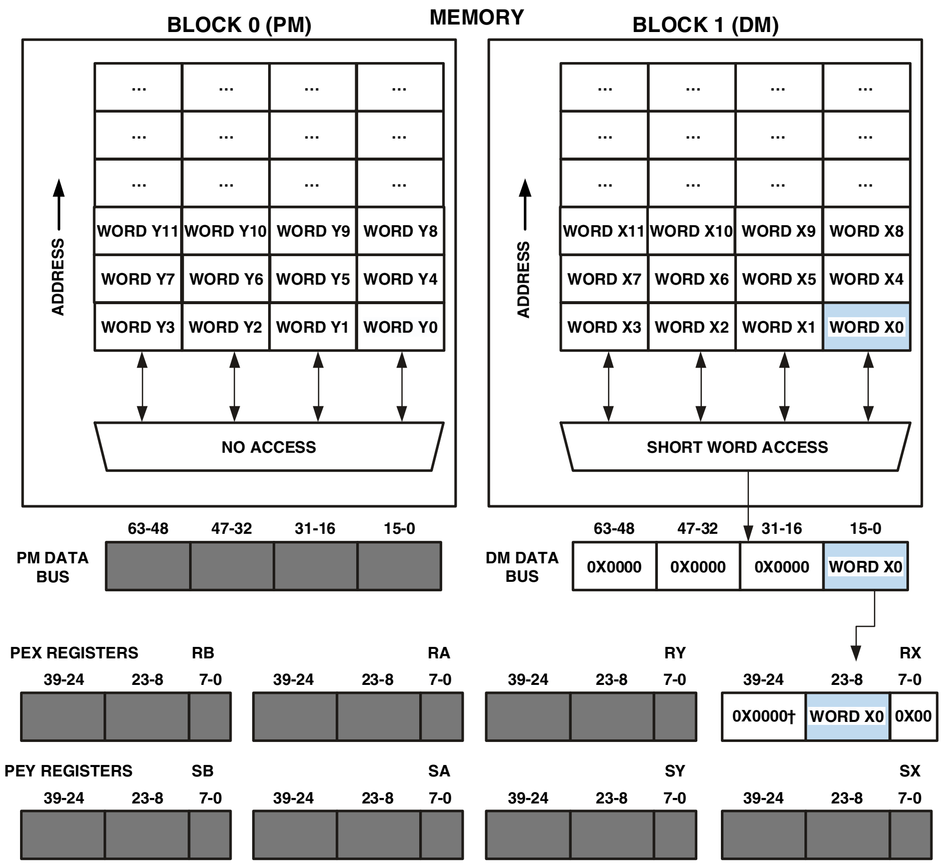
Access Type	Mode	Address Space			
		Long Word	Extended Precision	Normal Word	Short Word
Single Data Access	SISD mode	LW page 5-76	EW page 5-70	NW page 5-62	SW page 5-54
	SIMD mode	LW page 5-76	EW page 5-70	LW page 5-64	SWx2 page 5-56
	B-cast Load	LW Figure 5-38	EW Figure 5-36	NW Figure 5-34	SW Figure 5-32
Dual Data Access	SISD mode	LW page 5-78	EW page 5-72	NW page 5-66	SW page 5-58
	SIMD mode	LW page 5-80	EW page 5-74	LW page 5-68	SWx2 page 5-60
	B-cast Load	LW Figure 5-35	EW Figure 5-37	NW Figure 5-35	SW Figure 5-33
Symbols:LW = 64-bit data value (two 32-bit values), EW = 40-bit data value (48-bit value), NW = 32-bit data value, SW = 16-bit data value, and SWx2 = two 16-bit data values.					

Short Word Addressing of Single Data in SISD Mode

Figure 5-16 displays one possible SISD mode, single data, short word addressed access. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The 16-bit value for the short word access transfers using the least significant short word lane of the PM or DM data bus. The processor drives the other short word lanes of the data buses with zeros.

In SISD mode, the instruction accesses `PEx` registers to transfer data from memory. This instruction accesses `WORD X0` whose short word address has "00" for its least significant two bits of address. Other locations within this row have addresses with least significant two bits of "01", "10", or "11" and select `WORD X1`, `WORD X2`, or `WORD X3` from memory respectively. The syntax targets register, `RX`, in `PEx`. The example would target a `PEy` register if using the syntax `SX`.

The cross (†) in the `PEx` registers in Figure 5-16 indicates that the processor zero-fills or sign-extends the most significant 16 bits of the data register while loading the short word value into a 40-bit data register. This depends on the state of the `SSE` bit in the `MODE1` system register. For SW transfers, the least significant 8 bits of the data register are always zero.


MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y11 WORD Y10 WORD Y9 WORD Y8
WORD Y7 WORD Y6 WORD Y5 WORD Y4
WORD Y3 WORD Y2 WORD Y1 WORD Y0
WORD X11 WORD X10 WORD X9 WORD X8
WORD X7 WORD X6 WORD X5 WORD X4
WORD X3 WORD X2 WORD X1 WORD X0
NO ACCESS
SHORT WORD ACCESS
PM DATA BUS
DM DATA BUS
63-48 47-32 31-16 15-0
0X0000 0X0000 0X0000 WORD X0
PEX REGISTERS
RB RA RY RX
PEY REGISTERS
SB SA SY SX
39-24 23-8 7-0
0X0000† WORD X0 0X00

THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(SHORT WORD X0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD, SHORT WORD, SINGLE-DATA TRANSFERS ARE:
UREG = PM(SHORT WORD ADDRESS);
UREG = DM(SHORT WORD ADDRESS);
PM(SHORT WORD ADDRESS) = UREG;
DM(SHORT WORD ADDRESS) = UREG;

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES. DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-16. Short Word Addressing of Single Data in SISD Mode

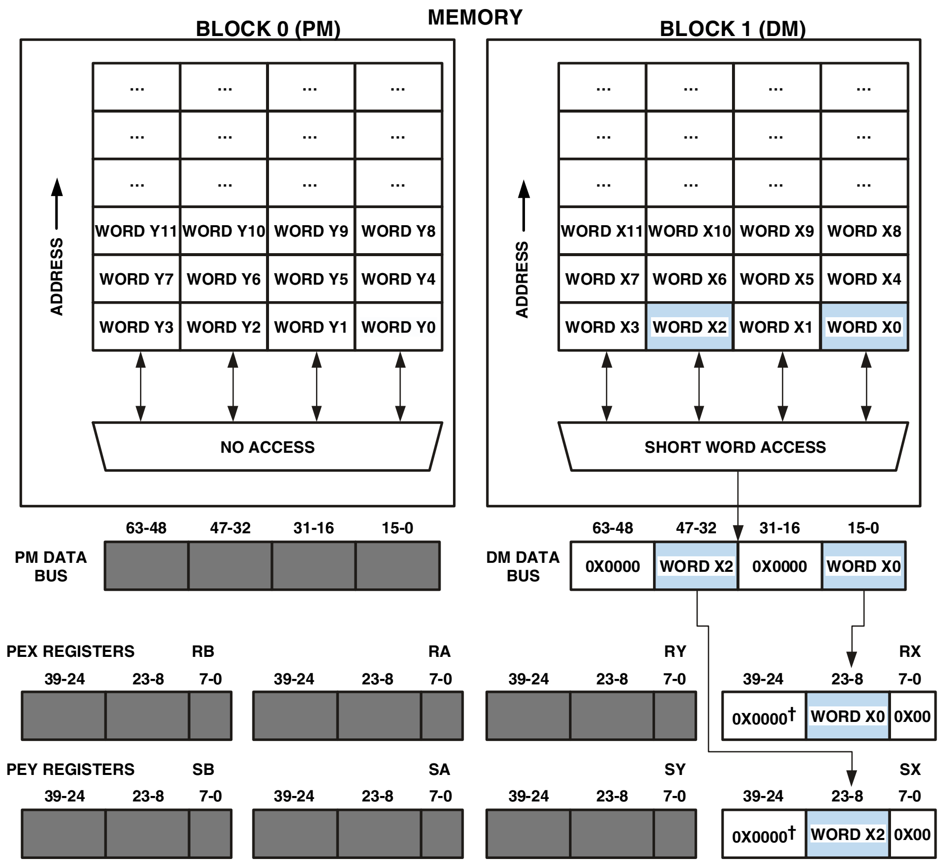
Short Word Addressing of Single Data in SIMD Mode

Figure 5-17 displays one possible SIMD mode, single data, short word addressed access. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The explicitly addressed (named in the instruction) 16-bit value transfers using the least significant short word lane of the PM or DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) short word value transfers using the 47-32 bit short word lane of the PM or DM data bus. The processor drives the other short word lanes of the PM or DM data buses with zeros.

The instruction explicitly accesses the register, RX, and implicitly accesses that register's complementary register, SX. This instruction uses a PEx register with an RX mnemonic. If the syntax named a PEy register SX as the explicit target the processor would use that register's complement RX as the implicit target. For more information on complementary registers, see "Secondary Processing Element (PEy)" on page 2-37.

The cross (†) in the PEx and PEy registers in Figure 5-17 indicates that the processor zero-fills or sign-extends the most significant 16 bits of the data register while loading the short word value into a 40-bit data register. This depends on the state of the SSE bit in the MODE1 system register. For short word accesses, the least significant 8 bits of the data register are always zero.

Figure 5-17 shows the data path for one transfer. The processor accesses short words sequentially in memory. Table 5-9 shows the pattern of SIMD mode short word accesses. For more information on arranging data in memory to take advantage of this access pattern, see "Arranging Data in Memory" on page 5-100.



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(SHORT WORD X0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, SHORT WORD, SINGLE-DATA TRANSFERS ARE:

|UREG = PM(SHORT WORD ADDRESS); |
|UREG = DM(SHORT WORD ADDRESS); |
|PM(SHORT WORD ADDRESS) = UREG; |
|DM(SHORT WORD ADDRESS) = UREG; |

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL DATA ACCESSES.
DUAL DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-17. Short Word Addressing of Single Data in SIMD Mode

Table 5-9. Short Word Addressing in SIMD Mode

Explicit Short Word Accessed	Implicit Short Word Accessed
Word X0 (Address two LSBs = 00)	Word X2 (Address two LSBs = 10)
Word X1 (Address two LSBs = 01)	Word X3 (Address two LSBs = 11)
Word X2 (Address two LSBs = 10)	Word X4 (Address two LSBs = 00)
Word X3 (Address two LSBs = 11)	Word X5 (Address two LSBs = 01)

Short Word Addressing of Dual-Data in SISD Mode

Figure 5-18 displays one possible SISD mode, dual-data, short word addressed access. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The 16-bit values for short word accesses transfer using the least significant short word lanes of the PM and DM data buses. The processor drives the other short word lanes of the data buses with zeros. Note that the accesses on both buses do not have to be the same word width. SISD mode dual-data accesses can handle any combination of short word, normal word, extended-precision normal word, or long word accesses. For more information, see "Mixed Word Width Addressing of Dual Data in SISD Mode" on page 5-82.

In SISD mode, the instruction explicitly accesses `PEx` registers. This instruction accesses `WORD X0` in block 1 and `WORD Y0` in block 0. Each of these words has a short word address with "00" for its least significant two bits of address. Other accesses within these 4-column location have the addresses with least significant two bits of "01", "10", or "11" and select `WORD X1/Y1`, `WORD X2/Y2`, or `WORD X3/Y3` from memory respectively. The syntax explicitly accesses registers, `RX` and `RY`, in `PEx`. The example would target `PEy` registers if using the syntax `SX` or `SY`.

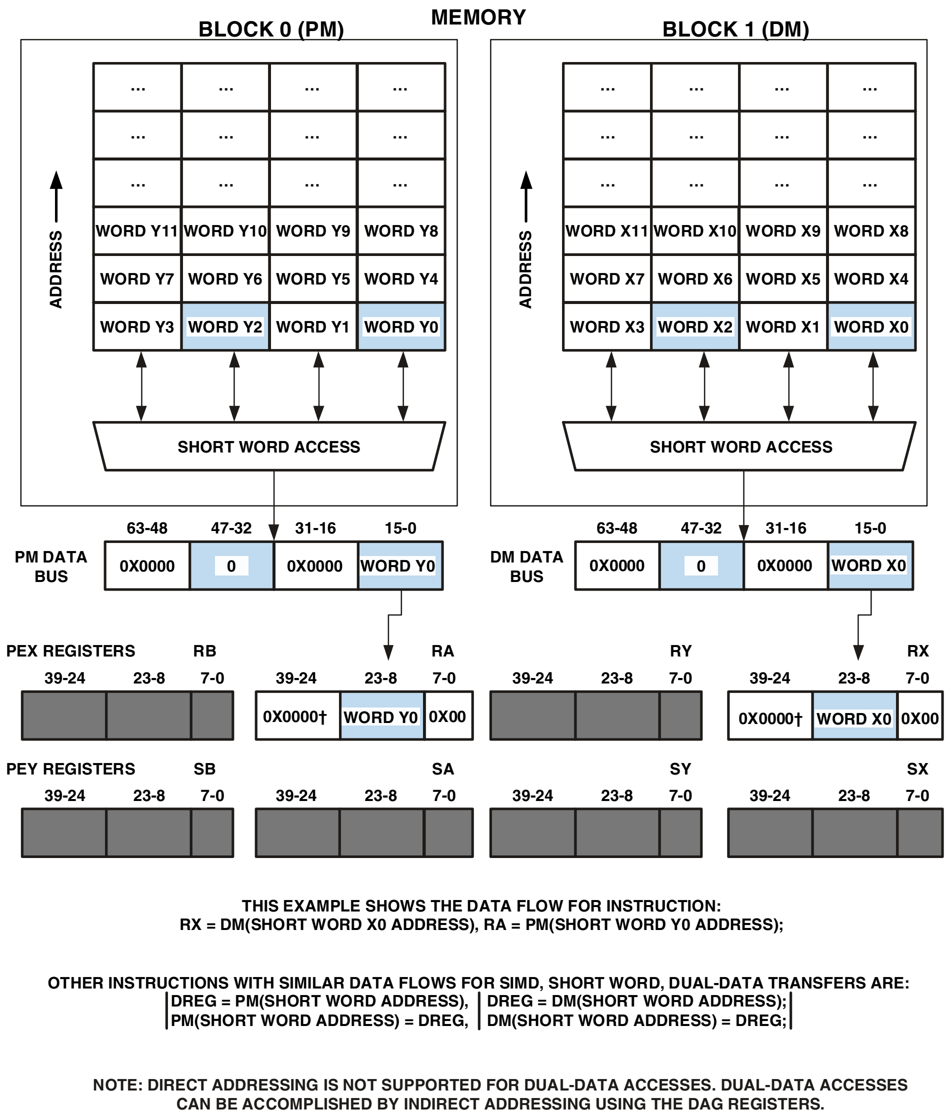

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y11 WORD Y10 WORD Y9 WORD Y8
WORD Y7 WORD Y6 WORD Y5 WORD Y4
WORD Y3 WORD Y2 WORD Y1 WORD Y0
WORD X11 WORD X10 WORD X9 WORD X8
WORD X7 WORD X6 WORD X5 WORD X4
WORD X3 WORD X2 WORD X1 WORD X0
SHORT WORD ACCESS
63-48 47-32 31-16 15-0
PM DATA BUS
0X0000 0 0X0000 WORD Y0
DM DATA BUS
0X0000 0 0X0000 WORD X0
PEX REGISTERS
RB RA RY RX
39-24 23-8 7-0
0X0000† WORD Y0 0X00
0X0000† WORD X0 0X00
PEY REGISTERS
SB SA SY SX
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(SHORT WORD X0 ADDRESS), RA = PM(SHORT WORD Y0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, SHORT WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(SHORT WORD ADDRESS), DREG = DM(SHORT WORD ADDRESS);
PM(SHORT WORD ADDRESS) = DREG, DM(SHORT WORD ADDRESS) = DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-18. Short Word Addressing of Dual-Data in SISD Mode

The cross (†) in the PEx registers in Figure 5-18 indicates that the processor zero-fills or sign-extends the most significant 16 bits of the data register while loading a short word value into a 40-bit data register. This depends on the state of the SSE bit in the MODE1 system register. For short word accesses, the least significant 8 bits of the data register are always zero.

Short Word Addressing of Dual-Data in SIMD Mode

Figure 5-19 displays one possible SIMD mode, dual-data, short word addressed access. For short word addressing, the processor treats the data buses as four 16-bit short word lanes. The explicitly addressed (named in the instruction) 16-bit values transfer using the least significant short word lanes of the PM and DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) short word values transfer using the 47-32 bit short word lanes of the PM and DM data buses. The processor drives the other short word lanes of the PM and DM data buses with zeros.

The accesses on both buses do not have to be the same word width. SIMD mode dual-data accesses can handle combinations of short word and normal word or extended-precision normal word and long word accesses. For more information, see "Mixed Word Width Addressing of Dual Data in SIMD Mode" on page 5-84.

The instruction explicitly accesses registers RX and RA, and implicitly accesses the complementary registers, SX and SA. This instruction uses a PEx registers with the RX and RA mnemonics. If the syntax named PEy registers SX and SA as the explicit targets, the processor would use those registers' complements, RX and RA, as the implicit targets. For more information on complementary registers, see "Secondary Processing Element (PEy)" on page 2-37.

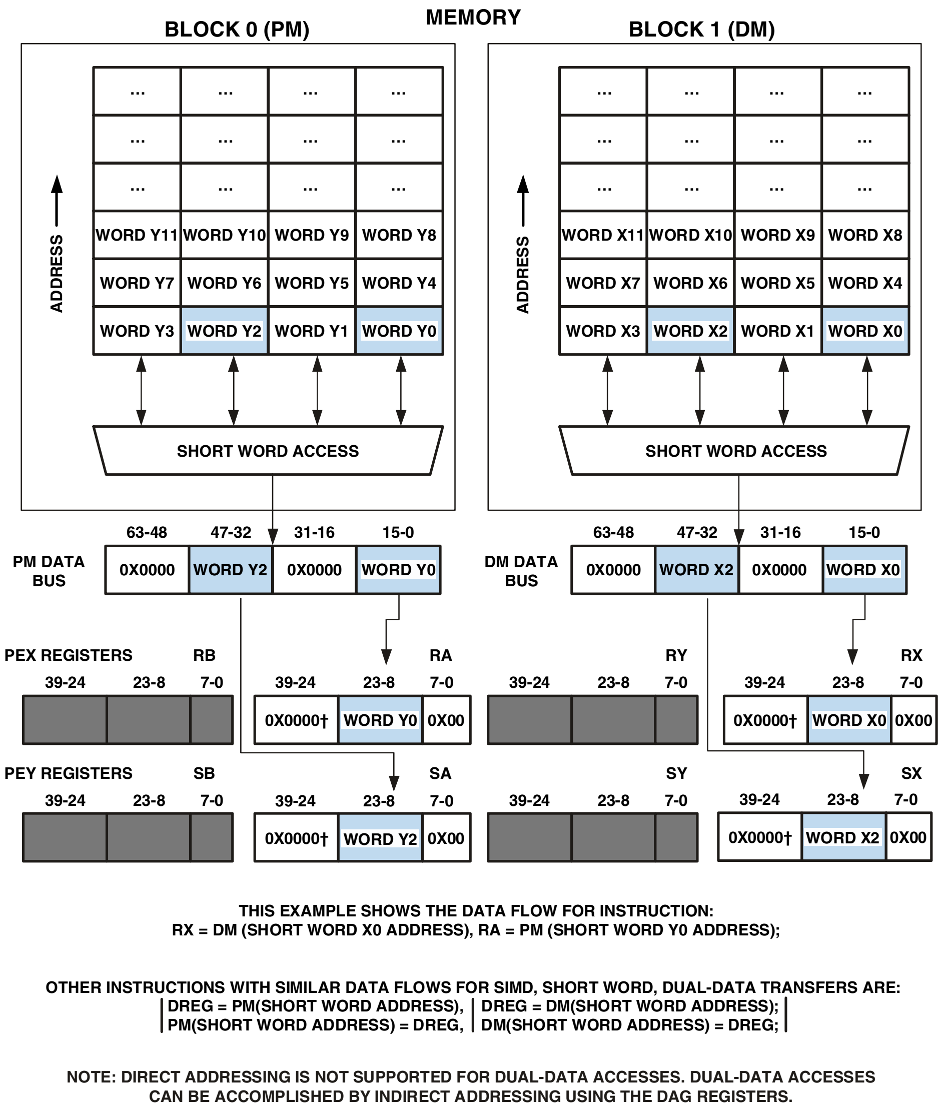

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
...
WORD Y11
WORD Y10
WORD Y9
WORD Y8
WORD Y7
WORD Y6
WORD Y5
WORD Y4
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X11
WORD X10
WORD X9
WORD X8
WORD X7
WORD X6
WORD X5
WORD X4
WORD X3
WORD X2
WORD X1
WORD X0
SHORT WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
0X0000
DM DATA BUS
PEX REGISTERS
RB
RA
RY
RX
39-24
23-8
7-0
0X0000†
0X00
PEY REGISTERS
SB
SA
SY
SX
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM (SHORT WORD X0 ADDRESS), RA = PM (SHORT WORD Y0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, SHORT WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(SHORT WORD ADDRESS), DREG = DM(SHORT WORD ADDRESS);
PM(SHORT WORD ADDRESS) = DREG, DM(SHORT WORD ADDRESS) = DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-19. Short Word Addressing of Dual-Data in SIMD Mode

The cross (†) in the PEx and PEy registers in Figure 5-19 indicates that the processor zero-fills or sign-extends the most significant 16 bits of the data registers while loading the short word values into the 40-bit data registers. For short word accesses, this depends on the state of the SSE bit in the MODE1 system register. For the short word accesses, the least significant 8 bits of the data register are always zero.

Figure 5-19 shows the data path for one transfer. For short word accesses, the processor accesses short words sequentially in memory. Table 5-9 on page 5-58 shows the pattern of SIMD mode short word accesses. For more information on arranging data in memory to take advantage of this access pattern, see "Arranging Data in Memory" on page 5-100.

32-Bit Normal Word Addressing of Single Data in SISD Mode

Figure 5-20 displays one possible SISD mode, single data, 32-bit normal word addressed access. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The 32-bit value for the normal word access transfers using the least significant normal word lane of the PM or DM data bus. The processor drives the other normal word lanes of the data buses with zeros.

In SISD mode, the instruction accesses a PEx register. This mode accesses WORD X0 whose normal word address has "0" for its least significant address bit. The other access within this 4-column location has an addresses with a least significant bit of "1" and selects WORD X1 from memory. The syntax targets register RX in PEx. The example would target a PEy register if using the syntax SX.

For normal word accesses, the processor zero-fills least significant 8 bits of the data register on loads and truncates these bits on stores to memory.

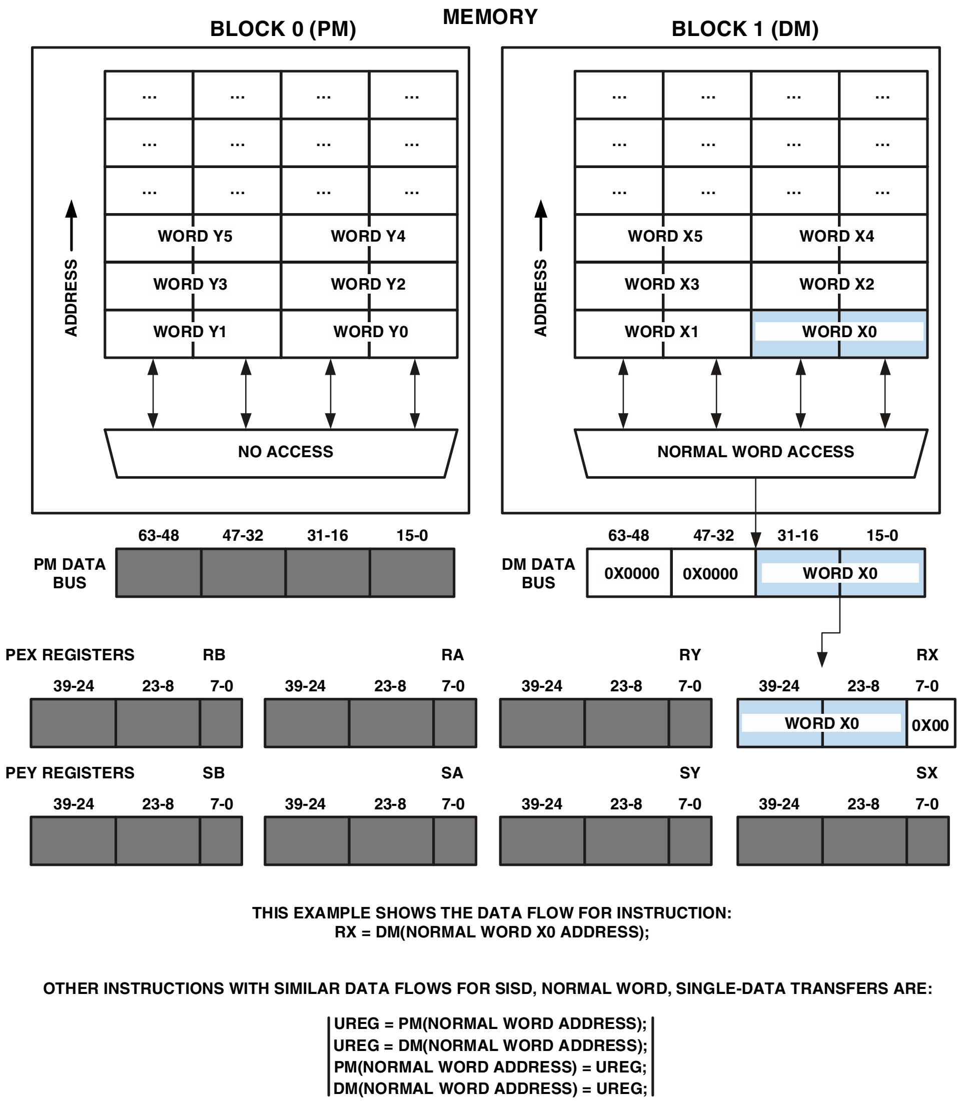

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y5
WORD Y4
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X5
WORD X4
WORD X3
WORD X2
WORD X1
WORD X0
NO ACCESS
NORMAL WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
0X0000
0X0000
WORD X0
PEX REGISTERS
RB
RA
RY
RX
39-24
23-8
7-0
WORD X0
0X00
PEY REGISTERS
SB
SA
SY
SX
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(NORMAL WORD X0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD, NORMAL WORD, SINGLE-DATA TRANSFERS ARE:
UREG = PM(NORMAL WORD ADDRESS);
UREG = DM(NORMAL WORD ADDRESS);
PM(NORMAL WORD ADDRESS) = UREG;
DM(NORMAL WORD ADDRESS) = UREG;

Figure 5-20. 32-Bit Normal Word Addressing of Single Data in SISD Mode

32-Bit Normal Word Addressing of Single Data in SIMD Mode

Figure 5-21 displays one possible SIMD mode, single data, normal word addressed access. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The explicitly addressed (named in the instruction) 32-bit value transfers using the least significant normal word lane of the PM or DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) Normal word value transfers using the most significant normal word lane of the PM or DM data bus.

In Figure 5-21, the explicit access targets the named register RX, and the implicit access targets that register's complementary register SX. This case uses a PEx register with an RX mnemonic. If the syntax named a PEy register SX as the explicit target, the processor would use that register's complement, RX, as the implicit target. For more information on complementary registers, see "Secondary Processing Element (PEy)" on page 2-37.

For normal word accesses, the processor zero-fills least significant 8 bits of the data register on loads and truncates these bits on stores to memory.

Figure 5-21 shows the data path for one transfer. For normal word accesses, the processor accesses normal words sequentially in memory. Table 5-9 shows the pattern of SIMD mode normal word accesses. For more information on arranging data in memory to take advantage of this access pattern, see "Arranging Data in Memory" on page 5-100.

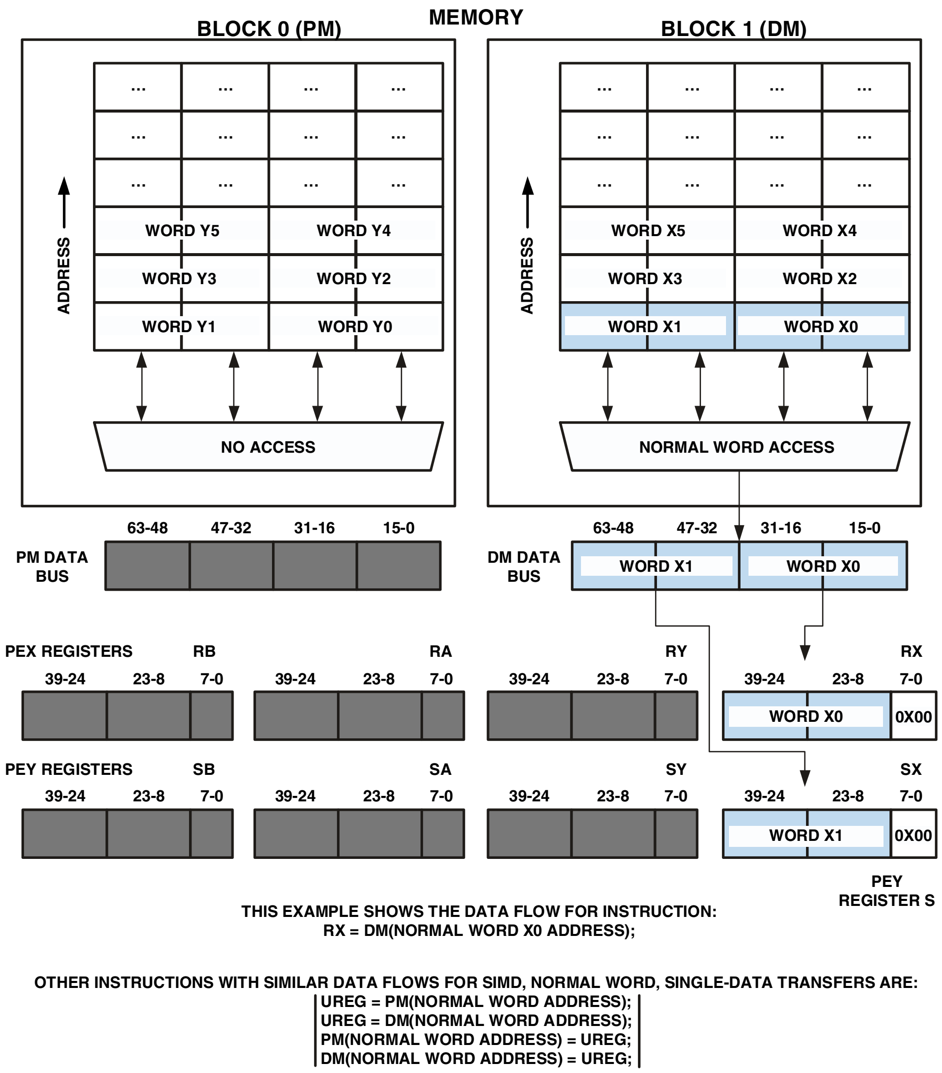

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y5
WORD Y4
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X5
WORD X4
WORD X3
WORD X2
WORD X1
WORD X0
NO ACCESS
NORMAL WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
PEX REGISTERS
RB
RA
RY
RX
39-24
23-8
7-0
0X00
PEY REGISTERS
SB
SA
SY
SX
PEY REGISTER S
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(NORMAL WORD X0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, NORMAL WORD, SINGLE-DATA TRANSFERS ARE:
UREG = PM(NORMAL WORD ADDRESS);
UREG = DM(NORMAL WORD ADDRESS);
PM(NORMAL WORD ADDRESS) = UREG;
DM(NORMAL WORD ADDRESS) = UREG;

Figure 5-21. 32-Bit Normal Word Addressing of Single Data in SIMD Mode

Table 5-10. Normal Word Addressing in SIMD Mode

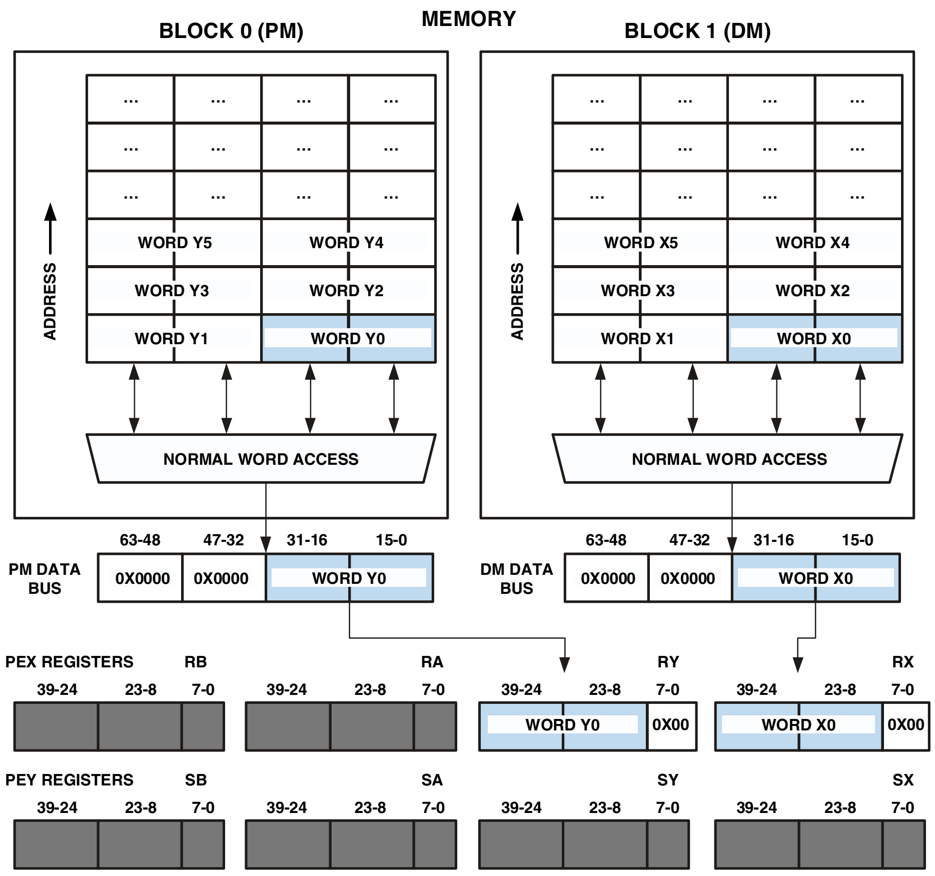
Explicit Normal Word Accessed	Implicit Normal Word Accessed
Word X0 (Address LSB = 0)	Word X1 (Address LSB = 1)
Word X1 (Address LSB = 1)	Word X2 (Address LSB = 0)

32-Bit Normal Word Addressing of Dual Data in SISD Mode

Figure 5-22 displays one possible SISD mode, dual data, 32-bit normal word addressed access. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The 32-bit values for normal word accesses transfer using the least significant normal word lanes of the PM and DM data buses. The processor drives the other normal word lanes of the data buses with zeros. Note that the accesses on both buses do not have to be the same word width. SISD mode dual-data accesses can handle any combination of short word, normal word, extended-precision normal word, or long word accesses. For more information, see "Mixed Word Width Addressing of Dual Data in SISD Mode" on page 5-82.

In Figure 5-22, the access targets `PEx` registers in a SISD mode operation. This case accesses `WORD X0` in block 1 and `WORD Y0` in block 0. Each of these words has a normal word address with "0" for its least significant address bit. Other accesses within these 4-column locations have the addresses with the least significant bit of "1" and select `WORD X1/Y1` from memory. The syntax targets registers `RX` and `RY` in `PEx`. The example would target `PEy` registers if using the syntax `SX` or `SY`.

For normal word accesses, the processor zero-fills least significant 8 bits of the data register on loads and truncates these bits on stores to memory.



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(NORMAL WORD X0 ADDRESS), RY = PM(NORMAL WORD Y0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD, NORMAL WORD, DUAL-DATA TRANSFERS ARE:
|DREG = PM(NORMAL WORD ADDRESS), | DREG = DM(NORMAL WORD ADDRESS); |
|PM(NORMAL WORD ADDRESS) = DREG, | DM(NORMAL WORD ADDRESS) = DREG; |

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-22. 32-Bit Normal Word Addressing of Dual Data in SISD Mode

32-Bit Normal Word Addressing of Dual Data in SIMD Mode

Figure 5-23 displays one possible SIMD mode, dual data, 32-bit normal word addressed access. For normal word addressing, the processor treats the data buses as two 32-bit normal word lanes. The explicitly addressed (named in the instruction) 32-bit values transfer using the least significant normal word lane of the PM or DM data bus. The implicitly addressed (not named in the instruction, but inferred from the address in SIMD mode) normal word values transfer using the most significant normal word lanes of the PM and DM data bus. Note that the accesses on both buses do not have to be the same word width. SIMD mode dual-data accesses can handle combinations of short word and normal word or extended-precision normal word and long word accesses. For more information, see "Mixed Word Width Addressing of Dual Data in SIMD Mode" on page 5-84.

In Figure 5-23, the explicit access targets the named registers `RX` and `RA`, and the implicit access targets those register's complementary registers `SX` and `SA`. This case uses a `PEx` registers with the `RX` and `RA` mnemonics. If the syntax named `PEy` registers `SX` and `SA` as the explicit targets, the processor would use those registers' complements `RX` and `RA` as the implicit targets. For more information on complementary registers, see "Secondary Processing Element (PEy)" on page 2-37.

For normal word accesses, the processor zero-fills least significant 8 bits of the data register on loads and truncates these bits on stores to memory.

Figure 5-23 shows the data path for one transfer. For normal word accesses, the processor accesses normal words sequentially in memory. Table 5-9 on page 5-58 shows the pattern of SIMD mode normal word accesses. For more information on arranging data in memory to take advantage of this access pattern, see "Arranging Data in Memory" on page 5-100.

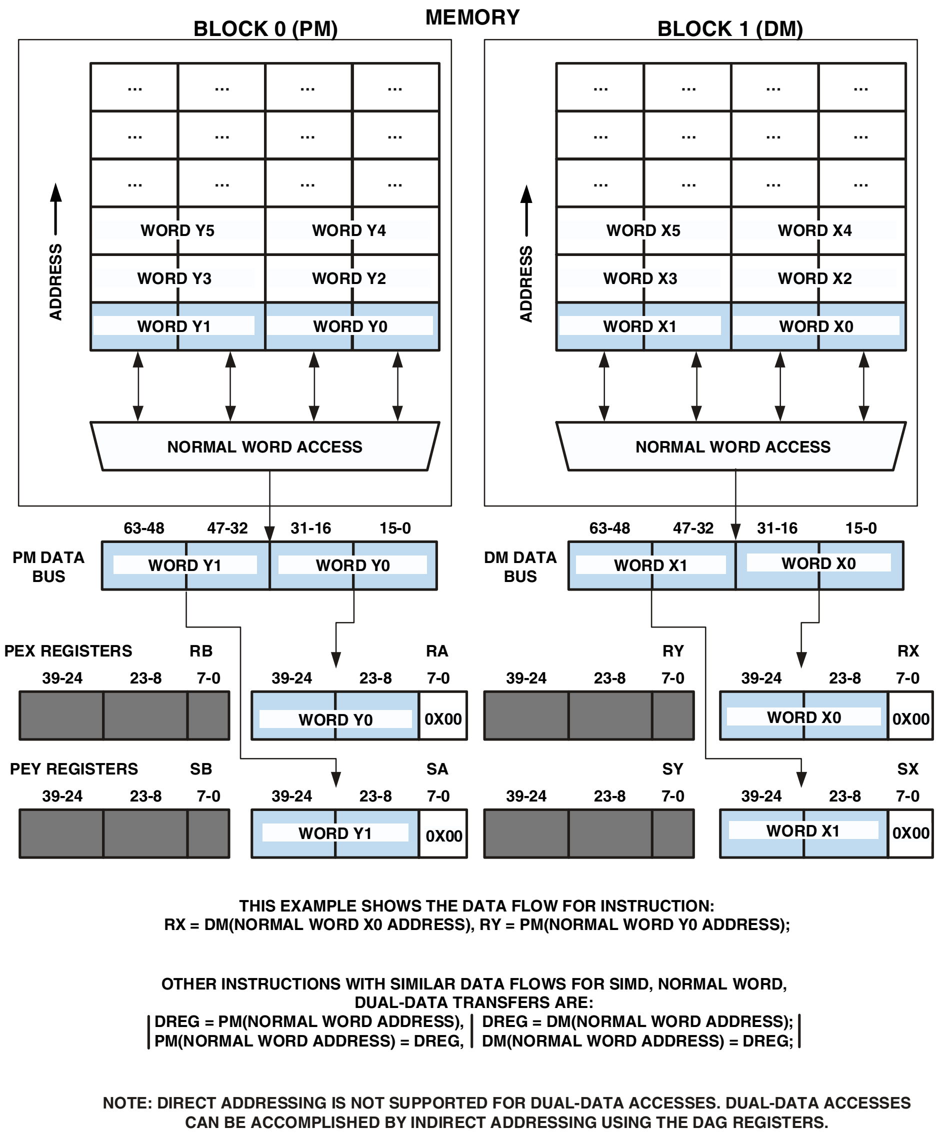
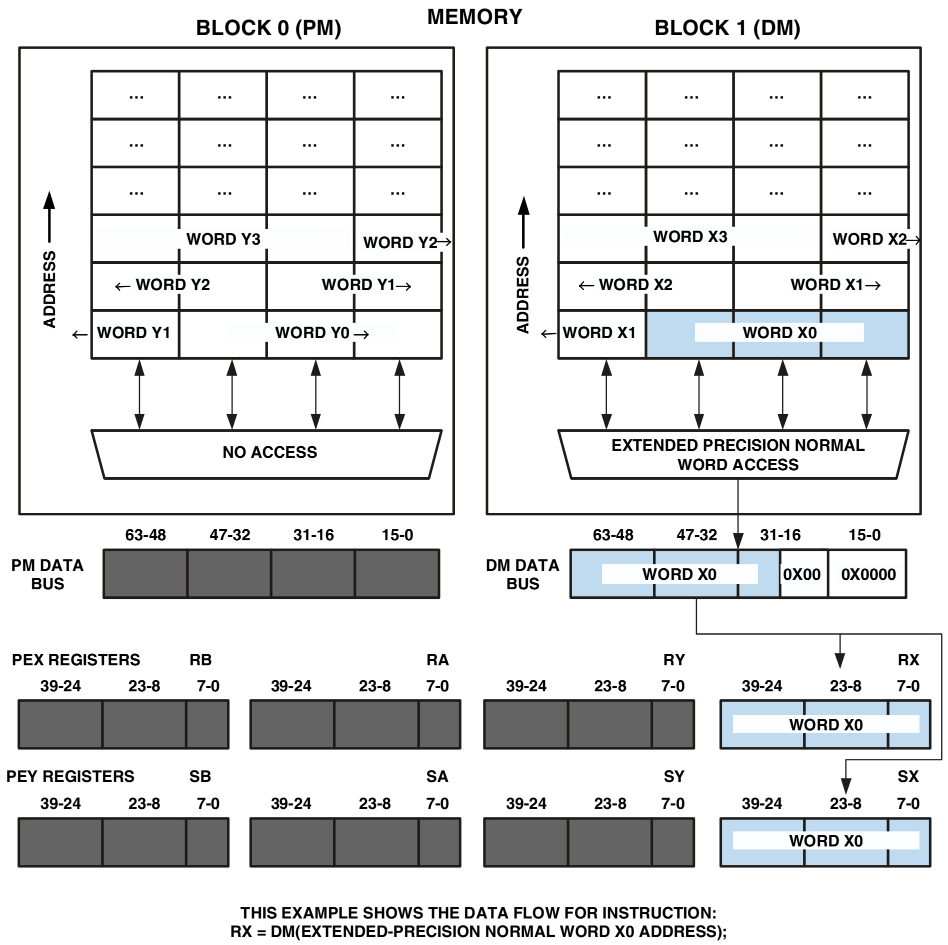

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y5
WORD Y4
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X5
WORD X4
WORD X3
WORD X2
WORD X1
WORD X0
NORMAL WORD ACCESS
PM DATA BUS
DM DATA BUS
63-48
47-32
31-16
15-0
PEX REGISTERS
PEY REGISTERS
RB
RA
RY
RX
SB
SA
SY
SX
39-24
23-8
7-0
0X00
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(NORMAL WORD X0 ADDRESS), RY = PM(NORMAL WORD Y0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, NORMAL WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(NORMAL WORD ADDRESS), | DREG = DM(NORMAL WORD ADDRESS);
PM(NORMAL WORD ADDRESS) = DREG, | DM(NORMAL WORD ADDRESS) = DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-23. 32-Bit Normal Word Addressing of Dual Data in SIMD Mode

Extended Precision Normal Word Addressing of Single Data

Figure 5-24 displays one possible single data, 40-bit extended-precision normal word addressed access. For extended-precision normal word addressing, the processor treats each data bus as a 40-bit extended-precision normal word lane. The 40-bit value for the extended-precision normal word access transfers using the most significant 40 bits of the PM or DM data bus. The processor drives the lower 24 bits of the data buses with zeros.

In Figure 5-24, the access targets a `PEx` register in a SISD or SIMD mode operation; extended-precision normal word single-data access operate the same in SISD or SIMD mode. This case accesses `WORD X0` with syntax that targets register `RX` in `PEx`. The example would target a `PEy` register if using the syntax `SX`.



OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, EXTENDED-PRECISION NORMAL WORD, SINGLE-DATA TRANSFERS ARE:

```
UREG = PM(EP NORMAL WORD ADDRESS);
UREG = DM(EP NORMAL WORD ADDRESS);
PM(EP NORMAL WORD ADDRESS) = UREG;
DM(EP NORMAL WORD ADDRESS) = UREG;
```

Figure 5-24. Extended Precision Normal Word Addressing of Single Data

Extended Precision Normal Word Addressing of Dual Data in SISD Mode

Figure 5-25 displays one possible SISD mode, dual data, 40-bit extended-precision normal word addressed access. For extended-precision normal word addressing, the processor treats each data bus as a 40-bit extended-precision normal word lane. The 40-bit values for the extended-precision normal word accesses transfer using the most significant 40 bits of the PM and DM data bus. The processor drives the lower 24 bits of the data buses with zeros. Note that the accesses on both buses do not have to be the same word width. SISD mode dual-data accesses can handle any combination of short word, normal word, extended-precision normal word, or long word accesses. For more information, see "Mixed Word Width Addressing of Dual Data in SISD Mode" on page 5-82.

In Figure 5-25, the access targets `PEx` registers in a SISD mode operation. This case accesses `WORD X0` in block 1 and `WORD Y0` in block 0 with syntax that targets registers `RX` and `RY` in `PEx`. The example would target a `PEy` registers if using the syntax `SX` or `SY`.

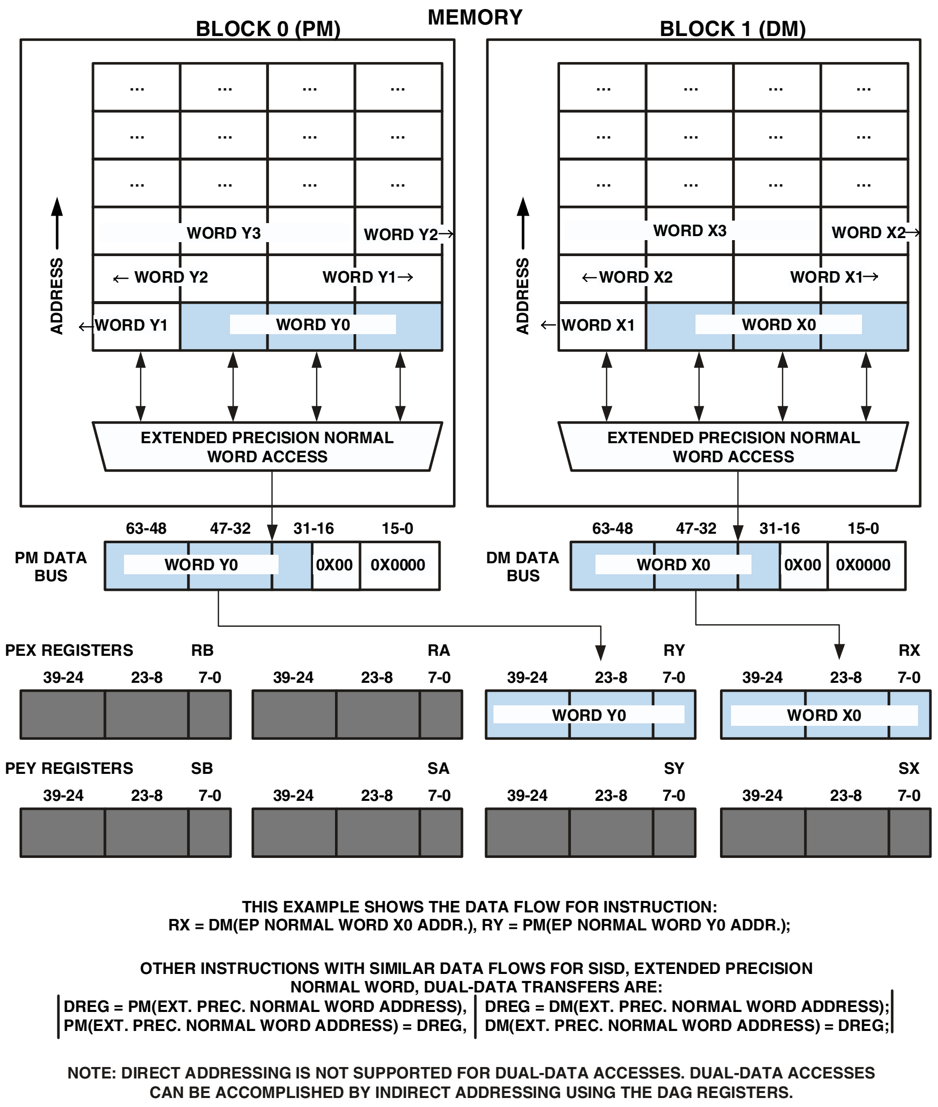

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X3
WORD X2
WORD X1
WORD X0
EXTENDED PRECISION NORMAL WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
0X00
0X0000
PEX REGISTERS
RB
RA
RY
RX
PEY REGISTERS
SB
SA
SY
SX
39-24
23-8
7-0
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(EP NORMAL WORD X0 ADDR.), RY = PM(EP NORMAL WORD Y0 ADDR.);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD, EXTENDED PRECISION NORMAL WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(EXT. PREC. NORMAL WORD ADDRESS),
PM(EXT. PREC. NORMAL WORD ADDRESS) = DREG,
DREG = DM(EXT. PREC. NORMAL WORD ADDRESS);
DM(EXT. PREC. NORMAL WORD ADDRESS) = DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-25. Extended-Precision Normal Word Addressing of Dual Data in SISD Mode

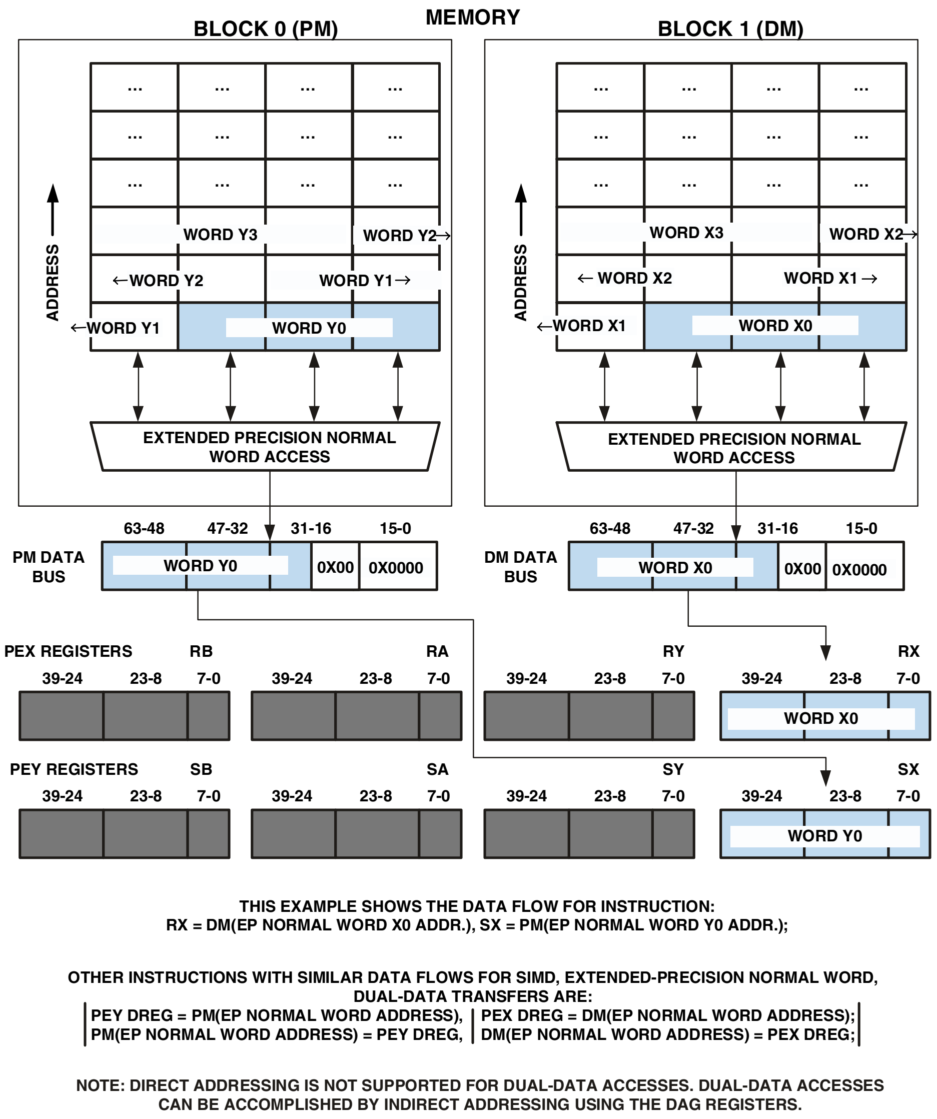
Extended-Precision Normal Word Addressing of Dual Data in SIMD Mode

Figure 5-26 displays one possible SIMD mode, dual data, 40-bit extended-precision normal word addressed access. For extended-precision normal word addressing, the processor treats each data bus as a 40-bit extended-precision normal word lane.

Because this word size approaches the limit of the data buses capacity, this SIMD mode transfer only moves the explicitly addressed locations and restricts data bus usage. The explicitly addressed (named in the instruction) 40-bit values transferred over the DM bus must source or sink a `PEx` data register, and the explicitly addressed (named in the instruction) 40-bit values transferred over the PM bus must source or sink a `PEy` data register; there are no implicit transfers in this mode. The 40-bit values for the extended-precision normal word accesses transfer using the most significant 40 bits of the PM and DM data bus. The processor drives the lower 24 bits of the data buses with zeros.

The accesses on both buses do not have to be the same word width. This special case of SIMD mode dual-data accesses can handle any combination of extended-precision normal word or long word accesses. For more information, see "Mixed Word Width Addressing of Dual Data in SIMD Mode" on page 5-84.

In Figure 5-26, the access targets `PEx` and `PEy` registers in a SIMD mode operation. This case accesses `WORD X0` in block 1 with syntax that targets register `RX` in `PEx` and accesses `WORD Y0` in block 0 with syntax that targets register `SX` in `PEy`.



MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X3
WORD X2
WORD X1
WORD X0
EXTENDED PRECISION NORMAL WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
0X00
0X0000
PEX REGISTERS
PEY REGISTERS
RB
RA
RY
RX
SB
SA
SY
SX
39-24
23-8
7-0
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(EP NORMAL WORD X0 ADDR.), SX = PM(EP NORMAL WORD Y0 ADDR.);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, EXTENDED-PRECISION NORMAL WORD, DUAL-DATA TRANSFERS ARE:
PEY DREG = PM(EP NORMAL WORD ADDRESS), PEX DREG = DM(EP NORMAL WORD ADDRESS);
PM(EP NORMAL WORD ADDRESS) = PEY DREG, DM(EP NORMAL WORD ADDRESS) = PEX DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-26. Extended-Precision Normal Word Addressing of Dual Data in SIMD Mode

Long Word Addressing of Single Data

Figure 5-27 displays one possible single data, long word addressed access. For long word addressing, the processor treats each data bus as a 64-bit long word lane. The 64-bit value for the long word access transfers using the full width of the PM or DM data bus.

In Figure 5-27, the access targets a `PEx` register in a SISD or SIMD mode operation; long word single-data access operate the same in SISD or SIMD mode. This case accesses `WORD X0` with syntax that explicitly targets register `RX` and implicitly targets its neighbor register `RY` in `PEx`. The example would target `PEy` registers if using the syntax `SX`. For more information on how neighbor registers (listed in Table 5-7 on page 5-49) work, see "Long Word (64-Bit) Accesses" on page 5-48.

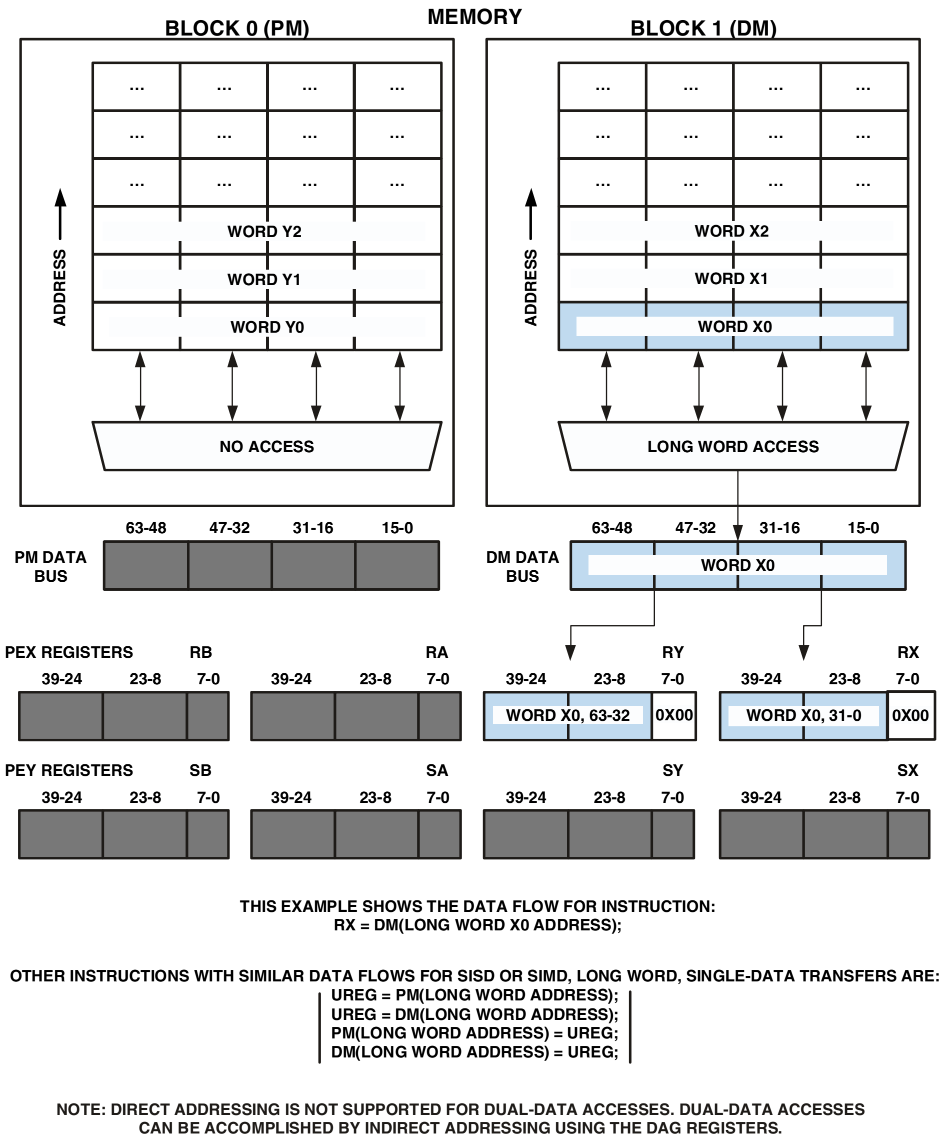

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y2
WORD Y1
WORD Y0
WORD X2
WORD X1
WORD X0
NO ACCESS
LONG WORD ACCESS
PM DATA BUS
DM DATA BUS
63-48
47-32
31-16
15-0
PEX REGISTERS
PEY REGISTERS
RB
RA
RY
RX
SB
SA
SY
SX
39-24
23-8
7-0
WORD X0, 63-32
WORD X0, 31-0
0X00
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD OR SIMD, LONG WORD, SINGLE-DATA TRANSFERS ARE:
UREG = PM(LONG WORD ADDRESS);
UREG = DM(LONG WORD ADDRESS);
PM(LONG WORD ADDRESS) = UREG;
DM(LONG WORD ADDRESS) = UREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES
CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-27. Long Word Addressing of Single Data

Long Word Addressing of Dual Data in SISD Mode

Figure 5-28 displays one possible SISD mode, dual data, long word addressed access. For long word addressing, the processor treats each data bus as a 64-bit long word lane. The 64-bit values for the long word accesses transfer using the full width of the PM or DM data bus.

In Figure 5-28, the access targets `PEx` registers in a SISD mode operation. This case accesses `WORD X0` and `WORD Y0` with syntax that explicitly targets registers `RX` registers `RA` and implicitly targets their neighbor registers `RY` and `RB` in `PEx`. The example would target `PEy` registers if using the syntax `SX` and `SA`. For more information on how neighbor registers (listed in Table 5-7 on page 5-49) work, see "Long Word (64-Bit) Accesses" on page 5-48.

Programs must be careful not to explicitly target neighbor registers in this case. While the syntax lets programs target these registers, one of the explicit accesses targets the other access's implicit target. The processor resolves this conflict by performing only the access with higher priority. For more information on the priority order of data register file accesses, see "Data Register File" on page 2-30.

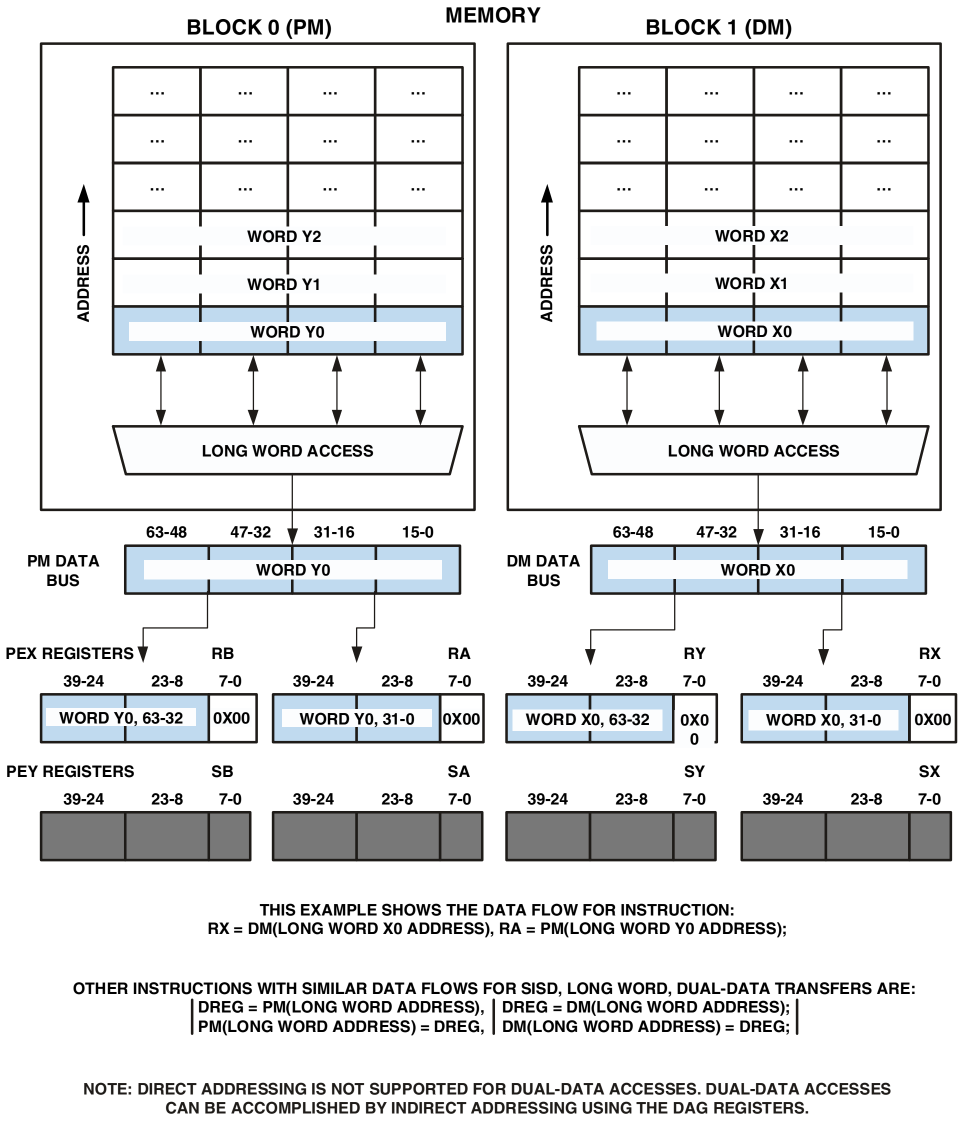

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y2
WORD Y1
WORD Y0
WORD X2
WORD X1
WORD X0
LONG WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
PEX REGISTERS
RB
RA
RY
RX
39-24
23-8
7-0
WORD Y0, 63-32
0X00
WORD Y0, 31-0
WORD X0, 63-32
0X0 0
WORD X0, 31-0
PEY REGISTERS
SB
SA
SY
SX
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS), RA = PM(LONG WORD Y0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD, LONG WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(LONG WORD ADDRESS), | DREG = DM(LONG WORD ADDRESS);
PM(LONG WORD ADDRESS) = DREG, | DM(LONG WORD ADDRESS) = DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-28. Long Word Addressing of Dual Data in SISD Mode

Long Word Addressing of Dual Data in SIMD Mode

Figure 5-29 displays one possible SIMD mode, dual data, long word addressed access targeting internal memory space. For long word addressing, the processor treats each data bus as a 64-bit long word lane. The 64-bit values for the long word accesses transfer using the full width of the PM or DM data bus.

Because this word size approaches the limit of the data buses capacity, this SIMD mode transfer only moves the explicitly addressed locations and restricts data bus usage. The explicitly addressed (named in the instruction) 64-bit values transferred over the DM bus must source or sink a `PEx` data register, and the explicitly addressed (named in the instruction) 64-bit values transferred over the PM bus must source or sink a `PEy` data register; there are no implicit transfers in this mode.

In Figure 5-29, the access targets `PEx` and `PEy` registers in a SIMD mode operation. This case accesses `WORD X0` in block 1 with syntax that targets register `RX` and its neighbor register `RY` in `PEx` and accesses `WORD Y0` in block 0 with syntax that targets register `SX` and its neighbor register `SY` in `PEy`. For more information on how neighbor registers (listed in Table 5-7 on page 5-49) work, see "Long Word (64-Bit) Accesses" on page 5-48.

The accesses on both buses do not have to be the same word width. This special case of SIMD mode dual-data accesses can handle any combination of extended-precision normal word or long word accesses. For more information, see "Mixed Word Width Addressing of Dual Data in SIMD Mode" on page 5-84.

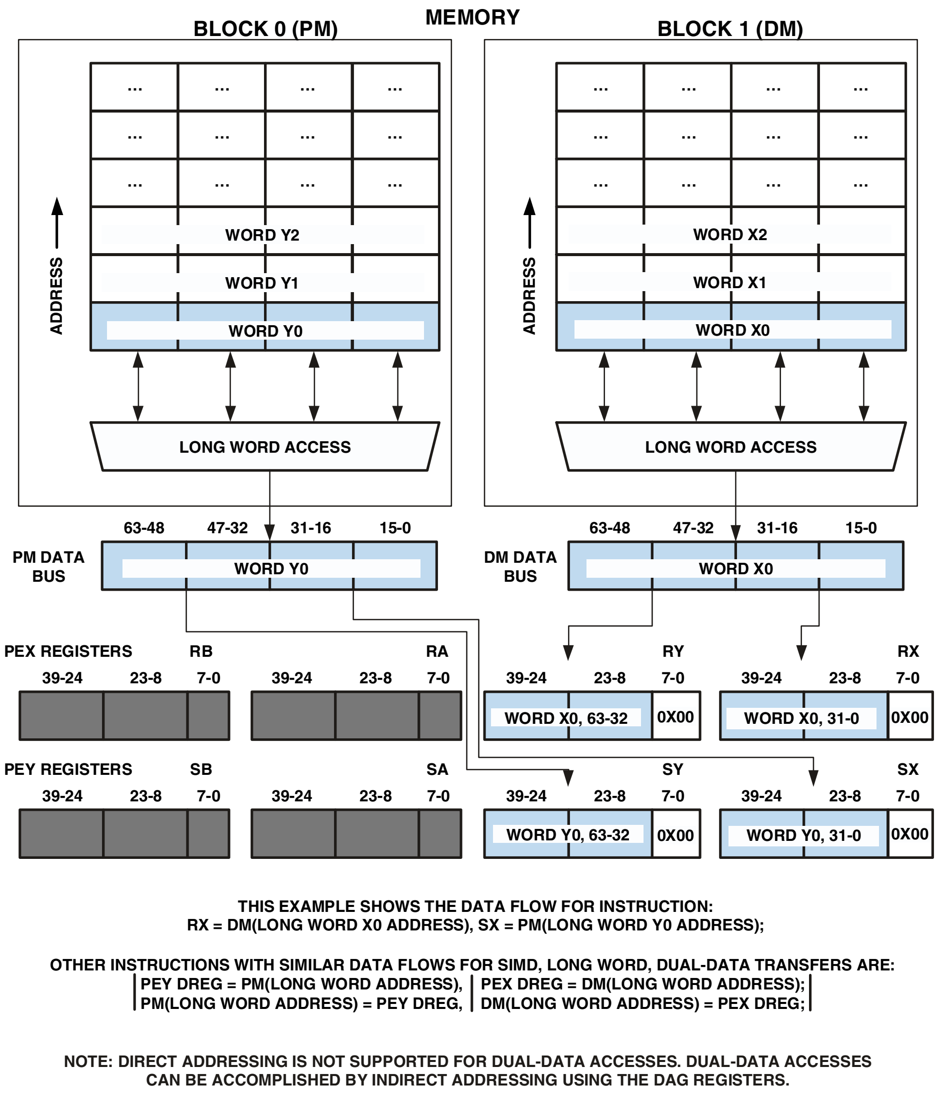
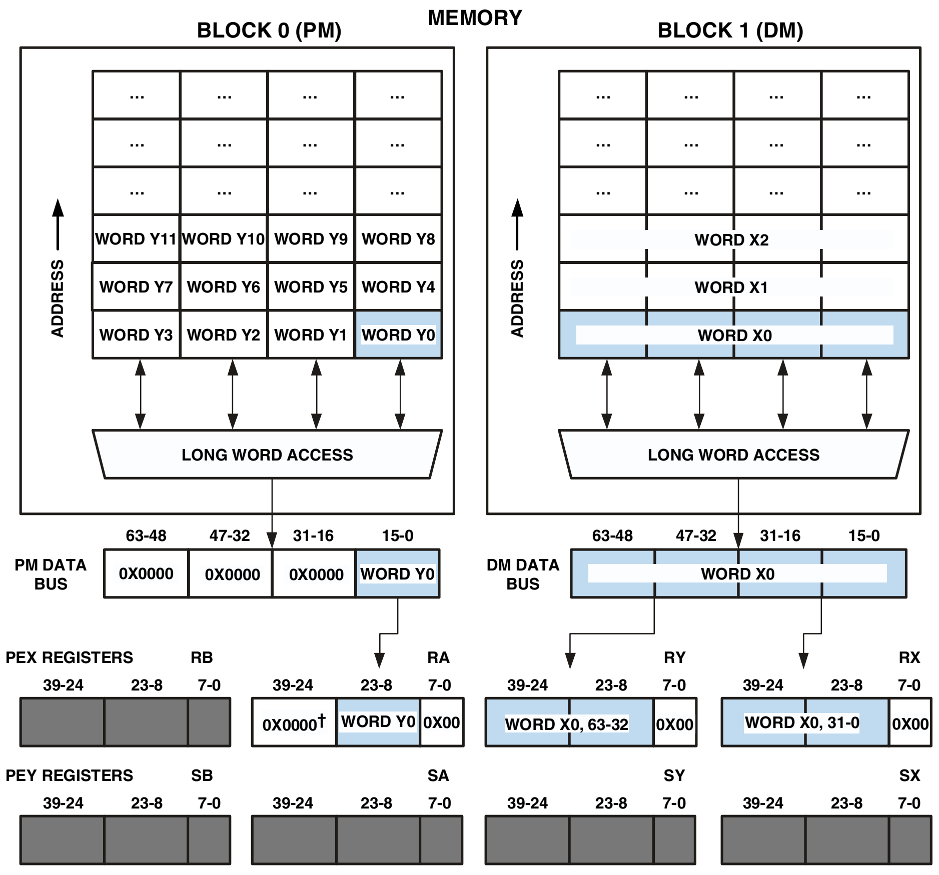

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
...
WORD Y2
WORD Y1
WORD Y0
WORD X2
WORD X1
WORD X0
ADDRESS
LONG WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
PEX REGISTERS
RB
RA
RY
RX
39-24
23-8
7-0
WORD X0, 63-32
0X00
WORD X0, 31-0
PEY REGISTERS
SB
SA
SY
SX
WORD Y0, 63-32
WORD Y0, 31-0
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS), SX = PM(LONG WORD Y0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, LONG WORD, DUAL-DATA TRANSFERS ARE:
PEY DREG = PM(LONG WORD ADDRESS), PEX DREG = DM(LONG WORD ADDRESS);
PM(LONG WORD ADDRESS) = PEY DREG, DM(LONG WORD ADDRESS) = PEX DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-29. Long Word Addressing of Dual Data in SIMD Mode

Mixed Word Width Addressing of Dual Data in SISD Mode

Figure 5-30 displays an example of a mixed word width, dual data, SISD mode access. This example shows how the processor transfers a long word access on the DM bus and transfers a short word access on the PM bus. The memory architecture permits mixing all other combinations of dual-data SISD mode short word, normal word, extended-precision normal word, and long word accesses.

In case of conflicting dual access to the data register file, the processor only performs the access with higher priority. For more information on how the processor prioritizes accesses, see "Data Register File" on page 2-30.



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS), RA = PM(SHORT WORD Y0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SISD, MIXED-WORD, DUAL-DATA TRANSFERS ARE:
|DREG = PM(SHORT, NORMAL, EP NORMAL, LONG ADD), | DREG = DM(SHORT, NORMAL, EP NORMAL, LONG ADD);|
|PM(SHORT, NORMAL, EP NORMAL, LONG ADD) = DREG, | DM(SHORT, NORMAL, EP NORMAL, LONG ADD) = DREG;|

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-30. Mixed Word Width Addressing of Dual Data in SISD Mode

Mixed Word Width Addressing of Dual Data in SIMD Mode

Figure 5-31 displays an example of a mixed word width, dual data, SIMD mode access. This example shows how the processor transfers a long word access on the DM bus and transfers an extended-precision normal word access on the PM bus.

ⓘ The memory architecture permits mixing SIMD mode dual data short word and normal word accesses or extended-precision normal word and long word accesses. No other combinations of mixed word dual-data SIMD mode accesses are permissible.

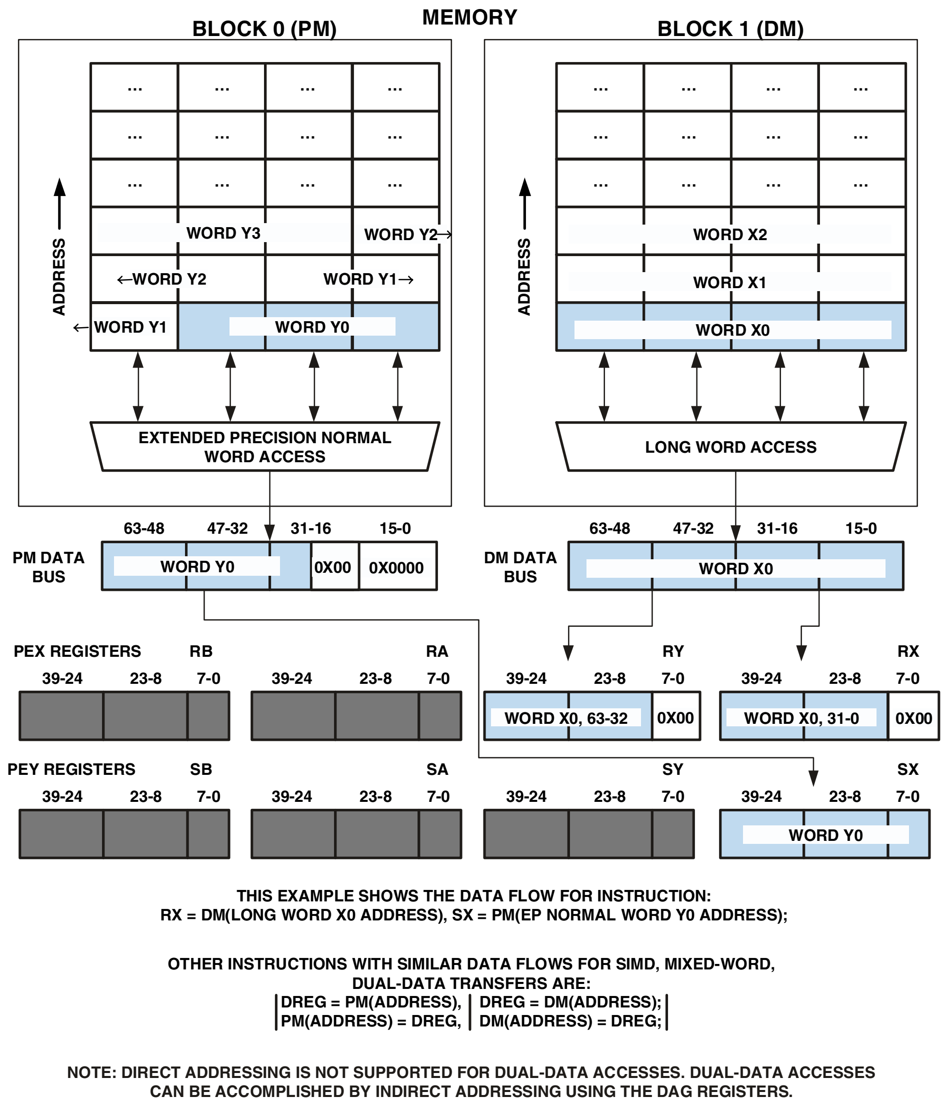

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y3
WORD Y2
WORD Y2
WORD Y1
WORD Y1
WORD Y0
WORD X2
WORD X1
WORD X0
EXTENDED PRECISION NORMAL WORD ACCESS
LONG WORD ACCESS
PM DATA BUS
DM DATA BUS
63-48
47-32
31-16
15-0
0X00
0X0000
PEX REGISTERS
PEY REGISTERS
RB
RA
RY
RX
SB
SA
SY
SX
39-24
23-8
7-0
WORD X0, 63-32
WORD X0, 31-0
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS), SX = PM(EP NORMAL WORD Y0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR SIMD, MIXED-WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(ADDRESS), DREG = DM(ADDRESS);
PM(ADDRESS) = DREG, DM(ADDRESS) = DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-31. Mixed Word Width Addressing of Dual Data in SIMD Mode

Broadcast Load Access

Figure 5-32 through Figure 5-39 provide examples of broadcast load accesses for single- and dual-data transfers. These examples show that the broadcast load's memory and register access is a hybrid of the corresponding non-broadcast SISD and SIMD mode accesses. The exceptions to this relation are broadcast load dual-data, extended-precision normal word and long word accesses. These broadcast accesses differ from their corresponding non-broadcast mode accesses.

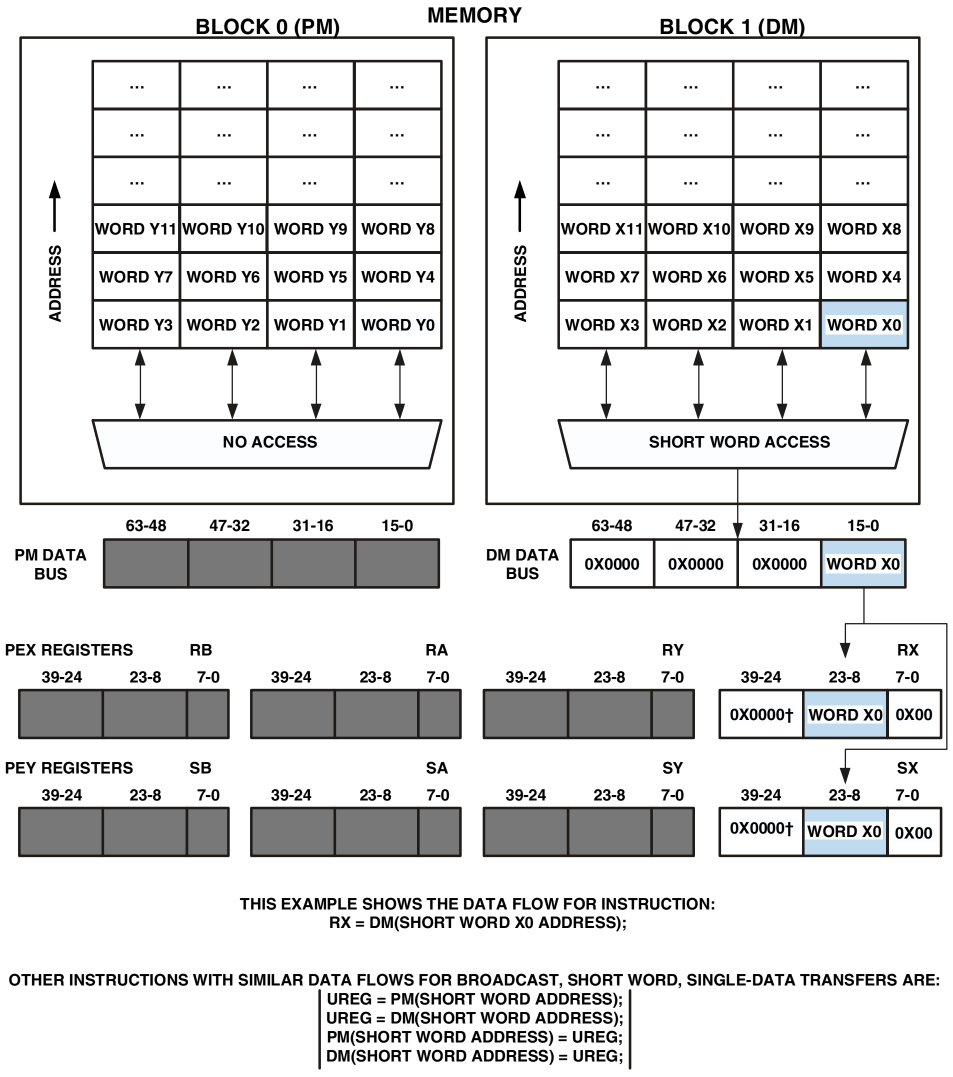
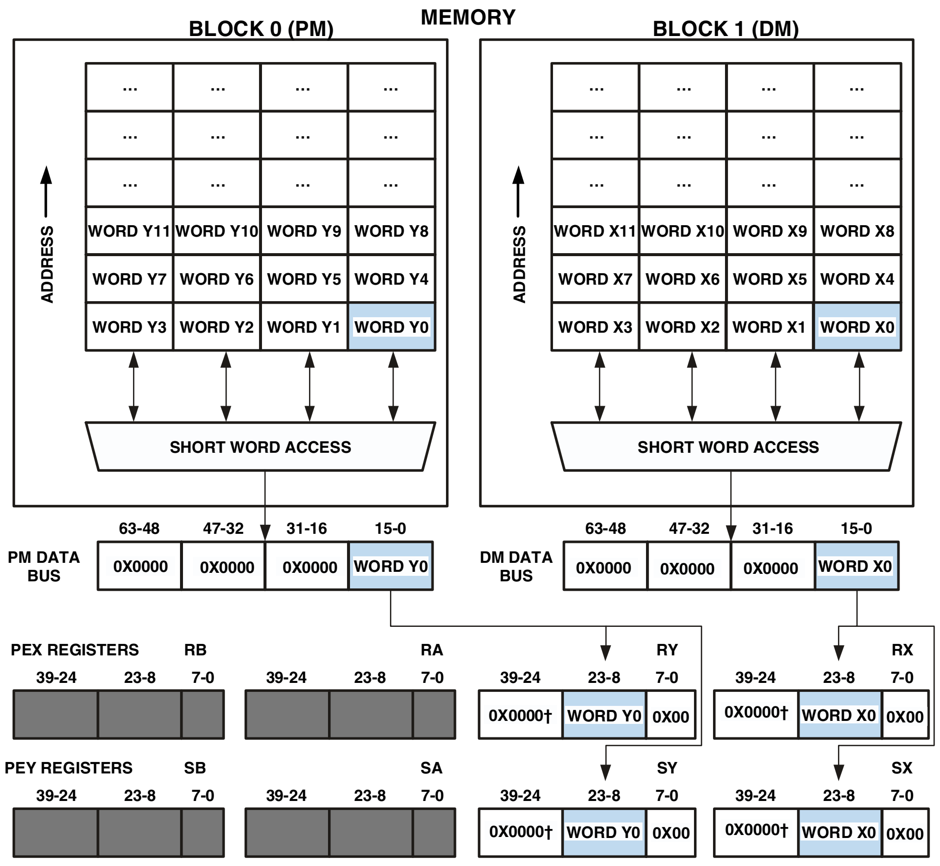

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y11 WORD Y10 WORD Y9 WORD Y8
WORD Y7 WORD Y6 WORD Y5 WORD Y4
WORD Y3 WORD Y2 WORD Y1 WORD Y0
WORD X11 WORD X10 WORD X9 WORD X8
WORD X7 WORD X6 WORD X5 WORD X4
WORD X3 WORD X2 WORD X1 WORD X0
NO ACCESS
SHORT WORD ACCESS
63-48 47-32 31-16 15-0
PM DATA BUS
DM DATA BUS
0X0000 0X0000 0X0000 WORD X0
PEX REGISTERS
RB RA RY RX
PEY REGISTERS
SB SA SY SX
39-24 23-8 7-0
0X0000† WORD X0 0X00
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(SHORT WORD X0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, SHORT WORD, SINGLE-DATA TRANSFERS ARE:
UREG = PM(SHORT WORD ADDRESS);
UREG = DM(SHORT WORD ADDRESS);
PM(SHORT WORD ADDRESS) = UREG;
DM(SHORT WORD ADDRESS) = UREG;

Figure 5-32. Short Word Addressing of Single Data in Broadcast Load



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(SHORT WORD X0 ADDRESS), RY = PM(SHORT WORD Y0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, SHORT WORD, DUAL-DATA TRANSFERS ARE:

| DREG = PM(SHORT WORD ADDRESS), | DREG = DM(SHORT WORD ADDRESS); |
| PM(SHORT WORD ADDRESS) = DREG, | DM(SHORT WORD ADDRESS) = DREG; |

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-33. Short Word Addressing of Dual Data in Broadcast Load

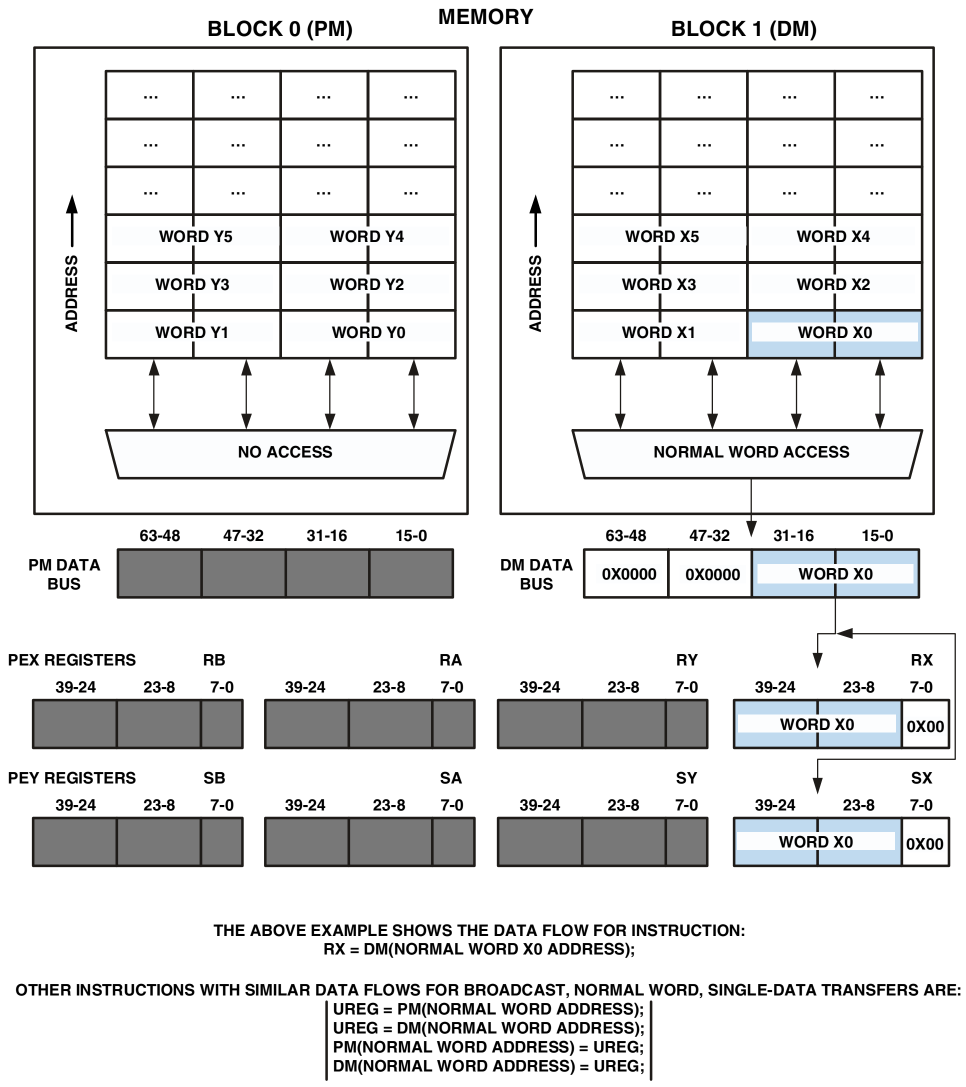
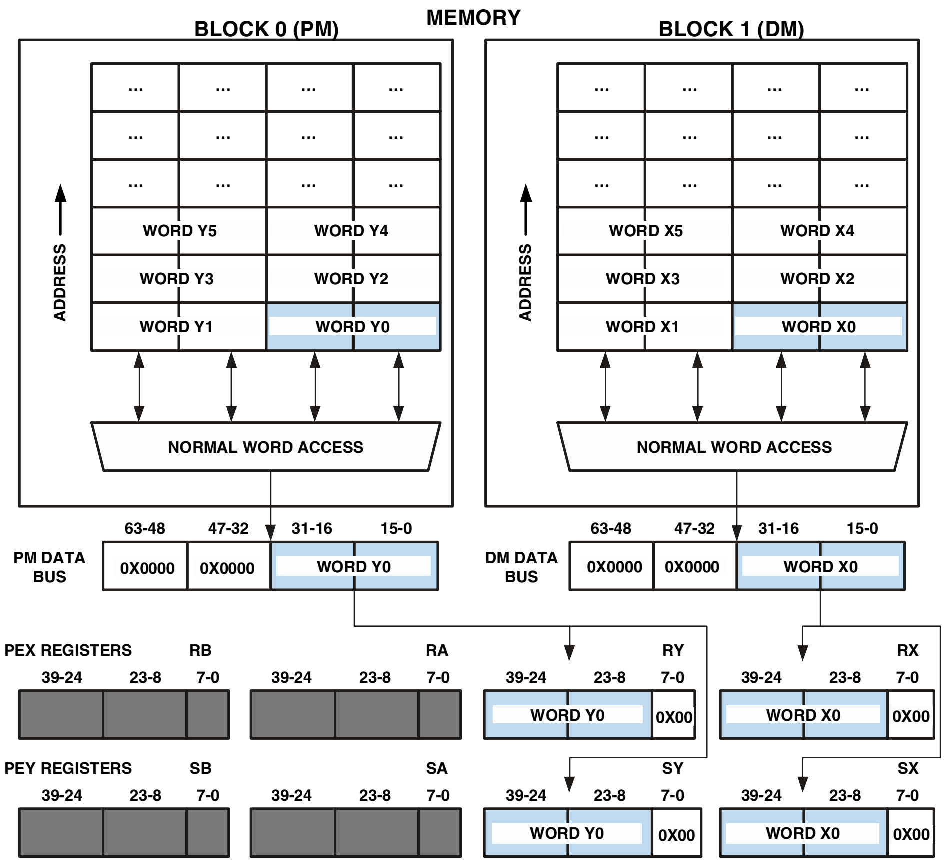

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y5
WORD Y4
WORD Y3
WORD Y2
WORD Y1
WORD Y0
WORD X5
WORD X4
WORD X3
WORD X2
WORD X1
WORD X0
NO ACCESS
NORMAL WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
DM DATA BUS
0X0000
0X0000
WORD X0
PEX REGISTERS
RB
RA
RY
RX
39-24
23-8
7-0
0X00
PEY REGISTERS
SB
SA
SY
SX
THE ABOVE EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(NORMAL WORD X0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, NORMAL WORD, SINGLE-DATA TRANSFERS ARE:
UREG = PM(NORMAL WORD ADDRESS);
UREG = DM(NORMAL WORD ADDRESS);
PM(NORMAL WORD ADDRESS) = UREG;
DM(NORMAL WORD ADDRESS) = UREG;

Figure 5-34. Normal Word Addressing of Single Data in Broadcast Load

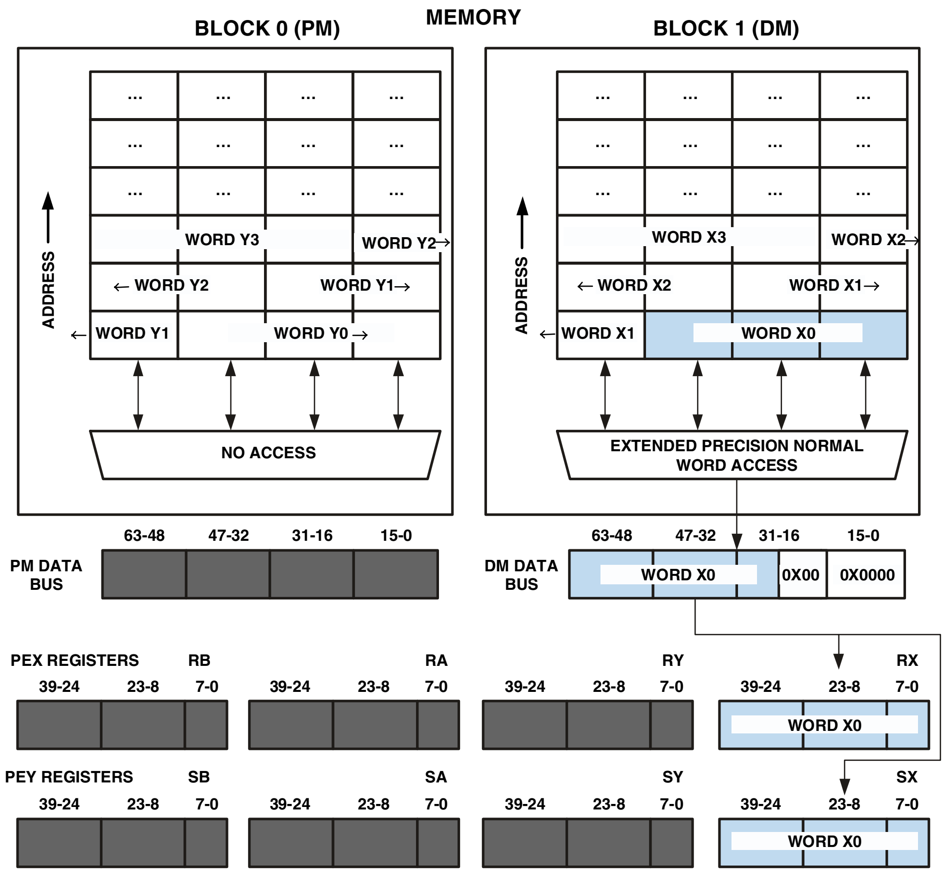


THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(NORMAL WORD X0 ADDRESS), RY = PM(NORMAL WORD Y0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, NORMAL WORD, DUAL-DATA TRANSFERS ARE:
|DREG = PM(NORMAL WORD ADDRESS), | DREG = DM(NORMAL WORD ADDRESS);|
|PM(NORMAL WORD ADDRESS) = DREG, | DM(NORMAL WORD ADDRESS) = DREG;|

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-35. Normal Word Addressing of Dual Data in Broadcast Load



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(EXTENDED-PRECISION NORMAL WORD X0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, EXTENDED-PRECISION NORMAL WORD, SINGLE-DATA TRANSFERS ARE:

```
UREG = PM(EP NORMAL WORD ADDRESS);
UREG = DM(EP NORMAL WORD ADDRESS);
PM(EP NORMAL WORD ADDRESS) = UREG;
DM(EP NORMAL WORD ADDRESS) = UREG;
```

Figure 5-36. Extended Precision Normal Word Addressing of Single Data in Broadcast Load

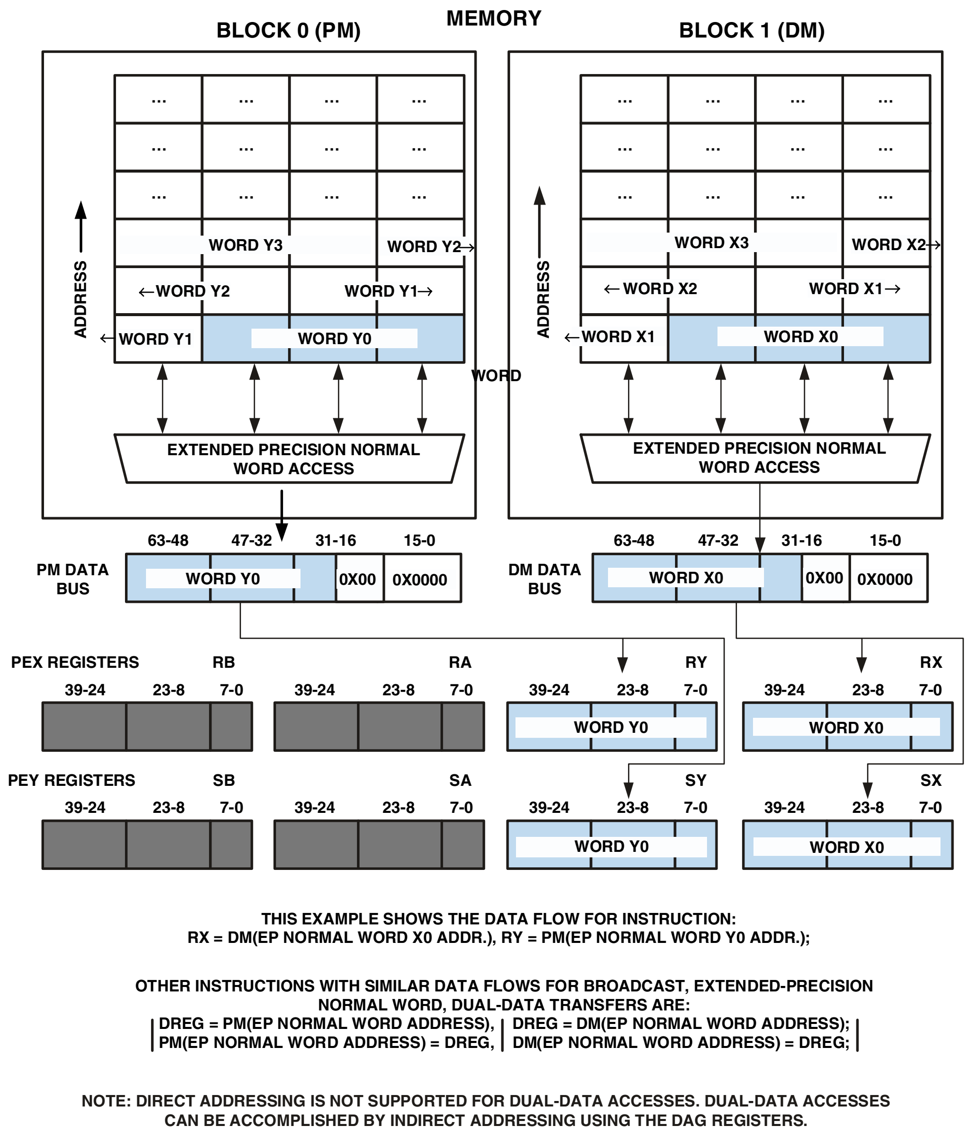

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y3
WORD Y2
WORD Y2
WORD Y1
WORD Y1
WORD Y0
WORD X3
WORD X2
WORD X2
WORD X1
WORD X1
WORD X0
WORD
EXTENDED PRECISION NORMAL WORD ACCESS
63-48
47-32
31-16
15-0
PM DATA BUS
0X00
0X0000
DM DATA BUS
PEX REGISTERS
RB
RA
RY
RX
PEY REGISTERS
SB
SA
SY
SX
39-24
23-8
7-0
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(EP NORMAL WORD X0 ADDR.), RY = PM(EP NORMAL WORD Y0 ADDR.);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, EXTENDED-PRECISION NORMAL WORD, DUAL-DATA TRANSFERS ARE:
DREG = PM(EP NORMAL WORD ADDRESS), | DREG = DM(EP NORMAL WORD ADDRESS);
PM(EP NORMAL WORD ADDRESS) = DREG, | DM(EP NORMAL WORD ADDRESS) = DREG;
NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-37. Extended Precision Normal Word Addressing of Dual Data in Broadcast Load

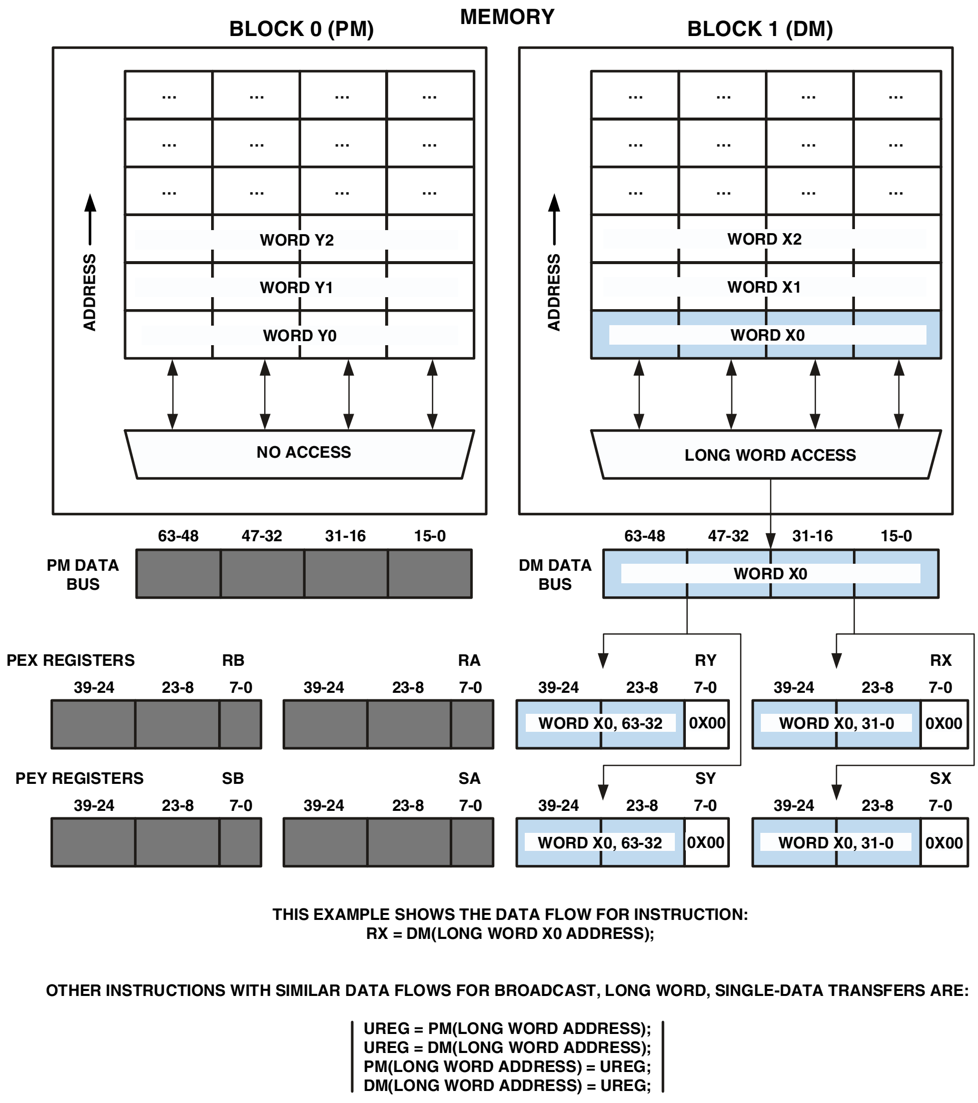
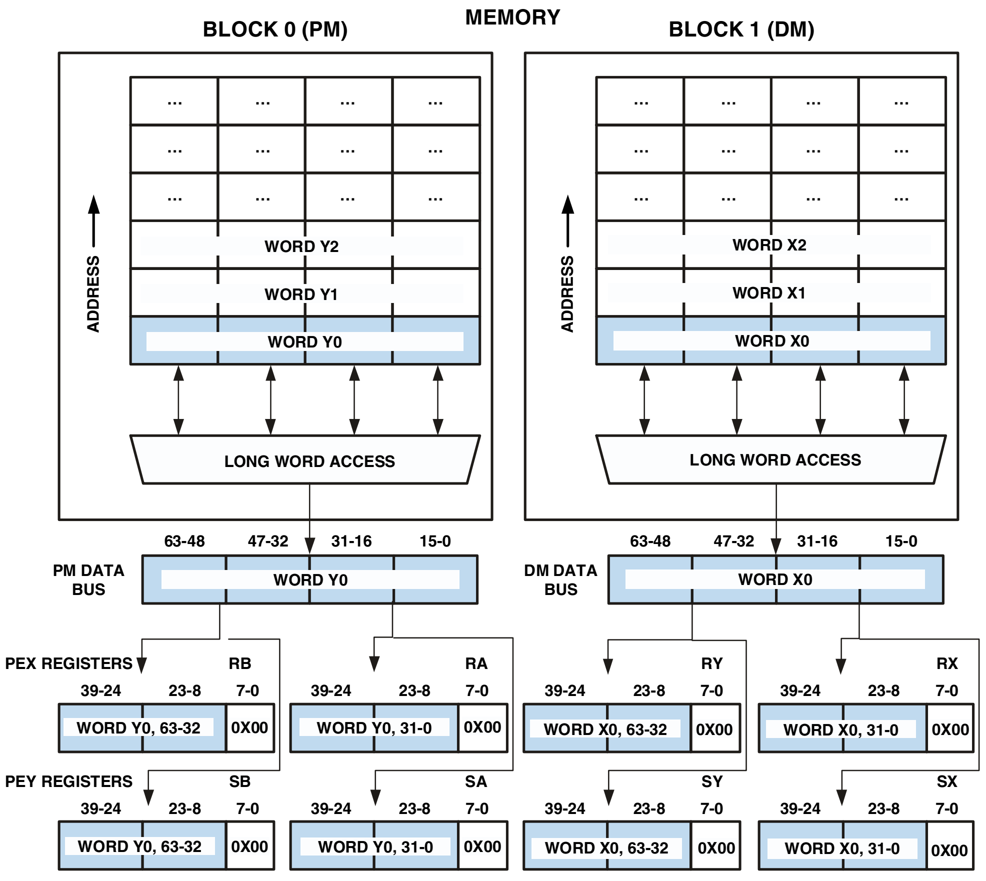

MEMORY
BLOCK 0 (PM)
BLOCK 1 (DM)
ADDRESS
WORD Y2
WORD Y1
WORD Y0
WORD X2
WORD X1
WORD X0
NO ACCESS
LONG WORD ACCESS
PM DATA BUS
DM DATA BUS
63-48
47-32
31-16
15-0
PEX REGISTERS
PEY REGISTERS
RB
RA
RY
RX
SB
SA
SY
SX
39-24
23-8
7-0
WORD X0, 63-32
WORD X0, 31-0
0X00
THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS);
OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, LONG WORD, SINGLE-DATA TRANSFERS ARE:
UREG = PM(LONG WORD ADDRESS);
UREG = DM(LONG WORD ADDRESS);
PM(LONG WORD ADDRESS) = UREG;
DM(LONG WORD ADDRESS) = UREG;

Figure 5-38. Long Word Addressing of Single Data in Broadcast Load



THIS EXAMPLE SHOWS THE DATA FLOW FOR INSTRUCTION:
RX = DM(LONG WORD X0 ADDRESS), RA = PM(LONG WORD Y0 ADDRESS);

OTHER INSTRUCTIONS WITH SIMILAR DATA FLOWS FOR BROADCAST, LONG WORD, DUAL-DATA TRANSFERS ARE:

DREG = PM(LONG WORD ADDRESS),	DREG = DM(LONG WORD ADDRESS);
PM(LONG WORD ADDRESS) = DREG,	DM(LONG WORD ADDRESS) = DREG;

NOTE: DIRECT ADDRESSING IS NOT SUPPORTED FOR DUAL-DATA ACCESSES. DUAL-DATA ACCESSES CAN BE ACCOMPLISHED BY INDIRECT ADDRESSING USING THE DAG REGISTERS.

Figure 5-39. Long Word Addressing of Dual Data in Broadcast Load

Shadow Write FIFO Considerations in SIMD Mode

The shadow write FIFOs is located between the internal memory array of the ADSP-21161 and core and the IOP busses that access the memory.

When performing SIMD reads that cross long word address boundaries and the data read resides in the shadow write FIFO, the read in SIMD mode causes unpredictable results for explicit accesses of odd normal word addresses in internal memory. The implicit part of this SIMD mode transfer incorrectly accesses the previous sequential even address when the data is in the shadow write FIFO.

When the read data resides in internal memory, a SIMD mode explicit access to normal word address 0x40001 results in an implicit access to the next sequential even address value. As shown in Table 5-11, a SIMD mode explicit access to normal word address 0x40001 result in an implicit access to normal word address 0x40002.

Table 5-11. Data Resides In Internal Memory

	Explicit "R0" R0=dm(I0,M0);		**Explicit "S0" S0=dm(I0,M0);**	
Explicit Address (I0)	**R0**	**S0**	**R0**	**S0**
0x40001	32-bit word at 0x40001	32-bit word at 0x40002	32-bit word at 0x40002	32-bit word at 0x40001

Table 5-12 illustrates operation when the previously written data still resides in the shadow write FIFO. For example, from a previous memory write instruction. A SIMD mode explicit access to normal word address 0x40001 results in an implicit access to normal word address 0x40000 if the reading of the data from 0x40001 occurs while the data is still in the shadow write FIFO. This access type results in an implicit access to the next sequential even address value.

Table 5-12. Data Resides In Shadow Write FIFO

	Explicit "R0" R0=dm(I0,M0);		**Explicit "S0" S0=dm(I0,M0);**	
Explicit Address (I0)	**R0**	**S0**	**R0**	**S0**
0x40001	32-bit word at 0x40001	32-bit word at 0x40000	32-bit word at 0x40000	32-bit word at 0x40001

To better demonstrate what results if the read data is in the shadow write FIFO versus internal memory, Table 5-13 shows the failing cases for a SIMD shadow aligned and non-aligned access when a SIMD read immediately follows a SIMD write.

Table 5-13. SIMD Write - SIMD Read Illegal Cases

Address of Write Data in Shadow Write FIFO	**Immediate Read after Write**	**Result**	**Resultant Register Address Contents**
0x50001[1]	r0=dm(0x50000)	Incorrect	r0=(0x50002), s0=(0x50001)
	r0=dm(0x50001)	Correct	r0=(0x50001), s0=(0x50002)
	r0=dm(0x50002)	Incorrect	r0=(0x50002)[2], s0=(0x50003)
0x50002[1]	r0=dm(0x50001)	Incorrect	r0=(0x50001), s0=(0x50002)[2]
	r0=dm(0x50002)	Correct	r0=(0x50002), s0=(0x50003)
	r0=dm(0x50003)	Incorrect	r0=(0x50003), s0=(0x50002)
0xA0002 [3]	r0=dm(0xA0000)	Incorrect	r0=(0xA0004), s0=(0xA0002)
	r0=dm(0xA0001)	Correct	r0=(0xA0001), s0=(0xA0003)
	r0=dm(0xA0002)	Correct	r0=(0xA0002), s0=(0xA0004)

Table 5-13. SIMD Write - SIMD Read Illegal Cases (Cont'd)

Address of Write Data in Shadow Write FIFO	Immediate Read after Write	Result	Resultant Register Address Contents
	r0=dm(0xA0003)	Correct	r0=(0xA0003), s0=(0xA0005)
	r0=dm(0xA0004)	Incorrect	r0=(0xA0004)[2], s0=(0xA0006)
0xA0003[3]	r0=dm(0xA0001)	Incorrect	r0=(0xA0005), s0=(0xA0003)
	r0=dm(0xA0002)	Correct	r0=(0xA0002), s0=(0xA0004)
	r0=dm(0xA0003)	Correct	r0=(0xA0003), s0=(0xA0005)2
	r0=dm(0xA0004)	Correct	r0=(0xA0004), s0=(0xA0006)
	r0=dm(0xA0005)	Incorrect	r0=(0xA0005)[2], s0=(0xA0007)
0xA0004[3]	r0=dm(0xA0002)	Incorrect	r0=(0xA0002), s0=(0xA0004)2
	r0=dm(0xA0003)	Correct	r0=(0xA0003), s0=(0xA0005)
	r0=dm(0xA0004)	Correct	r0=(0xA0004), s0=(0xA0006)
	r0=dm(0xA0005)	Correct	r0=(0xA0005), s0=(0xA0007)
	r0=dm(0xA0006)	Incorrect	r0=(0xA0006)[2], s0=(0xA0004)
0xA0005[3]	r0=dm(0xA0003)	Incorrect	r0=(0xA0003), s0=(0xA0005)2
	r0=dm(0xA0004)	Correct	r0=(0xA0004), s0=(0xA0006)
	r0=dm(0xA0005)	Correct	r0=(0xA0005), s0=(0xA0007)
	r0=dm(0xA0006)	Correct	r0=(0xA0006), s0=(0xA0008)
	r0=dm(0xA0007)	Incorrect	r0=(0xA0007), s0=(0xA0005)
0x28001	r0=dm(0x50000)	Correct	r0=(0x50000), s0=(0x50001)
	r0=dm(0x50001)	Incorrect	r0=(0x50001), s0=(0x50002)2
	r0=dm(0x50002)	Correct	r0=(0x50002), s0=(0x50003)
	r0=dm(0x50003)	Incorrect	r0=(0x50003), s0=(0x50002)
	r0=dm(0x50004)	Correct	r0=(0x50004), s0=(0x50005)
0x28001	r0=dm(0xA0001)	Correct	r0=(0xA0001), s0=(0xA0003)

Table 5-13. SIMD Write - SIMD Read Illegal Cases (Cont'd)

Address of Write Data in Shadow Write FIFO	Immediate Read after Write	Result	Resultant Register Address Contents
	r0=dm(0xA0002)	Incorrect	r0=(0xA0002), s0=(0xA0004)2
	r0=dm(0xA0003)	Incorrect	r0=(0xA0003), s0=(0xA0005)2
	r0=dm(0xA0004)	Correct	r0=(0xA0004), s0=(0xA0006)
	r0=dm(0xA0005)	Correct	r0=(0xA0005), s0=(0xA0007)
	r0=dm(0xA0006)	Incorrect	r0=(0xA0006), s0=(0xA0004)
	r0=dm(0xA0007)	Incorrect	r0=(0xA0007), s0=(0xA0005)
	r0=dm(0xA0008)	Correct	r0=(0xA0008), s0=(0xA000A)
0x50002	r0=dm(0x28000)	Correct	r0=(0x50000), r1=(0x50001)
	r0=dm(0x28001)	Correct	r0=(0x50002), r1=(0x50003)
	r0=dm(0x28002)	Correct	r0=(0x50004), r1=(0x50005)
0x50003	r0=dm(0x28000)	Correct	r0=(0x50000), r1=(0x50001)
	r0=dm(0x28001)	Incorrect	r0=(0x50004), r1=(0x50003)
	r0=dm(0x28002)	Incorrect	r0=(0x50004)[2], r1=(0x50005)
	r0=dm(0x28003)	Correct	r0=(0x50006), r1=(0x50007)
0x50002	r0=dm(0xA0001)	Correct	r0=(0xA0001), s0=(0xA0003)
	r0=dm(0xA0002)	Incorrect	r0=(0xA0002), s0=(0xA0004)2
	r0=dm(0xA0003)	Incorrect	r0=(0xA0003), s0=(0xA0005)2
	r0=dm(0xA0004)	Correct	r0=(0xA0004), s0=(0xA0006)
	r0=dm(0xA0005)	Correct	r0=(0xA0005), s0=(0xA0007)
	r0=dm(0xA0006)	Incorrect	r0=(0xA0006), s0=(0xA0004)
	r0=dm(0xA0007)	Incorrect	r0=(0xA0007), s0=(0xA0005)
	r0=dm(0xA0008)	Correct	r0=(0xA0008), s0=(0xA000A)
0xA0004	r0=dm(0x28000)	Correct	r0=(0x50000), r1=(0x50001)

Table 5-13. SIMD Write - SIMD Read Illegal Cases (Cont'd)

Address of Write Data in Shadow Write FIFO	Immediate Read after Write	Result	Resultant Register Address Contents
	r0=dm(0x28001)	Correct	r0=(0x50002), r1=(0x50003)
	r0=dm(0x28002)	Correct	r0=(0x50004), r1=(0x50005)
0xA0006	r0=dm(0x28000)	Correct	r0=(0x50000), r1=(0x50001)
	r0=dm(0x28001)	Incorrect	r0=(0x50002)[4], r1=(0x50003)
	r0=dm(0x28002)	Incorrect	r0=(0x50004)[2], r1=(0x50005)
	r0=dm(0x28003)	Correct	r0=(0x50006), r1=(0x50007)
0xA0004	r0=dm(0x50000)	Correct	r0=(0x50000), s0=(0x50001)
	r0=dm(0x50001)	Incorrect	r0=(0x50001), s0=(0x50002)2
	r0=dm(0x50002)	Correct	r0=(0x50002), s0=(0x50003)
	r0=dm(0x50003)	Incorrect	r0=(0x50003), s0=(0x50002)4
	r0=dm(0x50004)	Correct	r0=(0x50004), s0=(0x50005)

1 Normal word accesses
2 Old data from memory is accessed instead of new data in Shadow Write FIFO
3 Short word accesses
4 PEx and PEy data is partly from shadow and partly from memory

If the new written data resides in shadow write FIFO, then for normal and short word SIMD accesses, a write access to an even address followed by a read access to the adjacent (higher or lower) odd address results in incorrect SIMD access operation. Similarly, a write access to an odd address followed by a read access to the adjacent (higher or lower) even address results in incorrect SIMD access operation.

To prevent unexpected SIMD read results when a write is followed by a read from the same long word boundary addresses, two options are recommended. These two suggestions are independent of one another and can be used to work around the SIMD shadow write FIFO.

- Align all variables and arrays in memory to long word address boundaries using the `.ALIGN` assembler directive. Do not explicitly access odd normal word addresses or non-long word boundary aligned short word addresses in SIMD mode. Note that for program generated addresses which are odd, you cannot use the `.ALIGN` workaround. For example, this workaround cannot be used for indirect addressing using the index or pointer DAG registers.

  OR

- Include two `NOPs` or non-memory access instructions to clear the shadow write FIFO.

Arranging Data in Memory

Each processor's access to internal memory gets data from 4-columns (long, word) or 3-columns (instruction or extended-precision normal word), 2-column (normal word), or 1-column (short word) memory location. For more information on how the processor accesses 4- or 3-column data, see "Memory Organization and Word Size" on page 5-25.

To take advantage of the processor's data accesses to 4- and 3-column locations, programs must adjust the interleaving of data into memory locations to accommodate the memory access mode.

The following guidelines provide an overview of how programs should interleave data in memory locations. For more information and examples, see the *ADSP-21160 SHARC DSP Instruction Set Reference*:

- Programs can use odd or even modify values (1, 2, 3, …) to step through a buffer in **single-or dual-data, SISD or Broadcast load mode** regardless of the data word size (long word, extended-precision normal word, normal word, or short word).
- Programs should use multiple of 4 modify values (4, 8, 12, …) to step through a buffer of **short word data in single-or dual-data, SIMD mode.** Programs must step through a buffer twice, once for addressing even short word addresses and once for addressing odd short word addresses.
- Programs should use multiple of 2 modify values (2, 4, 6, …) to step through a buffer of **normal word data in single- or dual-data SIMD mode.**
- Programs can use odd or even modify values (1, 2, 3, …) to step through a buffer of **long word or extended-precision normal word data in single- or dual-data, SIMD mode**.

Executing Instructions From External Memory

The ADSP-21161 supports the execution of 48-bit wide program instructions from external memory devices of various widths. The processor can transparently pack and execute 8-bit, 16-bit or 32-bit external memory or execute 48-bit non-packed instructions. This requires that instructions be packed into external memory in a way that differs from the normal packing modes that exist for DMA accesses or host accesses.

This automatic instruction packing is performed only when the Program Sequencer initiates an external access to fetch an instruction with one of four instruction packing modes enabled in the SYSCON register: 8- to 48-bit, 16- to 48-bit, 32- to 48-bit or 48- to 48-bit packing.

Note that the processor only supports program execution from external memory bank 0.

The default packing mode the ADSP-21161 processor is 32- to 48-bit packing. Packed instruction execution for 8-, 16-, 32-, or 48-bit wide external memory is also supported and controlled by the IPACK[1:0] bits of the SYSCON register. Table 5-14 summarizes the packing mode configurations controlled by IPACK[1:0] bits.

There is a no packing 48-bit bus width mode available on the processor which assumes the EPD bus is 48 bits wide. This full instruction width execution from external memory is made possible by multiplexing 16 link port pins with DATA[15:0] enabling the program execution to run at full-rate. These additional 16 data lines should only be enabled when the link ports are not used. Data lines DATA[15:8] multiplex with L1DAT[7:0] and data lines DATA[7:0] multiplex with L0DAT[7:0]. Set the IPACK bits [1:0] of the SYSCON register to 01 in order to enable DATA[15:0] pins for a 48-bit wide external bus.

There are four boot and one no boot modes available on the processor. In the no-boot mode, the processor fetches instructions using a 32- to 48-bit packing. In a boot mode, the packing mode can be changed by writing the new execution packing mode to the IPACK bits before a fetch from external memory occurs. A host can write the new values into the processor or the software loader kernel can change the values during booting.

Table 5-14. External Instruction Execution Packing Modes

IPACK1	IPACK0	Packing Mode Description
0	0	32- to 48-bit packed instruction execution
0	1	Full 48-bit instruction execution / No-Pack mode (DATA[15:0] enabled) with unused L1DAT[7:0] and L0DAT[7:0].
1	0	16- to 48-bit packed instruction execution
1	1	8- to 48-bit packed instruction execution

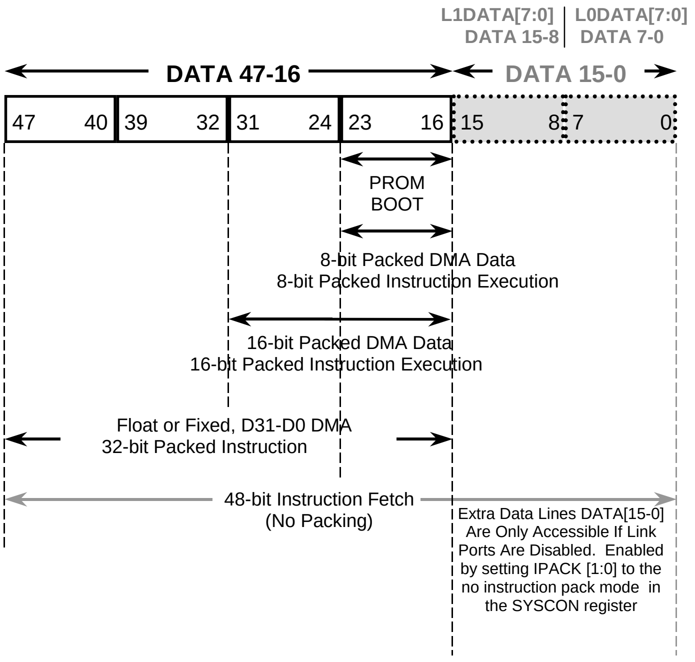

L1DATA[7:0]
DATA 15-8
L0DATA[7:0]
DATA 7-0
DATA 47-16
DATA 15-0
47
40
39
32
31
24
23
16
15
8
7
0
PROM BOOT
8-bit Packed DMA Data
8-bit Packed Instruction Execution
16-bit Packed DMA Data
16-bit Packed Instruction Execution
Float or Fixed, D31-D0 DMA
32-bit Packed Instruction
48-bit Instruction Fetch (No Packing)
Extra Data Lines DATA[15-0] Are Only Accessible If Link Ports Are Disabled. Enabled by setting IPACK [1:0] to the no instruction pack mode in the SYSCON register

Figure 5-40. ADSP-21161 External Data Alignment Options

When writing to bits 30 and 31(IPACK[1:0]) in the SYSCON register to enable the packed instruction mode, delay the instruction fetch from external memory by two instructions. This can be done by inserting two NOPs after a write to SYSCON register or by following the execution sequence shown in the code segment.

```
ext_isr_tabl_seg_dma10:
  jump int_codeaddr (db);
  ustatx = 0x80000000 ; /* change packing from 32-48 to 16-48 */
  dm(syscon) = ustatx;
 int_codeaddr:
  jump ext_codeaddr (db);
  ustatx = new_wait_value;
  dm(WAIT) = ustatx;
```

The following tables show the addresses for instructions packed in two, three or six consecutive locations in external memory:

- "48- to 48-Bit External Instruction Packing" on page 5-104
- "32- to 48-Bit External Instruction Packing" on page 5-105
- "16- to 48-Bit External Instruction Packing" on page 5-106
- "8- to 48-Bit External Instruction Packing" on page 5-107

For more information on instruction packing in external memory, see the *VisualDSP++ User's Guide for ADSP-21xxx Family DSPs*.

Table 5-15. 48- to 48-Bit External Instruction Packing

ADDRESS	DATA[47:0]
0x200000	Instr0[47:0]
0x200001	Instr1[47:0]
0x200002	Instr2[47:0]

Table 5-15. 48- to 48-Bit External Instruction Packing (Cont'd)

ADDRESS	DATA[47:0]
0x200003	Instr3[47:0]
0x200004	. . . .

For 48- to 48-bit full instruction width packing, the processor stores one instruction in every 48-bit word memory location. In this packing mode, no address translation is performed by the program sequencer. Instructions are executed from SDRAM at the core clock rate. By enabling `IPACK[1:0]`, the link port data pins `L1DAT[7:0]` and `L0DAT[7:0]` are activated as `DATA[15:0]`.

Table 5-16. 32- to 48-Bit External Instruction Packing

ADDRESS	DATA[47: 2]	DATA[31:16]
0x200000	Instr0[47:16]	
0x200001		Instr0[15:0]
0x200002	Instr1[47:16]	
0x200003		Instr1[15:0]
0x200004	. . . .	

For 32- to 48-bit instruction packing, the processor stores an instruction in two consecutive memory locations. In this packing mode, the first 32 bits of the 48-bit instruction are stored in an even location and the lower 16 bits of the 48-bit opcode are stored in the adjacent odd location in memory. The program sequencer automatically generates the correct external addresses based on the `IPACK` bits in the `SYSCON` register. The program sequencer generates addresses in groups of two physical locations.

To generate a corresponding address in external memory for the second part of the 48-bit instruction, the processor increments the internal logical address of the previous access by 1.

Table 5-17. 16- to 48-Bit External Instruction Packing

ADDRESS	DATA[31:16]
0x200000	Instr0[47:32]
0x200001	Instr0[31:16]
0x200002	Instr0[15:0]
0x200003	Unused Memory Space
0x200004	Instr1[47:32]
0x200005	Instr1[31:16]
0x200006	Instr1[15:0]
0x200007	Unused Memory Space

Similarly, for 16- to 48-bit instruction packing, the first 16 bits are stored at an even address and the remaining 16 bit segments are stored in consecutive locations. The program sequencer generates addresses in groups of four physical locations. For the remaining accesses, the previous internal logical address is incremented by 1. However, this leaves an unused 16-bit location after every three 16-bit valid instruction segments in the external memory. For example, the three 16 bit segments may be placed at 0x0200000, 0x0200001 and 0x0200002 respectively. The next instruction sixteen bit segments should be placed from address 0x200004 to 0x200007 and so on.

Table 5-18. 8- to 48-Bit External Instruction Packing

Address	DATA[23:16]
0x200000	Instr0[47:40]
0x200001	Instr0[39:32]
0x200002	Instr0[31:24]
0x200003	Instr0[23:16]
0x200004	Instr0[15:8]
0x200005	Instr0[7:0]
0x200006	Unused Memory Space
0x200007	Unused Memory Space
0x200008	Instr1[47:40]
0x200009	Instr1[39:32]
0x20000A	Instr1[31:24]
0x20000B	Instr1[23:16]
0x20000C	Instr1[15:8]
0x20000D	Instr1[7:0]
0x20000E	Unused Memory Space
0x20000F	Unused Memory Space

For 8- to 48-bit instruction packing, the first 8 bits are stored at an even address and the remaining 8-bit segments are stored in consecutive locations. The program sequencer generate addresses in groups of eight physical locations. For the remaining accesses, the previous internal logical address is incremented by 1. However, this leaves two unused 8-bit locations after every six 8-bit internal logical segments in the external memory. For example, the six 8-bit segments may be placed at 0x0200000, 0x0200001, 0x0200002, 0x0200003, 0x0200004 and 0x0200005 respectively. The next instruction eight bit segments should be placed from address 0x200008 to 0x20000D and so on.

In 32- to 48-bit packing mode, each access of external memory to fetch an instruction translates into two accesses to successive locations. In 16- to 48-bit packing mode, each access of external memory to fetch an instruction translates into three accesses to successive locations. In 8- to 48-bit packing mode, each access of external memory to fetch an instruction translates into six accesses to successive locations.

The processor core speed for instruction execution is affected by the type of external memory (SDRAM or non-SDRAM) and external memory width.

- For packed execution modes of 32- to 48-bit, 16- to 48-bit and 8- to 48-bit, with the SDCKR bit in the SDCTL register set (=1) and the program executing from SDRAM, the core instruction rate is 2, 3 or 6 times slower than executing from internal memory.
- When SDCKR=0, the core instruction rate is 4, 6 or 12 times slower. If the program is executing from SRAM or FLASH with a CLKIN-core clock ratio of 2:1, the core speed is reduced by the number of waitstates and a factor of 4, 6 or 12.

The effect of external memory accesses on core speed is shown in Table 5-19.

Table 5-19. External Memory Width Versus Core Speeds

External Memory Width	SDRAM		Non SDRAM (FLASH, SRAM, SBSRAM) for CLKIN-core clock ratio of 2:1
	SDCKR = 1	SDCKR = 0	
8-bit	Core Speed ÷ 6	Core Speed ÷ 12	Core Speed ÷ 12 x number of waitstates
16-bit	Core Speed ÷ 3	Core Speed ÷ 6	Core Speed ÷ 6 x number of waitstates
32-bit	Core Speed ÷ 2	Core Speed ÷ 4	Core Speed ÷ 4 x number of waitstates
48-bit	Core Speed	Core Speed ÷ 2	Core Speed ÷ 2 x number of waitstates

In summary, instruction access to external memory translate to one (full 48-bit data bus mode), two, three, or six accesses to successive locations depending on the instruction packing mode selected in bits 30 and 31 in the `SYSCON` register.

For 16- to 48-bit packing, one external address space (two bytes) is unused for every single instruction. Similarly, for 8- to 48-bit packing two external address spaces (two bytes) are unused for every single instruction. For 32- to 48-bit packing, every external address contains valid data. The next sections examine the addressing schemes and unused addresses for all three packing mode cases.

32- to 48-Bit Packing Address Generation Scheme

To generate a corresponding address in external memory for the first part of the instruction, the processor left-shifts the lower bits [19:0] to generate [20:1] bits (`ADDR20-0`)in external memory, while the processor leaves bits [23:21] unaltered.

`ADDR[0]` is 0 for the first access and 1 for the second in the case of operating in 32- to 48-bit packing mode. In this way, internal address 0x200000 on the PM address bus aligns with the beginning of external memory at address 0x200000.

Table 5-20. Address Generation Scheme for 32- to 48-bit Packing[1]

Segment	PM ADDR Bus	External Address - ADDR23-0
Seg 1	0x0200000 0x0200001 0x0200002 .... 0x02FFFFF	0x0200000/1 0x0200002/3 0x0200004/5 .... 0x03FFFFE/F
Seg 2	0x0400000 0x0400001 0x0400002 .... 0x04FFFFF	0x0400000/1 0x0400002/3 0x0400004/5 .... 0x05FFFFE/F
Seg 3	0x0600000 0x0600001 0x0600002 ...... 0x06FFFFF	0x0600000/1 0x0600002/3 0x0600004/5 ...... 0x07FFFFE/F

1 Note that segmented internal address ranges allows continuous addresses in external memory for 48- to 32-bit packing.

Total Program Size (32- to 48-Bit Packing)

Total external memory available is 14 Mwords (non-SDRAM) and 62 Mwords (SDRAM). Given that one instruction takes two external memory locations, the external program memory is 7 Mwords non-SDRAM space and 31 Mwords SDRAM space. This scheme limits the size of the contiguous program segment (internal) to 1 Mword. There are seven of these segments in bank 0 non-SDRAM space and 30 segments in bank 0 SDRAM space. See Table 5-22 on page 5-112 for a comparison of total program sizes based on different packing modes.

16- to 48-Bit Packing Address Generation Scheme

For a 16- to 48-bit packing the lower [18:0] bits of the address are left shifted by two positions to generate [20:2] bits of `ADDR` (address in external memory) while bits [23:21] are unaltered. `ADDR1-0` is 00 for the first access and 01 for the next access and 10 for the third access.

Table 5-21. Address Generation Scheme for 16- to 48-Bit Packing

Segment	PM ADDR Bus	External Address - ADDR23-0
Seg 1	0x0200000 0x0200001 0x0200002 ...... 0x027FFFF	0x0200000/1/2 0x0200004/5/6 0x0200008/9/A 0x03FFFFC/D/E
Seg 2	0x0400000 0x0400001 0x0400002 ...... 0x047FFFF	0x0400000/1/2 0x0400004/5/6 0x0400008/9/A 0x05FFFFC/D/E
Seg 3	0x0600000 0x0600001 0x0600002 ...... 0x067FFFF	0x0600000/1/2 0x0600004/5/6 0x0600008/9/A 0x07FFFFC/D/E
	......	......

Total Program Size (16- to 48-Bit Packing)

Total external memory available is 14 Mwords (non-SDRAM) and 62 Mwords (SDRAM). Given that one instruction takes four external memory locations, the external program memory is 3.5 Mwords non-SDRAM space and 15.5 Mwords SDRAM space. This scheme limits the size of the contiguous program segment (internal) to 0.5M. There are seven of these segments in bank 0 non-SDRAM space and 30 segments in bank 0 SDRAM space. See Table 5-23 on page 5-113 for a comparison of total program sizes based on different packing modes.

8- to 48-Bit Packing Address Generation Scheme

Similarly, for a 8- to 48-bit packing the lower [17:0] bits of the address are left shifted by three positions to generate [20:3] bits of `ADDR` while bits [23:21] are unaltered. This way internal address 0x200000 aligns with the beginning of external memory at 0x200000. However, this sort of execution packing gives variable maximum program lengths in external memory for different packing.

Table 5-22. Address Generation Scheme for 8- to 48-Bit Packing

Segment	PM ADDR Bus	External Address - ADDR23-0
Seg 1	0x0200000 0x0200001 0x0200002 ..... 0x023FFFF	0x0200000/1/2/3/4/5 0x0200008/9/A/B/C/D 0x0200010/1/2/3/4/5 ..... 0x03FFFF8/9/A/B/C/D
Seg 2	0x0400000 0x0400001 0x0400002 ...... 0x043FFFF	0x0400000/1/2/3/4/5 0x0400008/9/A/B/C/D 0x0400010/1/2/3/4/5 .... 0x05FFFF8/9/A/B/C/D
Seg 3	0x0600000 0x0600001 0x0600002 ..... 0x063FFFF	0x0600000/1/2/3/4/5 0x0600008/9/A/B/C/D 0x0600010/1/2/3/4/5 .... 0x07FFFF8/9/A/B/C/D
	......	......

Total Program Size (8- to 48-Bit Packing)

Total external memory available is 14 Mwords (non-SDRAM) and 62 Mwords (SDRAM). Given that one instruction takes eight external memory locations, the external program memory is 1.75 Mwords non-SDRAM space and 7.75 Mwords SDRAM space. This scheme limits the size of the contiguous program segment (internal) to 0.25 Mwords. There are seven of these segments in bank 0 non-SDRAM space and 30 segments in bank 0 SDRAM space.

No Packing (48- to 48-Bit) Address Generation Scheme

In no-packing 48- to 48-bit mode, execution at full-rate is supported and the size of the external program memory can be 14 Mwords non-SDRAM space or 62 Mwords SDRAM space. No packing is performed for data accesses to external memory.

Table 5-23. Total Program Size Comparison

	48- to 48-bit (Mwords)	32- to 48-bit (Mwords)	16- to 48-bit (Mwords)	8- to 48-bit (Mwords)
SRAM	14	7	3.5	1.75
SDRAM	62	31.34	15.67	7.83

6 I/O PROCESSOR

The processor's I/O processor manages Direct Memory Accessing (DMA) of processor memory through the external, SPI, link, and serial ports. Each DMA operation transfers an entire block of data. By managing DMA, the I/O processor lets programs move data as a background task while using the processor core for other processor operations. The I/O processor's architecture supports a number of DMA operations. These operations include the following transfer types:

Internal memory ↔ external memory or external peripherals

- Internal memory ↔ internal memory of other processors
- Internal memory ↔ host processor
- Internal memory ↔ serial port I/O
- Internal memory ↔ link port I/O
- Internal memory ↔ SPI I/O
- External memory ↔ external peripherals

This chapter describes the I/O processor and how it controls external port, link port, SPI port, and serial port operations.

DMA transfers between internal memory and external memory, multiprocessor memory, or a host use the processor's external port. For these types of transfers, a program provides the DMA controller with the internal memory buffer size and address, the address modifier, and the direction of transfer. After setup, the DMA transfers begin when the program enables the channel and continues until the I/O processor transfers the entire buffer to or from processor memory.

Similarly, DMA transfers between internal memory and link, serial, or SPI port have DMA parameters. When the I/O processor performs DMA between internal memory and one of these ports, the program sets up the parameters, and the I/O uses the port instead of the external bus.

The direction (receive or transmit) of the I/O port determines the direction of data transfer. When the port receives data, the I/O processor automatically transfers the data to internal memory. When the port needs to transmit a word, the I/O processor automatically fetches the data from internal memory.

The I/O processor also lets the processor system perform DMA transfers between an external device and external memory. This external to external transfer only uses the external port and I/O processor. External devices can control external port DMA transfers in two ways. If the external device can handle bus mastership, the external device master reads or writes to DMA buffers on the processor. External devices also can assert a DMA Request input ($\overline{\texttt{DMARx}}$) to request service.

To further minimize loading on the processor core, the I/O processor supports chained DMA operations. When using chained DMA, a program initiates a DMA transfer to automatically set up and start the next DMA transfer after the current one completes.

External bus packing and unpacking of 16-, 32-, 48-, or 64- bit words in internal memory is performed during DMA transfers from either 8-, 16-, or 32- bit wide external memory. Fourteen channels of DMA are available on the ADSP-21161; two channels are shared between the SPI interface and the link ports, eight channels are available via the serial ports, and four channels are available via the processor's external port for host processor, other ADSP-21161s, memory, or I/O transfers. Asynchronous off-chip peripherals can control two DMA channels using DMA

Request/Grant lines (`DMAR1-2`, `DMAG1-2`). Other DMA features include interrupt generation upon completion of DMA transfers and DMA chaining for automatic linked DMA transfers.

For information on connecting external devices to the external port, link ports, SPI port, or serial ports, see "External Port" on page 7-1, "Link Ports" on page 9-1, "Serial Peripheral Interface (SPI)" on page 11-1 or "Serial Ports" on page 10-1.

Figure 6-1 shows the processor's I/O processor, related ports, and buses. Figure 6-5 on page 6-23 shows more detail on DMA channel data paths.

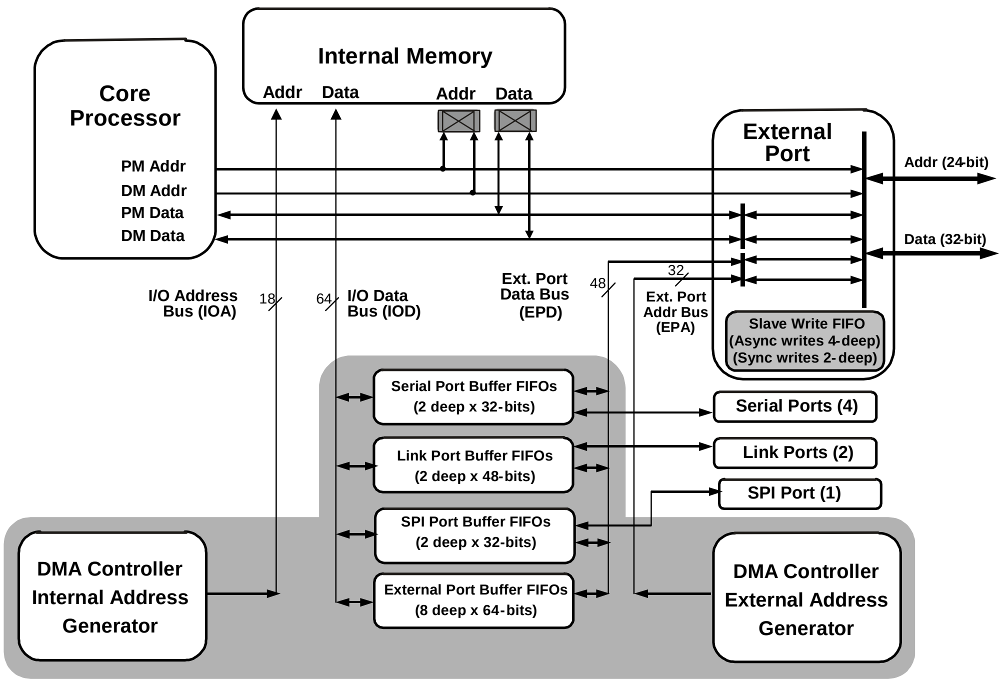

Internal Memory
Addr
Data
Addr
Data
Core Processor
PM Addr
DM Addr
PM Data
DM Data
External Port
Addr (24-bit)
Data (32-bit)
I/O Address Bus (IOA)
18
64
I/O Data Bus (IOD)
Ext. Port Data Bus (EPD)
48
32
Ext. Port Addr Bus (EPA)
Slave Write FIFO (Async writes 4-deep) (Sync writes 2-deep)
Serial Port Buffer FIFOs (2 deep x 32-bits)
Serial Ports (4)
Link Port Buffer FIFOs (2 deep x 48-bits)
Link Ports (2)
SPI Port (1)
SPI Port Buffer FIFOs (2 deep x 32-bits)
External Port Buffer FIFOs (8 deep x 64-bits)
DMA Controller Internal Address Generator
DMA Controller External Address Generator

Figure 6-1. I/O Processor Block Diagram

The Data Buffer Registers column in Figure 6-2 shows the data buffer registers for each port. These registers include:

- **External Port Buffer** (EBPx). These 64-bit buffers for the external port have eight-position FIFOs for transmitting or receiving data when interfacing with a host or external devices such as memory and memory mapped devices.
- **Link Port Buffer** (LBUFx). These buffers for the link ports have two-position FIFOs for transmitting or receiving DMA data when connected to another link port.
- **Serial Port Receive Buffer** (RXx). These receive buffers for the serial ports have two-position FIFOs for receiving data when connected to another serial device.
- **Serial Port Transmit Buffer** (TXx). These transmit buffers for the serial ports have two position FIFOs for transmitting data when connected to another serial device.
- **SPI Receive Buffer** (SPIRX). This receive buffer for the SPI port has two-position FIFOs for receiving data when connected to another serial device.
- **SPI Transmit Buffer** (SPITX). This transmit buffer for the SPI port has two position FIFOs for transmitting data when connected to another serial device.

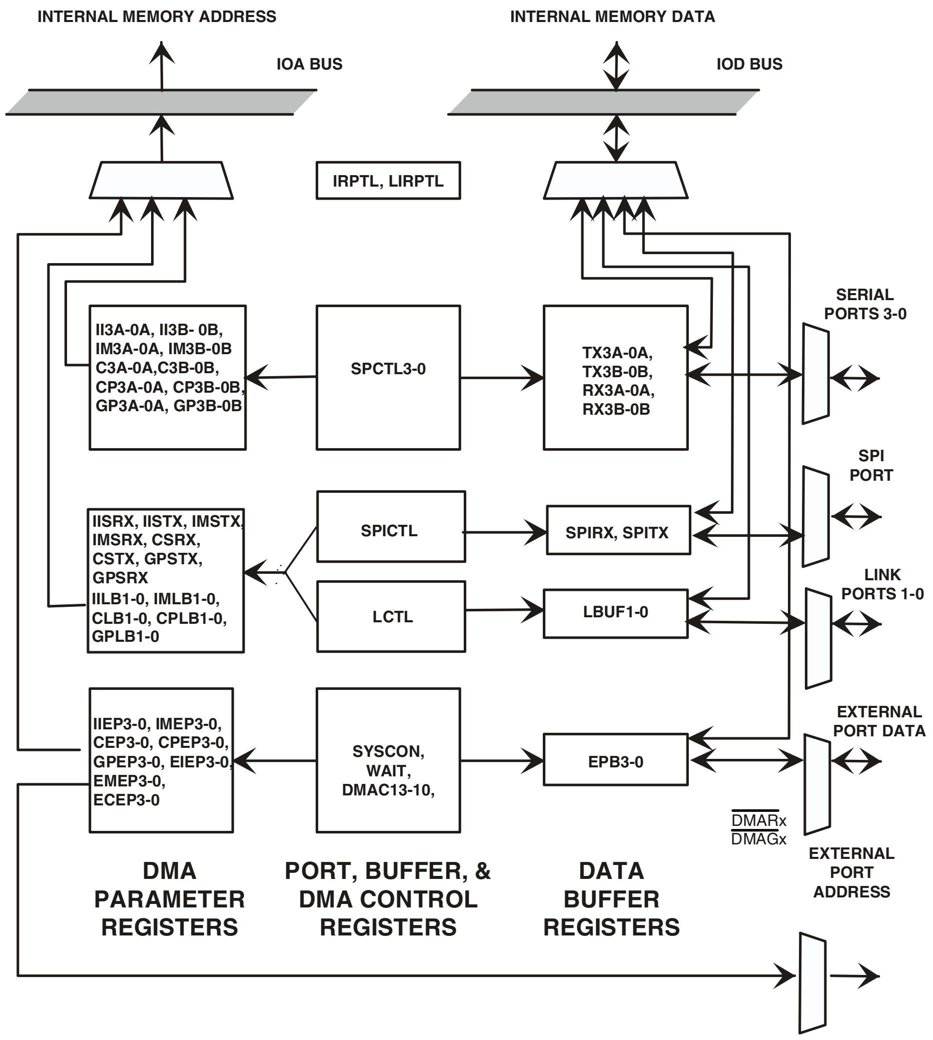

INTERNAL MEMORY ADDRESS
INTERNAL MEMORY DATA
IOA BUS
IOD BUS
IRPTL, LIRPTL
II3A-0A, II3B- 0B, IM3A-0A, IM3B-0B C3A-0A,C3B-0B, CP3A-0A, CP3B-0B, GP3A-0A, GP3B-0B
SPCTL3-0
TX3A-0A, TX3B-0B, RX3A-0A, RX3B-0B
SERIAL PORTS 3-0
SPI PORT
IISRX, IISTX, IMSTX, IMSRX, CSRX, CSTX, GPSTX, GPSRX
IILB1-0, IMLB1-0, CLB1-0, CPLB1-0, GPLB1-0
SPICTL
SPIRX, SPITX
LCTL
LBUF1-0
LINK PORTS 1-0
IIEP3-0, IMEP3-0, CEP3-0, CPEP3-0, GPEP3-0, EIEP3-0, EMEP3-0, ECEP3-0
SYSCON, WAIT, DMAC13-10,
EPB3-0
EXTERNAL PORT DATA
$\overline{\text{DMAR}}$x
$\overline{\text{DMAG}}$x
EXTERNAL PORT ADDRESS
DMA PARAMETER REGISTERS
PORT, BUFFER, & DMA CONTROL REGISTERS
DATA BUFFER REGISTERS

Figure 6-2. I/O Processor Registers

The Port, Buffer, and DMA Control Registers column in Figure 6-2 shows the control registers for the ports and DMA channels. These registers include:

- **System Configuration register** (SYSCON). This register configures packing, priority, and word order for the external port.
- **Waitstate and Access Mode register** (WAIT). This register configures handshake, idle cycle insertion, and waitstate insertion for external memory DMA accesses.
- **External Port DMA Control registers** (DMACx). These control registers for each external port DMA channel select the direction, format, handshake, and enable chaining, transfer mode, and DMA start.
- **Link Port Control register** (LCTL). This control register selects the direction, word width, transfer rate, and enable chaining and DMA start. This register assigns link buffers to link ports for link port operations. This register indicates link buffer packing and error status for link port operations.
- **Serial Port Control registers** (SPCTLx). These control registers for each port select the receive or transmit format, monitor FIFO status, enable chaining, and start DMA.
- **SPI Port Control register** (SPICTL). This control register configures and enables the SPI interface, selects the device as master or slave, and determines the data transfer and word size.

The DMA Parameter Registers column in Figure 6-2 shows the parameter registers for each DMA channel. These registers function similarly to data address generator registers and include:

- **Internal Index registers** (`IIx`). Index registers provide an internal memory address, acting as a pointer to the next internal memory DMA read or write location. These registers are 18 bits wide and are offset 0x40000 for internal addressing in normal word space.
- **Internal Modify registers** (`IMx`). Modify registers provide the signed increment by which the DMA controller post-modifies the corresponding internal memory index register after the DMA read or write. These registers are 16 bits wide.
- **Count registers** (`Cx`). Count registers indicate the number of words remaining to be transferred to or from internal memory on the corresponding DMA channel. These registers are 16 bits wide.
- **Chain Pointer registers** (`CPx`). Chain pointer registers hold the starting address of the Transfer Control Block (parameter register values) for the next DMA operation on the corresponding channel. These registers also control whether the I/O processor generates an interrupt when the current DMA process ends. These registers are 19 bits wide and are offset 0x40000 for internal addressing in normal word space.
- **General Purpose registers** (`GPx`). General purpose DMA registers hold an address or other value. These registers are 17 bits wide.
- **External Index registers** (`EIEPx`). Index registers provide an external memory address, acting as a pointer to the next external memory DMA read or write location. These registers only apply to external port `EPBx` DMA. These External Port DMA registers are 32 bits wide.

- **External Modify registers** (EMEPx). Modify registers provide the increment by which the DMA controller post-modifies the corresponding external memory index register after the DMA read or write. These registers only apply to external port EPBx DMA. These External Port DMA registers are 32 bits wide.
- **External Count registers** (ECEPx). External count registers indicate the number of words remaining to be transferred to or from external memory on the corresponding DMA channel. These registers only apply to external port EPBx DMA. These External Port DMA registers are 32 bits wide.

Figure 6-3 shows a block diagram of the I/O processor's address generator (DMA controller). Table 6-1 lists the parameter registers for each DMA channel. The parameter registers are uninitialized following a processor reset.

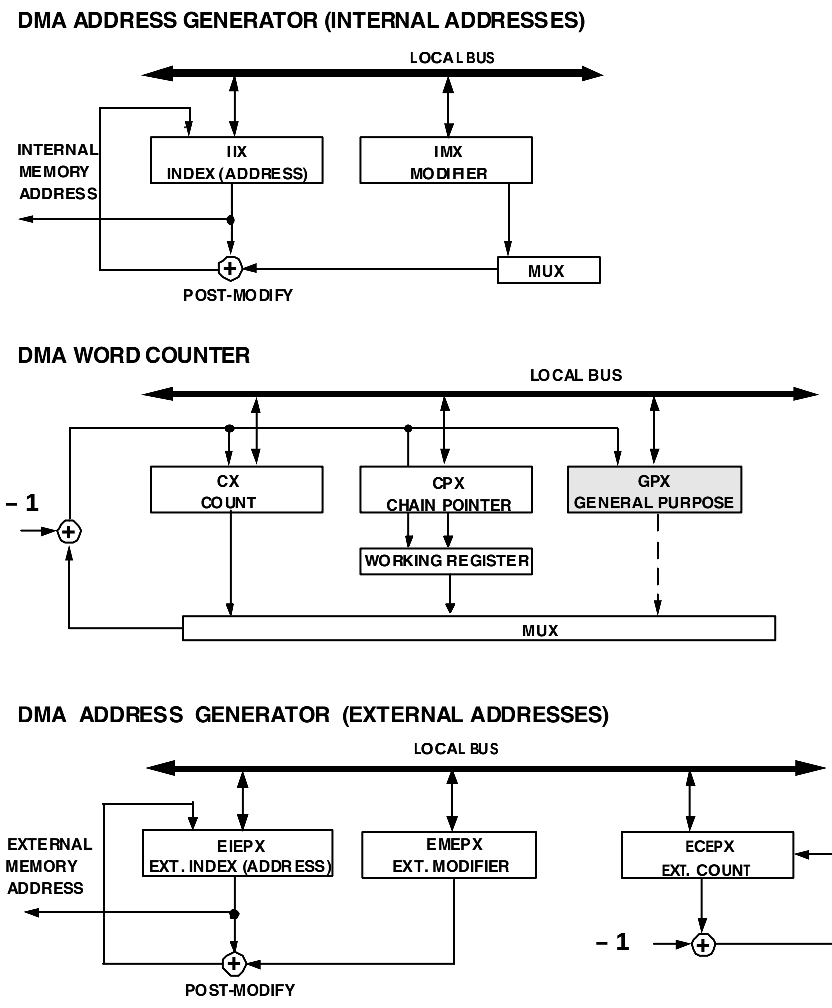
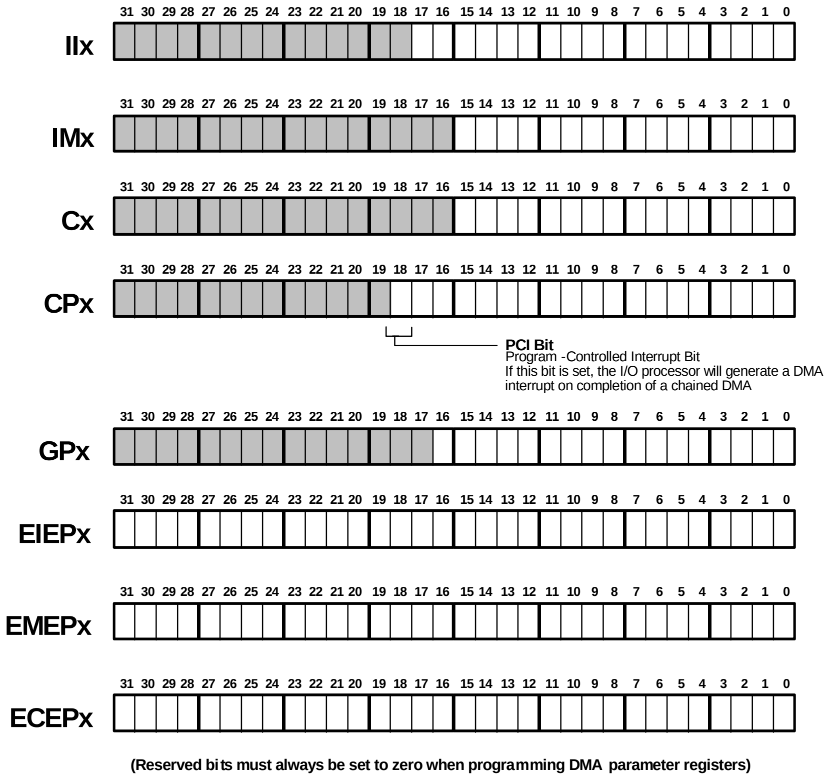


Figure 6-3. DMA Address Generator

The I/O processor generates addresses for DMA channels much the same way that the Data Address Generators (DAGs) generate addresses for data memory accesses. Each channel has a set of parameter registers (shown in Figure 6-4) including an index register (`IIx`) and modify register (`IMx`) that the I/O processor uses to address a data buffer in internal memory. The index register must be initialized with a starting address for the data buffer. As part of the DMA operation, the I/O processor outputs the address in the index register onto the processor's I/O address bus and applies the address to internal memory during each DMA cycle—a clock cycle in which a DMA transfer is taking place.



IIx
IMx
Cx
CPx
PCI Bit
Program -Controlled Interrupt Bit
If this bit is set, the I/O processor will generate a DMA interrupt on completion of a chained DMA
GPx
EIEPx
EMEPx
ECEPx
(Reserved bits must always be set to zero when programming DMA parameter registers)

Figure 6-4. IOP Parameter Registers

All addresses in the index (`IIx`) registers are offset by a value matching the processor's first internal Normal word addressed RAM location, before the I/O processor uses the addresses. For the ADSP-21161, this offset value is 0x0004 0000.

While DMA addresses must always be Normal word (32-bit) memory, the internal memory data transfer sizes may be 64-, 48-, or 32-bits. External memory data transfer sizes may be 32-, 16 or 8-bits. The I/O processor can transfer Short word data (16-bit) using the packing capability of the external port, serial port and SPI port DMA channels.

After transferring each data word to or from internal memory, the I/O processor adds the modify value to the index register to generate the address for the next DMA transfer and writes the modified index value to the index register. The modify value in the `IMx` register is a signed integer, which allows both increment and decrement modifies. The modify value `IMx` (which was fixed to 1 on the ADSP-21065L) can now have any positive or negative integer value because of SIMD mode.

ⓘ If the I/O processor modifies the index register past the maximum 18-bit value to indicate an address out of internal memory, the index wraps around to zero. With the offset for the ADSP-21161, the wrap around address is 0x0004 0000.

Each DMA channel has a count register (`Cx`) that loads the programs with a word count to be transferred. The I/O processor decrements the count register after each DMA transfer on that channel. When the count reaches zero, the I/O processor generates the interrupt for that DMA channel. For more information on DMA interrupts, see "Using I/O Processor Status" on page 6-121.

⃠ If a program loads the count (`Cx`) register with zero, the I/O processor does not disable DMA transfers on that channel. The I/O processor interprets the zero as a request for 2^{16} transfers. This count occurs because the I/O processor starts the first transfer before the testing the count value. The only way to disable a DMA channel is to clear its DMA enable bit. For more information, see "External Port Channel Transfer Modes" on page 6-46, "Link Port Channel Transfer Modes" on page 6-85, or "Serial Port Channel Transfer Modes" on page 6-99.

Each DMA channel also has a chain pointer register (CPx) and a general-purpose register (GPx). Chained DMA sequences are a set of multiple DMA sequences, each autoinitializing the next in line. The location of the parameters for the next sequence comes from the CPx register. These parameters are called a Transfer Control Block (TCB), and they set up DMA parameter values for autoinitializing the next DMA sequence in the chain. Programs can use the GP register for any purpose, but usually programs store the address of the previous TCB in this register during chained DMA. For more information, see "Chaining DMA Processes" on page 6-25.

The external port DMA channels each contain three additional parameter registers, the external index register (EIEPx), external modify register (EMEPx), and external count register (ECEPx). These three registers are not available for the serial port, SPI port and link port DMA channels. The I/O processor generates 32-bit external memory addresses using the EIEPx, EMEPx, and ECEPx registers, during DMA transfers between internal memory and external memory or devices.

Programs must load the ECEPx register with the count of external bus transfers in the DMA. If the external port is using word packing, the ECEPx count differs from the number of words transferred in the DMA.

Memory mapped devices can communicate with the I/O processor using an internal DMA request/grant handshake on an external port DMA channel. Each channel has a single request and a single grant. When a particular I/O port needs to perform transfers to or from internal memory, the channel asserts a request. The I/O processor prioritizes this request with all other valid DMA requests. The default channel priority is DMA channel 0 as highest and DMA channel 13 as lowest. Table 6-1 lists the DMA channels in priority order. For more information, see "Managing DMA Channel Priority" on page 6-22.

When a channel becomes the highest priority requester, the I/O processor services the channel's request. In the next clock cycle, the I/O processor starts the DMA transfer.

If a DMA channel is disabled, the I/O processor does not service requests for that channel, whether or not the channel has data to transfer.

The processor's 14 DMA channels are numbered as shown in Table 6-1. This table also shows the control, parameter, and data buffer registers that correspond to each channel.

Table 6-1. DMA Channel Registers: Controls, Parameters, and Buffers

DMA Chan#	Control Registers	Parameter Registers	Buffer Register	Description
0	SPCTL0	II0A, IM0A, C0A, CP0A, GP0A	RX0A, TX0A	Serial Port 0 A Data
1		II0B, IM0B, C0B, CP0B, GP0B	RX0B, TX0B	Serial Port 0 B Data
2	SPCTL1	II1A, IM1A, C1A, CP1A, GP1A	RX1A, TX1A	Serial Port 1 A Data
3		II1B, IM1B, C1B, CP1B, GP1B	RX1B, TX1B	Serial Port 1 B Data
4	SPCTL2	II2A, IM2A, C2A, CP2A, GP2A	RX2A, TX2A	Serial Port 2 A Data
5		II2B, IM2B, C2B, CP2B, GP2B	RX2B, TX2B	Serial Port 2 B Data

Table 6-1. DMA Channel Registers: Controls, Parameters, and Buffers (Cont'd)

DMA Chan#	Control Registers	Parameter Registers	Buffer Register	Description
6	SPCTL3	II3A, IM3A, C3A, CP3A, GP3A	RX3A, TX3A	Serial Port 3 A Data
7		II3B, IM3B, C3B, CP3B, GP3B	RX3B, TX3B	Serial Port 3 B Data
8	LCTL, SPICTL[1]	IILB0, IMLB0, CLB0, CPLB0, GPLB0 IISRX, IMSRX, CSRX, GPSRX	LBUF0, SPIRX	Link Buffer 0 SPI Receive
9		IILB1, IMLB1, CLB1, CPLB1, GPLB1 IISTX, IMSTX, CSTX, GPSTX	LBUF1 SPITX	Link Buffer 1 SPI Transmit
10	DMAC10	IIEP0, IMEP0, CEP0, CPEP0, GPEP0, EIEP0, EMEP0, ECEP0	EPB0	External Port FIFO Buffer 0
11[2]	DMAC11	IIEP1, IMEP1, CEP1, CPEP1, GPEP1, EIEP1, EMEP1, ECEP1	EPB1	External Port FIFO Buffer 1
12[3]	DMAC12	IIEP2, IMEP2, CEP2, CPEP2, GPEP2, EIEP2, EMEP2, ECEP2	EPB2	External Port FIFO Buffer 2
13	DMAC13	IIEP3, IMEP3, CEP3, CPEP3, GPEP3, EIEP3, EMEP3, ECEP3	EPB3	External Port FIFO Buffer 3

1 Link port and SPI DMA parameter register names correspond to the same IOP addresses since these peripherals share DMA channels 8 and 9. Since chaining is not supported for SPI DMA, a chain pointer register cannot be use for DMA operation.

2 The $\overline{\text{DMAR1}}$ and $\overline{\text{DMAG1}}$ pins are handshake controls for DMA channel 11.

3 The $\overline{\text{DMAR2}}$ and $\overline{\text{DMAG2}}$ pins are handshake controls for DMA channel 12.

All of the I/O processor's registers are memory-mapped in the processor's internal memory, ranging from address 0x0000 0000 to 0x0000 01FF. For more information on these registers, see "I/O Processor Registers" on page A-47.

Because the I/O processor registers are memory-mapped, the processor and external processors (host or multiprocessors) have access to program DMA operations. A processor sets up a DMA channel by writing the transfer's parameters to the DMA parameter registers. After the `IIx`, `IMx`, and `Cx` registers (among others) are loaded with a starting source or destination address, an address modifier, and a word count, the processor is ready to start the DMA.

The external ports, link ports, SPI port, and serial ports each have a DMA enable bit (`DEN`, `LxDEN`, `SPIEN`, or `SDEN`) in their channel control register. Setting this bit for a DMA channel with configured DMA parameters starts the DMA on that channel. If the parameters configure the channel to receive, the I/O processor transfers data words received at the buffer to the destination in internal memory. If the parameters configure the channel to transmit, the I/O processor transfers a word automatically from the source memory to the channel's buffer register. These transfers continue until the I/O processor transfers the selected number of words as determined by the count parameter.

To start a new (non-chained) DMA sequence after the current one is finished, programs must disable the channel (clear its `DEN` bit); write new parameters to the `IIx`, `IMx`, and `CEPx` registers; then enable the channel (set its `DEN` bit). For chained DMA operations, this disable-enable process is not necessary. For more information, see "Chaining DMA Processes" on page 6-25.

DMA Channel Allocation and Priorities

ADSP-21161 has 14 DMA channels including eight channels accessible via the serial ports, four via the external port and two via the link ports. SPI shares the link port channels for receive and transmit. It is assumed that if DMA is enabled in SPI, then link ports cannot use any of the DMA channels.Table 6-2 shows the DMA channel allocation for the ADSP-21161.

Table 6-2. DMA Channel Allocation and Parameter Register Assignments

DMA Channel #	Data Buffer	Parameter Registers	IOP Address of DMA Parameter Register	Description
0	RX0A or TX0A	II0A, IM0A, C0A, CP0A, GP0A	0x60 to 0x64	Serial Port 0 A data
1	RX0B or TX0B	II0B, IM0B, C0B, CP0B, GP0B	0x80 to 0x84	Serial Port 0 B data
2	RX1A or TX1A	II1A, IM1A, C1A, CP1A, GP1A	0x68 to 0x6C	Serial Port 1 A data
3	RX1B or TX1B	II1B, IM1B, C1B, CP1B, GP1B	0x88 to 0x8C	Serial Port 1 B data
4	RX2A or TX2A	II2A, IM2A, C2A, CP2A, GP2A	0x70 to 0x74	Serial Port 2 A data
5	RX2B or TX2B	II2B, IM2B, C2B, CP2B, GP2B	0x90 to 0x94	Serial Port 2 B data
6	RX3A or TX3A	II3A, IM3A, C3A, CP3A, GP3A	0x78 to 0x7C	Serial Port 3 A data
7	RX3B or TX3B	II3B, IM3B,C3B, CP3B, GP3B	0x98 to 0x9C	Serial Port 3 B dara

Table 6-2. DMA Channel Allocation and Parameter Register Assignments (Cont'd)

DMA Channel #	Data Buffer	Parameter Registers	IOP Address of DMA Parameter Register	Description
8	LBUF0/SPIRX	IILB0,IMLB0,CLB0, CPLB0,GPLB0 IISRX, IMSRX, CSRX, GPSRX (no CPx)	0x30 to 0x34	Link Buffer 0 / SPI Receive
9	LBUF1/SPITX	IILB1, IMLB1, CLB1, CPLB1, GPLB1 IISTX, IMSTX, CSTX, GPSTX (no CPx)	0x38 to 0x3C	Link Buffer 1 / SPI Transmit
10	EPB0	IIEP0, IMEP0, CEP0, CPEP0, GPEP0, EIEP0, EMEP0, ECEP0	0x40 to 0x47	External Port FIFO Buffer 0
11[1]	EPB1	IIEP1, IMEP1, CEP1, CPEP1, GPEP1, EIEP1, EMEP1, ECEP1	0x48 to 0x4F	External Port FIFO Buffer 1
12[2]	EPB2	IIEP2, IMEP2, CEP2, CPEP2, GPEP2, EIEP2, EMEP2, ECEP2	0x50 to 0x57	External Port FIFO Buffer 2
13	EPB3	IIEP3, IMEP3, CEP3, CPEP3, GPEP3, EIEP3, EMEP3, ECEP3	0x58 to 0x5F	External Port FIFO Buffer 3

1 $\overline{\text{DMAR1}}$ and $\text{D}\overline{\text{MAG1}}$ are handshake controls for DMA channel 11

2 $\overline{\text{DMAR2}}$ and $\overline{\text{DMAG2}}$ are handshake controls for DMA channel 12.

DMA channel 0 has the highest priority and DMA channel 13 has the lowest priority.

The DMA channel arbitration feature allows the link port or SPI channel group to rotate priority with the external port channels. This feature may be enabled by setting the PRROT bit in the SYSCON IOP register. The DMA controller can be programmed to use a rotating priority scheme for the four external port channels by setting the DCPR bit in the SYSCON register. The DMA controller can be programmed to use a rotating priority scheme for the two link port DMA channels (channels 8 and 9) by setting the LDCPR bit in the SYSCON register.

Each channel has a set of parameter registers (II, IM, C, CP, GP etc.) which are used to setup DMA transfers. DMA parameter register assignments for the various channels are shown in Table 6-2.

For ADSP-21160 programs to run on ADSP-21161 processor with no modifications, note that previously used mnemonics and the new mnemonics map to the same addresses whenever appropriate.

DMA Interrupt Vector Locations

Interrupts on the ADSP-21161 are generated at the end of a DMA transfer. This happens when the count register Cx for that channel decrements to zero. The interrupt vector locations for the each channel is listed in Table 6-3. The Link Port Interrupt vector locations and channels are listed in Table 6-4. The interrupt register diagram and bit descriptions are given in "Interrupt Mask Pointer Register (IMASKP)" on page A-32

Table 6-3. Interrupt Vector Locations

IRPTL/IMASK Bit #	Vector Address	DMA Channel	Data Buffer
10	0x28	0	RX0A or TX0A
11	0x2C	2	RX1A or TX1A
12	0x30	4	RX2A or TX2A
13	0x34	6	RX3A or TX3A

Table 6-3. Interrupt Vector Locations (Cont'd)

IRPTL/IMASK Bit #	Vector Address	DMA Channel	Data Buffer
10	0x28	1	RX0B or TX0B
11	0x2C	3	RX1B or TX1B
12	0x30	5	RX2B or TX2B
13	0x34	7	RX3B or TX3B
15	0x50	10	EPB0
16	0x54	11	EPB1
17	0x58	12	EPB2
18	0x5C	13	EPB3

Table 6-4. Link Port Interrupt Vector Locations

LIRPTL Bits Ptr/Mask/Latch	Vector Address	DMA Channel	Data Buffer
24/16/0 26/18/2	0x38 0x40	8	LBUF0 SPIRX
25/17/1 27/19/3	0x3C 0x44	9	LBUF1 SPITX

Booting Modes

The booting modes that are supported by the ADSP-21161 processor are given in Table 6-5.

Table 6-5. Booting Modes for ADSP-21161

EBOOT	LBOOT	$\overline{\text{BMS}}$	Booting Mode
1	0	output	EPROM Boot (connect $\overline{\text{BMS}}$ to EPROM chip select)[1]
0	0	1 (input)	Host Boot[1]
0	1	1 (input)	Link Boot[2]
0	1	0 (input)	Serial Boot (SPI)[3]
0	0	0 (input)	No Booting (processor executes from the external memory)

1 For the Host and EPROM boots, the DMA channel 10 (EPB0) is used.
2 For the link boot, the DMA channel 8 (LBUF0) is used.
3 Serial boot (SPI) uses DMA channel 8 (its mutually exclusive with the link ports).

DMA Controller Operation

DMA sequences start in different ways depending on whether DMA chaining is enabled. When chaining is not enabled, only the DMA enable bit (DEN) allows DMA transfers to occur. A DMA sequence starts when one of the following occurs:

- Chaining is disabled and the DMA enable bit (DEN) transitions from low to high.
- Chaining is enabled, DMA is enabled (DEN=1), and the CPx register address field is written with a non-zero value. In this case, TCB chain loading of the channel parameter registers occurs first.

- Chaining is enabled, the CPx register address field is non-zero, and the current DMA sequence finishes. Again, TCB chain loading occurs.

A DMA sequence ends when one of the following occurs:

- The count register Cx decrements to zero (both CEPx and ECEPx for external port channels).
- Chaining is disabled and the channel's DEN bit transitions from high to low. If the DEN bit goes low (=0) and chaining is enabled, the channel enters chain insertion mode and the DMA sequence continues. For more information, see "Inserting a TCB in an Active Chain" on page 6-28.

ⓘ When a program sets the DEN bit (=1) after a single DMA finishes, the DMA sequence continues from where it left off (for non-chained operations only). To start a new DMA sequence after the current one is finished, a program must first clear the DEN enable bit, write new parameters to the IIx, IMx, and Cx registers, then set the DEN bit to re-enable DMA. For chained DMA operations, these steps are not necessary. For more information, see "Chaining DMA Processes" on page 6-25.

⊘ If a DMA operation completes and the count register is rewritten before the DMA enable bit is cleared, the DMA transfer restarts at the new count.

Once a program starts a DMA process, the process is influenced by two external controls: DMA channel priority and DMA chaining. For more information, see "Managing DMA Channel Priority" on page 6-22 or "Chaining DMA Processes" on page 6-25.

Managing DMA Channel Priority

The DMA channels for each of the processor's I/O ports negotiate channel priority with the I/O processor using an internal DMA request/grant handshake. Each I/O port (link ports, serial port, SPI port, and external port) has one or more DMA channels, with each channel having a single request and a single grant. When a particular channel needs to read or write data to internal memory, the channel asserts an internal DMA request. The I/O processor prioritizes the request with all other valid DMA requests. When a channel becomes the highest priority requester, the I/O processor asserts the channel's internal DMA grant. In the next clock cycle, the DMA transfer starts. Figure 6-5 shows the paths for internal DMA requests within the I/O processor.

If a DMA channel is disabled (`DEN`, `LxDEN`, `SPIEN`, or `SDEN` bit =0), the I/O processor does not issue internal DMA grants to that channel, whether or not the channel has data to transfer.

Because more than one DMA channel can make a DMA request in a particular cycle, the I/O processor prioritizes DMA channel service. DMA channel prioritization determines which channel can use the IOD (I/O Data) bus to access memory. Default DMA channel priority is fixed prioritization by DMA channel type (serial ports, SPI port, link ports, or external port). Within the DMA channel types, the serial port DMA channels are always fixed priority, the external port DMA channels may be either fixed or rotated priority, and the link port DMA channels may be either fixed or rotated priority. Table 6-1 on page 6-13 lists the DMA channels in descending order of priority.

- For information on programming external port priority modes, see "External Port Channel Priority Modes" on page 6-43.
- For information on programming link port priority modes, see "Link Port Channel Priority Modes" on page 6-83.

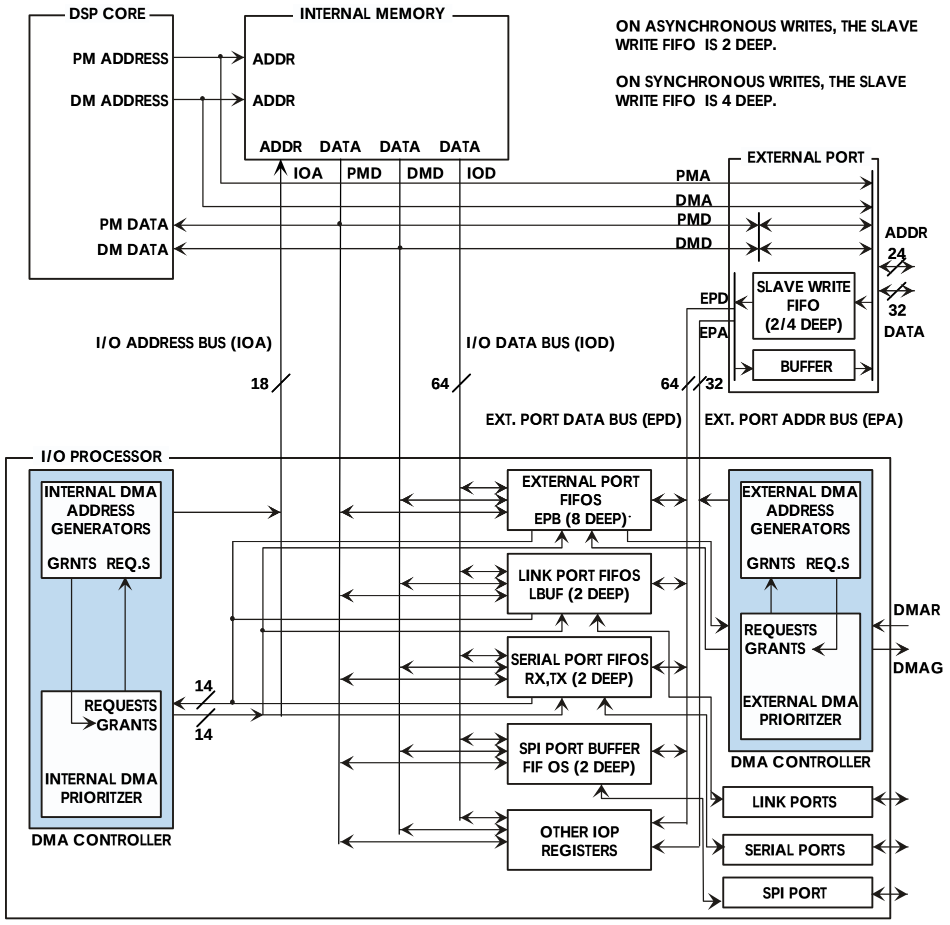


Figure 6-5. I/O Processor Request and Grant Paths

- For information on programming serial port priority modes, see "Serial Port Channel Priority Modes" on page 6-99.
- For information on programming SPI port priority modes, see "SPI DMA Channel Priority" on page 6-112.

The SPI port does not support DMA chaining.

The I/O processor determines which DMA channel has the highest priority internal DMA request during every cycle between each data transfer. Internal DMA channel arbitration differs from external bus arbitration. For more information on external bus arbitration, see "Multiprocessor Bus Arbitration" on page 7-93.

Processor core accesses of I/O processor registers and TCB chain loading are subject to the same prioritization scheme as the DMA channels. Applying this scheme uniformly prevents I/O bus contention, because these accesses are also performed over the internal I/O bus. TCB chain loading has a higher priority than external port accesses and link port/SPI port DMA accesses. This TCB priority permits chained serial port DMA, even when the external port is attempting an access in every cycle. For more information, see "Chaining DMA Processes" on page 6-25.

If a processor has the link ports enabled and active at the same time, the default priority scheme could hold off external port DMA channels for extended periods of time. Because this hold off could have a significant negative impact on external bus performance, the I/O processor permits rotating DMA channel priority between the link port channel group and external port channel group. For more information on using the `PRROT` bit to rotate priority between link ports and the external port, see "Link Port Channel Priority Modes" on page 6-83.

Chaining DMA Processes

DMA chaining lets the I/O processor automatically load DMA parameters and start the next DMA when the current DMA finishes. This feature permits unlimited multiple DMA transfers without processor core intervention. Using chaining, programs can set up multiple DMA operations with each operation can have different attributes.

To chain together multiple DMA operations, the I/O processor must load the next Transfer Control Block (DMA parameters) into the DMA parameter registers when the current DMA finishes (DMA count =0). The chain pointer register (`CPx`) points to the next set of DMA parameters, which are stored in internal memory. This process of loading the TCB into the parameter registers is called TCB chain loading.

Two controls enable chained DMA. Each DMA channel has a chaining enable bit (`CHEN`) in the channel's control register. When set, the `CHEN` bit directs the I/O processor to use the `CPx` register for chained DMA. Programs start the chained DMA by writing a non-zero address to the `CPx` register, directing the I/O processor to start the DMA with TCB chain loading. Programs can disable chained DMA by writing all zeros to the address field of the `CPx` register.

Chained DMA operations may only occur within the same channel. The processor does not support cross-channel chaining and the SPI port does not support DMA chaining

The `CPx` register is 19 bits wide, of which the lower 18 bits are the memory address field. Like other I/O processor address registers, the `CPx` registers value is offset to match the starting address of internal memory before being used by the I/O processor. On the ADSP-21161, this offset value is 0x0004 0000.

Bit 18 of the CPx register (shown in Figure 6-6) is the Program Controlled Interrupts (PCI) bit. If set, the PCI bit enables a DMA channel interrupt to occurs at the completion of the current DMA sequence.

The PCI bit only effects DMA channels that have chaining enabled (CHEN =1). Also, interrupt requests enabled by the PCI bit are maskable with the IMASK register.

Because the PCI bit is not part of the memory address in the CPx register, programs must be careful when writing and reading addresses to and from the register. To prevent errors, programs should mask out the PCI bit (bit 18) when copying the address in CPx to another address register.

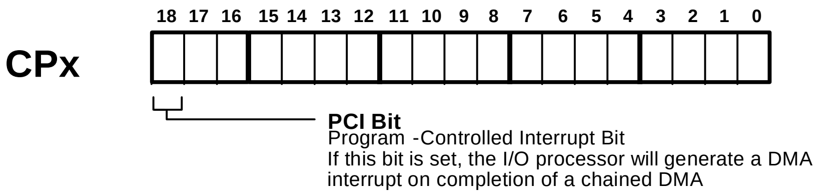

18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
CPx
PCI Bit
Program -Controlled Interrupt Bit
If this bit is set, the I/O processor will generate a DMA interrupt on completion of a chained DMA

Figure 6-6. CPX Register

During chained DMA, the channel's General Purpose (GP) register is a useful place to point to the last completed DMA sequence. This practice lets programs determine where the last full (or empty) data buffer is located.

Transfer Control Block (TCB) Chain Loading

During TCB chain loading, the I/O processor loads the DMA channel parameter registers with values retrieved from internal memory. The address in the CPx register points to the highest address of the TCB (containing the IIx or IIEPx parameter). The TCB values reside in consecutive memory locations.

Table 6-6 shows the TCB-to-register loading sequence for the external port, link port, and serial port DMA channels. The I/O processor reads each word of the TCB and loads it into the corresponding register. Programs must set up the TCB in memory in the order shown in Table 6-6, placing the `IIx` parameter at the address pointed to by the `CPx` register of the previous DMA operation of the chain.

Table 6-6. TCB Chain Loading Sequence

Address[1]	External Port	Link and Serial Ports
CPx + 0x0004 0000	IIEPx	IIx
CPx – 1 + 0x0004 0000	IMEPx	IMx
CPx – 2 + 0x0004 0000	CEPx	Cx
CPx – 3 + 0x0004 0000	CPEPx	CPx
CPx – 4 + 0x0004 0000	GPEPx	GPx
CPx – 5 + 0x0004 0000	EIEPx	
CPx – 6 + 0x0004 0000	EMEPx	
CPx – 7 + 0x0004 0000	ECEPx	
CPx – 8 + 0x0004 0000	–	

1 An "x" denotes the DMA channel used. Link, SPI, and serial ports use the first five locations only.

A TCB chain load request is prioritized like all other DMA operations. The I/O processor latches a TCB loading request and holds it until the load request has the highest priority. If multiple chaining requests are present, the I/O processor services the TCB registers for the highest priority DMA channel first. A channel which is in the process of chain loading cannot be interrupted by a higher priority channel. For a list of DMA channels in priority order, see Table 6-1 on page 6-13. For more information on DMA priority, see "Managing DMA Channel Priority" on page 6-22.

Setting Up and Starting the Chain

To setup and initiate a chain of DMA operations, use the following steps:

1. Set up all TCBs in internal memory.
2. Write to the appropriate DMA control register, setting the `DEN` DMA enable bit to 1 and the `CHEN` chaining enable bit to 1.
3. Write the address containing the `IIx` register value of the first TCB to the `CPx` register, starting the chain.

The I/O processor responds by autoinitializing the channel's parameter registers with the first TCB and starting the first transfer. When the transfer finishes, the I/O processor begins the next TCB chain load if the current chain pointer address is non-zero. The `CPx` address points to the next TCB.

⃠ The address field of the `CPx` registers is only 18 bits wide. If a program writes a symbolic address to bit 18 of `CPx`, there may be a conflict with the `PCI` bit. Programs should clear the upper bits of the address, then AND the `PCI` bit separately, if needed.

Inserting a TCB in an Active Chain

It is possible to insert a single DMA operation or another DMA chain within an active DMA chain. Programs may need to perform insertion when a high priority DMA requires service and cannot wait for the current chain to finish.

When DMA on a channel is disabled (DEN=0) and chaining on the channel is enabled (CHEN=1), the DMA channel is in chain insertion mode. This mode lets a program insert a new DMA or DMA chain within the current chain without effecting the current DMA transfer. Use the following sequence to insert a DMA subchain while another chain is active:

1. Enter chain insertion mode by setting CHEN=1 and DEN=0 in the channel's DMA control register. The DMA interrupt indicates when the current DMA sequence has completed.
2. Write the CPx register value into the CP position of the last TCB in the new chain.
3. Enter chained DMA mode by setting DEN=1 and CHEN=1.
4. Write the start address of the first TCB of the new chain into the CPx register.

Chain insertion mode operates the same as chained DMA mode (DEN=1, CHEN=1), except that when the current DMA transfer ends, automatic chaining is disabled and an interrupt request occurs. This interrupt request is independent of the PCI bit state.

Chain insertion should not be set up as an initial mode of operation. This mode should only be used to insert a DMA within an active DMA chaining operation.

External Port DMA

There are four external port DMA channels available on the ADSP-21161: channels 10, 11, 12 and 13. These DMA channels enable efficient data transfers between the processor's internal memory and external memory, peripherals, host processor, or other SHARCs. DMA transfers between the processor and any external devices that do not have bus master capability use these channels. Channels 10, 11, 12, and 13 are assigned to EPB0, EBP1, EPB2 and EPB3 buffers respectively, and are controlled by DMAC10, DMAC11, DMAC12 and DMAC13 DMA control registers.

The ADSP-21161 processor supports a number of DMA modes for external port DMA. The following sections describes typical external port DMA processes:

- "Setting Up External Port DMA" on page 6-68
- "Bootloading Through The External Port" on page 6-70
- "Boot Memory DMA Mode" on page 6-42
- "External Port Buffer Modes" on page 6-42
- "External Port Channel Priority Modes" on page 6-43
- "External Port Channel Transfer Modes" on page 6-46
- "External Port Channel Handshake Modes" on page 6-47

External Port Registers

The SYSCON, WAIT, and DMACx registers control the external port operating mode for the I/O processor. The following tables and figures describe the external port registers:

- Table A-10 on page A-34 lists all the bits in SYSCON
- Table A-11 on page A-37 lists all the bits in WAIT
- Table A-13 on page A-43 and Figure 6-8 on page 6-41 lists all the bits in DMACx

The following bits control external port I/O processor modes. Except for the FLSH bit, the control bits in the DMACx registers have a one cycle effect latency. The FLSH bit has a two cycle effect latency. Programs should not modify an active DMA channel's DMACx register other than to disable the channel by clearing the DEN bit. For information on verifying a channel's status with the DMASTAT register, see "Using I/O Processor Status" on page 6-121. Some other bits in SYSCON, WAIT, and DMACx setup non-DMA external port features. For information on these features, see "Setting External Port Modes" on page 7-3.

- **Boot Select Override.** SYSCON Bit 1 (BSO). This bit enables (if set, =1) or disables (if cleared, =0) access to Boot Memory Space. When BSO is set, the processor uses the $\overline{\text{BMS}}$ select line (instead of $\overline{\text{MS3-0}}$) to perform DMA channel 10 accesses to external memory.

- **Host Bus Width.** SYSCON Bits 5-4 (HBW). These bits select the host bus width as follows: 00=32-bit width, 01=16-bit width, 10=8-bit width (reset value).

- **Host Most Significant Word First Packing Select.** SYSCON Bit 7 (HMSWF). This bit selects the word packing order for host accesses as most-significant-word first (if set, =1) or least-significant-word first (if cleared, =0).

- **Buffer Hang Disable.** SYSCON Bit 16 (BHD). This bit controls whether the processor core proceeds (hang disabled if set, =1) or is held-off (hang enabled if cleared, =0) when the core tries to read from an empty EPBx, RXx, LBUFx or SPIRX buffer or tries to write to a full EPBx, TXx, LBUFx or SPITX buffer.

- **External Port DMA Channel Priority Rotation Enable.** SYSCON Bit 19 (DCPR). This bit enables (rotates if set, =1) or disables (fixed if cleared, =0) priority rotation among external port DMA channels (channel 10-13).

- **Handshake and Idle for DMA Enable.** WAIT Bit 30 (HIDMA). This bit enables (if set, =1) or disables (if cleared, =0) adding an idle cycle after every memory access for DMAs with handshaking ($\overline{\text{DMARx}}$-$\overline{\text{DMAGx}}$).

- **External Port DMA Enable.** `DMACx` Bit 0 (`DEN`). This bit enables (if set, =1) or disables (if cleared, =0) DMA for the corresponding external port FIFO buffer (`EPBx`).
- **External Port DMA Chaining Enable.** `DMACx` Bit 1 (`CHEN`). This bit enables (if set, =1) or disables (if cleared, =0) DMA chaining for the corresponding external port FIFO buffer (`EPBx`).
- **External Port Transmit/Receive Select.** `DMACx` Bit 2 (`TRAN`). This bit selects the transfer direction for the corresponding external port FIFO buffer (`EPBx`). If set (=1), the port transmits data from internal memory. If cleared (=0), the port receives data from external memory.
- **External Port Data Type Select.** `DMACx` Bit 5 (`DTYPE`). This bit selects the transfer data type (40/48=bit, 3-column if set, =1) (32/64-bit, 4-column if cleared, =0) for the corresponding external port FIFO buffer (`EPBx`).
- **External Port Packing Mode.** `DMACx` Bits 8-6 (`PMODE`). These bits select the packing mode for the corresponding external port FIFO buffer (`EPBx`) as follows: 000=reserved, 001=16 external to 32/64 internal packing, 010=16 external to 48 internal packing, 011=32 external to 48 internal packing, 100= no packing, 101=8 external to 48 internal packing, 110= 8 external to 32/64 internal packing, 111=reserved. During reset, the default is **PMODE** = 101.
- **Most Significant Word First.** `DMACx` Bit 9 (`MSWF`). When the buffer's **PMODE** is 001 or 010, this bit selects the packing order of 8-bit or 16-bit words (most significant first if set, =1) (least significant first if cleared, =0) for the corresponding external port FIFO buffer (`EPBx`).
- **Master Mode Enable.** `DMACx` Bit 10 (`MASTER`). This bit enables (if set, =1) or disables (if cleared, =0) master mode for the corresponding external port FIFO buffer (`EPBx`).

- **Handshake Mode Enable.** `DMACx` Bit 11 (`HSHAKE`) This bit enables (if set, =1) or disables (if cleared, =0) handshake mode for the corresponding external port FIFO buffer (`EPBx`).
- **External Handshake Mode Enable.** `DMACx` Bit 13 (`EXTERN`). This bit enables (if set, =1) or disables (if cleared, =0) external handshake mode for the corresponding external port FIFO buffer (`EPBx`).
- **External Port Bus Priority.** `DMACx` Bit 15 (`PRIO`). This bit selects the external bus access priority level (high if set, =1) (low if cleared, =0) for the corresponding external port FIFO buffer (`EPBx`).

External Port FIFO Buffers

DMA channels 10, 11, 12 and 13 are associated with the external port FIFO data buffers `EPB0`, `EPB1`, `EPB2`, and `EPB3`. Each buffer acts as an eight-location FIFO that has two ports: a read port and a write port. Each port can connect to either the EPD (External Port Data) or one of the following buses: the IOD (I/O Data) bus, the PM Data bus, or the DM Data bus.

The FIFO structure enables DMA transfers at full processor clock frequency with SDRAM or at the `CLKIN` system clock rate for host and other memories. This is possible because reads and writes for the same data can occur simultaneously through the FIFO's separate read and write ports. You can also use the external port FIFO buffers for non-DMA, single word data transfers too.

Do not attempt to make core reads or writes to or from an `EPBx` buffer when a DMA operation using that buffer is in progress. This corrupts the DMA data.

To flush (clear) an external port buffer, write 1 to the `FLSH` bit in the appropriate `DMACx` control register. The DMA for the channel must be disabled during the write operation. The `FLSH` bit is not latched internally and always reads as 0. Status can change in the following cycle. Do not enable and flush an external port buffer in the same cycle.

For DMA transfers between the processor's internal memory and external memory, the DMA controller must generate addresses in both memories. The external port DMA channels contain both `EIEPx` (External Index) and `EMEPx` (External Modify) registers to generate external addresses. The `EIEPx` register provides the external port address for the current DMA cycle. It is updated with the modifier value in `EMEPx` for the next external memory access.

External Port DMA Data Packing

Each external port buffer contains data packing logic to pack 8-, 16-, or 32-bit external bus words into 32/64 or 48-bit internal words. The packing logic works in reverse to unpack 32/64-bit data or 48-bit internal data into 8-, 16-, or 32-bit external data.

The external port data alignment is shown in Figure 6-7.

To support the wide range of data packing options provided for external DMA transfers, the `EIEPx` and `EMEPx` registers can generate addresses at a different rate than the internal address generating registers `IIEPx` and `IMEPx`. For this reason, the internal and external address generators operate independently, and the `ECEPx` (External Count) register serves as the external DMA word counter.

For example, when a 16-bit DMA device reads data from the processor's internal memory, two external 16-bit transfers occur for each 32-bit internal memory word. The `ECEPx` (external) word count is twice the value of the `CEPx` (internal) word count.

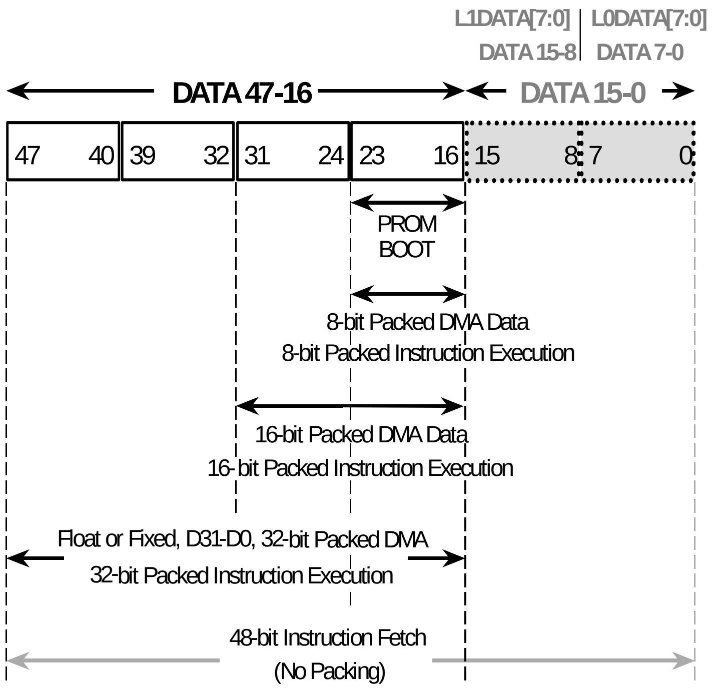


Figure 6-7. External Port Data Alignment

The `PMODE` bits in the `DMACx` control registers determine the packing mode for internal bus words while the `HBW` bits in the `SYSCON` register determine the packing mode for external bus words. Table 6-7 shows the packing modes of operation for the `PMODE[2:0]` that correspond to bits 8, 7, and 6 in the `DMACx` register.

During reset, the default value `PMODE` in `DMAC10` is 101 (8- to 48-bit packing for PROM or Host booting)

Table 6-7. Packing Mode Combinations

PMODE	HBW 8/16/32	Host Packing Mode (External:Internal)		
		IOP Buffers Internal Packing Fixed to 32-bit	Link Ports Buffers Internal Packing Fixed to 48-bit	External Port Buffers Uses PMODE, INT32 and DTYPE (1=48/40, 0=32/64)
000	-	Reserved		
001	01 (16-bit)	16 : 32	16 : 48	16 : 32/64
010	01 (16-bit)	16 : 32	16 : 48	16 : 48
011	00 (32-bit)	32 : 32	32 : 48	32 : 48
100	00 (32-bit)	32 : 32	32 : 48	32 : 32/64
101	10 (8-bit)	8 : 32	8 : 48	8 : 48
110	10 (8-bit)	8 : 32	8 : 48	8 : 32/64
111	-	Reserved		

Each external port DMA control register contains a three bit PS field that indicates the number of short words currently packed in the EPBx buffer. The PS status field behaves the same way during packing and unpacking operations. All packing functions are available for all types of DMA transfers. Table 6-8 shows the values of PS[2:0] that correspond to bits 23, 22, and 21 of the DMACx register.

Packing mode bit settings depend on whether the host access is processor-to-processor or processor-to-memory. To access another ADSP-21161 or memory, you must set the PMODE bits only (HBW bits have no effect) to pack and unpack individual data words for the following modes: master mode, paced master mode and handshake mode DMA.

For host accesses, to pack and unpack individual data words, you must set *both* the *PMODE* bits in the appropriate *DMACx* control register and the *HBW* bits in the *SYSCON* register. Table 6-7 shows the packing mode bit settings for access to IOP, link port and external port buffers.

Table 6-8. External Port FIFO Buffer Packing Status (Read Only)

PS[2:0]	EPBx Packing Status
000	Packing complete
001	1st stage
010	2nd stage
011	3rd stage
100	fifth stage of 8/48

For transfers to or from the `EPBx` data buffers, the packing mode is determined by the setting of the `HBW` bits of the `SYSCON` register AND the `PMODE` bits in the `DMACx` control register of each external port buffer.

The external port buffer can pack data in most significant word first (`MSWF`) order or in least significant word first (`LSWF`) order. Setting the bit `MSWF` to 1 in the `DMACx` control register selects MSW mode for both packing and unpacking operations. The `MSWF` bit has no effect when `PMODE`=111 or `PMODE`=000.

32-Bit Bus Downloading

The packing sequence for downloading processor instruction from a 32-bit bus (`PMODE`=011, `HBW`=00) takes three cycles for every two words, as shown in Table 6-9.

Table 6-9. Download Packing Sequence From a 32-Bit Bus

Transfer	Data Bus Pins 47-32	Data Bus Pins 31-16
First	Word 1; bits 47-32	Word 1; bits 31-16
Second	Word 2; bits 15-0	Word 1; bits 15-0
Third	Word 2; bits 47-32	Word 2; bits 31-16

For host transfers to or from the EPBx buffers, you must set the HBW bits in the SYSCON register to correspond to the external bus width. Note that the processor transfers 32-bit data on data bus lines DATA[47-16]. To transfer an odd number of instruction words, you must write a dummy access to flush the packing buffer and remove the unused word.

For 32- to 48-bit host packing, the processor ignores the HMSWF bit in the SYSCON register and the MSWF bit in the DMACx control register. For non-host accesses (for example, DMA master mode accesses to external memory) the processor uses the MSWF bit for packing and ignores the value of HMSWF in SYSCON.

16-Bit Bus Downloading

Table 6-10 and Table 6-11 show the packing sequence for downloading processor instructions from a 16-bit bus (PMODE=010, HBW=01). When interfacing to a host processor, the HMSWF bit determines whether the I/O processor packs to most significant 16-bit word first (=1) or least significant 16-bit word first (=0).

Table 6-10. Download Packing sequence for 16-bit bus (MSW first)

Transfer	Data Bus Pins 31-16
First	Word 1; bits 47-32
Second	Word 1; bits 31-16
Third	Word 1; bits 15-0

Table 6-11. Download Packing Sequence For 16-Bit Bus (LSW first)

Transfer	Data Bus Pins
First	Word 1; bits 15-0
Second	Word 1; bits 31-16
Third	Word 1; bits 47-32

8-Bit Bus Downloading

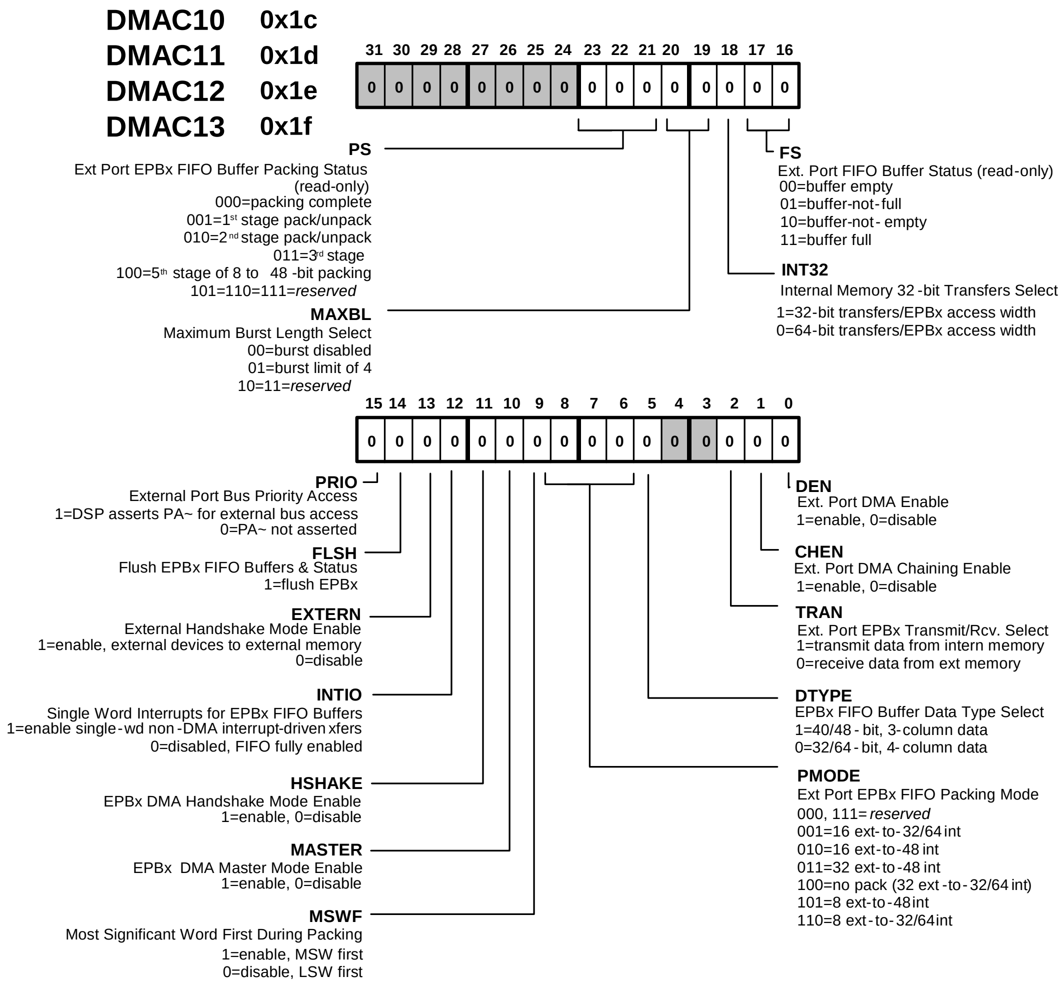
The packing sequence for downloading processor instructions from an 8-bit host (PMODE=101, HBW=10) takes six cycles for each word, as shown in Table 6-12 and Table 6-13. The HMSWF bit in SYSCON determines whether the I/O processor packs the most significant (=1) or least significant 8-bit word first (=0).

Table 6-12. Download Packing Sequence From 8-Bit Bus (MSW first)

Transfer	Data Bus Pins 23-16
First	Word 1; bits 47-40
Second	Word 1; bits 39-32
Third	Word 1; bits 31-24
Fourth	Word 1; bits 23-16
Fifth	Word 1; bits 15-8
Sixth	Word 1; bits 7-0

Table 6-13. Download Packing Sequence From 8-bit Bus (LSW first)

Transfer	Data Bus Pins 23-16
First	Word 1; bits 7-0
Second	Word 1; bits 15-8
Third	Word 1; bits 23-16
Fourth	Word 1; bits 31-24
Fifth	Word 1; bits 39-32
Sixth	Word 1; bits 47-40



DMAC10 0x1c
DMAC11 0x1d
DMAC12 0x1e
DMAC13 0x1f
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
PS
Ext Port EPBx FIFO Buffer Packing Status (read-only)
000=packing complete
001=1st stage pack/unpack
010=2nd stage pack/unpack
011=3rd stage
100=5th stage of 8 to 48 -bit packing
101=110=111=reserved
MAXBL
Maximum Burst Length Select
00=burst disabled
01=burst limit of 4
10=11=reserved
FS
Ext. Port FIFO Buffer Status (read-only)
00=buffer empty
01=buffer-not-full
10=buffer-not- empty
11=buffer full
INT32
Internal Memory 32 -bit Transfers Select
1=32-bit transfers/EPBx access width
0=64-bit transfers/EPBx access width
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
PRIO
External Port Bus Priority Access
1=DSP asserts PA~ for external bus access
0=PA~ not asserted
FLSH
Flush EPBx FIFO Buffers & Status
1=flush EPBx
EXTERN
External Handshake Mode Enable
1=enable, external devices to external memory
0=disable
INTIO
Single Word Interrupts for EPBx FIFO Buffers
1=enable single-wd non-DMA interrupt-driven xfers
0=disabled, FIFO fully enabled
HSHAKE
EPBx DMA Handshake Mode Enable
1=enable, 0=disable
MASTER
EPBx DMA Master Mode Enable
1=enable, 0=disable
MSWF
Most Significant Word First During Packing
1=enable, MSW first
0=disable, LSW first
DEN
Ext. Port DMA Enable
1=enable, 0=disable
CHEN
Ext. Port DMA Chaining Enable
1=enable, 0=disable
TRAN
Ext. Port EPBx Transmit/Rcv. Select
1=transmit data from intern memory
0=receive data from ext memory
DTYPE
EPBx FIFO Buffer Data Type Select
1=40/48 - bit, 3-column data
0=32/64 - bit, 4- column data
PMODE
Ext Port EPBx FIFO Packing Mode
000, 111=reserved
001=16 ext-to-32/64 int
010=16 ext-to-48 int
011=32 ext-to-48 int
100=no pack (32 ext-to-32/64 int)
101=8 ext-to-48 int
110=8 ext-to-32/64 int

Figure 6-8. DMAC Register

Boot Memory DMA Mode

The BSO bit in the SYSCON register enables Boot Memory Select Override, a mode in which the I/O processor supports DMA access to boot memory space. When BSO is set, the processor uses the $\overline{\text{BMS}}$ select line (instead of $\overline{\text{MS3-0}}$) to perform DMA channel 10 accesses to external memory.

When reading from 8-bit boot memory space, the processor uses 8- to 48-bit packing. Programs most often use this feature to finish loading programs and data after the processor completes its automatic 256-instruction bootload.

External Port Buffer Modes

The HBW, HMSWF, PMODE, MSWF, and BHD bits in the SYSCON and DMACx registers select a buffer's packing mode and disable buffer not-ready processor core stalls. The packing mode bits PMODE for processor and HBW for host select the external bus width and word size for transfers. **Packed data** or instructions are arranged in external memory according to the memory address that stems from their word size. For more information, see "Memory Organization and Word Size" on page 5-25. When data or instructions in external memory are **not packed**, the words are arranged in memory according to the external bus' data alignment. This data alignment appears in Figure 7-1 on page 7-2.

During reset, the default value for the PMODE bits in the DMAC10 register is 101 (8- to 48-bit packing for PROM/Host boot).

When the packing mode (PMODE or HBW) is set for a 16-bit bus, programs should set up the 16-bit word order. The 16-bit word order bits (MSWF for processor and HMSWF for host) control the order of 16-bit words being packed or unpacked in the 32-, 48-, or 64-bit word being transferred. If the MSWF or HMSWF bit is set (=1), the packing and unpacking is most significant 16-bit word first.

In addition to selecting the packing mode for external port processor transfers, programs must indicate the type of data in the transfer, using the Data Type (DTYPE) bit. For more information, see "External Port Channel Transfer Modes" on page 6-46.

The Buffer Hang Disable (BHD) bit lets the processor core proceed if the core tries to read from an empty EPBx, TXx, LBUFx or SPIRX buffer or tries to write to a full EPBx, RXx, LBUFx or SPITX buffer. The processor core still performs buffer accesses when buffer hang is disabled (FBHD=1). If the processor core attempts to read from an empty receive buffer, the core gets a repeat of the last value that was in the buffer. If the processor core attempts to write to a full buffer, the core overwrites the last value that was written to the buffer. Because these buffers are not initialized at reset, a read from a buffer that hasn't been filled since the reset returns an undefined value.

If an external port buffer's INTIO bit is set and DMA for that channel is not enabled, the external port channel is in single-word, interrupt-driven transfer mode. For more information, see "Using I/O Processor Status" on page 6-121.

External Port Channel Priority Modes

The DCPR and PRIO bits in the SYSCON and DMACx registers influence priority levels for an external port buffer and the external port in relation to external port DMA channels and external bus arbitration. For more information on prioritization operations, see "Managing DMA Channel Priority" on page 6-22.

Priority for DMA requests from external port channels can be fixed or rotated. When the DMA Channel Priority Rotate (DCPR) bit is cleared, the lowest number external port channel has the highest priority, ranging from highest-priority channel 10 to lowest-priority channel 13.

When the DCPR bit is set, the priority levels rotate. High priority shifts to a new channel after each packed single-word transfer. The I/O processor services a single-word transfer then rotates priority to the next higher numbered channel. Rotation continues until the I/O processor services all four external port channels. Figure 6-9 illustrates this process as described in the following steps:

1. At reset, external port channels have priority order—from high to low—10, 11, 12, and 13.
2. The external port performs a single transfer on channel 11.
3. The I/O processor rotates channel priority, changing it to 12, 13, 10, and 11 (because rotating priority is enabled for this example, DCPR=1).

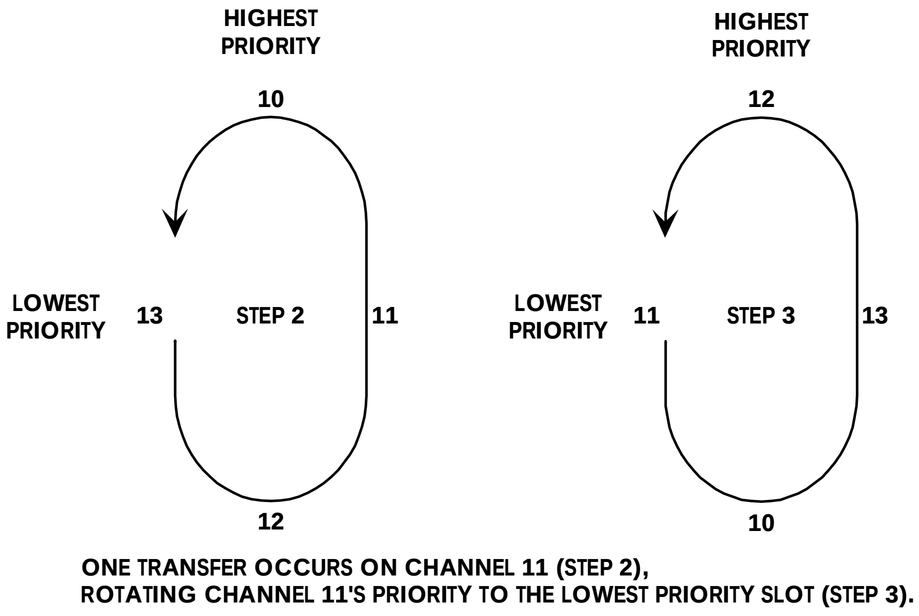

HIGHEST PRIORITY
10
LOWEST PRIORITY
13
STEP 2
11
12
HIGHEST PRIORITY
12
LOWEST PRIORITY
11
STEP 3
13
10
ONE TRANSFER OCCURS ON CHANNEL 11 (STEP 2),
ROTATING CHANNEL 11'S PRIORITY TO THE LOWEST PRIORITY SLOT (STEP 3).

Figure 6-9. Rotating External Port DMA Channel Priority

ⓘ Even though the external port channel DMA priority can rotate, the interrupt priorities of all DMA channels are fixed.

When external port DMA channel priority is fixed (`DCPR`=0), channel 10 has the highest priority, and channel 13 has the lowest priority. But, programs can redefine this priority order by assigning one of the other channels the highest priority. To change the fixed priority sequence of the external port DMA channels, a program could use the following procedure:

1. Disable all external port DMA channels except the one which is to have lowest priority.
2. Select rotating priority.
3. Cause at least one transfer to occur on the enabled channel.
4. Disable rotating priority and re-enable all of the external port DMA channels.

After completing this procedure, the channel immediately after the selected channel has the highest fixed priority.

In systems where multiple processors are using the external bus, the `PRIO` bit raises the priority level for external port DMA transfers. When a channel's `PRIO` bit is set, the I/O processor asserts the Priority Access ($\overline{\text{PA}}$) pin when that channel uses the external bus. The channel gets higher priority in bus arbitration, allowing the DMA to complete more quickly.

Programs can also rotate priority between external port and link port DMA channels. For more information, see "Link Port Channel Priority Modes" on page 6-83.

External Port Channel Transfer Modes

The DEN, CHEN, TRAN, and DTYPE bits in the DMACx register enable DMA and chained DMA and select the transfer direction and data type. The DMA enable (DEN) and Chained DMA enable (CHEN) bits work together to select an external port DMA channel's transfer mode. Table 6-14 lists the possible modes.

Table 6-14. External Port DMA Enable Modes

CHEN	DEN	DMA Enable Mode Description
0	0	Channel disabled (chaining disabled, DMA disabled)
0	1	Single DMA mode (chaining disabled, DMA enabled)
1	0	Chain insertion mode (chaining enabled, DMA enabled, auto-chaining disabled); For more information, see "Chaining DMA Processes" on page 6-25.
1	1	Chained DMA mode (chaining enabled, DMA enabled, auto-chaining enabled)

Because the external port is bidirectional, the I/O processor uses the Transmit select (TRAN) bit to determine the transfer direction (transmit or receive). Data flows from internal to external memory when in transmit mode. In transmit mode, the I/O processor fills the channel's EPBx buffer with data from internal memory when the channel's DEN bit is set.

The Data Type (DTYPE) bit determines how the DMA channel accesses columns of internal memory. If DTYPE is set, the data is 40- or 48-bit words, and the I/O processor makes 3-column internal memory accesses. If DTYPE is cleared, the data is 32- or 64-bit words, and the I/O processor makes 4-column internal memory accesses. For more information, see "Memory Organization and Word Size" on page 5-25.

The DTYPE for the transfer overrides the Internal Memory Data Width (IMDWx) setting for the internal memory block.

External Port Channel Handshake Modes

The `MASTER`, `HSHAKE`, `EXTERN`, and `HIDMA` bits in the `DMACx` and `WAIT` registers select the channel's DMA handshake and enable the hold cycles for host DMA. Table 6-15 summarizes the external port DMA modes. Table 6-16 shows how the `MASTER`, `HSHAKE`, and `EXTERN` bits work to select the channel's DMA handshake mode.

Table 6-15. External Port DMA Modes

Mode	Operation
Slave	Internal Memory <--> EPBx
Master	Internal Memory <--> EPBx <-->External Memory Uses strobes and address, No $\overline{\text{DMAR}}$ and $\overline{\text{DMAG}}$.
Paced Master	Internal Memory <--> EPBx <-->External Memory Uses strobes and address, Uses $\overline{\text{DMAR}}$, No $\overline{\text{DMAG}}$.
Handshake	Internal Memory <--> EPBx <-->External Memory No strobes and address, Uses $\overline{\text{DMAR}}$ and $\overline{\text{DMAG}}$.
External Handshake	External Memory <--> External Device Uses strobes and address, Uses $\overline{\text{DMAR}}$ and $\overline{\text{DMAG}}$.

Table 6-16. External Port DMA Handshake Modes: DMACx MASTER (M), HSHAKE (H), and EXTERN (E) Bits

E H M	DMA Mode of Operation
0 0 0	**Slave Mode.** The processor responds to the buffer's internal memory transfer activity based on the buffer status in the FS field, generating a DMA request whenever the buffer is not empty (on receive) or is not full (on transmit). During transmit (TRAN=1), the processor fills the EPBx buffer with data from internal memory when the program enables the buffer (DEN=1). For more information, see "Slave Mode" on page 6-55.
0 0 1	**Master Mode.** The processor attempts the internal memory DMA transfers indicated by the DMA counter (CEPx) based on the buffer status in the FS field, making transfers whenever the buffer is not empty (on receive) or is not full (on transmit). Systems using Master Mode should de-assert corresponding DMA request inputs, de-asserting $\overline{\text{DMAR1}}$ if channel 11 is in master mode and de-asserting $\overline{\text{DMAR2}}$ if channel 12 is in master mode. For more information, see "Master Mode" on page 6-50.
0 1 0	**Handshake Mode.** When in this mode, the processor generates a DMA request whenever the external device asserts the $\overline{\text{DMARx}}$ pin, then the processor asserts the $\overline{\text{DMAGx}}$ pin, transferring the data (and de-asserting $\overline{\text{DMAGx}}$) when the external devices de-asserts the $\overline{\text{DMARx}}$ pin. ⓘ Note that this mode only applies to external port buffers EPB1 and EPB2 and DMA channels 11 and 12. For more information, see "Handshake Mode" on page 6-57.
0 1 1	**Paced Master Mode.** The processor attempts the internal memory DMA transfers indicated by the DMA counter (CEPx), making transfers based on external DMA request inputs. The processor generates a DMA request whenever the external device asserts the $\overline{\text{DMARx}}$ pin. The processor controls the data transfer using the $\overline{\text{RD}}$ or $\overline{\text{WR}}$ and ACK pins and by applying the selected number of waitstates. ⓘ Note that this mode only applies to external port buffers EPB1 and EPB2 and DMA channels 11 and 12. For more information, see "Paced Master Mode" on page 6-54.

Table 6-16. External Port DMA Handshake Modes: DMACx MASTER (M), HSHAKE (H), and EXTERN (E) Bits (Cont'd)

E H M	DMA Mode of Operation
1 0 0	Reserved
1 0 1	Reserved
1 1 0	**External-Handshake Mode.** The processor responds to external memory DMA requests based on external DMA request inputs. This mode is identical to Handshake Mode, but applies to transfers between external memory and external devices. The processor generates a DMA request whenever the external device asserts the $\overline{\text{DMARx}}$ pin. The processor asserts the $\overline{\text{DMAGx}}$ pin, transferring the data (and de-asserting $\overline{\text{DMAGx}}$) when the external devices de-asserts the $\overline{\text{DMARx}}$ pin. Note that this mode only applies to external port buffers EPB1 and EPB2 and DMA channels 11 and 12. For more information, see "External-Handshake Mode" on page 6-66.
1 1 1	Reserved

For the handshake and external-handshake modes shown in Table 6-16, programs can insert an added idle cycle after every memory access. The handshake and Idle for DMA (HIDMA) bit in the WAIT register enables this added cycle, which reduces bus contention from devices with slow three-state timing or long recovery times.

Because external port DMA transfers can go between processor internal memory and external memory, the I/O processor must generate addresses for both memory spaces. The external port DMA channels have additional parameter registers (EIEPx, EMEPx, ECEPx) for external memory access.

To support data packing options for external memory DMA transfers, the EIEPx and EMEPx registers can generate addresses at a different rate than the internal address registers (IIEPx and IMEPx). Figure 6-5 on page 6-23 shows that the I/O processor has separate address generators for internal and external addresses. For this reason, when packing is used for external memory DMA, the external count (ECEPx) register indicates the number of external port transfers, not the number of internal memory words being transferred.

The DMA mode and other factors determine the size of the DMA data transfer on the external port. These other factors include the EIEPx, EMEPx, and ECEPx parameters; the PMODE, DTYPE, and MAXBL values in DMACx; and the transfer capacity available in the EPBx data buffer employed in the transfer. The internal I/O processor bus transfer size varies with the IIEPx, IMEPx, and CEPx parameters, and the PMODE, DMA mode, DTYPE, and INT32 values in DMACx. The following sections describe these DMA modes and transfer sizes in more detail:

- "Master Mode" on page 6-50
- "Paced Master Mode" on page 6-54
- "Slave Mode" on page 6-55
- "Handshake Mode" on page 6-57
- "External-Handshake Mode" on page 6-66

Master Mode

When the MASTER bit is set (=1) and the EXTERN and HSHAKE bits are cleared (=0) in the channel's DMACx register, the DMA channel is in master mode. A channel in this mode can independently initiate internal or external memory transfers.

Master mode applies to all external port DMA channels: 10, 11, 12, and 13. When interfacing to SDRAM memory, only master mode DMA can be used for external port DMA transfers between SDRAM and internal memory. $\overline{\text{DMARx}}$ and $\overline{\text{DMAGx}}$ pins cannot be used to pace or handshake DMA transfers using SDRAM interface pins.

To initiate a master mode DMA transfer, the processor sets up the channel's parameter registers and sets the channel's DMA enable (DEN) bit. A master mode DMA channel performing internal memory to external memory data transfer automatically performs enough transfers from internal memory to keep the EPBx buffer full. When the data transfer direction is external to internal, a master mode DMA channel also performs enough transfers from external memory to keep the EPBx buffer full.

The I/O processor uses the EIEPx, EMEPx, and ECEPx registers to access external processor memory in master mode DMA.

External Transfer Controls In Master Mode. In master mode, the processor determines the size of the external transfer from the channel's PMODE bits and EIEPx, EMEPx, and ECEPx registers. Table 6-8 on page 6-37 shows the packing mode selected by the PMODE bits, and Table 6-17 shows the external transfer size in master mode that results from the combination of the PMODE bits.

Table 6-17. Master Mode External Transfer Size

Transfer Size	32-bit	16-bit	8-bit
PMODE	011, 100	001, 010	110, 101
EIEP	X[1]	X	X
EMEP	X	X	X
ECEP	X	# of 16-bit xfers	# of 8-bit xfers
DTYPE	X	X	X
EPBx Depth	>=1	>=1	>=1

1 An X in Table 6-17 indicates any supported value.

32-bit External Transfers. The processor performs 32-bit transfers when PMODE= 011 (32- to 48-bit internal), or 100 (32-bit external-to-32-bit/64-bit internal). In PMODE=011or 100, all data transfers across the upper word of the data bus (DATA47-16) are as indicated in

Figure 7-1 on page 7-2. This mode supports all values of EIEPx, EMEPx, and ECEPx. ECEPx contains the number of 32-bit words to transfer. There must be at least one 32-bit EPBx FIFO entry available to support the 32-bit external transfer.

16-bit External Transfers. The processor performs 16-bit transfers when PMODE=001 (16-bit external-to-32/64-bit internal) or 010 (16-bit external to 48-bit internal). This mode supports all values of EIEPx, EMEPx, and ECEPx. The value ECEPx is programmed to the number of 16-bit words to transfer. There must be at least one 32-bit EPBx FIFO entry available to support the 16-bit external transfer. In PMODE=001, or 010, all data transfers across DATA31-16 as indicated in Figure 7-1 on page 7-2.

8-bit External Transfers. The processor performs 8-bit transfers when PMODE=110 (8-bit external to 32/64-bit internal) or 101 (8-bit external to 48-bit internal). This mode supports all values of EIEPx, EMEPx, and ECEPx. The value ECEPx is programmed to be the number of 8-bit words to transfer. There must be at least one 32-bit EPBx FIFO entry available to support the 8-bit external transfer. In PMODE=110 or 101, all data transfers across DATA23-16 as indicated in Figure 7-1 on page 7-2.

Internal Address/Transfer Size Generation. In master mode, the processor determines the size of the internal transfer from the channel's PMODE bits and IIEPx, IMEPx, and CEPx registers. Table 6-7 on page 6-36 shows the packing mode selected by the PMODE bits, and Table 6-18 shows the internal transfer size in master mode that results from the combination of the PMODE bits.

Table 6-18. Master Mode Internal Transfer Size Determination

Transfer Size	64-bit[1]	48-bit	32-bit
PMODE	001, 100, 110	010, 011, 101	001, 100, 110
IIEPx	depends on IM[2]	X[3]	X
IMEPx	-1 or 1	X	X

Table 6-18. Master Mode Internal Transfer Size Determination (Cont'd)

CEPx	even # of 32-bit words	# of 48-bit words	X
DTYPE	0	1	0
EPBx Depth	>1	>1	>=1
INT32	0	0	0 or 1

1 Including packed instructions.
2 If IMEPx is 1 for increment, IIEPx must be an even, 64-bit aligned Normal word address. If IMEPx is -1 for decrement, IIEPx must be an odd, Normal word address.
3 X indicates any supported value.

64-bit Internal Transfers. To enable internal 64-bit transfers and increment the internal IIEPx pointer, programs must set IIEPx to match the IMEPx selection as shown in Table 6-18. CEPx contains the number of 32-bit words to transfer, and should be set to an even number of 32-bit words. The processor decrements CEPx by 2 for each 64-bit transfer. For 64-bit transfers, PMODE must be set to 001 (16-bit-to-32/64-bit internal), 100 (32-bit external-to-32/64-bit internal) or 110 (8-bit external-to-32/64-bit internal). DTYPE and INT32 must be cleared. There must be at least two 32-bit EPBx FIFO entries available to support the 64-bit external transfer.

48-bit Internal Transfers. The processor can perform 48-bit internal transfers for DMA of packed or unpacked 48-bit instructions. Many applications can use internal 64-bit transfer for 48-bit instructions. This technique can provide greater throughput than 48-bit internal transfers.

In either of the 48-bit internal transfer modes in Table 6-18 (PMODE=101 and DTYPE=1 or PMODE=010 or 011 and DTYPE=0), the processor accesses the memory using instruction alignment (3-column read or write) for the EPBx buffer. In this case, IIEPx points to 48-bit words, and CEPx counts the number of 48-bit internal transfers.

32-bit Internal Transfers. The processor performs according to the conditions in Table 6-18. Under these additional conditions, the processor performs 32-bit transfers instead of 64- or 48-bit transfers: PMODE= 001 (16-bit external to 32-bit internal), or 100 (32-bit external to 32-bit internal), and IIEPx is not aligned to a 64-bit boundary, or IMEPx is < -1, or > 1, or CEPx is < 2, or EPBx depth < 2, or INT32 = 1, and DTYPE=0.

Paced Master Mode

When the MASTER and HSHAKE bits are set (=1) and the EXTERN bit is cleared (=0) in the channel's DMACx register, the DMA channel is in Paced Master mode. A channel in this mode can independently initiate internal or external memory transfers.

Paced Master mode applies only to external port DMA channels 11 and 12.

In Paced Master mode, the processor has the same control for address generation and transfer size as in master mode. For more information, see "Master Mode" on page 6-50. The difference between these modes is that in Paced Master mode external transfers are controlled and initiated (paced) by the $\overline{\text{DMARx}}$ signal as in handshake mode. For more information, see "Handshake Mode" on page 6-57.

The processor responds to the $\overline{\text{DMARx}}$ request only with the $\overline{\text{RD}}$, or $\overline{\text{WR}}$ strobes, depending on direction and data alignment. $\overline{\text{DMAGx}}$ is not asserted in Paced Master mode. This method lets the processor share the same buffer between the I/O processor and processor core without external gating. Paced Master mode accesses can be extended by the ACK input, by waitstates programmed in the WAIT register, and by holding the $\overline{\text{DMARx}}$ input low.

Slave Mode

When the MASTER, HSHAKE, and EXTERN bits in the channel's DMACx register are cleared (=0), the DMA channel is in slave mode. A channel in this mode cannot independently initiate external memory transfers.

To initiate a slave mode DMA transfer, an external device must read or write the channel's EPBx buffer. A slave mode DMA channel performing internal to external data transfer automatically performs enough transfers from internal memory to keep the EPBx buffer full. When the data transfer direction is external to internal, a slave mode DMA channel does not initiate any internal DMA transfers until the external device writes data to the channel's EPBx buffer. Note that the I/O processor does not use the EIEPx, EMEPx, and ECEPx registers in slave mode DMA

The following sequence describes a typical external to internal slave mode DMA operation where an external device transfers a block of data into the processor's internal memory:

1. The external device initializes the channel by writing the DMA channel's parameter registers (IIEPx, IMEPx, and CEPx) and DMACx control register.

2. The external device begins writing data to the EPBx buffer.

3. The EPBx buffer detects that data is present and asserts an internal DMA request to the I/O processor.

4. The I/O processor grants the request and performs the internal DMA transfer, emptying the EPBx buffer FIFO.

If the internal DMA transfer is held off, the external device can continue writing to the EPBx buffer because of its eight-deep FIFO. When the EPBx FIFO becomes full, the processor holds off the external device with the ACK signal (for synchronous accesses) or with the REDY signal (for asynchronous, host-driven accesses). This hold-off state continues until the I/O processor finishes the internal DMA transfer, freeing space in the EPBx buffer.

The following sequence describes a typical internal to external slave mode DMA operation where an external device transfers a block of data from the processor's internal memory:

1. The external device writes the DMA channel's parameter registers (IIEPx, IMEPx, and CEPx) and DMACx control register, initializing the channel and automatically asserting an internal DMA request to the I/O processor.

2. The I/O processor grants the request and performs the internal DMA transfer, filling the EPBx buffers FIFO.

3. The external device begins reading data from the EPBx buffer.

4. The EPBx buffer detects that there is room in the buffer (it is now partially empty) and asserts another internal DMA request to the I/O processor, continuing the process.

If the internal DMA transfers cannot fill the EPBx FIFO buffer at the same rate as the external device empties it, the processor holds off the external device with the ACK signal (for synchronous accesses) or with the REDY signal (for asynchronous, host-driven accesses) until valid data can be transferred to the EPBx buffer.

The processor only deasserts the ACK (or REDY) signal when the EPBx FIFO buffer (or pad data buffer) is full during a write. The ACK (or REDY) signal remains asserted at the end of a completed block transfer if the EPBx buffer is not full. For reads, the processor deasserts the ACK (or REDY) signal for each read to handle the latency of the read versus posting the write to a buffer.

In slave mode, the processor determines the size of the transfer based on the setting of channel's PMODE bits. Table 6-19 shows the transfer size in slave mode that results from the PMODE bits and Table 6-7 on page 6-36 shows the packing mode selected by the PMODE bits.

Table 6-19. Slave Mode Transfer Size Determination

Transfer Size (external↔internal)	32-bit↔ 32/64-bit	32-bit↔ 48-bit	16-bit↔ 32/64-bit[1]	16-bit↔ 48-bit1	8-bit↔ 32/64-bit[2]	8-bit↔ 48-bit2
PMODE	100	011	001	010	110	101
DTYPE	0	1	0	1	0	1

1 External device must be connected to DATA[31:16]

2 External device must be connected to DATA[23:16]

Handshake Mode

When the MASTER and EXTERN bits are cleared (=0) and the HSHAKE bit is set (=1) in the channel's DMACx register, the DMA channel is in handshake mode. A channel in this mode cannot independently initiate external memory transfers. Note that handshake mode only applies to DMA channels 11 and 12.

To initiate a handshake mode DMA transfer, an external device must assert an external DMA request, asserting $\overline{\text{DMAR1}}$ for access to EPB1 or $\overline{\text{DMAR2}}$ for access to EPB2. The buffers pass these request to the I/O processor, which prioritizes these requests with other internal DMA requests. When the external DMA request has the highest priority, the I/O processor asserts an external DMA grant, asserting $\overline{\text{DMAG1}}$ for EPB1 or $\overline{\text{DMAG2}}$ for EPB2. The grant signals the external device to read or write the EPBx buffer. A handshake mode DMA channel performing internal to external data transfer automatically performs enough transfers from internal memory to keep the EPBx buffer full. When the data transfer direction is external to internal, a handshake mode DMA channel does not initiate any internal DMA transfers until the external devices writes data to the channel's EPBx buffer.

The I/O processor does not use the EIEPx or EMEPx registers in handshake mode DMA. It uses the ECEPx registers.

Other than the $\overline{\text{DMARx}}$/$\overline{\text{DMAGx}}$ handshake, handshake mode DMA operations follow almost the same process as slave mode DMA operations. The exception is that in handshake mode DMAs from internal to external memory the external device must load the channel's ECEPx register with the number of external bus transfers.

In handshake mode, the processor determines the size of the transfer from the channel's parameter registers and PMODE bits. Table 6-7 on page 6-36 shows the packing mode selected by the PMODE bits, and Table 6-20 shows the transfer size in handshake mode that results from the combination of the read and write signals and PMODE bits.

Table 6-20. Handshake Mode Transfer Size Determination

Transfer Size (external↔internal)	32-bit↔ 32/64-bit[1]	32-bit↔ 48-bit2	16-bit↔ 32/64-bit[2]	16-bit↔ 48-bit2	8-bit↔ 32/64-bit[3]	8-bit↔ 48-bit2
PMODE	100	011	001	010	110	101
IIEPx	X[4]	X	X	X	X	X
IMEPx	X	X	X	X	X	X
CEPx	# of 32-bit words	# of 32-bit words	# of 16-bit words	# of 16-bit words	# of 8-bit words	# of 8-bit words
ECEPx	# of 32-bit words	6/4 * CEPx	2 * CEPx	3 * CEPx	4 * CEPx	6 * CEPx
DTYPE	0	1	0	1	0	1

1 External device must be connected to the upper half of the data bus (Data[47:16])
2 External device must be connected to Data[16:31])
3 External device must be connected to Data[16:23])
4 X indicates any legal value

DMA transfers are supported at the full system `CLKIN`/`CLKOUT` rates of 50MHz. However, full bandwidth at 2:1 core clock (`CCLK`) to `CLKIN`/`CLKOUT` ratio is not possible. Non synchronous timing specifications limit throughput for three DMA handshake modes: paced master mode, handshake mode and external handshake mode. The sampling rate of the $\overline{\texttt{DMARx}}$ signal by the internal circuitry of the ADSP-21161 processor prohibits maximum throughput at a `CCLK` to `CLKIN`/`CLKOUT` ratio of 2:1. For handshake mode DMA, the processor does not assert the $\overline{\texttt{MS3-0}}$ memory select lines (the address strobes). For information on $\overline{\texttt{DMARx}}$/$\overline{\texttt{DMAGx}}$ handshake timing, see Figure 6-10.

`CCLK` to `CLKIN` ratios of 3:1 and 4:1 with $\overline{\texttt{CLKDBL}}$ =1 and `CCLK` to `CLKIN` ratios of 4:1, 6:1 and 8:1 with $\overline{\texttt{CLKDBL}}$ =0 support full speed throughput at the `CLKIN` frequency. If the maximum $\overline{\texttt{DMARx}}$/$\overline{\texttt{DMAGx}}$ throughput at 50MHz is needed, synchronize the assertions and deassertions of $\overline{\texttt{DMARx}}$ with respect to `CLKOUT`. Refer to the *ADSP-21161N DSP Microcomputer Data Sheet* for specific timing information.

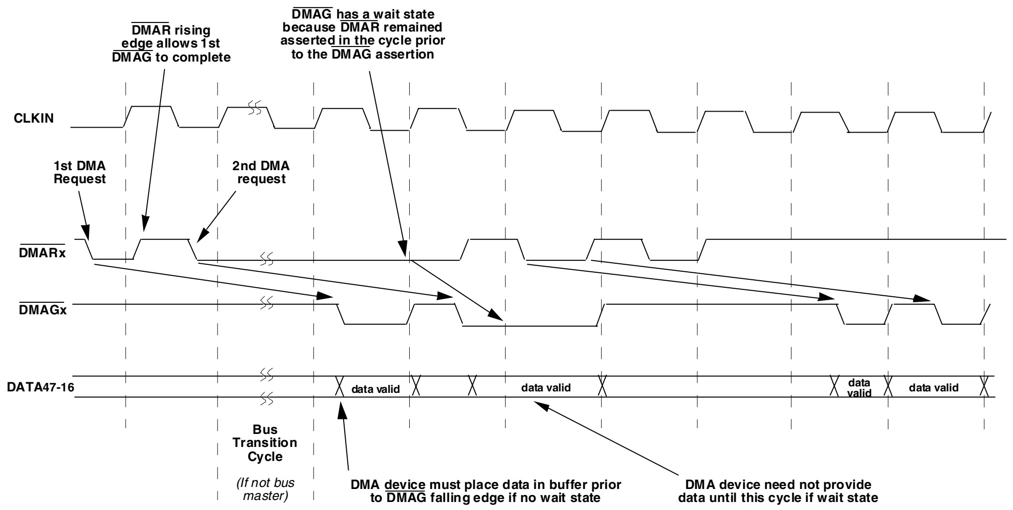

DMAR rising edge allows 1st DMAG to complete
DMAG has a wait state because DMAR remained asserted in the cycle prior to the DMAG assertion
CLKIN
1st DMA Request
2nd DMA request
DMARx
DMAGx
DATA47-16
data valid
data valid
data valid
data valid
Bus Transition Cycle
(If not bus master)
DMA device must place data in buffer prior to DMAG falling edge if no wait state
DMA device need not provide data until this cycle if wait state

Figure 6-10. Handshake DMA Timing (Asynchronous Requests)

The I/O processor uses the rising and falling edges of $\overline{\texttt{DMARx}}$ in the $\overline{\texttt{DMARx}}$/$\overline{\texttt{DMAGx}}$ handshake as prompts for DMA operations. On the falling edge of $\overline{\texttt{DMARx}}$, the edge signals the I/O processor to begin a DMA access. On the rising edge of $\overline{\texttt{DMARx}}$, the edge signals the I/O processor to complete the DMA access.

The following sequence describes the process for requesting access to an `EPBx` buffer in handshake mode:

1. The external device asserts the buffer's $\overline{\texttt{DMARx}}$ signal, placing an external DMA request for access to the `EPBx` buffer.

2. The `EPBx` buffer detects the falling edge of the $\overline{\texttt{DMARx}}$ signal and passes the external DMA request to the I/O processor, synchronizing the DMA operation with the processor's system clock.

   To be recognized in a particular cycle, the $\overline{\texttt{DMARx}}$ low transition must meet the signal setup time from the processor data sheet. If the transition is slower than the setup time, the signal may not take effect until the following cycle.

3. The I/O processor prioritizes the external DMA request with other internal DMA requests. If the processor is not already bus master, the processor arbitrates for the external bus when the external DMA request has the highest priority, unless the `EPBx` buffer is blocked.

   If the `EPBx` buffer is full during a write or empty during a read, the buffer is blocked. The processor does not begin external bus arbitration until the I/O processor services the `EPBx` buffer, returning it to the unblocked state empty for writing or full for reading.

4. The processor becomes bus master and asserts $\overline{\texttt{DMAGx}}$.

   The processor keeps $\overline{\texttt{DMAGx}}$ asserted until the cycle after the external device deasserts $\overline{\texttt{DMARx}}$. By holding $\overline{\texttt{DMARx}}$ asserted, the external device holds the processor until the external device is ready to proceed. If the external device does not need to extend the DMA grant cycle, the external device can deassert $\overline{\text{DMARx}}$ immediately (not waiting for $\overline{\text{DMAGx}}$), providing the $\overline{\text{DMARx}}$ assertion time meets the timing requirements from the processor data sheet. The responding $\overline{\text{DMAGx}}$ in this case is a short pulse, and the processor only uses the external bus for one cycle.

The I/O processor has a three-cycle DMA pipeline and a seven-deep external request counter. The I/O processor's DMA pipeline is similar to the program sequencer's fetch–decode–execute instruction pipeline. The I/O processor processes the DMA pipeline in the following stages:

- It recognizes the DMA request and arbitrates internal DMA priority during the DMA fetch cycle.
- It generates the DMA address and arbitrates external bus access during the DMA decode cycle.
- It transfers DMA data during the DMA execute cycle.

Because the I/O processor has a three-cycle DMA pipeline, there is a minimum delay of three cycles before the processor asserts $\overline{\text{DMAGx}}$. This delay is in addition to any delay from internal DMA arbitration, so the external device must not assume that the DMA grant can arrive within two cycles even if higher priority DMA operations are disabled and the external bus is available for the transfer.

The I/O processor's external request counter increments each time the external device asserts $\overline{\text{DMARx}}$ and decrements each time the processor replies by asserting $\overline{\text{DMAGx}}$. The external request counter records up to seven requests, so the external device can make up to seven requests before the first one has been serviced.

If the processor cannot immediately service the DMA requests in the external request counter, the processor services the requests on a prioritized basis. The external DMA device is responsible for keeping track of requests, monitoring grants, and pipelining the data when operating at full speed.

If the external device makes more than seven $\overline{\texttt{DMARx}}$ without receiving a grant, the delayed grant causes unpredictable results.

The processor only asserts $\overline{\texttt{DMAGx}}$ for the number of $\overline{\texttt{DMARx}}$ requests indicated by the external request counter. If the external devices make more requests than the count indicates, the processor $\overline{\texttt{DMAGx}}$ assertions cannot match the number of external device requests. To clear this mismatch, programs can clear the buffer and the external request counter using the flush bit ($\overline{\texttt{FLSH}}$) in the channel's `DMACx` register.

To prevent holding off the processor, the external device must service the processor's data requirements when it asserts the $\overline{\texttt{DMAGx}}$ grant signal. The external device should immediately supply data for writes to the processor or immediately accept data on reads from the processor. External interfaces can handle this I/O by placing the data in an external FIFO. When performing DMA operations at the full `CLKIN` speed of the processor, the system may need a three-deep external FIFO to handle the latency between request and grant. Programs on the external device can optimize operation of this FIFO by issuing three requests rapidly and making the next requests conditional on when the processor issues a grant.

The external devices must follow the conditions in Figure 6-11 when enabling or disabling handshake mode for an external port DMA channel:

- The processor ignores a disabled (transitioning from disabled to enabled) DMA channel's $\overline{\texttt{DMARx}}$ and $\overline{\texttt{DMAGx}}$ pins and ignores internal $\overline{\texttt{DMARx}}$ assertions for up to two processor core clock cycles after the instruction that enables the channel in handshake mode.

- The external devices must maintain $\overline{\text{DMARx}}$ deasserted (kept high, not low or changing) during the instruction that enables DMA in handshake mode. Before using the channel for the first time, programs flush the DMA channel, asserting the `FLSH` bit in the `DMACx` control register. This action is not required during chain insertion.
- The processor deasserts $\overline{\text{DMAGx}}$ if a program disables the channel while $\overline{\text{DMARx}}$ and $\overline{\text{DMAGx}}$ are asserted (=0). This action clears the channel's active status bit, avoiding a potential deadlock condition.

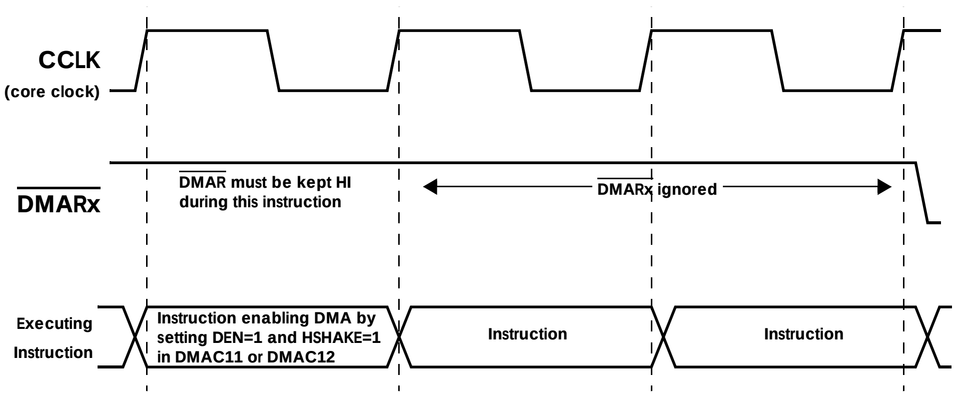

CCLK
(core clock)
$\overline{\text{DMARx}}$
$\overline{\text{DMAR}}$ must be kept HI during this instruction
DMARx ignored
Executing Instruction
Instruction enabling DMA by setting DEN=1 and HSHAKE=1 in DMAC11 or DMAC12
Instruction
Instruction

Figure 6-11. $\overline{\text{DMARx}}$ Delay After Enabling Handshake DMA

ADSP-21161 processors in a multiprocessing cluster may share a $\overline{\text{DMAGx}}$ signal, because only the bus master drives $\overline{\text{DMAGx}}$. On the bus slaves, $\overline{\text{DMAGx}}$ is three-stated. This state eliminates the need for external gating if more than one processor or the host needs to drive the DMA buffer. Systems may need a pullup resistor on this line if the host is not connected to the pin or does not drive it when it acquires the bus. $\overline{\text{DMAGx}}$ has the same timing and transitions as the $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes in asynchronous access mode. For more information, see "Bus Arbitration Protocol" on page 7-95. $\overline{\text{DMAGx}}$ responds to the $\overline{\text{SBTS}}$ and $\overline{\text{HBR}}$ signals in the same way as the read and write strobes.

DMA Handshake Idle Cycle

Idle cycles can be inserted during DMA handshaking to interface with slower devices. Normally a bus idle cycle occurs when an asynchronous read is followed by an immediate write to an external memory bank or when crossing bank boundary. During this idle cycle, the address and data lines continue to drive the previous value. $\overline{\texttt{RD}}$, $\overline{\texttt{WR}}$ and $\overline{\texttt{DMAGx}}$ lines deassert.

If the asynchronous read is immediately followed by a write, the processor recognizes that a write request is pending during the idle cycle. Therefore, the $\overline{\texttt{MSx}}$ lines do not deassert during the idle cycle. Instead, the lines are driven with their previous value (asserted).

Idle cycles can be inserted after every memory access by setting the `HIDMA` bit in the `WAIT` register for DMAs with handshaking. For a handshake mode DMA transfer, the $\overline{\texttt{MSx}}$ lines are never asserted. When an external handshake mode DMA is enabled with a bus idle cycle inserted in between the transfers, the `MSx` lines do not deassert during the bus idle cycle if the I/O processor recognizes a pending `DMARx` request. If there are no pending `DMARx` requests, $\overline{\texttt{MSx}}$ lines do deassert.

Figure 6-12 shows an external handshake mode DMA transfer on channel 11 with three $\overline{\texttt{DMAR1}}$ pulses asserted. The `HIDMA` bit is set in the `WAIT` register in order to insert bus idle cycles between two handshake transfers. The first data transfer is to location 0x255000 in bank 0 and the second transfer is to location 0x255001. An idle cycle is inserted between the two transfers. Note that the first two $\overline{\texttt{DMAR1}}$ pulses are sequential. Therefore, during this idle cycle, the I/O processor recognizes that there is a $\overline{\texttt{DMAR1}}$ request pending. As a result of the pending request, the $\overline{\texttt{MS0}}$ line is not deasserted.

The third data transfer is to location 0x255002. Again, an idle cycle is inserted between the second and third transfers. However, the third DMA transfer request happens after some time has transpired and following the 2nd $\overline{\texttt{DMAR1}}$ pulse. In this case, the I/O processor recognizes that there are

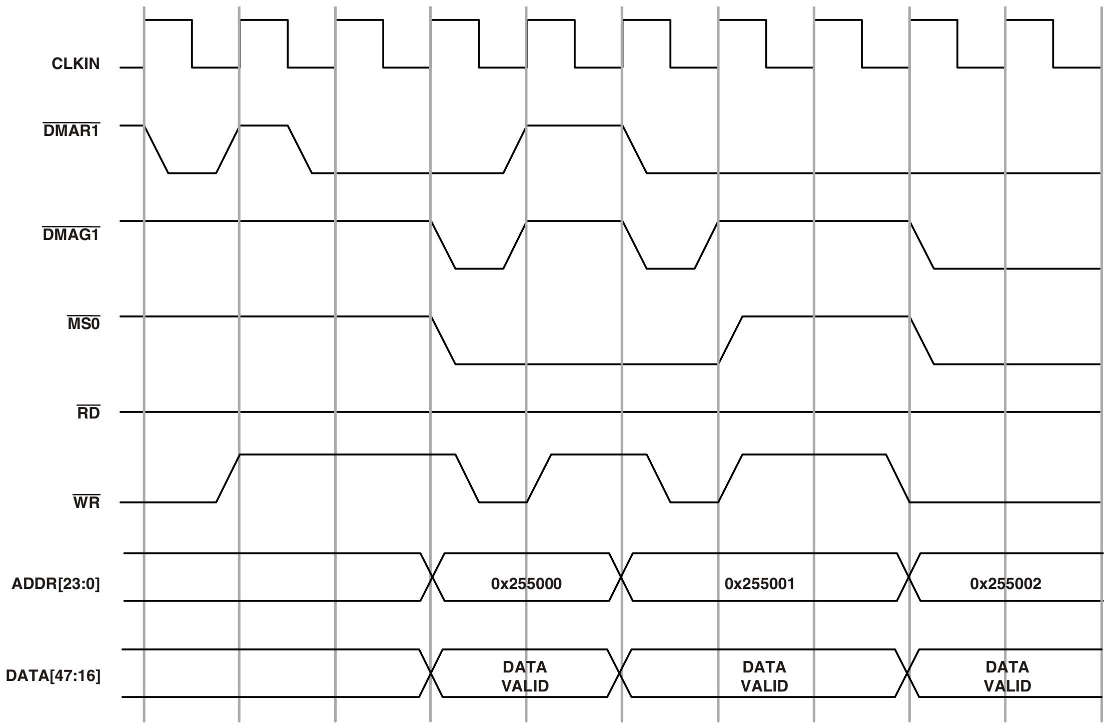


Figure 6-12. DMA Handshake Idle Cycle

no more $\overline{\texttt{DMAR1}}$ requests pending. Therefore, during the idle cycle between the second and third transfers, the $\overline{\texttt{MS0}}$ line goes high. $\overline{\texttt{MS0}}$ goes low again when the 3rd data transfer occurs.

Systems must be evaluated to determine if the idle cycle during a external handshake DMA with an activated $\overline{\texttt{MSx}}$ line has an adverse impact on the chip selected memory devices or peripherals. The $\overline{\texttt{RD}}$, $\overline{\texttt{WR}}$, and $\overline{\texttt{DMAG}}$ strobes are inactive during the idle cycle, and therefore the $\overline{\texttt{MSx}}$ lines being activated should not affect interconnection to other devices as long as $\overline{\texttt{RD}}$ and $\overline{\texttt{WR}}$ remain inactive. Otherwise, an idle cycle insertion between DMA handshake transfers cannot be used.

External-Handshake Mode

External-handshake mode is identical to handshake mode, except that external-handshake mode transfers data between external memory and an external device. This section describes the differences between handshake mode and external-handshake mode. For more information, see "Handshake Mode" on page 6-57.

When the `MASTER` bit is cleared (=0) and the `HSHAKE` and `EXTERN` bits are set (=1) in the channel's `DMACx` register, the DMA channel is in external-handshake mode. A channel in this mode cannot independently initiate external memory transfers.

ⓘ Like handshake mode, external-handshake mode only applies to DMA channels 11 and 12.

⃠ Do not use external handshake mode DMA on an external memory bank that has SDRAM mapped and connected to its $\overline{\text{MSx}}$ line.

To initiate an external-handshake mode DMA transfer, an external device must assert an external DMA request, asserting $\overline{\text{DMAR1}}$ for access to DMA channel 11or $\overline{\text{DMAR2}}$ for access to DMA channel 12. The channels pass these request to the I/O processor, which prioritizes these requests with other internal DMA requests. When the external DMA request has the highest priority, the I/O processor asserts an external DMA grant, asserting $\overline{\text{DMAG1}}$ for channel 11 or $\overline{\text{DMAG2}}$ for channel 12. The grant signals the external device to read or write the external bus. An external-handshake mode DMA channel performing external to external data transfer automatically generates external memory addresses and strobes for transfers between external memory and the external device.

ⓘ Unlike handshake mode, the I/O processor must use the `EIEPx`, `EMEPx`, and `ECEPx` registers in external-handshake mode DMA. Also unlike handshake mode, the data for DMA channels 11 and 12 does not pass through the `EPB1` or `EPB2` buffers.

During external-handshake mode transfers, the I/O processor generates external memory access cycles. $\overline{\text{DMARx}}$ and $\overline{\text{DMAGx}}$ operate the same as in handshake mode, but the processor also outputs addresses, $\overline{\text{MS3-0}}$ memory selects, and $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes, and responds to ACK. On external memory writes, the processor asserts $\overline{\text{DMAGx}}$ until the external device releases the ACK line or any of the processor waitstates expire. The external memory access by the external devices responds as if the processor core were making the access. For more information, see "External Port" on page 7-1.

Because the I/O processor accesses external memory in external-handshake mode, programs must load the DMA channel's EIEPx, EMEPx, and ECEPx parameter registers and the DMAC10 or DMAC11 PMODE bits. These settings let the I/O processor generate the external memory addresses and word count.

External-handshake mode does not support chained DMA interrupts. Because no internal DMA transfers occur in external-handshake mode, the PCI bit in the channel's CPEPx register cannot disable the DMA interrupt. Programs must use the IMASK register to mask this interrupt.

In external-handshake mode, the processor does not perform packing. The processor does determine the size of the transfer from the channel's parameter registers, PMODE bits. Table 6-21 shows the transfer size in external handshake mode that results from the combination of the read and write signals and PMODE bits. For 32-bit memory transfers to an external device, PMODE must be set to the no packing mode (=100) in the DMACx register.

Setting Up External Port DMA

The method for setting up and starting an external port DMA sequence varies slightly with the selection of transfer and DMA handshake for the channel.

- For more information on transfer modes, see "External Port Channel Transfer Modes" on page 6-46.
- For more information on DMA handshake modes, see"External Port Channel Handshake Modes" on page 6-47.

Table 6-21. External Handshake Mode Transfer Size

Transfer Size (memory↔device)	32-bit memory↔32-bit device[1]
PMODE	100
EIx	X[2]
EMx	X
ECx	X
DTYPE	0

1 External device must be connected to the upper half of the data bus (Data[47:16])

2 X indicates any legal value

The following sequence describes a typical external to internal DMA operation where an external device transfers a block of data into the processor's internal memory:

1. The processor or host (depending on the mode) writes to the DMA channel's parameter registers (`IIEPx`, `IMEPx`, and `CEPx`) and the `DMACx` register, initializing the channel for receive (`TRAN`=0).
2. The processor or host (depending on the mode) sets the channel's `DEN` bit to 1 enabling the DMA process.

3. The external device begins writing data to the EPBx buffer through the external port.

4. The EPBx buffer detects data is present and asserts an internal DMA request to the I/O processor.

5. The I/O processor grants the request and performs the internal DMA transfer, emptying the EPBx buffer FIFO.

The following sequence describes a typical internal to external DMA operation where an external device transfers a block of data from the processor's internal memory:

1. The processor or host (depending on the mode) writes the DMA channel's parameter registers (IIEPx, IMEPx, and CEPx) and the DMACx register, initializing the channel for transmit (TRAN=1).

2. The processor or host (depending on the mode) sets the channel's DEN bit to 1 enabling the DMA process. Because this is a transmit, setting DEN automatically asserts an internal DMA request to the I/O processor.

3. The I/O processor grants the request and performs the internal DMA transfer, filling the EPBx buffer's FIFO.

   The processor may signal the start of this transfer depending on the mode.

4. The external device begins reading data from the EPBx buffer through the external port. The processor may signal the start of this transfer depending on the mode.

5. The EPBx buffer detects that there is room in the buffer because it is now partially empty and asserts another internal DMA request to the I/O processor, continuing the process.

Bootloading Through The External Port

The processor can boot from an EPROM or host processor through the external port. The `DMAC10` control register is initialized for booting in each case. Each booting mode packs boot data into 48-bit instructions. EPROM and host boot use channel 10 of the I/O processor's DMA controller to transfer the instructions to internal memory. For EPROM booting, the processor reads data from an 8-bit external EPROM. For host booting, the processor accepts data from a 8-, 16- or 32-bit host microprocessor (or other external device).

It is important to note that DMA channel differences between the ADSP-21161 and previous SHARC processors (ADSP-2106x) introduce some booting differences. Even with these differences, the ADSP-21161 supports the same boot capability and configuration as the ADSP-2106x processors.

The `DMACx` register default values differ because the ADSP-21161 has additional parameters and different DMA channel assignments. The EPROM and Host boot modes use `EPB0`, DMA channel 10.

Like the ADSP-2106x, the ADSP-21161 boots from `DATA23-16`.

For EPROM or host booting the ADSP-21161, the Program sequencer automatically unmasks the DMA channel 10 channel interrupt, initializing the `IMASK` register to 0x00008003.

The processor determines the booting mode at reset from the `EBOOT`, `LBOOT`, and $\overline{\text{BMS}}$ pin inputs. When `EBOOT`=1 and `LBOOT`=0, the processor boots from an EPROM through the external port and uses $\overline{\text{BMS}}$ as the memory select output. When `EBOOT`=0, `LBOOT`=0, and $\overline{\text{BMS}}$ =1, the processor boots from a host through the external port. For a list showing how to select different boot modes, see the Boot Memory Select pin description in the table Table 13-11 on page 13-72.

When using any of the power-up booting modes, address 0x0004 0004 should not contain a valid instruction since it is not executed during the booting sequence. A `NOP` or `IDLE` instruction should be placed at this location.

In EPROM booting through the external port, an 8-bit wide boot EPROM must be connected to data bus pins 23-16 (`DATA23-16`). The lowest address pins of the processor should be connected to the EPROM's address lines. The EPROM's chip select should be connected to $\overline{\text{BMS}}$ and its output enable should be connected to $\overline{\text{RD}}$.

In a multiprocessor system, the $\overline{\text{BMS}}$ output is only driven by the ADSP-21161 bus master. This allows wire-ORing of multiple $\overline{\text{BMS}}$ signals for a single common boot EPROM. Systems can boot any number of ADSP-21161's from a single EPROM using the same code for each processor or differing code for each.

During reset, the processor's `ACK` line is internally pulled high with a 20kΩ equivalent resistor and is held high with an internal keeper latch. It is not necessary to use an external pullup resistor on the `ACK` line during booting or at any other time.

After the boot process loads 256 words into memory locations 0x4 0000 through 0x4 00FF, the processor begins executing instructions. Because most processor programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. Analog Devices supplies loading routines (loader kernels) that can load entire programs. These routines come with the development tools. For more information on loader kernels, see the development tools documentation.

Host Processor Booting

When host booting mode is configured, the ADSP-21161 enters slave mode after reset and waits for the host to download the boot program. After reset the ADSP-21161 processor goes into an idle state, identical to that caused by the `IDLE` instruction, with the program counter (PC) set to address 0x0004 0004. The parameter registers for the external port DMA channel 10 are initialized as shown in Table 6-22.

Table 6-22. DMA Channel 10 Parameter Register Initialization for Host Booting

Parameter Register	Initialization Value
IIEP0	0x0004 0000
IMEP0	uninitialized (increment by 1 is automatic)
CEP0	0x0100 (256 instruction words)
CPEP0	uninitialized
GPEP0	uninitialized
EIEP0	uninitialized
EMEP0	uninitialized
ECEP0	uninitialized

Table 6-22 shows how the DMA channel 10 parameter registers are initialized at reset for host booting.The count register (`CEP0`) is initialized to 0x0100 for transferring 256 words to internal memory. The `DMAC10` control register is initialized to 0x00000161. The default value sets up external port transfers as follows:

- `DEN` = 1, external port enabled
- `MSWF` = 0, LSW first

- PMODE = 101, 8- to 48-bit packing
- DTYPE = 1, three-column data

The external port DMA Channel 10 (DMAC10) becomes active following reset; it is initialized to 0x0000 0161. This enables the external port DMA and selects DTYPE for instruction words. The packing mode bits (PMODE) in the DMACx register are set to 8- to 48-bit packing. The host bus width (HBW) and word order (HMSWF) bits must be programmed in the SYSCON register.

For each 48-bit word of boot image, an 8-bit host performs the following sequence of operations:

1. Assert $\overline{\text{HBR}}$ and $\overline{\text{CS}}$.
2. Wait for $\overline{\text{HBG}}$. After the host receives the host bus grant signal back from the ADSP-21161 processor, it can start downloading instructions or it can change the reset initialization conditions of the ADSP-21161 processor by writing to any of the IOP control registers.
3. Write the six subwords to the external port buffer, EPB0. This buffer corresponds to DMA channel 10. The host must use data pins DATA23-16.
4. Deassert $\overline{\text{CS}}$ and $\overline{\text{HBR}}$. The processor samples the inactive $\overline{\text{HBR}}$ and allows a host transition cycle. The processor can access the bus for external memory initialization.

For 16 and 32-bit host bus widths, the HBW bits in the SYSCON register must be modified. The host must use the data lines as follows:

16-bit host bus width = 3 subwords using data pins DATA31-16

32-bit host bus width = 2 subwords using data pins DATA47-16

PROM Booting

When the EPROM boot mode is configured, the external port DMA Channel 10 (DMAC10) becomes active following reset; it is initialized to 0000 0561. This enables the external port DMA and selects DTYPE for instruction words. 8- to 48-bit packing is forced with least-significant-word first.

The RBWS and RBAM fields of the WAIT register are initialized to perform asynchronous access and to generate seven wait states (eight cycles total) for the EPROM access in external memory space. Note that wait states defined for boot memory are applied to $\overline{\text{BMS}}$-asserted accesses.

Table 6-23 shows how the DMA channel 10 parameter registers are initialized at reset for EPROM. The count register (CEP0) is initialized to 0x0100 for transferring 256 words to internal memory. The external count register (ECEP0), which is used when external addresses are generated by the DMA controller, is initialized to 0x0600 (for example, 0x0100 words with six bytes per word). The DMAC10 control register is initialized to 0000 0561. The default value sets up external port transfers as follows:

- DEN = 1, external port enabled
- MSWF = 0, LSW first
- PMODE = 101, 8- to 48-bit packing
- DTYPE = 1, three-column data

Table 6-23. DMA Channel 10 Parameter Register Initialization for EPROM Booting

Parameter Register	Initialization Value
IIEP0	0x0004 0000
IMEP0	uninitialized (increment by 1 is automatic)
CEP0	0x0100 (256 instruction words)
CPEP0	uninitialized
GPEP0	uninitialized
EIEP0	0x0080 0000
EMEP0	uninitialized (increment by 1 is automatic)
ECEP0	0x0600 (256 words x 6 bytes/word)

At system start-up, when the processor's $\overline{\texttt{RESET}}$ input goes inactive, the following sequence occurs:

1. The processor goes into an idle state, identical to that caused by the `IDLE` instruction. The program counter (`PC`) is set to address 0x0004 0004.

2. The DMA parameter registers for channel 10 are initialized as shown in Table 6-23.

3. $\overline{\texttt{BMS}}$ becomes the boot EPROM chip select.

4. 8-bit Master Mode DMA transfers from EPROM to internal memory begin, on the external port data bus lines 23-16.

5. The external address lines (`ADDR23-0`) start at 0x0080 0000 and increment after each access.

6. The $\overline{\texttt{RD}}$ strobe asserts as in a normal memory access with seven wait states (eight cycles).

The processor's DMA controller reads the 8-bit EPROM words, packs them into 48-bit instruction words, and transfers them to internal memory until 256 words have been loaded. The EPROM is automatically selected by the $\overline{\text{BMS}}$ pin; other memory select pins are disabled.

The DMA external count register (ECEP0) decrements after each EPROM transfer. When ECEP0 reaches zero, the following wake-up sequence occurs:

1. The DMA transfers stop.
2. The External Port DMA Channel 10 interrupt (EP0I) is activated.
3. $\overline{\text{BMS}}$ is deactivated and normal external memory selects are activated.
4. The processor vectors to the EP0I interrupt vector at 0x0004 0050.

At this point the processor has completed its booting mode and is executing instructions normally. The first instruction at the EP0I interrupt vector location, address 0x0004 0050, should be an RTI (Return from Interrupt). This process returns execution to the reset routine at location 0x0004 0005 where normal program execution can resume. After reaching this point, a program can write a different service routine at the EP0I vector location 0x0004 0050.

External Port DMA Programming Examples

This section provides two programming examples written for the ADSP-21161 processor. The example shown in Listing 6-1 demonstrates how the I/O processor uses DMA to read from the external port receive buffer and write to the external port transmit buffer after an interrupt. The example shown in Listing 6-2 demonstrates how the I/O processor uses DMA chaining to read from the external port receive buffer and write to the external port transmit buffer.

Listing 6-1. External Port DMA Example

```
/*_____________________________________________________________
ADSP-21161 Internal-to-External Memory (External Port) DMA
Example
This example shows an internal to external memory no packing
32-bit DMA transfer.
_____________________________________________________________*/

#include "def21161.h"
#define  N 8

.GLOBALinit_int_to_ext_memory_DMA;

.SECTION/DM     dm_data;

.VAR source[N]= 0x11111111,
0x22222222,
0x33333333,
0x44444444,
0x55555555,
0x66666666,
0x77777777,
0x88888888;

.SECTION/DM     segsdram;
.VAR dest[8];

/*__________start of DMA initialization
routine___________________*/

.SECTION/PMpm_code;

init_int_to_ext_memory_DMA:
```

```
r0=0;dm(DMAC10)=r0;         /* Clear DMA Control Register */

r0=source;dm(IIEP0) = r0;  /* Write source address to IIEP0 */
r0=1;dm(IMEP0)=r0;          /* Write internal address modify
                                 to IMEP0 */
r0=@source;dm(CEP0)=r0;     /* Load internal DMA 10 Count
                                  Register */

r0=dest; dm(EIEP0)=r0;      /* Write destination address to
                                 EIEP0 register */
r0=1; dm(EMEP0)=r0;         /* Write external address modify
                                 to EMEP0 */
r0=@dest;dm(ECEP0)=r0;      /* Load external DMA 10 Count
                                  Register */

/* master mode, no packing mode [PMODE=100] */
/* transmit data from int>ext, enable EP DMA */
/* DMAC10=b#00000000000000000000010100000101; */
   ustat1 = 0x00000000;
   bit set ustat1 MASTER | PMODE4 | TRAN | DEN;
   dm(DMAC10)=ustat1;

bit set imask EP0I;    /* Unmask external port buffer 0
                            DMA interrupt */

rts;
```

Listing 6-2. External Port Chained DMA Example

```
/*_________________________________________________________________
 ADSP-21161 Internal-to-External Memory (External Port)
 Chained DMA Example

This example shows an internal to external memory, no packing
32-bit chained DMA transfer.
___________________________________________________________________*/

#include "def21161.h"
#define  N 8

.GLOBALint_to_ext_memory_chainDMA;

.SECTION/DM     dm_data;
.VAR source[N]= 0x11111111,
0x22222222,
0x33333333,
0x44444444,
0x55555555,
0x66666666,
0x77777777,
0x88888888;

.VAR tcb[8] = N, /* ECx */
 1, /* EMx */
0, /* EIx */
0, /* GPx */
0, /* CPx */
N, /* Cx  */
1, /* IMx */
0; /* IIx */
```

```
.SECTION/DM     segsdram;
.VAR dest[8];


/*________start of DMA initialization routine__________*/

.SECTION/PMpm_code;

int_to_ext_memory_chainDMA:
r0=source;
dm(tcb + 7) = r0;  /* Write Source1 address to II tcb_a */
r0=dest;
dm(tcb + 2) = r0;    /* Write Dest1 address to EI slot in tcb_a */
r0=tcb + 7;
r1= b#10000000000000000000;
r0=r0 or r1;/* set PCI Bit */
dm(tcb + 4) = r0; /* Write tcb address to CP slot in tcb */

r0=0;
dm(DMAC10)=r0;       /* Clear DMA Control Register */
r0=b#00000000000000000000010100000111;
dm(DMAC10)=r0;   /* dma enable, Chain enable,int>ext, master mode
*/
r0=tcb + 7;
dm(CPEP0) =r0;         /* Load CP register*/

bit set imask EP0I;

rts;
```

Link Port DMA

There are two link ports DMA channels available on the ADSP-21161: channels 8 and 9. These two channels are shared with the SPI port. Channel 8 is assigned to link port 0 while channel 9 is assigned to link port 1. These bidirectional ports transfer data to other processors or link port peripherals.

The processor support a number of DMA modes for link port DMA. The following sections describe typical link port DMA processes:

- "Setting Up Link Port DMA" on page 6-86
- "Bootloading Through The Link Port" on page 6-88
- "Link Port Buffer Modes" on page 6-83
- "Link Port Channel Priority Modes" on page 6-83
- "Link Port Channel Transfer Modes" on page 6-85

Link Port Registers

The SYSCON and LCTL registers control the link ports operating modes for the I/O processor.

- Table A-18 on page A-60 lists all the bits in SYSCON.
- Table A-25 on page A-93 lists all the bits in LCTL.

The following bits control link port I/O processor modes. The control bits in the LCTL registers have a one cycle effect latency. Programs should not modify an active DMA channel's bits in the LCTL register other than to disable the channel by clearing the LxDEN bit. For information on verifying a channel's status with the DMASTAT register, see "Using I/O Processor Status" on page 6-121.

Some other bits in LCTL setup non-DMA link port features. For information on these features, see "Setting Link Port Modes" on page 9-5.

- **Link Port DMA Channel Priority Rotation Enable.** SYSCON Bit 20 (LDCPR). This bit enables (rotates if set, =1) or disables (fixed if cleared, =0) priority rotation between link port DMA channels 8 and 9.
- **Link–External Port DMA Channel Priority Rotation Enable.** SYSCON Bit 21 (PRROT). This bit enables (rotates if set, =1) or disables (fixed if cleared, =0) priority rotation between link port DMA channels 8 and 9 and external port DMA channels 10 to 13.
- **Link Port assignment for LBUFx.** LCTL Bits 9-0 and 23-22 correspond to link buffer 0. LCTL Bits 19-10 and 25-24 correspond to link buffer 1.
- **Link Buffer Enable. LCTL** Bits 0 and 10 (LxEN). This bit enables (if set, =1) or disables (if cleared, =0) the corresponding link buffer (LBUFx).
- **Link Buffer DMA Enable.** LCTL Bits 1 and 11 (LxDEN). This bit enables (if set, =1) or disables (if cleared, =0) DMA transfers for the corresponding link buffer (LBUFx).
- **Link Buffer DMA Chaining Enable.** LCTL Bits 2 and 12 (LxCHEN). This bit enables (if set, =1) or disables (if cleared, =0) DMA chaining for the corresponding link buffer (LBUFx).
- **Link Buffer Transfer Direction. LCTL** Bits 3 and 13 (LxTRAN). This bit selects the transfer direction (transmit if set, =1) (receive if cleared, =0) for the corresponding link buffer (LBUFx).

- **Link Buffer Extended Word Size.** LCTL Bits 4 and 14 (LxEXT). This bit selects the transfer extended word size (48-bit if set, =1) (32-bit if cleared, =0) for the corresponding link buffer (LBUFx). Programs must not change a buffer's LxEXT setting while the buffer is enabled.

Link Port Buffer Modes

The LABx bits in the LCTL register assign link ports to link buffers and enable link buffers. Bit 19 LAB0 enables link buffer 0 while Bit 20 LAB1 enables link buffer 1. To enable a link buffer, a program sets the buffer's LxEN bit in LCTL. To disable a link buffer, a program clears the buffer's LxEN bit in LCTL. The LCTL bit descriptions appear in "Link Port Buffer Control Registers (LCTL) Bit Definitions" on page A-93.

When the processor disables the buffer (LxEN transitions from high to low), the processor clears the corresponding LxSTATx and LRERRx bits.

Link Port Channel Priority Modes

The LDCPR and PRROT bits in the SYSCON register select priority levels for the link port buffers in relation to the priority of other link port buffer and the other I/O ports.

The Link Port DMA Channel Priority Rotation Enable (LDCPR) bit enables (rotates if set, =1) or disables (fixed if cleared, =0) priority rotation between link port DMA channels 8 and 9. Rotating priority distributes link port DMA channels' access to the I/O bus. When channel priority is rotating, the processor arbitrates I/O bus access between contending link port DMA channels, forcing the channels to take turns. When channel priorities fixed, the lower numbered link port DMA channel always has priority over the higher numbered channel when contending for I/O bus access.

When LDCPR is set (rotating priority), high priority shifts to a new channel after each single-word transfer. The following steps illustrate this process:

1. At reset, link port channels have priority order—from high to low.
2. The link port performs a single transfer on channel 8.
3. The I/O processor rotates channel priority, changing it from 8 to 9.

ⓘ Even though the link port channel DMA priority can rotate, the interrupt priorities of all DMA channels are fixed.

When a program uses fixed priority for the link port DMA channels, the I/O processor assigns the higher priority to channel 8 and the lower priority to channel 9. For a list of all channel assignments, see Table 6-1 on page 6-13.

Programs can change the fixed priority order, assigning a different channel to the highest priority. The following example shows how to change the fixed priority sequence of the link port DMA channels:

1. Disable all link port DMA channels except the one immediately above the channel that is to have highest priority.
2. Select rotating priority by setting the LDCPR bit.
3. Cause at least one transfer to occur on the enabled channel.
4. Disable rotating priority and re-enable all of the link port DMA channels.

The channel immediately after the selected channel now has the highest fixed priority.

Programs can also rotate priority between the link port and external port DMA channels. The DMA Channel Priority Rotation Enable (PRROT) bit enables (rotates if set, =1) or disables (fixed if cleared, =0) priority rotation between link port DMA channels 8 and 9 and external port DMA channels 10 to 13.

Rotating priority distributes link port and external port DMA channels' access to the I/O bus. When channel priority is rotating, the processor arbitrates I/O bus access between contending link port and external port DMA channels, forcing the channel types to take turns. When channel priority is fixed, any link port DMA channel always has priority over any external port DMA channel when contending for I/O bus access.

Link Port Channel Transfer Modes

The LxDEN, LxCHEN, LxTRAN, and LxEXT bits in the LCTL register enable link port DMA, and chained DMA and select the transfer direction and format. The link DMA enable (LxDEN) and link Chained DMA enable (LxCHEN) bits work together to select a link port DMA channel's transfer mode. Table 6-24 lists the modes.

Table 6-24. Link Port DMA Enable Modes

LxCHEN	LxDEN	DMA Enable Mode Description
0	0	Channel disabled (chaining disabled, DMA disabled)
0	1	Single DMA mode (chaining disabled, DMA enabled)
1	0	Chain insertion mode (chaining enabled, DMA enabled, auto-chaining disabled); For more information, see "Chaining DMA Processes" on page 6-25.
1	1	Chained DMA mode (chaining enabled, DMA enabled, auto-chaining enabled)

Because link ports are bidirectional, the I/O processor uses the link Transmit select (LxTRAN) bit to determine the transfer direction (transmit or receive). Data flows from internal to external memory when in transmit mode. In transmit mode, the I/O processor fills the channel's LBUFx buffer when the channel's LxDEN bit is set.

The Link Extended Word Size (LxEXT) bit determines how the DMA channel accesses columns of internal memory. If LxEXT is set, the data is 40- or 48-bit words, and the I/O processor makes 3-column internal memory accesses. If LxEXT is cleared, the data is 32-bit words, and the I/O processor makes 2-column internal memory accesses. For more information, see "Memory Organization and Word Size" on page 5-25.

The LxEXT for the transfer overrides the Internal Memory Data Width (IMDWx) setting for the internal memory block.

Setting Up Link Port DMA

The method for setting up and starting an link port DMA sequence varies slightly with the transfer mode for the channel. For more information on DMA transfer modes, see "Link Port Channel Transfer Modes" on page 6-85.

The following sequence describes a typical external to internal DMA operation where an external device transfers a block of data into the processor's internal memory using a link port.

1. The processor or host (depending on the mode) assigns the DMA channel's link buffer to a link port using the channel's LABx bits in the LCTL register.
2. The processor or host (depending on the mode) enables the DMA channel's link buffer, setting the buffer's LxEN bit in the channel's LCTL register. The processor or host selects a words size (32- or 40/48-bits) using the LxEXT in the channel's LCTL register.

3. The processor or host (depending on the mode) writes the DMA channel's parameter registers (IILBx, IMLBx, and CLBx) and LCTL control register, initializing the channel for receive (LxTRAN=0).

4. The processor or host (depending on the mode) sets (=1) the channel's LxDEN bit enabling the DMA process.

5. The external device begins writing data to the LBUFx buffer through the link port.

6. The LBUFx buffer detects data is present and asserts an internal DMA request to the I/O processor.

7. The I/O processor grants the request and performs the internal DMA transfer, emptying the LBUFx buffer FIFO.

In general, the following sequence describes a typical internal to external DMA operation where an external device transfers a block of data from the processor's internal memory using a link port:

1. The processor or host (depending on the mode) assigns the DMA channel's link buffer to a link port using the channel's LABx bits in the LCTL register.

2. The processor or host (depending on the mode) enables the DMA channel's link buffer, setting the buffer's LxEN bit in the channel's LCTL register. The processor or host selects a words size (32- or 40/48-bits) using the LxEXT in the channel's LCTL register.

3. The processor or host (depending on the mode) writes the DMA channel's parameter registers (IILBx, IMLBx, and CLBx) and LCTL control register, initializing the channel for transmit (LxTRAN=1).

4. The processor or host (depending on the mode) sets (=1) the channel's LxDEN bit enabling the DMA process. Because this is a transmit, setting LxDEN automatically asserts an internal DMA request to the I/O processor.

5. The I/O processor grants the request and performs the internal DMA transfer, filling the LBUFx buffer's FIFO.
6. The external device begins reading data from the LBUFx buffer (through the link port).
7. The LBUFx buffer detects that there is room in the buffer (it is now partially empty) and asserts another internal DMA request to the I/O processor, continuing the process.

Bootloading Through The Link Port

One of the processor's booting modes is booting the processor through the link port. Link port booting uses DMA channel 8 of the I/O processor to transfer the instructions to internal memory. In this boot mode, the processor receives 4-bit wide data in link buffer 0.

After the boot process loads 256 words into memory locations 0x40000 through 0x400FF, the processor begins executing instructions. Because most processor programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. Analog Devices supplies loading routines (loader kernels) that load an entire program through the selected port. These routines come with the development tools. For more information on loader kernels, see the development tools documentation.

It is important to note that DMA channel differences between the ADSP-21161 and previous SHARC processors (ADSP-2106x) introduce some booting differences. Even with these differences, the ADSP-21161 supports the same boot capability and configuration as the ADSP-2106x processors. For link booting the ADSP-21161, the program sequencer automatically unmasks the DMA channel 8 interrupt, initializing the LIRPTL register to 0x00010000 and IMASK register to 0x00004003.

The processor determines the booting mode at reset from the EBOOT, LBOOT, and $\overline{\text{BMS}}$ pin inputs. When EBOOT=0, LBOOT=1, and $\overline{\text{BMS}}$=1, the processor boots through the link port. For a list showing how to select different boot modes, see the Boot Memory Select pin description in the table "Booting Modes" on page 13-72.

When using any of the power-up booting modes, address 0x0004 0004 should not contain a valid instruction since it is not executed during the booting sequence. A NOP or IDLE instruction should be placed at this location.

In link port booting, the processor gets boot data from another processor's link port or 4-bit wide external device after system powerup.

The external device must provide a clock signal to the link port assigned to link buffer 0. The clock can be any frequency, up to a maximum of the processor clock frequency. The clock's falling edges strobe the data into the link port. The most significant 4-bit nibble of the 48-bit instruction must be downloaded first.

Table 6-25 shows how the DMA channel 8 parameter registers are initialized at reset for EPROM booting. The count register (CLB0) is initialized to 0x0100 for transferring 256 words to internal memory. The LCTL register is overridden during link port booting to allow link buffer 0 to receive 48-bit data.

Table 6-25. DMA Channel 8 Parameter Register Initialization For Link Port Booting

Parameter Register	Initialization Value
IILB0	0x0004 0000
IMLB0	uninitialized (increment by 1 is automatic)
CLB0	0x0100 (256 instruction words)

Table 6-25. DMA Channel 8 Parameter Register Initialization For Link Port Booting (Cont'd)

Parameter Register	Initialization Value
CPLB0	uninitialized
GPLB0	uninitialized

In systems where multiple processors are not connected by the parallel external bus, booting can be accomplished from a single source through the link ports. To simultaneously boot all of the processors, a parallel common connection should be made to link buffer 0 on each of the processors. If only a daisy chain connection exists between the processors' link ports, then each processor can boot the next one in turn. Link buffer 0 must always be used for booting.

Link Port DMA Programming Examples

This section provides two programming examples written for the ADSP-21161 processor. The example shown in Listing 6-3 demonstrates how the I/O processor uses DMA chaining to read from the link port receive buffer and write to the link port transmit buffer. The example shown in Listing 6-4 demonstrates how the I/O processor uses DMA to read from the link port receive buffer and write to the link port transmit buffer after an interrupt.

Listing 6-3. DMA-Chained Link Loopback Example

```
/*____________ ADSP-21161 DMA-Chained LINK Loopback Example
This example shows an internally looped-back link port 32-bit
transfer. DMA is used to transfer the data to and from the buff-
ers. Loopback is achieved by assigning the transmit and receive
link buffers to the same port (Port 0). ______________________*/

#include "def21161.h"
#define N 8

.section/pm seg_rth; /*Reset vector from ldf file*/
nop;
jump start;

.section/dm seg_dmda;/*Data section from ldf file*/
.var source[N]= 0X11111111, 0X22222222, 0X33333333, 0X44444444,
0X55555555, 0X66666666, 0X77777777, 0X88888888;
.var dest[N];
.var txtcb_source[8]=0,0,0,0,0,N,1,source; /*DMA TCB settings*/
.var rxtcb_dest[8]=0,0,0,0,0,N,1,dest; /*DMA TCB settings*/

/*_____________________Main Routine________________________*/
.section/pm seg_pmco;/*Main code section described in .ldf file*/

start:
ustat1 = dm(SYSCON);
bit clr ustat1 BHD;       /*Disable Buffer Hang*/
dm(SYSCON) = ustat1;
imask = 0;              /*Clear IMASK and IRPTL registers*/
irptl = 0;
bit set imask LPISUMI;   /*Enable Link port interrupts*/
bit set lirptl LP1MSK;   /*Enable Link port 1 interrupt*/
bit set mode1 IRPTEN;   /*Enable global interrupts*/
```

```
r0 = 0; dm(LCTL) = r0;
ustat1=dm(LCTL);

/*LCTL REGISTER-->LBUF0=TX, LBUF1=RX, 2x CLK RATE, LBUF 0 & 1
ENABLED, LBUF 0 & 1 -> PORT 0 DMA Enabled, DMA Chain Enabled*/
bit clr ustat1 L0TRAN | LAB0 | LAB1 | L0CLKD0 | L1CLKD0;
bit set ustat1 L1TRAN | L1EN | L0EN | L0CLKD1 | L1CLKD1 | L0DEN |
L1DEN | L0CHEN | L1CHEN;

dm(LCTL)=ustat1;

r1 = 0x0003FFFF;            /* CPX register mask */
r0 = txtcb_source + 7;     /* Get DMA chaining int. mem. ptr
                               with tx buf address */
r0 = r1 AND r0;             /* Mask the pointer */
r0 = BSET r0 BY 18;         /* Set the pci bit */
dm(txtcb_source + 4) = r0;  /* Write DMA transmit block chain
                                pointer to TCB buffer */
dm(CPLB1) = r0;          /* Transmit blk chain ptr, init.LP1
                              DMA transfers */

r0 = rxtcb_dest + 7;
r0 = r1 AND r0;/* Mask the pointer */
r0 = BSET r0 BY 18;/* Set the pci bit */
dm(rxtcb_dest + 4) = r0;       /* Write DMA receive block chain
                                   pointer to TCB buffer*/
dm(CPLB0) = r0;                /* Receive block chain pointer,
                                   Initiate LP0 DMA transfers */

wait: idle;
jump wait;
```

Listing 6-4. Interrupt DMA-Driven Link Loopback Example

```
/*_____________________________________________________________
  ADSP-21161 Interrupt DMA-Driven LINK Loopback Example
This example shows an internally looped-back link port 32-bit
transfer. DMA is used to write to and read from the buffers.
Loopback is achieved by assigning the transmit and receive link
buffers to the same port. (Port 0)
_____________________________________________________________*/

#include "def21161.h"
#define N 8

.section/pm seg_rth; /*Reset vector from ldf file*/
nop;
jump start;

.section/dm seg_dmda;   /*Data segment section from ldf file*/
.var source[N]= 0X11111111, 0X22222222, 0X33333333, 0X44444444,
0X55555555, 0X66666666, 0X77777777, 0X88888888;

.var dest[N];

.section/pm lp1i_svc;   /*Link Port 1 Vector from ldf file*/
jump lpISR1;rti;rti;rti;

.section/pm lp0i_svc;   /*Link Port 0 Vector from ldf file*/
jump lpISR0;rti;rti;rti;
```

```
/*_____________________Main Routine________________________*/
.section/pm seg_pmco;/*Main code section from ldf file*/
start:
r0 = 0; DM(LCTL) = r0;

r0=source;
dm(IILB0)=r0;   /*Set DMA tx index to start of source buffer*/
r0=dest;
dm(IILB1)=r0;   /*Set DMA rx index to start of destination
buffer*/

r0=@source;
dm(CLB0)=r0;   /*Set DMA count to length of data buffers*/
dm(CLB1)=r0;

r0=1;
dm(IMLB0)=r0;  /*Set DMA modify (stride) to 1*/
dm(IMLB1)=r0;

ustat1 = dm(SYSCON);   /*Disable Buffer Hang*/
bit clr ustat1 BHD;
dm(SYSCON) = ustat1;

imask = 0; lirptl = 0;

/*Enable Global,Link Port and Link Port Buffer 1 interrupt */
bit set imask LPISUMI;
bit set lirptl LP1MSK | LP0MSK;
bit set mode1 IRPTEN | CBUFEN;

ustat1=dm(LCTL);

/*LCTL Register-->LBUF1=TX, LBUF0=RX, 1/4x CCLK RATE, LBUF 0 & 1
ENABLED, LBUF 0 & 1 -> PORT 0 Link buffer 0 & 1 DMA Enabled*/
```

```
bit clr ustat1 L1TRAN | L0CLKD0 | L1CLKD0 | LAB0 | LAB1;
bit set ustat1 L0TRAN | L1EN | L0EN | L0CLKD1 | L1CLKD1 | L0DEN |
L1DEN;

dm(LCTL)=ustat1;

wait:
idle;
jump wait;

lpISR0: rti;
lpISR1: rti;
```

Serial Port DMA

Serial Port DMA provides a mechanism for receiving or transmitting an entire block of serial data before an interrupt is generated. The processor's on-chip DMA controller handles DMA transfers, allowing the processor core to continue running until the entire block of data is transmitted or received. There are eight serial port channels available on the ADSP-21161 for DMA transfers: channels 0 through 7. Each of the serial port channels can be configured to transmit or receive data. The A path for each sport allows expansion or compression of data.

The processor supports a number of DMA modes for serial port DMA. The following sections describe typical serial port DMA processes:

- "Setting Up Serial Port DMA" on page 6-100
- "Serial Port Buffer Modes" on page 6-97
- "Serial Port Channel Priority Modes" on page 6-99
- "Serial Port Channel Transfer Modes" on page 6-99

Serial Port Registers

The SPCTLx registers control the serial port operating mode for the I/O processor. Figure 6-13 lists all the bits in SPCTLx.

The following bits control serial port I/O processor modes. The control bits in the SPCTLx registers have a one cycle effect latency. Programs should not modify an active DMA channel's bits in the SPCTLx registers; other than to disable the channel by clearing the SDEN bit. To change an inactive serial port's operating mode, programs should clear a serial port's control register before writing new settings to the control register. For information on verifying a channel's status with the DMASTAT register, see "Using I/O Processor Status" on page 6-121.

Some other bits in SPCTLx setup non-DMA serial port features. For information on these features, see "Serial Port DMA" on page 6-95.

- **Serial Port Enable.** SPCTLx Bit 0 (SPEN_A) and Bit 24 (SPEN_B). These bits enables (if set, =1) or disables (if cleared, =0) the corresponding serial port. SPEN_A corresponds to the A channel (companding). SPEN_B corresponds to the B channel (no companding). You can enable one or both of these bits.

- **Data Type Select.** SPCTLx Bits 2-1 (DTYPE). These bits select the data type formatting for normal and multi-channel reception as follows: (normal/multichannel= format) 00/x0=Right-justify and zero-fill unused MSBs, 01/x1=Right-justify and sign-extend unused MSBs, 10/0x=Compand using µ-law, 11/1x=Compand using A-law.

- **Serial Word Endian Select.** SPCTLx Bit 3 (SENDN). This bit selects little endian words (LSB first, if set, =1) or big endian words (MSB first, if cleared, =0).

- **Serial Word Length Select. SPCTLx** Bits 8-4 (SLEN). These bits select the word length –1 in bits. Word sizes can be from 3-bit (SLEN=2) to 32-bit (SLEN=31).

- **16-bit to 32-bit Word Packing Enable.** SPCTLx Bit 9 (PACK). This bit enables (if set, =1) or disables (if cleared, =0) 16- to 32-bit word packing.
- **Serial Port DMA Enable.** SPCTLx Bit 18 (SDEN_A) and Bit 20 (SDEN_B).These bits enable (if set, =1) or disable (if cleared, =0) the serial port's A or B channel DMA.
- **Serial Port DMA Chaining Enable.** SPCTLx Bit 19 (SCHEN_A) and Bit 21 (SCHEN_B). These bits enable (if set, =1) or disables (if cleared, =0) the serial port's A or B channel DMA chaining.

Serial Port Buffer Modes

The SPEN, SENDN, SLEN, and PACK bits in the SPCTLx registers enable the serial port and select the transfer format.

To enable a serial port transmit or receive buffer, a program sets the buffer's SPEN bit in the SPCTLx register. To disable a serial port transmit or receive buffer, a program clears the buffer's SPEN_A or SPEN_B bit in the SPCTLx register.

If a serial port buffer is enabled and DMA for that channel is not enabled, the serial port is in single-word, interrupt-driven transfer mode. For more information, see "Using I/O Processor Status" on page 6-121.

Each serial port buffer allows independent settings for the three transfer format features: bit order, word length, and word packing. For transferring little endian words (LSB first, if set, =1) to or from little endian devices, the serial port buffers have a Serial Word Endian Select (SENDN) bit. This bit selects little endian words (LSB first, if set, =1) or big endian words (MSB first, if cleared, =0). The Serial Word Length Select (SLEN) bit field selects the transfer word length (-1) in bits. Word sizes can be from 3-bit (SLEN=2) to 32-bit (SLEN=31).

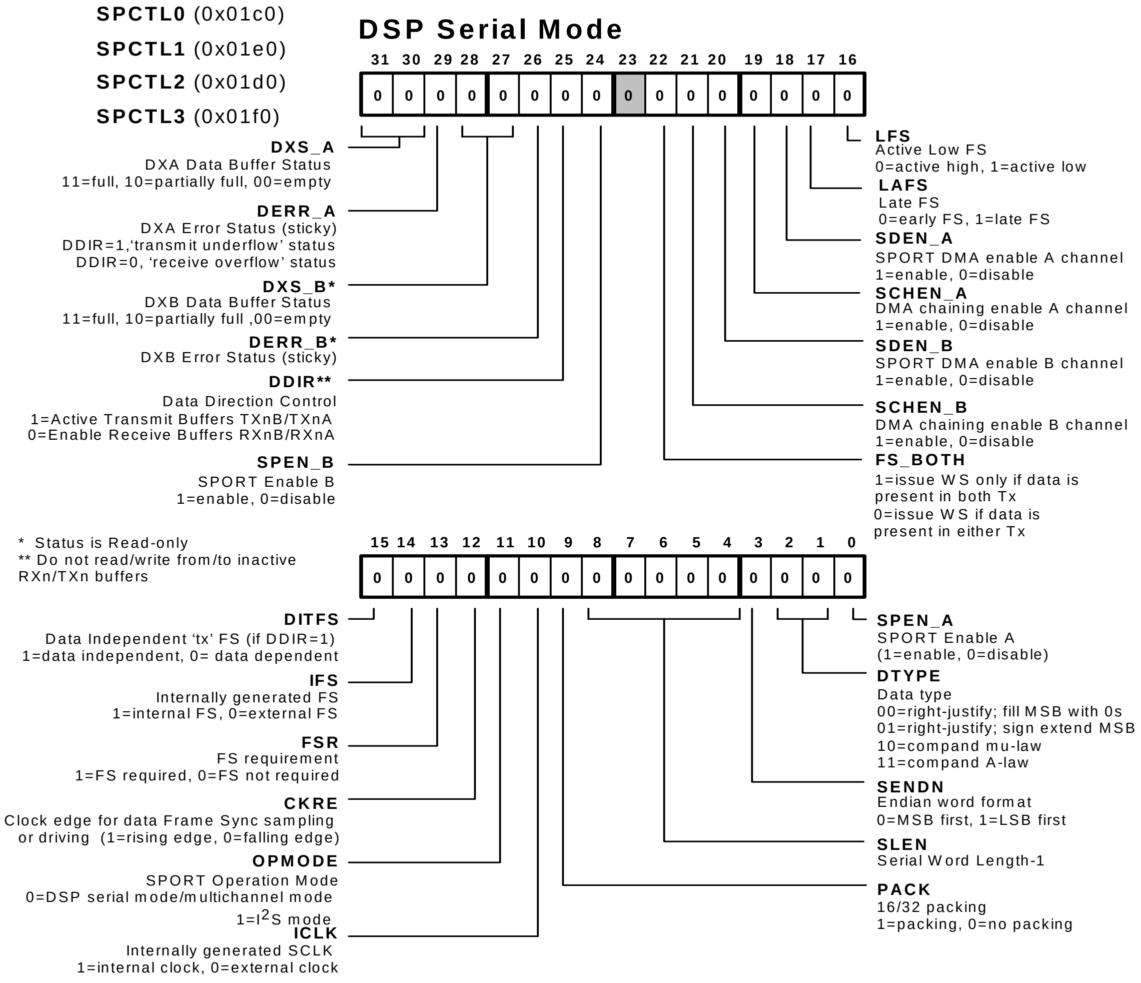

SPCTL0 (0x01c0)
SPCTL1 (0x01e0)
SPCTL2 (0x01d0)
SPCTL3 (0x01f0)
DSP Serial Mode
DXS_A
DXA Data Buffer Status
11=full, 10=partially full, 00=empty
DERR_A
DXA Error Status (sticky)
DDIR=1,'transmit underflow' status
DDIR=0, 'receive overflow' status
DXS_B*
DXB Data Buffer Status
11=full, 10=partially full ,00=empty
DERR_B*
DXB Error Status (sticky)
DDIR**
Data Direction Control
1=Active Transmit Buffers TXnB/TXnA
0=Enable Receive Buffers RXnB/RXnA
SPEN_B
SPORT Enable B
1=enable, 0=disable
LFS
Active Low FS
0=active high, 1=active low
LAFS
Late FS
0=early FS, 1=late FS
SDEN_A
SPORT DMA enable A channel
1=enable, 0=disable
SCHEN_A
DMA chaining enable A channel
1=enable, 0=disable
SDEN_B
SPORT DMA enable B channel
1=enable, 0=disable
SCHEN_B
DMA chaining enable B channel
1=enable, 0=disable
FS_BOTH
1=issue WS only if data is present in both Tx
0=issue WS if data is present in either Tx
* Status is Read-only
** Do not read/write from/to inactive RXn/TXn buffers
DITFS
Data Independent 'tx' FS (if DDIR=1)
1=data independent, 0= data dependent
IFS
Internally generated FS
1=internal FS, 0=external FS
FSR
FS requirement
1=FS required, 0=FS not required
CKRE
Clock edge for data Frame Sync sampling or driving (1=rising edge, 0=falling edge)
OPMODE
SPORT Operation Mode
0=DSP serial mode/multichannel mode
1=I2S mode
ICLK
Internally generated SCLK
1=internal clock, 0=external clock
SPEN_A
SPORT Enable A
(1=enable, 0=disable)
DTYPE
Data type
00=right-justify; fill MSB with 0s
01=right-justify; sign extend MSB
10=compand mu-law
11=compand A-law
SENDN
Endian word format
0=MSB first, 1=LSB first
SLEN
Serial Word Length-1
PACK
16/32 packing
1=packing, 0=no packing

Figure 6-13. SPCTLx Register – DSP Serial Mode

If the serial word length is 16-bits or smaller, the serial port can pack two of these words into the serial port buffer. The 16-bit to 32-bit word Packing Enable (PACK) bit can enable this packing because the I/O processor performs 32-bit transfers between the serial port buffers and processor memory.

In addition to selecting the endian, length, and packing modes for serial port processor transfers, programs must indicate the type of data in the transfer, using the Data Type (DTYPE) bit. For more information, see "Serial Port Channel Transfer Modes" on page 6-99.

Serial Port Channel Priority Modes

Serial port DMA transfers always take priority over external port, SPI port, or link port DMA transfers. For more information on prioritization operations, see "Managing DMA Channel Priority" on page 6-22.

Serial Port Channel Transfer Modes

The SDEN_A, SDEN_B, SCHEN_A, SCHEN_B, and DTYPE bits in the SPCTLx register enable serial port DMA, chained DMA, and select the format. The DMA enable (SDEN) and Chained DMA enable (SCHEN) bits work together to select a serial port DMA channel's transfer mode. Table 6-26 lists the modes.

Table 6-26. Serial Port DMA Enable Modes

SCHEN A or B	SDEN A or B	DMA Enable Mode Description
0	0	Channel disabled (chaining disabled, DMA disabled)
0	1	Single DMA mode (chaining disabled, DMA enabled)
1	0	Chain insertion mode (chaining enabled, DMA enabled, auto-chaining disabled); For more information, see "Chaining DMA Processes" on page 6-25.
1	1	Chained DMA mode (chaining enabled, DMA enabled, auto-chaining enabled)

Because serial port buffers are bidirectional, the I/O processor does not need an indicator to determine the transfer direction (transmit or receive). Data flows from internal to external devices using a transmit (TXx) buffer. When transmitting serial data as DMA, the I/O processor fills the channel's TXx buffer when the channel's SDEN bit is set.

Setting Up Serial Port DMA

The method for setting up and starting an serial port DMA sequence varies slightly with the transfer mode for the channel. For more information on DMA transfer modes, see "Serial Port Channel Transfer Modes" on page 6-99.

In general, the following sequence describes a typical external to internal DMA operation where an external device transfers a block of data into the processor's internal memory using a serial port:

1. The processor or host (depending on the mode) enables the DMA channel's serial port, setting the port's SPEN_A or SPEN_B bit in the port's SPCTLx register. The processor or host selects a words size using the DTYPE in the port's SPCTLx register. When you clear DDIR(= 0), the program configures SPORT A and B data pins as receivers and activates the RXA and RXB registers.

2. The processor or host (depending on the mode) writes to the DMA channel's parameter registers (IIx, IMx, and Cx) and SPCTLx control register, initializing the channel for receive.

3. The processor or host (depending on the mode) sets (=1) the channel's SDEN_A or SDEN_B bit enabling the DMA process.

4. The external device begins writing data to the RXx buffer through the serial port.

5. The RXx buffer detects data is present and asserts an internal DMA request to the I/O processor.
6. The I/O processor grants the request and performs the internal DMA transfer, emptying the RXx buffer.

In general, the following sequence describes a typical internal to external DMA operation where an external device transfers a block of data from the processor's internal memory using a serial port:

1. The processor or host (depending on the mode) enables the DMA channel's serial port, setting the port's SPEN bit in the port's SPCTLx register. The processor or host selects a words size using the DTYPE in the port's SPCTLx register. The DDIR bit is set (=1) to enable the serial interface as a transmitter. The program activates the TX buffers allowing data to transmit out of the SPORT A and B data pins.
2. The processor or host (depending on the mode) writes to the DMA channel's parameter registers (IIx, IMx, and Cx) and SPCTLx control register, initializing the channel for transmit.
3. The processor or host (depending on the mode) sets (=1) the channel's SDEN bit enabling the DMA process. Because this is a transmit, setting SDEN_A or SDEN_B automatically asserts an internal DMA request to the I/O processor.
4. The I/O processor grants the request and performs the internal DMA transfer, filling the TXx buffer.
5. The external device begins reading data from the TXx buffer through the serial port.
6. The TXx buffer detects that there is room in the buffer because it is now "partially empty" and asserts another internal DMA request to the I/O processor, continuing the process.

When programming the serial port channel (A or B) as a transmitter only the corresponding TXA and TXB become active, while the receive buffers RXA and RXB remain inactive. Similarly, when the SPORT channel A and B is programmed as receive only the corresponding RX0A and RX0B is activated.

When performing core driven transfers, programs must write to the proper buffer depending on the direction setting in the SPCTL register (DDIR). For DMA-driven transfers the serial port logic performs the data transfer from internal memory to/from the appropriate buffer depending on the DDIR bit setting.

If the inactive SPORT data buffers are read or written to by core while the port is already being enabled, the SPORT does not operate correctly. If, for example, the SPORT is programmed to be a transmitter, while at the same time, the core reads from the receive buffer of the same SPORT, the core hangs, just as it would if it was reading an empty buffer which was currently active. This locks up the core permanently until the SPORT is reset.

The program must set the direction bit along with serial port enable and DMA enable bits before initiating any operations on the SPORT data buffers. If the processor operates on the inactive transmit or receive buffers while the SPORT is enabled it can cause unpredictable results.

SPORT DMA Programming Examples

This section provides two programming examples written for the ADSP-21161 processor. The example shown in Listing 6-5 demonstrates how the I/O processor uses DMA chaining to read from the SPORT receive buffer and write to the SPORT transmit buffer. The example shown in Listing 6-6 demonstrates how the I/O processor uses DMA to read from the SPORT receive buffer and write to the SPORT transmit buffer.

Listing 6-5. DMA-Chained Sport Loopback Example

```
/*______________________________________________________________
ADSP-21161 DMA-Chained SPORT Loopback Example
This example shows an internally looped-back SPORT 32-bit trans-
fer. The transfer buffer (TX2A) and receive buffer (RX0A) are
both handled via DMA chaining.
______________________________________________________________*/
#include "def21161.h"
#define N 8

.section/pm seg_rth;     /*Reset vector from ldf file*/
nop;
jump start;

.section/dm seg_dmda;
.var source[N]= 0X11111111, 0X22222222, 0X33333333, 0X44444444,
0X55555555, 0X66666666, 0X77777777, 0X88888888;

.var dest[N];

.var txtcb[8] = 0,0,0,0,0,N,1,source;   /*DMA tcb settings*/
.var rxtcb[8] = 0,0,0,0,0,N,1,dest;

.section/pm sp0i_svc;
jump IRQ; rti;rti;rti;

.section/pm sp2i_svc;
jump IRQ; rti;rti;rti;

/*----------------Main Routine---------------------------*/
.section/pm seg_pmco;
start:
```

```
ustat3=dm(SYSCON);
bit clr ustat3 BHD;         /*Disable Buffer Hang*/
dm(SYSCON)=ustat3;

bit set imask SP0I |SP2I;       /*Unmask SPORT 0 & 2 Interrupts*/
bit set mode1 CBUFEN | IRPTEN;  /*Enable Circ Buffers &
                                   Interupts*/

r0 = 0x00001000;
/*Set the SPL bit in the SPxxMCTL register to enable loopback*/
dm(SP02MCTL)=r0;

r0 = 0x0;       /*Externally generated clock and framesync*/
dm(DIV0) = r0;
r0 = 0x000c21f1;

/*Set bits SPEN_A, SLEN0-4, FSR--enable the A channel, set the
word length to 32 bits, require frame synch, and enable DMA and
DMA Chaining.*/
dm(SPCTL0)=r0;

r0=0x00270004;
/*TCLKDIV=[FCCLK(96Mhz)/2xFSCLK((19.2Mhz)]-1=0x0004*/
/*TFSDIV=[FSCLK(9.6Mhz)/TFS(.24Mhz)]-1=0x0027*/
dm(DIV2)=r0;
r0=0x20c65f1;

/*Set bits SPEN_A, SLEN0-4, ICLK, IFS, FSR, DDIR--enable the A
channel, set the word length to 32 bits, generate internal frame-
synch and clock, require frame synch, set for transmit, and
enable DMA and DMA Chaining.*/
dm(SPCTL2)=r0;

r1=0x0003FFFF;       /*CPx register mask*/
```

```
r0=txtcb+7;           /*Get DMA chaining memory pntr containing
                       tx buff address*/
r0=r1 AND r0;         /*Mask the pointer*/
r0= BSET r0 by 18;    /*Set the PCI bit*/
dm(txtcb+4)=r0;      /*Write DMA transmit block chain pntr to
                       TCB buffer*/
dm(CP2A)=r0;         /*Transmit block chain pointer, init SP2
                        DMA transfers*/

r0=rxtcb+7;
r0=r1 AND r0;
r0=BSET r0 by 18;
dm(rxtcb+4)=r0;
dm(CP0A)=r0;        /*Initiate SP0 DMA transfers*/

wait: idle;
jump wait;

IRQ: rti;
```

Listing 6-6. DMA-Driven Sport Loopback Example

```
/*____________________________________________________________
ADSP-21161 DMA-Driven SPORT Loopback Example
This example shows an internally looped-back SPORT 32-bit trans-
fer. The transfer buffer (TX2A) and receive buffer (RX0A) are
both handled via DMA.
____________________________________________________________*/
#include "def21161.h"
#define N 8

.section/pm seg_rth; /*Reset vector from ldf file*/
nop;
jump start;
```

```
.section/dm seg_dmda;
.var source[N]= 0X11111111, 0X22222222, 0X33333333, 0X44444444,
0X55555555, 0X66666666, 0X77777777, 0X88888888;

.var dest[N];

.section/pm sp0i_svc;
jump IRQ; rti;rti;rti;

.section/pm sp2i_svc;
jump IRQ; rti;rti;rti;

/*-----------------Main Routine----------------------------*/
.section/pm seg_pmco;
start:
r0=source;
dm(II2A)=r0;   /*Set DMA tx index to start of source buffer*/
r0=dest;
dm(II0A)=r0;   /*Set DMA rx index to start of dest buffer*/

r0=@source;
dm(C0A)=r0;   /*Set DMA count to length of data buffers*/
dm(C2A)=r0;

r0=1;
dm(IM0A)=r0;   /*Set DMA modify (stride) to 1.*/
dm(IM2A)=r0;

ustat3=dm(SYSCON);
bit clr ustat3 BHD;   /*Disable Core Buffer Hang*/
dm(SYSCON)=ustat3;

bit set imask SP0I |SP2I;   /*Unmask Sport 0&2 interrupts*/
bit set mode1 CBUFEN | IRPTEN;   /*Enable Circ. Buffer & Global
                                    Inters*/

r0 = 0x00001000;
/*Set the SPL bit in the SPxxMCTL register to enable loopback*/
dm(SP02MCTL)=r0;

r0 = 0x0;   /*Externally generated clock and framesync*/
dm(DIV0) = r0;
r0 = 0x000421f1;

/*Set bits SPEN_A, SLEN=32, FSR--enable the A channel, set the
word length to 32 bits, and require frame synch.*/
dm(SPCTL0)=r0;

r0=0x00270004;
/*TCLKDIV=[FCCLK(96Mhz)/2xFSCLK((19.2Mhz)]-1=0x0004*/
/*TFSDIV=[FSCLK(9.6Mhz)/TFS(.24Mhz)]-1=0x0027*/
dm(DIV2)=r0;
r0=0x20465f1;

/*Set bits SPEN_A, SLEN=32, ICLK, IFS, FSR, DDIR--enable the A
channel, set the word length to 32 bits, generate internal frame-
synch and clock, require frame synch, and set for transmit.*/
dm(SPCTL2)=r0;

wait: idle;
jump wait;

IRQ: rti;
```

SPI Port DMA

There are two DMA channels available on the ADSP-21161 for SPI port transfers: channels 8 and 9. These two channels are shared with the link port. Channel 8 which is assigned to SPI receive buffer SPIRX handles receive data while channel 9 which is assigned to SPI transmit buffer SPITX handles transmit data.

The following sections describe typical SPI port DMA processes:

- "Setting up SPl Port DMA" on page 6-112
- "Bootloading Through the SPI Port" on page 6-113
- "SPI Port Buffer" on page 6-109
- "SPI DMA Channel Priority" on page 6-112

SPI Port Registers

The SPICTL register controls the SPI port operating mode for the I/O processor. Figure 6-14 lists all the bits in SPICTL.

The following bits control SPI port I/O processor modes. The control bits in the SPICTL registers have a one cycle effect latency. Programs should not modify an active DMA channel's SPICTL register; other than to disable the channel by clearing the SPIEN bit. For information on verifying a channel's status with the DMASTAT register, see "DMA Channel Status Register (DMASTAT)" on page A-90. For information on SPI port status, see "SPI Port Status Register" on page A-115.

The following bits in SPICTL setup DMA SPI port features:

- **SPI Port Enable.** SPICTL Bit 0 (SPIEN). This bit enables (if set, =1) or disables (if cleared, =0) the SPI port.
- **Data Format.** SPICTL Bits 6 (DF). This bit selects the data format. When set (=1), the MSB is sent/received first. When cleared (=0), the LSB is sent/received first.
- **SPI Word Length Select.** SPICTL Bits 8-7 (WL). These bits select the word length. Word sizes can be 8-bit (WL = 00), 16-bit (WL = 01) or 32-bit (WL = 11).
- **Word Packing Enable. SPICTL** Bit 28 (PACKEN). This bit enables (if set, =1) 8- to 32-bit packing or (if cleared, =0) disables the packing. If this bit is enabled, the receiver packs the received byte whereas the transmitter unpacks the data before sending it. For more information on packing formats, see "SPI Word Packing" on page 11-24. This bit should be 1 only in 8-bit data word length (**WL=00**).
- **SPI Port Receive DMA Enable. SPICTL** Bit 27 (RDMAEN). This bit enables (if set, =1) or disables (if cleared, =0) DMA transfers from the receive data buffer. At SPI boot this bit is set to 1 to enable the booting process through the SPI port.
- **SPI Port Transmit DMA Enable.** SPICTL Bit 13 (TDMAEN). This bit enables (if set, =1) or disables (if cleared, =0) DMA transfers to the transmit data buffer. At SPI boot this bit is 0.

SPI Port Buffer

The SPIEN bit in the SPICTL register enables the SPI port. The SPI port shares channel 8 with link buffer 0 for the receive function. It shares channel 9 with link buffer 1 for the transmit function. Data is loaded into SPITX from internal memory by the DMA controller. Once the SPI is

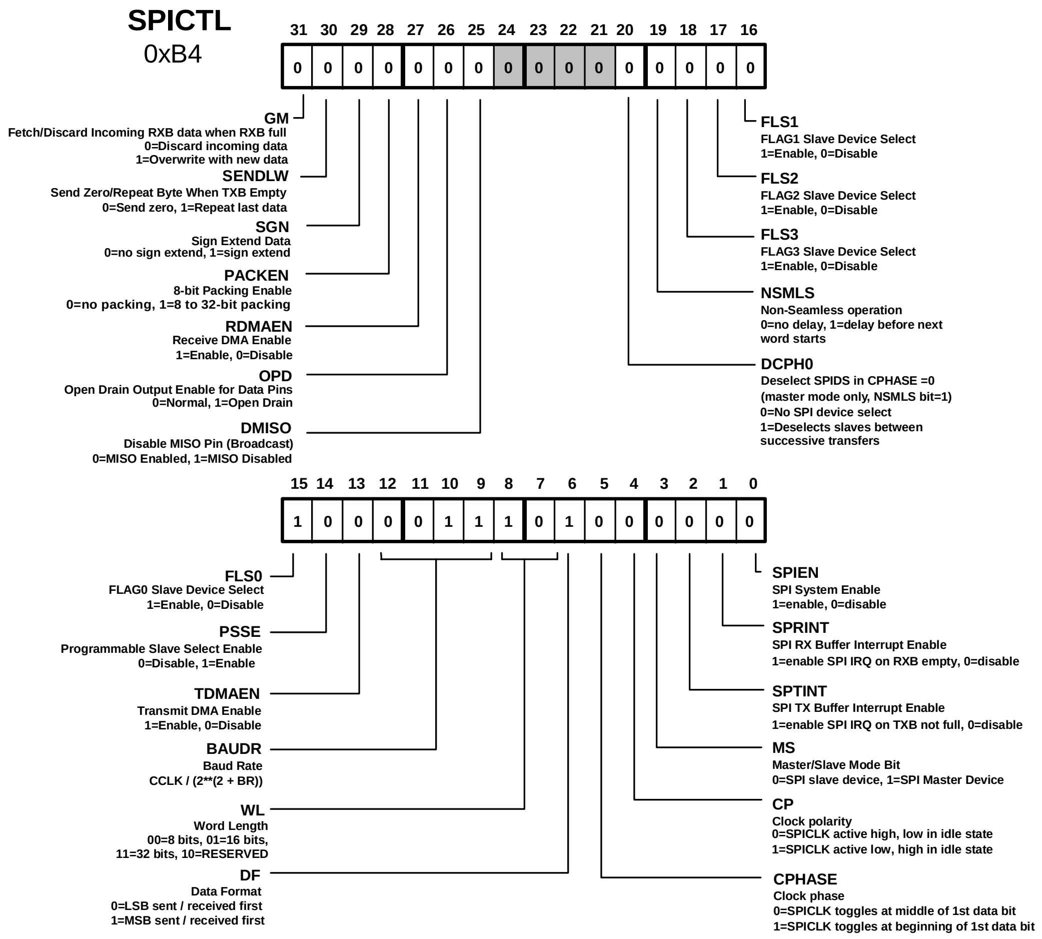

SPICTL
0xB4
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
GM
Fetch/Discard Incoming RXB data when RXB full
0=Discard incoming data
1=Overwrite with new data
SENDLW
Send Zero/Repeat Byte When TXB Empty
0=Send zero, 1=Repeat last data
SGN
Sign Extend Data
0=no sign extend, 1=sign extend
PACKEN
8-bit Packing Enable
0=no packing, 1=8 to 32-bit packing
RDMAEN
Receive DMA Enable
1=Enable, 0=Disable
OPD
Open Drain Output Enable for Data Pins
0=Normal, 1=Open Drain
DMISO
Disable MISO Pin (Broadcast)
0=MISO Enabled, 1=MISO Disabled
FLS1
FLAG1 Slave Device Select
1=Enable, 0=Disable
FLS2
FLAG2 Slave Device Select
1=Enable, 0=Disable
FLS3
FLAG3 Slave Device Select
1=Enable, 0=Disable
NSMLS
Non-Seamless operation
0=no delay, 1=delay before next word starts
DCPH0
Deselect SPIDS in CPHASE =0
(master mode only, NSMLS bit=1)
0=No SPI device select
1=Deselects slaves between successive transfers
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0
FLS0
FLAG0 Slave Device Select
1=Enable, 0=Disable
PSSE
Programmable Slave Select Enable
0=Disable, 1=Enable
TDMAEN
Transmit DMA Enable
1=Enable, 0=Disable
BAUDR
Baud Rate
CCLK / (2**(2 + BR))
WL
Word Length
00=8 bits, 01=16 bits,
11=32 bits, 10=RESERVED
DF
Data Format
0=LSB sent / received first
1=MSB sent / received first
SPIEN
SPI System Enable
1=enable, 0=disable
SPRINT
SPI RX Buffer Interrupt Enable
1=enable SPI IRQ on RXB empty, 0=disable
SPTINT
SPI TX Buffer Interrupt Enable
1=enable SPI IRQ on TXB not full, 0=disable
MS
Master/Slave Mode Bit
0=SPI slave device, 1=SPI Master Device
CP
Clock polarity
0=SPICLK active high, low in idle state
1=SPICLK active low, high in idle state
CPHASE
Clock phase
0=SPICLK toggles at middle of 1st data bit
1=SPICLK toggles at beginning of 1st data bit

Figure 6-14. SPICTL Register

enabled, data in SPITX is automatically loaded into the transmit shift register. After a word is received completely in the receive shift register, it is automatically transferred to the SPIRX. The data in SPIRX is moved into internal memory by the DMA controller All DMA transfers are 32-bit words. To disable the SPI port, clear the SPIEN bit in the SPICTL register, which also clears the status of the buffers in the SPISTAT register. The bits in the SPI control register (SPICTL) are shown in Figure A-38 on page A-120.

If the SPI port is enabled without enabling DMA, the SPI port is either in single-word, interrupt-driven data transfer mode (if the corresponding interrupt enable bits in the SPICTL is set) or is in core-driven data transfer mode. The software must do the data transfers to the SPI data buffers. For more information on the different SPI transfer modes, see "Master Mode Operation" on page 11-25. For more information on transfer status, see "Using I/O Processor Status" on page 6-121.

The SPI allows independent settings for the three transfer format features: bit order, word length, and word packing.

The SPI port buffer has a SPI data format (DF) bit, which when cleared (=0) can transmit data as little endian words (LSB first) to or from little endian devices. This bit selects big endian words (MSB first, if set, =1) or little endian words (LSB first, if cleared, =0).

The SPI Word Length (WL) bit field selects the transfer word length. Word sizes can be 8-bit (WL = 00), 16-bit (WL = 01) or 32-bit (WL = 11). If the SPI word length is 8-bits or smaller, the SPI port can pack two of these words into the SPI port data buffer. The 8-bit to 32-bit Word Packing Enable (PACKEN) bit can enable this packing because the I/O processor performs 32-bit transfers between the SPI port buffer and processor memory. If this bit is enabled, the transmitter unpacks the data before sending it, while the receiver packs the received byte. For more information on packing formats, see "SPI Word Packing" on page 11-24. This bit should be 1 only in 8-bit data word length (WL= 00).

SPI DMA Channel Priority

SPI shares DMA channels with the link port. The receive DMA is shared with link port 0 DMA while the transmit DMA is shared with link port 1. SPI port DMA transfers have the same priority as link port DMA transfers. If SPI DMAs are enabled, you should disable the link port DMAs. For more information on prioritization operations, see "Managing DMA Channel Priority" on page 6-22.

Setting up SPI Port DMA

In general, the following sequence describes a typical external to internal DMA operation where an external device transfers a block of data into the processor's internal memory using a SPI port:

1. The processor or host (depending on the mode) enables the DMA channel's serial port, setting the port's SPIEN bit in the port's SPICTL register. The processor or host selects a words size using the WL bits in the port's SPICTL register.
2. The processor or host (depending on the mode) writes the DMA channel's parameter registers (IISRx, IMSRx, and CSRx) and SPICTL control register, initializing the channel for receive.
3. Depending on the mode, the processor or host sets the channel's RDMAEN bit to 1 enabling the DMA process.
4. The external device begins writing data to the SPIRX buffer through the SPI port.
5. The SPIRX buffer detects data is present and asserts an internal DMA request to the I/O processor.
6. The I/O processor grants the request and performs the internal DMA transfer, emptying the SPIRX buffer.

In general, the following sequence describes a typical internal to external DMA operation where an external device transfers a block of data from the processor's internal memory using a serial port:

1. The processor or host (depending on the mode) enables the DMA channel's serial port, setting the port's SPIEN bit in the port's SPICTL register. The processor or host selects a words size using the WL bits in the port's SPICTL register.
2. The processor or host (depending on the mode) writes the DMA channel's parameter registers (IISTx, IMSTx, and CSTx) and SPICTL control register, initializing the channel for transmit.
3. The processor or host (depending on the mode) sets the channel's TDMAEN bit to 1 enabling the DMA process. Because this is a transmit, setting TDMAEN automatically asserts an internal DMA request to the I/O processor.
4. The I/O processor grants the request and performs the internal DMA transfer, filling the SPITX buffer.
5. The external device begins reading data from the SPITX buffer through the SPI port.
6. The SPITX buffer detects that there is room in the buffer because it is now partially empty and asserts another internal DMA request to the I/O processor, continuing the process.

Bootloading Through the SPI Port

One of the processor's booting modes is booting the processor through the SPI port. SPI port booting uses DMA channel 8 of the I/O processor to transfer the instructions to internal memory. In this boot mode, the processor receives 32-bit wide data in the SPIRX buffer.

During the boot process the program loads 256 words into memory locations 0x40000 through 0x400FF. The processor subsequently begins executing instructions. Because most programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. Analog Devices supplies loading routines (loader kernels) that load an entire program through the selected port. These routines come with the development tools. For more information on loader kernels, see the development tools documentation.

For SPI booting the ADSP-21161, the Program sequencer automatically unmasks the DMA channel 8 interrupt, initializing the `SPICTL` register to 0x0A001F81 and `IMASK` register to 0x00004003.

The processor determines the booting mode at reset from the `EBOOT`, `LBOOT`, and $\overline{\texttt{BMS}}$ pin inputs. When `EBOOT`=0, `LBOOT`=1, and $\overline{\texttt{BMS}}$=0, the processor boots through the SPI Port. For a list showing how to select different boot modes, see the Boot Memory Select pin description in the table "Booting Modes" on page 13-72.

When using any of the power-up booting modes, address 0x0004 0004 should not contain a valid instruction since it is not executed during the booting sequence. A `NOP` or `IDLE` instruction should be placed at this location.

In SPI Port Booting, the processor gets boot data from another processor's SPI port or another SPI compatible device after system powerup.

Table 6-27 on page 6-115 shows how the DMA channel 8 parameter registers are initialized at reset for EPROM booting. The count register (`CSRX`) is initialized to 0x0180 for transferring 256 words to internal memory. The SPI Control Register (`SPICTL`) is configured to 0x0A001F81 upon reset during on SPI boot. The default value sets up SPI transfers as follows:

- `SPIEN` = 1, SPI enabled
- `MS` = 0, slave device

- DF = 0, LSB first
- WL = 11, 32-bit SPI receive shift register word length
- BAUDR = 1111 (at 100 MHz, SPICLK = 763 Hz)
- DMISO = 1, MISO disabled
- RDMA = 1, SPIRX DMA enabled on channel 8

This configuration sets up the SPIRX register for 32-bit serial transfers. The SPIRX DMA channel 8 parameter registers are configured to DMA in 0x180 number of 32-bit words into internal memory normal word address space starting at 0x40000. Once the 32-bit DMA transfer completes, the data is then accessed as 3-column 48-bit instructions. The processor executes a 256 (0x100) word loader kernel upon completion of the 32-bit, 0x180 word DMA. Note that for 16-bit SPI hosts, shift two words into the 32-bit receive shift register before a DMA transfer to internal memory occurs. For 8-bit SPI hosts, shift four words into the 32-bit receive shift register before a DMA transfer to internal memory occurs.

Table 6-27. DMA Channel 8 Parameter Register Initialization for SPI Port Booting

Parameter Register	Initialization Value
IISRX	0x0004 0000
IMSRX	uninitialized (increment by 1 is automatic)
CSRX	0x0180 (256 instruction words)
GPSRX	uninitialized

SPI Port DMA Programming Examples

This section provides two programming examples written for the ADSP-21161 processor. The example shown in Listing 6-7 demonstrates how the I/O processor uses DMA to read from the SPI port receive buffer and write to the SPI port transmit buffer. The example shown in Listing 6-8 demonstrates how the I/O processor uses DMA to read from the SPI port receive buffer and write to the SPI port transmit buffer after an interrupt.

Listing 6-7. DMA-Driven SPI Loopback

```
/*_______________________________________________________________
ADSP-21161 DMA-Driven SPI Loopback Example

This example shows looped-back SPI 32-bit transfer. On this
peripheral loop-back is performed by externally connecting the
hardware MOSI and MISO pins on the processor. The transfer buffer
and receive buffer are both handled via DMA. Hardware loop-back
does not require the use of flags as device selects so the FLS
bits do not need to be used as they would in an SPI transfer
between two different SPI devices (non-loop-back.)
_____________________________________________________________*/
#include <def21161.h>
#define size 10

/* vector code for reset vector from ldf file */
.section/pm   seg_rth;
Chip_Reset:     idle; jump start; nop; nop;

/* vector code for receive interrupt vector from ldf file */
.section/pm spiri_svc;
nop; nop; jump finish; nop;
```

```
.section/dm seg_dmda;
/* transmit buffer */
.var spi_tx_buf[size] =0x11111111,0x22222222, 0x33333333,
0x44444444, 0x55555555,0x66666666, 0x77777777, 0x88888888,
0x99999999,0xaaaaaaaa;
/* receive buffer */
.var spi_rx_buf[size];

.section/pm seg_pmco;

start:
r0=spi_tx_buf;  /* configure index register for SPI transmit */
dm(IISTX)=r0;

r0=@spi_tx_buf;  /* configure count register for SPI transmit */
dm(CSTX)=r0;

r0=1;       /* configure modify register for SPI transmit */
dm(IMSTX)=r0;

r0=spi_rx_buf;   /* configure index register for SPI receive */
dm(IISRX)=r0;

r0=@spi_rx_buf;   /* configure count register for SPI receive */
dm(CSRX)=r0;

r0=1;
dm(IMSRX)=r0;    /* configure modify register for SPI receive */

ustat1 = dm(SYSCON);
bit clr ustat1 BHD;   /*  Clear Buffer Hang Disable in SYSCON */
dm(SYSCON) = ustat1;

bit set LIRPTL SPIRMSK ;   /* enable SPI RX interrupts */
bit set MODE1 IRPTEN | CBUFEN;  /* allow global interrupts and
                                   circular buffer enable */
bit set IMASK LPISUMI;       /* unmask spi interrupts */

r0=0x00000000;    /* initially clear SPI control register */
dm(SPICTL)=r0;
ustat1=dm(SPICTL);

/*  set up options for the SPI port */

bit set ustat1  SPIEN | SPRINT | SPTINT | MS | CPHASE | DF | WL32
| BAUDR5 | SGN | GM | RDMAEN | TDMAEN;

/* enable spi port, spitx and spirx interrupts, master device
spiclk toggles at beginning of first data transfer bit, MSB first
format, 32 bit word length, baud rate sign extend, get more new
data even if receive buffer is full enable transmit and receive
dma */

dm(SPICTL) = ustat1;  /* start transfer by configuring SPICTL */

wait: idle; jump wait;

finish:rti;
```

Listing 6-8. Interrupt DMA-Driven SPI Loopback Example

```
/*___________________________________________________________
ADSP-21161 Interrupt DMA-Driven SPI Loopback Example

This example shows an externally looped-back SPI 32-bit transfer.
DMA is used to write to and read from the buffers.  Loopback is
achieved by physically connecting the MOSI and MISO pins external
to the processor.
```

```
____________________________________________________________*/

#include "def21161.h"
#define size 10

/*  PM interrupt vector code */
.SECTION/pm seg_rth;
Reserved_1: rti; nop; nop; nop;
Chip_Reset: idle; jump start; nop; nop;

.SECTION/DMseg_dmda;
.var spi_tx_buf[size] =0x11111111,
                     0x22222222,
         0x33333333,
         0x44444444,
         0x55555555,
         0x66666666,
         0x77777777,
         0x88888888,
          0x99999999,
        0xaaaaaaaa;
.var spi_rx_buf[size];

.SECTION/PMseg_pmco;
.GLOBAL SPI_register_init;
.GLOBALSPI_lpbk_irq_test;

start:
ustat1 = dm(SYSCON);   /* Clear Buffer Hang Disable in SYSCON */
bit clr ustat1 BHD;
dm(SYSCON) = ustat1;
bit set mode1 CBUFEN;   /* set circular buffer enable */
SPIDMA_tx:
r0=spi_tx_buf;dm(IILB1)=r0;
```

```
r0=@spi_tx_buf;dm(CLB1)=r0;
r0=1;dm(IMLB1)=r0;
SPIDMA_rx:
r0=spi_rx_buf;dm(IILB0)=r0;
r0=@spi_rx_buf;dm(CLB0)=r0;
r0=1;dm(IMLB0)=r0;
r0=0x00000000;dm(SPICTL)=r0;   /* Initially clear SPI control
                                       reg.*/

ustat1=dm(SPICTL);
bit set ustat1
SPIEN|SPRINT|SPTINT|MS|CPHASE|DF|WL32|BAUDR5|PSSE|DCPH0|SGN|GM|R
DMAEN|TDMAEN;
bit clr ustat1
CP|FLS0|FLS1|FLS2|FLS3|SMLS|DMISO|OPD|PACKEN|SENDLW;
dm(SPICTL) = ustat1;

bit set LIRPTL SPIRMSK | SPITMSK;  /* enable SPI TX & SPI RX */
interrupts
bit set MODE1 IRPTEN;       /*  Allow global interrupts */

wait: jump start;
```

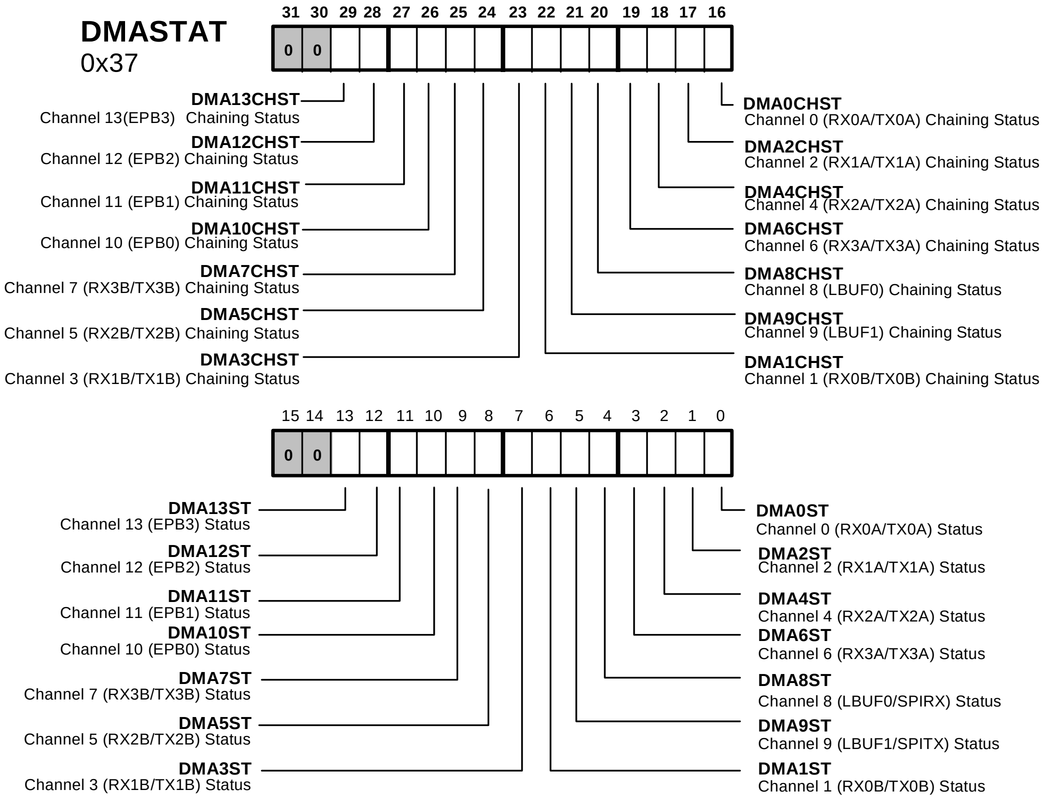
Using I/O Processor Status

The I/O processor monitors the status of data transfers on DMA channels and indicates status in the DMASTAT, IRPTL, and LIRPTL registers.

- Table A-9 on page A-27 lists all the bits in IRPTL.
- Table A-10 on page A-34 lists all the bits in LIRPTL.
- A discussion of DMASTAT appears in "DMA Channel Status Register (DMASTAT)" on page A-90.

The DMA controller of ADSP-21161 processor maintains the status information of the channels in a read only register, DMASTAT. Bits 0-13 indicate which DMA channel is active; bits 16-29 indicate the chaining status of the channels.

- Bit definitions for the DMASTAT register are defined in Table 6-28 and in Figure 6-15.
- Bit definitions for the SPISTAT register are defined in Table A-29 on page A-115.


DMASTAT
0x37
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0
DMA13CHST Channel 13(EPB3) Chaining Status
DMA12CHST Channel 12 (EPB2) Chaining Status
DMA11CHST Channel 11 (EPB1) Chaining Status
DMA10CHST Channel 10 (EPB0) Chaining Status
DMA7CHST Channel 7 (RX3B/TX3B) Chaining Status
DMA5CHST Channel 5 (RX2B/TX2B) Chaining Status
DMA3CHST Channel 3 (RX1B/TX1B) Chaining Status
DMA0CHST Channel 0 (RX0A/TX0A) Chaining Status
DMA2CHST Channel 2 (RX1A/TX1A) Chaining Status
DMA4CHST Channel 4 (RX2A/TX2A) Chaining Status
DMA6CHST Channel 6 (RX3A/TX3A) Chaining Status
DMA8CHST Channel 8 (LBUF0) Chaining Status
DMA9CHST Channel 9 (LBUF1) Chaining Status
DMA1CHST Channel 1 (RX0B/TX0B) Chaining Status
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0
DMA13ST Channel 13 (EPB3) Status
DMA12ST Channel 12 (EPB2) Status
DMA11ST Channel 11 (EPB1) Status
DMA10ST Channel 10 (EPB0) Status
DMA7ST Channel 7 (RX3B/TX3B) Status
DMA5ST Channel 5 (RX2B/TX2B) Status
DMA3ST Channel 3 (RX1B/TX1B) Status
DMA0ST Channel 0 (RX0A/TX0A) Status
DMA2ST Channel 2 (RX1A/TX1A) Status
DMA4ST Channel 4 (RX2A/TX2A) Status
DMA6ST Channel 6 (RX3A/TX3A) Status
DMA8ST Channel 8 (LBUF0/SPIRX) Status
DMA9ST Channel 9 (LBUF1/SPITX) Status
DMA1ST Channel 1 (RX0B/TX0B) Status

* Channel Active Status: 1=Active [transferring data or waiting to transfer current block, and not transferring TCB]
0= Inactive [DMA transter complete, or in TCB chain loading]

** Channel Chaining Status: 1=Chaining is *Enabled* and currently transferring TCB, or is *Pending* to transfer TCB,
0 = Chaining Disabled

Status does not change on the master ADSP-21161 processor during external port DMA until the external portion is completed (for example, the EPBx buffers are emptied).

If in chain insertion mode (DEN=0, CHEN=1), then *channel chaining status* will never go to a 1. Therefore, test *channel status* to see if it is ready so that your program can rewrite the chain pointer (CPx) register.

Figure 6-15. DMA STAT Register

Table 6-28. DMASTAT Register Definitions

Bit #	DMA Channel #	Definitions
0	0	Status[1] (RX0A or TX0A)
1	2	Status[1] (RX1A or TX1A)
2	4	Status[1] (RX2A or TX2A)
3	6	Status[1] (RX3A or TX3A)
4	8	Status[1] (LBUF0/SPIRX)
5	9	Status[1] (LBUF1/SPITX)
6	1	Status[1] (RX0B or TX0B)
7	3	Status[1] (RX1B or TX1B)
8	5	Status[1] (RX2B or TX2B)
9	7	Status[1] (RX3B or TX3B)
10	10	Status[1] (EPB0)
11	11	Status[1] (EPB1)
12	12	Status[1] (EPB2)
13	13	Status[1] (EPB3)
14 - 15		Reserved
16	0	Chaining Status[2] (RX0A or TX0A)
17	2	Chaining Status[2] (RX1A or TX1A)
18	4	Chaining Status[2] (RX2A or TX2A)
19	6	Chaining Status[2] (RX3A or TX3A)
20	8	Chaining Status[2] (LBUF0)
21	9	Chaining Status[2] (LBUF1)
22	1	Chaining Status[2] (RX0B or TX0B)
23	3	Chaining Status[2] (RX1B or TX1B)
24	5	Chaining Status[2] (RX2B or TX2B)

Table 6-28. DMASTAT Register Definitions (Cont'd)

Bit #	DMA Channel #	Definitions
25	7	Chaining Status[2] (RX3B or TX3B)
26	10	Chaining Status[2] (EPB0)
27	11	Chaining Status[2] (EPB1)
28	12	Chaining Status[2] (EPB2)
29	13	Chaining Status[2] (EPB3)
30-31		Reserved

1 Channel Active status: 1-active, 0 = inactive
2 Channel Chaining status: 1 = chaining enabled/pending, 0 = chaining disabled

The I/O processor reports on DMA in progress, DMA complete, or DMA channel not ready status as follows:

- All DMA channels can be active or inactive. If a channel is active, a DMA is in progress on that channel. The I/O processor indicates the active status by setting the channel's bit in the `DMASTAT` register.
- When an unchained (single-block) DMA process reaches completion on any DMA channel, the I/O processor generates that DMA channel's interrupt. It does this by setting the DMA channel's interrupt latch bit in the `IRPTL` or `LIRPTL` register. The DMA process is complete when the count in `CEPx`=0 (for Slave mode and Handshake modes) or when the count in `ECEPx`=0 (for External Handshake mode) or when the count in `CEPx`=0 and `ECEPx`=0 (for Master mode and Paced Master mode).
- When a DMA process in a chained DMA sequence reaches completion (the count in `Cx`=0 or `CEPx`=0) on any DMA channel, the I/O processor generates an interrupt if the `PCI` bit in the channels `CPx` register is set. The only exception is external-handshake mode.

The I/O processor also generates that DMA channel's interrupt when the last block in a chained DMA reaches completion regardless of the PCI setting.

- When a DMA channel's buffer not being used for a DMA process, the I/O processor can generate an interrupt on single word writes or reads of the buffer. This interrupt service differs slightly for each port. For more information on single-word interrupt-driven transfers, see "External Port Status" on page 6-127, "Link Port Status" on page 6-131, and "Serial Port Status" on page 6-135.

Using the DMA Channel Status Register (DMASTAT), programs can check which DMA channels are performing a DMA or chained DMA. For each channel, the I/O processor sets the channel's active status bit if DMA for that channel is enabled and a DMA sequence is in progress on that channel. The I/O processor sets the channel's chaining status bit if a chained DMA sequence is in progress or pending on that channel.

There is a one cycle latency between a change in DMA channel status and the status update in the DMASTAT register.

As an alternative to interrupt-driven DMA, programs can poll the DMASTAT register to determine when a single DMA sequence is done. To poll channel status, programs read DMASTAT. If both status bits for the channel are cleared, the DMA sequence has completed.

If chaining is enabled on a DMA channel, programs should not use polling to determine channel status. Polling could provide inaccurate information in this case because the next DMA sequence might be under way by the time the polled status is returned.

During interrupt-driven DMA, programs use the interrupt mask bits in the IMASK and LIRPTL registers to selectively mask DMA channel interrupts that the I/O processor latches into the IRPTL and LIRPTL registers.

The I/O processor only generates a DMA complete interrupt when the channel's count register decrements to zero as a result of actual DMA transfers. Writing zero to a count register does not generate the interrupt.

A channel interrupt mask in `IMASK` and `IRPTL` masks out DMA complete interrupts for a channel, but other types of interrupt masking are also available. These other types of interrupt masking include:

- By clearing a channels `PCI` bit during chained DMA, programs mask the DMA complete interrupt for a DMA processes within a chained DMA sequence.
- By masking the `LPISUM` interrupt, programs mask out the logical Oring of link port interrupt status.
- By masking the `LSRQ` interrupt, programs mask out link port service requests to link ports that do not have an assigned link buffer.

These lower levels of interrupt masking let programs limit some of the conditions that can cause DMA channel interrupts.

Each DMA channel has its own interrupt. Although the external port and link port channel access priority can rotate, the interrupt priorities of all DMA channels are fixed.

In processor systems using I/O processor interrupts, an external device may need to change the processor's interrupt mask. This task presents a challenge because the `IMASK` register is not memory-mapped and is not directly accessible to external devices through the external port. To read or write `IMASK` through the external port, programs can set up an interrupt vector routine to handle this task. The `VIRPT` vector interrupt register may be used for this task.

The I/O processor can also generate non-DMA single-word interrupts for I/O port operations that do not use DMA. In this case, the I/O processor generates a DMA interrupt when data becomes available at the receive buffer or when the transmit buffer does not have new data to transmit. Generating DMA interrupts in this fashion lets programs implement interrupt-driven I/O under control of the processor core. Care is needed because multiple interrupts can occur if several I/O ports transmit or receive data in the same cycle.

External Port Status

The I/O processor monitors the status of data transfers on the external port. DMA channel status for the external port is described in "Using I/O Processor Status" on page 6-121. This section describes external port specific status features, such as buffer status, buffer control, and single-word interrupt-driven transfers.

Bits in the SYSTAT, SYSCON and DMACx registers indicate and control the status of external port buffers.

- Table A-21 on page A-69 lists all the bits in SYSTAT.
- Table A-18 on page A-60 lists all the bits in SYSCON.
- Table A-24 on page A-80 and Figure 6-16 lists all the bits in the DMACx register.
- For a description of the IOP registers, see the Registers appendix of this manual.

The following bits influence external port buffer status:

- **Host Packing Status.** SYSTAT bits 24-22 (HPS). These bits indicate the host's packing status.
- **External Port Packing Status.** DMACx Bits 23-21(PS). These bits indicate the corresponding FIFO buffer's packing status. Table 6-29 shows the available bit setting.

- **Single-Word Interrupt Enable.** DMACx Bit 12 (INTIO). This bit enables (if set, =1) or disables (if cleared, =0) single-word, non-DMA, interrupt-driven transfers for the corresponding external port FIFO buffer (EPBx). To avoid spurious interrupts, programs must not change a buffer's INTIO setting while the buffer is enabled.
- **Flush DMA Buffers and Status.** DMACx Bit 14 (FLSH). This bit flushes (when set, =1) settings for the corresponding external port FIFO buffer (EPBx).
- **External Port FIFO Buffer Status.** DMACx bit 17-16 (FS). These bits indicate the corresponding external port FIFO buffer's status. Table 6-30 shows the available setting.

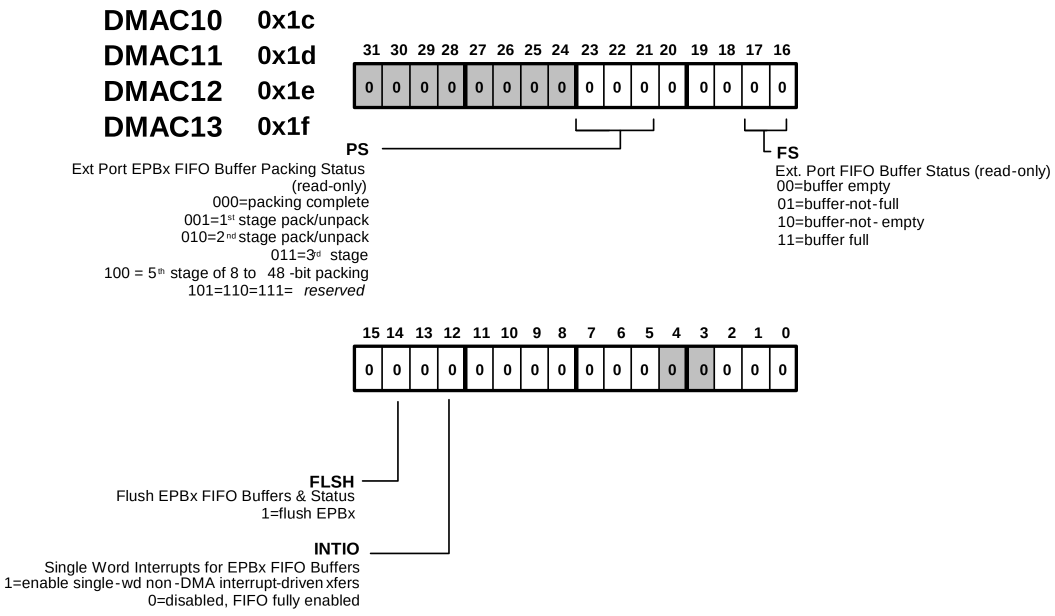

DMAC10 0x1c
DMAC11 0x1d
DMAC12 0x1e
DMAC13 0x1f
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
PS
Ext Port EPBx FIFO Buffer Packing Status (read-only)
000=packing complete
001=1st stage pack/unpack
010=2nd stage pack/unpack
011=3rd stage
100 = 5th stage of 8 to 48 -bit packing
101=110=111= reserved
FS
Ext. Port FIFO Buffer Status (read-only)
00=buffer empty
01=buffer-not-full
10=buffer-not- empty
11=buffer full
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
FLSH
Flush EPBx FIFO Buffers & Status
1=flush EPBx
INTIO
Single Word Interrupts for EPBx FIFO Buffers
1=enable single-wd non-DMA interrupt-driven xfers
0=disabled, FIFO fully enabled

Figure 6-16. DMAC Register–Status Bits Only

The HPS bits in the SYSTAT and PS bits in the DMACx registers indicate an external buffer's packing status. These bits are read-only, and the processor clears these bits when DEN is cleared (changes from 1 to 0).

Table 6-29. Processor (PS) and Host (HPS) Packing Status

PS or HPS	Packing Status
000	packing complete (6th stage of 8- to 48-bit packing, 4th stage of 8- to 32-bit packing, etc.)
001	1st stage
010	2nd stage
011	3rd stage
100	fifth stage of 8/48

The FS bits in the DMACx registers indicate an external buffer's FIFO status. These bits are read-only. The processor clears these bits when DEN is cleared (changes from 1 to 0).

Table 6-30. External Port Buffer FIFO Status

FS	FIFO Buffer Status
00	buffer empty
01	buffer-not-full
10	buffer-not-empty
11	buffer full

For transmit (TRAN=1), buffer-not-full means that the buffer has space for one normal word, and buffer-not-empty means that the buffer has space for two-or-more normal words. For receive (TRAN=0), buffer-not-full means that the buffer contains one normal word, and buffer-not-empty means that the buffer contains two or more normal words. Any type of full status (01, 10, or 11) in receive mode indicates that new (unread) data is in the buffer.

When a program sets (=1) the FLSH bit, the processor flushes the settings for the corresponding external port FIFO buffer (EPBx). Flushing these settings does the following: clears (=0) the FS and PS status bits, clears (=0) the FIFO buffer and DMA request counter, clears any partially packed words. There is a two-cycle effect latency in completing the flush operation. Programs must not set a buffer's FLSH during the same write that enables the buffer. Also, programs must not set a buffer's FLSH bit while the DMA channel is active. Programs should determine the channel's active status by reading the corresponding bit in the DMASTAT register.

Status does not change on the master processor during external port DMA until the external portion is completed (for example, the EPBx buffers are emptied). If in chain insertion mode (DEN=0, CHEN=1), then channel chaining status never goes to 1. Programs should test channel status to see if it is ready before re-writing the chain pointer (CPx).

The INTIO bit in the DMACx registers support single-word interrupt-driven transfers for each corresponding external port buffer. These non-DMA transfers are available under the following conditions:

The external port DMA channel's DEN bit is cleared (DMA disabled).

- The external port DMA channel's INTIO bit is set enabling interrupt-driven I/O.
- The external port DMA channel's buffer is not empty on an external read or not full on an external write.

Under these conditions, the I/O processor generates that DMA channel's interrupt on the single word transfer to or from that channel's external port buffer.

Link Port Status

The I/O processor monitors the status of data transfers on the link ports. DMA channel status for the link ports is described in "Using I/O Processor Status" on page 6-121. This section describes link ports specific status features, such as buffer status, buffer control, and single-word interrupt-driven transfers.

The LRSQ (Link Service Request) register allows a disabled link port to respond to a link port transmit or receive request from another processor. Bits in the LSRQ registers indicate and control status of link port buffers.The following bits influence link port buffer status:

- **Link Port x Transmit Mask.** LSRQ Bit 4 and 6 (LxTM). These bits mask (if set, =1) or unmask (if cleared, =0) the L0TRQ through L1TRQ status bits.
- **Link Port x Receive Mask.** LSRQ Bit 5 and 7 (LxRM). These bits mask (if set, =1) or unmask (if cleared, =0) the L0RRQ and L1RRQ status bits.
- **Link Port x Transmit Request Status (read-only).** LSRQ Bit 20 and 22 (LxTRQ). If set (=1), these bits indicate that the corresponding link port (0 or 1) is disabled, but has a request to transmit data.
- **Link Port x Receive Request Status (read-only).** LSRQ Bit 21and 23 (LxRRQ). If set (=1), these bits indicate that the corresponding link port (0 or 1) is disabled, but has a request to receive data.

The Link Port Status Register (LSRQ) is shown in Figure 6-16. The status bits in the Link Port Control Register (LCTL) are shown in Figure 6-17 on page 6-132.

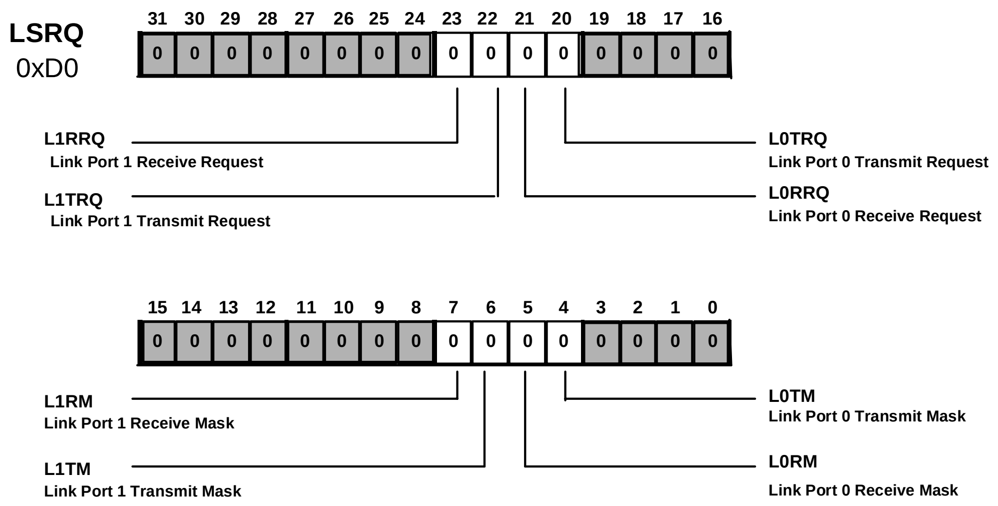


Figure 6-17. LSRQ Register

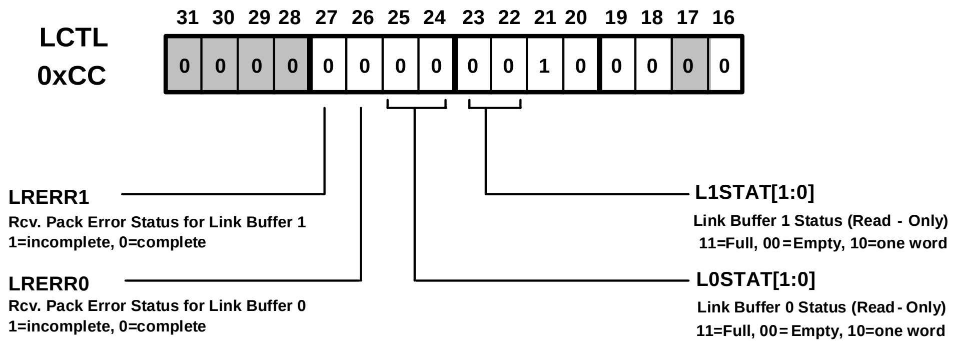

LCTL
0xCC
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
LRERR1
Rcv. Pack Error Status for Link Buffer 1
1=incomplete, 0=complete
LRERR0
Rcv. Pack Error Status for Link Buffer 0
1=incomplete, 0=complete
L1STAT[1:0]
Link Buffer 1 Status (Read - Only)
11=Full, 00=Empty, 10=one word
L0STAT[1:0]
Link Buffer 0 Status (Read - Only)
11=Full, 00=Empty, 10=one word

Figure 6-18. LCTL Register – Status Bits

The LRERRx bits in the LCTL register indicate a link port buffer's receive packing status. When the buffer is ready to receive and pack a new word, the processor clears (=0) LRERRx. If this bit remains set (=1) after the buffer receives a word, a link transfer error (for example, a clock glitch) has occurred. These bits are read-only, and the processor clears these bits when LxEN is cleared (changes from 1 to 0). Table 6-31 shows the available settings.

Table 6-31. Link Port Buffer Receive Packing Status

LRERRx	Receive Packing Status
0	pack complete (reset value)
1	pack not complete

The LxSTATx bits in the LCTL register indicate a link buffer's FIFO status. When transmitting, these bits indicate when the buffer has space for more data. When receiving, these status bits indicate when the buffer contains new (unread) data. These bits are read-only. The processor clears these bits when LxEN is cleared (changes from 1 to 0) and empties the buffer. Table 6-32 shows the available settings.

Table 6-32. Link Port Buffer FIFO Status

LxSTATx	FIFO Buffer Status
00	buffer empty
01	reserved
10	one word
11	buffer full

The LCTL register lets programs assign link buffers to link ports. Bits LAB0 and LAB1 in the LCTL register assign link buffers to link ports. Because this mapping allows link ports to be unassigned (no buffer), the I/O processor has an interrupt (LSRQI) to notify programs that an external device has made a read or write request on a disabled link port.

When an LSRQI interrupt is latched into the IRPTL register, programs use the transmit (LxTRQ) and receive (LxRRQ) request bits in LSRQ register to determine which port has a request. The LSRQ register's bits indicate the following:

- For a transmit request (LxTRQ=1), the LSRQI interrupt indicates that the link port (0 or 1) is disabled, but another processor has requested more data by setting the link port's acknowledge (LxACK=1).
- For a receive request (LxRRQ=1), the LSRQI interrupt indicates that the link port is disabled, but another processor has requested to send data by setting the link port's clock (LxCLK=1).

To control sources of link port service requests, the I/O processor lets programs mask these service requests. The LSRQ register provides mask bits for transmit (LxTM) and receive (LxRM) link service requests.

The LxEN bits in the LCTL register support single-word interrupt-driven transfers for each corresponding link port buffer. These non-DMA transfers are available under the following conditions:

- The link port DMA channel's LxDEN bit is cleared (DMA disabled).
- The link port DMA channel's LxEN bit is set enabling the link buffer.
- The link port DMA channel's buffer is not empty on receive or not full on transmit.

Under these conditions, the I/O processor generates that DMA channel's interrupt on the single word transfer to or from that channel's link port buffer.

Serial Port Status

The I/O processor monitors the status of data transfers on the serial ports. DMA channel status for the serial ports is described in "Using I/O Processor Status" on page 6-121. This section describes serial ports specific status features, such as buffer status, transmit buffer underflow, receive buffer overflow, and single-word interrupt-driven transfers.

Bits in the SPCTLx registers indicate and control status of serial port buffers. For more information, see "SPORT Serial Control Registers (SPCTLx)" on page A-100.

The following bits influence serial port buffer status:

- **DXA Error Status (sticky, read-only).** SPCTLx Bit 29 (DERR_A). This bit indicates (if set, =1 and DDIR =1) whether the serial transmit operation has underflowed or (if cleared, =0 and DDIR =0)the serial receive has overflowed on the A path.
- **DXS_A Data Buffer Status (read-only).** SPCTLx Bits 31-30 (DXS_A). These bits indicate the status of the serial port's DXA data buffer. See Table 6-33 for available bit settings.
- **DXB Error Status (sticky, read-only).** SPCTLx Bit 26 (DERR_B). This bit indicates (if set, =1 and DDIR =1) whether the serial transmit operation has underflowed or (if cleared, =0 and DDIR =0)the serial receive has overflowed on the B path.
- **DXS_B Data Buffer Status (read-only).** SPCTLx Bits 28-27 (DXS_B). These bits indicate the status of the serial port's DXB data buffer. See Table 6-33 for available bit settings.

The DXS_A and DXS_B bits in the SPCTLx registers indicate a serial port transmit or receive buffer's FIFO status. Status bits are read-only. Disabling the serial port (setting SPEN=0), clears the status bits and empties the buffer. The bits may change state if the data is read or written by the processor core while the serial port is disabled. Table 6-33 shows the available settings.

Table 6-33. Serial Port Transmit and Receive Buffer FIFO Status

DXS_A or DXS_B	FIFO Buffer Status
00	buffer empty
01	reserved
10	partially full
11	buffer full

The DERR_A and DERR_B bits in the SPCTLx registers indicate a serial port transmit underflow or receive overflow to the buffer's FIFO. Status bits are read-only. Disabling the serial port (setting SPEN=0), clears the status bits and empties the buffer. These overflow and underflow bits are sticky; once set, they remain set regardless of buffer status until the serial port is disabled.

The SPEN bit in the SPCTLx register support single-word interrupt-driven transfers for each corresponding serial port transmit or receive buffer. These non-DMA transfers are available under the following conditions:

The serial port DMA channel's SDEN bit is cleared (DMA disabled).

- The serial port DMA channel's SPEN bit is set (enabling the serial port transmit or receive buffer).
- The serial port DMA channel's buffer is not empty on receive or not full on transmit.

Under these conditions, the I/O processor generates that DMA channel's interrupt on the single word transfer to or from that channel's serial port buffer.

SPI Port Status

The I/O processor monitors the status of data transfers on the SPI port. DMA channel status for the SPI port is described in "Using I/O Processor Status" on page 6-121. This section describes SPI port specific status features, such as buffer status, transmit or receive buffer errors, and transfer completion test.

Bits in the SPISTAT register indicate and control status of SPI port buffers, SPIRX and SPITX. Table A-29 on page A-115 and Figure 6-19 lists all the bits in SPISTAT. The following bits influence SPI port buffer status:

- **SPI Transmit Transfer Completion.** SPISTAT Bit 0 (SPIF). This bit is set (=1) when the SPI transfer is complete. For example, the following condition is met: the transmit data buffer is empty and the last data has been transmitted out of the transmit shift register. The bit is cleared (=0) when the transfer is active.

- **Transmit Error (sticky, read-only).** SPISTAT Bit 2 (TXE). This bit indicates an error in the transmission. This bit is set (=1) when the transmit data buffer is empty and the last data has been transmitted out of the transmit shift register. If you are not servicing the interrupt quickly enough and not updating the contents of SPITX so that it is available to be transferred to the transmit shift register when required, this bit is set.

- **Transmit Data Buffer Status (read-only).** SPISTAT Bit 3-4 (TXS). These bits indicate the status of the SPI port transmit buffer (SPITX). If TXS =00, the buffer is empty. See Table 6-34 for available TXS bit settings.

- **Receive Error (sticky, read-only).** SPISTAT Bit 5 (RBSY). This bit indicates an error in the receive operation. This bit is set (=1) when the SPITX data buffer is full and the last data has been received into the receive shift register. If you are not servicing the interrupt quickly enough and not transferring the contents of SPIRX, this bit is set.

- **Receive Data Buffer Status (read-only).** SPISTAT Bits 6-7 (RXS). These bits indicate the status of the SPI port receive buffer (SPIRX). If RXS =00, the buffer is empty. See Table 6-34 for available RXS bit settings.

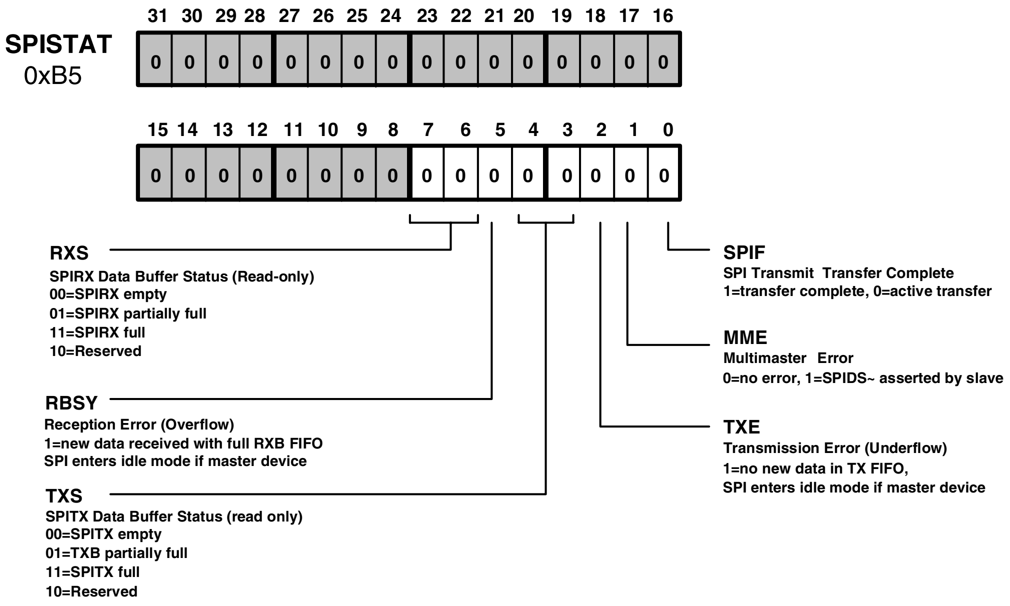

SPISTAT
0xB5
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RXS
SPIRX Data Buffer Status (Read-only)
00=SPIRX empty
01=SPIRX partially full
11=SPIRX full
10=Reserved
RBSY
Reception Error (Overflow)
1=new data received with full RXB FIFO
SPI enters idle mode if master device
TXS
SPITX Data Buffer Status (read only)
00=SPITX empty
01=TXB partially full
11=SPITX full
10=Reserved
SPIF
SPI Transmit Transfer Complete
1=transfer complete, 0=active transfer
MME
Multimaster Error
0=no error, 1=SPIDS~ asserted by slave
TXE
Transmission Error (Underflow)
1=no new data in TX FIFO,
SPI enters idle mode if master device

Figure 6-19. SPISTAT Register

The TXS and RXS bits in the SPISTAT registers indicate a SPI port transmit (SPITX) or receive (SPIRX) buffer's FIFO status. Disabling the SPI port (setting SPIEN=0), clears the status bits and empties the buffer. TXS and RXS may change state if the data is read or written by the processor core while the SPI port is disabled. Table 6-34 shows the available settings.

Table 6-34. SPI Port Transmit and Receive Buffer FIFO Status

TXS or RXS	FIFO Buffer Status
00	buffer empty
01	partially full
10	reserved
11	buffer full

The TXE and RBSY bits in the SPISTAT registers indicate a SPI port transmit underflow or receive overflow to the buffer's FIFO. Status bits are read-only. Disabling the SPI port (setting SPIEN=0), clears the status bits and empties the buffer. These overflow and underflow bits are sticky; once set, they remain set regardless of buffer status until the SPI port is disabled.

Under these conditions, the I/O processor generates that DMA channel's transfer request over the IOD bus on the single word transfer to the SPITX data buffer or from the SPIRX data buffers.

Optimizing DMA Throughput

This section discusses overall DMA throughput when several DMA channels are trying to access internal or external memory at the same time. Table 6-35 summarizes the advantages of different system configurations.

Internal Memory DMA

The DMA channels arbitrate for access to the processor's internal memory. The DMA controller determines, on a cycle-by-cycle basis, which channel is allowed access to the internal I/O bus and consequently which channel can read or write to internal memory. The priority order of the DMA channels appears in Table 6-1 on page 6-3.

Each DMA transfer takes one clock cycle even when different DMA channels are being allowed access on sequential cycles; for example, there is no overall throughput loss in switching between channels. Thus, two link port DMA channels, each transferring one byte per cycle, would have one half the I/O transfer rate as one external port DMA channel transferring data to internal memory on every cycle. Any combination of link ports, serial ports, and external port transfers has the same maximum transfer rate.

External Memory DMA

When the DMA transfer is between processor internal memory and external memory, the external memory may have one or more wait states. External memory wait states, however, do not reduce the overall internal DMA transfer rate if other channels have data available to transfer. In other words, the processor's internal I/O data bus cannot be held up by an incomplete external transfer.

Table 6-35. Configurations for Processor – Processor DMA

Processor Config. (Data Source)	Processor Config. (Data Destination)	C/T[1]	Advantages, Disadvantages
Bus Master DMA Master Mode (MASTER= 1) TRAN=1, EIEPx = address of EPBx buffer in destination, EMEPx= 0	Bus Slave DMA Slave Mode (MASTER= 0), TRAN= 0	1	Advantage: Destination automatically generates interrupt upon completion. Disadvantage: DMA must be programmed on both source and destination.
Bus Slave DMA Slave Mode (MASTER= 0), TRAN= 1	Bus Master DMA Master Mode (MASTER= 1), TRAN=0, EIEPx = address of EPBx buffer in source, EMEPx=0	3[2]	Advantage: Source automatically generates interrupt upon completion. Disadvantages: Slower throughput. DMA must be programmed on both source and destination.

1 C/T is throughput in cycles/transfer.
2 Maximum burst read throughput: 3-2-2-2

Figure 6-20 shows an example DMA hardware interface. The following should be noted in this figure.

- Because $\overline{\texttt{DMARx}}$ and $\overline{\texttt{DMAGx}}$ are tied together, only one of the processors may have DMA enabled at a time.
- $\overline{\texttt{DMAGx}}$ is only driven by the processor bus master.
- The DMA Write Grant signal can be the combination of $\overline{\texttt{WR}}$ and $\overline{\texttt{MS3-0}}$ instead of $\overline{\texttt{DMAG2}}$ if paced master mode is used.
- The DMA Read Grant signal can be the combination of $\overline{\texttt{RD}}$ and $\overline{\texttt{MS3-0}}$ instead of $\overline{\texttt{DMAG1}}$ if paced master mode is used.
- DMA transfers may be to either processor or to external memory (in external handshake mode).

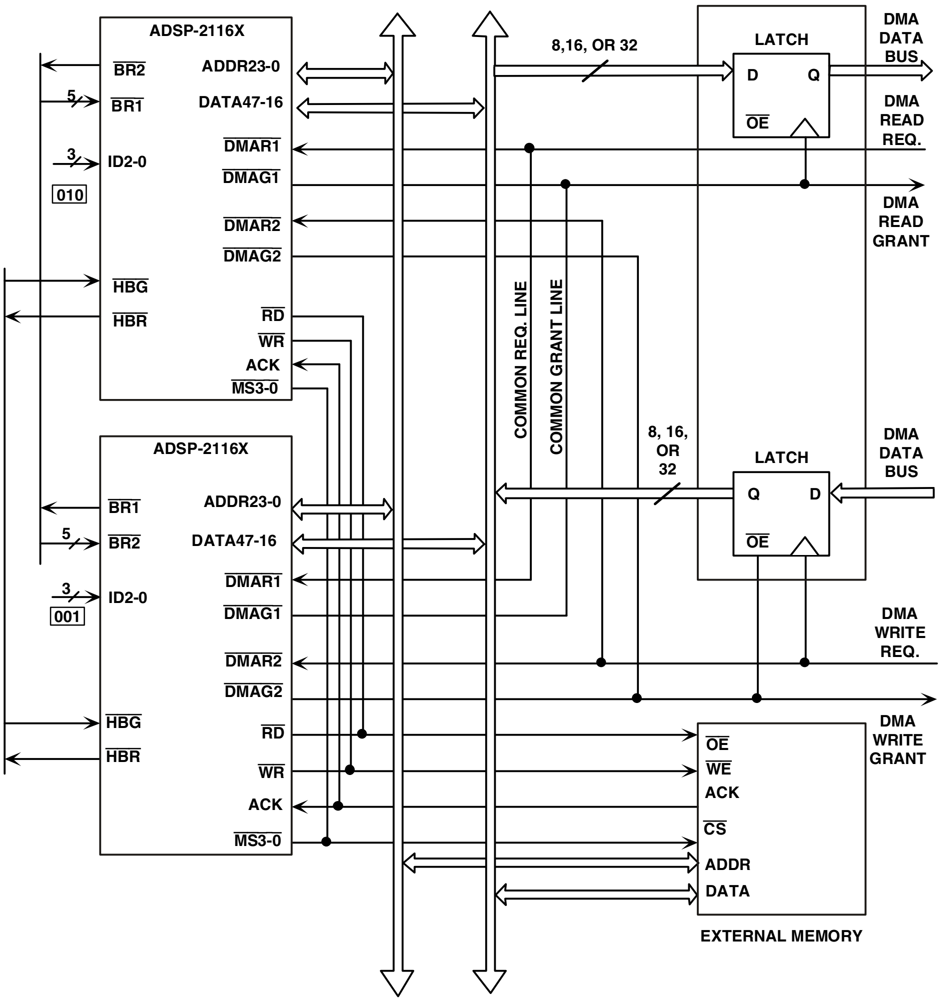


Figure 6-20. Example DMA Hardware Interface

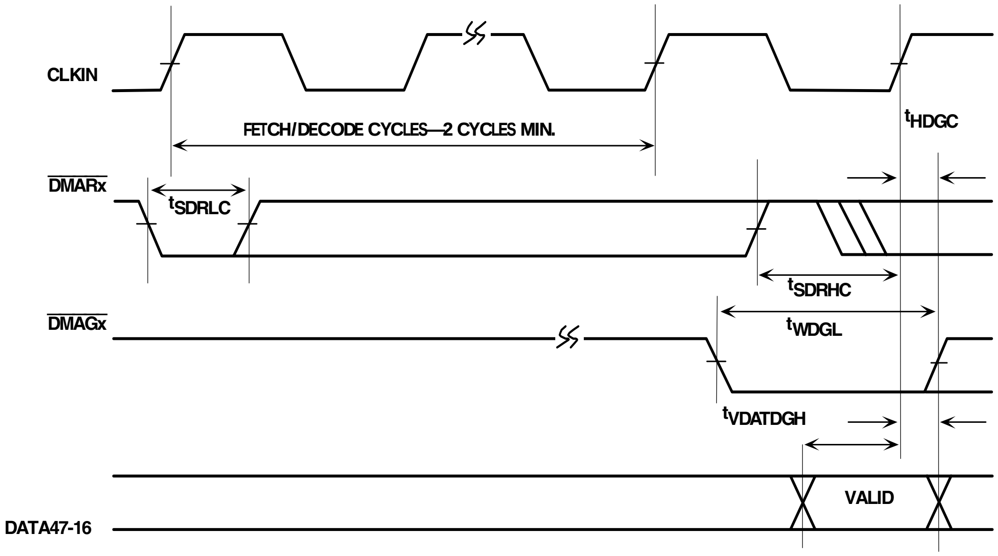


Figure 6-21. $\overline{\text{DMAR}}$ and $\overline{\text{DMAG}}$ Timing

Figure 6-21 shows $\overline{\text{DMAR}}$ and $\overline{\text{DMAG}}$ timing. The following should be noted in this figure.

- $\overline{\text{DMARx}}$ setup times relate to the use of the signal in that cycle by the processor. DMA requests may be asserted asynchronously to CLKIN.
- $\overline{\text{DMAGx}}$ drives DATA47-16 if the processor is receiving. $\overline{\text{DMAGx}}$ latches DATA47-16 if the processor is transmitting.

When data is to be transferred from internal to external memory, the internal memory data is first placed in the external port's EPBx buffer by the DMA controller; the external memory access begins independently once the data is detected in the EPBx buffer. Likewise, for external-to-internal DMA, the internal DMA request is not be made until the external memory data is in the EPBx buffer. In both cases, the external DMA address generator—the EIEPx and EMEPx parameter registers—maintains the external address until the data transfer is completed. The internal and external address generators of a DMA channel are decoupled and operate independently.

When EXTERN mode DMA transfers occur between an external device and external memory, no internal resources of the processor are utilized and internal DMA throughput is not affected.

System-Level Considerations

Slave mode DMA is useful in systems with a host processor because it allows the host to access any processor internal memory location indirectly through DMA while limiting the address space the host must recognize—only the address space of the processor's I/O processor registers. Slave mode DMA is also useful for processor-to-processor DMA transfers.

Slave mode DMA has one drawback when interfacing to a slow host—the fact that the external bus is held up during the transfer (whether initiated by the processor or the host) and no other transactions can proceed. To overcome this, the handshake DMA mode may be used.

In handshake mode, the host does not have to master the bus in order to make a DMA request, nor does the processor (in master mode) have to wait on the bus for the transfer to complete. Instead, the host asserts the $\overline{\text{DMARx}}$ pin. When the processor is ready to make the transfer, it can complete it in one bus cycle. For more information, see "Handshake Mode" on page 6-57.

7 EXTERNAL PORT

The ADSP-21161 processor's external port extends its address and data buses off-chip. Using these buses and external control lines, systems can interface the processor with external memory, 8-, 16- or 32-bit host processors, and other DSPs. Because many of the external port operations relate to external memory accessing or I/O processing, this chapter refers to the memory and I/O processor chapters ("Memory" on page 5-1 and "I/O Processor" on page 6-1) frequently.

This chapter describes connection and timing issues for the external port. The main sections of this chapter describe the interfaces that are available through the external port. These interfaces include:

- "External Memory Interface" on page 7-3
- "Host Processor Interface" on page 7-42
- "Multiprocessor (MP) Interface" on page 7-87

Data alignment through the external port is identical for these interfaces. Figure 7-1 shows the external port's data alignment.

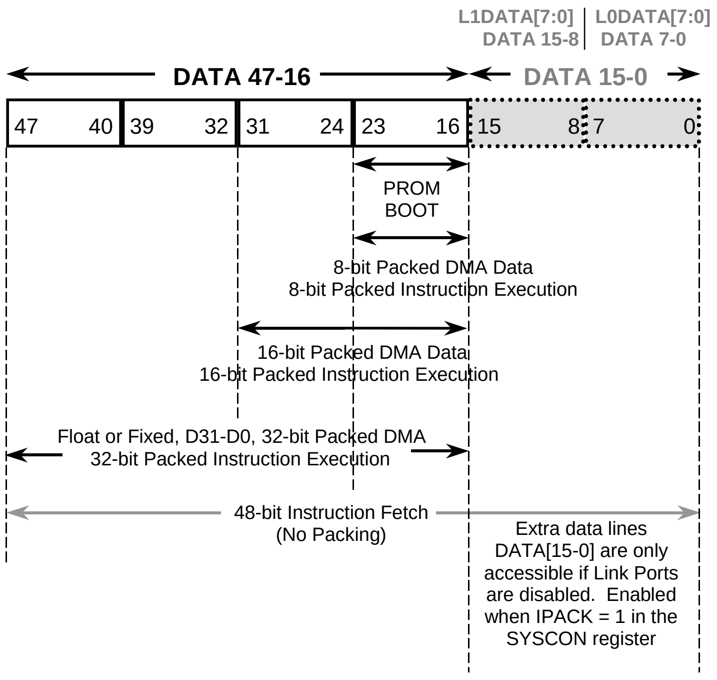

L1DATA[7:0]
DATA 15-8
L0DATA[7:0]
DATA 7-0
DATA 47-16
DATA 15-0
47
40
39
32
31
24
23
16
15
8
7
0
PROM
BOOT
8-bit Packed DMA Data
8-bit Packed Instruction Execution
16-bit Packed DMA Data
16-bit Packed Instruction Execution
Float or Fixed, D31-D0, 32-bit Packed DMA
32-bit Packed Instruction Execution
48-bit Instruction Fetch
(No Packing)
Extra data lines DATA[15-0] are only accessible if Link Ports are disabled. Enabled when IPACK = 1 in the SYSCON register

Figure 7-1. ADSP-21161 External Data Alignment Options

Setting External Port Modes

This section describes the various ways to use the external port for data transfer. The SYSCON, WAIT, and DMACx registers control the external port operating mode. Table A-18 on page A-60 lists all the bits in SYSCON, Table A-17 on page A-68 lists all the bits in WAIT, and Table A-24 on page A-80 lists all the bits in DMACx.

- For information about setting up memory access modes (synchronous versus asynchronous interface), see "Setting Data Access Modes" on page 5-32.
- For information on setting DMA through the external port, see "External Port DMA" on page 6-29.
- For information on using external port interrupts, see "Using I/O Processor Status" on page 6-121.

There is a 3:1 bus conflict resolution ratio at the external port interface (three internal buses to one external bus) in addition to the 2:1 or greater clock ratio between the processor's internal clock and the external system clock. Systems that fetch instructions or data through the external port must tolerate at least one cycle—and possibly many additional cycles—of latency for non-SDRAM accesses. SDRAM accesses operate at the core clock rate.

External Memory Interface

In addition to its on-chip SRAM, the processor provides addressing of up to 64 megawords SRAM or SBSRAM or 254 megawords of off-chip SDRAM memory through its external port. This external address space includes multiprocessor memory space—the IOP register space of all other ADSP-21161s connected in a multiprocessor system—as well as external memory space—the region for standard addressing of off-chip memory.

Figure 7-2 shows how the buses and control signals extend off-chip, connecting to external memory. Table 7-1 defines the processor pins used for interfacing to external memory. The processor's memory control signals permit direct connection to fast static RAM devices. Memory mapped peripherals and slower memories can also connect to the processor using a user-defined combination of programmable waitstates and hardware acknowledge signals.

External memory can hold instructions and data. Packed instructions can be executed directly from 32-bit, 16-bit, or 8-bit wide external memories using 32- to 48-bit, 16- to 48-bit or 8- to 48-bit execution packing modes supported by the external port and program sequencer. The external port can also be configured to have a 48-bit wide external data bus for 48-bit non-packed execution of instructions when link ports are not used. The link port data lines are multiplexed with the data lines D0 to D15 and are enabled through control bits in the memory mapped control register SYSCON. Data packing of 32- to 48-bit, 16- to 48-bit, 8- to 48-bit, 32- to 32/64-bit, 16- to 32/64-bit or 8- to 32/64-bit is supported for DMA transfers directly from 32-bit, 16-bit, or 8-bit wide external memories to and from 32-, 48-, or 64-bit internal memory. Figure 7-1 shows how the processor transfers different data word sizes over the external port.

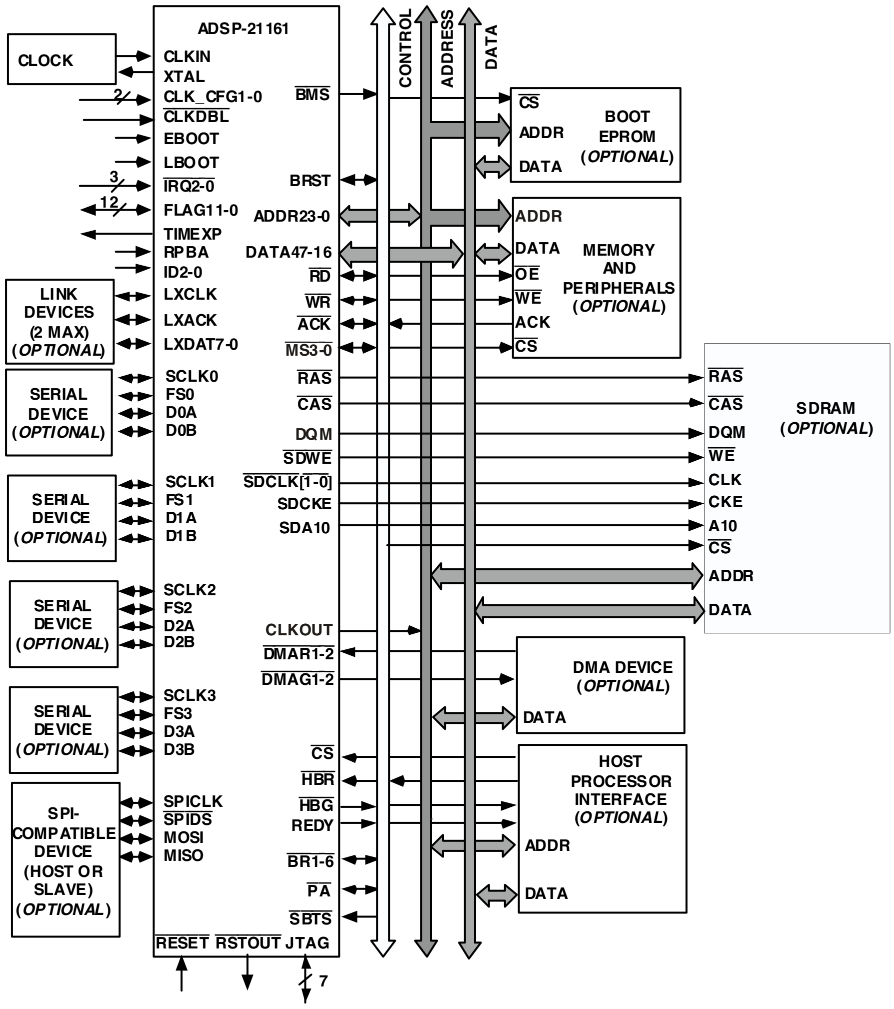


Figure 7-2. ADSP-21161 Processor System

Table 7-1. External Memory Interface Signals

Pin	Type	Function
ACK	I/O/S	**Memory Acknowledge**. External devices can deassert ACK (low) to add wait states to an external memory access. ACK is used by I/O devices, memory controllers, or other peripherals to hold off completion of an external memory access. The ADSP-21161 processor deasserts ACK as an output to add wait states to a synchronous access of its IOP registers. ACK has a 20 kΩ internal pull-up resistor that is enabled during reset or on processors with ID2-0=00x.
ADDR23-0	I/O/T	**External Bus Address**. The ADSP-21161 processor outputs addresses for external memory and peripherals on these pins. In a multiprocessor system the bus master outputs addresses for read/writes of the IOP registers of other ADSP-21161 processors while all other internal memory resources can be accessed indirectly via DMA control (that is, accessing IOP DMA parameter registers). The ADSP-21161 processor inputs addresses when a host processor or multiprocessing bus master is reading or writing its IOP registers. A keeper latch on the processor's ADDR23-0 pins maintains the input at the level it was last driven. This latch is only enabled on the processors with ID2-0=00x.
BRST	I/O/T	**Sequential Burst Access.** BRST is asserted by ADSP-21161 processor to indicate that data associated with consecutive addresses is being read or written. A slave device samples the initial address and increments an internal address counter after each transfer. The incremented address is not pipelined on the bus. A master ADSP-21161 processor in a multiprocessor environment can read slave external port buffers (EPBx) using the burst protocol. BRST is asserted after the initial access of a burst transfer. It is asserted for every cycle after that, except for the last data request cycle (denoted by $\overline{\text{RD}}$ or $\overline{\text{WR}}$ asserted and BRST negated). A keeper latch on the processor's BRST pin maintains the input at the level it was last driven. This latch is only enabled on processors with ID2-0=00x.
I (Input), S (Synchronous), o/d (Open Drain), O (Output), A (Asynchronous), a/d (Active Drive), T (Three-state, when $\overline{\text{SBTS}}$ or $\overline{\text{HBR}}$ is asserted, or when the processor is a bus slave)		

Table 7-1. External Memory Interface Signals (Cont'd)

Pin	Type	Function
CLKIN	I	**Local Clock In.** Used in conjunction with XTAL. CLKIN is the ADSP-21161 processor clock input. It configures the ADSP-21161 processor to use either its internal clock generator or an external clock source. Connecting the necessary components to CLKIN and XTAL enables the internal clock generator. Connecting the external clock to CLKIN while leaving XTAL unconnected configures the ADSP-21161 processor to use the external clock source such as an external clock oscillator.The ADSP-21161 processor external port cycles at the frequency of CLKIN. The instruction cycle rate is a multiple of the CLKIN frequency; it is programmable at power-up via the CLK_CFG1-0 pins. CLKIN may not be halted, changed, or operated below the specified frequency.
CLKOUT	O/T	**Local Clock Out.** CLKOUT is 1x or 2x and is driven at either 1x or 2x the frequency of CLKIN frequency by the current bus master. The frequency is determined by the $\overline{\text{CLKDBL}}$ pin. This output is three-stated when the ADSP-21161 processor is not the bus master. A keeper latch on the processor's CLKOUT pin maintains the output at the level it was last driven. This latch is only enabled on processors with ID2-0=00x. If $\overline{\text{CLKDBL}}$ enabled, CLKOUT = 2xCLKIN If $\overline{\text{CLKDBL}}$ disabled, CLKOUT = 1xCLKIN Note: CLKOUT is only controlled by the $\overline{\text{CLKDBL}}$ pin and operates at either 1xCLKIN or 2xCLKIN. Do not use CLKOUT in multiprocessing systems. Use CLKIN instead.
I (Input), S (Synchronous), o/d (Open Drain), O (Output), A (Asynchronous), a/d (Active Drive), T (Three-state, when $\overline{\text{SBTS}}$ or $\overline{\text{HBR}}$ is asserted, or when the processor is a bus slave)		

Table 7-1. External Memory Interface Signals (Cont'd)

Pin	Type	Function
DATA47-16	I/O/T	**External Bus Data**. The ADSP-21161 processor inputs and outputs data and instructions on these pins. Pull-up resistors on unused data pins are not necessary. A keeper latch on the processor's DATA47-16 pins maintains the input at the level it was last driven. This latch is only enabled on the processors with ID2-0=00x. **Note:** DATA[15:8] pins (multiplexed with L1DATA[7:0]) can also be used to extend the data bus if the link ports are disabled and not used. In addition, DATA[7:0] pins (multiplexed with L0DATA[7:0]) can also be used to extend the data bus if the link ports are not used. This allows execution of 48-bit instructions from external SBSRAM (system clock speed-external port), SRAM (system clock speed-external port) and SDRAM (core clock or one-half the core clock speed). The IPACKx Instruction Packing Mode Bits in SYSCON should be set correctly (IPACK1-0 = 0x1) to enable this full instruction Width/No-packing Mode of operation.
LxDAT7-0 [DAT15-0]	I/O [I/O/T]	**Link Port Data** (Link Ports 0-1). Each LxDAT pin has a 50 kΩ internal pull-down resistor that is enabled or disabled by the LxPDRDE bit of the LCTL register. **Note:** L1DATA[7:0] are multiplexed with the DATA[15:8] pins L0DATA[7:0] are multiplexed with the DATA[7:0] pins. If link ports are disabled and are not be used, then these pins can be used as additional data lines for executing instructions at up to the full clock rate from external memory.
$\overline{\text{MS3-0}}$	I/O/T	**Memory Select Lines**. These outputs are asserted (low) as chip selects for the corresponding banks of external memory. Memory bank sizes are fixed to 16 Mwords for non-SDRAM and 64 Mwords for SDRAM. The $\overline{\text{MS3-0}}$ outputs are decoded memory address lines. In asynchronous access mode, the $\overline{\text{MS3-0}}$ outputs transition with the other address outputs. In synchronous access modes, the $\overline{\text{MS3-0}}$ outputs assert with the other address lines; however, they de-assert after the first CLKIN cycle in which ACK is sampled asserted. In a multiprocessor systems, the MSx signals are tracked by slave SHARCs.
I (Input), S (Synchronous), o/d (Open Drain), O (Output), A (Asynchronous), a/d (Active Drive), T (Three-state, when $\overline{\text{SBTS}}$ or $\overline{\text{HBR}}$ is asserted, or when the processor is a bus slave)		

Table 7-1. External Memory Interface Signals (Cont'd)

Pin	Type	Function
$\overline{\text{RD}}$	I/O/T	**Memory Read Strobe.** $\overline{\text{RD}}$ is asserted whenever ADSP-21161 processor reads a word from external memory or from the IOP registers of other ADSP-21161 processors. External devices, including other ADSP-21161 processors, must assert $\overline{\text{RD}}$ for reading from a word of the ADSP-21161 processor IOP register memory. In a multiprocessing system, $\overline{\text{RD}}$ is driven by the bus master. $\overline{\text{RD}}$ has a 20 kΩ internal pull-up resistor that is enabled for processors with ID2-0=00x.
$\overline{\text{WR}}$	I/O/T	Memory Write Low Strobe. $\overline{\text{WR}}$ is asserted when ADSP-21161 processor writes a word to external memory or IOP registers of other ADSP-21161 processors. External devices must assert WR for writing to ADSP-21161 processor's IOP registers. In a multiprocessing system, WR is driven by the bus master. $\overline{\text{WR}}$ has a 20 kΩ internal pull-up resistor that is enabled for processors with ID2-0=00x.

I (Input), S (Synchronous), o/d (Open Drain), O (Output), A (Asynchronous), a/d (Active Drive), T (Three-state, when $\overline{\text{SBTS}}$ or $\overline{\text{HBR}}$ is asserted, or when the processor is a bus slave)

Banked External Memory

The processor divides external memory into four equal-size, fixed banks. Bank sizes are 16 Mword for non-SDRAM and 64 Mword for SDRAM. By mapping peripherals into different banks, systems can accommodate I/O devices with different timing requirements. For information on configuring these memory banks for waitstates and synchronous or asynchronous access modes, see "Setting Data Access Modes" on page 5-32.

On the ADSP-21161 processor, Bank 0 starts at address 0x20 0000 in external memory and is followed in order by Banks 1, 2, and 3. When the processor generates an address located within one of the four banks, the processor asserts the corresponding memory select line, $\overline{\text{MS}}$3-0.

The $\overline{\text{MS}}$3-0 outputs serve as chip selects for memories or other external devices, eliminating the need for external decoding logic. For more information, see "Timing External Memory Accesses" on page 7-13. The $\overline{\text{MS}}$3-0 lines are decoded memory address lines that change at the same time as the other address lines. When no external memory access is occurring, the $\overline{\text{MS}}$3-0 lines are inactive.

Unlike previous SHARCs, strobe assertion for conditional instructions occurs only when the instruction condition code evaluates as true.

Boot Memory

Most often, the processor only asserts the $\overline{\text{BMS}}$ memory select line when the processor is reading from a boot EPROM. This line allows access to a separate external memory space for booting. Both ROM boot memory waitstates and the mode of the WAIT register are applied to $\overline{\text{BMS}}$-selected accesses.

The $\overline{\text{BMS}}$ output is only driven by the processor bus master. For more information on booting, see "Bootloading Through The External Port" on page 6-70 or "Bootloading Through The Link Port" on page 6-88.

It is also possible to write to boot memory using $\overline{\text{BMS}}$. For more information, see "Using Boot Memory" on page 5-35.

Idle Cycle

A bus idle cycle is an inactive bus cycle that the processor automatically generates to avoid data bus driver conflicts. Such a conflict can occur when a device with a long output disable time continues to drive after $\overline{\text{RD}}$ is deasserted while another device begins driving on the following cycle. Idle cycles are also required to provide time for a slave in one bank to three-state its ACK driver, before the slave in the next bank enables its ACK driver in the synchronous access modes. Figure 7-3 shows idle cycle insertion between a synchronous read and a zero-wait, synchronous write in cycle 3.

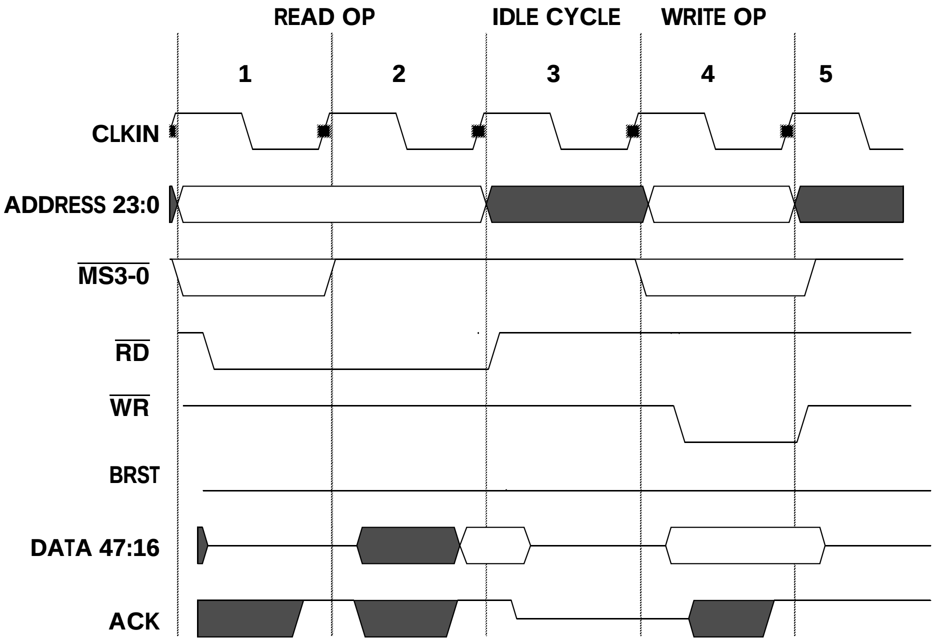


Figure 7-3. Idle Cycle Example

All timing diagrams show the default data bus width DATA[47:16]. When the full bus is enabled for 48-bit non-packed execution of instructions or transfers of data with the PX register, the data bus width is 48 bits, DATA47:0.

To avoid this data bus driver conflict, the processor generates an idle cycle in the following cases:

- On a transition from a read operation to a write operation in the same bank.
- On a transition from one bank or multiprocessor memory ID space to any other bank or multiprocessor slave ID space, independent of access mode.

ⓘ Unlike previous SHARCs, the ADSP-21161 processor does not support idle cycle insertion on a page boundary crossing.

Data Hold Cycle

The data hold cycle is another configurable memory access feature for adding cycles much like waitstates, as discussed in "Setting Data Access Modes" on page 5-32. A hold time cycle is an inactive bus cycle that the processor automatically generates at the end of a read or write to allow a longer hold time for address and data. The address, data (if a write), and bank select (if in banked external memory) remain unchanged and are driven for one cycle after the read or write strobes are deasserted. The processor inserts the data hold cycle only in asynchronous mode and only if the number of programmed waitstates code (EBxWS) is 010-111. Figure 7-4 demonstrates a hold time cycle appended to an asynchronous write access (EBxWS=011).

ⓘ The ADSP-21161 processor does not append an Idle cycle after a Hold cycle.

Multiprocessor Memory Space Waitstates and Acknowledge

Multiprocessor memory space uses only the synchronous transfer protocols, using the zero-waitstate access for writes and a minimum one-waitstate access for reads. Slave processors deassert ACK if more access

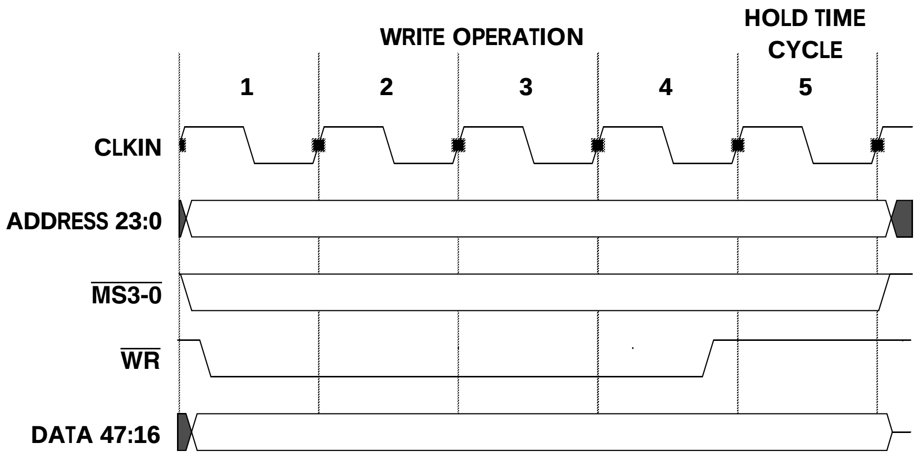


Figure 7-4. Hold Time Cycle Example

time is required. DMA burst transfers are only defined for direct read access of a processor slave's external port buffers (EPBx). For more information, see "Multiprocessor (MP) Interface" on page 7-87.

The ADSP-21161 processor does not support the MMSWS bit from previous SHARCs.

Timing External Memory Accesses

Memory access timing for external memory space and multiprocessor space is the same. For exact timing specifications, refer to the *ADSP-21161N DSP Microcomputer Data Sheet.*

The processor can interface to external memories and memory-mapped peripherals that operate asynchronously with respect to CLKIN. The processor also supports synchronous external memories and memory-mapped peripherals. Synchronous devices derive all of their bus timing from CLKIN of the processor.

ⓘ CLKOUT with $\overline{\text{CLKDBL}}$ tied low can be used as a clock source to peripherals only in single processor systems.

The synchronous interface mode supports DMA burst transfers, which can significantly improve bus throughput for large, contiguous block transfers. The synchronous interface protocols are compatible with Synchronous Burst SRAMS (SBSRAMs) from a variety of vendors. In a multiprocessing system, the ADSP-21161 processor must be the bus master in order to access external memory.

ⓘ When interfacing to synchronous external memories, CLKIN must be used to provide the clock source to the synchronous device.

Asynchronous Mode Interface Timing

Figure 7-5 shows typical timing for an asynchronous read or write of external memory. Here, the CLKIN clock signal indicates that the access occurs within a single CLKIN cycle. All timing for the master processor is derived synchronously from CLKIN. The asynchronous slave mode modifies the basic synchronous access to better support slaves whose timing is not derived from CLKIN.

Figure 7-6 shows timing relationships used by the asynchronous external access mode. In this mode,

- The strobes assert and deassert based on timing derived from an internal clock whose frequency is twice that of the core clock. (This differs from synchronous mode where the strobes assert from the same edge.) The trailing edge timing is derived from the rising edge of the internal version of CLKIN.

- The $\overline{\text{MSx}}$ memory select lines are held stable for the entire access. (This differs from synchronous read or synchronous write—minimum 2-cycle—modes where the memory select lines are deasserted after the first cycle of the transfer that uses ACK.)

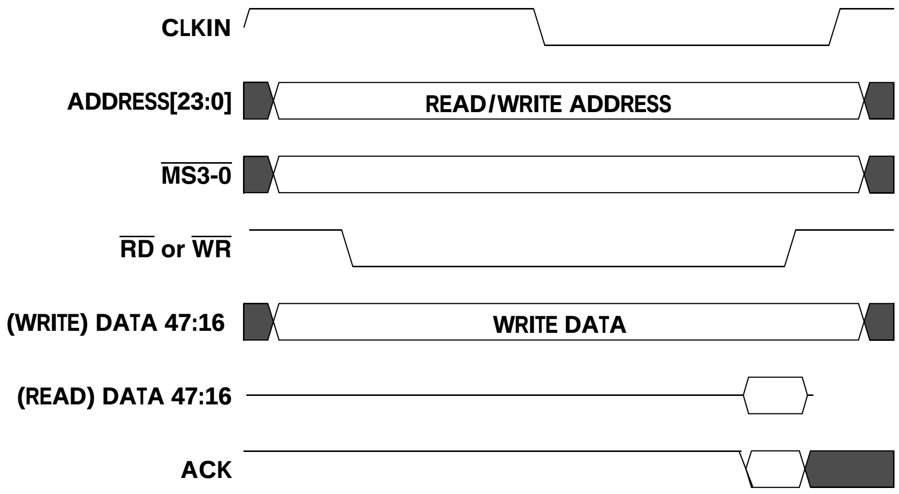


Figure 7-5. External Memory Asynchronous Access Cycle

- For read operations, DATA47:16 are sampled by the processor on the rising edge of the RD. This differs from synchronous mode where DATA47:16 are sampled by the internal version of CLKIN.

- Asynchronous memories or memory mapped devices that require added waitstates through the deassertion of ACK must be configured for a minimum of one internal waitstate due to a potential lack of sufficient decode time for ACK delay from address/selects Refer to *ADSP-21161N DSP Microcomputer Data Sheet* for timing specifications.

Asynchronous Mode Read—Bus Master

Processor bus master reads of external memory, in asynchronous mode, occur with the following sequence of events as shown in Figure 7-5.

1. The processor samples ACK synchronously. If ACK is asserted, the processor drives the read address and asserts a memory select signal (MS3-0) to indicate the selected bank. A memory select signal is not

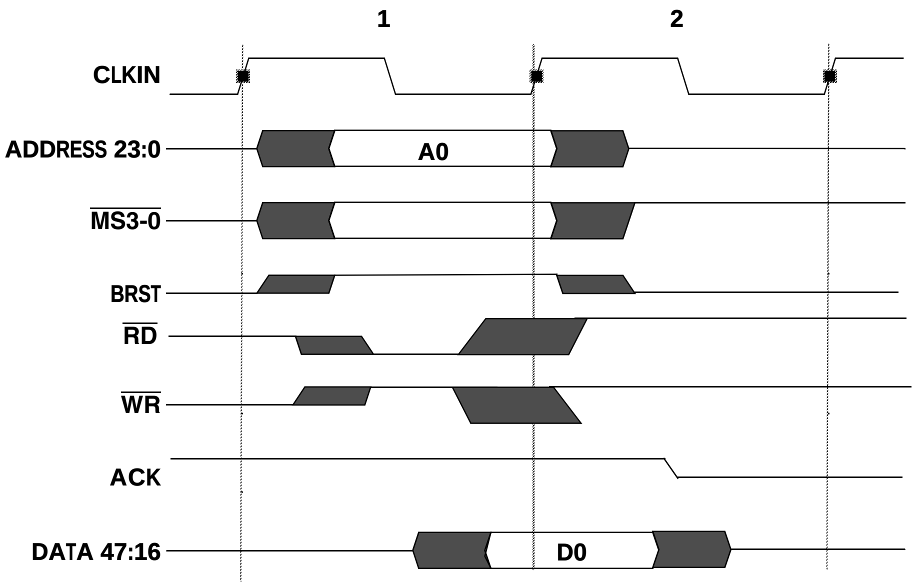


Figure 7-6. Asynchronous Access Timing Derivation

deasserted between successive accesses of the same memory bank. If `ACK` is sampled deasserted, the processor waits one `CLKIN` cycle to sample `ACK` again.

2. The processor asserts the read strobe.

3. The processor checks whether waitstates are needed. If so, the memory select and read strobe remain active for additional cycles. Waitstates are determined by a combination of the state of the external acknowledge signal (`ACK`) **AND** the internally programmed waitstate count.

4. The processor deasserts the read strobe in the cycle where no further waitstates are indicated. The data bus (DATA47:16) is sampled on the rising edge of the read strobe.

5. If a hold cycle is programmed for the accessed bank (via the EBxWS parameter of the WAIT register), the address bus and memory selects are held stable for an additional cycle. If initiating another read memory access to the same bank, the processor drives the address and memory select for that access in the next cycle.

Asynchronous Mode Write—Bus Master

Processor bus master writes to external memory, in asynchronous mode, occur with the following sequence of events as shown in Figure 7-5.

1. The processor samples ACK synchronously. If ACK is asserted, the processor drives the write address and asserts a memory select signal ($\overline{\text{MS}}$3-0) to indicate the selected bank. A memory select signal is not deasserted between successive accesses of the same memory bank. The processor also drives the write data (DATA47:16). If ACK is sampled deasserted, the processor waits one CLKIN cycle to sample ACK again.

2. The processor asserts the write strobes.

3. The processor checks whether waitstates are needed. If so, the memory select and write strobe remain active for additional cycles. Waitstates are determined by a combination of the state of the external acknowledge signal (ACK) **AND** the internally programmed waitstate count.

4. The processor deasserts the write strobes near the end of the cycle where no further waitstates are indicated.

5. The processor three-states its data outputs, unless the next access is also a write to the same bank, or if a hold cycle is programmed for the accessed bank using the EBxWS parameter of the WAIT register. If

a Hold cycle is inserted, the address bus, data bus, and memory selects are held stable for an additional cycle. If initiating another memory access to the same bank, the processor drives the address and memory select for the next access in the following cycle.

Synchronous Mode Interface Timing

Any slave addressed by a processor in a bank configured for synchronous transfer mode must use a clock with frequency and phase characteristics similar to CLKIN on the processor. The slave samples all inputs, and drives all outputs on the rising edge of this clock.

Except for zero-waitstate writes, the slave must assert ACK at least twice for each access; once to acknowledge the address/command (strobe assertion) and once (if not a burst) or more to acknowledge the data transfer. Due to insufficient decode time, the first ACK can be due to the keeper latch (internal pullup enabled for ID=00x) holding the assertion of ACK from the previous slave.

The following notes apply to all synchronous access modes:

- A slave recognizes the start of a valid bus operation by synchronously sampling one or more of the strobes and ACK asserted. ACK assertion is by the previous bus slave, allowing a new bus access to launch.
- For each of the non-burst, synchronous read/write accesses (except zero-waitstate writes), the master recognizes the end of the access as the cycle in which:
    1. The slave samples or drives data in response to a valid operation driven by the master (read or write),
    2. The slave asserts ACK to the master (except for zero-waitstate write operations), and

   3. The number of waitstates for read or write access to that bank has occurred—asserting `ACK` does not terminate the wait count early.

- The program must select a number of waitstates that is consistent with the access time for the slave addressed by that external memory bank.

- For the zero-waitstate writes, the access can only be extended beyond one clock cycle by deasserting `ACK` in the cycle of the transfer. This extension can occur on back-to-back writes in which `ACK` is deasserted due to full write buffer capacity from the previous write. Otherwise, slaves can asynchronously deassert `ACK` in the first cycle.

- Deasserting `ACK` during the initial command phase does inhibit waitstate count and change of bus signals. After the first `ACK` assertion, deasserting `ACK` for the data phase does not inhibit waitstate counting.

- Only one slave (or driver for `ACK`) should be allocated per external memory bank. More than one slave may introduce `ACK` drive contention.

- The read/write strobes for an access do not assert until `ACK` is sampled asserted. This conditional strobe assertion delays the start of an access until `ACK` is asserted by the previous slave. This sampling is because the slave target of a single-cycle write operation may have deasserted `ACK` in the cycle (due to a previous write access), to stall further writes to that slave. To provide a cycle for the previous slave to three-state its `ACK` driver before the next slave drives `ACK`, the next operation to a new bank must not launch on the bus.

- Write/read access stalls (no state change, other than internal wait-state counting) on the bus if ACK is deasserted in cycle(s) of data transfer.
- The last read/write operation must be acknowledged via ACK before a transition to a new bus master (BTC), bank, or multiprocessor space slave occurs. The master always inserts an Idle cycle on this transition. No pipelining can occur across these boundaries.

Synchronous Mode Read—Bus Master

An example synchronous read cycle appears in Figure 7-7. Propagation delays are not shown in this timing diagram. Because a synchronous access requires a rising clock edge for the slave to sample the asserted signals of the master (and for the master to sample the signals of the slave), the minimum read access in the synchronous mode is two CLKIN cycles.

In synchronous access mode, the waitstate selection in the WAIT register (EBxWS) must be 001 or greater. EBxWS=000 is not supported in synchronous access mode.

This example demonstrates a minimum latency, one-waitstate, 32-bit (normal word) read, from external memory.

Bus master synchronous reads from external memory occur with the following sequence of events as shown in Figure 7-7:

1. (cycle 1) If ACK is sampled as asserted at the beginning of cycle 1, the processor drives the read address and asserts a memory select signal ($\overline{\text{MS3-0}}$) to indicate the selected bank. The processor asserts the $\overline{\text{RD}}$ strobe. The read strobe is not deasserted between successive read accesses of the same memory bank.
2. (cycle 2) If ACK was sampled as deasserted at the beginning of the cycle (not shown), the $\overline{\text{MSx}}$ strobes would remain asserted. If ACK was sampled asserted, the $\overline{\text{MSx}}$ strobes would deassert. The slave must be capable of detecting that $\overline{\text{MSx}}$ was asserted in cycle 1 and

must retain this information internally. If ACK was deasserted by the previous slave (for a single-cycle write), deassertion of the $\overline{\text{MSx}}$ is delayed.

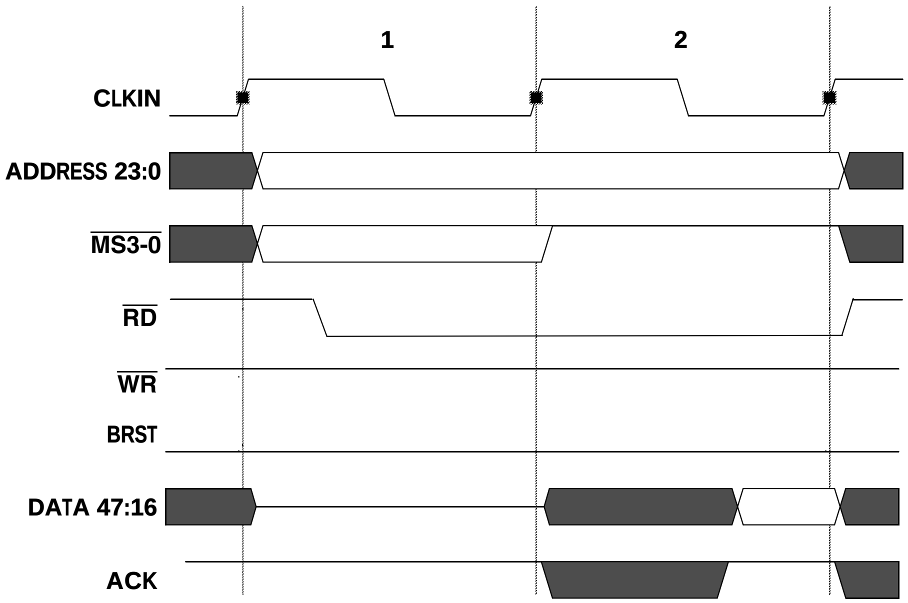


Figure 7-7. Typical Synchronous Read Timing

3. (cycle 2) The processor checks whether more than one waitstate is needed. If so, the read strobe remains active for additional cycle(s). Waitstates are determined by a combination of the state of the external acknowledge signal (ACK) **AND** the programmed waitstate count.

4. (end of cycle 2) The data bus (DATA47:16) is sampled on the rising edge of CLKIN.

5. (cycle 3) If initiating another read memory access to the same bank, the processor drives the address, memory select, and strobe for the next access.

Figure 7-8 shows back-to-back reads to the same bank with the second access stalled for one cycle by the slave deasserting ACK. This example assumes that the EBxWS=001 for this bank, indicating one internal waitstate.

Synchronous Write, Zero-Waitstate Mode

Figure 7-9 on page 7-24 shows typical synchronous write cycle timing. Propagation delays are not shown in this timing diagram. Synchronous access requires a rising clock edge for the slave to sample the asserted signals of the master (and for the master to sample the signals of the slave). In the case of writes, the latency can be reduced to a single cycle if the slave always latches the bus signals on each clock cycle (it does not sample ACK). For example, the slave can not sample the bus, decode that it is being addressed as a slave, and sample the write data of the bus in the following cycle. The slave samples the bus each cycle and decodes the sampled value to determine if that slave was addressed by the write operation. If the slave's write queue goes full with that write, the slave deasserts ACK in the cycle after the write operation transferred on the bus. Any subsequent bus operation (read or write) stalls until ACK is sampled asserted, as shown in cycle 2 of Figure 7-9.

The example demonstrates a minimum latency, zero-waitstate, 32-bit write in cycle 1 followed by a write to the same bank. This write stalls because ACK is deasserted in cycle 2 in response to the write in cycle 1. The second access is a 32-bit write to external memory.

The zero-waitstate write mode provides the highest performance if the slave has sufficient write buffer storage. Systems should use this mode where the slave can always accept one write transfer (unless ACK is deasserted) and can generally accept more than one write. If the slave has only one store buffer, such that it always deasserts ACK after the first write, the

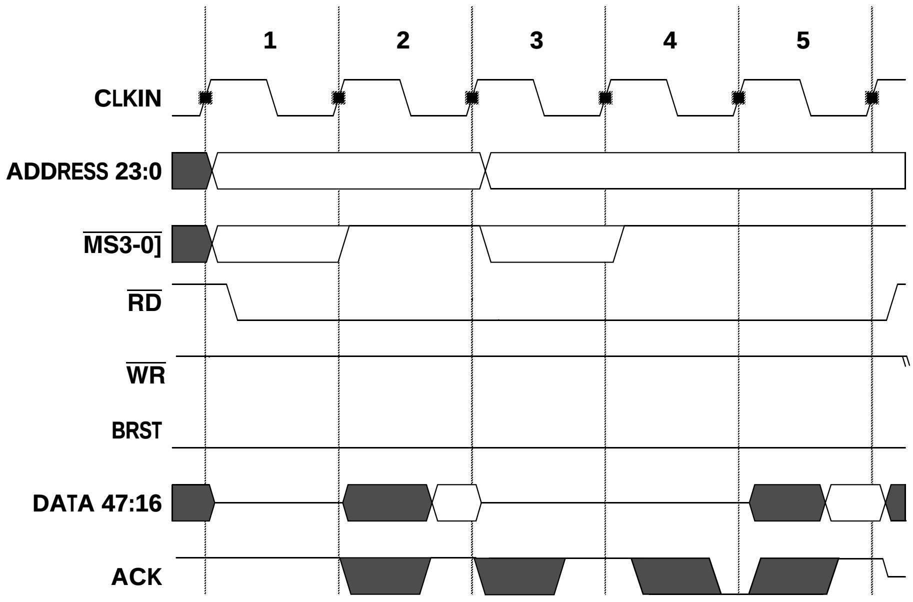


Figure 7-8. Two Synchronous Reads From Same Bank

one-waitstate write mode may be the better choice. The zero-waitstate write mode is targeted towards ASIC/FPGA designs, which implement multiple write buffers (including processor as a slave), and fully pipelined synchronous devices such as SBSRAMs.

Slaves that do not support bursting protocols do not need to connect to the `BRST` signal.

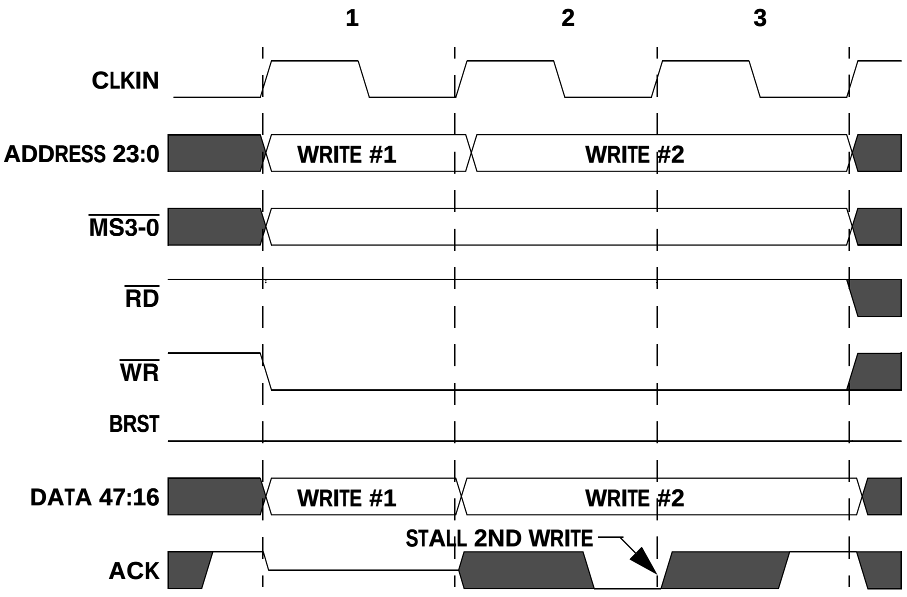


Figure 7-9. Typical Synchronous Write Example

Bus master synchronous writes to external memory occur with the following sequence of events as shown in Figure 7-9:

1. (cycle 1 in Figure 7-9) If `ACK` is sampled asserted at the start of cycle 1, the processor bus master drives the write address and asserts a memory select signal ($\overline{\text{MS3-0}}$) to indicate the selected bank. The processor asserts the $\overline{\text{WR}}$ strobe. The write strobe is not deasserted between successive write accesses of the same memory bank.

2. (cycle 1) The previous slave three-states `ACK`. Note that the previous slave could have driven `ACK` deasserted through cycle 1 if a write in the previous cycle caused its write queue to fill. Only one slave is supported per bank, and any bank transition has an `IDLE` cycle inserted to provide time for the slave to three-state `ACK`.

3. (cycle 2) The processor is initiating another write memory access to the same bank. It drives the address, memory select, and strobe for the next access.

4. (cycle 2) The slave, having decoded that it received a valid write operation in the previous cycle, detects that it cannot accept further bus operations until an element in the write queue becomes available, so it deasserts ACK.

5. (cycle 3) The processor samples ACK deasserted by the slave. It inserts waitstates until ACK is sampled asserted. The write ends in the cycle where ACK is asserted by the slave (end of cycle 3).

Figure 7-10 shows a zero waitstate write, followed by a synchronous read from the same bank. The slave addressed by both accesses determines in cycle 2 that it has no more write capacity. It deasserts ACK in this cycle, in response to the write in cycle 1. In cycle 3, the slave determines that it is now addressed by the master to perform a read and asserts ACK to acknowledge the transfer. The slave asserts ACK in cycle 4 when read data is available to complete the data transfer. The memory select for the read access is held asserted by the master until cycle 4, because ACK was deasserted in cycle 2.

Synchronous Write, One Waitstate Mode

Because some synchronous slaves cannot support a free-running latch function to capture zero-wait bus writes, the processor also supports a minimum two-cycle (minimum one-waitstate) write access. This mode is set using the bank Access Mode bits (EBxAM). For more information on access modes, see Table A-20 on page A-66.

The one-waitstate, synchronous write access is shown in the second write of Figure 7-11. In this example, the first access is to a bank configured for asynchronous writes (cycle 1). In Figure 7-11, this condition is shown by the deassertion of the write strobe before the rising edge of CLKIN for cycle 2. In cycle 2, a bank transition occurs, and an idle cycle is inserted to

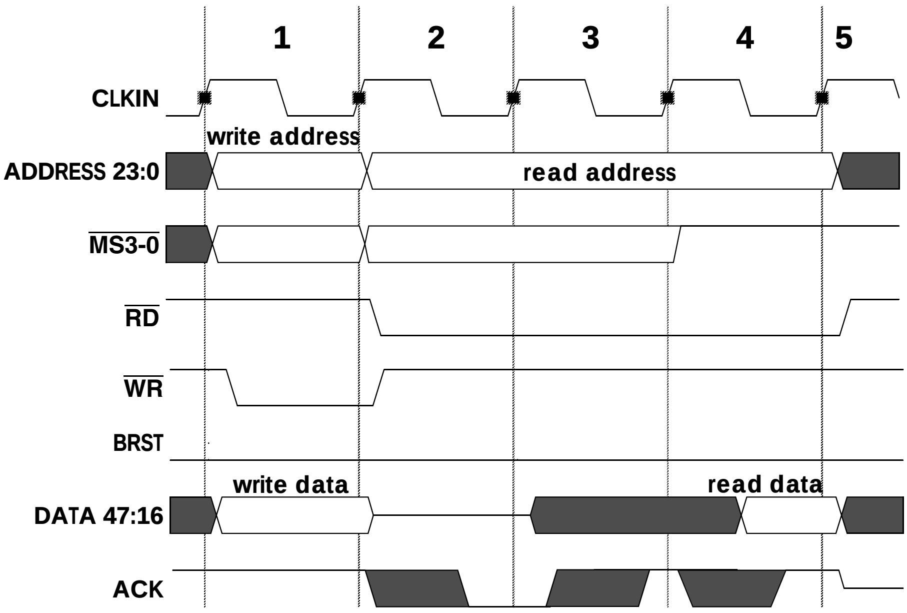


Figure 7-10. Synchronous Write Followed by Synchronous Read Example

allow the slaves to transition ownership of ACK. In cycle 3, the second write begins, to a new bank configured for one-waitstate write access. The address and data are held for a minimum of two cycles. Similar to the synchronous read, $\overline{\text{MSx}}$ deassert in the second cycle of the write (cycle 4), and the waitstate counter decrements if ACK is sampled asserted. The access can be held off the bus by deasserting ACK in cycle 2, or extended by deasserting ACK in cycle 3 (unlikely for a synchronous slave) or cycle 4.

Synchronous Burst Mode Interface Timing

Synchronous burst mode provides improved performance on synchronous operations. The processor supports a DMA-mastered burst mode. If the addressed slave supports this burst transfer, after the one or more waitstates associated with access to the first 32-bit read data transfer, contiguous data can transfer on each subsequent clock cycle, up to a maximum of four 32-bit transfers. Burst accesses support only 32-bit data transfers. Partial data bus width transfers are not supported.

For burst transfers, the master drives the address of the first access on the bus during the entire burst transfer. The master does not increment the address for the slave. Because the maximum length of the burst transfer is four, slaves only need a 2-bit address incrementer to generate the offset address from the address driven by the master on the bus. Table 7-2 shows burst length determination as a function of initial address. If the DMA channel has sufficient data to transfer, it initiates a new burst transfer starting at `ADDR1-0` = 00, 01, or 10 when it wins bus arbitration. Bursts always terminate when `ADDR1-0`=11.

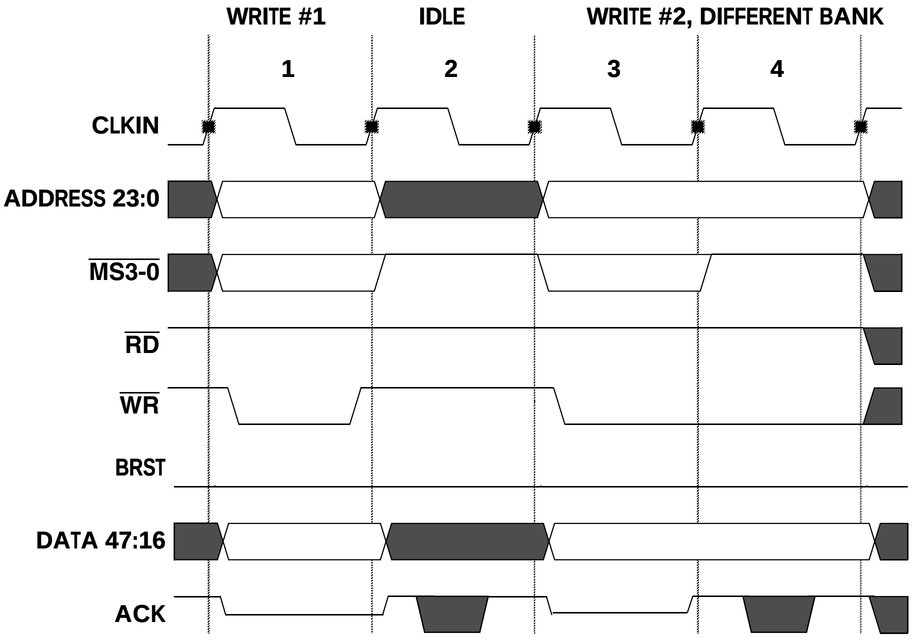


Figure 7-11. Asynchronous Write Followed By Synchronous Write - One-Waitstate Mode

Table 7-2. Linear Burst Address Order

First Address[1:0] (external)	Second Address (internal)	Third Address (internal)	Fourth Address (internal)
00	01	10	11
01	10	11	Burst Terminated[1]
10	11	Burst Terminated[1]	
11	Burst Terminated[2]		

1 Master always terminates burst when internal address[1:0] = 11

2 Master transfers this case as a single synchronous access

An example of a synchronous burst read of length three appears in Figure 7-12. Here, the bank used in the transfer has two waitstates.

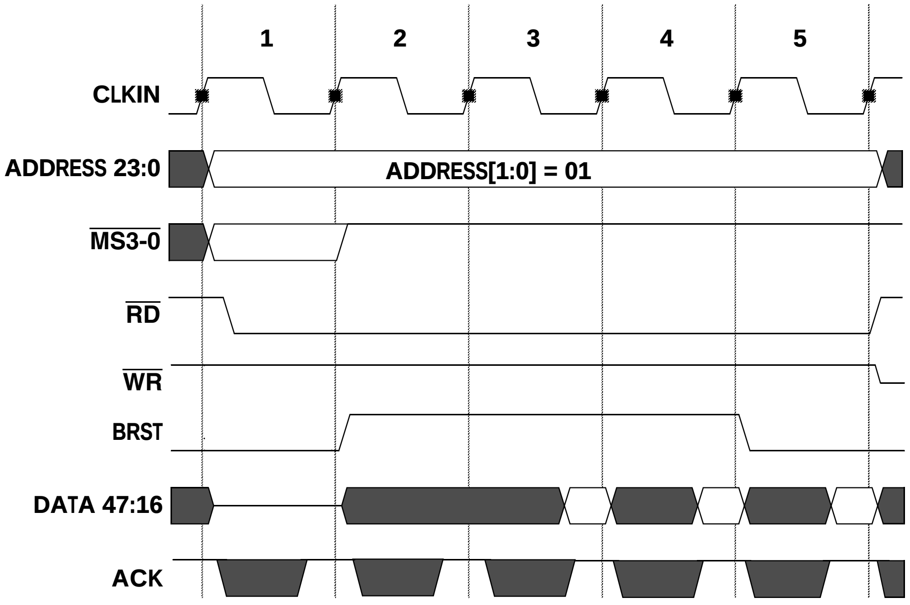


Figure 7-12. External Memory Synchronous Burst Read Example

Burst Length Determination

The DMA arbitration logic reduces the initial access latency by bursting up to the maximum burst length of four when possible, assuming the channel is burst enabled. When a DMA channel wins internal I/O processor arbitration, the channel drives the internal buses as with a non-burst transfer. At the same time, the I/O processor detects whether it can perform a burst transfer, according to the following criteria:

1. The `DMAC` burst enable (`MAXBL1-0`) control bit field is set for that DMA channel.
2. The `EM` register is set to 0 or 1. A value of 0 does not increment `EI`. This is useful when bursting to or from a registered data port, buffer, or register, such as the `EPBx` FIFOs of another processor.
3. The `EC` register is greater than or equal to two (32-bit) words.
4. The `EPBx` FIFO for that channel has at least two 32-bit words to transfer for an external burst write or has at least two empty 32-bit elements to receive data for an external burst read.
5. The two least significant bits of the DMA channel external address are not set (`ADDR1-0` does not equal 11).

Burst Stall Criteria

If the I/O processor determines that it can perform a burst transfer (according to the burst length criteria), the arbitration between the processor core and the I/O processor locks the effective arbitration grant to that DMA channel until:

1. The DMA channel external `ADDR1-0` = 11. By disconnecting the burst on this boundary, a modulo4 (`ADDR23-0`) is effectively implemented, which is required by SBSRAMs, and other slaves with

limited address incrementing capability. For processor-based systems, slaves only need a 2-bit counter to support the address incrementing function of the burst.

2. Space in the EPB FIFO drops to less than two 32-bit elements (if an external bus read), or less than four valid 32-bit elements for external bus writes. This almost full or empty detection is required by the master logic to deassert BRST on the cycle before the end of the burst.

3. EC goes to < 2; the burst pin must negate at EC=1.

4. $\overline{\text{HBR}}$ and $\overline{\text{SBTS}}$ are asserted on the external bus, indicating the deadlock resolution case in which the processor must three-state its outputs and switch into slave mode. For more information, see "Deadlock Resolution" on page 7-82. Assertion of either signal alone does not terminate the burst early. $\overline{\text{HBR}}$ assertion does not receive an $\overline{\text{HBG}}$ until the burst finishes. $\overline{\text{SBTS}}$ assertion causes the master to three-state outputs and insert waitstates.

If any of these conditions occur, normal arbitration between the processor core and I/O processor for the external bus occurs. If the same bursting channel wins arbitration again, a new burst is initiated, introducing at least one lost or dead cycle in the burst throughput for reads.

When arbitration occurs, the DMA channel loses arbitration if any of the following conditions are detected:

1. Higher priority external request for the bus:

    a. $\overline{\text{HBR}}$ asserted

    b. $\overline{\text{BRx}}$ asserted and BMAX time out has occurred

    c. $\overline{\text{BRx}}$ asserted and $\overline{\text{PA}}$ asserted, but not by this master

2. Higher priority internal I/O processor requester:

   a. Processor core request (DAGs or program sequencer)

   b. A higher priority request from another DMA channel or direct read/write access causes this channel to lose arbitration. For more information, see "I/O Processor" on page 6-1.

Synchronous Burst Reads

External memory synchronous burst reads occur with the following sequence of events as shown in Figure 7-12:

1. (cycle 1 in Figure 7-12) If ACK is sampled asserted at the beginning of cycle 1, the processor drives the read address and asserts a memory select signal ($\overline{\text{MS}}$3-0) to indicate the selected bank.

2. (cycle 1) The processor asserts $\overline{\text{RD}}$ strobe to indicate a read request of the slave.

3. (cycle 2) As with the non-burst synchronous read, the processor deasserts the $\overline{\text{MSx}}$ output signal, asserts the BRST output signal, and enables waitstate counting if ACK is sampled asserted at the end of cycle 1.

4. (cycle 2) The processor checks whether more than one waitstates (2 waitstates for this example) is needed. If so, BRST and the read strobe remain active for additional cycle(s).

5. (cycle 3) The slave samples BRST asserted, informing it that the master requests at least one more transfer after the current transfer is acknowledged via ACK by the slave.

6. (cycle 3) The programmed number of waitstates has been counted, and the slave is driving 32-bits of valid data and asserting the ACK signal. This ends the first access.

7. (cycle 4) The slave drives the next 32-bits of contiguous data and asserts ACK. If the slave needs more time to service any one transfer within the burst, it can deassert ACK to stall the bus transfer.

8. (cycle 4) The slave samples BRST asserted, informing it that the master requests at least one more 32-bit transfer.

9. (cycle 5) The master deasserts BRST to inform the slave that this is the last transfer of the burst. In this example, the master deasserts BRST due to the address modulo4 function. The two LSBs of the initial address = 01. The slave increments the address as 01->10->11. This is the maximum offset needed to support from the initial address.

10. (cycle 5) The slave drives valid data for the last transfer, and asserts ACK.

11. (cycle 6) If initiating another burst read memory access to the same bank, the processor asserts the address, memory select, and strobes for the next access. This introduces at least two dead cycles in the back-to-back burst throughput, because the initial waitstate count applies to the first access of the second burst.

12. (cycle 6) With BRST sampled deasserted, the slave concludes its service of the burst request by three-stating the DATA47:16 and ACK drivers.

As a master, the processor supports burst reads on each of the four external port DMA channels. Each channel has an independent burst enable control field (MAXBL1-0).

As a slave, the processor supports read bursts from the EPBx buffers (with the EPBx read). For more information, see "Multiprocessor (MP) Interface" on page 7-87 and "Host Processor Interface" on page 7-42.

ⓘ Because reads of the EPBx FIFO are destructive, the processor slave must deassert ACK on each transfer of the burst to guarantee that it samples the deasserted BRST input before performing the EPBx FIFO read. If your system design uses a similar destructive read data buffer, use precaution when burst reads of the buffer are supported.

Synchronous Burst Writes

The processor can master burst read and write operations in the one-wait-state write access mode (EBxAM=10) if one or more DMA channels are configured appropriately. The processor can master non-burst, zero-wait-state, writes in every cycle. Burst write transfers are not supported in this access mode. Synchronous external devices require at least one cycle of write access latency (for example, bus bridges, SDRAM controllers, and others). These devices may be able to optimize throughput for burst write operations, based on the contiguous, incrementing block transfer information conveyed by the burst protocol. Burst accesses support only 32-bit data transfers; partial data bus width transfers are not supported.

An example of a synchronous burst write appears in Figure 7-13. Here, the bank used in the transfer has the one-waitstate mode, for the first write of the burst.

External memory synchronous burst writes occur with the following sequence of events as shown in Figure 7-13:

1. (cycle 1 in Figure 7-13) If ACK is sampled asserted at the start of cycle 1, the processor drives the write address and asserts a memory select signal ($\overline{\text{MS}}$3-0) to indicate the selected bank. The processor also drives valid data in this cycle. The processor asserts the $\overline{\text{WR}}$ strobe to indicate a write command to the slave.

2. (cycle 2) The slave samples the write command and address. At this point, the slave does not see that a burst write is in progress—the access looks identical to a non-burst synchronous write. If the slave

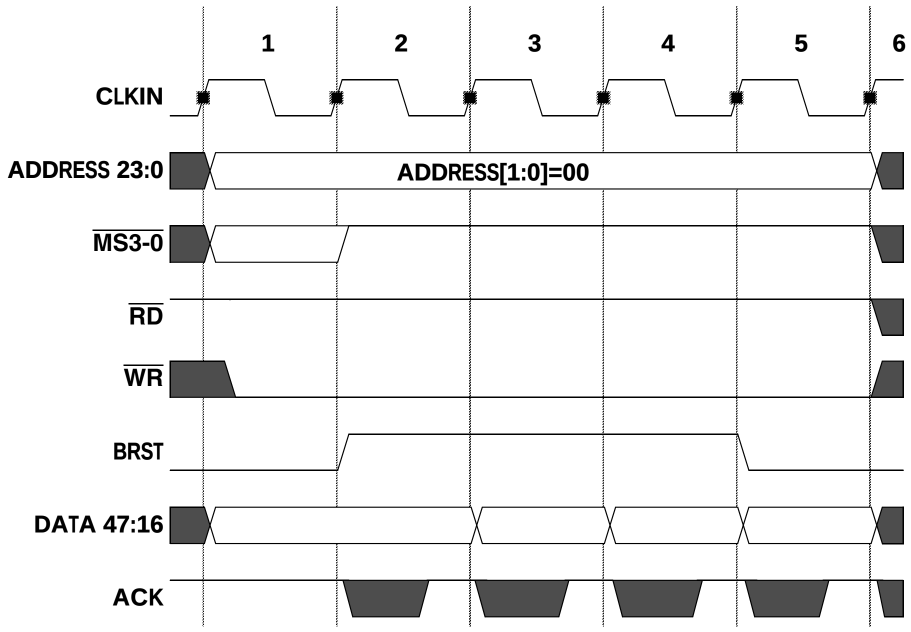


Figure 7-13. External Memory Synchronous Burst Write Example

cannot accept the write command, it deasserts ACK in this cycle to stall the bus until it can. In this example, it has buffer capacity to accept all of the data of the burst, so ACK stays asserted.

3. (cycle 2) If ACK was sampled asserted at the start of the cycle, the processor asserts the BRST output signal and deasserts the $\overline{\text{MSx}}$ output signal.

4. (cycle 3) The processor samples ACK asserted by the slave at the start of the cycle. It increments the data bus to the second of four data transfers within the burst.

5. (cycle 3) The slave samples BRST asserted at the start of the cycle, informing it that the master is writing at least one more 32-bit transfer. The slave samples the second of four data transfers within the burst and asserts ACK.

6. (cycle 4) The processor samples ACK asserted by the slave at the start of the cycle. It increments the data bus to the third of four data transfers within the burst.

7. (cycle 4) The slave samples BRST asserted at the start of the cycle, informing it that the master is writing at least one more 32-bit transfer. The slave also samples the third of four data transfers within the burst, and asserts ACK. If the slave needs more time to service any one transfer within the burst, it can deassert ACK to stall the bus transfer.

8. (cycle 5) The processor samples ACK asserted by the slave at the start of the cycle. It increments the data bus to the last of four data transfers within the burst. The master deasserts BRST to inform the slave that this is the last transfer of the burst.

9. (cycle 5) The slave samples BRST asserted at the start of the cycle, informing it that the master is writing at least one more 32-bit transfer. The slave samples the fourth of four data transfers within the burst and asserts ACK.

10. (cycle 6) If initiating another write burst memory access to the same bank, the processor asserts the address, memory select, and strobes for the next access. This introduces at least one dead cycle in the back-to-back burst throughput, because the initial waitstate count applies to the first access of the second burst.

11. (cycle 6) With BRST sampled deasserted, the slave concludes its service of the burst request by three-stating the ACK driver.

As a master, the processor supports burst writes on each of the four external port DMA channels. Each channel has an independent burst enable control field (MAXBL1-0).

ⓘ As a slave, ADSP-21161 processor does not support burst writes. Bursting is enabled by setting MAXBL1-0 to 01 in the DMACx register. Enabling bursting can corrupt data transmitted during DMA master writes because the MAXBL bit setting is not ignored when the BRST signal is asserted. The ADSP-21161 only supports processor-to-processor single cycle writes. Therefore, no improvement in throughput performance is achieved by enabling bursting. To enable ADSP-21161 to ADSP-21161 DMA driven write transfers, set MAXBL1-0 to 00.

Using External SBSRAM

The processor can connect to a variety of synchronous burst static RAMs (SBSRAMs) with a glueless interface—no external logic required. These synchronous memories can provide high throughput, especially when using the burst read transfer modes. The processor has features to support SBSRAMs from several memory vendors.

The processor can support both flow-through and fully-pipelined SBSRAMs. Using flow-through devices delivers lower latency and higher system performance when a system is designed properly.

ⓘ CLKIN must be used as the clock source for SBSRAM. You cannot use an external crystal when interfacing with SBSRAM.

⊘ Do not use CLKOUT as the clock source for the SBSRAM device. Using an external crystal in conjunction with $\overline{\text{CLKDBL}}$ to generate a CLKOUT frequency is not supported. Negative hold times can result from the potential skew between CLKIN and CLKOUT.

The processor can support SBSRAMs on any of the four external memory banks. The processor supports SBSRAM single transfer reads and writes and SBSRAM burst read transfer operations.

ⓘ Burst write transfers are not supported, because the single-write feature of SBSRAMs achieves the same throughput level, with less complexity.

SBSRAM support is enabled by configuring the bank access mode (`EBxAM`) bits for synchronous, one-cycle writes and waitstate (`EBxWS`) bits for one waitstate (flow-through SBSRAMs) or two waitstates (fully pipelined SBRAMs). For more information on programming access modes and waitstates, see the `WAIT` register bits in Table A-20 on page A-66.

If burst read transfer functionality is needed, one or more of the external port DMA channels must be configured appropriately. Because burst transfers are controlled at the DMA channel, the DMA sequence must make sure that the DMA burst transfer addresses a memory bank or slave that supports the read burst transfer.

Figure 7-14 and Table 7-3 show how the processor I/O should be connected to the SBSRAM I/O. Table 7-3 assumes a 512 Kbyte SBSRAM array consisting of one bank with a 3.3V, 32K x 32 device. The names of the SBSRAM signals may vary from between vendors.

🚫 Figure 7-14 is for illustrative purposes—actual system designs may differ and must be carefully analyzed to determine the actual system topology.

The SBSRAM devices are fully synchronous devices, except for the output enable. The processor issues commands and updates the SBSRAM address latches, as a controller, using the $\overline{\texttt{ADSC}}$ input of the SBSRAMs, rather than the $\overline{\texttt{ADSP}}$ processor input. Using the $\overline{\texttt{ADSC}}$ SBSRAM input enables single cycle writes and simplifies SBSRAM deselect operations.

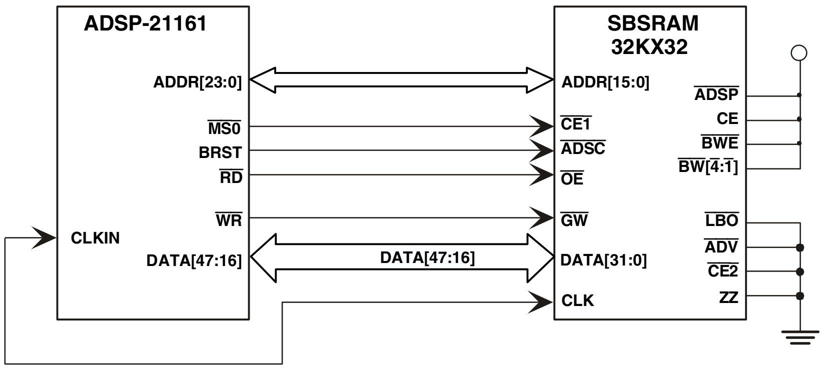


Figure 7-14. SBSRAM System Interface Example

Table 7-3. ADSP-21161 to SBSRAM Signal Mapping

ADSP-21161	SBSRAM	Comment
CLKIN	CLK	Clock input of SBSRAM should be driven by CLKIN of the processor.
ADDR15-0	ADDR15-0	Address connection
$\overline{\text{MSx}}$	$\overline{\text{CE}}$	Chip Enable, active low
BRST	$\overline{\text{ADSC}}$	Address Status Controller, active low
$\overline{\text{RD}}$	$\overline{\text{OE}}$	Asynchronous Output Enable of SBSRAM, active low
$\overline{\text{WR}}$	$\overline{\text{GW}}$	Global Write Enable of SBSRAM, active low
DATA47:16	DATA31-0	I/O of SBSRAM (High word of bus, odd address)
No connect	CE	Chip Enable, active high, always asserted (Vdd)
No connect	$\overline{\text{CE2}}$	Second Chip Enable, always asserted (GND)
No connect	$\overline{\text{ADSP}}$	Always Deasserted (Vdd)
No connect	$\overline{\text{ADV}}$	Always Asserted (GND)
No connect	$\overline{\text{BWE}}$	Byte Write Enable, always deasserted (Vdd)
No connect	$\overline{\text{BW4-1}}$	Byte Write Selects, always deasserted (Vdd)

Table 7-3. ADSP-21161 to SBSRAM Signal Mapping (Cont'd)

ADSP-21161	SBSRAM	Comment
No connect	$\overline{\mathrm{LBO}}$	Linear Burst Order, active low, always asserted (GND)
No connect	ZZ	Sleep Mode Enable, active high, always deasserted (GND)

By asserting the $\overline{\mathtt{ADV}}$ (advance address) input to the SBSRAM, the device is continuously attempting to burst. This input is ignored when $\overline{\mathtt{ADSC}}$ is asserted. Because the $\mathtt{BRST}/\overline{\mathtt{ADSC}}$ signal is always low for a single access or the first access of a burst, the SBSRAM always updates its address latches correctly. For the subsequent transfers (up to three, after the initial access) of a read burst, the SBSRAM samples $\mathtt{BRST}/\overline{\mathtt{ADSC}}$ high. The asserted $\overline{\mathtt{ADV}}$ correctly advances the internal address count of the SBSRAM.

The processor issues four types of bus operations to the SBSRAMs, as shown in Table 7-4.

Table 7-4. SBSRAM Partial Truth Table

SBSRAM Operation	$\overline{\mathrm{CE1}}$ $\overline{\mathrm{MSx}}$	$\overline{\mathrm{ADSC}}$ BRST	$\overline{\mathrm{ADV}}$[1]	$\overline{\mathrm{GW}}$ $\overline{\mathrm{WR}}$	$\overline{\mathrm{OE}}$ $\overline{\mathrm{RD}}$	I/O
Read cycle, begin burst	L[2]	L	X	H	L	Data
Write cycle, begin burst	L	L	X	L	H	Hi-Z
Read cycle, continue burst	X	H	L	H	L	Data
Deselect Cycle	H	L	X	X	X	Hi-Z
All other signal inputs held static per Figure 7-14						

1 $\overline{\mathrm{ADV}}$ statically held asserted, low

2 L=low, H=High, X=don't care, Hi-Z=three-stated, high impedance output

Single read or write transfers, and the first transfer of a burst read, use the read or write cycle and begin burst bus operation. Burst write transfers are not supported. The subsequent transfers (up to three) of a read burst use the read cycle and continue burst bus operation. The last cycle of any read access performs a deselect bus operation ensure that the SBSRAM data buffers remain three-stated for accesses to other banks.

The write operations are achieved by configuring the appropriate bank of the processor to synchronous minimum one-cycle write mode. The synchronous read waitstate count should be programmed to one for flow-through SBSRAMs, or two for fully pipelined SBSRAMs.

SBSRAMs are not stalled, or suspended, by assertion of ACK in this configuration. Systems should not deassert ACK during any SBSRAM access. The processor has a weak pullup device on ACK; ACK does not need to be driven during an access to a slave which does not or cannot control ACK.

Figure 7-15 demonstrates a burst read of the flow-through SBSRAM, followed by a single write to the SBSRAM, and a single read of the SBSRAM. For burst operations, the deasserting BRST is not required in the last cycle of the burst transfer. The processor's burst protocols also support ASIC/FPGA systems. The pipelined end-of-burst indicator may be useful in these systems.

The SBSRAM array size can be increased from the example by using higher density devices or implementing multiple banks of SBSRAM. Multiple banks are possible using the depth expansion feature of the SBSRAMs and the multiple memory select outputs of the processor.

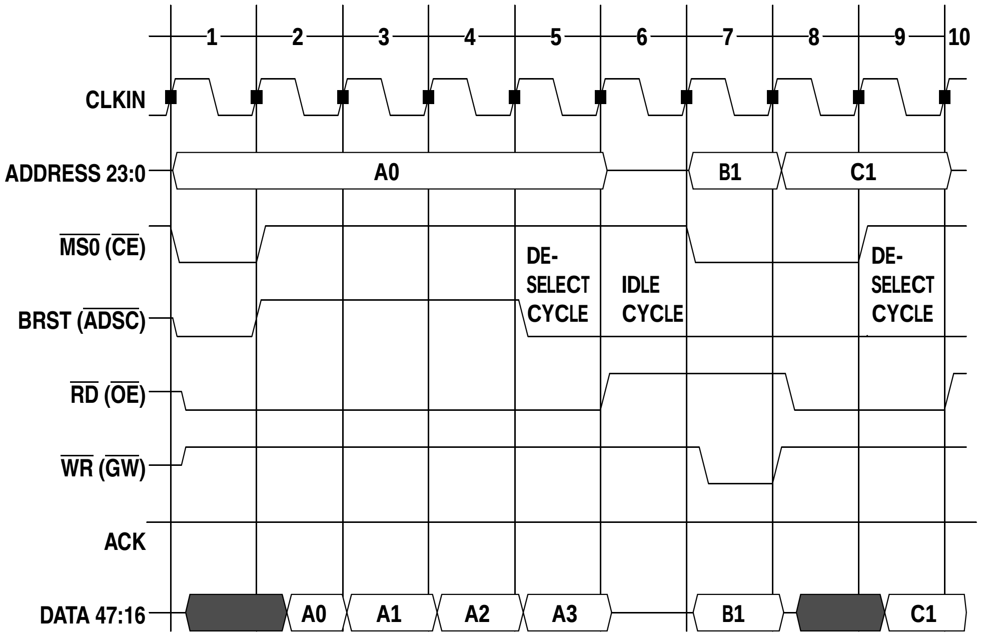

1
2
3
4
5
6
7
8
9
10
CLKIN
ADDRESS 23:0
A0
B1
C1
$\overline{\text{MS0}}$ ($\overline{\text{CE}}$)
BRST ($\overline{\text{ADSC}}$)
$\overline{\text{RD}}$ ($\overline{\text{OE}}$)
$\overline{\text{WR}}$ ($\overline{\text{GW}}$)
ACK
DATA 47:16
DE-SELECT CYCLE
IDLE CYCLE
DE-SELECT CYCLE
A0
A1
A2
A3
B1
C1

Figure 7-15. SBSRAM – Burst Read, Single Write, Single Read

SBSRAM Restrictions

SBSRAM (or other synchronous peripherals such as bridge chips) is restricted using the same external clock generator source provided to the CLKIN pin of the processor.

Do not use CLKOUT as the clock source to the SBSRAM. The clock source connected to both the CLKIN and the clock input of the SBSRAM must be a clock source provided by an external oscillator or other clock source. External crystals in conjunction with the internal clock generator (and $\overline{\text{CLKDBL}}$) should not be used to generate a CLKOUT source for the SBSRAM.

Host Processor Interface

The ADSP-21161 processor's host interface supports connecting the processor to 8-, 16- or 32-bit microprocessor buses. By providing an address, a data bus, and memory control signals—such as read, write and chip select—a host may access any device on the processor bus as if it were a memory. The processor accommodates asynchronous data transfers, allowing the host to use a different clock frequency. For maximum host throughput and low and high pulse widths for $\overline{\text{WR}}$ and $\overline{\text{RD}}$, refer to the *ADSP-21161N processor Microcomputer Data Sheet*.

The ADSP-21161 processor host processor interface does not support synchronous data transfers.

Figure 7-16 shows an example of how to connect a host processor to the ADSP-21161 processor and Table 7-5 defines the processor pins used in host processor interfacing.

The host accesses the ADSP-21161 processor through its external port. Figure 6-5 on page 6-23 shows a block diagram of the external port, I/O processor, and FIFO data buffers, illustrating the on-chip data paths for host-driven transfers. The four external port DMA channels are available for use by the host—DMA transfers of code and data can be performed with low software overhead.

The host processor requests and controls the processor's external bus with the host bus request ($\overline{\text{HBR}}$) and host bus grant ($\overline{\text{HBG}}$) signals. Host logic does not need to duplicate the distributed multiprocessor arbitration protocol of the DSPs. After the host gets control of the bus, the host transfers data asynchronously. The host bus may be 8-, 16-, or 32-bits wide for asynchronous transfers.

The host also uses the chip select ($\overline{\text{CS}}$) and ready (REDY) signals. After getting control of the bus, the host can read and write to any of the processor's I/O processor registers, including the EPBx FIFO buffers. The host uses I/O processor registers such as SYSCON and SYSTAT to control and

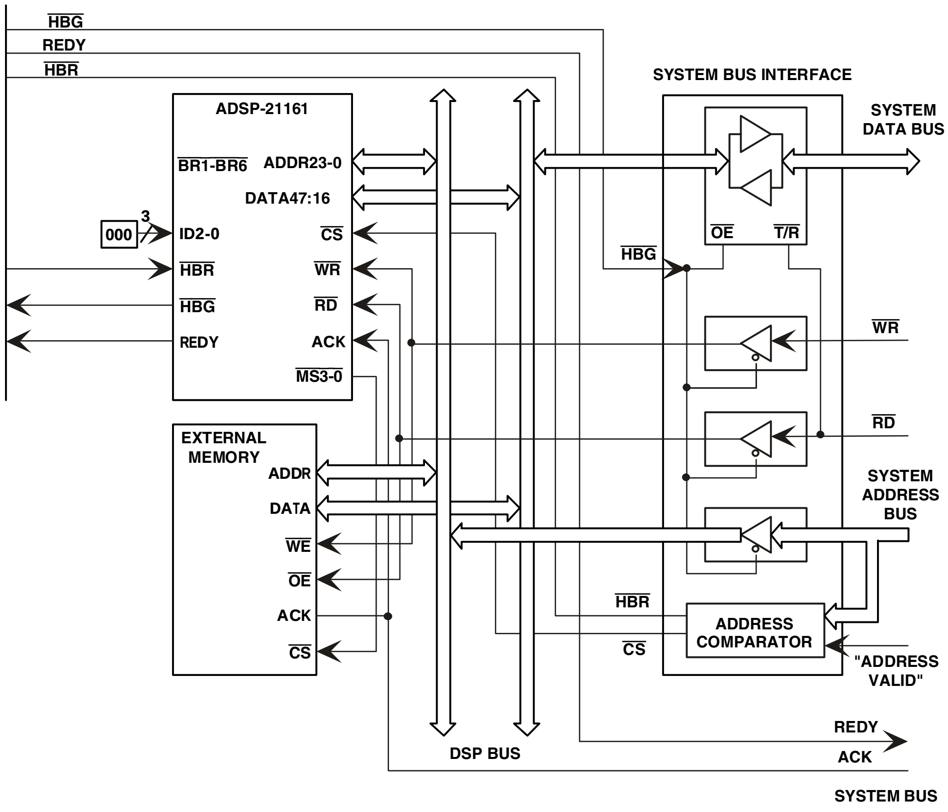


Figure 7-16. Example Processor to Host System Interface

monitor the processor and to set up DMA transfers. DMA transfers are controlled by the processor's I/O processor after they are set up by the host. In a multiprocessor system, the host can access the I/O processor registers of every ADSP-21161.

Data written to and read from the processor can be packed or unpacked into different word widths. The host bus width control bits (`HBW`) in the `SYSCON` register configure data packing and unpacking.

Table 7-5. Host Interface Signals

Signal	Type	Definition
$\overline{\text{HBR}}$	I/A	**Host Bus Request**. Must be asserted by a host processor to request control of the ADSP-21161 processor's external bus. When $\overline{\text{HBR}}$ is asserted in a multiprocessing system, the ADSP-21161 processor that is bus master relinquishes the bus and asserts $\overline{\text{HBG}}$. To relinquish the bus, the ADSP-21161 processor places the address, data, select, and strobe lines in a high impedance state. $\overline{\text{HBR}}$ has priority over all ADSP-21161 processor bus requests ($\overline{\text{BR6-1}}$) in a multiprocessing system.
$\overline{\text{HBG}}$	I/O	**Host Bus Grant.** $\overline{\text{HBG}}$ acknowledges an $\overline{\text{HBR}}$ bus request, indicating that the host processor may take control of the external bus. $\overline{\text{HBG}}$ is asserted (held low) by the processor until $\overline{\text{HBR}}$ is released. In a multiprocessing system, $\overline{\text{HBG}}$ is output by the processor bus master and is monitored by all others.
$\overline{\text{CS}}$	I/A	**Chip Select**. Asserted by host processor to select the ADSP-21161 processor.

I=Input, S=Synchronous, (o/d)=Open Drain, O=Output, A=Asynchronous, (a/d)=Active Drive

Acquiring the Bus

For a host processor to gain access to the processor, the host must first assert $\overline{\text{HBR}}$, the host bus request signal. $\overline{\text{HBR}}$ has priority over all $\overline{\text{BRx}}$ multiprocessor bus requests. When asserted, $\overline{\text{HBR}}$ causes the current master to give up the bus to the host after the it finishes the current bus operation. If the current operation is a burst transfer, the change in bus mastership interrupts the transfer on a modulo4 boundary.

The current bus master signals that it is transferring ownership of the bus by asserting $\overline{\texttt{HBG}}$ (low) when the current bus operation ends. The cycle in which control of the bus is transferred to the host is called a Host Transition Cycle (HTC).

Bus slaves respond to $\overline{\texttt{HBG}}$ assertion with or without the assertion of $\overline{\texttt{HBR}}$. Therefore erroneous assertions of $\overline{\texttt{HBG}}$ (glitching, etc.) can cause slave DSPs to believe that the host is the current bus master.

Figure 7-17 shows the timing for the host acquiring the bus. $\overline{\texttt{HBG}}$ is asserted while the bus master releases control of the bus and remains asserted until $\overline{\texttt{HBR}}$ is sampled deasserted by the ADSP-21161 processor. The cycle in which control of the bus is released by the bus master is called the processor's Bus Transition Cycle (BTC). $\overline{\texttt{HBG}}$ freezes the multiprocessor bus arbitration during the time that the host has control of the bus. $\overline{\texttt{HBG}}$ may be used to enable the host's signal buffers, as shown in Figure 7-16 on page 7-43, Figure 7-24 on page 7-80, and Figure 7-25 on page 7-81. Arbitration is frozen due to the current bus master continuously asserting its $\overline{\texttt{BRx}}$. While $\overline{\texttt{HBG}}$ is asserted in a multiprocessor system, the DSPs continue to assert their $\overline{\texttt{BRx}}$ outputs, as in normal operation, but no BTC occurs. The current bus master keeps its $\overline{\texttt{BRx}}$ output asserted throughout the entire time the host controls the bus.

After $\overline{\texttt{HBR}}$ is asserted, and before $\overline{\texttt{HBG}}$ is given, $\overline{\texttt{HBG}}$ floats for 1 t_{CK} (1 CLKIN cycle). To avoid erroneous grants, $\overline{\texttt{HBG}}$ should be pulled up with a 20kΩ to 50kΩ external resistor.

After the host gets control of the bus, the host can perform transfers with the processor or other system components. To initiate transfers, the host asserts (low) the $\overline{\texttt{CS}}$ and $\overline{\texttt{HBR}}$ inputs of the processor that it intends to access and performs the read or write. The processor does not respond to $\overline{\texttt{CS}}$ until $\overline{\texttt{HBG}}$ is asserted.

The host may also communicate directly with system peripherals, such as SBSRAMs. These transfers occur using the protocol of the peripheral or using the external handshake mode of DMA channels 10 and 11 to control the memory or peripheral. With DMA handshaking, the host only needs to source or sink the data with the correct timing. Either of these solutions may require additional hardware support for the host.

The host is responsible for driving the following signals during the HTC in which it gains control of the bus: ADDR23-0, $\overline{RD}$, $\overline{WR}$, and $\overline{DMAGx}$ (if used in the system). These signals must be driven by the host while the host is bus master. Also, the host must drive or weakly pull up or down the $\overline{MS}$3-0, BRST, CLKIN, $\overline{DMAG1}$, and $\overline{DMAG2}$ signals as required. The bus master three-states these lines, letting the host use them.

The processor with device ID=000 or 001 enables internal pullup devices on the $\overline{MS}$3-0, $\overline{RD}$, $\overline{WR}$, $\overline{DMAR1}$, $\overline{DMAR2}$, $\overline{DMAG1}$, and $\overline{DMAG2}$ signals. The pullup provides a weak current source to hold these signals in the deasserted state when driven to that state.

Excessive system noise can cause these weakly driven signals ($\overline{MS}$3-0, $\overline{RD}$, $\overline{WR}$, $\overline{DMAR1}$, $\overline{DMAR2}$, $\overline{DMAG1}$, and $\overline{DMAG2}$) to be sampled asserted.

The processor with device ID=000 or 001 enables its keeper latches on ADDR23-0 and DATA47-16, BRST, and CLKOUT, so these signals are weakly pulled to the last value driven on them if any of these signals remain undriven for multiple cycles.

During read-modify-write operations, the host should keep $\overline{HBR}$ asserted to avoid temporary loss of bus mastership. $\overline{HBR}$ must remain asserted until after the host completes the last data transfer.

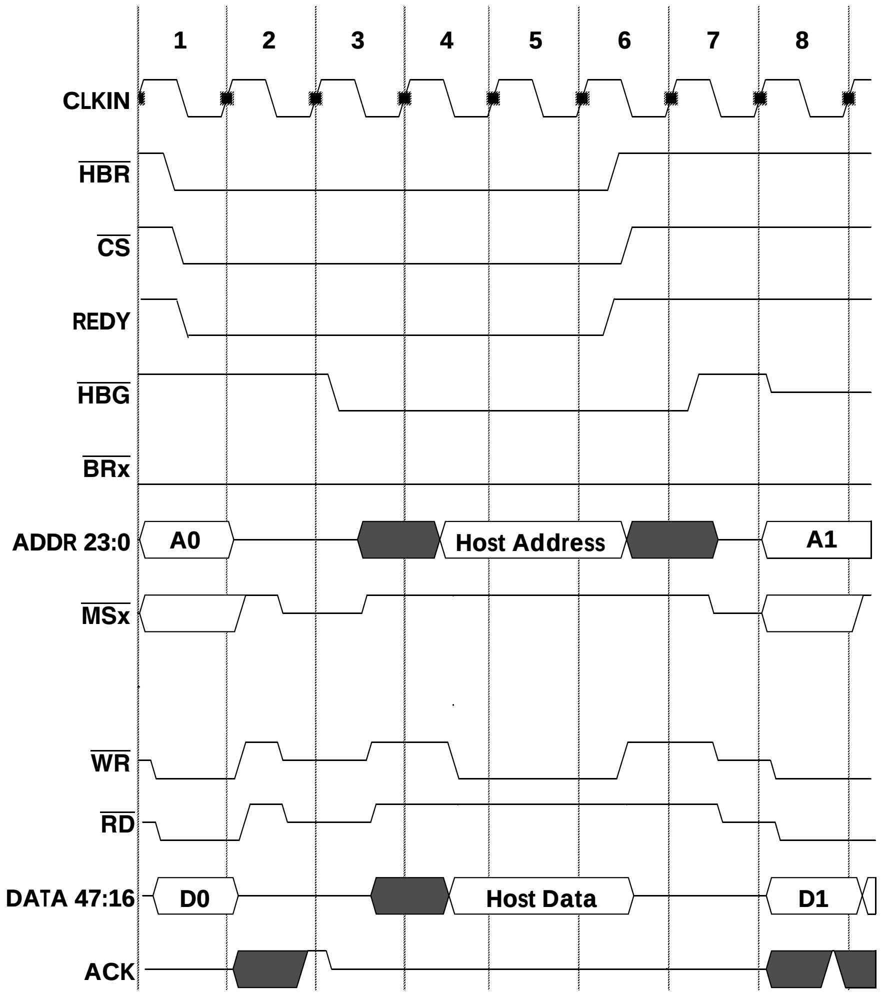


Figure 7-17. Example Timing for Host Acquisition of Bus

The following restrictions apply to bus acquisition by the host:

- If $\overline{\texttt{HBR}}$ is asserted while the processor is in reset, it does not respond with $\overline{\texttt{HBG}}$ until after reset and multiprocessor synchronization is completed.
- The host should keep $\overline{\texttt{HBR}}$ asserted until after the host completes its last data transfer and is ready to give up bus ownership.
- If $\overline{\texttt{SBTS}}$ is asserted after $\overline{\texttt{HBR}}$, the processor enters slave mode and suspends any unfinished access to the external bus.
- In uniprocessor systems (with `ID2-0`=000), the host must assert $\overline{\texttt{CS}}$ in the same cycle as $\overline{\texttt{HBR}}$ to initiate an asynchronous access.

After the host finishes its task, it can relinquish control of the bus by deasserting $\overline{\texttt{HBR}}$. The bus master responds by deasserting $\overline{\texttt{HBG}}$ in the cycle after sampling $\overline{\texttt{HBR}}$ deasserted. In the cycle following deassertion of $\overline{\texttt{HBG}}$, the bus master assumes control of the bus and normal multiprocessor arbitration resumes.

Asynchronous Transfers

To initiate asynchronous transfers after acquiring control of the processor's external bus, the host must assert the $\overline{\texttt{CS}}$ input of the processor to be accessed. The host then drives the address of the I/O processor register to access. To simplify the hardware requirements for external interface logic, only the address bits shown in Table 7-6 need to be driven.

Table 7-6. Address Fields For Asynchronous Host Accesses

Address Bits[1]	Comments
ADDR8-0	Must be driven in all cases.
ADDR16-9	Floating
ADDR19-17	S field[2], floating
ADDR20	M field[2], must be 0

or

ADDR23-21	E field[2], One of these bits must be 1.
ADDR25-24	V field[2], virtual.
ADDR27-26	V field[2], virtual, $\overline{\text{MSx}}$.

1 Setup and hold times for these address lines are specified in the Data Sheet.
2 For a complete description of these address fields, see "Multiprocessor Memory" on page 5-19.

Table 7-6 applies to all asynchronous host access cases, including multiprocessor systems. Fewer address bits may need to be driven depending on the system. For example, in a uniprocessor system, the host does not need to drive the `ADDR20` address pins. Use the following guidelines when designing a system that uses asynchronous host accesses.

- A host can only access IOP register space on the ADSP-21161 processor.
- The ADSP-21161 processor now uses 9 address lines to access the IOP registers.
- The ADSP-21161 processor does not support the Instruction Word Transfer (`IWT`) function from previous SHARC DSPs. 48-bit instructions can be transferred by configuring the host packing mode to one of the 48-bit internal transfer modes.

Host accesses to non-existent IOP register addresses are not supported. These accesses result in a host bus grant ($\overline{\text{HBG}}$) hang. Therefore, ensure that host accesses generate valid IOP register addresses.

When using asynchronous transfers and direct access to IOP register space, only the lower 9 bits, ADDR8-0, need be supplied by the host. The upper address bits can be configured as Table 7-6.

Asynchronous write operations are latched at the I/O pads in a four-deep FIFO buffer; this buffer is called the slave write FIFO and appears in Figure 6-5 on page 6-23. This buffering allows previously written words to be re-synchronized while a new word is being written. For maximum host throughput and low and high pulse widths for $\overline{\text{WR}}$ and $\overline{\text{RD}}$, refer to the *ADSP-21161N DSP Microcomputer Data Sheet*.

A host may write to several ADSP-21161s simultaneously (a broadcast write) by asserting each of their $\overline{\text{CS}}$ pins. Each processor accepts the write as if it were the only device being addressed. Because the REDY output is wire-ORed (if configured as an open-drain output), REDY only appears asserted when all selected DSPs are ready, unless REDY is actively pulled up. ACK is not active when $\overline{\text{CS}}$ is asserted.

To eliminate the need for a host to drive the multiprocessor address lines (ADDR20-17) in systems with only one processor (ID2-0=000), the processor with ID2-0=000 does not recognize synchronous accesses to these addresses. The host must drive these address lines with 0000 or one of the ADDR23-21 address pins must be driven high to select an address in external memory if the processor's ID2-0 is anything other than 000. To account for buffer delays when sampling the REDY signal, systems must make sure that REDY is properly re-synchronized by the host.

Host Transfer Timing

When a processor's $\overline{\mathrm{CS}}$ chip select is asserted (low), the selected processor deasserts the REDY signal. Refer to the *ADSP-21161N DSP Microcomputer Data Sheet* for exact timing specifications.

As shown in Figure 7-18, the processor deasserts REDY in response to $\overline{\mathrm{CS}}$. The host can assert $\overline{\mathrm{CS}}$ before or after $\overline{\mathrm{HBR}}$ is asserted. When $\overline{\mathrm{HBG}}$ is not asserted, this timing is determined by the tTRDYHG switching characteristic specified in the "Multiprocessor Bus Request and Host Bus Request" timing data in the *ADSP-21161N DSP Microcomputer Data Sheet*.

REDY is asserted prior to $\overline{\mathrm{RD}}$ or $\overline{\mathrm{WR}}$ being asserted and becomes deasserted only if the processor is not ready for the read or write to complete—the only exception is when $\overline{\mathrm{CS}}$ is first asserted. The REDY pin is an open-drain output to facilitate interfacing to common buses. It can be changed to an active-drive output by setting the ADREDY bit in the SYSCON register.

Figure 7-18 shows the timing of a host write cycle, including details of data packing and unpacking. This timing applies to the example host interface hardware shown in Figure 7-25 on page 7-81 and has the following sequence:

1. The host asserts the address. $\overline{\mathrm{HBR}}$ and $\overline{\mathrm{CS}}$ are decoded from the host bus interface address comparator and do not need to be supplied directly by the host. The selected processor deasserts REDY immediately.

2. The host asserts $\overline{\mathrm{WR}}$ and drives data according to the timing requirements specified in the *ADSP-21161N DSP Microcomputer Data Sheet*.

3. The selected processor asserts REDY when it is ready to accept the data. This transition occurs after the current bus master has completed its current transfer and has asserted $\overline{\mathrm{HBG}}$. $\overline{\mathrm{HBG}}$ enables the host interface buffers to drive onto the processor bus.

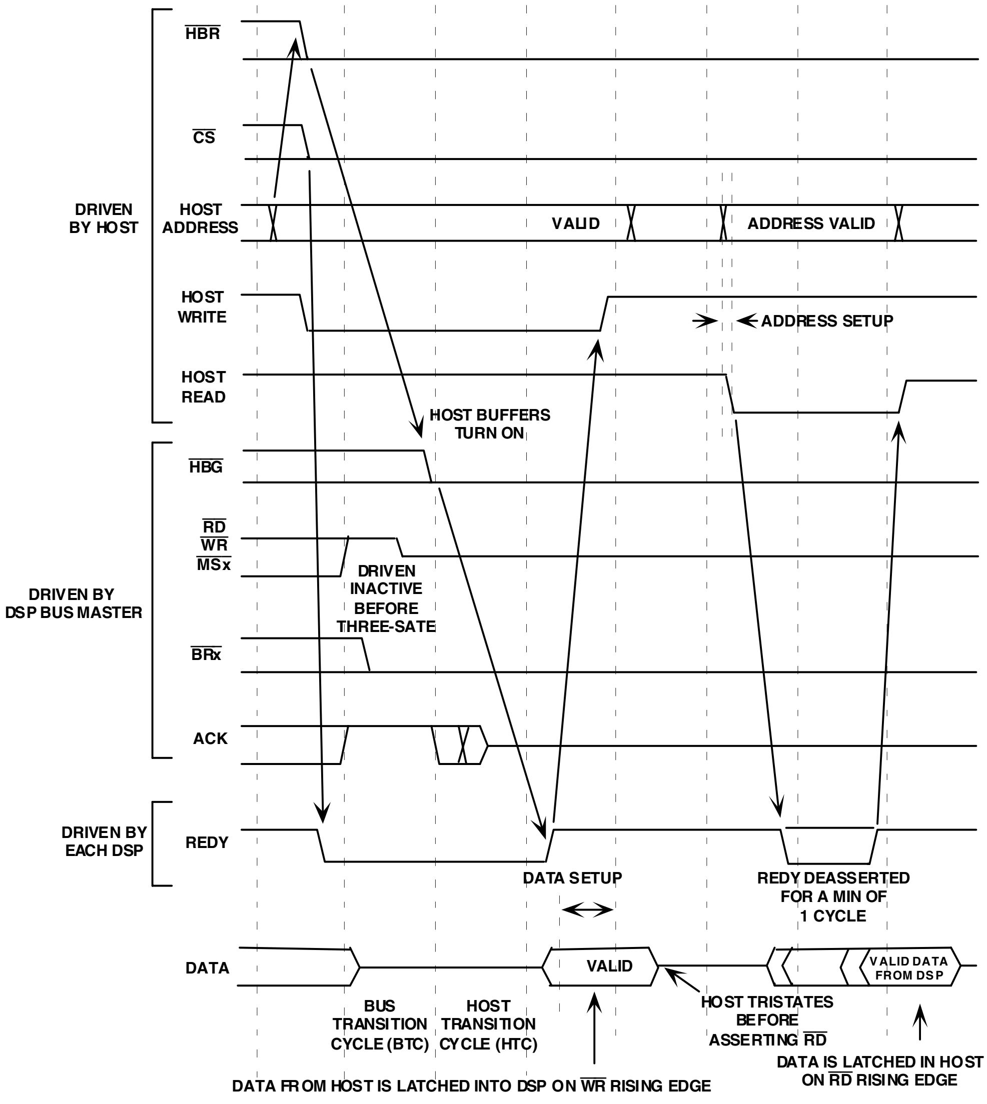

HBR
CS
DRIVEN BY HOST
HOST ADDRESS
VALID
ADDRESS VALID
HOST WRITE
ADDRESS SETUP
HOST READ
HOST BUFFERS TURN ON
HBG
RD
WR
MSx
DRIVEN INACTIVE BEFORE THREE-SATE
DRIVEN BY DSP BUS MASTER
BRx
ACK
DRIVEN BY EACH DSP
REDY
DATA SETUP
REDY DEASSERTED FOR A MIN OF 1 CYCLE
DATA
VALID
VALID DATA FROM DSP
BUS TRANSITION CYCLE (BTC)
HOST TRANSITION CYCLE (HTC)
HOST TRISTATES BEFORE ASSERTING RD
DATA IS LATCHED IN HOST ON RD RISING EDGE
DATA FROM HOST IS LATCHED INTO DSP ON WR RISING EDGE

Figure 7-18. Example Timing for Host Read and Write Cycles

4. The host deasserts $\overline{\mathtt{WR}}$ when REDY is high and stops driving data.

5. The selected processor latches data on the rising edge of $\overline{\mathtt{WR}}$.

After the first word, the write sequence is:

1. The host asserts $\overline{\mathtt{WR}}$ and drives data according to the timing requirements specified in the *ADSP-21161N DSP Microcomputer Data Sheet*.

2. The processor deasserts REDY if it is not ready to accept data.

3. The host deasserts $\overline{\mathtt{WR}}$ when REDY is high and stops driving data.

4. The selected processor latches data on the rising edge of $\overline{\mathtt{WR}}$.

More than one processor may have its $\overline{\mathtt{CS}}$ pin asserted at any one time during a write, but not during a read because of bus conflicts.

Figure 7-18 also shows the timing of a host read cycle. This timing applies to the bus interface hardware in Figure 7-25 on page 7-81 and has the following sequence:

1. The host asserts the address. $\overline{\mathtt{HBR}}$ and the appropriate $\overline{\mathtt{CS}}$ line are decoded by the host bus interface address comparator. The selected processor deasserts REDY immediately and asserts $\overline{\mathtt{HBG}}$.

2. The host asserts $\overline{\mathtt{RD}}$.

3. The selected processor drives data onto the bus and asserts REDY when the data is available.

4. The host latches the data and deasserts $\overline{\mathtt{RD}}$.

After the first word, the read sequence is:

1. The host asserts $\overline{\texttt{RD}}$.
2. The selected processor deasserts $\overline{\texttt{REDY}}$ then asserts `REDY`, driving data when it becomes available.
3. The host deasserts $\overline{\texttt{RD}}$ when `REDY` is high and latches the data.

Host Interface Deadlock Resolution With SBTS

In host systems, the host may need to recover the processor from a slave deadlock condition. When a host processor uses $\overline{\texttt{SBTS}}$ and $\overline{\texttt{HBR}}$ for deadlock resolution, $\overline{\texttt{SBTS}}$ operates differently than when the host uses only $\overline{\texttt{SBTS}}$.

By asserting both $\overline{\texttt{SBTS}}$ and $\overline{\texttt{HBR}}$, the host places the ADSP-21161 in slave mode. `ACK`, $\overline{\texttt{HBG}}$, `REDY`, and the data bus may all be active in slave mode. If the ADSP-21161 was performing an external access (which did not complete) in the same cycle that $\overline{\texttt{SBTS}}$ and $\overline{\texttt{HBR}}$ were asserted, the access is suspended until $\overline{\texttt{SBTS}}$ and $\overline{\texttt{HBR}}$ are both deasserted again.

As with previous SHARCs, this functionality—using $\overline{\texttt{SBTS}}$ and $\overline{\texttt{HBR}}$ together—can be used for host/processor deadlock resolution. If $\overline{\texttt{SBTS}}$ and $\overline{\texttt{HBR}}$ are asserted while bus lock is set, the processor three-states its bus signals, but does not go into slave mode. For more information, see "Deadlock Resolution" on page 7-82.

If $\overline{\texttt{SBTS}}$ and $\overline{\texttt{HBR}}$ are asserted while an external DMA access is occurring, $\overline{\texttt{HBG}}$ is not asserted until the access is completed.

The processor also supports burst transfers, which can be truncated by assertion of $\overline{\texttt{HBR}}$ and $\overline{\texttt{SBTS}}$. If the DMA transaction was a burst transfer, when the host relinquishes control of the local bus, the processor resumes the burst transfer, starting at the address of the last operation that did not complete.

Slave Reads and Writes

The host can directly access the I/O processor registers of a processor by reading or writing the appropriate address.

These accesses are invisible to the slave processor's core. They do not degrade internal memory or internal bus performance. This capability is important, because it lets the processor core continue program execution uninterrupted.

The host can directly read or write the I/O processor registers to control and configure the processor or to set up DMA transfers for indirect read/write access to internal memory.

IOP Shadow Registers

To ease host and multiprocessor system operations, the I/O processor registers include registers that shadow or mirror some processor core system registers, including the program counter (PC), and MODE2_SHDW registers. These registers facilitate system start up and debug, by letting the host (or another processor in an multiprocessor system) interrogate these processor core registers. These shadow registers are read only and lag the value of the registers they shadow by one internal core clock. For more information, see "PC Shadow Register (PC_SHDW)" on page A-77 and "MODE2 Shadow Register (MODE2_SHDW)" on page A-78.

ⓘ The silicon revision field of the MODE2 shadow register MODE2_SHDW is now used for differentiating between silicon revisions. These corresponding bits in the MODE2 (foreground) register are now reserved. The application program must read the MODE2_SHDW register bits [31:25] to identify the silicon revision. MODE2_SHDW is a memory-mapped IOP register whose address is 0x11.

Instruction Transfers

For 8-, 16- or 32-bit host interfaces, the ADSP-21161 can pack and unpack 48-bit instructions or 40-bit extended precision normal word data based on the host packing mode selected with the `HBW` bits in the `SYSCON` register.

Slave Write Latency

The processor handles asynchronous (from a host) and synchronous (from another processor) writes differently. This difference influences the latency for the writes.

When a bus slave receives data from an asynchronous write, the processor latches the data and address in a four-level FIFO buffer. For synchronous writes, this buffer is two levels deep. This buffer is called the slave write FIFO and appears in Figure 6-5 on page 6-23. In the following cycle, the slave write FIFO attempts to complete the write internally. This buffering lets the host (or bus master) perform writes at the full clock rate.

ⓘ The slave's core cannot explicitly read the slave write FIFO. Also, the processor cannot determine the slave write FIFO's status.

Writes to the I/O processor registers from the slave write FIFO usually occur in the following one or two cycles or when any current DMA transfer is completed. The write takes more than two cycles only if a direct write in the previous cycle was held off by a full buffer.

If the slave write FIFO is full when a write is attempted, the processor deasserts `ACK` (or `REDY`) until the FIFO is not full. Unless higher priority on-chip DMA transfers are occurring, the slave write FIFO usually empties out within one cycle, creating a one-cycle write latency.

Slave reads are held off when there is data in the slave write FIFO—this prevents false data reads and out-of-sequence operations.

Slave Reads

When a read of an ADSP-21161 occurs, the address is latched on-chip by the I/O processor and REDY is deasserted asynchronously. When the data is available, the I/O processor drives the data and asserts REDY.

I/O processor register reads have a maximum throughput of one access per every three CLKIN cycles. As a slave, the processor supports burst read accesses, which improve throughput for I/O processor register reads of EPBx FIFOs only. Maximum throughput for synchronous burst direct read accesses is summarized in Table 7-7. For hosts, the processor does not support the synchronous burst protocol.

Table 7-7. Direct Read Latencies for a 1:2 Clock Ratio

Access Type	Latency (CLKIN cycles)
Single Read of I/O processor register	3
Burst Read of I/O processor registers (EPBx only)	3-2-2-2

Broadcast Writes

Broadcast writes allow simultaneous transmission of data to all of the ADSP-21161 processors in a multiprocessing system. The host processor can perform broadcast writes to the same I/O processor register on all of the slaves. Broadcast writes can be used to implement reflective semaphores in a multiprocessing system.

The host processor must assert the $\overline{\text{CS}}$ input of all processors in the system and the address of the appropriate memory mapped I/O processor register for a broadcast write.

Unlike previous SHARCs, the ADSP-21161 processor does not include a broadcast write memory space into its address space and therefore processor to processor broadcast writes are not supported.

Data Transfers Through the EPBx Buffers

The host processor can transfer data to and from the ADSP-21161 through the external port FIFO buffers, EPB0, EPB1, EPB2, and EPB3. Each of these buffers, which are part of the I/O processor register set, is an eight-location FIFO, 64-bit wide (or sixteen-location, 32-bit wide). These buffers support single-word transfers, DMA transfers, and sequential burst accesses. DMA transfers are handled internally by the I/O processor, but single-word transfers must be handled by the processor core.

The processor supports synchronous burst read transfers (32-bit only) from the EPBx FIFOs as a slave. Burst write transfers are not supported.

To perform a burst read transfer from an EPBx buffer, the master issues a starting burst address pointing to one of the EPBx buffer addresses in I/O processor control register space. The slave does not increment an EPBx burst read address, and the master limits the burst transfer length to the modulo4 address boundary restriction.

For information on external port transfers, see "External Port Channel Transfer Modes" on page 6-46. For information on external port handshaking, see "External Port Channel Handshake Modes" on page 6-47.

ⓘ To support debugging buffer transfers, the processor has a Buffer Hang Disable (BHD) bit. When set (=1), this bit prevents the processor core from detecting a buffer-related stall condition, permitting debugging of this type of stall condition. For more information, see the BHD discussion on page 6-43.

DMA Transfers

The host processor can also set up DMA transfers to and from the ADSP-21161. After the host gets control of the processor, the host can access the on-chip DMA control and parameter registers to set up an external port DMA operation. DMA is the most efficient way to transfer blocks of data. For DMA programming examples, see "External Port DMA Example" on page 6-77 and "External Port Chained DMA Example" on page 6-79.

- **DMA Transfers to Internal Memory.** The host can set up external port DMA channels to transfer data to and from internal memory.
- **DMA Transfers to External Memory.** The host can set up an external port DMA channel to transfer data directly to external memory using the DMA request and grant lines ($\overline{\text{DMARx}}$, $\overline{\text{DMAGx}}$).

For more information, see "Setting Up External Port DMA" on page 6-68.

ⓘ The host may also use the $\overline{\text{DMARx}}$/$\overline{\text{DMAGx}}$ handshake signals for a DMA transfer as a bus slave, but may not use DMA as a bus master while $\overline{\text{HBR}}$ retains control of the bus.

Host Data Packing

The host interface uses the same data packing features as the I/O processor uses. The "8- to 32-Bit Data Packing" on page 7-66 and "48-Bit Instruction Packing" on page 7-74 sections describe timing for these data packing operations.

ⓘ For transfers to or from the EPBx data buffers, the packing mode is determined by the setting of the HBW bits of the SYSCON register **AND** the PMODE bits in the DMACx control register for each external port buffer.

For host accesses, to pack and unpack individual data words, you must set **both** the PMODE bits in the appropriate DMACx control register and the HBW bits in the SYSCON register. Table 7-8 shows the packing mode bit settings for access to IOP, link port and external port buffers.

Table 7-8. Packing Mode Combinations

PMODE	HBW 8/16/32	Host Packing Mode (External:Internal)		
		IOP Buffers Internal Packing Fixed to 32-bit	**Link Ports Buffers** Internal Packing Fixed to 48-bit	**External Port Buffers** Uses PMODE, INT32 & DTYPE (1=48/40, 0=32/64)
000	–	Reserved		
001	01 (16-bit)	16 : 32	16 : 48	16 : 32/64
010	01 (16-bit)	16 : 32	16 : 48	16 : 48-bit
011	00 (32-bit)	32 : 32	32 : 48	32 : 48-bit
100	00 (32-bit)	32 : 32	32 : 48	32 : 32/64
101	10 (8-bit)	8 : 32	8 : 48	8 : 48
110	10 (8-bit)	8 : 32	8 : 48	8 : 32/64
111	–	Reserved		

The ADSP-21161 provides a glueless interface to 8-, 16-, and 32-bit hosts. Three differences between the ADSP-21161 and the ADSP-21160 are:

- Connection of 8-bit hosts (in addition to 16- or 32-bit hosts) is supported.
- There is limited direct access to IOP register space. A host processor and other ADSP-21161s in a multiprocessing configuration can only directly access the memory mapped IOP registers of an ADSP-21161. A host can only use asynchronous access to ADSP-21161 registers (by using $\overline{\text{CS}}$ of the processor). The lower nine bits of the 24-bit address bus are decoded to select an IOP register for any access into the ADSP-21161's internal memory.

- Synchronous broadcast write is not supported by the ADSP-21161 because there is no broadcast memory space. However, the host can simultaneously write to the same address on all the processors asynchronously by asserting $\overline{\text{CS}}$ for each processor simultaneously during a write without any multiprocessor memory offset.

The host data bus is connected to the ADSP-21161 data bus in a LSB-alignment to the default 32-bit active data bus DATA47-16. For example, data pin 0 (D0) of host data bus connects to DATA16 of ADSP-21161 data bus, data pin 1 (D1) of the host data bus connects to DATA17 of the ADSP-21161 data bus, and so on.

Depending on the register access, the processor packs/unpacks data as 32 bits, 48 bits, or up to 64 bits. A host can indirectly transfer data (via DMA) to and from internal memory by writing or reading to/from EPBx. To support this, several packing options are available. The newly defined Host Bus Width (HBW) bits 5 and 4 in the SYSCON register control the host data packing. They are described in Table 7-9 on page 7-65. Host Packing Status (HPS) bits 24-22 have also been redefined in SYSTAT. They are described in Table A-21 on page A-69.

Packing Mode Variations For Host Accesses

The host interface (using $\overline{\text{HBR}}$, $\overline{\text{HBG}}$, $\overline{\text{CS}}$) uses data packing logic to allow the packing of 8-, 16-, and 32-bit external bus words into 32-, 48-, and 64-bit internal words. The packing logic is fully reversible; packing and unpacking of data is performed for both directions of data transfer to external data.

For 32-bit, 16-bit, and 8-bit host processors accessing IOP register space, the processor can pack and unpack data to or from internal memory, independent of the setting of the PMODE bits in the DMACx register, to either 32-bit, 48-bit, or up to 64-bits internal packing depending on the type of host access. Although the packing mode for host access is configurable, it can sometimes revert to fixed packing modes depending on the IOP register accessed. In most cases, when a host accesses IOP control/status registers, the processor defaults to internal data packing and unpacking to a 32-bit access (independent of the setting of the PMODE bits in the DMACx register). LBUFx buffer access is limited to 48-bits internal packing, ignoring the PMODE bits in DMACx. EPBx buffer access always depends on the PMODE bits, DTYPE and INT32 bits in DMACx.

The three host access cases are described in the following sections:

- "IOP Register Host Accesses" on page 7-62
- "LINK Port Buffer Access" on page 7-63
- "EPBx Buffer Accesses" on page 7-64

IOP Register Host Accesses

For accesses to all IOP registers except EPBx and LBUFx, the host data is fixed to packed or unpacked to/from 32-bit internal data word. In most cases, when accessing an IOP control or status register, or serial port and SPI data buffers (TXn/RXn, SPIRX/SPITX), the PMODE bits in the DMACx register are ignored. A fixed packing mode of 8-, 16- or 32-bit external to 32-bit internal is selected. This is because all IOP registers except LBUFx and EPBx are 32 bits wide.

Ensure that host accesses generate valid IOP register addresses. Host accesses to non-existent IOP register addresses are not supported, and can result in host bus grant ($\overline{\text{HBG}}$) hang.

Host access of IOP control/status registers and SPORT/SPI data buffers (*except* EPBx and LBUFx) will pack or unpack to 32 bits internally, ignoring the value of PMODE in DMACx. The HBW bits in the SYSCON register are used as a reference to set the external packing mode.

For example, when interfacing the ADSP-21161 to an 8-bit microcontroller, the HBW bits are set in the SYSCON register to specify a host bus width of 8 bits. This results in an 8-bit external to 32-bit internal fixed data packing mode to an IOP register. Table 7-8 on page 7-60 shows the packing mode combinations.

LINK Port Buffer Access

The link buffers LBUF0 and LBUF1 can also be accessed by an external host processor, using direct reads and writes to IOP register space. However, there are differences in how data is accessed with the link buffers compared to other IOP control/status registers. When the host processor reads or writes to these buffers, the external packing data access width is also determined by the host bus width bits in the SYSCON register while the internal packing mode is restricted to 48 bits.

Hosts accesses to the link port buffers pack or unpack to 48 bits internally, ignoring the value of PMODE in DMACx. The HBW bits in the SYSCON register are used to set the external packing mode.

In the case where a host processor reads or writes to the LBUF0 and LBUF1 link buffers, the PMODE bits in the DMACx external port DMA control register are ignored and are fixed to a special 48-bit internal packing mode. This fixed 48-bit internal packing mode is required because the ADSP-21161 link port buffers transmit and receive 48-bit words. Depending on the HBW bits in SYSCON, the appropriate external to 48-bit internal memory packing mode are selected. The available bit settings are shown in Table 7-8 on page 7-60.

It may be desirable in some applications for a host processor to transfer instruction opcodes to another SHARC indirectly via the directly connected SHARC's link port by reading or writing the opcode data to or from the LBUF0 and LBUF1 link buffers through the external port. For example, with a 16-bit host, the packing mode internally defaults to 48-bit packed transfers such that the packing mode is 16-bit external to 48-bit internal packed data transfers.

EPBx Buffer Accesses

The external port buffers, EPB0, EPB1, EPB2, and EPB3 can also be accessed by an external host processor, using direct reads and writes to IOP register space. There are differences in how data is accessed with the EPBx buffers as compared with other IOP control/status registers. When the host processor reads or writes to external port buffers, the packing mode indicated by the PMODE bits in the corresponding DMACx register are selected.

Host accesses to the external port buffers pack or unpack according to the packing mode specified with the PMODE bits in DMACx.

Depending on the HBW bits in SYSCON and PMODE in DMACx, the appropriate packing mode are selected as shown in Table 7-8 on page 7-60.

There is no direct write pending bit in SYSTAT (as in the ADSP-21160) since the ADSP-21161 does not have a direct write FIFO. However, the ADSP-21161 processor has two newly defined bits in SYSTAT for checking the status of the slave write FIFO.

The following bits in the SYSTAT register affect host access:

- **Synchronous Slave Write FIFO Data Pending.** SYSTAT Bit 20 (SSWPD). Since a host cannot be synchronous, this bit is set for synchronous access by another ADSP-21161. The bit is set (=1) when synchronous slave IOP register write is pending. The bit is cleared (=0) when the direct write is complete.

- **Slave Write FIFO Data Pending.** SYSTAT Bit 21 (SWPD). This status bit is set for any host or SHARC write access to an IOP register. If a host processor attempts to write data through the asynchronous protocol, this status bit is set. The bit is set (=1) when a slave (asynchronous or synchronous) write to an IOP register is pending. The bit is cleared (=0) when there is no slave write pending. The processor clears SWPD when the direct write is complete.

- **Host Packing Order**. SYSCON Bit 7 (HMSWF).This bit determines whether the I/O processor packs the most significant or least significant word first for 8-bit and 16-bit hosts. For 32- to 32/64 and 32- to 48-bit packing, the processor ignores the HMSWF bit in the SYSCON register and the MSWF bit in the DMACx register.

Host packing examples are shown below for host direct read/write access to IOP control/status registers, TXn/RXn, SPIRX/SPITX and LBUFx data buffers. The default internal packing is 32-bit for host accesses to IOP control/status registers and 48-bit for host accesses to LBUFx, ignoring PMODE bits in DMACx. If the HMSWF bit is set (=1), the packing and unpacking is most significant word first. If the HMSWF bit is cleared (=0), the packing and unpacking is least significant word first.

Table 7-9. Packing sequence for 32-bit IOP Register Data

Transfer	Data Bus Pins 23-16 (8-bit bus, LSW first)	Data Bus Pins 31-16 (16-bit bus, MSW first)
First	Word 1; bits 7-0	Word 1; bits 31-16
Second	Word 1; bits 15-8	Word 1; bits 15-0
Third	Word 1; bits 23-16	
Fourth	Word 1; bits 31-24	

Table 7-10. Packing Sequence for Accessing 48-bit LBUFx Data

Transfer	Data Bus Pins 31-16 (16-bit bus, MSW first)	Data Bus Pins 23-16 (8-bit bus, MSW first)
First	LBUFx; bits 47-32	LBUFx; bits 47-40
Second	LBUFx; bits 31-16	LBUFx; bits 39-32
Third	LBUFx; bits 15-0	LBUFx; bits 31-24

Table 7-10. Packing Sequence for Accessing 48-bit LBUFx Data (Cont'd)

Transfer	Data Bus Pins 31-16 (16-bit bus, MSW first)	Data Bus Pins 23-16 (8-bit bus, MSW first)
Fourth		LBUFx; bits 23-16
Fifth		LBUFx; bits 15-8
Sixth		LBUFx; bits 7-0

Table 7-11. Packing Sequence for Accessing 48-bit LBUFx Data From a 32-bit bus (MSW First)

Transfer	Data Bus Pins 47-32	Data Bus Pins 31-16
First	LBUFx 1; bits 47-32	LBUFx 1; bits 31-16
Second	LBUFx 2; bits 15-0	LBUFx 1; bits 15-0
Third	LBUFx 2; bits 47-32	LBUFx 2; bits 31-16

To write a single 48-bit word or an odd number of 48-bit words to LBUFx, write a dummy access to completely fill the packing buffer.

8- to 32-Bit Data Packing

The processor latches incoming data on pins DATA23-16 for 8- to 32-bit packing on an 8-bit host bus. Similarly, the processor drives outgoing data on DATA23-16 with the other lines equal to zeroes. The sequence of events for 32-bit packing and unpacking for writes and reads are shown in Figure 7-19 on page 7-71.

When a host reads a 32-bit word with 8-bit unpacking using the typical bus interface hardware shown in Figure 7-25 on page 7-81, the following sequence of events occurs:

- The host initiates a read cycle by driving an address, asserting $\overline{\text{CS}}$, and asserting $\overline{\text{RD}}$ (low).

- The selected processor deasserts REDY, latches the address, and performs an internal read to get the data.
- When the processor has the data, it asserts REDY and drives the first 8-bit word.
- The host latches the data and deasserts $\overline{\text{RD}}$ (high).
- The host initiates another read access, driving the address of the data to be accessed then asserting $\overline{\text{RD}}$.
- The processor transmits the second 8-bit word.
- The host initiates another read access, driving the address of the data to be accessed then asserting $\overline{\text{RD}}$.
- The processor transmits the third 8-bit word.
- The host initiates another read access, driving the address of the data to be accessed then asserting $\overline{\text{RD}}$.
- The processor transmits the final 8-bit word. 8- to 32-bit packing is complete.

When a host writes a 32-bit word with 8-bit packing using the typical bus interface hardware shown in Figure 7-25 on page 7-81, the following sequence of events occurs:

- The host initiates a write cycle by driving the write address, asserting $\overline{\text{CS}}$, and asserting $\overline{\text{WR}}$ (low).
- The processor asserts REDY when it is ready to accept data.
- The host drives the address and the first 8-bit word and deasserts $\overline{\text{WR}}$ (high).
- The processor latches the first 8-bit word.

- The host drives the address and initiates another write cycle for the second 8-bit word by asserting $\overline{\text{WR}}$.
- The processor latches the second 8-bit word.
- The host drives the address and initiates another write cycle for the third 8-bit word by asserting $\overline{\text{WR}}$.
- The processor latches the third 8-bit word.
- The host drives the address and initiates another write cycle for the fourth 8-bit word by asserting $\overline{\text{WR}}$.
- When the processor has accepted the fourth word, it performs an internal write to its memory-mapped I/O processor register. If the processor's internal write has not completed by the time another host access occurs, the processor holds off that access with REDY.

The packing sequence for downloading 32-bit data from a 8-bit host bus takes four cycles for every word, as illustrated in as shown in Table 7-12. The endian format of the transfers is controlled by the HMSWF bit in the SYSCON register. If HMSWF=0, the least significant 8-bit word is packed first. If HMSWF=1, the most significant 8-bit word is packed first.

Table 7-12. 8- to 32-Bit Word Packing, HMSWF=1
(Host Bus <-> ADSP-21161)

Transfer	Data Bus Pins 23-16
First transfer	Word1, bits 31-24
Second transfer	Word1, bits 23-16
Third transfer	Word1, bits 15-8
Fourth transfer	Word1, bits 7-0

16- to 32-Bit Packing

For a 16-bit host bus, the processor latches incoming data on pins `DATA31-16`. Similarly, the processor drives outgoing data on `DATA31-16` with the other lines equal to zeroes. The sequence of events for 32-bit packing and unpacking is different for writes and reads.

When a host reads a 32-bit word with 16-bit unpacking using the bus interface hardware shown in Figure 7-25 on page 7-81, the following sequence of events occurs as illustrated in Figure 7-23 on page 7-73:

- The host initiates a read cycle by driving an address, asserting $\overline{\texttt{CS}}$, and asserting $\overline{\texttt{RD}}$ (low).
- The selected processor deasserts `REDY`, latches the address, and performs an internal read to get the data.
- When the processor has the data, it asserts `REDY` and drives the first 16-bit word.
- The host latches the data and deasserts $\overline{\texttt{RD}}$ (high).
- The host initiates another read access, driving the address of the data to be accessed then asserting $\overline{\texttt{RD}}$.
- The processor transmits the second 16-bit word (16 to 32-bit packing is complete).

When a host writes a 32-bit word with 16-bit packing using typical bus interface hardware shown in Figure 7-25 on page 7-81, the following sequence of events occurs as illustrated in Figure 7-23 on page 7-73:

- The host initiates a write cycle by driving the write address, asserting $\overline{\texttt{CS}}$, and asserting $\overline{\texttt{WR}}$ (low).
- The processor asserts `REDY` when it is ready to accept data.
- The host drives the address and the first 16-bit word and deasserts $\overline{\texttt{WR}}$ (high).

- The processor latches the first 16-bit word.
- The host drives the address and initiates another write cycle for the second 16-bit word by asserting $\overline{\texttt{WR}}$.
- When the processor has accepted the second word, it performs an internal write to its memory-mapped I/O processor register. If the processor's internal write has not completed by the time another host access occurs and the 4 deep asynchronous slave FIFO is full, the processor holds off that access with `REDY`.

The packing sequence for downloading or uploading instructions over an 16-bit host bus takes two cycles for every 32-bit word, as shown in Table 7-13. The endian format of the transfers is controlled by the `HMSWF` bit in the `SYSCON` register. If `HMSWF`=0, the least significant 16-bit word is packed first. If `HMSWF`=1, the most significant 16-bit word is packed first.

Table 7-13. 16- to 32-Bit Word Packing, HMSWF=1 (Host Bus <-> ADSP-21161)

Transfer	Data Bus Pins 31-16
First transfer	Word1, bits 31-16
Second transfer	Word1, bits 15-0

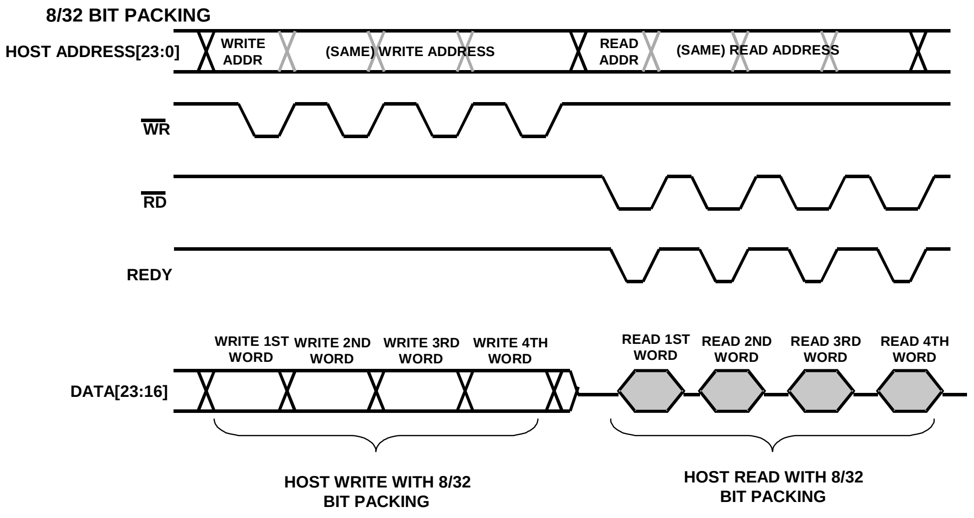


Figure 7-19. Timing for 8- to 32-Bit Host Data Packing

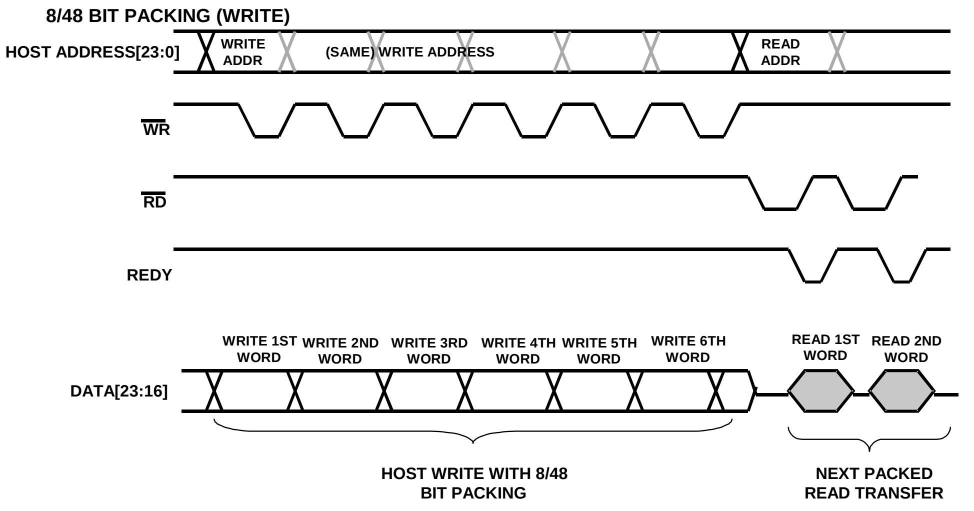


Figure 7-20. Timing for 8- to 48-Bit Host Data Packing (Write)

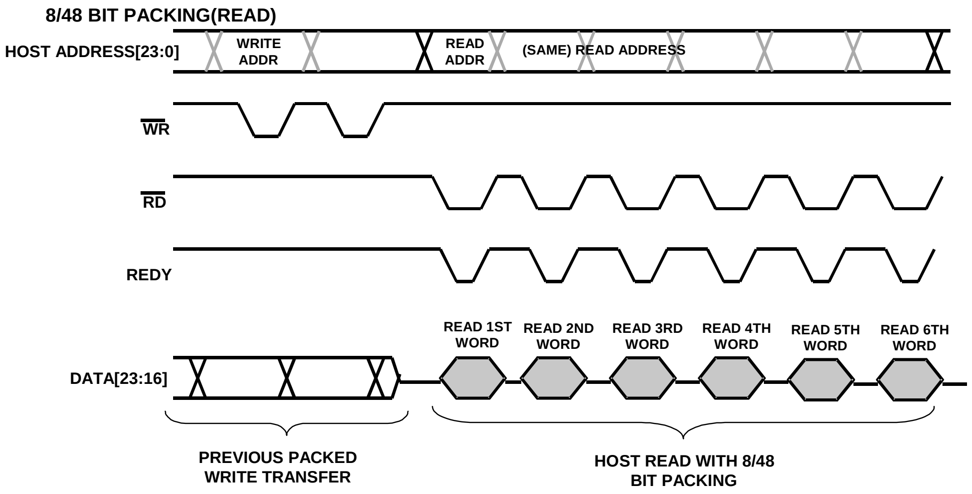


Figure 7-21. Timing for 8- to 48-Bit Host Data Packing (Read)

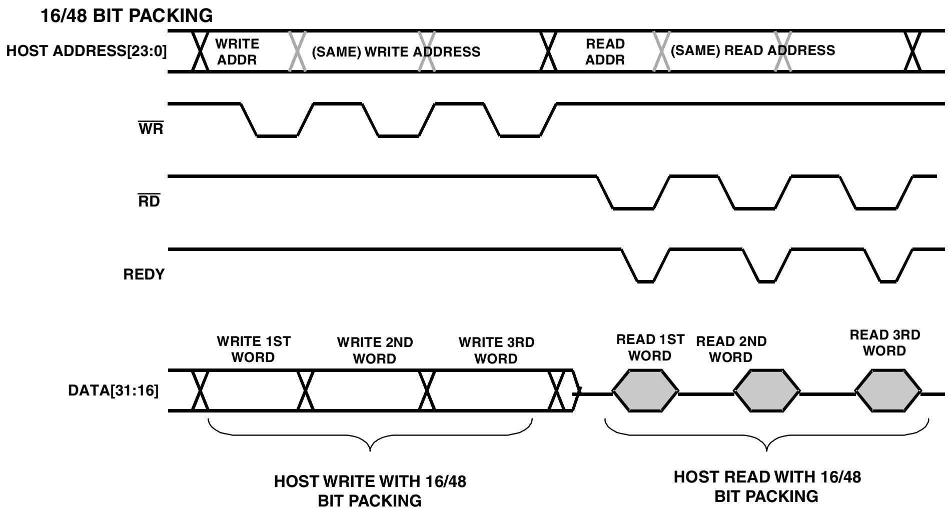


Figure 7-22. Timing for 16- to 48-Bit Host Data Packing

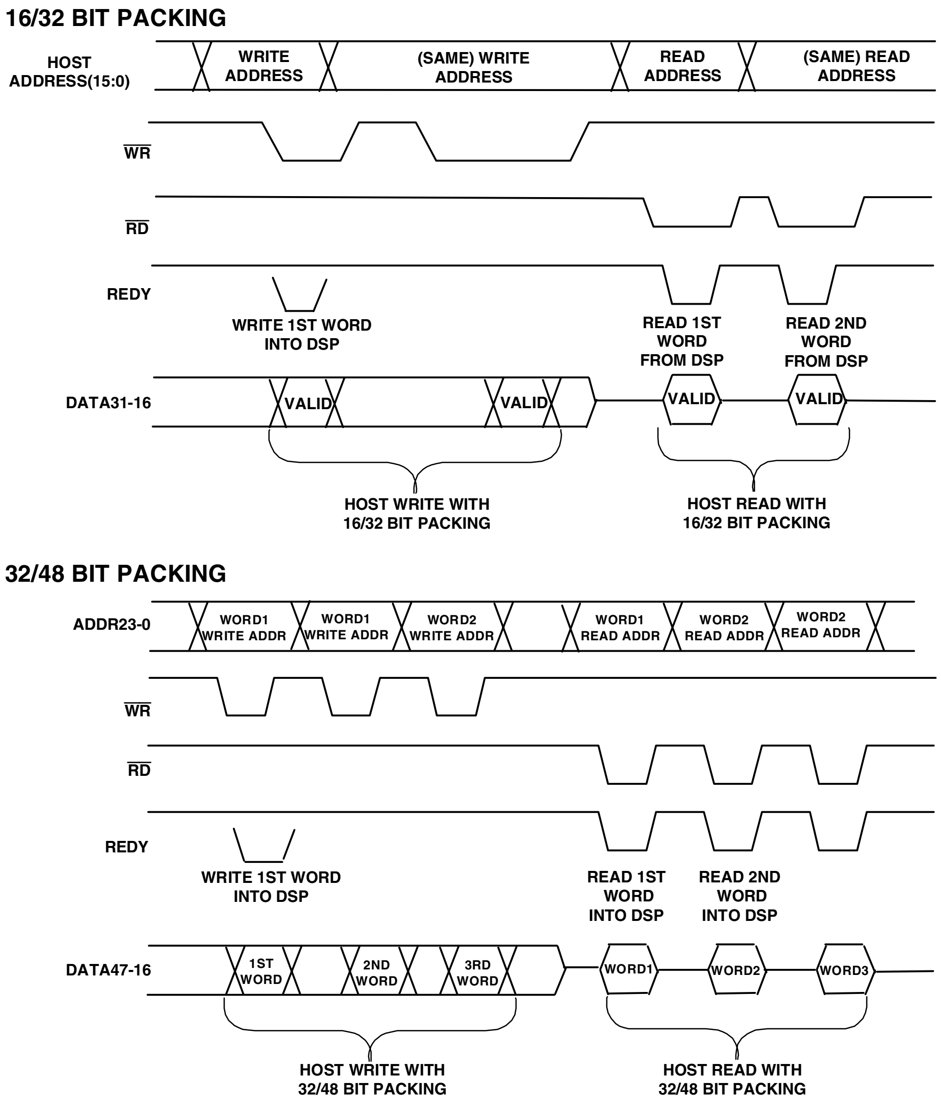


Figure 7-23. Timing for Host Data Packing

If the processor is waiting for another 8- or 16-bit word from the host to complete the packed word, the HPS field in the SYSTAT register is non-zero. For more information, see "Host Interface Status" on page 7-76.

ⓘ Because there is only one packing buffer for the host interface, the host must complete each packed transfer before another is begun. For more information, see "External Port Status" on page 6-127.

48-Bit Instruction Packing

The host can also download and upload 48-bit instructions over its 8-, 16-, or 32-bit bus.

32- to 48-Bit Packing

The packing sequence for downloading instructions from a 32-bit host bus (HBW=00) takes 3 cycles for every 2 words, as illustrated in Table 7-14. Data (32-bit) is transferred on data bus lines 47-16 (DATA47-16). If an odd number of instruction words are transferred, the packing buffer must be flushed by a dummy access to remove the unused word.

40-bit extended precision data may be transferred using the 48-bit packing mode. For more information on memory allocation for different word widths, see "Memory Organization and Word Size" on page 5-25.

Table 7-14. 32- to 48-Bit Word Packing (Host Bus <-> ADSP-21161)

Transfer	Data Bus Lines 47-32	Data Bus Lines 31-16
First transfer	Word1, bits 47-32	Word1, bits 31-16
Second transfer	Word2, bits 15-0	Word1, bits 15-0
Third transfer	Word2, bits 47-32	Word2, bits 31-16

ⓘ The HMSWF bit of SYSCON is ignored for 32- to-48-bit packing.

When a host writes a 48-bit word with 32-bit packing using typical bus interface hardware shown in Figure 7-25 on page 7-81, the sequence of events occurs as illustrated in Figure 7-23 on page 7-73.

16- to 48-Bit Packing

The packing sequence for downloading or uploading instructions over a 16-bit host bus takes three cycles for every 48-bit word, as shown in Table 7-15.

Table 7-15. 16- to 48-Bit Word Packing, HMSWF=1 (Host Bus <-> ADSP-21161)

Transfer	Data Bus Pins 31-16
First transfer	Word1, bits 47-32
Second transfer	Word1, bits 31-16
Third transfer	Word1, bits 15-0

When a host writes a 48-bit word with 16-bit packing using typical bus interface hardware shown in Figure 7-25 on page 7-81, the sequence of events occurs as illustrated in Figure 7-22 on page 7-72.

8- to 48-Bit Packing

The packing sequence for downloading or uploading instructions over an 8-bit host bus takes six cycles for every 48-bit word, as shown in Table 7-16. The endian format of the transfers is controlled by the `HMSWF` bit in the `SYSCON` register. If `HMSWF`=0, the least significant word is packed first. If `HMSWF`=1, the most significant word is packed first.

When a host writes a 48-bit word with 8-bit packing using typical bus interface hardware shown in Figure 7-25 on page 7-81, the sequence of events occurs as illustrated in Figure 7-23 on page 7-73.

Table 7-16. 8- to 48-Bit Word Packing, HMSWF=1 (Host Bus <-> ADSP-21161)

Transfer	Data Bus Pins 23-16
First transfer	Word1, bits 47-40
Second transfer	Word1, bits 39-32
Third transfer	Word1, bits 31-24
Fourth transfer	Word1, bits 23-16
Fifth transfer	Word1, bits 15-8
Sixth transfer	Word1, bits 7-0

Host Interface Status

The SYSTAT register provides status information for host and multiprocessor systems. For more information on the SYSTAT register, see Table A-21 on page A-69.

Interprocessor Messages and Vector Interrupts

After getting control of the ADSP-21161, the host processor communicates with it by writing messages to the memory-mapped I/O processor registers. In a multiprocessor system, the host can access the I/O processor registers of every ADSP-21161.

The MSGRx registers are general-purpose registers that can be used for message passing between the host and the ADSP-21161. They are also useful for semaphores and resource sharing between multiple DSPs. The MSGRx and VIRPT registers can be used for message passing in the following ways:

- **Message Passing.** The host can use any of the eight message registers, MSGR0 through MSGR7, to communicate with the processor.

- **Vector Interrupts.** The host can issue a vector interrupt to the processor by writing the address of an interrupt service routine to the VIRPT register. When serviced, this high priority interrupt causes the processor to branch to the service routine at that address.

The MSGRx and VIRPT registers also support shared-bus multiprocessing through the external port. Because these registers may be shared resources within a single processor, conflicts may occur—your system software must prevent this. For further discussion of I/O processor register access conflicts, see "I/O Processor Registers" on page A-47.

Message Passing (MSGRx)

There are three possible software protocols that the host can use for communicating with the processor through the MSGRx message registers: vector-interrupt-driven, register handshake, and register write-back.

For the vector-interrupt-driven method, the host fills predetermined MSGRx registers with data, and triggers a vector interrupt by writing the address of the service routine to VIRPT. The service routine should read the data from the MSGRx registers and then write 0 into VIRPT. This signals the host that the routine is complete. The service routine also could use one of the processor's FLAG11-0 pins to indicate completion.

For the register handshake method, four of the MSGRx registers are designated as follows: a receive register (R), a receive handshake register (RH), a transmit register (T), and a transmit handshake register (TH). To pass data to the ADSP-21161processor, the host would write data into T and then write a 1 into TH. When the ADSP-21161 sees a 1 in TH, it reads the data from T and then writes back a 0 into TH. When the host sees a 0 in TH, it knows that the transfer is complete. A similar sequence of events occurs when the ADSP-21161 passes data to the host through R and RH.

The register write-back method is similar to register handshaking, but uses only the T and R data registers. The host writes data to T. When the ADSP-21161 sees a non-zero value in T, it retrieves it and writes back a 0 to T. A similar sequence occurs when the host is receiving data. This simpler method works well when the data to be passed does not include 0.

Host Vector Interrupts (VIRPT)

Vector interrupts are used for interprocessor commands between the host and a ADSP-21161 or between two ADSP-21161s. When the external processor writes an address to the ADSP-21161's VIRPT register, the write triggers a vector interrupt. For more information, see "Multiprocessing Interrupts" on page 3-49.

To use the ADSP-21161's vector interrupt feature, the host can perform the following sequence of actions:

1. Poll the processor's VIRPT register until the host reads a certain token value (for example, zero).
2. Write the vector interrupt service routine address to VIRPT.
3. When the service routine is finished, the processor writes the token back into VIRPT to indicate that it is finished and that another vector interrupt can be initiated.

System Bus Interfacing

A processor subsystem, consisting of several DSPs with local memory, may be viewed as one of several processors connected together by a system bus. Examples of such systems are the EISA bus, PCI bus, or several processor subsystems. The processors in this kind of system arbitrate for the system bus using an arbitration unit. Each device on the bus that needs to become a bus master must be able to drive a bus request signal and respond to a bus grant signal. The arbitration unit determines which request to grant in any given cycle.

Access to the Processor Bus – Slave Processor

Figure 7-24 shows an example of a interface to a system bus that isolates the local processor bus from the system bus. When the system is not accessing the DSPs, the local bus supports transfers between other local DSPs and local external memory or devices.

When the system needs to access a processor, the system executes a read or write to the address range of the processor subsystem. The external address comparator detects a local access and asserts $\overline{\text{HBR}}$ and one of the appropriate $\overline{\text{CS}}$ lines. The processor holds off the system bus with REDY until the processor is ready to accept the data. The $\overline{\text{HBG}}$ signal enables the system bus buffers. The buffers' direction for data is controlled by the read or write signals. To avoid glitching the $\overline{\text{HBR}}$ line when addresses are changing, the address comparator may be qualified by an enable signal from the system or qualified by the system read or write signals. These methods cause $\overline{\text{HBR}}$ to be deasserted each time system read or write is deasserted or the address is changed. Because these techniques deassert $\overline{\text{HBR}}$ with each access, the overhead of an HTC occurs as part of each access.

Access to the System Bus – Master Processor

Figure 7-25 shows a bidirectional system interface in which the processor subsystem can access the system bus by becoming a bus master. Before beginning the access, the processor first requests permission to become the bus master by generating the System Bus Request signal (SBR). A bus arbitration unit determines when to respond with SBR. Here, each system bus master generates and responds to its own unique pair of signals.

The method a processor uses to arbitrate for the system bus depends on whether the access is from the processor processor core or the I/O processor. For more information, see "Processor Core Access to System Bus" on page 7-82 and "DMA Access to System Bus" on page 7-84.

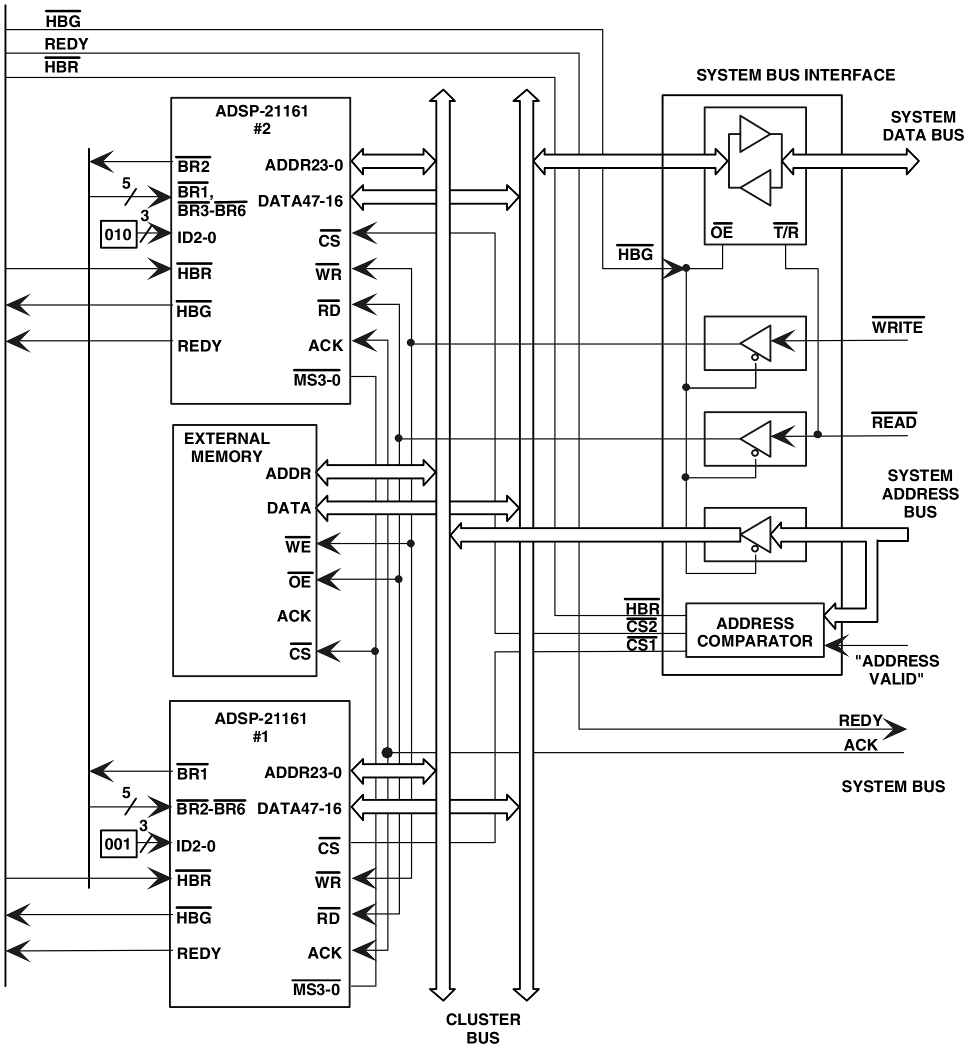


Figure 7-24. Slave System Bus Interface

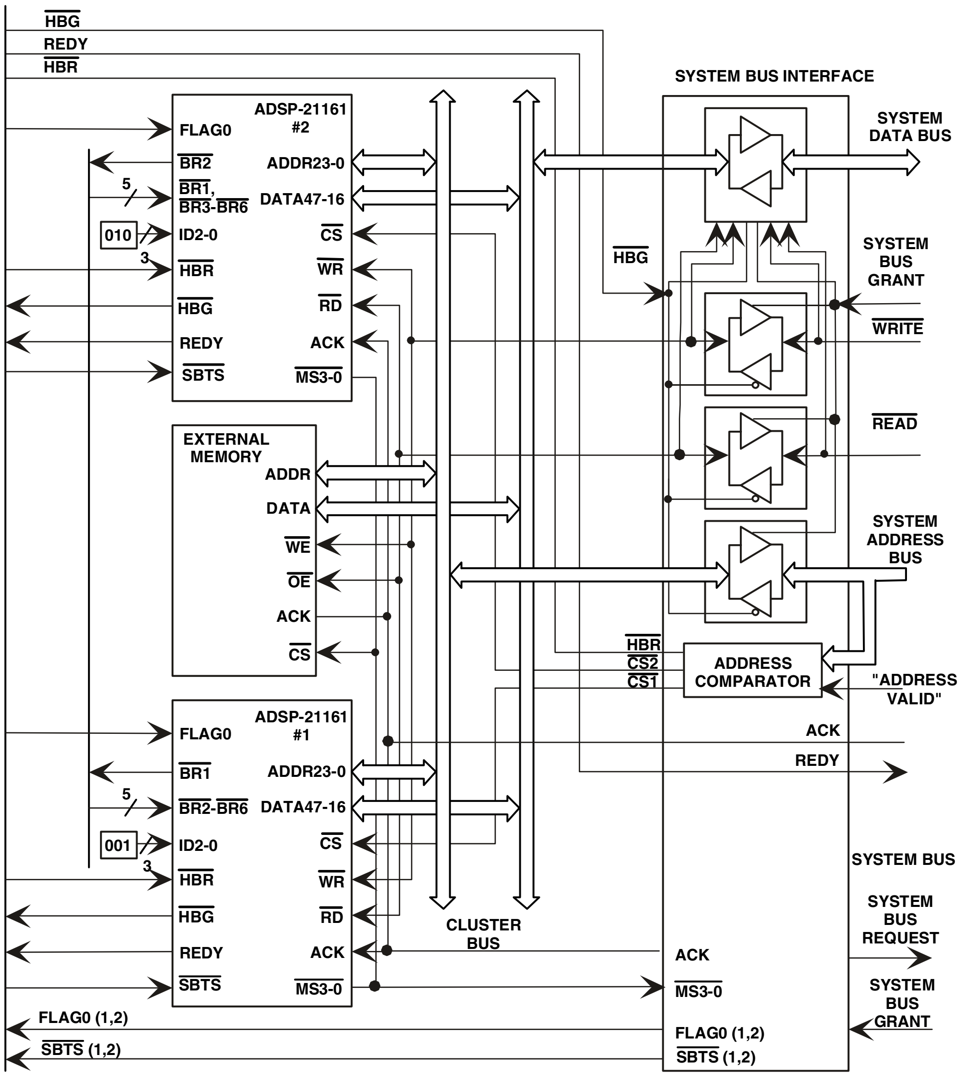


Figure 7-25. Bidirectional System Bus Interface

Processor Core Access to System Bus

The processor core may arbitrate for the system bus by setting a flag and waiting for SBG on another flag. This technique has the benefit of not stalling the local bus while waiting. If SBG is tied to an interrupt pin, the processor can continue processing while waiting.

Another method for the processor access is to attempt the access assuming that the system bus is available. The processor then either waits or aborts the access if the bus is not available. The processor begins the access to the system bus by asserting one of the memory select lines, $\overline{\text{MS}}$3-0. This assertion also asserts SBR. If the system bus is not available (for example, SBG is deasserted), the processor should be held off with ACK. This approach is simple, but stalls the processor and the local bus when the system bus is accessed while it is busy. To overcome this stall, programs can use the Type 10 instruction:

```
IF condition JUMP(addr), ELSE compute, DM(addr)=dreg;
```

This instruction aborts the bus access if the condition (SBG) is not true and causes the program to branch to a try-again-later routine. This method works well if SBG is asserted most of the time. If the Type 10 instruction is not used, a deadlock condition can result if an access is attempted before the bus is granted.

The processor samples FLAG inputs at the CLKIN frequency except when $\overline{\text{CLKDBL}}$ is enabled. When $\overline{\text{CLKDBL}}$ is enabled, the processor samples FLAG inputs at the CLKOUT frequency. FLAG outputs must be held stable for at least one full CLKIN cycle.

Deadlock Resolution

When both the processor subsystem and the system try to access each other's bus in the same cycle, a deadlock may occur in which neither access can complete; ACK stays deasserted.

Normally, the master processor responds to an $\overline{\text{HBR}}$ request by asserting $\overline{\text{HBG}}$ after the completion of the current access. If the processor is accessing the system bus at the same time, $\overline{\text{HBG}}$ is not asserted, because this current access cannot complete—this condition results in a deadlock in which neither access can complete. The deadlock may be broken by asserting the Suspend Bus Three-state ($\overline{\text{SBTS}}$) input for one or more cycles after the deadlock is detected—when the system bus to local bus buffer is requested from both sides.

The combination of $\overline{\text{SBTS}}$ and $\overline{\text{HBR}}$ puts the master processor into slave mode and suspends the processor core's external access. This suspension lets the system access to the local bus proceed, after the processor asserts $\overline{\text{HBG}}$. The combination of $\overline{\text{HBR}}$ and $\overline{\text{SBTS}}$ should only be applied when there is a deadlock caused by a processor access to the system bus. $\overline{\text{SBTS}}$ should not be used when there is a local bus transfer, because the $\overline{\text{WR}}$ signal is asserted twice—once before the $\overline{\text{SBTS}}$ is asserted and once after the access resumes. For processor-to-processor transfers on the local bus, this double assertion violates the slave timing requirements.

The following sequence of actions allows the host processor to suspend an ongoing processor access and gain access to its internal resources, provided that: 1) the access originates from the processor's core, not the DMA controller, 2) a DRAM page miss is not detected for that memory access, and 3) bus lock is not enabled.

1. After $\overline{\text{HBR}}$ is asserted, the host asserts $\overline{\text{SBTS}}$ for one or more cycles. If $\overline{\text{SBTS}}$ is asserted one or more cycles after $\overline{\text{HBR}}$ is recognized, $\overline{\text{HBG}}$ is guaranteed to be asserted in the next cycle. $\overline{\text{SBTS}}$ should be deasserted before $\overline{\text{HBR}}$ is deasserted.

2. The host drives the $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobe to their correct values after $\overline{\text{HBG}}$ is asserted. The host may then perform as many accesses as desired.

3. The host has full control of the bus and may access any of the processors or peripherals on the bus.

4. The host deasserts $\overline{\mathtt{HBR}}$. $\overline{\mathtt{HBG}}$ is deasserted when the internal read buffer is empty.

5. One cycle after the processor deasserts $\overline{\mathtt{HBG}}$, it restarts its suspended access.

DMA Access to System Bus

Using the $\overline{\mathtt{SBTS}}$ and $\overline{\mathtt{HBR}}$ inputs to resolve a system bus deadlock, as described in "Deadlock Resolution" on page 7-82, cannot be used for DMA transfers, because after a DMA word transfer has begun in the ADSP-21161, it must be completed (for example, it must receive the `ACK` signal). If $\overline{\mathtt{SBTS}}$ and $\overline{\mathtt{HBR}}$ are asserted during a DMA access, the $\overline{\mathtt{HBG}}$ pin is not asserted until the access cycle has completed. If the single DMA access is not allowed to complete, a deadlock condition may result.

To prevent system bus deadlock when using DMA, programs must make sure that SBG has been asserted before the DMA sequence begins. If a higher priority access is needed, the DMA sequence may be held off (by asserting $\overline{\mathtt{HBR}}$) at any time after a word has been transferred. Systems must ensure that `SBG` is asserted before $\overline{\mathtt{HBR}}$ is deasserted to prevent the possibility of another deadlock occurring. When the DMA sequence is complete, the DMA interrupt service routine should clear the external `SBR` flag.

Because the system bus is likely to be considerably slower than the local bus, performance on the local bus may be improved considerably by using handshake mode DMA. In this case, the `SBG` signal is tied to the DMA request line, $\overline{\mathtt{DMARx}}$. The local and system bus accesses are only initiated when the system bus is available.

Using a FIFO in the system interface unit, to allow DMA data from the local bus to be posted, may also increase performance on the local bus when using a slow system bus.

Multiprocessing With Local Memory

Figure 7-26 shows how several subsystems may be connected together on a system bus for high throughput. The gate array implements bus arbitration when the system bus is accessed. The buffers isolate the local buses from the system bus.

The example system in Figure 7-26 works in the following way:

- A processor requests the system bus with SBR when it asserts the $\overline{\text{MS2}}$ line. The gate array arbitrates between the SBR lines and then enables the highest priority group by asserting SBG, which is tied to FLAG0.
- The master processor may connect to system memory or to other processor groups. When the bus buffer is enabled, the read or write strobe enables should be asserted with a delay to allow the address to stabilize.
- To access a processor slave in another group, the master processor addresses that group's multiprocessor memory space. The gate array detects group multiprocessor memory space from three high-order address bits and asserts the $\overline{\text{HBR}}$ line for the selected group. When $\overline{\text{HBG}}$ is asserted, the gate array enables the slave's bus buffer. The high-order group address bits are cleared by the buffer to allow the group to decode the address as local multiprocessor memory space. The access is asynchronous because the $\overline{\text{CS}}$ line is asserted. The single waitstate option for the bus should be enabled.
- If two groups access each other in the same cycle, a deadlock may occur. The $\overline{\text{SBTS}}$ pin may be used to clear the deadlock.

ADSP-21161 to Microprocessor Interface

A ADSP-21161 without external memory may connect to a host microprocessor's bus. Depending on the microprocessor's I/O capabilities, the interface may not require any buffers. This type of connection assumes

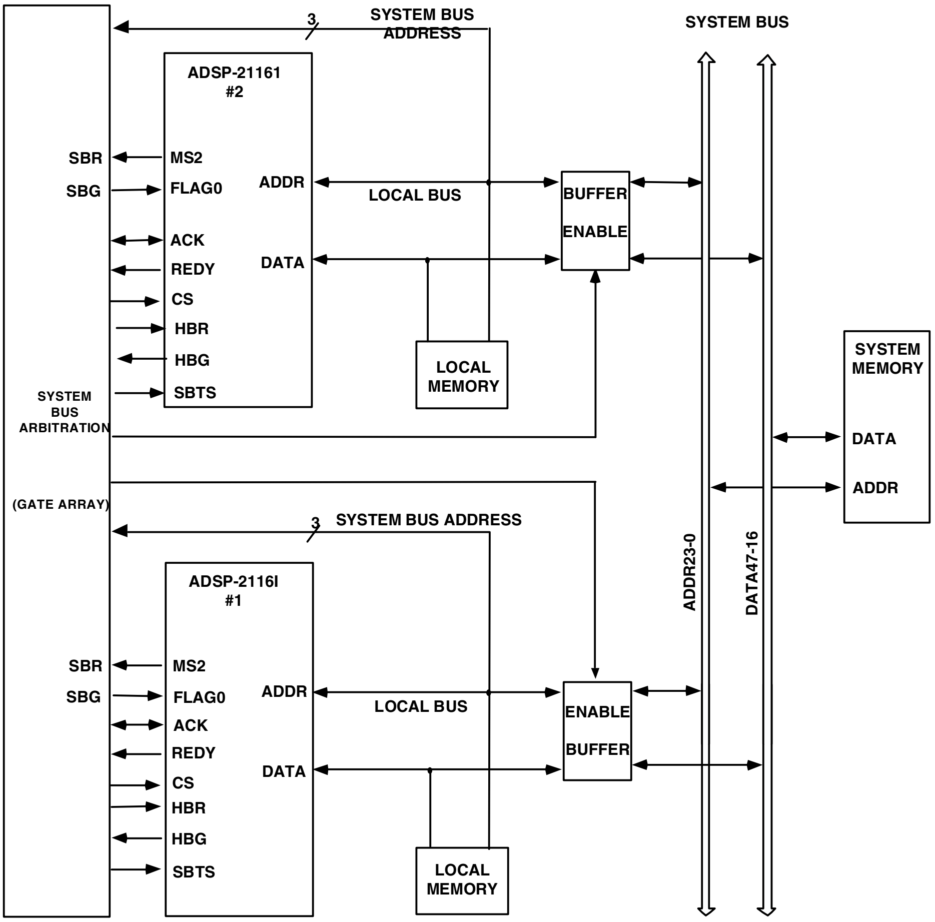


Figure 7-26. Subsystems on a System Bus

that the processor can execute its application from internal memory most of the time and only occasionally needs to request an external access. The host microprocessor should always keep the $\overline{\text{HBR}}$ request asserted unless it sees $\overline{\text{BR1}}$ asserted (for the $\overline{\text{BRx}}$ line of the processor with ID=001). The host can then deassert $\overline{\text{HBR}}$ to allow the processor to perform an external access when the host is ready to give up its bus. Usually, the host can read or write to the processor as needed. The host accesses the processor by asserting $\overline{\texttt{CS}}$ and handshaking with `REDY`. The $\overline{\texttt{HBG}}$ is not necessary in this system.

Multiprocessor (MP) Interface

The ADSP-21161 processor supports connecting to other ADSP-21161 processors to create multiprocessing processor systems. This support includes:

- Distributed, on-chip arbitration for the shared external bus
- A unified multiprocessor address space that makes the I/O processor registers of all processors directly accessible to each processor (and host interface)
- Dedicated hardware support for interprocessor communication (for example, reflective semaphores)
- Dedicated, point-to-point communication channels between processors using the link ports

Figure 7-27 illustrates a basic multiprocessing system. In a multiprocessor system with several processors sharing the external bus, any of the processors can become the bus master. The bus master has control of the bus, which consists of the `DATA47-16`, `ADDR23-0`, and associated control lines.

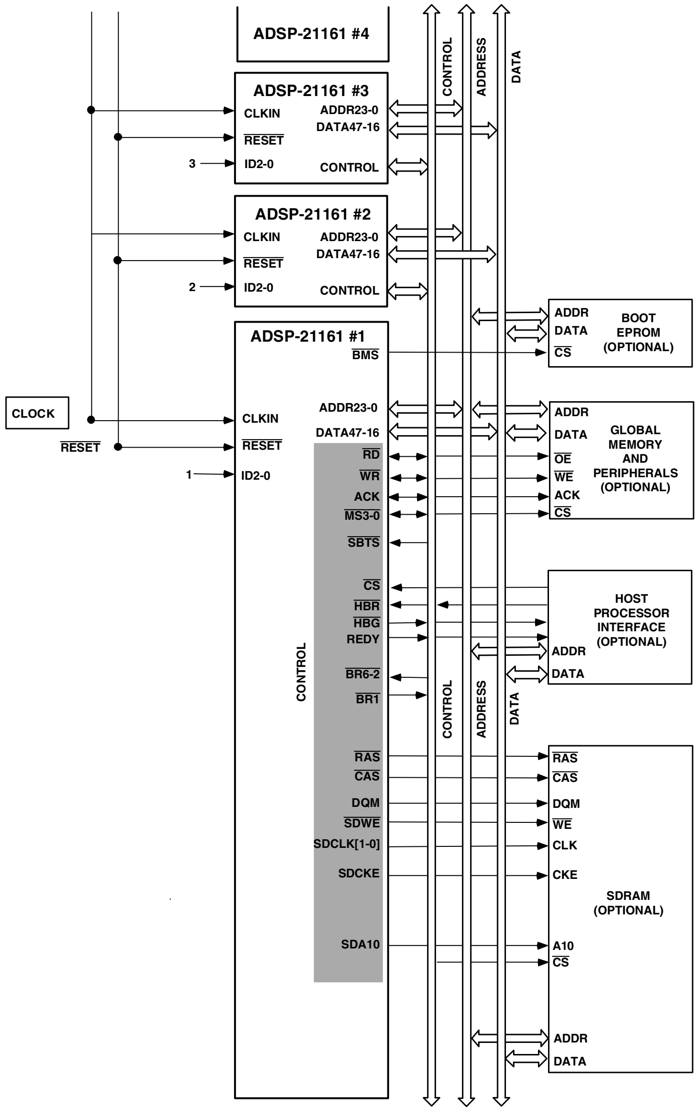


Figure 7-27. ADSP-21161 Multiprocessor System

Table 7-17 shows the external port signals for multiprocessor processor arbitration and communication.

Table 7-17. Signal for Cluster Multiprocessor Systems

Signal Types	Signals
Synchronization	CLKIN, $\overline{\text{RESET}}$
Arbitration	$\overline{\text{BR6-1}}$, $\overline{\text{PA}}$[1]
Bused Information	ADDR23-0, DATA47-16
Master Controls	$\overline{\text{RD}}$, $\overline{\text{WR}}$, BRST
Slave Control	ACK
Host Interface[2]	$\overline{\text{HBR}}$, $\overline{\text{HBG}}$, $\overline{\text{CS}}$, REDY, $\overline{\text{SBTS}}$

1 Optional, only needed if Priority Access function is used
2 Optional, only needed if Host Interface is used.

The I/O processor registers of the system's processors make up the multiprocessor memory space. Multiprocessor memory space is mapped into the unified address space of each processor. For more information, see the multiprocessor memory map in Figure 5-8 on page 5-20.

After a processor becomes the bus master, it can read and write to any of the slave's I/O processor registers, including their external port FIFO data buffers. For example, the master processor may write to a slave's I/O processor registers to set up DMA transfers or to send a vector interrupt.

The ADSP-21161 processor only supports direct reads and writes to I/O processor registers. However, internal memory can be accessed indirectly through `EPBx` DMA transfers.

Multiprocessing System Architectures

Multiprocessor systems typically use one of two schemes to communicate between processor nodes. One scheme uses dedicated point-to-point communication channels. In the other scheme, nodes communicate through a single shared global memory over a parallel bus.

The ADSP-21161 supports point-to-point communication—data flow multiprocessing—through its two link ports. Also, the ADSP-21161 supports a shared parallel bus communication—cluster multiprocessing—through its link ports and external port. The following sections provide more detail on on data flow multiprocessing and cluster multiprocessing.

Data Flow Multiprocessing

Data flow multiprocessing works for applications requiring high computational bandwidth, but requiring only limited flexibility. The program partitions its algorithm sequentially across multiple processors and passes data through a line of processors, as shown in Figure 7-28.

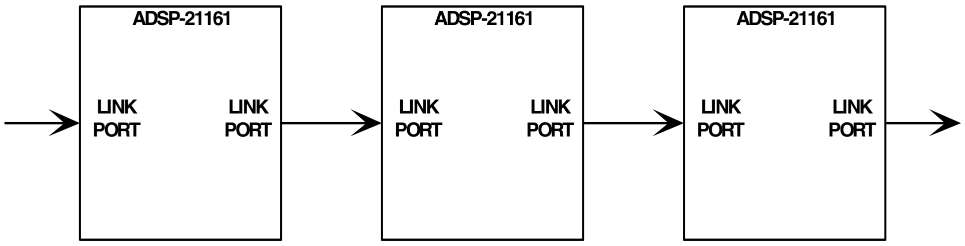


Figure 7-28. Data Flow Multiprocessing

The ADSP-21161 provides complete support for data flow multiprocessing applications, because the processor eliminates the need for interprocessor data FIFOs and external memory. The internal memory of the processor is usually large enough to contain both code and data for most applications using data-flow system topology. Data flow systems only require a number of processors and point-to-point signals connecting them. This design yields savings in complexity, board space, and system cost. For more information on connecting multiple processors using link ports, see "Host Processor Access To Link Buffers" on page 9-14.

Cluster Multiprocessing

Cluster multiprocessing works for applications where flexibility is required. This flexibility is needed when a system must be able to support a variety of different tasks, some of which may be running concurrently. The cluster multiprocessing configuration is shown in Figure 7-29. Also, the processor has an on-chip host interface that lets a cluster be interfaced to a host processor or another cluster.

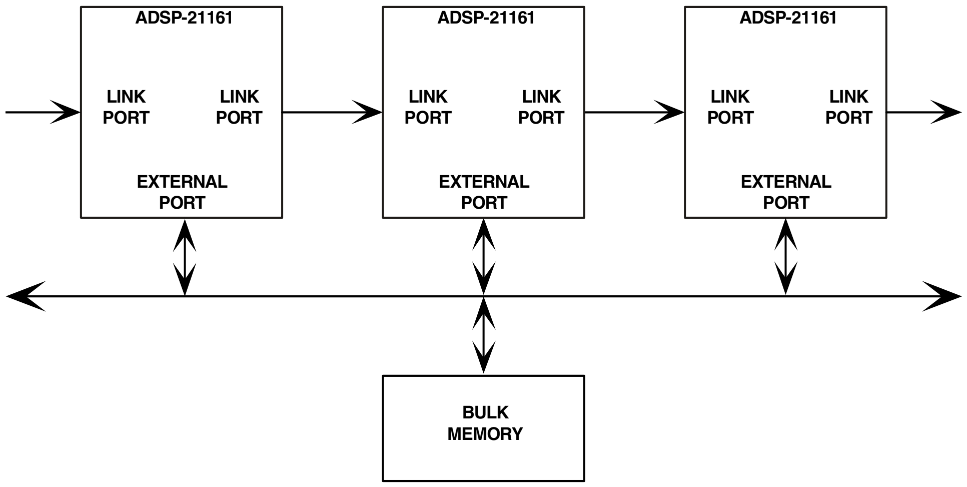


Figure 7-29. Cluster Multiprocessing

Cluster multiprocessing systems include multiple ADSP-21161s connected to a parallel bus that supports interprocessor access of on-chip memory-mapped registers and access to shared global memory. In a typical cluster of processors, up to six processors and a host can arbitrate for the bus. The on-chip bus arbitration logic lets these processors share the common bus. The ADSP-21161's features (such as large internal memory, link ports, and external port FIFOs) help eliminate the need for any extra hardware in the cluster multiprocessor configuration. External memory, both local and global, can frequently be eliminated in this type of system.

The ADSP-21161 supports fixed and rotating priority schemes. Other supported techniques include bus locking, timed release, DMA prioritization, and core processor access preemption of background DMA transfers. The on-chip arbitration logic lets transitions in bus mastership take up to only one cycle of overhead. Bus requests are generated implicitly when a processor accesses an external address. Because each processor monitors all bus requests and applies the same priority logic to the requests, each can independently determine who is the next bus master.

After getting mastership of the bus, a processor can access external memory and the I/O processor registers of all other processors (slaves) in the system. A processor can directly transfer data to another processor or set up a DMA channel to transfer the data. The processors are mapped into a common memory map—to identify the address space of each processor within the unified memory map of the system cluster. Also, each processor has a unique ID. The processor's I/O processor registers and external memory are all part of the unified address space.

The cluster configuration allows the processors to have a very fast node-to-node data transfer rate. Clusters also allow a simple, efficient software communication model. For example, all of the required setup operations for a DMA transfer can be accomplished by a single processor on one side of the transfer. The other processor is not interrupted until the DMA transfer is complete.

The ADSP-21161's internal memory facilitates I/O in multiprocessor systems. The on-chip, dual-ported RAM supports full-speed inter-processor DMA transfers concurrent with dual accesses by the processor's processor core. Because no cycles are stolen from the processor core, the processor's full performance is maintained during these accesses.

Link Port Data Transfers In A Cluster. A bottleneck exists within the cluster because only two processors can communicate over the shared bus during each cycle—other processors are held off until the bus is released. Because the processor can also perform point-to-point link port transfers within a cluster, systems can eliminate this bottleneck by setting up data communication through the link ports. Data links between processors can be dynamically set up and initiated over the common bus. Both link ports can operate simultaneously on each processor.

A disadvantage of the link ports is that individual transfers occur at a much lower rate than that of the shared parallel bus. Because the link ports' 8-bit data path is smaller than the processor's native word size, the transfer of each word requires multiple clock cycles. Link ports may also require more software overhead and complexity because they must be set up on both sides of the transfers before they can occur.

SIMD Multiprocessing. For certain classes of applications such as radar imaging, a Single-Instruction Multiple-Data (SIMD) array of processors may be the most efficient topology to coordinate a large number of processors in a single system. The SIMD array of Figure 7-29 on page 7-91 consists of multiple processors connected in a two- or three-dimensional mesh. The data link ports provide nearest neighbor communications and through-routing of data. A single master processor provides the instruction stream that the array executes. Data flow in and out the array can be managed through multiple serial port streams.

Multiprocessor Bus Arbitration

Multiple processors can share the external bus with no additional arbitration logic. Arbitration logic is included on-chip to allow the connection of up to six processors and a host processor.

The processor accomplishes bus arbitration through the $\overline{\text{BR1-6}}$, $\overline{\text{HBR}}$, and $\overline{\text{HBG}}$ signals. $\overline{\text{BR1-6}}$ arbitrate between multiple processors, and $\overline{\text{HBR}}$/$\overline{\text{HBG}}$ pass control of the bus from the processor bus master to the host and back. The priority scheme for bus arbitration is determined by the setting of the RPBA pin. Table 7-18 defines the processor pins used in multiprocessing systems.

Table 7-18. MultiprocessingPins

Signal	Type	Definition
$\overline{\text{BR6-1}}$	I/O/S	**Multiprocessing Bus Requests.** Used by multiprocessing to arbitrate for bus mastership. A processor only drives its own $\overline{\text{BRx}}$ line (corresponding to the value of its ID2-0 inputs) and monitors all others. In a multiprocessor system with less than six processors, the unused $\overline{\text{BRx}}$ pins should be tied high; the processor's own $\overline{\text{BRx}}$ line must not be tied high or low because it is an output.
ID2-0	I	**Multiprocessing ID**. Determines which multiprocessing bus request ($\overline{\text{BR}}$1 - $\overline{\text{BR}}$6) is used by ADSP-21161 processor. ID = 001 corresponds to BR1, ID = 010 corresponds to BR2, and so on. Use ID = 000 or ID = 001 in single-processor systems. These lines are a system configuration selection that should be hardwired or only changed at reset.
RPBA	I	**Rotating Priority Bus Arbitration Select.** When RPBA is high, rotating priority for multiprocessor bus arbitration is selected. When RPBA is low, fixed priority is selected. This signal is a system configuration selection which must be set to the same value on every processor. If the value of RPBA is changed during system operation, it must be changed in the same CLKIN cycle on every processor.
$\overline{\text{PA}}$	(a/d) I/O/S	**Priority Access.** The processor slave may assert the $\overline{\text{PA}}$ signal to interrupt background DMA transfers and gain access to the external bus. This signal is asserted when a processor slave's processor core requests the bus or if an external DMA channel requests the bus with the DMACx PRIO control bit set. The $\overline{\text{PA}}$ signal is an active drive output, which may be asserted (low) by one or more slaves. It is deasserted (high) by the master. A protocol is used to avoid driver contention.
I = Input, S = Synchronous, (o/d) = Open Drain; O = Output, A = Asynchronous, (a/d) = Active Drive		

The ID2-0pins provide a unique identity for each processor in a multiprocessing system. The first processor should be assigned ID=001, the second should be assigned ID=010, and so on. One of the processors must be assigned ID=001 in order for the bus synchronization scheme to function properly.

The processor with ID=001 holds the external bus control lines stable during reset.

When the ID2-0 inputs of a processor are equal to 001, 010, 011, 100, 101, or 110, the processor configures itself for a multiprocessor system and maps its I/O processor registers into the multiprocessor memory space. ID=000 configures the processor for a single-processor system. ID=111 is reserved and should not be used.

A processor in a multiprocessor system can determine which processor is the current bus master, by reading the CRBM2-0 bits of the SYSTAT register. These bits give the value of the ID2-0 inputs of the current bus master.

Conditional instructions can be written that depend upon whether the processor is the current bus master in a multiprocessor system. The assembly language mnemonic for this condition code is BM, and its complement is Not BM (not bus master). The BM condition indicates whether the processor is the current bus master. For more information, see "Conditional Sequencing" on page 3-53. To use the bus master condition, the condition code select (CSEL) field in the MODE1 register must be zero or the condition is always evaluated as false.

Bus Arbitration Protocol

The Bus Request ($\overline{\text{BR1-6}}$) pins are connected between each processor in a multiprocessing system, with the number of $\overline{\text{BRx}}$ lines used equal to the number of processors in the system. Each processor drives the $\overline{\text{BRx}}$ pin that corresponds to its ID2-0 inputs and monitors all others. If less than six processors are used in the system, the unused $\overline{\text{BRx}}$ pins should be tied high.

When one of the slave processors needs to become bus master, it automatically initiates the bus arbitration process by asserting its $\overline{\text{BRx}}$ line at the beginning of the cycle. Later in the same cycle, the processor samples the value of the other $\overline{\text{BRx}}$ lines.

The cycle in which mastership of the bus is passed from one processor to another is called a Bus Transition Cycle (BTC). A bus transition cycle occurs when the current bus master's $\overline{\text{BRx}}$ pin is deasserted and one or more of the slave's $\overline{\text{BRx}}$ pins is asserted. The bus master can retain bus mastership by keeping its $\overline{\text{BRx}}$ pin asserted. Also, the bus master does not always lose bus mastership when it deasserts its $\overline{\text{BRx}}$ line—another $\overline{\text{BRx}}$ line must be asserted by one or more of the slaves at the same time. In this case, when no other $\overline{\text{BRx}}$ is asserted, the master does not lose any bus cycles.

By observing all of the $\overline{\text{BRx}}$ lines, each processor can detect when a bus transition cycle occurs and which processor has become the new bus master. A bus transition cycle is the only time that bus mastership is transferred.

After conditions determine that a bus transition cycle is going to occur, every processor in the system evaluates the priority of the $\overline{\text{BRx}}$ lines asserted within that cycle. For a description of bus arbitration priority, see "Bus Arbitration Priority (RPBA)" on page 7-98. The processor with the highest priority request becomes the bus master on the following cycle, and all of the processors update their internal records to indicate which processor is the current bus master. This information can be read from the current bus master field, CRBM, of the SYSTAT register. Figure 7-29 on page 7-91 shows typical timing for bus arbitration.

The actual transfer of bus mastership is accomplished by the current bus master three-stating the external bus—DATA47-16, ADDR23-0, CLKOUT[1], $\overline{\text{RD}}$, $\overline{\text{WR}}$, BRST, $\overline{\text{MS3-0}}$, $\overline{\text{HBG}}$, $\overline{\text{DMAG2-1}}$—at the end of the bus transition cycle and the new bus master driving these signals at the beginning of the next cycle.

[1] For a complete description of CLKOUT functionality, see Table 13-1 on page 13-4.

The bus strobes ($\overline{\text{RD}}$, $\overline{\text{WR}}$) and $\overline{\text{MS3-0}}$ are driven high (inactive) before three-stating occurs. ACK must be sampled high by the new master before it starts a new bus operation. For more information, see Figure 7-30.

During bus transition cycle delays, execution of external accesses are delayed. When one of the slave processors needs to perform an external read or write, it automatically initiates the bus arbitration process by asserting its $\overline{\text{BRx}}$ line. This read or write is delayed until the processor receives bus mastership. If the read or write was generated by the processor's processor core (not the I/O processor), program execution stops on that processor until the instruction is completed.

The following steps occur as a slave acquires bus mastership and performs an external read or write over the bus as shown in Figure 7-31 on page 7-100.

1. The slave determines that it is executing an instruction which requires an off-chip access. It asserts its $\overline{\text{BRx}}$ line at the beginning of the cycle. Extra cycles are generated by the core processor (or I/O processor) until the slave acquires bus mastership.

2. To acquire bus mastership, the slave waits for a bus transition cycle in which the current bus master deasserts its $\overline{\text{BRx}}$ line. If the slave has the highest priority request in the bus transition cycle, it becomes the bus master in the next cycle. If not, it continues waiting.

3. At the end of the bus transition cycle the current bus master releases the bus, and the new bus master starts driving.

During the CLKIN cycle in which the bus master deasserts its $\overline{\text{BRx}}$ output, it three-states its outputs in case another bus master wins arbitration and enables its drivers in the next CLKIN cycle. If the current bus master retains control of the bus in the next cycle, it enables its bus drivers, even if it has no bus operation to run.

The processor with ID=00x enables internal keeper latches, or pullup devices, on key signals, including the address and data buses, strobes, and ACK. These devices provide a weak current source or sink—approximate 20KΩ impedance—to keep these signals from drifting near input receiver thresholds when all drivers are three-stated.

When the bus master stops using the bus, its $\overline{\text{BRx}}$ line is deasserted, allowing other processors to arbitrate for mastership if they need it. If no other processors are asserting their $\overline{\text{BRx}}$ line when the master deasserts its $\overline{\text{BRx}}$, the master retains control of the bus and continues to drive the memory control signals until: 1) it needs to use the bus again, or 2) another processor asserts its $\overline{\text{BRx}}$ line.

While a slave waits to be a master for a DMA transfer, it asserts $\overline{\text{BRx}}$. If that slave's core accesses the DMA address registers, the $\overline{\text{BRx}}$ is deasserted during that access. See "I/O Processor Registers Memory Map" on page A-51.

Bus Arbitration Priority (RPBA)

To resolve competing bus requests, there are two available priority schemes: fixed and rotating. The RPBA pin selects the scheme. When RPBA is high, rotating priority bus arbitration is selected, and when RPBA is low, fixed priority is selected.

The RPBA pin must be set to the same value on each processor in a multiprocessing system. If the value of RPBA is changed during system operation, it must be changed synchronously to CLKIN and must meet a setup time that lets all processors recognize the change in the same cycle. The priority scheme changes in that (same) cycle.

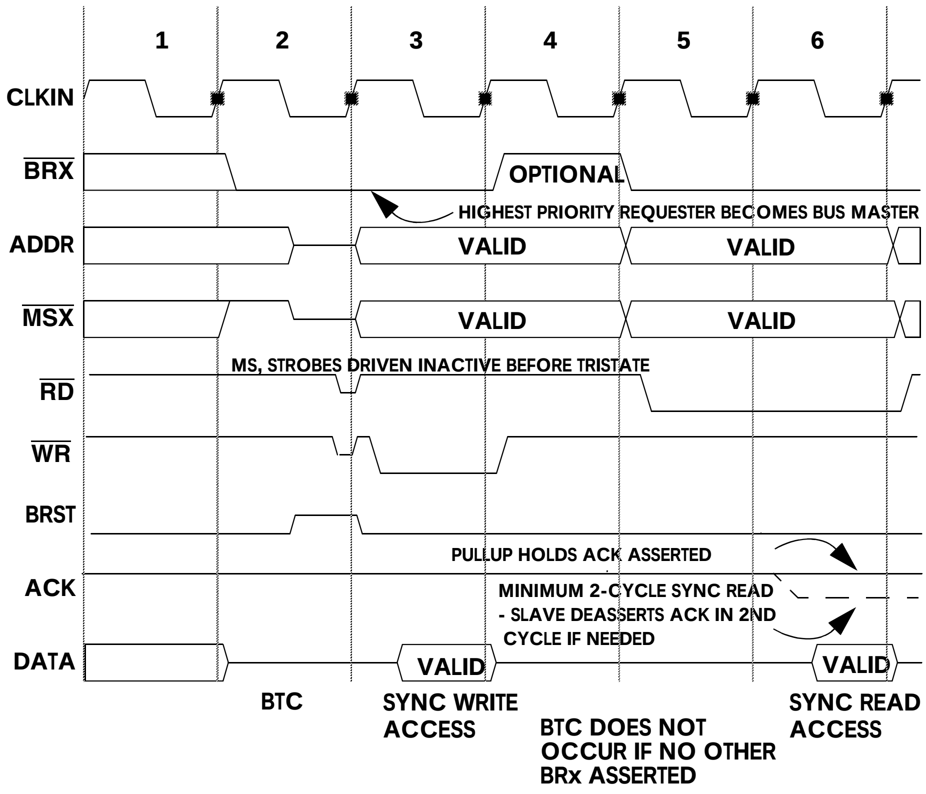

1
2
3
4
5
6
CLKIN
BRX
OPTIONAL
HIGHEST PRIORITY REQUESTER BECOMES BUS MASTER
ADDR
VALID
VALID
MSX
VALID
VALID
MS, STROBES DRIVEN INACTIVE BEFORE TRISTATE
RD
WR
BRST
PULLUP HOLDS ACK ASSERTED
ACK
MINIMUM 2-CYCLE SYNC READ
- SLAVE DEASSERTS ACK IN 2ND
CYCLE IF NEEDED
DATA
VALID
VALID
BTC
SYNC WRITE ACCESS
BTC DOES NOT OCCUR IF NO OTHER BRx ASSERTED
SYNC READ ACCESS

Figure 7-30. Bus Request and Read/Write Timing

In the fixed priority scheme, the processor with the lowest ID number among the competing bus requests becomes the bus master. If, for example, the processor with ID=010 and the processor with ID=100 request the bus simultaneously, the processor with ID=010 becomes bus master in the following cycle.

Each processor knows the ID of the other processors requesting the bus, because the ID corresponds to the $\overline{\text{BRx}}$ line being used for each processor.

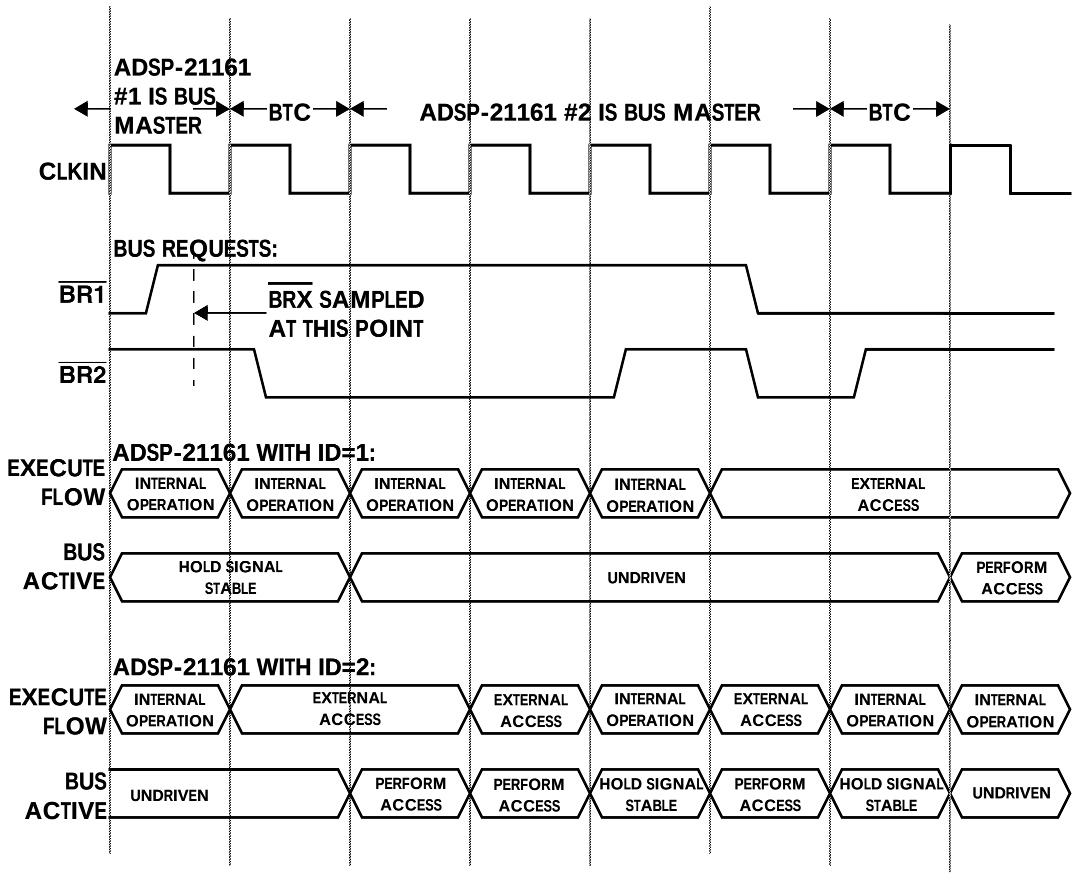


Figure 7-31. Bus Arbitration Timing

The rotating priority scheme gives roughly equal priority to each processor. When rotating priority is selected, the priority of each processor is reassigned after every transfer of bus mastership. Highest priority is rotated from processor to processor as if they were arranged in a circle—

the processor located next to (one place down from) the current bus master is the one that receives highest priority. Table 7-19 shows an example of how rotating priority changes on a cycle-by-cycle basis.

Table 7-19. Rotating Priority Arbitration Example

Cycle Number	Hardwired Processor IDs & Priority[1]					
	ID1	ID2	ID3	ID4	ID5	ID6
1[2]	M	1	2-$\overline{BR}$	3	4	5
2	4	5-$\overline{BR}$	M-$\overline{BR}$	1	2	3
3	4	5-$\overline{BR}$	M	1	2	3
4	5-$\overline{BR}$	M	1	2	3	4-$\overline{BR}$
5[3]	1-$\overline{BR}$	2	3	4	5	M

1 The following symbols appear in these cells: 1-5 = assigned priority, M = bus mastership (in that cycle), $\overline{BR}$ = requesting bus mastership with $\overline{BRx}$

2 Initial priority assignments

3 Final priority assignments

Bus Mastership Timeout

In either the fixed or rotating priority scheme, systems may need to limit how long a bus master can retain the bus. Systems can limit bus mastership by forcing the bus master to deassert its $\overline{\text{BRx}}$ line after a specified number of CLKIN cycles and giving the other processors a chance to acquire bus mastership.

To set up a bus master timeout, a program must load the BMAX register (Figure 7-32) with the maximum number of CLKIN cycles (minus 2) that allows the processor to retain bus mastership. This equation is shown below

BMAX = (maximum # of bus mastership CLKIN cycles) – 2

Internal processor clock cycles are a multiple of CLKIN cycles.

The minimum value for BMAX is 2, which lets the processor retain bus mastership for four CLKIN cycles. Setting BMAX=1 is not allowed. To disable the bus master timeout function, set BMAX=0.

Each time a processor acquires bus mastership, its BCNT register is loaded with the value in BMAX. BCNT is then decremented in every CLKIN cycle that the master performs a read or write over the bus and any other (slave) processors are requesting the bus. Any time the bus master deasserts its $\overline{\text{BRx}}$ line, BCNT is reloaded from BMAX.

When BCNT decrements to zero, the bus master first completes its off-chip read/write and then deasserts its own $\overline{\text{BRx}}$ (any new off-chip accesses are delayed)—this allows transfer of bus mastership. If the ACK signal is holding off an access when BCNT reaches zero, bus mastership is not relinquished until the access can complete.

If BCNT reaches zero while a burst transfer is in progress, the bus master completes the burst transfer before deasserting its $\overline{\text{BRx}}$ output. If BCNT reaches zero while bus lock is active, the bus master does not deassert its $\overline{\text{BRx}}$ line until bus lock is removed. If $\overline{\text{HBR}}$ is being serviced, BCNT stops decrementing and continues only after $\overline{\text{HBR}}$ is deasserted.

Bus lock is enabled by the BUSLK bit in the MODE2 register. For more information, see "Bus Lock and Semaphores" on page 7-110.

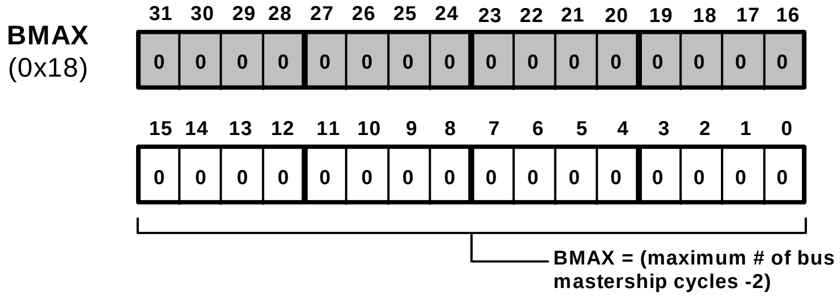


Figure 7-32. BMAX Register

Priority Access

The Priority Access signal ($\overline{\text{PA}}$) lets external bus accesses by a slave processor take priority over ongoing DMA transfers. Normally when external port DMA transfers are in progress, the slave processors cannot use the external bus until the DMA transfer is finished. By asserting its $\overline{\text{PA}}$ pin, the slave processor can acquire the bus without waiting for the DMA operation to complete. The $\overline{\text{PA}}$ signal can also be asserted by a slave with a high-priority DMA access pending on the external bus.

If the $\overline{\text{PA}}$ signal is not used in a multiprocessor system, the processor bus master does not give up the bus to another processor until: 1) a cycle in which it does not perform an external bus access or 2) a bus timeout. If a slave processor needs to send a high priority message or perform an important data transfer, it normally must wait until any DMA operation completes. Using the $\overline{\text{PA}}$ signal lets the slave perform its higher priority bus access with less delay.

Each of the DMACx registers has a PRIO bit that raises that DMA channel to a higher priority than all other internal DMA channels that do not have the PRIO bit set. Unless configured differently with the EBPR bit in the SYSCON register, this channel still has lower priority (internally) than the core. Programs should be careful to minimize the number of DMA channels enabled to high priority status in the multiprocessor system, because both core and (external) high priority DMA requests from slaves are arbitrated at the same priority level. For example, a slave core cannot arbitrate bus ownership away from a high priority DMA transfer, unless the bus timeout (BMAX function) occurs.

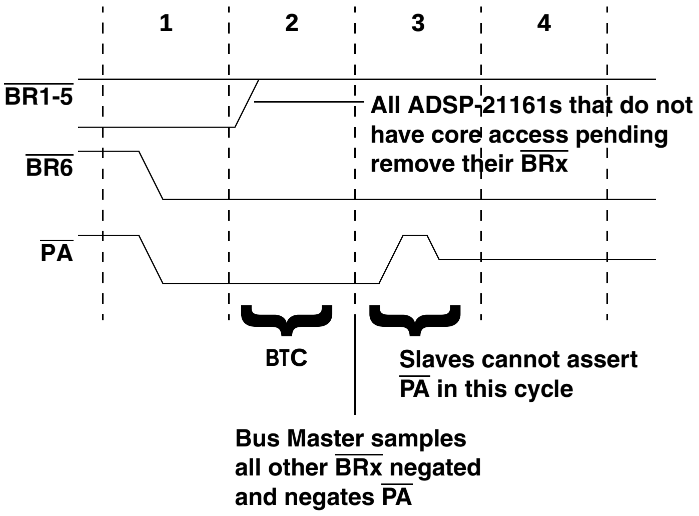
When $\overline{\text{PA}}$ is asserted, the current processor bus master deasserts its $\overline{\text{BRx}}$ output, and gives up the bus, provided:

1. Its core does not have an external access pending, and
2. None of its external bus DMA channels have pending high-priority bus requests.

All processor slaves also deassert their $\overline{\text{BRx}}$ outputs, if each slave meets the same provisions. The current bus master never asserts $\overline{\text{PA}}$, because it already has control of the bus. If the current master detects a condition that would assert $\overline{\text{PA}}$ while it is bus master, it performs that high priority operation before giving up bus ownership.

In the CLKIN cycle after $\overline{\text{PA}}$ has been asserted, only the processor slaves with a pending high priority access have their bus requests asserted. Bus arbitration proceeds as usual with the highest priority device becoming the master when the previous bus master releases its $\overline{\text{BRx}}$ output.

The new master samples all $\overline{\text{BRx}}$ inputs after gaining bus mastership—during the cycle that follows the BTC. If no other bus requests are asserted, the master is the only device driving $\overline{\text{PA}}$, and the master deasserts and three-states $\overline{\text{PA}}$ in this cycle as shown in Figure 7-33.



1
2
3
4
BR1-5
BR6
PA
All ADSP-21161s that do not have core access pending remove their BRx
BTC
Slaves cannot assert PA in this cycle
Bus Master samples all other BRx negated and negates PA

Figure 7-33. Example $\overline{\text{PA}}$ Deassertion

If the master samples other $\overline{\text{BRx}}$ inputs as asserted, multiple devices are driving $\overline{\text{PA}}$, and the new bus master cannot deassert $\overline{\text{PA}}$. The new bus master three-states its $\overline{\text{PA}}$ driver in this case. All processor slaves recognize the cycle following the BTC. They do not assert $\overline{\text{PA}}$ during this cycle, unless they were already driving their $\overline{\text{BR}}$ and $\overline{\text{PA}}$ outputs in the BTC. This behavior is demonstrated in Figure 7-34.

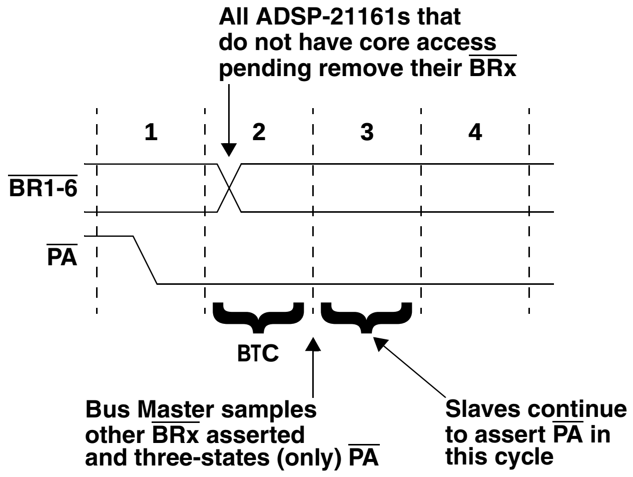

All ADSP-21161s that do not have core access pending remove their $\overline{\text{BRx}}$
1
2
3
4
$\overline{\text{BR1-6}}$
$\overline{\text{PA}}$
BTC
Bus Master samples other $\overline{\text{BRx}}$ asserted and three-states (only) $\overline{\text{PA}}$
Slaves continue to assert $\overline{\text{PA}}$ in this cycle

Figure 7-34. Example of $\overline{\text{PA}}$ Driven by Multiple Slaves

Bus Synchronization After Reset

When a multiprocessing system is reset ($\overline{\text{RESET}}$ asserted), the bus arbitration logic on each processor must synchronize, making sure that only one processor drives the external bus. One processor must become the bus master, and all other processors must recognize which one it is before actively arbitrating for the bus. The bus synchronization scheme also lets the system safely bring individual processors into and out of reset.

One of the processors in the system must be assigned ID=001 in order for the bus synchronization scheme to function properly. This processor also holds the external bus control lines stable during reset. Bus arbitration synchronization is disabled if the processor is in a single-processor system (ID=000).

To synchronize their bus arbitration logic and define the bus master after a system reset, the multiple processors obey the following rules:

- All processors except the one with ID=001 deassert their $\overline{\text{BRx}}$ line during reset. They keep their $\overline{\text{BRx}}$ deasserted for at least two cycles after reset and until their bus arbitration logic is synchronized[1].

- After reset, a processor considers itself synchronized when it detects a cycle in which only one $\overline{\text{BRx}}$ line is asserted. The processor identifies the bus master by recognizing which $\overline{\text{BRx}}$ is asserted and updates its internal record to indicate the current master.

- The processor with ID=001 asserts its $\overline{\text{BRx}}$ ($\overline{\text{BR1}}$) during reset and for at least two cycles after reset. If no other $\overline{\text{BRx}}$ lines are asserted during these cycles, the processor with ID=001 drives the memory control signals to prevent them from glitching. Although it is asserting its $\overline{\text{BRx}}$ and driving the memory control signals during these cycles, this processor does not perform reads or writes over the bus.

If the processor with ID=001 is synchronized by the end of the two cycles following reset, it becomes the bus master. If it is not synchronized at this time, it deasserts its $\overline{\text{BRx}}$ ($\overline{\text{BR1}}$) and waits until it is synchronized.

ⓘ When a processor has synchronized itself, it sets the BSYN bit in the SYSTAT register.

[1] For a complete description of the functionality of the internal reset signal, $\overline{\text{RSTOUT}}$, see Table 13-1 on page 13-4.

If one processor comes out of reset after the others have synchronized and started program execution, that processor may not be able to synchronize immediately (for example, if it detects more than one $\overline{\texttt{BRx}}$ line asserted). If the un-synchronized processor tries to execute an instruction with an off-chip read or write, it cannot assert its $\overline{\texttt{BRx}}$ line to request the bus and execution is delayed until it can synchronize and correctly arbitrate for the bus.

Synchronization cannot occur while $\overline{\texttt{HBG}}$ is asserted, because bus arbitration is suspended while the bus is controlled by a host. If $\overline{\texttt{HBR}}$ is asserted immediately after reset and no bus arbitration has taken place, the processor with `ID`=001 is considered to be the last bus master.

The processor with `ID`=001 maintains correct logic levels on the $\overline{\texttt{RD}}$, $\overline{\texttt{WR}}$, $\overline{\texttt{MS3-0}}$, and $\overline{\texttt{HBG}}$ signals during reset. Because the "001" processor can be accidently reset by an erroneous write to the soft reset bit (`SRST`) of the `SYSCON` register, it behaves in the following manner during reset:

- While it is in reset, the processor with `ID`=001 attempts to gain control of the bus by asserting $\overline{\texttt{BR1}}$.
- While it is in reset, the processor with `ID`=001 drives the $\overline{\texttt{RD}}$, $\overline{\texttt{WR}}$, $\overline{\texttt{MS3-0}}$, $\overline{\texttt{DMAG1}}$, $\overline{\texttt{DMAG2}}$, and $\overline{\texttt{HBG}}$ signals only if it determines that it has control of the bus. For the processor to decide it has control of the bus, two conditions must be true: 1) $\overline{\texttt{BR1}}$ was asserted and no other $\overline{\texttt{BRx}}$ lines were asserted in the previous cycle, and 2) $\overline{\texttt{HBG}}$ was deasserted in the previous cycle.

The processor with `ID`=001 continues to drive the $\overline{\texttt{RD}}$, $\overline{\texttt{WR}}$, $\overline{\texttt{MS3-0}}$, $\overline{\texttt{DMAG1}}$, $\overline{\texttt{DMAG2}}$, and $\overline{\texttt{HBG}}$ signals for two cycles after reset, as long as neither $\overline{\texttt{HBG}}$ nor any other $\overline{\texttt{BRx}}$ lines are asserted. At the end of the second cycle it assumes bus mastership (if it is synchronized), and normal bus arbitration begins in the following cycle. If it is not synchronized, it deasserts $\overline{\texttt{BR1}}$, stops driving the memory control signals and does not arbitrate for the bus until it becomes synchronized.

Although the bus synchronization scheme allows individual processors to be reset, the processor with ID=001 may fail to drive the memory control signals if it is in reset while any other processors are asserting their $\overline{\text{BRx}}$ line.If the processor with ID=001 has asserted $\overline{\text{HBG}}$ while it is in reset, it is synchronized when $\overline{\text{RSTOUT}}$ is deasserted[1]. This lets the host start using the bus while the processors are still in reset. If a host processor attempts to reset the processor bus master (which is driving the $\overline{\text{HBG}}$ output), the host immediately loses control of the bus.

During reset[2], the ACK line is pulled high internally by the processor bus master with a 20 kΩ equivalent resistor.

Booting Another processor

If the system uses one processor to boot another processor over the cluster bus, the master processor must (for maximum efficiency) do the following to communicate to the slave processor through the external port interface:

1. Program the PMODE field in DMAC10 of the booting processor for 32- to 48-bit packing. This modification must be made to the boot loader kernel as well.

2. Write 48-bit words to EPB0 on the booting processor.

[1] For a complete description of the functionality of the internal reset signal, $\overline{\text{RSTOUT}}$, see Table 13-1 on page 13-4.

[2] For a complete description of the functionality of the internal reset signal, $\overline{\text{RSTOUT}}$, see Table 13-1 on page 13-4.

Multiprocessor Writes and Reads

A processor bus master can read or write to the I/O processor registers of a slave processor. For more information, see "Slave Reads and Writes" on page 7-55.

For synchronous write accesses, the slave write FIFO functions as a 2-deep FIFO. One or both of the stages may be used to store write accesses. If a synchronous write to this processor completes by the end of cycle N and if this is the first write to be stored in the slave write FIFO (for example, due to stalled write to the EPBx FIFO), then the ACK deasserts in cycle N+2. If a subsequent write to the same slave processor completes in cycle N+1, the access is correctly stored in the second stage of slave FIFO. Independent of this access in cycle N+1, an access in cycle N+2 is stalled on the bus due to a deasserted ACK. Only when the slave write FIFO is empty is ACK asserted again.

Each processor bus slave monitors addresses driven on the external bus and responds to any that fall within its region of multiprocessor memory space. These accesses are invisible to the slave processor's processor core. They do not degrade internal memory or internal bus performance as seen by the core. This feature lets the processor core continue program execution uninterrupted.

The processor bus master can read and write the slave's I/O processor registers (for example, SYSCON, SYSTAT) to send a vector interrupt or to set up DMA transfers.

For information on topics relevant to multiprocessing, see the following referenced sections:

- **IOP Shadow Registers.** For more information, see "IOP Shadow Registers" on page 7-55.
- **Slave Write Latency.** For more information, see "Slave Write Latency" on page 7-56.

- **Slave Reads.** For more information, see "Slave Reads" on page 7-57.
- **Shadow Write FIFO.** For more information, see "Slave Reads" on page 7-57.
- **Data Transfers Through the EPBx Buffers.** For more information, see "Data Transfers Through the EPBx Buffers" on page 7-58.
- **Interprocessor Messages & Vector Interrupts.** For more information, see "Interprocessor Messages and Vector Interrupts" on page 7-76.

Instruction Transfers

Multiprocessor instruction transfers to or from internal memory of processor should use 32-bit transfers for maximum performance. The 48-bit internal transfers use one of the slave `EPBx` FIFOs and the packing mode function (`PMODE`) of the DMA channel (32- to 48-bit).

Maximum throughput is achieved by transferring packed instructions to or from internal memory, using DMA transfers with 32- to 48-bit packing.

Bus Lock and Semaphores

Semaphores can be used in multiprocessor systems to allow the processors to share resources such as memory or I/O. A semaphore is a flag that can be read and written by any of the processors sharing the resource. The value of the semaphore tells the processor when it can access the resource. Semaphores are also useful for synchronizing the tasks being performed by different processors in a multiprocessing system.

With the use of its bus lock feature, the processor has the ability to read and modify a semaphore in a single indivisible operation—a key requirement of multiprocessing systems.

Because both external memory and each processor's I/O processor registers are accessible by every other processor, semaphores can be located almost anywhere. Read-modify-write operations on semaphores can be performed if all of the processors obey two simple rules:

1. A processor must not write to a semaphore unless it is the bus master. This is especially important if the semaphore is located in the processor's own internal memory or I/O processor registers.
2. When attempting a read-modify-write operation on a semaphore, the processor must have bus mastership for the duration of the operation.

Both of these rules apply when a processor uses its bus lock feature, which retains its mastership of the bus and prevents the other processors from simultaneously accessing the semaphore.

Bus lock is requested by setting the `BUSLK` bit in the `MODE2` register. When this happens, the processor initiates the bus arbitration process by asserting its $\overline{\mathtt{BRx}}$ line. When it becomes bus master, it locks the bus by keeping its $\overline{\mathtt{BRx}}$ line asserted even when it is not performing an external read or write. Host Bus Request ($\overline{\mathtt{HBR}}$) is also ignored during a bus lock. When the $\overline{\mathtt{BUSLK}}$ bit is cleared, the processor gives up the bus by deasserting its $\overline{\mathtt{BRx}}$ line.

While the `BUSLK` bit is set, the processor can determine if it has acquired bus mastership by executing a conditional instruction with the Bus Master (`BM`) or Not Bus Master (`Not BM`) condition codes, for example:

```
IF NOT BM JUMP(PC,0); /* Wait for bus mastership */
```

If it has become the bus master, the processor can proceed with the external read or write. If not, it can clear its `BUSLK` bit and try again later.

A read-modify-write operation is accomplished with the following steps:

1. Request bus lock by setting the BUSLK bit in MODE2.
2. Wait for bus mastership to be acquired.
3. Wait until Slave Write Pending bit (SWPD) is zero.
4. Read the semaphore, test it, then write to it.

Locking the bus prevents other processors from writing to the semaphore while the read-modify-write is occurring. After bus mastership is acquired, check the SWPD bit's status in SYSTAT to ensure that a semaphore write by another processor is not pending.

If the semaphore is reflective, located in one of the processor's I/O processor register, the processor must write to it only when it has bus lock.

Multiprocessor Interface Status

The SYSTAT register provides status information for host and multiprocessor systems. Figure 7-35 shows the status bits in this register.

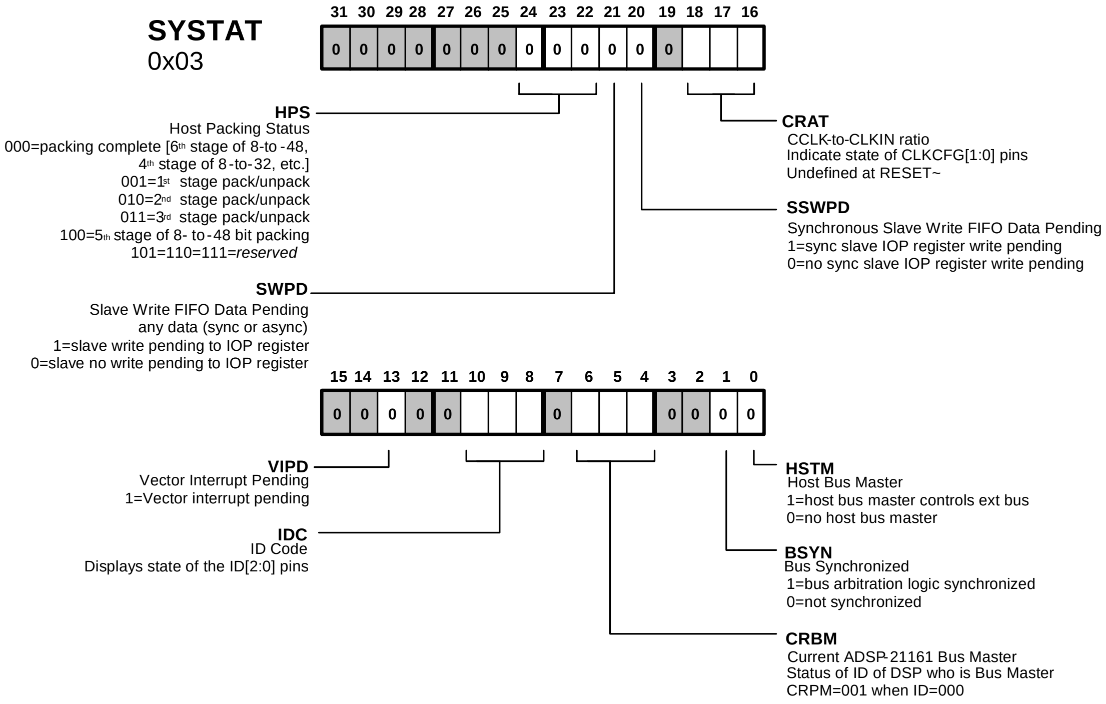

SYSTAT
0x03
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0
HPS
Host Packing Status
000=packing complete [6th stage of 8-to-48, 4th stage of 8-to-32, etc.]
001=1st stage pack/unpack
010=2nd stage pack/unpack
011=3rd stage pack/unpack
100=5th stage of 8- to-48 bit packing
101=110=111=reserved
SWPD
Slave Write FIFO Data Pending
any data (sync or async)
1=slave write pending to IOP register
0=slave no write pending to IOP register
CRAT
CCLK-to-CLKIN ratio
Indicate state of CLKCFG[1:0] pins
Undefined at RESET~
SSWPD
Synchronous Slave Write FIFO Data Pending
1=sync slave IOP register write pending
0=no sync slave IOP register write pending
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0
VIPD
Vector Interrupt Pending
1=Vector interrupt pending
IDC
ID Code
Displays state of the ID[2:0] pins
HSTM
Host Bus Master
1=host bus master controls ext bus
0=no host bus master
BSYN
Bus Synchronized
1=bus arbitration logic synchronized
0=not synchronized
CRBM
Current ADSP-21161 Bus Master
Status of ID of DSP who is Bus Master
CRPM=001 when ID=000

Figure 7-35. SYSTAT Register

8 SDRAM INTERFACE

The ADSP-21161 processor's synchronous DRAM (SDRAM) interface enables it to transfer data at either the core clock frequency or one-half the core clock frequency. The synchronous approach, coupled with the ability to transfer data at the core clock frequency, supports data transfer at a high throughput—up to 400 Mbytes/second for a 32-bit bus width, and 600 Mbytes/second for 48-bit bus width.

All inputs are sampled and all outputs are valid on the rising edge of the clock `SDCLK`. The SDRAM's flexible interface allows you to connect SDRAMs to any one or more of the four external memory banks of the ADSP-21161 processor or to all four banks simultaneously.

The ADSP-21161 processor's SDRAM controller provides a glueless interface with standard SDRAMs. It supports:

- SDRAMs of 16 Mbits, 64 Mbits, 128 Mbits, and 256 Mbits with configurations 4-bit, 8-bit, 16-bit and 32-bit wide devices
- Additional buffers between ADSP-21161 processor and SDRAM
- Zero wait state, 100 Mwords/second with some access types
- Up to 254.68 Mwords [3x(64M) + 62.68M] of SDRAM in external memory
- SDRAM page sizes of 2048, 1024, 512, and 256 words
- A programmable refresh counter to coordinate between varying clock frequencies and the SDRAM's required refresh rate
- Buffering for multiple SDRAMs connected in parallel

- Shared SDRAM devices in a multiprocessing system
- A separate A10 pin that enables applications to precharge SDRAM before issuing a refresh command
- Connection to up to four external memory banks (0 to 3) of the ADSP-21161 processor
- Self-refresh, low-power mode
- Two power-up options

The following are definitions used throughout this chapter:

- **Bank Activate command.** Activates the selected bank and latches in a new row address. It must be applied before a read or write command.
- **Burst length.** Determines the number of words that the SDRAM inputs or outputs after detecting a write or read command, respectively. The processor supports burst length ONE mode only.

  During a burst length of one cycle, the ADSP-21161 processor SDRAM controller applies the command every cycle and keeps accessing the data. See also, page size on page 8-3.
- **Burst type.** Determines the order in which the SDRAM delivers or stores burst data after detecting a read or write command, respectively. The processor supports sequential accesses only.
- **CAS latency.** The delay, in clock cycles, between when the SDRAM detects the read command and when it provides the data at its output pins. The speed grade of the device and the application's clock frequency determine the value of the CAS latency.

  The application must program the CAS latency value into the SDCTL register after power up.

- CBR Automatic Refresh (CAS before RAS) mode. In this mode, the SDRAM drives its own refresh cycle with no external control input. At cycle end, all SDRAM banks are precharged (idle).
- **DQM Data I/O Mask function.** This signal is asserted during a precharge command or when a burst stop command interrupts a burst write. When asserted during a write cycle, this signal interrupts and disables the write operation immediately.
- **SDCTL Register.** IOP register that contains programmable SDRAM control and configuration parameters that support different vendor's timing and power-up sequence requirements.
- **Mode Register.** The SDRAM's configuration register that contains user-defined parameters corresponding to the processor's `SDCTL` register. After initial power-up and before executing a read or write command, the application must program the `MODE` register.
- **Page Size.** The size, in words, of the SDRAM's page. The processor supports 2048-, 1024-, 512-, and 256-word page sizes. Page size is a programmable option in the `SDCTL` register.
- **Precharge Command.** Precharges an active bank.
- **SDRDIV Programmable Refresh Counter**. An IOP register containing a refresh counter value. The clock supplied to the SDRAM can vary between 20 and 100 MHz. This counter enables applications to coordinate `CLK` rate with the SDRAM's required refresh rate.
- **Self-Refresh.** The SDRAM's internal timer initiates automatic refresh cycles periodically, without external control input. This command places the SDRAM device in a low-power mode. Self-refresh is a programmable option in the `SDCTL` register.

- **t_{RAS}**. Active Command time. Required delay between issuing an activate command and issuing a precharge command. A vendor-specific value. This option is programmable in the SDCTL register.
- **t_{RC}**. Bank Cycle time. The required delay between successive Bank Activate commands to the same bank. This vendor-specific value is defined as: $t_{RC} = t_{RP} + t_{RAS}$.

  The processor fixes the value of this parameter, so it is a non-programmable option.
- **t_{RCD}**. $\overline{\text{RAS}}$ to $\overline{\text{CAS}}$ delay. The required delay between a ACT command and the start of the first read or write operation. This vendor-specific value is programmable in SDCTL.
- **t_{RP}**. Precharge time. Required delay between issuing a precharge command and issuing an activate command. This vendor-specific value is programmable in SDCTL.

Figure 8-1 shows the SDRAM controller's interface between the internal SHARC core and the external SDRAM device. (Note that in full instruction with no pack mode, the data bus extends to 48 bits, DATA47:00.)

The ADSP-21161 processor normally generates an external memory address, which then asserts the corresponding $\overline{\text{MSx}}$ select, along with $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes. These control signals are intercepted by the SDRAM controller. The memory access to SDRAM is based on the mapping of the addresses and memory selects. The configuration is programmed in the SDCTL register. The SDRAM controller can hold off the processor core or I/O processor with an internally connected acknowledge signal (ACK), as determined by refresh, nonsequential access, or page miss latency overhead.

The SDRAM controller provides a glueless interconnection between the SDRAM control, address, and data pins and the processor's internal Harvard Architecture busses. The internal 32-bit address bus is multiplexed by the SDRAM controller to generate the corresponding chip select, row address, column address, and bank select signals to the SDRAM.

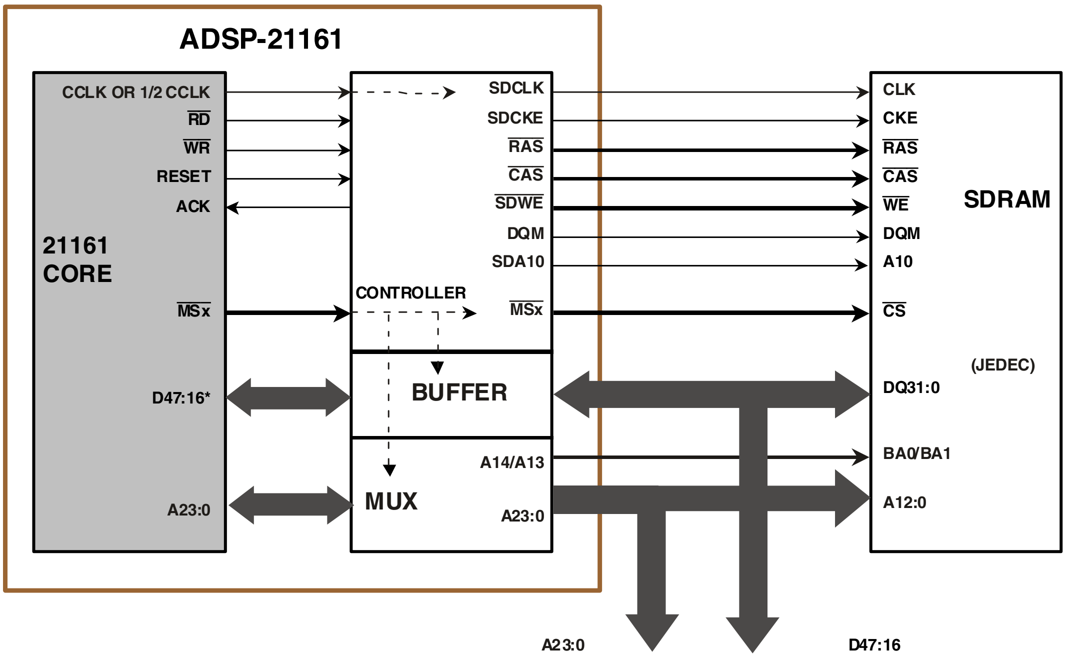


Figure 8-1. SDRAM Controller Interface

Figure 8-2 shows a block diagram of the ADSP-21161 processor's SDRAM interface to four 8-bit SDRAMs. In this single processor example, the SDRAM interface connects to four 1M x 8 x2 (2M x 8) SDRAM devices to use 2M of 32-bit words. The same address and control bus communicates to all four SDRAM devices. The following connections are made:

- SDCKE connects to the CKE of the SDRAM devices
- SDCLK0 SDRAM clock connects to the CLK pins
- $\overline{\text{SDWE}}$ connects to all $\overline{\text{WE}}$
- $\overline{\text{MSx}}$ pin connects to all chip selects ($\overline{\text{CS}}$)
- All $\overline{\text{CAS}}$, $\overline{\text{RAS}}$, and DQM signals are connected together between the processor and all of the SDRAM devices

Notice that the data bus shows the processor's default bus width, DATA[47:16]. For full non-packed instruction execution mode, the data bus can be extended to DATA[47:0] with the use of available disabled link port data pins. The A[10] pin of all SDRAM devices are connected to a separate SDA10 pin on the processor to allow the SDRAM controller to retain control of all SDRAMs for any non-SDRAM accesses during host bus requests.

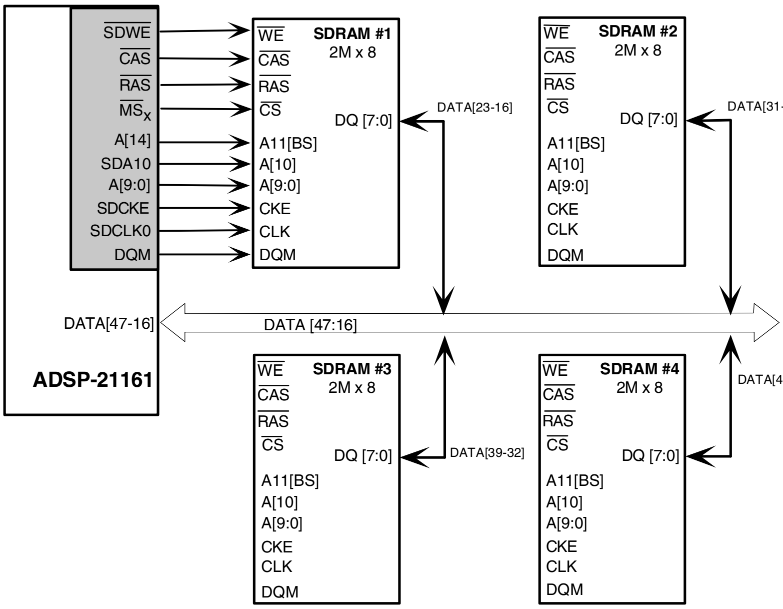


Figure 8-2. ADSP-21161 Processor's Block Diagram

SDRAM Pin Connections

Table 8-1 describes the ADSP-21161 processor SDRAM controller pins and the connections for each pin. The pins are defined as Input (I), Output (O), Synchronous (S), or High Impedance (T).

Table 8-1. SDRAM Pin Connections by Type

Pin	Type	Description
$\overline{\text{CAS}}$	I/0/T	SDRAM Column Address Select pin. Connect to SDRAM's CAS buffer pin.
DQM	0/T	SDRAM Data Mask pin. Connect to SDRAM's DQM buffer pin.

Table 8-1. SDRAM Pin Connections by Type (Cont'd)

Pin	Type	Description
$\overline{\text{MSx}}$	0/T	Memory select pin of external memory bank configured for SDRAM. Connect to SDRAM's CS (Chip Select) pin.
$\overline{\text{RAS}}$	I/0/T	SDRAM Row Address Select pin. Connect to SDRAM's RAS pin.
SDA10	0/T	SDRAM A10 pin. SDRAM interface uses this pin to retain control of the SDRAM device during host bus requests. Connect to SDRAM's A10 pin.
SDCKE	I/0/T	SDRAM Clock Enable pin. Connect to SDRAM's CKE (Clock Enable) pin.
SDCLK0	I/0/S/T	SDRAM SDCLKO output pin. Connect to the SDRAM's CLK pin.
SDCLK1	0/S/T	SDRAM SDCLK1 output pin. Connect to the SDRAM's CLK pin.
$\overline{\text{SDWE}}$	I/0/T	SDRAM Write Enable pin. Connect to SDRAM's WE or W buffer pin.

SDRAM Timing Specifications

To support key timing requirements and power up sequences for different SDRAM vendors, the ADSP-21161 processor provides programmability for `tRAS`, `tRP`, `tRCD`, and a power up sequence mode.

The $\overline{\texttt{CAS}}$ latency should be programmed in the `SDCTL` register based on the frequency of the operation. Refer to the SDRAM data sheet of the vendor for more details.

For other parameters, the controller assumes:

Bank Cycle Time, `tRC = tRAS + tRP`

SDRAM Control Register (SDCTL)

SDRAMs are available from several vendors, including IBM, Micron Electronics, Toshiba, Samsung Electronics, and NEC. Each vendor has different SDRAM product requirements for the power-up sequence and the timing parameters -tRAS (ACT to PRE command delay), tRCD and tRP (PRE to ACT command delay). Use only SDRAMS that comply with Joint Electronic Device Engineering Council (JEDEC) specifications. In order to support multiple vendors, the ADSP-21161 processor SDCTL register can be programmed to meet these requirements. The SDCTL register is an I/O processor register which does not support bitwise operations.

Figure 8-3 provides bit descriptions for the SDRAM Register. Table A-22 on page A-73 provides bit descriptions.

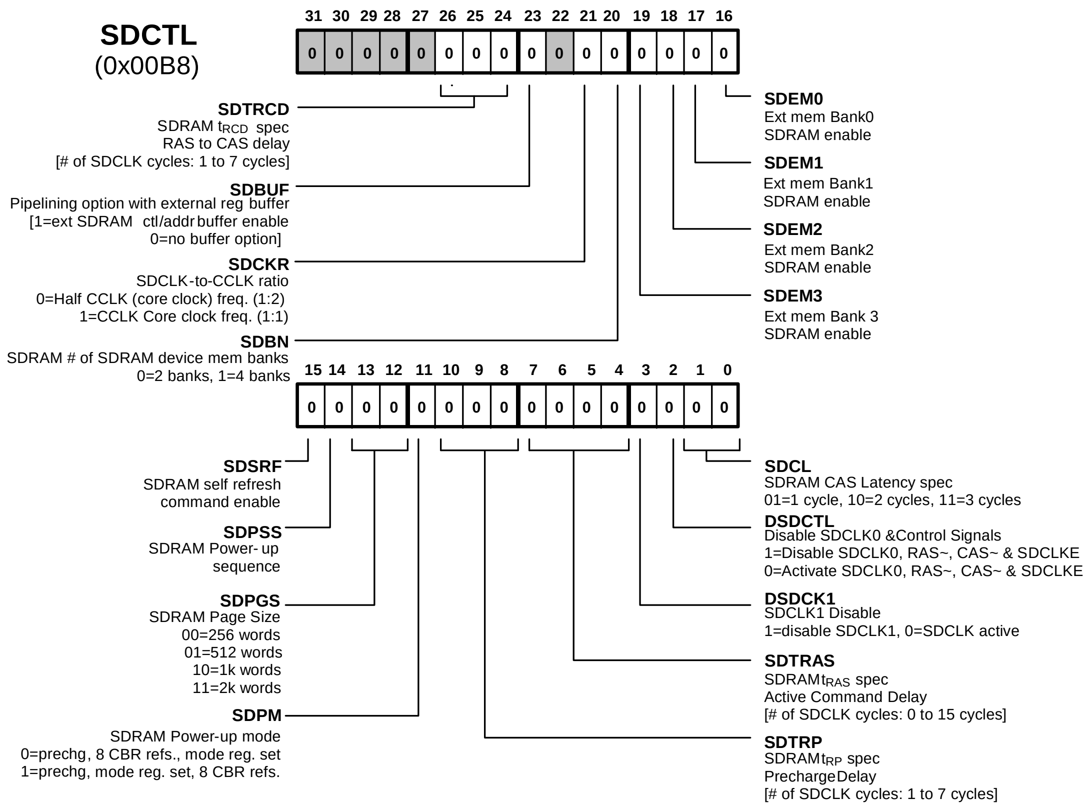


Figure 8-3. SDCTL Register

SDRAM Configuration for Runtime

The ADSP-21161 processor supports 16 Mbits, 64 Mbits, 128 Mbits, and 256 Mbits SDRAM devices with 4-bit, 8-bit, 16-bit, and 32-bit configurations. Page sizes of 256, 512, 1024, and 2048 words are supported in the available the densities and configurations mentioned above. Each external memory bank has address space of 64 Mwords for SDRAMs.

The `SDCTL` register of the ADSP-21161 processor stores the configuration information of the SDRAM interface. Writing configuration parameters initiates commands to the SDRAM that take effect immediately.

Before starting the SDRAM powerup sequence, complete the following steps:

1. Write to the `WAIT` register to set the waitstates to zero (`EBxWS=000`) for each bank that has SDRAM mapped to it.
2. Set the `SDRDIV` register at initial power-up. In the `SDRDIV` register, a memory-mapped IOP register, configure the value for the SDRAM refresh counter.
3. Write all of the SDRAM configuration parameter values to the `SDCTL` register.

When the SDRAM controller is programmed with the register buffer option enabled, do not perform non-SDRAM write accesses to external memory until the power-up sequence is completed by the SDRAM controller. External memory non-SDRAM writes do not function correctly whenever the SDRAM controller is configured for `SDBUF`=1 (register buffering) option and the power up sequence has not yet been completed by the SDRAM controller. The `MRS` command that is applied by the SDRAM controller conflicts with the non-SDRAM write access started by either the core or DMA controller.

In the `SDCTL` register, set the parameter bits as follows:

- Set the SDRAM clock enables (`DSDCTL` and `DSDCK1`).
- Select the number of banks that the SDRAM contains (`SDBN`).
- Select the external memory banks configured for and connected to an SDRAM (`SDEMx`).
- Set the SDRAM buffering option (`SDBUF`).

- Select the CAS latency value (SDCL).
- Select the SDRAM page size (SDPGS).
- Select the SDRAM power-up mode (SDPM).
- Start the SDRAM power-up sequence (SDPSS).
- Start SDRAM self-refresh mode (SDSRF).
- Set the Active Command Delay (SDTRAS).
- Set the precharge delay (SDTRP).
- Set the $\overline{\text{RAS}}$-to-$\overline{\text{CAS}}$ delay (SDTRCD).
- Set the SDCLK to Core Clock Ratio (SDCKR).

In systems where several SDRAM devices are connected in parallel, buffering may be required to meet overall system timing requirements. The ADSP-21161 processor supports the pipelining of the address and control signals to enable buffering between ADSP-21161 processor and SDRAM. The pipeline bit (SDBUF) in the SDCTL register enables this mode. When this bit is set, the data for write accesses are delayed by one cycle, allowing the address and controls to be externally latched. In read accesses, data is sampled by ADSP-21161 processor one cycle later. To support the higher clock load requirements, two SDCLK pins are provided to eliminate the need for off-chip clock buffers. An option is provided in the SDCTL register (bits 2 and 3) to allow the SDRAM controller to three-state one or *both* the SDCLK pins. The SDCKR bit in the control register can be used to set the SDCLK to core clock ratio. The interface can run at full core clock frequency or at half the core clock frequency, depending upon the setting for this bit.

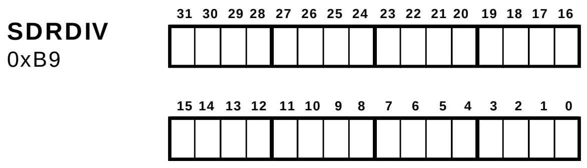
Setting the Refresh Counter Value (SDRDIV)

Since the clock supplied to the SDRAM can vary between 20 MHz and 100 MHz, the processor provides a programmable refresh counter (SDRDIV) to coordinate the supplied clock rate with the SDRAM device's required refresh rate.

Write to SDRDIV the delay, in a number of clock cycles, that must occur between consecutive refresh commands.

Write the delay value to the SDRDIV register before writing the SDRAM parameter values to the SDCTL register.

To calculate the value of the refresh counter for which to program the SDRDIV register, use the equation shown in Figure 8-4.



$$\text{SDRDIV} = \frac{f_{CCLK}}{\text{SDRAM refresh rate cycle}} - \text{CL} - \text{tRP} - 5$$

Figure 8-4. SDRDIV Register and Calculation

SDCLK is 1x CCLK or 2x CCLK, as determined by the SDCKR bit *and* SDCTL *register.*

Where:

f_{CCLK} = CLKCFG × f_{CLKOUT}
CL = CAS latency programmed into the SDCTL register
tRP = tRP specification programmed in the SDCTL register

CLK_CFG = 2 for 2:1 CCLK-to-CLKOUT clock ratio
= 3 for 3:1 CCLK-to-CLKOUT clock ratio
= 4 for 4:1 CCLK-to-CLKOUT clock ratio

CCLK is defined as the internal core-clock frequency. CLKOUT is 1xCLKIN or 2xCLKIN, depending on whether $\overline{\text{CLKDBL}}$ is tied high or low during $\overline{\text{RESET}}$. The signals SDCLK0 and SDCLK1 can operate at either 1xCCLK or 1/2 CCLK, as determined by the SDCKR in the SDCTL register.

For example, for an IBM SDRAM with:

Reference rate = 4096 cycles/64ms

CLKIN = 25 MHz

$\overline{\text{CLKDBL}}$ enabled

Therefore, CLKOUT = 50 MHz

CLK_CFG = 2, for 2:1 PLL ratio

CL = 2

tRP = 2

The equation yields:

$$\text{SDRDIV} = \left\lfloor 2\text{x}\frac{50\text{x}10^6}{4096\frac{1}{64\text{x}10^{-3}}}\right\rfloor - 2 - 2 - 5 = 1554(\text{decimal}) = 0\text{x}612$$

Setting the SDRAM Clock Enables

Systems with several SDRAM devices connected in parallel require buffering between the processor and multiple SDRAM devices to reduce capacitive loading. Buffering, however, may also generate increased clock loads.

To meet higher clock load requirements, the processor provides two SDRAM clock control pins, SDCLK0 and SDCLK1. These pins eliminate the need for off-chip clock buffers.

The DSDCTL and DSDCK1 in the SDCTL register provide control for the SDRAM clock control pins. The DSDCTL bit, if set (=1), enables high impedance for all of the SDRAM control pins (DQM, $\overline{\text{CAS}}$, $\overline{\text{RAS}}$, SDWE, and SDCKE) and the SDCLK0 pin. The DSDCTL bit, if cleared (=0), disables all SDRAM control pins.

The DSDCK1 bit, if set (=1), enables the SDCLK1 pin and places it into a high impedance state only. The DSDCK1 bit, if cleared (=0), disables SDCLK1.

If your system does not use SDRAM, set both DSDCTL and DSDCK1 to 1.

If your system uses SDRAM, but the clock load is minimal, set DSDCTL to 0 and DSDCK1 to 1 This setting enables the SDCLK0 pin and all related SDRAM control pins, but disables the second clock pin SDCLK1.

If your system uses SDRAM and has a heavy clock load such as a system using registered buffers and eight 4-bit SDRAMs to get 32-bit data, set both DSDCTL and DSDCK1 to 0. This setting enables SDCLK0, SDCLK1, and all SDRAM control pins. In this configuration, SDCLK0 and SDCLK1 can each share half of the clock load.

Setting the Number of SDRAM Banks (SDBN)

The SDBN bit defines the number of banks in the SDRAM device. The SDRAM controller uses this value and the value assigned to the SDPGS (page size) bit to map the address bits on the processor's internal 32-bit address (DMA/PMA/EPA) bus into SDRAM column address, row address, and bank select address. The SDBN bits in the SDCTL register select the number of banks the SDRAM as follows: 0 = 2 banks, 1 = 4 banks.

Setting the External Memory Bank (SDEMx)

The `SDCTL` register can be programmed to select the external memory banks that have SDRAM devices by using the `SDEMx` bits. For example, if external memory banks 1 and 3 have SDRAMs, `SDEM1` and `SDEM3` bits are written with 1. However the controller tracks only the previously accessed page/bank.

When using SDRAM, connect its $\overline{\texttt{CS}}$ line to any of the processor's external memory banks $\overline{\texttt{MS}}\texttt{3-0}$. In the `SDCTL` register, configure that bank for SDRAM operation.

Program a zero (0) wait state for the external memory bank to which the SDRAM device maps by setting `EBxWS` to 000 in the `WAIT` register.
Do not use external handshake mode DMA on the external memory bank mapped to an SDRAM device.

The `SDEMx` bits in the `SDCTL` register configure the processor's external memory banks for SDRAM operation as follows:

`SDEM [0-3] = 0000`	bits 16-19, No SDRAM enabled
`SDEM0 = 1`	Bank 0 SDRAM Enable
`SDEM1 = 1`	Bank 1 SDRAM Enable
`SDEM2 = 1`	Bank 2 SDRAM Enable
`SDEM3 = 1`	Bank 3 SDRAM Enable

Setting the SDRAM Buffering Option (SDBUF)

Systems that use several SDRAM devices connected in parallel may require buffering between the processor and multiple SDRAM devices in order to meet overall system timing requirements.

To meet such timing requirements and enable intermediary buffering, the processor supports pipelining of SDRAM address and control signals.

The pipeline bit `SDBUF` (bit 23) in the `SDCTL` register enables this mode:

`SDBUF = 0`	Disable pipelining
`SDBUF = 1`	Enable pipelining

When `SDBUF` is set (=1), the SDRAM controller delays the data in write accesses by one cycle, enabling the processor to latch the address and controls externally. In read accesses, the SDRAM controller samples data one cycle later.

Figure 8-5 shows another single processor example in which the SDRAM interface connects to multiple banks of SDRAM to provide 512 M of SDRAM in a 4-bit I/O configuration. This configuration results in 16 M x 32-bit words. In this example, `OxA` and `OxB` output from the registered buffers are the same signal, but are buffered separately. In the registered buffers, a delay of one clock cycle occurs between the input (Ix) and its corresponding output (`OxA` or `OxB`).

Selecting the CAS Latency Value (SDCL)

The CAS latency value defines the delay, in number of clock cycles, between the time that the SDRAM detects the read command and the time that it provides the data at its output pins. This parameter facilitates matching the SDRAM operation with the processor's ability to latch the data output.

CAS latency does not apply to write cycles.

The `SDCL` bits in the `SDCTL` register select the CAS latency value as follows: 01 = 1 clock cycle, 10 = 2 clock cycles and 11= 3 clock cycles.

Generally, the frequency of the operation determines the value of the CAS latency. For more details, see the SDRAM device documentation.

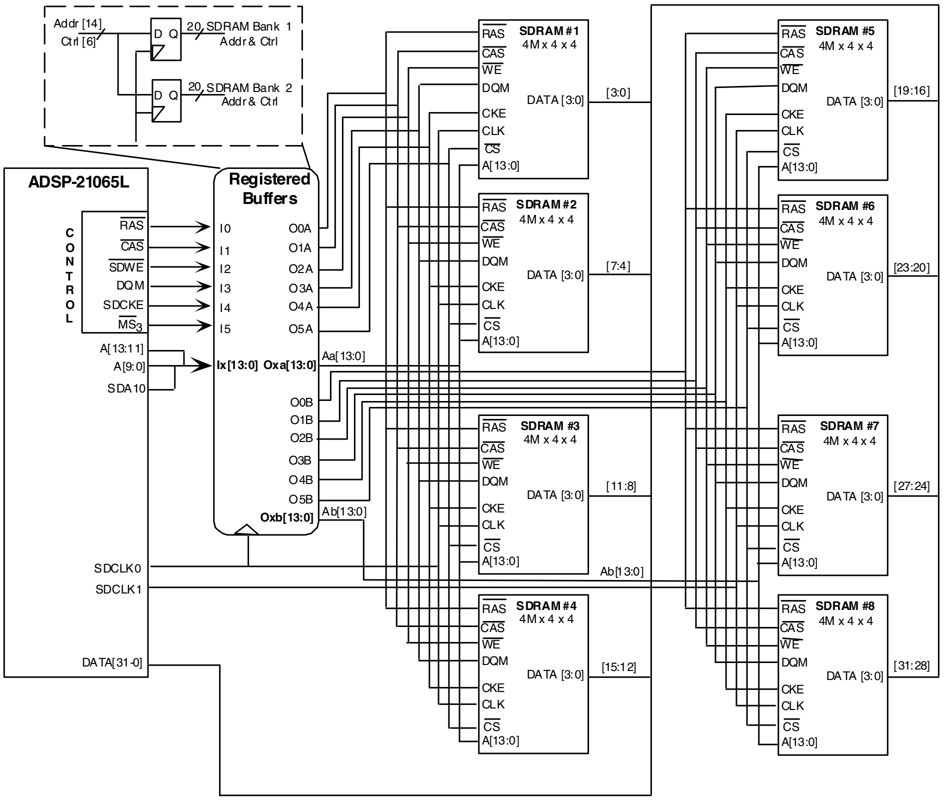


Figure 8-5. Uniprocessor System With Multiple SDRAM Devices

Selecting the SDRAM Page Size (SDPGS)

The processor's SDRAM controller SDPGS bit defines for the page size, in number of words, of the SDRAM's banks. The SDRAM controller uses this value and the value assigned to the SDBN (number of banks) bit to map the address bits on the processor's internal 32-bit address (DMA/PMA/EPA) bus into SDRAM column address, row address, and bank select address.

Page length depends on the I/O organization and column addressing of the SDRAM's internal banks. For example, a 16 Mbits SDRAM organized as 2 M x 4 I/O x 2 banks has a page size of 1024 words.

The SDPGS bits (bits 12 and 13) in the SDCTL register select the SDRAM page length: 00= 256 words, 01 = 512 words, 10= 1024 words and 11 = 2048 words.

Setting the SDRAM Power-Up Mode (SDPM)

To avoid unpredictable start-up modes, SDRAM devices must follow a specific initialization sequence during power up. The processor provides two commonly used power-up options.

The SDPM bit (bit 11) in the SDCTL register selects the SDRAM power-up mode. When the SDPM bit is cleared (=0), the SDRAM controller sequentially issues: a PRE command, eight CBR refresh cycles, and an MRS (Mode Register Set) command. When the SDPM bit is set (=1), the SDRAM controller issues, in this order: a PRE command, an MRS (Mode Register Set) command, and eight CBR refresh cycles.

For details, see the SDRAM device documentation.

Starting the SDRAM Power-Up Sequence (SDPSS)

Before starting the power-up sequence, write to the SDCTL register to configure the SDRAM parameters. Be sure to write to all the register bits, regardless of the number of parameter values that do not change.

To start the SDRAM power-up sequence, write 1 to the SDPSS bit (bit 14) in the SDCTL register. The SDPSS bit always reads as zero (0). The initialization sequence executed during power-up depends on the value of the SDPM bit.

ⓘ Initialize the SDRDIV register before the ADSP-21161 processor starts the SDRAM power-up sequence. After power up, make sure that the processor waits one cycle before writing the SDCTL register to issue another SDRAM command.

For more details, see the SDRAM device documentation.

Starting Self-Refresh Mode (SDSRF)

The processor supports SDRAM self-refresh mode. In self-refresh mode, the SDRAM performs refresh operations internally, without external control, which reduces the SDRAM's power consumption.

The SDSRF bit (bit 15) in the SDCTL register enables and disables the self-refresh option:

SDSRF = 0	Disable self-refresh mode
SDSRF = 1	Enable self-refresh mode

When SDSRF is set (=1), the processor's SDRAM controller issues a SREF command to the SDRAM device or devices, putting them into self-refresh mode immediately. For details, see "Self Refresh Command (SREF)" on page 8-39.

Selecting the Active Command Delay (SDTRAS)

The tRAS value (Active Command Delay) defines the required delay, in number of clock cycles, between the time the SDRAM controller issues an ACT command and the time it issues a PRE command.

The SDTRAS bits (bits 4, 5, 6, and 7) in the SDCTL register select the tRAS value. For example:

SDTRAS=0001	1 clock cycle
SDTRAS=0010	2 clock cycles
SDTRAS=0111	7 clock cycles
SDTRAS=1111	15 clock cycles

For more details, see the SDRAM device documentation.

Selecting the Precharge Delay (SDTRP)

The tRP value (precharge delay) defines the required delay, in number of clock cycles, between the time the SDRAM controller issues a PRE command and the time it issues an ACT command.

The SDTRP bits (bits 8, 9, and 10) in the SDCTL register select the tRP value. For example:

SDTRP = 001	1 clock cycle
SDTRP = 010	2 clock cycles
SDTRP = 111	7 clock cycles

Selecting the RAS-to-CAS Delay (SDTRCD)

The vendor-specific SDRAM value tRCD defines the required delay in number of clock cycles between an ACT command and the start of the first read or write operation. The SDTRCD[2:0] bits in the SDCTL register select the tRCD ($\overline{\text{RAS}}$ to $\overline{\text{CAS}}$ delay) value as follows: 001= 1 clock cycle, 010= 2 clock cycles, and 111= 7 clock cycles. For more details, see the SDRAM device documentation.

SDTRP, SDTRAS, and SDTRCD settings represent the number of core clock (CCLK) cycles.

SDRAM Controller Standard Operation

The ADSP-21161 processor SDRAM controller uses a burst length one for page read/write operations. Burst length determines the maximum number of column locations that can be accessed for a given read or write operation.

The ADSP-21161 processor supports burst length one mode. This does not have an adverse impact on the throughput as compared to burst lengths of 2, 4, 8 and full page. Instead of applying the first address and continuing access to data on successive clocks (during which the controller drives the `NOP` command), the ADSP-21161 processor SDRAM controller applies the command at every cycle continuously accessing the data.

Table 8-2 lists the data throughput rates for the processor's core or DMA read/write accesses to SDRAM. The following assumptions are made for the information in this table:

- SDCLK is running at core clock speed (`SDCKR =1`)
- CAS latency = 2 cycles (`SDCL=2`)
- No SDRAM buffering (`SDBUF=0`)
- Precharge (`tRP`) = 2 cycles (`SDTRP=2`)
- Active command time (`tRAS`) = 3 cycles (`SDTRAS=3`)
- `tRCD` = 2 cycles (`SDTRCD=2`)

Understanding DAG and DMA Operation

For either core-driven accesses via the DAGs or DMA data transfers to and from SDRAM, one full page can be accessed at full throughput if the data address generator or external address increment is equal to one. If the

Table 8-2. Throughput for Core or DMA Read/Write Operations

Accesses	Operations	Page	Throughput per CCLK (32-bit words)[1, 2]
Sequential, uninterrupted	Read	Same	1 word/1 cycle
Sequential, uninterrupted	Write	Same	1 word/1 cycle
Nonsequential, Uninterrupted	Read	Same	1 word/5 cycles (CL + 3)
Nonsequential, Uninterrupted	Write	Same	1 word/1 cycle
Both	Alternating read/write	Same	Average rate = 3 cycles per word (reads = 5 cycles, writes = 1 cycle)
Nonsequential	Reads	Different	1 word/10 cycles (t_{RP} + CL + t_{RCD} + 4)
Nonsequential	Writes	Different	1 word/7 cycles (t_{RP} + t_{RCD} + 3)
Auto refresh before read	Reads	Different	1 word/15 cycles ($2t_{RP}$ + t_{RAS} + CL + t_{RCD} + 4)
Auto refresh before write	Writes	Different	1 word/11 cycles ($2t_{RP}$ + t_{RAS} + t_{RCD} + 2)

1 When executing 48-bit packed instructions from 32-, 16-, or 8-bit SDRAM memories:
- Add one clock cycle to the throughput value or to the average access rate for 32-bit wide SDRAM
- Add three clock cycles to the throughput value or to the average access rate for 16-bit wide SDRAM
- Add six clock cycles to the throughput value or to the average access rate for 8-bit wide SDRAM

2 With SDRAM buffering enabled (SBUF=1), replace any instance of (CL) with (CL + 1).

modify register or external address register is greater than a value of 1, then one full page can be written at full throughput, but reads increase the amount of processing time required.

Whenever a page miss happens, the SDRAM controller executes a PRE command followed by a bank activate command before executing a read/write command. For SDRAM reads, a latency (equal to $\overline{\text{CAS}}$ latency) exists from the start of the read command until data is available from the SDRAM. For the first read in a sequence of reads, the latency always exists. Subsequent reads will not have latency if the address is sequential and uninterrupted.

A fresh access to SDRAM always aligns to the CLKIN rising edge. So, interrupted access to SDRAM incur the overhead of additional cycles, depending on the CLK CFG setting. For example, WRT-NOP-WRT-NOP-WRT has a 6-cycle overhead for CLK-CFG-2:1 and SDCKR=1. Every write in the above sequence starts at the rising edge of CLKIN, and two core cycles transpire in every CLKIN. The last WRT completes in the first core cycle of the third CLKIN cycle (which is the ninth core cycle). If the three writes had been consecutive, the third write would be over by the third core cycle of the first CLKIN. As a result, the writes complete six core clock cycles later.

Programmable refresh counter provides that can be used to set up a count, depending on the required refresh rate and the clock rate used. The refresh count is specified in the SDRDIV, a memory mapped IOP register. For more information on SDRDIV, see "Setting the Refresh Counter Value (SDRDIV)" on page 8-13.

Multiprocessing Operation

In a multiprocessing environment, the SDRAM is shared among two or more ADSP-21161 processors. SDRAM input signals (including clock) are always driven by the bus master. The slave processors track the commands that the master processor issues to the SDRAM. This feature or function helps to synchronize the SDRAM refresh counters and to prevent needless refreshing operations. A simplified multiprocessing is shown in Figure 8-6.

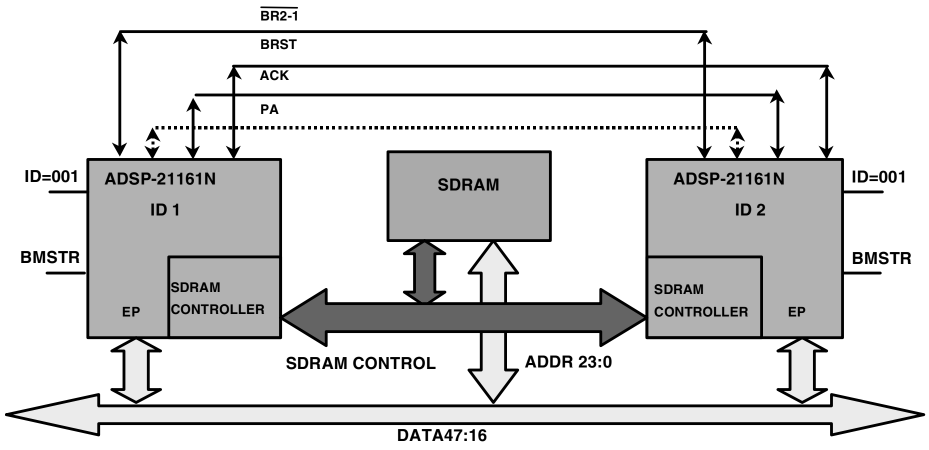


Figure 8-6. Multiprocessing: Dual Processor System Example

When a ADSP-21161 processor receives the bus mastership, it executes a `PRE` command prior to the first access to SDRAM. This occurs only if the previous master had accessed the SDRAM. In the user application code, the `SDCTL` and `SDRDIV` registers of both ADSP-21161 processors must be initialized to the same value. If there is no SDRAM used in the system (as indicated in `SDCTL`), then the bus transition process is the same as in the ADSP-21160.

Accessing SDRAM

To access SDRAM, the SDRAM controller multiplexes the internal 32-bit non-multiplexed address into a row address, a column address, and a bank select address for the SDRAM device, as shown in Figure 8-7. Lower bits are mapped into the column, next bit/bits are mapped into the bank select, and remaining bits are mapped into the row. This mapping is based on the page size and the number of banks in SDRAM (entered into the `SDCTL` register).

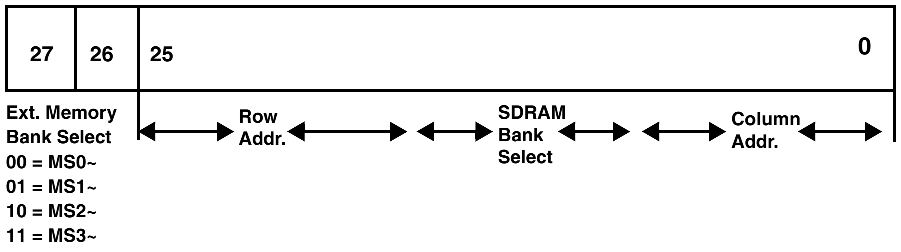


Figure 8-7. Multiplexed 32-Bit SDRAM Address

Based on the values programmed in the SDCTL register for page size and number of SDRAM banks, the SDRAM controller maps bits as follows:

- the lower ADDR bits into the column address
- the next bit or bits into the bank select address
- the remaining higher order bits into the row address

The following tables show how the SDRAM controller maps the SDRAM address bits on the processor's internal address bus to its external address pins that connect to the SDRAM. The internal and external address bus pins in the tables are defined as follows:

EA = External address pins
IA = Internal address bus

For 16 M SDRAMs, A11 is the Bank Select pin. When using a 16 M SDRAM, connect the processor's A14 pin to the SDRAM's A11 pin.

Address Mapping for SDRAM

Table 8-3 through Table 8-7 provide information needed for interfacing to various SDRAMs.

Table 8-3. SDRAM Size = 16 Mbit

16 Mbit SDRAM (Page Size x No. of Banks)	Column Address (Page Access)	Bank Select	Row Address (Bank Activate)
256 x 2	IA[7:0]=>EA[7:0]	IA[8]=>EA[14]	IA[19:9]=>EA[10:0]
512 x 2	IA[8:0]=>EA[8:0]	IA[9]=>EA[14]	IA[20:10]=>EA[10:0]
1024 x 2	IA[9:0] =>EA[9:0]	IA[10]=>EA[14]	IA[21:11]=>EA[10:0]

Table 8-4. SDRAM Size = 64 Mbit

64 Mbit SDRAM (Page Size x No. of Banks)	Column Address	Bank Select	Row Address
256 x 2	IA[7:0]=> EA[7:0]	IA[8] =>EA[14]	IA[21:9]=>EA[12:0]
512 x 2	IA[8:0]=>EA[8:0]	IA[9]=>EA[14]	IA[22:10]=>EA[12:0]
1024 x 2	IA[9:0]=>EA[9:0]	IA[10]=>EA[14]	IA[23:11]=>EA[12:0]
256 x 4	IA[7:0]=>EA[7:0]	IA[9:8]=>EA[14:13]	IA[21:10]=>EA[11:0]
512 x 4	IA[8:0]=>EA[8:0]	IA[10:9]=>EA[14:13]	IA[22:11]=>EA[11:0]
1024 x 4	IA[9:0]=>EA[9:0]	IA[11:10]=>EA[14:13]	IA[23:12]=>EA[11:0]

Table 8-5. SDRAM Size = 128 Mbits

128 Mbit SDRAM (Page Size x No. of Banks)	Column Address	Bank Select	Row Address
256 x 4	IA[7:0]=>EA[7:0]	IA[9:8]=>EA[14:13]	IA[21:10]=>EA[11], SDA10, EA [9:0]
512 x 4	IA[8:0]=>EA[8:0]	IA[10:9]=>EA[14:13]	IA[22:11]=>EA[11:0]
1024 x 4	IA[9:0]=>EA[9:0]	IA[11:10]=>EA[14:13]	IA[23:12]=>EA[11:0]
2048 x 4	IA[10:0]=>EA[11, 9:0]	IA[12:11]=>EA[14:13]	IA[24:13]=>EA[11:0]

Table 8-6. SDRAM Size = 256 Mbit

256 Mbit SDRAM (Page Size x No. of Banks)	Column Address	Bank Select	Row Address
512 x 4	IA[8:0]=>EA[8:0]	IA[10:9]=>EA[14:13]	IA[23:11]=>EA[12:0]
1024 x 4	IA[9:0]=>EA[9:0]	IA[11:10]=>EA[14:13]	IA[24:12]=>EA[12:0]
2048 x 4	IA[10:0]=>EA[11, 9:0]	IA[12:11]=>EA[14:13]	IA[25:13]=>EA[12:0]

Table 8-7. Address Ranges for Various SDRAM Device Densities and Page Size Combinations

SDRAM Device Size		Page Size	Address Range
16 Mbit[1]	1Mx16	256	0 - 0x000F FFFF (1 Mwords)
	2Mx8	512	0 - 0x001F FFFF (2 Mwords)
	4Mx4	1024	0 - 0x003F FFFF (4 Mwords)
64 Mbit	2Mx32	512	0 - 0x001F FFFF (2 Mwords)
	4Mx16	256	0 - 0x003F FFFF (4 Mwords)
	8Mx8	512	0 - 0x007F FFFF (8 Mwords)
	16Mx4	1024	0 - 0x00FF FFFF (16 Mwords)

Table 8-7. Address Ranges for Various SDRAM Device Densities and Page Size Combinations (Cont'd)

SDRAM Device Size		Page Size	Address Range
128 Mbit [2]	4Mx32	1024	0 - 0x003F FFFF (4 Mwords)
	8Mx16	512	0 - 0x007F FFFF (8 Mwords)
	16Mx8	1024	0 - 0x00FF FFFF (16 Mwords)
	32Mx4	2048	0 - 0x01FF FFFF (32 Mwords)
256 Mbit	16Mx16	512	0 - 0x00FF FFFF (16 Mwords)
	32Mx8	1024	0 - 0x01FF FFFF (32 Mwords)
	64Mx4	2048	0 - 0x03FF FFFF (64 Mwords)

1 16M and 64M devices do not have a page size of 2048.
2 128M and 256M devices do not have a page size of 256.

Understanding DQM Operation

The processor's DQM (Data I/O Mask) pin is used only during the SDRAM powerup sequence and during a precharge command.

Executing a Parallel Refresh Command During Host Control

The ADSP-21161 processor's SDRAM interface includes a separate A10 pin (SDA10) to enable the controller to execute a parallel refresh command with any non-SDRAM access. This separate pin allows the SDRAM controller to precharge the SDRAM before it issues a refresh command.

Connecting this pin to the SDRAM's A10 line, instead of ADDR10 to precharge the SDRAM device, enables the processor to retain control of the SDRAM device while a host requests (using the $\overline{\text{HBR pin}}$) and controls the external ADDR23-0 bus. Figure 8-8 shows an example ADSP-21161 system containing both a host and SDRAM. During host bus requests, the processor still retains mastership of the control pins of the SDRAM ($\overline{\text{RAS}}$, $\overline{\text{CAS}}$,

$\overline{\mathtt{SDWE}}$, `SDCKE`, `SDCLK`, $\overline{\mathtt{MSx}}$ and `SDA10`) when the host assumes control of the system bus—$\overline{\mathtt{HBG}}$ is asserted. As a result, the single processor (or master processor in a multiprocessor system) can issue `REF` commands as required.

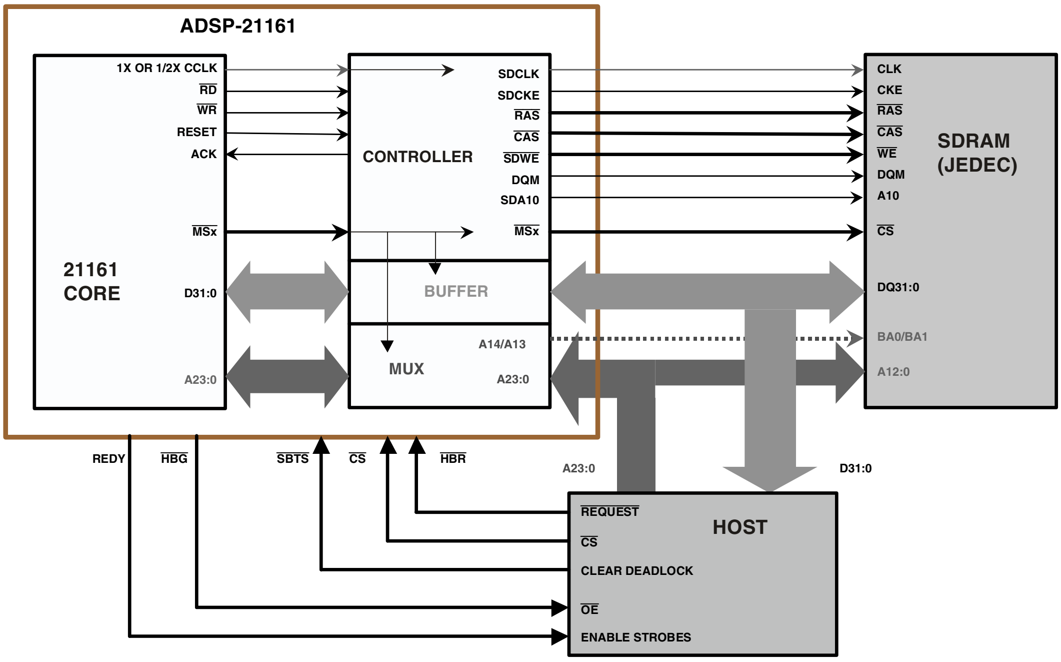


Figure 8-8. SDRAM Interface – Bus Slave

Powering Up After Reset

After reset, once the `SDCTL` register is written to in the user application code, the controller initiates the selected power-up sequence. The exact sequence is determined by `SDPM` bit of the `SDCTL` register. In a multiprocessing environment, the power-up sequence is initiated by any one of the ADSP-21161 processors. Note that a software reset does not reset the controller and does not re-initiate a power-up sequence.

Entering and Exiting Self-Refresh Mode

Writing 1 to the `SDSRF` bit in the `SDCTL` register causes the SDRAM controller to issue an `SREF` command to the SDRAM device.

During entry into Self refresh, make sure that no SDRAM accesses are occurring and that the SDRAM has stopped bursting out data.

Once the SDRAM device enters into self-refresh mode, the SDRAM controller resets the `SDSRF` bit in the `SDCTL` register. The `SDSRF` bit always reads as 0, regardless of a pending request. The SDRAM controller ignores other self-refresh requests (`SDSRF=1`) when the SDRAM device is already in self-refresh mode.

The application cannot clear the `SDSRF` bit (`SDSRF=0`) to cancel self-refresh mode. The SDRAM device exits self-refresh mode only when it receives a core or DMA access request from the SDRAM controller.

SDRAM Controller Commands

This section describes each command that the SDRAM controller uses to manage the SDRAM interface. These commands are transparent to applications.

A summary of the various commands used by the on chip controller for the SDRAM interface is as follows:

- **ACT (bank activate).** Activates a page in the required bank
- **MRS (mode register set).** Initializes the SDRAM operation parameters during the power-up sequence
- **PRE (precharge).** Precharges the active bank
- **Read/Write**

- **REF (refresh).** Causes the SDRAM to enter refresh mode and generate all addresses internally
- **SREF (self-refresh).** Places the SDRAM in self-refresh mode, in which it controls its refresh operations internally

Bank Activate (ACT) Command

A Bank Activate (`ACT`) command is required if the next data access is on a different page. The SDRAM controller executes a precharge (`PRE`) command followed by bank active (`ACT`) command to activate the page in the required bank. Only one bank is active at a time.

The SDRAM pin state during the `ACT` command is shown in Table 8-9.

Table 8-8. Pin State During ACT Command

Pin	**State**
$\overline{\text{MSx}}$	Low
$\overline{\text{CAS}}$	High
$\overline{\text{RAS}}$	Low
$\overline{\text{SDWE}}$	High
SDCKE	High

Mode Register Set (MRS)

Mode Register Set (`MRS`) is a part of the power up sequence. `MRS` initializes SDRAM operation parameters by using address bits `A0-A15` of the SDRAM as data input. An SDRAM power-up sequence is initiated by writing 1 to the `SDPSS` bit in `SDCTL` register. The exact power up sequence is determined by the `SDPM` bit of the `SDCTL` register.

MRS initializes the following SDRAM parameters:

- Burst length = 1, bits 2-0, hardwired to zero in ADSP-21161 processor
- Wrap type = sequential, bit 3, hardwired to zero in ADSP-21161 processor
- Ltmode = latency mode ($\overline{\text{CAS}}$ latency), bits 6-4, programmable in SDCTL
- Bits (14-7) always 0, hardwired in the ADSP-21161 processor

While executing mode register set command, the SDRAM controller sets the unused address pins to zero. During the two clock cycles following MRS, ADSP-21161 processor does not issue any other commands. The SDRAM pin state during the MRS command is shown in Table 8-10.

Table 8-9. Pin State During MRS Command

Pin	State
$\overline{\text{MSx}}$	Low
$\overline{\text{CAS}}$	Low
$\overline{\text{RAS}}$	Low
$\overline{\text{SDWE}}$	Low
SDCKE	High

Precharge Command (PRE)

The PRE command is issued to precharge the active bank. The SDRAM controller executes this command if the data to be accessed is located in a different bank or in a different page in the same bank. After power up, a PRE command is issued to the SDRAM device's banks.

The SDRAM pin state during the PRE command is shown in Table 8-10.

Table 8-10. Table 10. Pin State During PRE Command

Pin	State
$\overline{\text{MSx}}$	Low
$\overline{\text{CAS}}$	High
$\overline{\text{RAS}}$	Low
$\overline{\text{SDWE}}$	Low
SDCKE	High
SDA10	High

Read/Write Command

The SDRAM controller executes a Read/Write command if the next read/write data falls in the present (currently active) page.

In general, a Read interrupts a previous Read when the next access is a nonsequential address but a page miss does not occur. When a page miss does occur, the SDRAM controller precharges and activates (PRE and ACT commands) the SDRAM before issuing a Read or Write command. If the internal refresh counter (SDRDIV) asserts a refresh request, any new access is delayed until a refresh command is executed.

Read Commands

For the Read command, the $\overline{\text{CAS}}$, $\overline{\text{MSx}}$ and SDA10 are asserted low to enable the SDRAM to latch the column address. The start address is set according to the column address. The delay between Active and Read commands is determined by the tRCD parameter (see "SDRAM Timing Specifications" on page 8-8). Data is available after the tRCD and $\overline{\text{CAS}}$ latency requirements are met.

The SDRAM read timing is shown in Figure 8-9 and the pin state during the Read command is shown in Table 8-11.

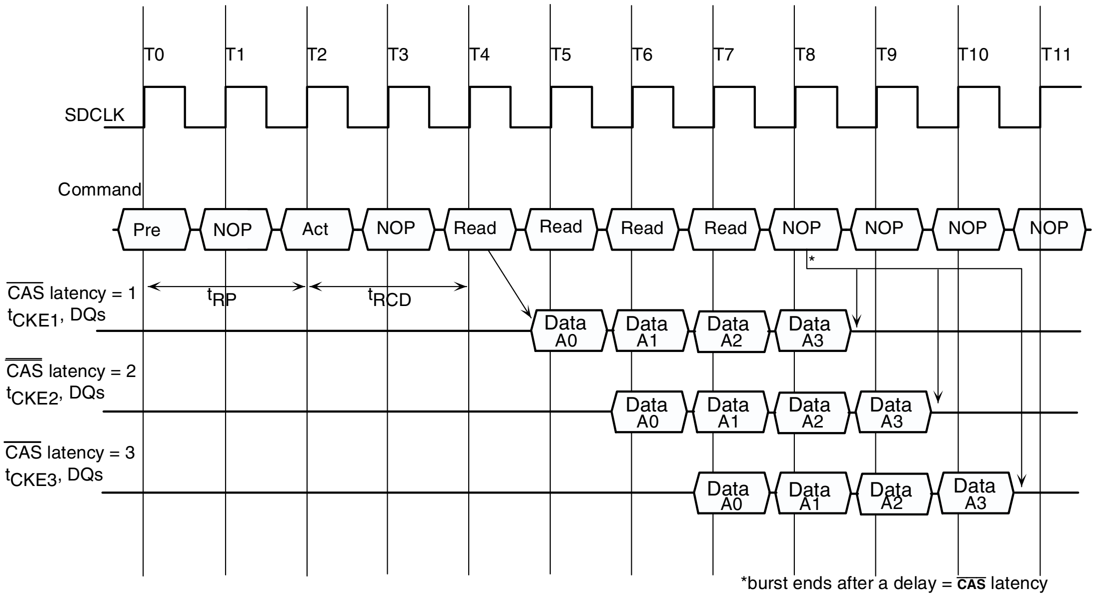


Figure 8-9. Read Timing Diagram

Table 8-11. Pin State During a Read Command

Pin	**State**
$\overline{\text{MSx}}$	Low
$\overline{\text{CAS}}$	Low
$\overline{\text{RAS}}$	High
$\overline{\text{SDWE}}$	High
SDCKE	High
SDA10	Low

Write Commands

For the write command, $\overline{\text{CAS}}$, $\overline{\text{MSx}}$, $\overline{\text{SDWE}}$, and SDA10 are asserted low to enable the SDRAM to latch the column address. Data is also asserted in the same cycle. The start address is set according to the column address. The write timing is shown in Figure 8-10

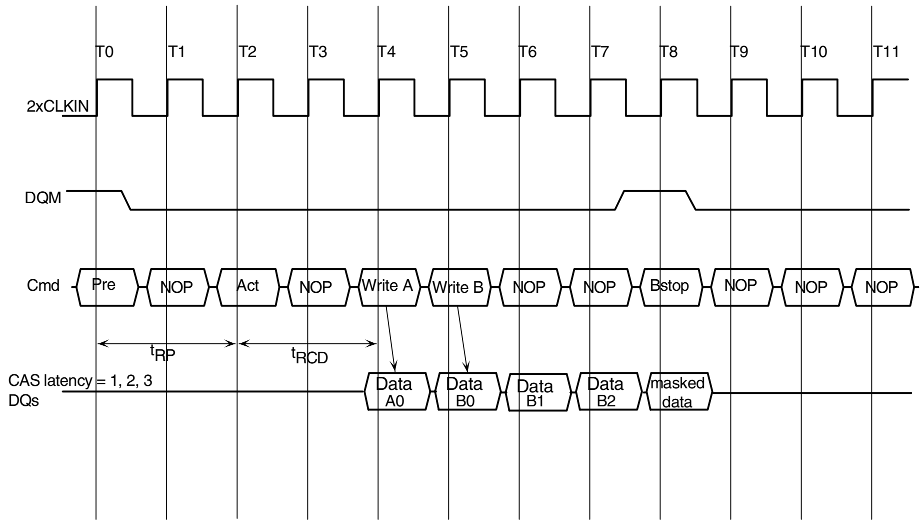


Figure 8-10. Write Timing Diagram

The SDRAM pin state during the Write command is shown in Table 8-12 below:

Table 8-12. Pin State During Write Command

Pin	State
$\overline{\text{MSx}}$	Low
$\overline{\text{CAS}}$	Low
$\overline{\text{RAS}}$	High
$\overline{\text{SDWE}}$	Low

Table 8-12. Pin State During Write Command (Cont'd)

Pin	State
SDCKE	High
SDA10	Low

DMA Transfers

In cases where a DMA channel is performing reads from SDRAM, the SDRAM controller issues a read command if at least one location is available in the external port DMA buffer (EPBx) FIFO. Whenever the FIFO is full, a NOP command is issued.

In cases where a DMA channel is performing writes to SDRAM, the SDRAM controller issues a write command if at least one word is available in the EPBx buffer. Whenever no data is available to write, an NOP command is issued.

Refresh (REF) Command

This command is a request to the SDRAM to perform a CBR ($\overline{\text{CAS}}$ before $\overline{\text{RAS}}$) transaction. REF causes all addresses to be generated internally in the SDRAM. This command is issued to all the external banks having SDRAMs as defined by the SDEM bits.

Before executing the REF command, the SDRAM controller executes a precharge (PRE) command to the active bank (after meeting tRAS min). The next active (ACT) command is given by the controller only after a minimum delay equal to tRC.

Setting the Delay Between Refresh Commands

The SDRDIV register in the ADSP-21161 processor is used to set the number of clock cycles between two REF commands. Program the SDRDIV register before writing to the SDCTL register. An internal CBR REF request is made to the SDRAM controller based on this refresh divisor value. The controller completes the present burst before servicing the refresh request. The master ADSP-21161 processor always performs the refresh command.

Understanding Multiprocessing Operation

In a multiprocessing environment, all ADSP-21161 processors share the SDRAM. While the ADSP-21161 processor bus master always drives SDRAM input signals (including the clock), the slave ADSP-21161 processors track the commands the master processor issues to the SDRAM. This tracking helps to synchronize the SDRAM refresh counters and to prevent needless refreshing operations.

Whenever a ADSP-21161 processor needs to transfer the bus mastership to other ADSP-21161 processor, it transfers the bus only after meeting `tRAS` min - 1 number of cycles for the presently active row. If the refresh timer makes a refresh request during this process, the present bus master executes a refresh command (after executing precharge command to SDRAM). The current bus master continues to hold the bus for `tRAS` min – 1 cycles before giving up the bus to the new bus master.

If the `REF` request arrives from the refresh counter during a bus transition cycle, the new bus master immediately issues a `REF` command. The new bus master becomes aware of this request because the refresh counter is running on all ADSP-21161 processors. The reloading of the refresh counter occurs synchronously on all processors, as the slaves watch the external SDRAM control pins to see when the refresh command is executed by the master. When a processor receives the bus mastership, it executes a `PRE` command prior to the first access to the SDRAM.

The current ADSP-21161 processor bus master retains mastership of the control pins of the SDRAM ($\overline{\text{RAS}}$, $\overline{\text{CAS}}$, $\overline{\text{SDWE}}$, `SDCKE`, `SDCLK`, $\overline{\text{MSx}}$, `SDA10`) when the host assumes control of the system bus - $\overline{\text{HBG}}$ is asserted. This enables the master ADSP-21161 processor to issue a `REF` command as required.

The SDRAM pin state during the REF command is shown in Table 8-13 below:

Table 8-13. Pin State During REF Command

Pin	State
$\overline{\text{MSx}}$	Low
$\overline{\text{CAS}}$	Low
$\overline{\text{RAS}}$	Low
$\overline{\text{SDWE}}$	High
SDCKE	High

Self Refresh Command (SREF)

The SREF command causes the SDRAM to perform refresh operations internally, without any external control. Before executing the SREF command, the SDRAM precharges the active bank.

SREF mode is enabled by writing a 1 to the SDSRF bit of the SDCTL register.

During entry into SREF, make sure that no SDRAM accesses are occurring and the SDRAM has stopped bursting data. The controller automatically asserts a SREF exit cycle if a SDRAM access occurs during the SREF period. After executing a SREF exit command, the controller waits for 2 + tRC cycles to execute a CBR ($\overline{\text{CAS}}$ before $\overline{\text{RAS}}$) refresh cycle if the refresh counter is expired already. After the CBR refresh command, the SDRAM controller waits for tRC number of cycle before executing a bank activate command.

The SDRAM pin state during the SREF command is shown in Table 8-14.

Table 8-14. Pin State During SREF Command

Pin	State
$\overline{\text{MSx}}$	Low
$\overline{\text{CAS}}$	Low
$\overline{\text{RAS}}$	Low
$\overline{\text{SDWE}}$	High
SDCKE	Low

Programming Example

This section provides a programming example written for the ADSP-21161 processor. The example shown in Listing 8-1 demonstrates how to set up the SDRAM controller to work with the ADSP-21161 processor EZ-KIT Lite®.

Listing 8-1. SDRAM Controller Setup for 21161 EZ-KIT Lite

```
/**************************************************************
*   Setup for the SDRAM Controller for 21161 EZ-KIT Lite *
*   Assumes SDRAM part# Micron MT48LC16M16A1-7SE *
*     SDCLK=100MHz
*     tCK=8ns min @ CL=2    -> SDCL=1 [CAS Latency]       *
*     tRAS=50ns min     -> SDTRAS=3 [active command delay] *
*     tRP=20ns min        -> SDTRP=2 [precharge delay]     *
*     tRCD=20 ns min      -> SDTRCD=2 [CAS-to-RAS delay]   *
*     tREF=64ms/4K rows                                    *
*    ->SDRDIV=(2(30MHz)-CL-tRP-4)64ms/4096=937cycles       *
*   3 SDRAMs by 16 bits wide total = 16Mbit x 48           *
*   Mapped to MS0 addresses 0x00200000-0x002fffff          *
*
**************************************************************/
```

```
#include "def21161.h"

.SEGMENT/PM    pm_code;
.GLOBAL        init_21161_SDRAM_controller;

init_21161_SDRAM_controller:
ustat1=dm(WAIT);
bit clr ustat1 0x000FFFFF;      // Clear MSx waitstate and mode
dm(WAIT)=ustat1;

ustat1=0x1000;                  //Refresh rate
dm(SDRDIV)=ustat1;

ustat1=dm(SDCTL);               // Mask in SDRAM settings
// SDCTL = 0x02214231;
// SDCLKx = CCLK frequency, no SDRAM buffering option, 2 SDRAM banks
// SDRAM mapped to bank 0 only, no self-refresh, page size 256 words
// SDRAM powerup mode is prechrg, 8 CRB refs, and then mode reg set
   cmd
// tRCD = 2 cycles, tRP=2 cycles, tRAS=3 cycles, SDCL=1 cycle
// SDCLK0, SDCLK1, RAS, CAS and SDCLKE activated
bit set ustat1
SDTRCD2|SDCKRx1|SDBN2|SDEM0|SDPSS|SDPGS256|SDTRP2|SDTRAS3|SDCL1;
bit clr ustat1 SDBUF|SDEM3|SDEM2|SDEM1|SDSRF|SDPM|DSDCK1|DSDCTL;
dm(SDCTL)=ustat1;

rts;
```

9 LINK PORTS

The ADSP-21161 processor has two 8-bit wide link ports, which can connect to other processor or peripheral link ports. These bidirectional ports have eight data lines, an acknowledge line, and a clock line. Link ports can operate at frequencies up to the same speed as the processor's internal clock, letting each port transfer up to 8 bits of data per internal clock cycle. Link ports also have the following features:

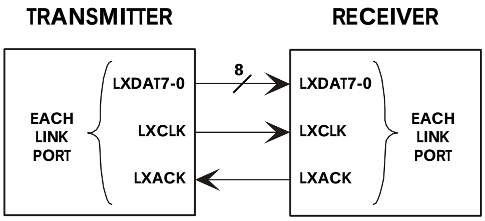
- Operate independently and simultaneously.
- Pack data into 32- or 48-bit words; this data can be directly read by the processor or DMA-transferred to or from on-chip memory.
- Are accessible by the external host processor, using direct reads and writes.
- Have double-buffered transmit and receive data registers.
- Include programmable clock and acknowledge controls for link port transfers. Each link port has its own dedicated DMA channel.
- Provide high-speed, point-to-point data transfers to other processors, allowing differing types of interconnections between multiple DSPs.

ADSP-21161 processor link ports are logically (but not electrically) compatible with previous SHARC processor (ADSP-2106x family) link ports. For more information, see "Link Data Path and Compatibility Modes" on page 9-9.

Table 9-2 lists the pins associated with each link port. Each link port consists of eight data lines (LxDAT7-0), a link clock line (LxCLK), and a link acknowledge line (LxACK). The LxCLK line allows asynchronous data transfers and the LxACK line provides handshaking. When configured as a transmitter, the port drives both the data and LxCLK lines. When configured as a receiver, the port drives the LxACK line. Figure 9-1 shows link port connections.

Table 9-1. Link Port Pins

Link Port Pin(s)	Link Port Function
LxDAT7-0	Link Port x Data
LxCLK	Link Port x Clock
LxACK	Link Port x Acknowledge
"x" denotes the link port number, 0-1	



"X" DENOTES THE LINK PORT NUMBER, 0-5.

Figure 9-1. Link Port Pin Connections

The link port data pins (`L0DAT7-0` and `L1DAT7-0`)are multiplexed internally with data lines `DATA15-0`. If link ports are used, you cannot execute full instruction width (48-bit) transfers. To perform 48-bit transfers, you must set the correct bits `IPACK[1:0]` in the `SYSCON` register and disable the link ports.

Link Port to Link Buffer Assignment

There are two buffers, LBUF0 and LBUF1, that buffer the data flow through the link ports. These buffers are independent of the link ports and may be connected to any of the two link ports. The link ports receive and transmit data on their LxDAT7-0 data pins. Any of the two link buffers may be assigned to handle data for a particular link port. The data in the link buffers can be accessed with DMA or processor core control.

ⓘ "Link port x" does not necessarily connect to "link buffer x."

One link control register (LCTL) controls the two link ports. The link assignment register and common control information have been combined into the link control register in the ADSP-21161 processor.

Link assignment bits in LCTL (similar to the LAR functionality in the ADSP-21160) assign the link buffer-to-port connections. Memory-to-memory transfers may be accomplished by assigning the same link port to two buffers, setting up a loopback mode.

Figure 9-2 shows a block diagram of the link ports and link buffers.

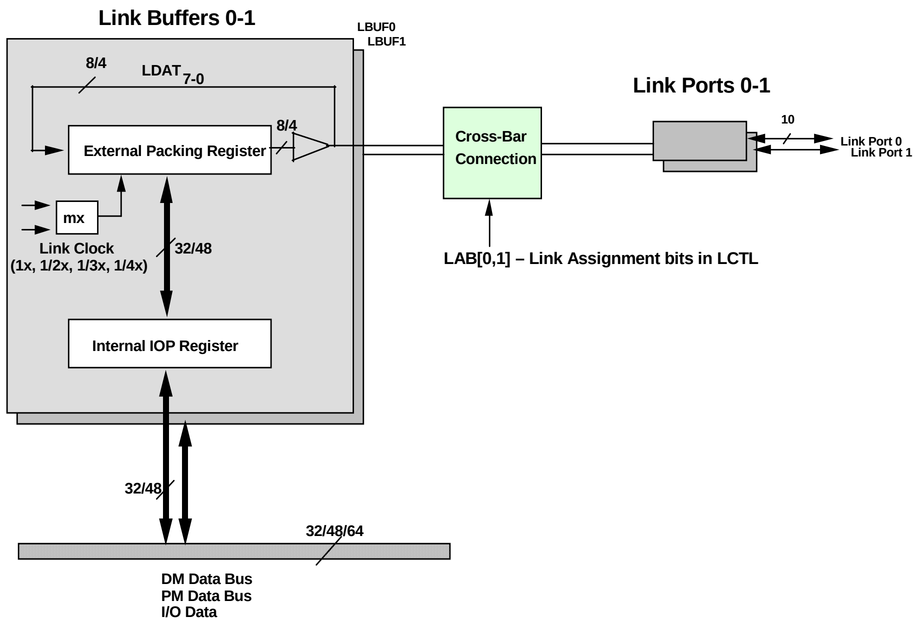


Figure 9-2. Link Ports and Buffers

Link Port DMA Channels

DMA channels 8 and 9 support buffers 0 and 1. The buffer channel pairings are listed in Table 9-2. For more information, see "Link Port DMA" on page 6-81.

Do not enable SPI and link port DMA simultaneously. SPI and link port are mutually exclusive when one of the peripherals is enabled.

Table 9-2. DMA Channel/Link Buffer Pairing

DMA Channel #	Link Buffer Supported
DMA Channel 8	Link Buffer 0
DMA Channel 9	Link Buffer 1

Link Port Booting

Systems may boot the processor through a link port. For more information, see "Bootloading Through The Link Port" on page 6-88.

Setting Link Port Modes

The SYSCON and LCTL registers (Figure 9-3) control the link ports operating modes for the I/O processor. Table A-18 on page A-60 lists all the bits in SYSCON and Table A-25 on page A-93 lists all the bits in LCTL.

The following bits control link port modes. Bits in the SYSCON and LCTL registers setup DMA and I/O processor related link port features. For information on these features, see "Link Port DMA" on page 6-81.

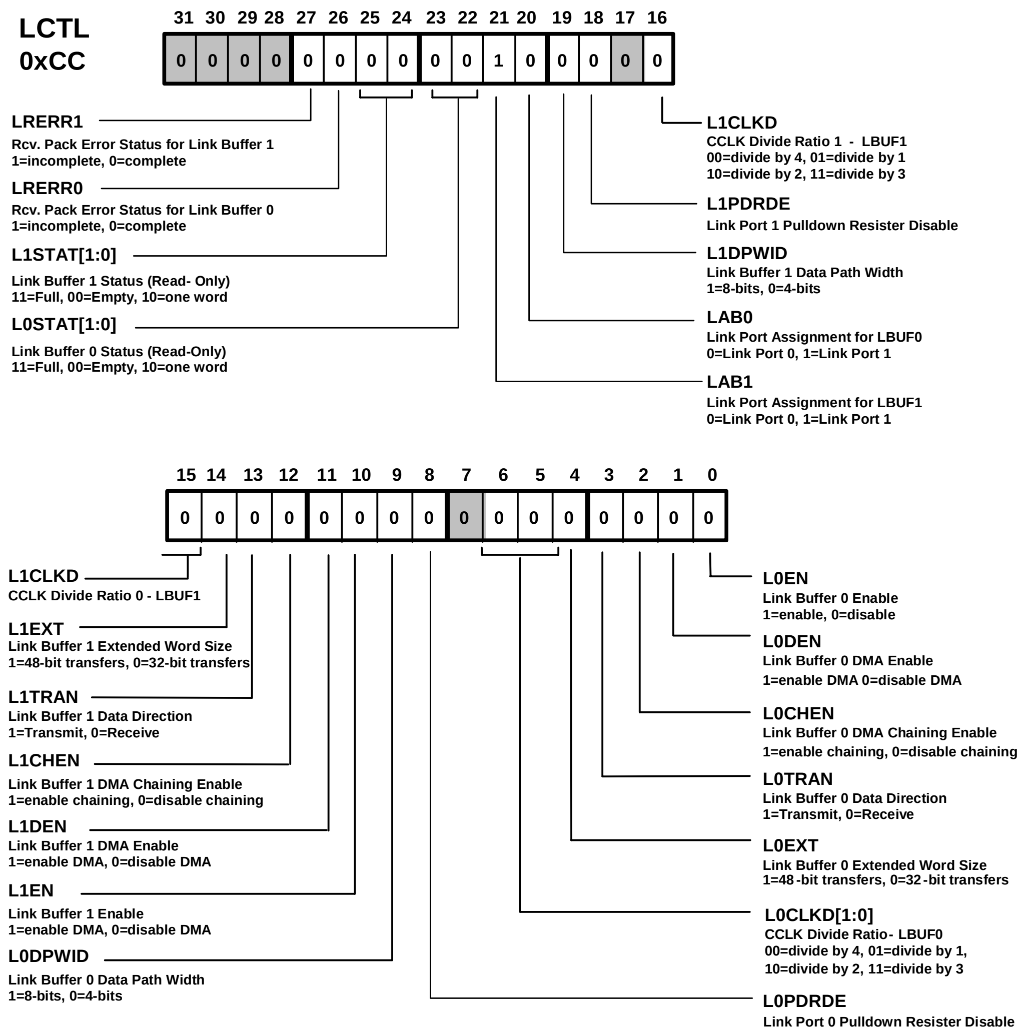

LCTL
0xCC
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
LRERR1
Rcv. Pack Error Status for Link Buffer 1
1=incomplete, 0=complete
LRERR0
Rcv. Pack Error Status for Link Buffer 0
1=incomplete, 0=complete
L1STAT[1:0]
Link Buffer 1 Status (Read- Only)
11=Full, 00=Empty, 10=one word
L0STAT[1:0]
Link Buffer 0 Status (Read-Only)
11=Full, 00=Empty, 10=one word
L1CLKD
CCLK Divide Ratio 1 - LBUF1
00=divide by 4, 01=divide by 1
10=divide by 2, 11=divide by 3
L1PDRDE
Link Port 1 Pulldown Resister Disable
L1DPWID
Link Buffer 1 Data Path Width
1=8-bits, 0=4-bits
LAB0
Link Port Assignment for LBUF0
0=Link Port 0, 1=Link Port 1
LAB1
Link Port Assignment for LBUF1
0=Link Port 0, 1=Link Port 1
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
L1CLKD
CCLK Divide Ratio 0 - LBUF1
L1EXT
Link Buffer 1 Extended Word Size
1=48-bit transfers, 0=32-bit transfers
L1TRAN
Link Buffer 1 Data Direction
1=Transmit, 0=Receive
L1CHEN
Link Buffer 1 DMA Chaining Enable
1=enable chaining, 0=disable chaining
L1DEN
Link Buffer 1 DMA Enable
1=enable DMA, 0=disable DMA
L1EN
Link Buffer 1 Enable
1=enable DMA, 0=disable DMA
L0DPWID
Link Buffer 0 Data Path Width
1=8-bits, 0=4-bits
L0EN
Link Buffer 0 Enable
1=enable, 0=disable
L0DEN
Link Buffer 0 DMA Enable
1=enable DMA 0=disable DMA
L0CHEN
Link Buffer 0 DMA Chaining Enable
1=enable chaining, 0=disable chaining
L0TRAN
Link Buffer 0 Data Direction
1=Transmit, 0=Receive
L0EXT
Link Buffer 0 Extended Word Size
1=48-bit transfers, 0=32-bit transfers
L0CLKD[1:0]
CCLK Divide Ratio- LBUF0
00=divide by 4, 01=divide by 1,
10=divide by 2, 11=divide by 3
L0PDRDE
Link Port 0 Pulldown Resister Disable

Figure 9-3. LCTL Register

Link Port Control Register (LCTL) Bit Descriptions

Note: x denotes 0 for LBUF0-related control bits, or 1 for LBUF1-related control bits.

- **Link Buffer Enable.** Bits 0 and 10 (LxEN). This bit enables (if set, =1) or disables (if cleared, =0) the corresponding link buffer (LBUF0 or LBUF1). When the processor disables the buffer (LxEN transitions from high to low), the processor clears the corresponding LxSTAT and LxRERR bits.

- **Link Buffer DMA Enable.** Bits 1 and 11 (LxDEN). This bit enables (if set, =1) or disables (if cleared, =0) DMA transfers for the corresponding link buffer (LBUF0 or LBUF1).

- **Link Buffer DMA Chaining Enable.** Bits 2 and 12 (LxCHEN). This bit enables (if set, =1) or disables (if cleared, =0) DMA chaining for the corresponding link buffer (LBUF0 or LBUF1)

- **Link Buffer Transfer Direction.** Bits 3 and 13 (LxTRAN). This bit selects the transfer direction (transmit if set, =1) (receive if cleared, =0) for corresponding link buffer (LBUF0 or LBUF1).

- **Link Buffer Extended Word Size.** Bits 4 and 14 (LxEXT). This bit selects the transfer extended word size (48-bit if set, =1) (32-bit if cleared, =0) for the corresponding link buffer (LBUF0 or LBUF1). Programs must not change a buffer's LxEXT setting while the buffer is enabled.

  The buffer's LxEXT setting overrides the internal memory block's setting IMDWx for Normal word width. Whether buffer is set for 48- or 32- bit words, programs must index (IIx) the corresponding DMA channel with a Normal word address.

- **Link Port Clock Divisor.** Bits 6-5 and 16-15 (LxCLKD). These bits select the transfer clock divisor for link buffer x (LBUF0 or LBUF1). The transfer clock equals the processor core clock divided by LxCLKD, where L0CLKD[6-5] and L1CLKD[16-15]is: 01=1, 10=2, 11=3, or 00=4.

- **Link Port Pulldown Resistor Disable.** Bit 8 and 18 (LxPDRDE).This bit disables (if set, =1) or enables (if cleared, =0) the internal pull-down resistors on the LxCLK, LxACK, and LxDAT7-0 pins of the corresponding unassigned or disabled link port; this bit applies to the port which is not necessarily the port assigned to link buffer x (LBUF0 or LBUF1). For revisions 0.3, 1.0 and 1.1, LxCLK,LxDAT7-0 and LxACK have a 50kΩ internal pulldown resistor. For revisions 1.2 and greater, LxDAT7-0 has a 20kΩ internal pulldown resistor. See Table 13-3 for a description of resistor values of the pins.

  Systems should not leave link port pins (LxCLK, LxACK, and LxDAT7-0) unconnected without clearing the corresponding LxP-DRDE bit or applying an external pulldown. In systems where several DSPs share a link port, only one processor should have this bit cleared.

- **Link Port Data Path Width.** Bits 9 and 19 (LxPDPWID). This bit selects the link port data path width (8-bit if set, =1) (4-bit if cleared, =0) for the corresponding link buffer (LBUF0 or LBUF1).

  Systems using a 4-bit width should connect the lower link port data pins (LxDAT3-0) for data transfers and leave the upper pins (LxDAT7-4) unconnected. In the 4-bit mode, the processor applies pulldowns to the upper pins.

- **Link Port Assignments for LBUF0.** Bit 20 (LAB0). This bit assigns link buffer 0 to link port 1 if set (=1) or link port 0 if cleared (=0).

- **Link Port Assignments for LBUF1.** Bit 21 (LAB1). This bit assigns link buffer 1 to link port 1 if set (=1) or link port 0 if cleared (=0).

- **Link Buffer Status.** Bits 23-22 and 25-24 (LxSTAT). These bits identify the status of the corresponding link buffer as follows: 11=full, 00=empty, 10=one word.
- **Receive Packing Error Status.** Bit 27 and 26 (LRERRx). This bit indicates if the packed bits in the corresponding link buffer were receive completely (=0), without error, or incompletely (=1).

If multiple link ports are bussed together and the link port pull-down resistor is enabled on all the processors, the line is heavily loaded. Ensure only one processor has this functionality.

The processor's internal clock (CCLK) is the CLKIN frequency multiplied by a clock ratio (CLK_CFG1-0)and the CLKDBL pin (1:1 or 2:1 ratio). For more information, see the clock ratio pin description in Table 13-1 on page 13-4.

When link buffers are enabled or disabled, the I/O processor may generate unwanted interrupt service requests if Link Service Request (LSRQ) interrupts are unmasked. To avoid unwanted interrupts, programs should mask the LSRQ interrupts while enabling or disabling link buffers. For more information, see "Using Link Port Interrupts" on page 9-17.

Link Data Path and Compatibility Modes

The link ports can transmit and received data using all eight of the link port's data pins (LxDAT7-0) or the four lower data pins (LxDAT3-0). The LxDPWID bit in the LCTL register selects the link port data path width (8-bit if set, =1) (4-bit if cleared, =0). Before changing the mode of the link port, disable the link port. After the port is disabled, a new control word can be written to LCTL.

When LxDPWID is cleared (4-bit data path), the ADSP-21161 processor can be connected to link ports of previous SHARC processors (ADSP-2106x family). The link port receiver must run at the same speed or faster than the transmitter. Connecting to an ADSP-2106x may require that the ADSP-21161 processor be configured for 1/2 core clock rate operation. For more information, see "Using Link Port Handshake Signals" on page 9-10.

Using Link Port Handshake Signals

The LxCLK and LxACK pins of each link port allow handshaking for asynchronous data communication between DSPs. Other devices that follow the same protocol may also communicate with these link ports. The processor link ports are backward compatible with the SHARC link ports for basic transfers, including LSRQ functions.

A SHARC compatible link can be enabled by adjusting the upper LxCLKD bit in the LCTL register and by clearing the LxDPWID bit in the LCTL register, enabling the 4-bit data path.

The link port receiver must run at the same speed or faster than the transmitter. Connecting to an ADSP-2106x may require that the ADSP-21161 processor be configured for 1/2 core clock rate operation.

A link-port-transmitted word consists of 4 bytes (for a 32-bit word) or 8 nibbles or 6 bytes (for a 48-bit word) or 12 nibbles. The transmitter asserts the clock (LxCLK) high with each new byte of data. The falling edge of LxCLK is used by the receiver to latch the byte. The receiver asserts LxACK when it is ready to accept another word in the buffer. The transmitter samples LxACK at the beginning of each word transmission (that is, after every 4 or 6 bytes). If LxACK is deasserted at that time, the transmitter does not transmit the new word. For more information, see Figure 9-4. The transmitter leaves LxCLK high and continues to drive the first byte if LxACK is deasserted. When LxACK is eventually asserted again, LxCLK goes low and begins transmission of the next word. If the transmit buffer is empty, LxCLK remains low until the buffer is refilled, regardless of the state of LxACK.

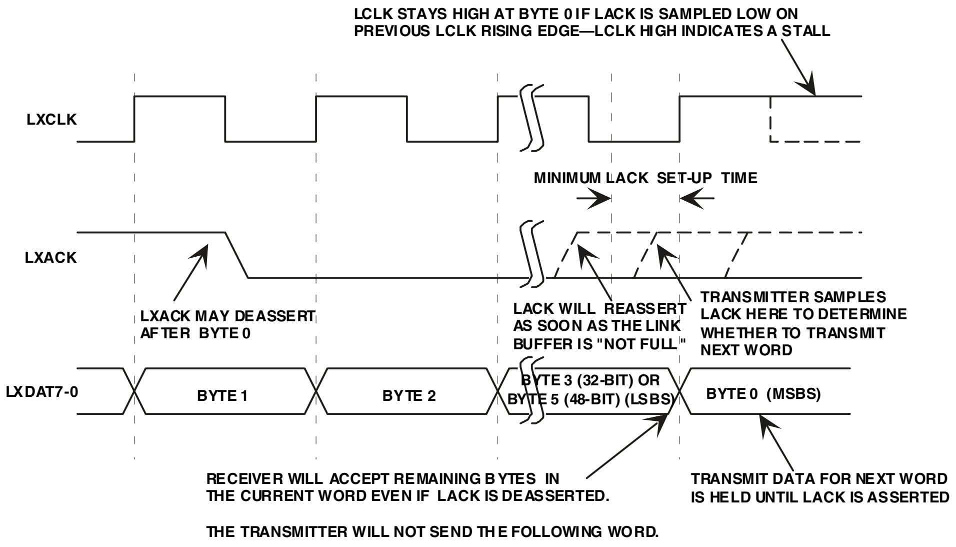

LCLK STAYS HIGH AT BYTE 0 IF LACK IS SAMPLED LOW ON PREVIOUS LCLK RISING EDGE—LCLK HIGH INDICATES A STALL
LXCLK
MINIMUM LACK SET-UP TIME
LXACK
LXACK MAY DEASSERT AFTER BYTE 0
LACK WILL REASSERT AS SOON AS THE LINK BUFFER IS "NOT FULL"
TRANSMITTER SAMPLES LACK HERE TO DETERMINE WHETHER TO TRANSMIT NEXT WORD
LXDAT7-0
BYTE 1
BYTE 2
BYTE 3 (32-BIT) OR BYTE 5 (48-BIT) (LSBS)
BYTE 0 (MSBS)
RECEIVER WILL ACCEPT REMAINING BYTES IN THE CURRENT WORD EVEN IF LACK IS DEASSERTED.
THE TRANSMITTER WILL NOT SEND THE FOLLOWING WORD.
TRANSMIT DATA FOR NEXT WORD IS HELD UNTIL LACK IS ASSERTED

Figure 9-4. Link Port Handshake Timing

The receive buffer may fill if a higher priority DMA, core I/O processor register access, direct read, direct write or chain loading operation is occurring. `LxACK` may de-assert when it anticipates the buffer may fill. `LxACK` is reasserted by the receiver as soon as the internal DMA grant signal has occurred, freeing a buffer location.

Data is latched in the receive buffer on the falling edge of `LxCLK`. The receive operation is purely asynchronous and can occur at any frequency up to the processor clock frequency.

When a link port is not enabled, `LxDAT7-0`, `LxCLK` and `LxACK` are three-stated. When a link port is enabled to transmit, the data pins are driven with whatever data is in the output buffer, `LxCLK` is driven high and `LxACK` is three-stated. When a port is enabled to receive, the data pins and `LxCLK` are three-stated and `LxACK` is driven high.

To allow a transmitter and a receiver to be enabled (assigned and link buffer enabled) at different times, LxACK, LxCLK, and LxDAT7-0 may be held low with their internal pull-down resistor if LxPDRDE is cleared when the link port is disabled. LxDAT7-0 is kept at the previously driven value by internal keeper latches on the link port data lines if LxPDRDE is cleared when the link port is disabled. If the transmitter is enabled before the receiver, LxACK is low and the transmission is held off. If the receiver is enabled before the transmitter, LxCLK is held low by the pulldown and the receiver is held off. If many link ports are bused together, the systems may need to enable only one of the internal resistors to pull down each bused pin, so the bused lines are not pulled down too strongly or too heavily loaded.

Refer to Table 13-1 on page 13-4 for detailed pin descriptions and Table 13-3 on page 13-22 for more information on pull down resistors.

LxACK, LxCLK, and LxDAT7-0 should not be left unconnected unless external pull-down resistors are used.

Using Link Buffers

Each link buffer consists of an external and an internal 48-bit register. For more information, see Figure 9-2 on page 9-4. When transmitting, the internal register is used to accept core data or DMA data from internal memory. When receiving, the external register performs the packing and unpacking for the link port, most significant nibble or byte first. These two registers form a two-stage FIFO for the LBUFx buffer. Two writes (32- or 48-bit) can occur to the register by the DMA or the core, before it signals a full condition. As each word is unpacked and transmitted, the next location in the FIFO becomes available and a new DMA request is made. If the register becomes empty, the LxCLK signal is de-asserted. When transmitting, only the number of words written are transmitted.

Full/empty status for the link buffer FIFOs is given by the LxSTAT bits of the LCTL register. This status is cleared for a link buffer when its LxEN enable bit is cleared in the LCTL register.

During receiving, the external buffer is used to pack the receive link port data (most significant nibble or byte first) and pass it to the internal register before DMA-transferring it to internal memory. This buffer is a two-deep FIFO. If the processor's DMA controller does not service it before both locations are filled, the LxACK signal is de-asserted.

The link buffer width may be selected to be either 32 or 48 bits. This selection is made individually for each buffer with the LxEXT bits in the LCTLx register. For 40-bit extended precision data or 48-bit instruction transfers, the width must be set to 48 bits.

Core Processor Access To Link Buffers

In applications where the latency of link port DMA transfers to and from internal memory is too long, or where a process is continuous and has no block boundaries, the processor core may read or write link buffers directly using the full or empty status bit of the link buffer to automatically pace the operation. The full or empty status of a particular LBUFx buffer can be determined by reading the LxSTATx bits in LCTL. DMA should be disabled when using this capability (LxDEN=0).

If a read is attempted from an empty receive buffer, the core stalls (hangs) until the link port completes reception of a word. If a write is attempted to a full transmit buffer, the core stalls until the external device accepts the complete word. Up to four words (2 in the receiver and 2 in the transmitter) may be sent without a stall before the receiver core or DMA must read a link buffer register.

To support debugging buffer transfers, the processor has a Buffer Hang Disable (BHD) bit. When set (=1), this bit prevents the processor core from detecting a buffer-related stall condition, permitting debugging of this type of stall condition. For more information, see the BHD discussion on page on page 6-43.

Host Processor Access To Link Buffers

When a 32-bit, 16-bit, or 8-bit host processor normally accesses IOP register space (with the exception of LBUFx and EPBx buffers), the ADSP-21161 defaults to pack and unpack data internally (independent of the setting of the PMODE bits in the DMACx register) to a 32-bit access.

The link buffers LBUF0 and LBUF1 can also be accessed by an external host processor, using direct reads and writes to IOP register space. However, there is a difference in how data is accessed with the link buffers compared to other IOP registers accessed as 32-bit data. Host accessing link port buffers pack or unpack to 48-bits internally, ignoring the value of PMODE in DMACx, but using the HBW bits in SYSCON to set the external packing mode.

In the case where a host processor reads or writes to the LBUF0 and LBUF1 link buffers, the PMODE bits in DMACx external port DMA control register are ignored and are hardwired to a special 48-bit internal packing mode. A fixed packing mode for an 8-, 16-, or 32-bit (corresponding to the host bus width (HBW) bits in SYSCON) external host to 48-bits internal is selected. This fixed 48-bit internal packing mode is required due to the fact that the ADSP-21161 link port buffers can transmit/receive 48-bit words.

It may be desirable in some applications for a host processor to transfer instruction opcodes to another SHARC indirectly via the directly connected SHARC's link port by reading or writing the opcode data to or from the LBUF0 and LBUF1 link buffers through the external port. For example, with a 16-bit host, the packing mode internally defaults to 48-bit packed transfers. The packing mode is 16 external to 48-bit internal.

Depending on the HBW (host bus width) bits in SYSCON, the appropriate 48-bit internal packing mode is selected. Table 6-7 on page 6-36 summarizes the packing mode bit settings for access to link port buffers.

Host packing examples are shown below for host direct read/write access to LBUFx link port data buffers. When interfacing to a host processor, the HMSWF bit determines whether the I/O processor packs to most significant 16-bit word first (=1)or least significant 16-bit word first (=0). The packing mode defaults to 48-bit internal packing for host accesses to LBUFx, ignoring PMODE value in DMACx.

Table 9-3. Packing Sequence for 16-Bit Bus (MSW First)

Transfer	Data Bus Pins 31-16
First	Word 1; bits 47-32
Second	Word 1; bits 31-16
Third	Word 1; bits 15-0

Table 9-4. Packing Sequence for 16-Bit Bus (LSW First)

Transfer	Data Bus Pins
First	Word 1; bits 15-0
Second	Word 1; bits 31-16
Third	Word 1; bits 47-32

Table 9-5. Packing Sequence from 8-bit bus (MSW first)

Transfer	Data Bus Pins 23-16
First	Word 1; bits 47-40
Second	Word 1; bits 39-32
Third	Word 1; bits 31-24
Fourth	Word 1; bits 23-16
Fifth	Word 1; bits 15-8
Sixth	Word 1; bits 7-0

Table 9-6. Packing sequence from 8-bit bus (LSW first)

Transfer	Data Bus Pins 23-16
First	Word 1; bits 7-0
Second	Word 1; bits 15-8
Third	Word 1; bits 23-16
Fourth	Word 1; bits 31-24
Fifth	Word 1; bits 39-32
Sixth	Word 1; bits 47-40

To write a single 48-bit word or an odd number of 48-bit words to LBUFx, write a dummy access to completely fill the packing buffer, or write the HPFLSH bit in SYSCON to flush the partially filled packing buffer and remove the unused word. The HPFLSH bit clears the HPS bits in SYSTAT as well.

Using Link Port DMA

DMA channels 8-9 support link buffers 0-1. These DMA channels are shared with the SPI transmit and receive buffers. A maskable interrupt is generated when the DMA block transfer has completed. For more information on link port interrupts, see "Using Link Port Interrupts" on page 9-17. For more information on link port DMA, see "Link Port DMA" on page 6-81.

⊘ The link port channels share DMA channels 8 and 9 with the SPI transmit and receive buffers. Do not enable SPI and link port DMA simultaneously. SPI and link port are mutually exclusive when one of the peripherals is enabled.

In chained DMA operations, the processor automatically sets up another DMA transfer when the current DMA operation completes. The chain pointer register (`CPLB0`, and `CPLB1`) is used to point to the next set of buffer parameters stored in memory. The processor's DMA controller automatically downloads these buffer parameters to set up the next DMA sequence. For information on setting up DMA chaining, see "Chaining DMA Processes" on page 6-25.

Using Link Port Interrupts

Three types of interrupts are dedicated to the link ports:

The I/O processor generates a DMA channel interrupt when a DMA block transfer through the link port with DMA enabled (`LxDEN`=1) finishes.

- The I/O processor generates a DMA channel interrupt when DMA for the link buffer channel is disabled (`LxDEN`=0) and the buffer is not full (for transmit) or the buffer is not empty (for receive).
- The I/O processor generates a Link Services Request (`LSRQ`) interrupt when an external source accesses a disabled link port, an unassigned link port or assigned port with buffer disabled.

Although the link ports and the SPI port share DMA channels 8 and 9, there are different interrupt vector locations dedicated for these two peripherals. The `LIRPTL` register controls both the link port and SPI transmit/receive interrupt latching and masking functions. The IRPTL register controls a single global link port interrupt that latches the LPISUM bit. This bit indicates whether at least one of the two unmasked link port interrupt is latched. Refer to Figure 9-5 and Table A-10 on page A-34 for complete bit description of the LIRPTL register.

During reset, if a link port boot is enabled, the mask bit for LBUF0 (bit 16) is set (for example, the interrupt is unmasked). If a SPI boot is enabled, the mask bit for SPI receive (bit 18) is set.

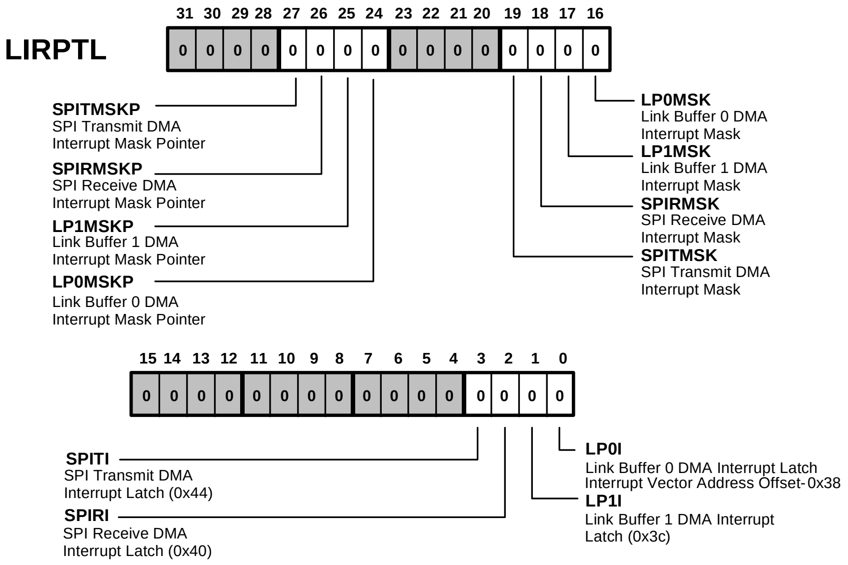

LIRPTL
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
SPITMSKP
SPI Transmit DMA Interrupt Mask Pointer
SPIRMSKP
SPI Receive DMA Interrupt Mask Pointer
LP1MSKP
Link Buffer 1 DMA Interrupt Mask Pointer
LP0MSKP
Link Buffer 0 DMA Interrupt Mask Pointer
LP0MSK
Link Buffer 0 DMA Interrupt Mask
LP1MSK
Link Buffer 1 DMA Interrupt Mask
SPIRMSK
SPI Receive DMA Interrupt Mask
SPITMSK
SPI Transmit DMA Interrupt Mask
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
SPITI
SPI Transmit DMA Interrupt Latch (0x44)
SPIRI
SPI Receive DMA Interrupt Latch (0x40)
LP0I
Link Buffer 0 DMA Interrupt Latch Interrupt Vector Address Offset-0x38
LP1I
Link Buffer 1 DMA Interrupt Latch (0x3c)

Figure 9-5. LIRPTL Register

Link Port Interrupts With DMA Enabled

A link port interrupt is generated when the DMA operation is done—when the block transfer has completed and the DMA count register is zero.

One way programs can use this interrupt is to send additional control information at the end of a block transfer. Because the receive DMA buffer is empty when the DMA block has completed, the external bus master can send up to two additional words to the slave processor's buffer, which has space for the two words. When the slaves's DMA completes, there is an interrupt. In the associated interrupt service routing, the buffer can be read in order to use these control words to determine the next course of action.

Link Port Interrupts With DMA Disabled

If DMA is disabled for a link port buffer, then the buffer may be written or read by the processor core as a memory-mapped I/O processor register.

If the DMA is disabled but the associated link buffer is enabled, then a maskable interrupt is generated whenever a receive buffer is not empty or when a transmit buffer is not full. This interrupt is the same interrupt vector associated with the completion of the DMA block transfers.

The interrupt latch bit in `LIRPTL` may be unmasked by the corresponding mask bit in the same register. When initially enabling the mask bit, the corresponding latch bit in `LIRPTL` should be cleared first to clear out any request that may have been inadvertently latched.

The interrupt service routine should test the buffer status after each read or write to check when the buffer is empty or full, in order to determine when it should return from interrupt. This will reduce the number of interrupts it must service.

Link Port Service Request Interrupts (LSRQ)

Link port service requests let a disabled (unassigned or assigned with buffer disabled) link port cause an interrupt when an external access is attempted. The transmit and receive request status bits of the `LSRQ` register indicate when another processor is attempting to send or receive data through a particular link port. Two processors can communicate without prior knowledge of the transfer direction, link port number, or exactly when the transfer is to occur. The LRSQ register is shown in Figure 9-6 and described in Table A-26 on page A-98.

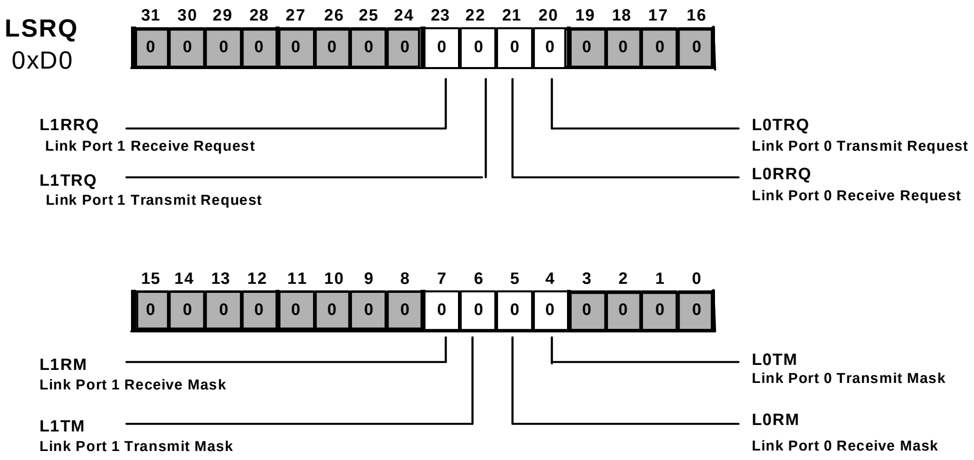


Figure 9-6. LSRQ Register

In Figure 9-6, for transmit request status bits, LxTRQ=1 means LxACK=1, LxTM=1, and LxEN=0; for receive request status bits, LxRRQ=1 means LxCLK=1, LxRM=1, and LxEN=

When LxACK or LxCLK is asserted externally, a link service request (LSR) is generated in a disabled (unassigned or assigned with buffer disabled) link port. LSRs are not generated for a link port that is disabled by loopback mode. Each LSR is gated by mask bits before being latched in the LSRQ register. The two possible receive LSRs and the two possible transmit LSRs are gated by mask bits and then ORed together to generate the link service request interrupt. The LSRQ interrupt request may be masked by the LSRQI mask bit of the IMASK register. When the mask bit is set, the interrupt is allowed to pass into the interrupt priority encoder. A diagram of this logic appears in Figure 9-7.

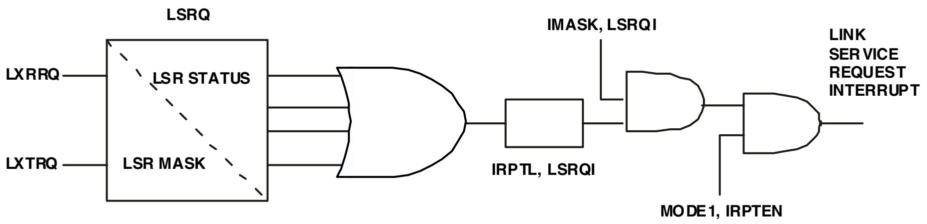


Figure 9-7. Logic for Link Port Interrupts

The interrupt routine must read the LSRQ register to determine which link port to service and whether it is a transmit or receive request. LSR interrupts have a latency of two cycles. Note that the link service request interrupt is different from the link receive and transmit interrupt—this is also true in IMASK.

The 32-bit LSRQ register holds the masked link status of each link port and the corresponding interrupt mask bits. The link service request status of the port is set whenever the port is not enabled and one of LxACK or LxCLK is asserted high. The LSRQ status bits are read-only. Table A-26 on page A-98 shows the individual bits of the LSRQ register.

To determine which link port to service, programs can transfer LSRQ to a register Rx (in the register file) and use the leading 0s detect instruction: Rn=LEFTZ Rx. Here, Rn indicates which link port is active in order of priority.

If link service requests are in use, they should be masked out when the assigned link buffers are being enabled, disabled, or when the link port is being unassigned in LCTL. Otherwise, spurious service requests may be generated.

The need for masking is due to a delay before LxCLK or LxACK (if already asserted) signals are pulled (if pulldowns enabled) or driven externally (if pulldowns disabled) below logic threshold. During this delay, these signals are sampled asserted and generate an LSRQ.

ⓘ To avoid the possibility of spurious interrupts, programs should mask the LSRQ interrupt or the appropriate request bit in the LSRQ register and allow a delay before unmasking. Alternatively, programs can mask the LSRQ interrupt and poll the appropriate request status bit until it is cleared and then unmask the interrupt.

Detecting Errors on Link Transmissions

Transmission errors on the link ports may be detected by reading the LRERRx bits (bits 26 and 27) in the LCTL register. These bits reflects the status of each nibble or byte counter. The LRERRx bit is cleared (=0) if the pack counter of the corresponding link buffer is zero—a multiple of 8 or 12 nibbles or bytes have been received. If LRERR is set (=1) when a transmission has completed, then an error occurred during transmission.

ⓘ The DMA word count provides an exact count of the number of words to be transferred.

To allow checking of this status, the transmitter and receiver should follow a protocol such as the following:

- **Transmitter Protocol**—To make use of the LRERRx status, one additional dummy word should always be transmitted at the end of a block transmission. The transmitter must then deselect the link port and re-enable as a receiver to allow the receiver to send an appropriate message back to the transmitter.

- **Receiver Protocol**—When the receiver has received the data block, indicated by a the same interrupt vector associated with the completion of the link port DMA, it checks that it has received an additional word in the link buffer and then reads the LRERR bit. The receiver may then clear the link buffer (LxEN=0) and transmit the appropriate message back to the transmitter on the same, or a different, link port.

Link Port Programming Examples

This section provides two programming examples written for the ADSP-21161 processor. The example shown in Listing 9-1 demonstrates how the core directly writes to the link port transmit buffer and reads from the link port receive buffer after an interrupt. The example shown in Listing 9-2 demonstrates how the core directly reads from the link port receive buffer and writes to the link port transmit buffer.

Listing 9-1. Interrupt Core-Driven Link Loopback Example

```
/*_________________________________________________________________
ADSP-21161 Interrupt Core-Driven LINK Loopback Example

This example shows an internally looped-back link port 32-bit
transfer. The core directly writes to the transfer link buffer
(LBUF1) and reads from the receive link buffer (LBUF0). The core
will hang on  the read of LBUF0 until the data is ready. Loopback
is achieved by assigning the transmit and receive link buffers to
the same port. (Port 0)
_________________________________________________________________*/

#include "def21161.h"
#define N 8

.section/pm seg_rth;  /* Reset vector from ldf file */
nop;
jump start;
```

```
.section/dm seg_dmda;  /* Data section from ldf file */
.var source[N]= 0X11111111, 0X22222222, 0X33333333, 0X44444444,
0X55555555, 0X66666666, 0X77777777, 0X88888888;

.var dest[N];

.section/pm lp1i_svc;   /*Link Port 1 interrupt vector from ldf
                           file */
jump lpISR1; nop; nop; nop;

.section/pm lp0i_svc;   /*Link Port 0 interrupt vector from ldf
                           file */
jump lpISR1; nop; nop; nop;

/*_____________________Main Routine________________________*/
.section/pm seg_pmco;  /* Main code section from ldf file */
start:
B0=source;   /* Set pointers for source and dest */
L0=@source;
B1=dest;
L1=@dest;

/*Enable Global, Link Port, and Link Port Buffer 1 interrupts*/
bit set imask LPISUMI;
bit set lirptl LP1MSK;
bit set mode1 IRPTEN | CBUFEN;  /* Enable circular buffers */

ustat1=dm(LCTL);

/* LCTL REGISTER--LBUF1=TX, LBUF0=RX, 1/2x CCLK RATE, LBUF 0 &
1ENABLED, LBUF 0 & 1 -> PORT 0 */
bit clr ustat1 L0TRAN | LAB0 | LAB1 | L0CLKD0 | L1CLKD0;
bit set ustat1 L1TRAN | L1EN | L0EN | L0CLKD1 | L1CLKD1;
dm(LCTL)=ustat1;
```

```
wait: idle;
jump wait;

lpISR1:          /* Link Port Service Routine */
R0=dm(I0,1);     /* Get data for TX */
dm(LBUF1)=R0;    /* Write data to LBUF1 */
R1=dm(LBUF0);    /* Read data-core will hang here until data is
                    received. */
dm(I1,1)=R1;     /* Store incoming data to dest buffer */
rti;
```

Listing 9-2. Core-Driven Link Loopback Example

```
/*______________________________________________________________
  ADSP-21161 Core-Driven LINK Loopback Example

This example shows an internally looped-back link port 32-bit
transfer.  The core directly writes to the transfer link buffer
(LBUF1) and reads from the receive link buffer (LBUF0). The core
will hang on  the read of LBUF0 until the data is ready. Loopback
is achieved by assigning the transmit and receive link buffers to
the same port. (Port 0)
______________________________________________________________*/
#include "def21161.h"
#define N 8

.section/pm seg_rth;    /* Reset vector from ldf file */
nop;
jump start;

.section/dm seg_dmda;   /* Data section from ldf file */
.var source[N]= 0X11111111, 0X22222222, 0X33333333, 0X44444444,
0X55555555, 0X66666666, 0X77777777, 0X88888888;
```

```
.var dest[N];

/*_________________________Main Routine_________________________*/
.section/pm seg_pmco;  /* Main code section from ldf */
start:
r0=0; DM(LCTL)=r0;    /* Clear LCTL register */

B0=source;          /* Set up pointers for source and dest */
L0=@source;
B1=dest;
L1=@dest;

ustat1=dm(LCTL);
/* LCTL REGISTER-->LBUF1=TX, LBUF0=RX, 2x CLK RATE, LBUF 0 & 1
ENABLED, LBUF 0 & 1 -> PORT 0 */
bit clr ustat1 L0TRAN | L0CLKD0 | L1CLKD0 | LAB0 | LAB1;
bit set ustat1 L1TRAN | L1EN | L0EN | L0CLKD1 | L1CLKD1;
dm(LCTL)=ustat1;

lcntr=N, do transfer until lce;
R0=dm(I0,1);    /* Test data to TX */
dm(LBUF1)=R0;   /* Write data to LBUF1 */
R1=dm(LBUF0);   /* Read data-core will hang here until data is
                   received. */
transfer: dm(I1,1)=R1;   / *Store incoming data to dest buffer */

wait: idle;
jump wait;
```

Using Token Passing With Link Ports

When two DSPs communicate using a link port only one can be the transmitter or receiver. Token passing is a protocol that assists the DSPs alternate control. Figure 9-8 shows a flow chart of the token passing process.

In token passing, the token is a software flag that passes between the processors. At reset, the token (flag) is set to reside in the link port of one device, making it the master and the transmitter. When a receiver link port (slave) wants to become the master, it may assert its LxACK line (request data) to get the master's attention. The master knows, through software protocol, whether it is supposed to respond with actual data or whether it is being asked for the token.

The token release word can be any user-defined value. Since both the transmitter and receiver are expecting a code word, this does not need to be exclusive of normal data transmission.

If the master wishes to give up the token, it may send back a user-defined token release word and thereafter clear its token flag. Simultaneously, the slave examines the data sent back and if it is the token release word, the slave will set its token, and can thereafter transmit. If the received data is not the token release word, then the slave must assume the master was beginning a new transmission.

Through software protocol, the master can also ask to receive data by sending the token release word without the LxACK (data request) going low first.

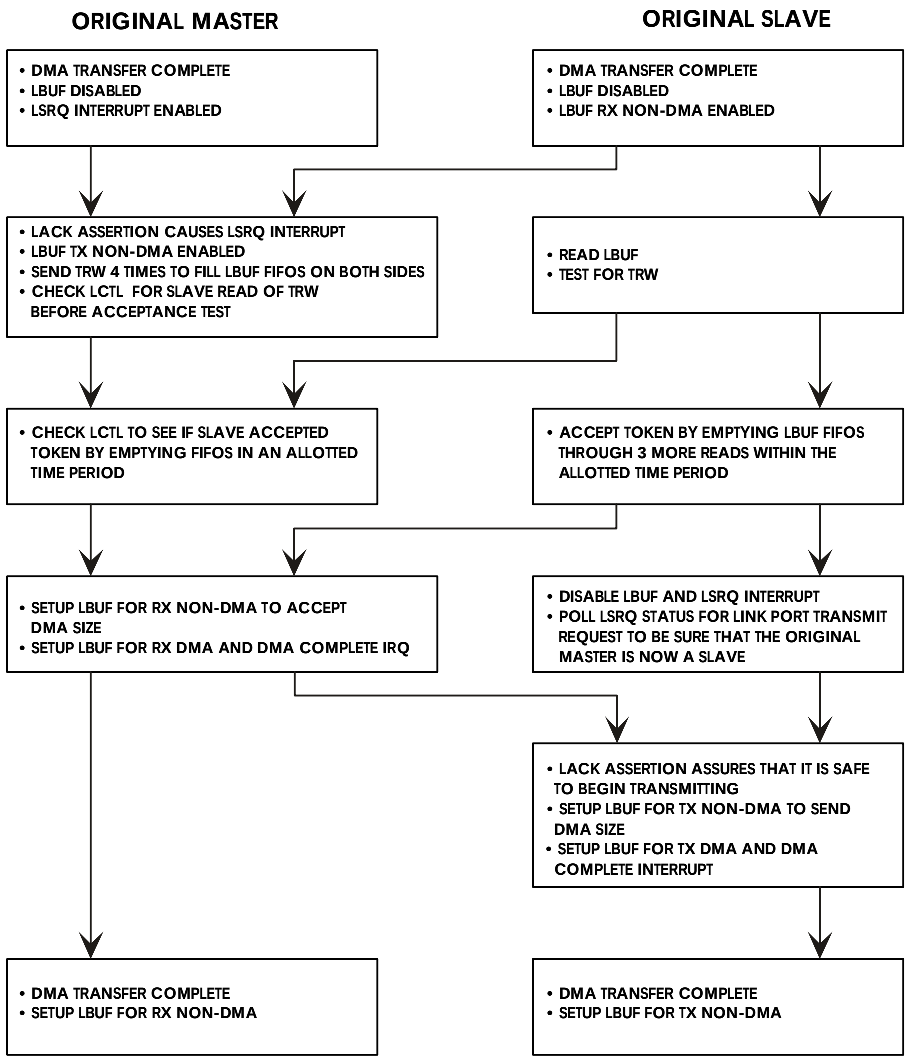

ORIGINAL MASTER
ORIGINAL SLAVE
• DMA TRANSFER COMPLETE
• LBUF DISABLED
• LSRQ INTERRUPT ENABLED
• DMA TRANSFER COMPLETE
• LBUF DISABLED
• LBUF RX NON-DMA ENABLED
• LACK ASSERTION CAUSES LSRQ INTERRUPT
• LBUF TX NON-DMA ENABLED
• SEND TRW 4 TIMES TO FILL LBUF FIFOS ON BOTH SIDES
• CHECK LCTL FOR SLAVE READ OF TRW BEFORE ACCEPTANCE TEST
• READ LBUF
• TEST FOR TRW
• CHECK LCTL TO SEE IF SLAVE ACCEPTED TOKEN BY EMPTYING FIFOS IN AN ALLOTTED TIME PERIOD
• ACCEPT TOKEN BY EMPTYING LBUF FIFOS THROUGH 3 MORE READS WITHIN THE ALLOTTED TIME PERIOD
• SETUP LBUF FOR RX NON-DMA TO ACCEPT DMA SIZE
• SETUP LBUF FOR RX DMA AND DMA COMPLETE IRQ
• DISABLE LBUF AND LSRQ INTERRUPT
• POLL LSRQ STATUS FOR LINK PORT TRANSMIT REQUEST TO BE SURE THAT THE ORIGINAL MASTER IS NOW A SLAVE
• LACK ASSERTION ASSURES THAT IT IS SAFE TO BEGIN TRANSMITTING
• SETUP LBUF FOR TX NON-DMA TO SEND DMA SIZE
• SETUP LBUF FOR TX DMA AND DMA COMPLETE INTERRUPT
• DMA TRANSFER COMPLETE
• SETUP LBUF FOR RX NON-DMA
• DMA TRANSFER COMPLETE
• SETUP LBUF FOR TX NON-DMA

Figure 9-8. Token Passing Flow Chart

To use the example, the example code is to be loaded on both the original master and the original slave. The code is ID intelligent for multiprocessor systems: ID1 is the original master (transmitter) and ID2 is original slave (receiver). The master transmits a buffer via DMA through link port 0 using `LBUF1` and the slave receives through link port 0 using `LBUF0`. The slave then requests the token by generating an `LSRQ` interrupt in the disabled link port of the master (LPORT0). The master responds by sending the token release word and waiting to see if it is accepted. The slave checks to see that it is the token release word and accepts the token by emptying the master's link buffer FIFO within a predetermined amount of time. If the token is accepted the slave becomes the master and transmits a buffer of data to the new slave. If the token is rejected, the master transmits a second buffer. When complete, the original master will finish by setting up `LBUF0` to receive without DMA, and the original slave sets up `LBUF1` to transmit without DMA.

The following is a list of the areas of concern when a program implements a software protocol scheme for token passing:

- The program must make sure that both link buffers are not enabled to transmit at the same time. In the event that this occurs, data may be transmitted and lost due to the fact that neither link port is driving `LxACK`. In the example, the `LSRQ` register status bits are polled to ensure that the master becomes the slave before the slave becomes the master, avoiding the two transmitter conflict.

- The program must make sure that the link interrupt selection matches the application. If a status detection scheme using the status bits of the `LSRQ` register is to be used, it is important to note the following: If a link port that is configured to receive is disabled while `LxACK` is asserted, there is an RC delay before the 50kΩ pulldown resistor[1] on `LxACK` (if enabled) can pull the value below logic

[1] LxACK has a 20kΩ pulldown resistor for revisions 1.2 and higher.

threshold. If the appropriate request status bit is unmasked in the LSRQ register (in this instance), then an LSR is latched and the LSRQ interrupt may be serviced, even though unintended, if enabled.

- The program must make sure that synchronization is not disrupted by unrelated influences at critical sections where timing control loops are used to synchronize parallel code execution. Disabling of nested interrupts is one technique to control this.

Designing Link Port Systems

The ADSP-21161 processor link ports support I/O with peripherals and other processor link ports. While link ports require few connections, there are a number of design issues that systems using these ports must accommodate.

Terminations for Link Transmission Lines

The link ports are designed to allow long distance connections to be made between the driver and the receiver. This is possible because the links are self-synchronizing—the clock and data are transmitted together. Only relative delay, not absolute delay between clock and data is relevant.

In addition, the LxACK signal inhibits transmission of the next word, not of the current nibble or byte. For example, the current word is always allowed to complete transmission. This allows delays of 3 to 5 cycles for the LxACK signal to reach the transmitter.

The links are designed to drive transmission lines with characteristic impedances of 50Ω or greater. A higher transmission line impedance reduces the on-chip effect of driver impedance variations for distances longer than six inches.

The ADSP-21161 processor contains internal series resistance equivalent to 50Ω on all I/O drivers except the CLKIN and XTAL pins. Therefore, for traces longer than six inches, external series resisters on control, link port data, clock or frame sync pins are not required to dampen reflections from transmission line effects for point-to-point connections.

Peripheral I/O Using Link Ports

The example shown in Figure 9-9 shows how a multiprocessing system can use link ports to connect to local memories and I/O devices. An ASIC implements the interface between the link port and DRAM or an I/O device. This minimal hardware solution frees the processor's external bus for other shared-bus communication. The DRAM and ASIC may be implemented on a single 10-pin SIMM module.

Accesses to the DRAM over a link is most efficient under DMA control. The ASIC receives DMA control information from the link port and sets up the access to the DRAM. It unpacks 16-bit data words from the DRAM or packs 8-bit bytes from the link. At the end of the DMA transfer, an interrupt lets the processor send new control information to the ASIC. The ASIC always reverts to receive mode at the end of a transfer. The LxACK signal is deasserted by the ASIC whenever a page change, memory refresh cycle, or any other access to the DRAM occurs.

Memory modules may be shared by multiple DSPs when the link port is bused. Each link port supports 100 Mbyte per second access throughput for either instructions or data. The ASIC is responsible for generating the clock when transmitting to the processor. The ASIC is also responsible for generating sequential DMA addresses based on a start address and word count.

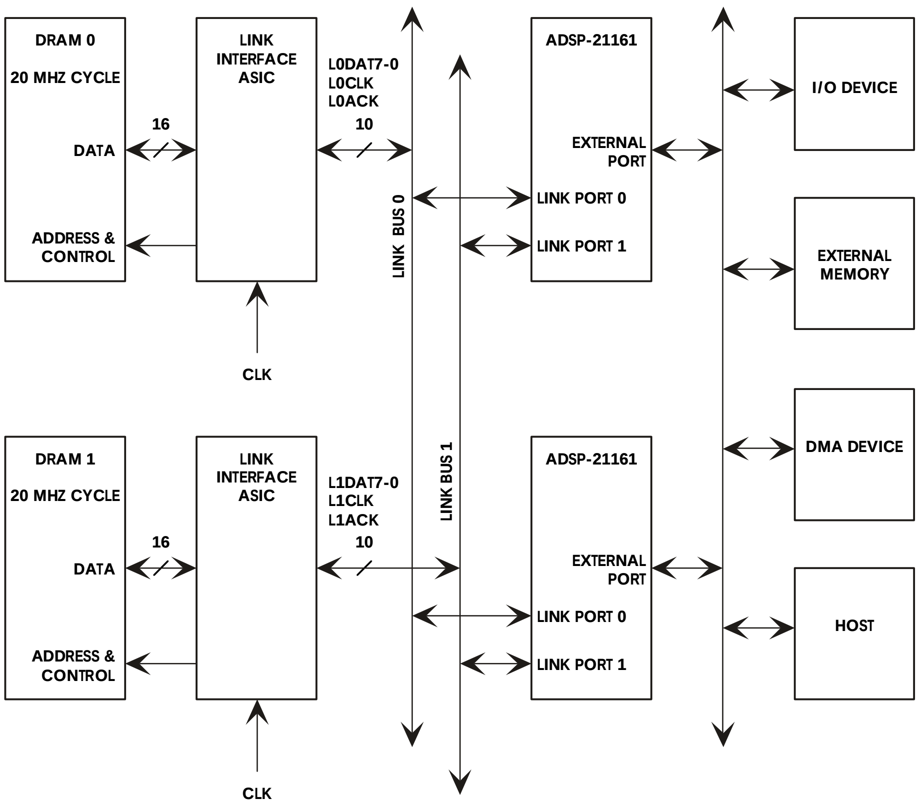


Figure 9-9. Local DRAM With Link Ports

Data Flow Multiprocessing With Link Ports

Figure 9-10 shows examples of different link port communications schemes.

For more information on the multiprocessor interface, see "Multiprocessing System Architectures" on page 7-90.

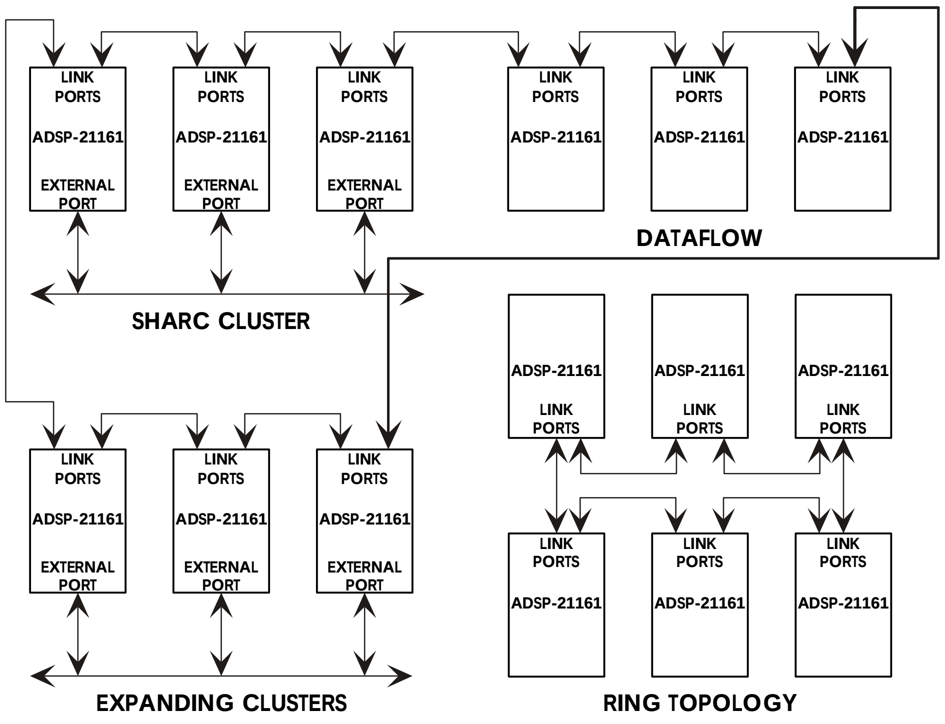


Figure 9-10. Link Port Communication Examples

10 SERIAL PORTS

The ADSP-21161 processor has four independent, synchronous serial ports (SPORTs) that provide an I/O interface to a wide variety of peripheral devices: SPORT0, SPORT1, SPORT2 and SPORT3. Each serial port has its own set of control registers and data buffers. With a range of clock and frame synchronization options, the SPORTs allow a variety of serial communication protocols and provide a glueless hardware interface to many industry-standard data converters and CODECs.

Serial ports can operate at half the full clock rate of the processor, at a maximum data rate of n/2 Mbit/s, where n equals the processor core-clock frequency. Bidirectional (transmit or receive) functions provide greater flexibility for serial communications. Serial port data can be automatically transferred to and from on-chip memory using DMA block transfers. In addition to standard synchronous serial mode, each serial port offers a Time Division Multiplexed (TDM) multichannel mode and I^2S mode.

Serial ports offer the following features and capabilities:

- Two bi-directional channels per serial port, configurable as either transmitters or receivers. Each serial port can be configured as two receivers or two transmitters, permitting two unidirectional streams into or out of the same serial port. This bi-directional functionality provides greater flexibility for serial communications. Two SPORTs can be combined to allow full-duplex, dual-stream communications.

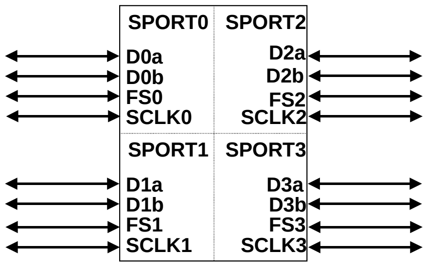
- Double-buffers data – all serial data pins have programmable receive and transmit functions and thus have one transmit and one receive data buffer register and a bi-directional shift register associated with each serial data bin. Double-buffering provides additional time to service the SPORT.
- Compression/decompression – A-law and μ-law hardware companding on transmitted and received words.
- Provides internally-generated serial clock and frame sync signals in a wide range of frequencies, or accepts clock and frame sync input from an external source.
- Performs interrupt-driven, single-word transfers to and from on-chip memory controlled by the processor core.
- Executes DMA transfers to and from on-chip memory. Each SPORT can automatically receive or transmit an entire block of data.
- Permits chaining of DMA operations for multiple data blocks.
- Three operation modes: standard DSP serial, I^2S, and multichannel. In I^2S mode, one or both channels on each SPORT can transmit or receive. Each channel either transmits or receives left and right channels. In standard DSP serial and I^2S modes, when both A and B channels are used, they transmit or receive data simultaneously, sending or receiving bit 0 on the same edge of the serial clock, bit 1 on the next edge of the serial clock, and so on. In multichannel mode, SPORT0 or SPORT1 can receive A channel data, and SPORT2 or SPORT3 transmits A channel data selectively from up to 128 channels of a time-division-multiplexed serial bitstream. This mode is useful for T1 or H.100/H.110 interfaces. In multichannel mode, SPORT0 and SPORT2 work as a pair, and SPORT1 and SPORT3 work as a pair.

- Can be configured to transfer data words between 3 and 32 bits in length, either MSB-first or LSB-first. Words must be between 8 and 32 bits in length for I^2S mode.
- 128-channel TDM is supported in multichannel mode operation.

Receive comparison and 2-dimensional DMA are not supported in the ADSP-21161 processor.

Serial Port Pins

Figure 10-1 shows the pin connections between serial ports. A serial port receives serial data on one of its bi-directional serial data pins configured as an input or transmits serial data on the bi-directional serial data pins configured as an output. It can receive or transmit on both channels simultaneously and uni-directionally, where the pair of data pins can both be configured as either transmitters or receivers.



SPORTS Pin List:
D0a = SPORT0 channel A data (Rx or Tx)
D0b = SPORT0 channel B data (Rx or Tx)
SCLKx0 = SPORT0 Serial clock
FS0 = SPORT0 Frame sync
D1a = SPORT1 channel A data (Rx or Tx)
D1b = SPORT1 channel B data (Rx or Tx)
SCLKx1 = SPORT1 Serial clock
FS1 = SPORT1 Frame sync
D2a = SPORT2 channel A data (Rx or Tx)
D2b = SPORT2 channel B data (Rx or Tx)
SCLKx2 = SPORT2 Serial clock
FS2 = SPORT2 Frame sync
D3a = SPORT3 channel A data (Rx or Tx)
D3b = SPORT3 channel B data (Rx or Tx)
SCLKx3 = SPORT3 Serial clock
FS3 = SPORT3 Frame sync

Figure 10-1. SPORT Pins

ⓘ The A and B channel data pins on each SPORT cannot transmit and receive data simultaneously for full-duplex operation. Two SPORTs must be combined to achieve full-duplex operation. The DDIR bit in the SPCTL register controls the same direction for both the A and B channel pins. Therefore, the direction of the A and B channel on a particular SPORT must be the same.

Serial communications are synchronized to a clock signal. Every data bit must be accompanied by a clock pulse. Each serial port can generate or receive its own clock signal (SCLKx). Internally-generated serial clock frequencies are configured in the DIVx registers. the A and B channel data pins shift data based on the rate of SCLKx.

In addition to the serial clock signal, data may be signaled by a frame synchronization signal. The framing signal can occur at the beginning of an individual word or at the beginning of a block of words. The configuration of frame sync signals depends upon the type of serial device connected to the processor. Each serial port can generate or receive its own frame sync signal (FS) for transmitting or receiving data. Internally-generated frame sync frequencies are configured in the DIVx registers. Both the A and B channel data pins shift data based on the corresponding FSx pin.

Figure 10-2 shows a block diagram of a serial port. The SCLKx and FSx signals are internally connected to all four A and B channel data buffers. The setting of the DDIR bit enables the data buffer path, which, once activated, responds by shifting data in response to a frame sync at the rate of SCLKx. Your application program must use the correct serial port data buffers, according to the value of DDIR bit. The DDIR bit enables the transmit data buffers for the transmission of A and B channel data, or it enables the receive data buffers for the reception of A and B channel data. Inactive data buffers are not used.

ⓘ The DDIR bit in the SPCTLx register affects the operation of the transmit data path or the receive data path. The data path includes the data buffers and the shift registers. When DDIR = 0, the primary and secondary RXx data registers and receive shift registers are activated, and the transmit path is disabled. When DDIR = 1, the primary and secondary TXx data register and transmit shift registers are activated, and the receive path is disabled.

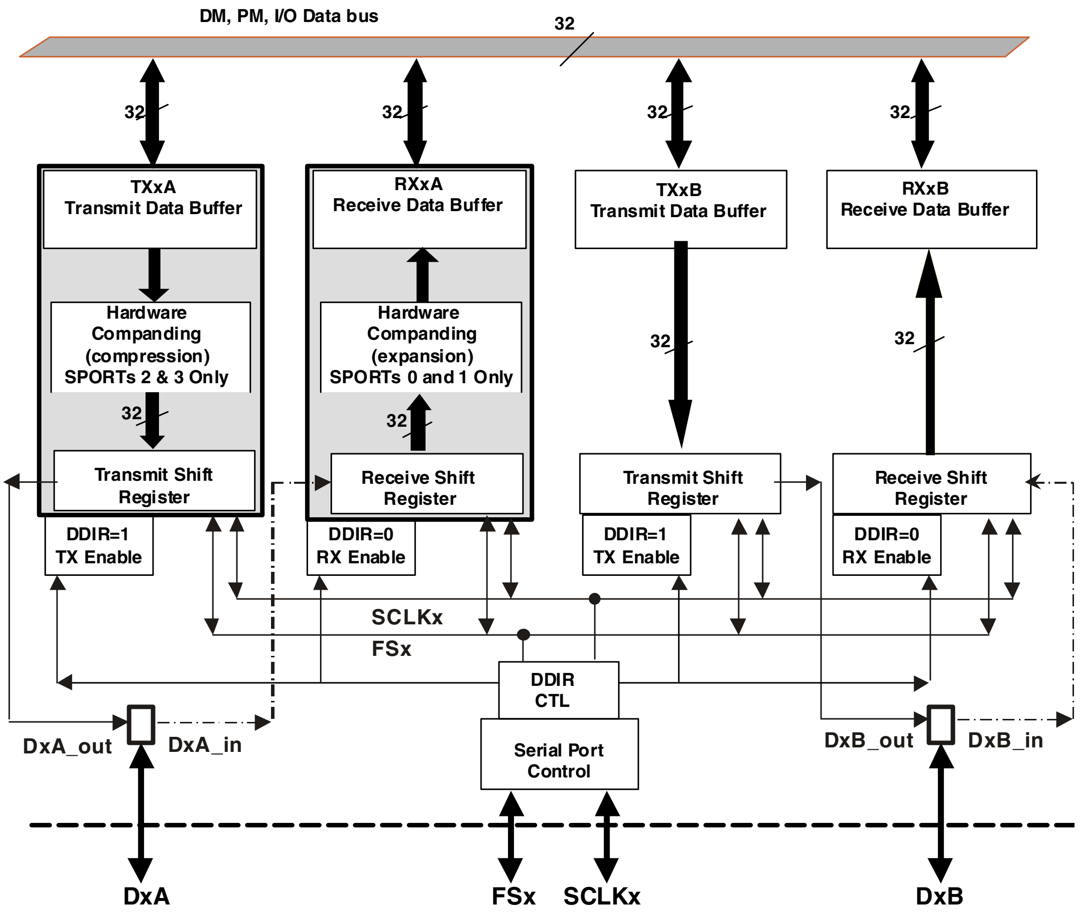


Figure 10-2. Serial Port Block Diagram

If the serial data pin is configured as a serial transmitter, the data to be transmitted is written to the TXxA/TXxB buffer. The data is (optionally) compressed in hardware on the primary A channel (SPORT2 and

SPORT3 only), then automatically transferred to the transmit shift register. Companding is not supported on the secondary B channels, thus the data is automatically transferred from the TXxB buffer to the shift register. The data in the shift register is then shifted out on the SPORT's Dxy pin, synchronous to the SCLKx clock. If framing signals are used, the FSx signal indicates the start of the serial word transmission. The Dxy pin is always driven (for example, three-stated) if the serial port is enabled (SPEN_A or SPEN_B =1 in the SPCTLx control register), unless it is in multichannel mode and an inactive time slot occurs.

When the SPORT is configured as a transmitter (DDIR=1), the TXxA and TXxB registers and the channel transmit shift registers respond to SCLKx and FSx for transmission of data. The receive RXxA and RXxB buffer registers and receive shift registers are inactive and do not respond to SCLKx and FSx signals. Since these registers are inactive, reading from an empty buffer will cause the core to hang indefinitely.

Do not read from the inactive RXxA and RXxB registers (since the receive buffer status is always empty) if the SPORTs are configured as transmitters (DDIR bit = '1' in SPCTL), as this will cause a core hang indefinitely.

If the serial data pin is configured as a serial receiver (DDIR=0), the receive portion of the SPORT shifts in data from the Dxy pin, synchronous to the SCLKx receive clock. If framing signals are used, the FSx signal indicates the beginning of the serial word being received. When an entire word is shifted in on the primary A channel, the data is (optionally) expanded (SPORT0 and SPORT1 only), then automatically transferred to the RXxA buffer. When an entire word is shifted in on the secondary channel, it is automatically transferred to the RXxB buffer (companding is not supported on the secondary B channels).

When the SPORT is configured as a receiver (DDIR=0), the RXxA and RXxB registers, along with the corresponding A and B channel receive shift registers are activated, responding to SCLKx and FSx for reception of data. The transmit TXxA and TXxB buffer registers and transmit A and B shift registers are inactive and do not respond to the SCLKx and FS. Since the TXxA and TXxB registers are inactive, writing to a transmit data buffer will cause the core to hang indefinitely.

Do not write to the inactive TXxA and TXxB registers if the SPORTs are configured as receivers (DDIR bit = '0' in SPCTL). If the core keeps writing to the inactive buffer, the transmit buffer status will become full. Since data is never transmitted out of the deactivated transmit data buffers, this results in a core hang indefinitely.

The SPORTs are not UARTs and cannot communicate with an RS-232 device or any other asynchronous communications protocol. One way to implement RS-232 compatible communications with the ADSP-21161 processor is to use two of the FLAG pins as asynchronous data receive and transmit signals. For an example, see Chapter 11 “Software UART” in the *Digital Signal Processing Applications Using The ADSP-2100 Family, Volume 2*.

SPORT Interrupts

Each serial port has a transmit DMA interrupt and a receive DMA interrupt. For each SPORT, both the A and B channel transmit or receive data buffers share the same interrupt vector. If a given SPORT is configured to transmit data, both the TXxA and TXxB data buffers use the interrupt vector when previous data has been transmitted. If the SPORT is configured to receive data, both the RXxA and RXxB data buffers use the interrupt vector when new data has been received. When serial port DMA is not enabled, interrupts occur based on the SPORT transmit or receive FIFO status. If on the transmit side the FIFO is empty or on the receive side the FIFO is full, interrupts is generated. The priority of the serial port interrupts is shown in Table 10-1.

Table 10-1. Priority of the Serial Port Interrupts

Interrupt Name[1]	Interrupt
SP0I	SPORT0 DMA Channels 0 and 1 (Highest Priority)
SP1I	SPORT1 DMA Channels 2 and 3
SP2I	SPORT2 DMA Channels 4 and 5
SP3I	SPORT3 DMA Channels 6 and 7 (Lowest Priority)

1 The interrupt names are defined in the def21161.h file supplied with the ADSP-21xxx Development Software.

SPORT interrupts occur on the second system clock (CLKIN) after the last bit of the serial word is latched in or driven out.

SPORT Reset

There are two ways to reset the serial ports: a software reset and a hardware reset. Each method has a different effect on the serial port.

A software reset of the SPEN enable bit(s) disables the serial port(s) and aborts any ongoing operations. Status bits are also cleared. The serial ports are ready to start transmitting or receiving data two SCLK cycles after they are enabled in the SPCTLx control register. No serial clocks are lost from this point on.

A hardware reset ($\overline{\text{RESET}}$) disables the whole processor including the serial ports by clearing the SPCTLx control register. Any ongoing operations are aborted.

SPORT Control Registers and Data Buffers

ADSP-21161 processor has four serial ports. Each SPORT has two data paths corresponding to the A and B Channel. These data buffers are TXxA and RXxA (primary) and TXxB and RXxB (secondary). Channel A and B in all four SPORTS operate in parallel, for example, they share clock and control signals. Companding is supported only on primary channels.

For more information, see "Serial Port Registers" on page A-100.

The following is the list of registers that each SPORT has (where x = 0, 1, 2, or 3):

- Four 32-bit, 2-deep data buffers (TXxA/RXxA and TXxB/RXxB)
- One 32-bit clock and frame sync divide register (DIVx)
- One 32-bit control register (SPCTLx)
- Four 32-bit multichannel select receive registers (MR1CSx, MR0CSx)
- Four 32-bit multichannel select transmit registers (MT2CSx, MT3CSx)
- Four 32-bit multichannel receive compand select signals (MR1CCSx, MR0CCSx)
- Four 32-bit multichannel transmit compand select signals (MT2CCSx, MT3CCSx)
- One multichannel control register (SPxyMCTL)

The registers used to control and configure the serial ports are part of the IOP register set. Each SPORT has its own set of 32-bit control registers and data buffers.

The SPORT control registers are programmed by writing to the appropriate address in memory. The symbolic names of the registers and individual control bits can be used in processor programs. The definitions for these symbols are contained in the file `def21161.h` located in the `INCLUDE` directory of the ADSP-21xxx Development Software. The `def21161.h` file is shown in the registers appendix section "Register and Bit #Defines (def21161.h)" on page A-121. All control and status bits in the SPORT registers are active high unless otherwise noted.

Since the SPORT registers are memory-mapped, they cannot be written with data directly from memory. Instead, they must be written from (or read into) core registers, usually one of the general-purpose universal registers of the(`R15-R0`) register file. The SPORT control registers can also be written or read by external devices (for example, another processor or a host processor) to set up a serial port DMA operation.

Table 10-2 provides a complete list of the SPORT registers, showing the memory-mapped IOP address and a brief description of each register.

Table 10-2. SPORT Registers

Register	IOP Address	Reset	Description
SPCTL0	0x1C0	0x0000 0000	SPORT0 serial control register
TX0A	0x1C1	None	SPORT0 transmit data buffer; A channel data
TX0B	0x1C2	None	SPORT0 transmit data buffer; B channel data
RX0A	0x1C3	None	SPORT0 receive data buffer; A channel data
RX0B	0x1C4	None	SPORT0 receive data buffer; B channel data
DIV0	0x1C5	None	SPORT0 divisor for transmit/receive SCLKx0 and FS0
CNT0	0x1C6	None	SPORT0 count register
MR0CS0	0x1C7	None	SPORT0 multichannel receive select 0 (Channels 31-0)
MR0CCS0	0x1C8	None	SPORT0 multichannel receive compand select 0 (Channel 31-0)
MR0CS1	0x1C9	None	SPORT0 multichannel receive select 1 (Channels 63-32)

Table 10-2. SPORT Registers (Cont'd)

Register	IOP Address	Reset	Description
MR0CCS1	0x1CA	None	SPORT0 multichannel receive compand select 1 (Channel 63-32)
MR0CS2	0x1CB	None	SPORT0 multichannel receive select 2 (Channels 95-64)
MR0CCS2	0x1CC	None	SPORT0 multichannel receive compand select 2 (Channel 95-64)
MR0CS3	0x1CD	None	SPORT0 multichannel receive select 3 (Channels 127-96)
MR0CCS3	0x1CE	None	SPORT0 multichannel receive compand select 3 (Channel 127-96)
	0x1CF		Reserved
SPCTL2	0x1D0	0x0000 0000	SPORT2 serial control register
TX2A	0x1D1	None	SPORT2 transmit data buffer; A channel data
TX2B	0x1D2	None	SPORT2 transmit data buffer; B channel data
RX2A	0x1D3	None	SPORT2 receive data buffer; A channel data
RX2B	0x1D4	None	SPORT2 receive data buffer; B channel data
DIV2	0x1D5	None	SPORT2 divisor for transmit/receive SCLKx1 and FS1
CNT2	0x1D6	None	SPORT2 Count Register
MT2CS0	0x1D7	None	SPORT2 multichannel transmit select 0 (Channels 31-0)
MT2CCS0	0x1D8	None	SPORT2 multichannel transmit compand select 0 (Channel 31-0)
MT2CS1	0x1D9	None	SPORT2 multichannel transmit select 1 (Channels 63-32)
MT2CCS1	0x1DA	None	SPORT2 multichannel transmit compand select 1 (Channel 63-32)

Table 10-2. SPORT Registers (Cont'd)

Register	IOP Address	Reset	Description
MT2CS2	0x1DB	None	SPORT2 multichannel transmit select 2 (Channels 95-64)
MT2CCS2	0x1DC	None	SPORT2 multichannel transmit compand select 2 (Channel 95-64)
MT2CS3	0x1DD	None	SPORT2 multichannel transmit select 3 (Channels 127-96)
MT2CCS3	0x1DE	None	SPORT2 multichannel transmit compand select 3 (Channel 127-96)
SP02MCTL	0x1DF	None	SPORTs 0/2 multichannel control register
SPCTL1	0x1E0	0x0000 0000	SPORT1 serial control register
TX1A	0x1E1	None	SPORT1 transmit data buffer; A channel data
TX1B	0x1E2	None	SPORT1 transmit data buffer; B channel data
RX1A	0x1E3	None	SPORT1 receive data buffer; A channel data
RX1B	0x1E4	None	SPORT1 receive data buffer; B channel data
DIV1	0x1E5	None	SPORT1 divisor for transmit/receive SCLKx0 and FS0
CNT1	0x1E6	None	SPORT1 Count Register
MR1CS0	0x1E7	None	SPORT1 multichannel receive select 0 (Channels 31-0)
MR1CCS0	0x1E8	None	SPORT1 multichannel receive compand select 0 (Channel 31-0)
MR1CS1	0x1E9	None	SPORT1 multichannel receive select 1 (Channels 63-32)
MR1CCS1	0x1EA	None	SPORT1 multichannel receive compand select 1 (Channel 63-32)
MR1CS2	0x1EB	None	SPORT1 multichannel receive select 2 (Channels 95-64)

Table 10-2. SPORT Registers (Cont'd)

Register	IOP Address	Reset	Description
MR1CCS2	0x1EC	None	SPORT1 multichannel receive compand select 2 (Channel 95-64)
MR1CS3	0x1ED	None	SPORT1multichannel receive select 3 (Channels 127-96)
MR1CCS3	0x1EE	None	SPORT1 multichannel receive compand select 3 (Channel 127-96)
	0x1EF		Reserved
SPCTL3	0x1F0	0x0000 0000	SPORT3 serial control register
TX3A	0x1F1	None	SPORT3 transmit data buffer; A channel data
TX3B	0x1F2	None	SPORT3 transmit data buffer; B channel data
RX3A	0x1F3	None	SPORT3 receive data buffer; A channel data
RX3B	0x1F4	None	SPORT3 receive data buffer; B channel data
DIV3	0x1F5	None	SPORT3 divisor for transmit/receive SCLKx1 and FS1
CNT3	0x1F6	None	SPORT3 count register
MT3CS0	0x1F7	None	SPORT3 multichannel transmit select 0 (Channels 31-0)
MT3CCS0	0x1F8	None	SPORT3 multichannel transmit compand select 0 (Channel 31-0)
MT3CS1	0x1F9	None	SPORT3 multichannel transmit select 1 (Channels 63-32)
MT3CCS1	0x1FA	None	SPORT3 multichannel transmit compand select 1 (Channel 63-32)
MT3CS2	0x1FB	None	SPORT3 multichannel transmit select 2 (Channels 95-64)
MT3CCS2	0x1FC	None	SPORT3 multichannel transmit compand select 2 (Channel 95-64)

Table 10-2. SPORT Registers (Cont'd)

Register	IOP Address	Reset	Description
MT3CS3	0x1FD	None	SPORT3 multichannel transmit select 3 (Channels 127-96)
MT3CCS3	0x1FE	None	SPORT3 multichannel transmit compand select 3 (Channel 127-96)
SP13MCTL	0x1FF	None	SPORTs 1/3 multichannel control register

Serial Port Control Registers (SPCTLx)

The main control register for each serial port is the serial port control register, SPCTLx. These registers are defined in Figure 10-3 through Figure 10-7. When changing operating modes, a serial port control register should be cleared before the new mode is written to the register.

The Transmit Underflow Status bit (TUVF_A/DERR_A and TUVF_B/DERR_B) is set when the FSx signal occurs from either an external or internal source while the TXxA or TXxB buffer is empty. The internally generated FS may be suppressed whenever TXxA or TXxB is empty by clearing the DITFS control bit.

When DITFS is cleared (the default setting) the frame sync signal (FSx) is dependent upon new data being present in the transmit buffer. The FSx signal is only generated for new data. Setting DITFS to 1 selects data-independent frame syncs which causes the FSx signal to be generated whether or not new data is present. With each FSx signal, the SPORT will transmit the contents of the transmit buffer. Serial port DMA typically keeps the transmit buffer full. When the DMA operation is complete the last bit in the transmit buffer is continuously transmitted.

The DXS_A or DXS_B status bits indicate whether the DXA or DXB buffer is full (11), empty (00), or partially full (10). To test for space in DXA/DXB, test whether DXS_A (bit 30) is equal to zero for the A channel, or whether

DXS_B (bit 27) is equal to zero for the B channel. To test for the presence of any data in DXA/DXB, test whether DXS_A (bit 31) is equal to one for the A channel, or whether DXS_B (bit 28) is equal to one for the B channel.

There is one global control and status register for each paired SPORT (SPORT0 and SPORT2, SPORT1 and SPORT3) for multichannel operation, SP02MCTL and SP13MCTL, to define the number of channels, provide status of the current channel, enable multichannel operation, and set the multichannel frame delay. Since ADSP-21161 processor supports 128 TDM operations, the number of bits is increased to seven and are stored in a separate register, SP02MCTL or SP13MCTL. The SPxyMCTL register is shown in Figure A-35 on page A-111.

The SPCTLx registers control the serial ports' operating modes for the I/O processor. Table 10-3 lists all the bits in SPCTLx.

Table 10-3. SPCTLx Control Bits Comparison in Three SPORT Modes of Operation

Bit	I^{2}S Mode	Standard DSP Serial Mode	Multichannel Mode Receive Control Bits (SPORT0 and SPORT1)	Multichannel Mode Transmit Control Bits (SPORT2 and SPORT3)
0	SPEN_A	SPEN_A	Reserved	Reserved
1	Reserved	DTYPE	DTYPE	DTYPE
2	Reserved	DTYPE	DTYPE	DTYPE
3	Reserved	SENDN	SENDN	SENDN
4	SLEN0	SLEN0	SLEN0	SLEN0
5	SLEN1	SLEN1	SLEN1	SLEN1
6	SLEN2	SLEN2	SLEN2	SLEN2
7	SLEN3	SLEN3	SLEN3	SLEN3
8	SLEN4	SLEN4	SLEN4	SLEN4
9	PACK	PACK	PACK	PACK

Table 10-3. SPCTLx Control Bits Comparison in Three SPORT Modes of Operation (Cont'd)

Bit	I^{2}S Mode	Standard DSP Serial Mode	Multichannel Mode Receive Control Bits (SPORT0 and SPORT1)	Multichannel Mode Transmit Control Bits (SPORT2 and SPORT3)
10	MSTR	ICLK	ICLK	Reserved
11	OPMODE	OPMODE	OPMODE	OPMODE
12	Reserved	CKRE	CKRE	CKRE
13	Reserved	FSR	Reserved	Reserved
14	Reserved	IFS	IRFS	Reserved
15	DITFS	DITFS	Reserved	Reserved
16	L_FIRST	LFS	LRFS	LTDV
17	Reserved	LAFS	Reserved	Reserved
18	SDEN_A	SDEN_A	SDEN_A	SDEN_A
19	SCHEN_A	SCHEN_A	SCHEN_A	SCHEN_A
20	SDEN_B	SDEN_B	Reserved	Reserved
21	SCHEN_B	SCHEN_B	Reserved	Reserved
22	FS_BOTH	FS_BOTH	Reserved	Reserved
23	Reserved	Reserved	Reserved	Reserved
24	SPEN_B	SPEN_B	Reserved	Reserved
25	DDIR	DDIR	Reserved	Reserved
26	DERR_B	DERR_B	Reserved	Reserved
27	DXS_B	DXS_B	Reserved	Reserved
28	DXS_B	DXS_B	Reserved	Reserved
29	DERR_A	DERR_A	ROVF_A	TUVF_A
30	DXS_A	DXS_A	RXS_A	TXS_A
31	DXS_A	DXS_A	RXS_A	TXS_A

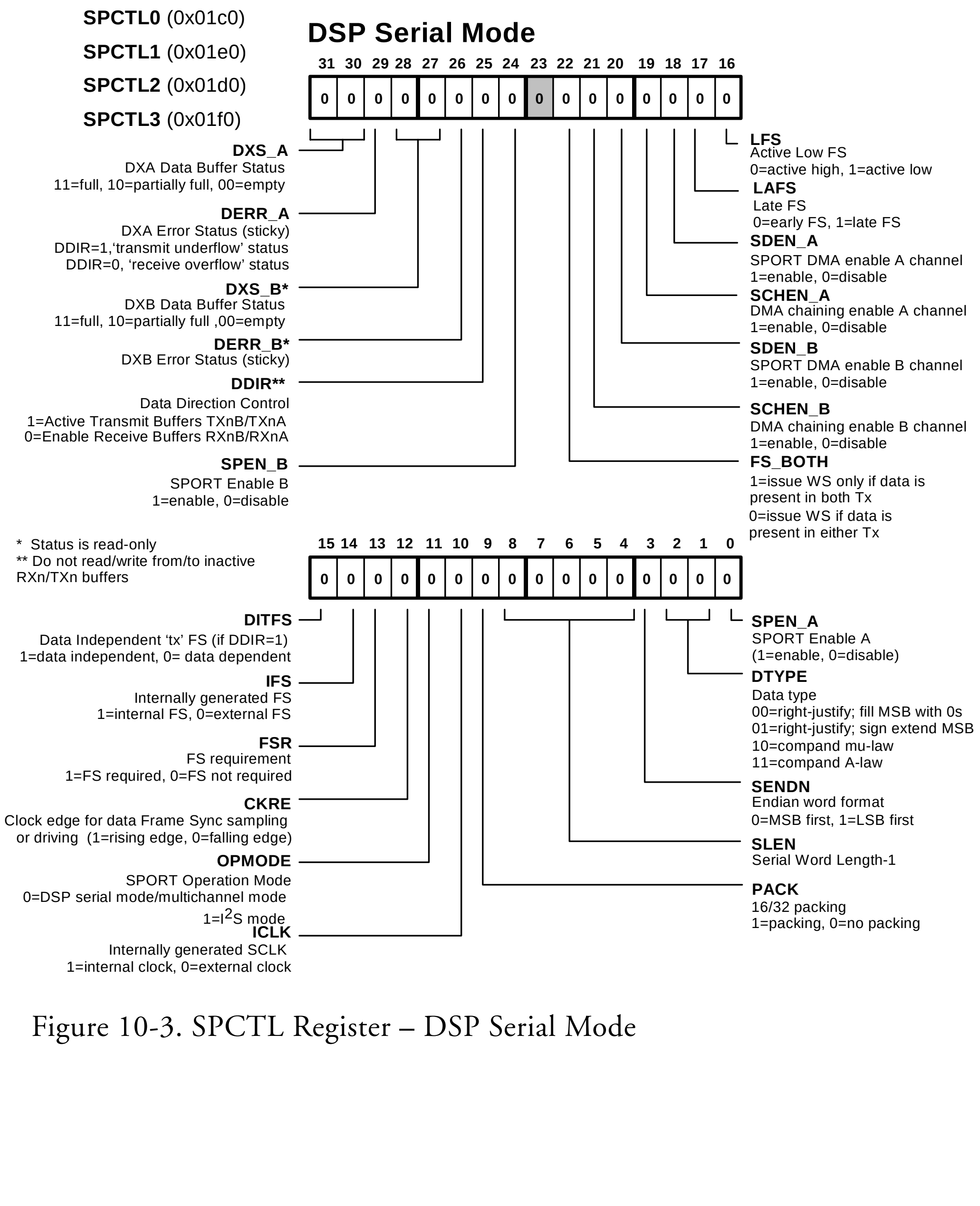
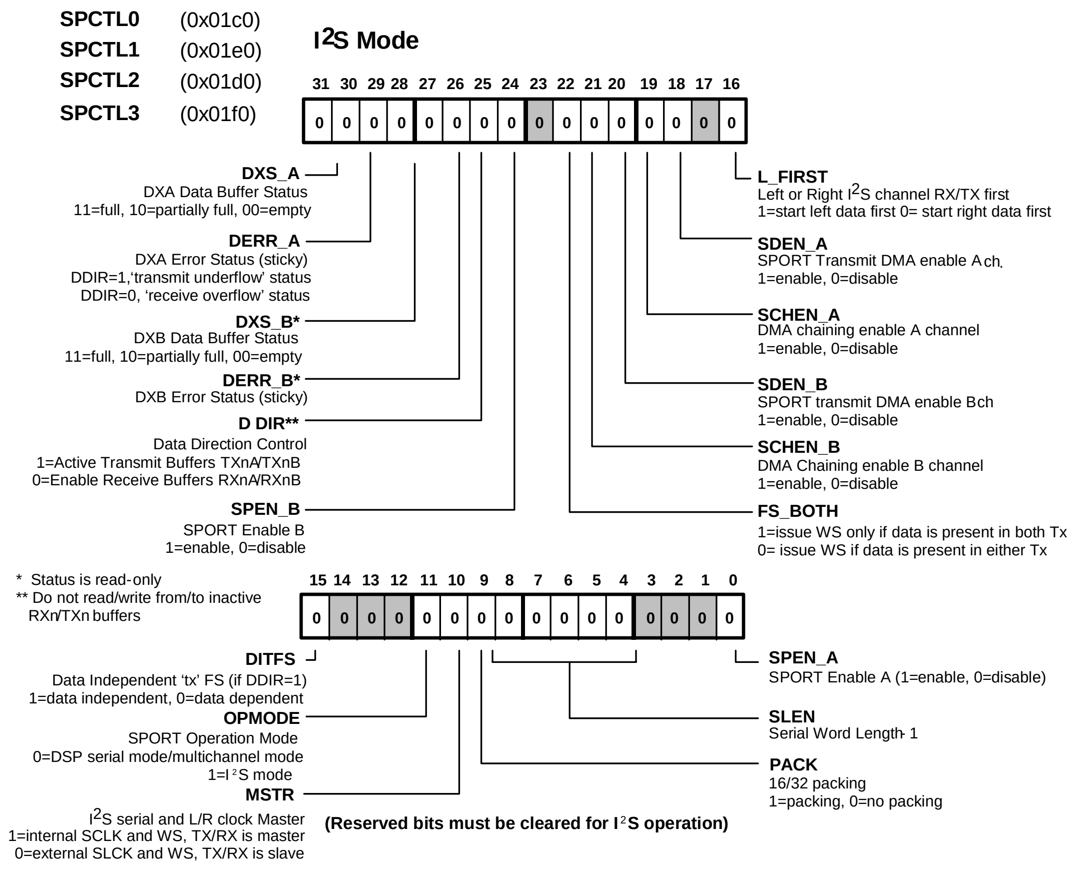

SPCTL0 (0x01c0)
SPCTL1 (0x01e0)
SPCTL2 (0x01d0)
SPCTL3 (0x01f0)
DSP Serial Mode
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
DXS_A
DXA Data Buffer Status
11=full, 10=partially full, 00=empty
DERR_A
DXA Error Status (sticky)
DDIR=1,'transmit underflow' status
DDIR=0, 'receive overflow' status
DXS_B*
DXB Data Buffer Status
11=full, 10=partially full ,00=empty
DERR_B*
DXB Error Status (sticky)
DDIR**
Data Direction Control
1=Active Transmit Buffers TXnB/TXnA
0=Enable Receive Buffers RXnB/RXnA
SPEN_B
SPORT Enable B
1=enable, 0=disable
LFS
Active Low FS
0=active high, 1=active low
LAFS
Late FS
0=early FS, 1=late FS
SDEN_A
SPORT DMA enable A channel
1=enable, 0=disable
SCHEN_A
DMA chaining enable A channel
1=enable, 0=disable
SDEN_B
SPORT DMA enable B channel
1=enable, 0=disable
SCHEN_B
DMA chaining enable B channel
1=enable, 0=disable
FS_BOTH
1=issue WS only if data is present in both Tx
0=issue WS if data is present in either Tx
* Status is read-only
** Do not read/write from/to inactive RXn/TXn buffers
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
DITFS
Data Independent 'tx' FS (if DDIR=1)
1=data independent, 0= data dependent
IFS
Internally generated FS
1=internal FS, 0=external FS
FSR
FS requirement
1=FS required, 0=FS not required
CKRE
Clock edge for data Frame Sync sampling or driving (1=rising edge, 0=falling edge)
OPMODE
SPORT Operation Mode
0=DSP serial mode/multichannel mode
1=I2S mode
ICLK
Internally generated SCLK
1=internal clock, 0=external clock
SPEN_A
SPORT Enable A
(1=enable, 0=disable)
DTYPE
Data type
00=right-justify; fill MSB with 0s
01=right-justify; sign extend MSB
10=compand mu-law
11=compand A-law
SENDN
Endian word format
0=MSB first, 1=LSB first
SLEN
Serial Word Length-1
PACK
16/32 packing
1=packing, 0=no packing

Figure 10-3. SPCTL Register – DSP Serial Mode



SPCTL0 (0x01c0)
SPCTL1 (0x01e0)
SPCTL2 (0x01d0)
SPCTL3 (0x01f0)
I2S Mode
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
DXS_A
DXA Data Buffer Status
11=full, 10=partially full, 00=empty
DERR_A
DXA Error Status (sticky)
DDIR=1,'transmit underflow' status
DDIR=0, 'receive overflow' status
DXS_B*
DXB Data Buffer Status
11=full, 10=partially full, 00=empty
DERR_B*
DXB Error Status (sticky)
D DIR**
Data Direction Control
1=Active Transmit Buffers TXnA/TXnB
0=Enable Receive Buffers RXnA/RXnB
SPEN_B
SPORT Enable B
1=enable, 0=disable
L_FIRST
Left or Right I2S channel RX/TX first
1=start left data first 0= start right data first
SDEN_A
SPORT Transmit DMA enable A ch.
1=enable, 0=disable
SCHEN_A
DMA chaining enable A channel
1=enable, 0=disable
SDEN_B
SPORT transmit DMA enable B ch
1=enable, 0=disable
SCHEN_B
DMA Chaining enable B channel
1=enable, 0=disable
FS_BOTH
1=issue WS only if data is present in both Tx
0= issue WS if data is present in either Tx
* Status is read-only
** Do not read/write from/to inactive RXn/TXn buffers
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
DITFS
Data Independent 'tx' FS (if DDIR=1)
1=data independent, 0=data dependent
OPMODE
SPORT Operation Mode
0=DSP serial mode/multichannel mode
1=I2S mode
MSTR
I2S serial and L/R clock Master
1=internal SCLK and WS, TX/RX is master
0=external SLCK and WS, TX/RX is slave
SPEN_A
SPORT Enable A (1=enable, 0=disable)
SLEN
Serial Word Length- 1
PACK
16/32 packing
1=packing, 0=no packing
(Reserved bits must be cleared for I2S operation)

Figure 10-4. SPCTLx Register – I^{2}S Mode

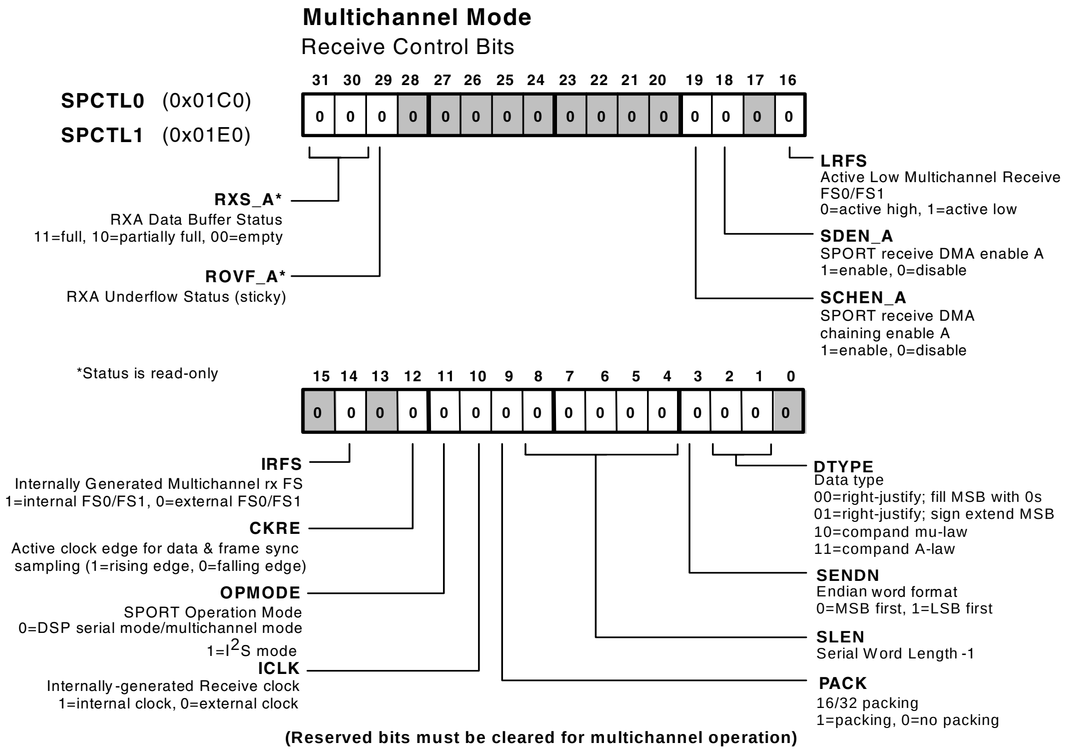

Multichannel Mode
Receive Control Bits
SPCTL0 (0x01C0)
SPCTL1 (0x01E0)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RXS_A*
RXA Data Buffer Status
11=full, 10=partially full, 00=empty
ROVF_A*
RXA Underflow Status (sticky)
LRFS
Active Low Multichannel Receive FS0/FS1
0=active high, 1=active low
SDEN_A
SPORT receive DMA enable A
1=enable, 0=disable
SCHEN_A
SPORT receive DMA chaining enable A
1=enable, 0=disable
*Status is read-only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
IRFS
Internally Generated Multichannel rx FS
1=internal FS0/FS1, 0=external FS0/FS1
CKRE
Active clock edge for data & frame sync sampling (1=rising edge, 0=falling edge)
OPMODE
SPORT Operation Mode
0=DSP serial mode/multichannel mode
1=I2S mode
ICLK
Internally-generated Receive clock
1=internal clock, 0=external clock
DTYPE
Data type
00=right-justify; fill MSB with 0s
01=right-justify; sign extend MSB
10=compand mu-law
11=compand A-law
SENDN
Endian word format
0=MSB first, 1=LSB first
SLEN
Serial Word Length -1
PACK
16/32 packing
1=packing, 0=no packing
(Reserved bits must be cleared for multichannel operation)

Figure 10-5. SPCTL Receive Control Bits in Multichannel Mode for SPORT0 and SPORT1

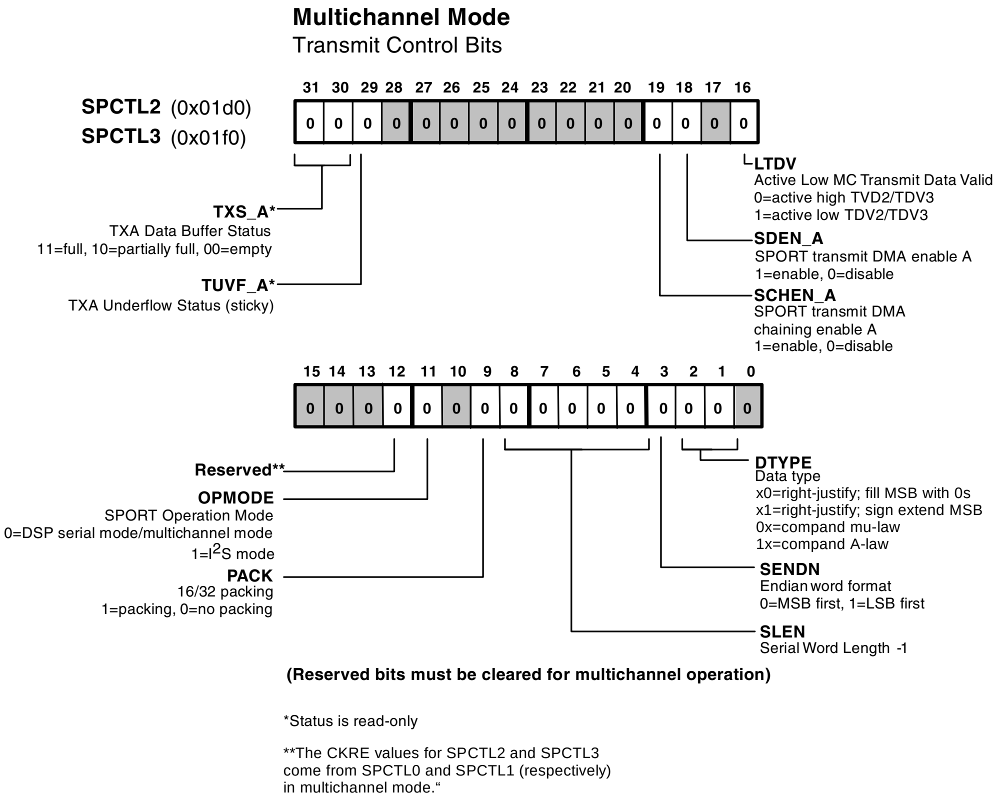


Figure 10-6. SPCTL Transmit Control Bits in Multichannel Mode for SPORT2 and SPORT3

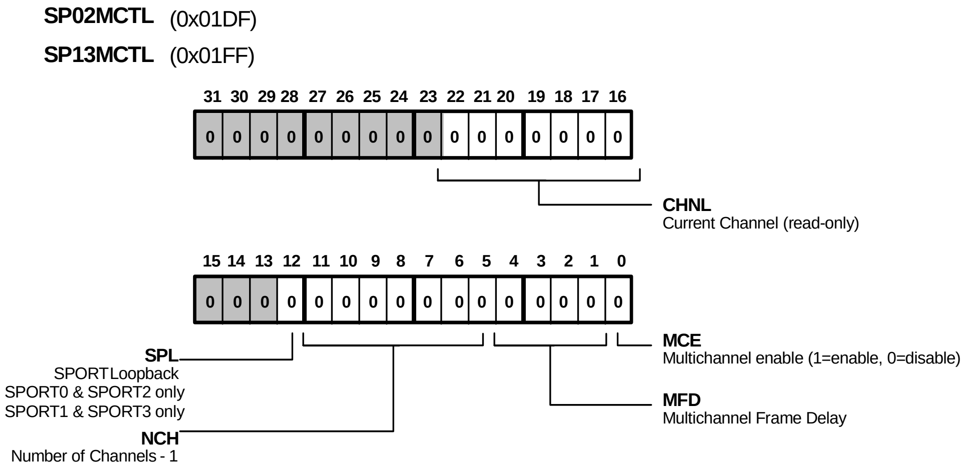


Figure 10-7. SPxyMCTL Control Bits for Multichannel Mode

The following bits control serial port modes and are part of the SPCTLx control registers. Other bits in the SPCTLx registers set up DMA and I/O processor related serial port features.

- **Current Channel Selected.** SP02MCTL or SP13MCTL Bits 16-22 (CHNL). These read-only, sticky status bits identify the currently selected transmit channel slot (0 to 127). These bits apply to multichannel mode only.
- **Clock Rising Edge Select.** SPCTLx Bit 12 (CKRE).This bit selects whether the serial port uses the rising edge (if set, =1) or falling edge (if cleared, =0) of the clock signal for sampling data and the frame sync. This bit applies to DSP standard serial and multichannel modes only.

- **Data Direction Control.** `SPCTLx` Bit 25 (`DDIR`). This bit controls the data direction of the serial port channel A and B pins.

  0 = SPORT is configured to receive on both channels A and B

  1 = SPORT is configured to transmit on both channels A and B

  When configured to receive, the `RXxA` and `RXxB` buffers are activated, while the receive shift registers are controlled by `SCLKx` and `FS`. The `TXxA` and `TXxB` buffers are inactive.

  When configured to transmit, the `TXxA` and `TXxB` buffers are activated, while the transmit shift registers are controlled by `SCLKx` and `FSx`. The `RXxA` and `RXxB` buffers are inactive. This bit applies to all registers for I^{2}S and DSP standard serial modes.

Reading from or writing to inactive buffers will cause a core hang indefinitely until the SPORT is cleared. A hardware reset or host reset will clear the SPORT.

- **Data Independent Transmit Frame Sync Select.** `SPCTLx` Bit 15 (`DITFS`).This bit selects whether the serial port uses a data-independent transmit frame sync (sync at selected interval, if set to 1) or a data-dependent TFS (sync when data is in the transmit buffer, if cleared to 0) when `DDIR`=1.

  When `DITFS` =0, a transmit `FSx` signal is generated only when new data is in the SPORT channel's transmit data buffer. Applications must also program the `DIVx` register.

  When `DITFS` = 1, a transmit `FSx` signal is generated, regardless of the validity of the data present in the SPORT channel's transmit data buffer. The processor generates the transmit `FSx` signal at the frequency specified by the value loaded in the `DIV` register.
  This bit applies to all `SPCTLx` registers in I^{2}S and DSP standard serial modes, and `SPCTL2` and `SPCTL3` register transmit control for multichannel mode.

- **DXS Data Buffer Status.** SPCTLx Bits 30 and 31(DXS_A) and Bits 27 and 28 (DXS_B). These read-only, sticky bits indicate the status of the serial port's data buffer as follows: 11= buffer full, 00= buffer empty, 10=buffer partially full, 01= reserved. These bits apply to I^2S and DSP standard serial modes.

When the SPORT is configured as a transmitter, these bits reflect transmit buffer status for the TXxA and TXxB registers. When the SPORT is configured as a receiver, these bits reflect receive buffer status for the RXxA and RXxB registers.

- **Data Buffer Error Status (sticky, read-only).** SPCTLx Bit 29 and 26 (DERR).These bits indicate whether the serial transmit operation has underflowed (if set, =1 and DDIR=1) or a receive operation has overflowed (if cleared, =0 and DDIR=0) in the DXA and DXB data buffers. These bits apply to I^2S and DSP standard serial modes.

  When the SPORT is configured as a transmitter, this bit provides transmit underflow status and indicates whether the FSx signal (from internal or external source) occurred while the DXS buffer was empty. The SPORTs transmit data whenever they detect a FSx signal.

  0 = No FS signal occurred.

  1 = FS signal occurred.

  When the SPORT is configured as a receiver, these bits provide receive overflow status. As a receiver, it indicates when the channel has received new data while the RXS_A buffer is full. New data overwrites existing data.

  0 = No new data.

  1 = New data.

- **Data Type Select.** SPCTLx Bits 2-1 (DTYPE).These bits select the companding and MSB data type formatting of serial words loaded into the transmit and receive buffers. The transmit shift register does not zero fill or sign-extend transmit data words. This bit applies to DSP standard serial and multichannel modes only.

  For standard mode, selection of companding mode and MSB format are exclusive:

  00 = Right justify; fill unused MSBs with 0s.

  01 = Right justify; sign-extend into unused MSBs.

  10 = Compand using μ_law. (Primary channels only)

  11 = Compand using A_law. (Primary channels only)

  For multichannel mode, selection of companding mode and MSB format are independent:

  x0 = Right justify; fill unused MSBs with 0s.

  x1 = Right justify; sign-extend into unused MSBs.

  0x = Compand using μ_law.

  1x = Compand using A_law.

- **Frame Sync Both Enable.** SPCTLx Bit 22 (FS_BOTH). This bit applies when the SPORTS channels A and B are configured to transmit data. If set (=1), this bit issues word select only when data is present in **both** transmit buffers, TX0A and TX0B. If cleared (=0), a word select is issued if data is present in either transmit buffers. This bit applies to I^{2}S and DSP standard serial modes only.

- **Internal Transmit Clock Select.** SPCTLx Bit 10 (ICLK). This bit selects the internal (if set, =1) or external (if cleared, =0) transmit or receive clock. This bit applies to DSP standard serial and multichannel modes for SPCTL0 and SPCTL1 registers. In these modes only, set this parameter separately for all four SPORTs, where each SPCTL register contains an ICLK bit.

- **Receive Multichannel Frame Sync Source.** SPCTL0 and SPCTL1 Bit 14 (IRFS).This bit selects whether the serial port uses an internal clock generated frame sync (if set, =1) or an external (if cleared, =0) source. This bit applies to multichannel mode only.
- **Internal Frame Sync Select.** SPCTLx Bit 14 (IFS).This bit selects whether the serial port uses an internal clock generated frame sync (if set, =1) or an external (if cleared, =0) source. This bit applies to DSP standard serial mode only.
- **Late Transmit Frame Sync Select.** SPCTLx Bit 17 (LAFS). This bit selects when to generate the frame sync signal. This bit selects a late frame sync if set (=1) during the first bit of each data word. This bit selects an early frame sync if cleared (=0) during the serial clock cycle immediately preceding the first data bit. This bit applies to DSP standard serial mode only.
- **Left/Right Channel Transmit or Receive First.** SPCTLx Bit 16 (L_FIRST).This bit selects the left channel first (if set, =1) or right channel first (if cleared, =0) for transmit or receive. This bit applies to I^{2}S mode only.
- **Low Active Frame Sync Select.** SPCTLx Bit 16 (LFS).This bit selects the logic level of the (transmit or receive) frame sync signals. Active high (0) is the default. This bit selects an active low frame sync (if set, =1) or active high frame sync (if cleared, =0). This bit applies to DSP standard serial mode only.
- **Active State Multichannel Receive Frame Sync Select.**SPCTL0 and SPCTL1 Bit 16 (LRFS).This bit selects the logic level of the multichannel received frame sync signals as active low (inverted) if set (=1) or active high if cleared (=0). Active high (0) is the default. This bit applies to multichannel modes only.

- **Active State Transmit Data Valid.** SPCTL2 and SPCTL3 Bit 16 (LTDV).This bit selects the logic level of the transmit data valid signals (TDV2, TDV3) pins as active low (inverted) if set (=1) or active high if cleared (=0). These pins are actually FS2 and FS3 reconfigured as outputs during multichannel operation, indicating which timeslots have valid data to transmit. Active high (0) is the default. This bit applies to multichannel mode only.

- **Multichannel Mode Enable.** SP02MCTL and SP13MCTL Bit 0 (MCE). Standard and multichannel modes only. in the registers. One of two configuration bits that enable and disable multichannel mode on both the receive or transmit serial port channels. If MCE is cleared (=0), then multichannel operation is disabled. If MCE is set (=1) and OPMODE is cleared (=0), then multichannel operation is enabled. This bit applies to DSP standard serial and multichannel modes only.

- **Multichannel Frame Delay.** SP02MCTL and SP13MCTL Bit 1-4 (MFD).These bits set the interval, in terms of serial clock cycles, between the multichannel frame sync pulse and the first data bit. These bits provide support for different types of T1 interface devices. Valid values range from 0 to 15. Values of 1 to 15 correspond to the number of intervening serial clock cycles. A value of 0 corresponds to no delay. The multichannel frame sync pulse is concurrent with first data bit. This bit applies multichannel mode only.

- **SPORT Transmit or Receive Master Mode**. SPCTLx Bit 10 (MSTR). This bit selects the clock and word-select source for transmitting or for receiving. If set (=1), the SPORT uses the internal clock, and the word-select source transmitter or receiver is the master. If cleared (=0), the SPORT transmitter or receiver is a slave. This bit applies to I^{2}S mode only.

- **Number of Multichannel Slots** (minus one). SP02MCTL and SP13MCTL Bit 5 -11 (NCH).These bits select the number of channel slots (maximum of 128) to use for multichannel operation. Valid values for actual number of channel slots range from 1 to 128. This bit applies to multichannel mode only. Use the following formula to calculate the value for NCH:
  NCH = Actual number of channel slots -1.

- **SPORT Operation Mode.** SPCTLx Bit 11 (OPMODE). This bit enables if set (=1) or disables if cleared (=0) the I^2S mode. When this bit is set, the processor ignores the MCE bit. When this bit is cleared, the MCE bit determines whether the SPORT is in DSP serial mode (MCE=0) or multichannel mode (MCE=1).

- **16-bit to 32-bit Word Packing Enable.** SPCTLx Bit 9 (PACK).This bit enables (if set, =1) or disables (if cleared, =0) 16- to 32-bit word packing. This bit applies to all operation modes.

- **Frame Sync Required Select.** SPCTLx Bits 13 (FSR).This bit selects whether the serial port requires (if set, =1) or does not require (if cleared, =0) a transfer frame sync. Only a single frame sync signal is required to initiate communications. The frame sync is ignored after the first bit received. This bit applies to DSP standard serial mode only.

- **Receive Overflow Status (read-only, sticky**). SPCTL0 and SPCTL1 Bit 29 (ROVF). These bits indicate when the channel has received new data if set (=1) or not if cleared (=0) while the RXS_A buffer is full. New data overwrites existing data. This bit applies to multi-channel mode only.

- **Receive Data Buffer Status Channel A (read-only).** SPCTL0 and SPCTL1 Bits 30 and 31 (RXS_A). These bits indicate the status of the channel's receive buffer contents as follows: 00 = buffer empty, 01 = reserved, 10 = buffer partially full, 11 = buffer full. These bits apply to multichannel mode only.

- **Serial Port DMA Chaining Enable.** SPCTLx Bits 19 and 21 (SCHEN_A and SCHEN_B).These bits enable (if set, =1) or disable (if cleared, =0) serial port's channels A and B DMA chaining. Bit 21 applies to I^{2}S and DSP standard serial modes only for secondary (B) SPORT channels.

- **Serial Port DMA Enable.** SPCTLx Bits 18 and 20 (SDEN_A and SDEN_B).These bits enable (if set, =1) or disable (if cleared, =0) the serial port's channel DMA. Bit 20 applies to I^{2}S and DSP standard serial modes only for secondary (B) SPORT channels.

- **Serial Word Endian Select.** SPCTLx Bit 3 (SENDN).This bit selects little endian words (LSB first, if set, =1) or big endian words (MSB first, if cleared, =0). This bit applies to DSP standard serial and multichannel modes only.

- **Serial Word Length Select.** SPCTLx Bit 4-8 (SLEN).These bits select the word length in bits. Word sizes can be from 3-bit (SLEN=2) to 32-bit (SLEN=31). These bits apply to all operation modes. Use the following formula to calculate the value for SLEN:
  SLEN = Actual serial word length – 1

SLEN cannot equal 0 or 1.

- **Serial Port Enable.** SPCTLx Bits 0 and 24 (SPEN_A and SPEN_B).This bit enables (if set, =1) or disables (if cleared, =0) the corresponding serial port channel A or B. Clearing this bit aborts any ongoing operation and clears the status bits. The SPORTS are ready to transmit or receive two cycles after enabling. This bit apply to I^{2}S and DSP standard serial modes only.

- **SPORT Loopback Mode.** SP02MCTL or SP13MCTL Bit 12 (SPL). This bit enables, if set (=1), or disables, if cleared (=0), the channel loopback mode. Loopback mode enables you to run internal tests and to debug applications. Loopback works only under the following SPORT configurations:
    - SPORT0 (configured as a receiver or transmitter) together with SPORT2 (configured as a transmitter or receiver). SPORT0 can only be paired with SPORT2, and controlled via the SPL bit in the SP02MCTL register.
    - SPORT1 (configured as a receiver or transmitter) together with SPORT3 (configured as a transmitter or receiver). SPORT1 can only be paired with SPORT3, and controlled via the SPL bit in the SP13MCTL register.

  Either of the two paired SPORTs can be set up to transmit or receive, depending on their DDIR bit configurations. This bit applies to DSP standard serial and I^{2}S modes only.

- **Transmit Underflow Status (sticky, read-only).** SPCTL2 and SPCTL3 Bit 29 (TUVF_A).This bit indicates (if set, =1) whether the multichannel FSx signal (from internal or external source) occurred while the TXS buffer was empty. The SPORTs transmit data whenever they detect an FSx signal. If cleared (=0), No FSx signal occurred. This bit applies to multichannel mode only when the SPORTs are configures as transmitters.

- **Transmit Data Buffer Status (sticky, read-only).** SPCTL2 and SPCTL3 Bits 30 and 31(TXS_A). These bits indicate the status of the serial port channel's transmit buffer as follows: 11=buffer full, 00=buffer empty, 10=buffer partially full. These bits apply to multichannel mode only.

Register Writes and Effect Latency

SPORT register writes are internally completed at the end of the same CLKIN cycle in which they occur. The newly written value to the SPORT register can be read back on the very next cycle. When a read of one of the SPCTLx control registers is immediately followed by a write to that register, the write may take two cycles to complete.

After a write to a SPORT register, control and mode bit changes generally take effect in the second CLKIN cycle after the write is completed. The serial ports are ready to start transmitting or receiving two CLKIN cycles after they are enabled (in the SPCTLx control register). No serial clocks are lost from this point on.

Transmit and Receive Data Buffers

The transmit registers (TX0A, TX0B, TX1A, TX1B, TX2A, TX2B, TX3A, and TX3B) are the 32-bit transmit data buffers for SPORT0, SPORT1, SPORT2, and SPORT3, respectively. These buffers must be loaded with the data to be transmitted if the SPORT is configured to transmit on the A and B channels. The data is loaded automatically by the DMA controller or loaded manually by the program running on the processor core.

The receive registers (RX0A, RX0B, RX1A, RX1B, RX2A, RX2B, RX3A, and RX3B) are the receive data buffers for SPORT0, SPORT1, SPORT2, and SPORT3 respectively. These 32-bit buffers become active when the SPORT is configured to receive data on the A and B channels. When a SPORT is configured as a receiver, the RXxA and RXxB registers are automatically loaded from the receive shifter when a complete word has been received. The data is then loaded to internal memory by the DMA controller or read directly by the program running on the processor core.

Word lengths of less than 32 bits are automatically right-justified in the receive and transmit buffers.

The transmit buffers act like a two-location FIFO because they have a data register plus an output shift register as shown in Figure 10-2 on page 10-5. Two 32-bit words may be stored in the transmit queue at any one time. When the transmit register is loaded and any previous word has been transmitted, the register contents are automatically loaded into the output shifter. An interrupt occurs when the output transmit shifter has been loaded, signifying that the transmit buffer is ready to accept the next word (for example, the transmit buffer is not full). This interrupt does not occur when serial port DMA is enabled or when the corresponding mask bit in the IMASK register is cleared.

In I^2S and DSP Standard serial port modes, the DERR_A and DERR_B overflow/underflow status bits are set when an overflow or underflow occurs. In multichannel mode, the DERR_A bits are redefined due to the fixed-directional functionality of the SPCTLx registers. When the SPCTL0 and SPCTL1 registers are configured for multichannel mode, the receive overflow bit ROVF_A indicates when the A channel has received new data while the RXS_A buffer is full. Similarly, when the SPCTL2 and SPCTL3 registers are configured for multichannel mode, the transmit overflow bit TUVF_A indicates that a new frame sync signal (FS0/FS1) occurred while the TXS_A buffer was empty.

The DERR_A (Bit 29) overflow/underflow status bit in the SPCTLx register becomes fixed in multichannel mode only as either the RUVF_A overflow status bit (SPORTs 0 and 1) or TUVF_A underflow status bit (SPORTs 2 and 3).

When the SPORT is configured as a transmitter (DDIR =1), a transmit underflow status bit is set in the serial port control register when a transmit frame sync occurs and no new data has been loaded into the transmit buffer. The TUVF_A/DERR_A status bit is sticky and is only cleared by disabling the serial port.

When the SPORT is configured as a receiver (DDIR =0), the receive buffers are activated. The receive buffers act like a three-location FIFO because they have two data registers plus an input shift register. Two complete

32-bit words can be stored in the receive buffer while a third word is being shifted in. The third word overwrites the second if the first word has not been read out (by the processor core or the DMA controller). When this happens, the receive overflow status bit is set in the serial port control register. Almost three complete words can be received without the receive buffer being read before overflow occurs. The overflow status is generated on the last bit of the third word. The `ROVF_A/DERR_A` status bit is sticky and is cleared only by disabling the serial port.

An interrupt is generated when the receive buffer has been loaded with a received word (for example, the receive buffer is not empty). When the corresponding bit in the `IMASK` register is set, this interrupt is unmasked.

If your program causes the core processor to attempt to read from an empty receive buffer or a write to a full transmit buffer, the access is delayed until the buffer is accessed by the external I/O device. This delay is called a core processor hang. If you do not know whether the core processor can access the receive or transmit buffer without a hang, the buffer's status should be read first (in `SPCTLx`) to determine if the access can be made.

To support debugging buffer transfers, the ADSP-21161 processor has a Buffer Hang Disable (`BHD`) bit. When set (=1), this bit prevents the processor core from detecting a buffer-related stall condition, permitting debugging of this type of stall condition. For more information, see the `BHD` discussion on on page 6-43.

The status bits in `SPCTLx` are updated during reads and writes from the core processor even when the serial port is disabled. Disable the serial port when writing to the receive buffer or reading from the transmit buffer.

When programming the serial port channel (A or B) as a transmitter, only the corresponding `TXxA` and `TXxB` buffers become active while the receive buffers `RXxA` and `RXxB` remain inactive. Similarly, when the SPORT channel A and B is programmed as receive only the corresponding `RXxA` and `RXxB` is activated. Do not attempt to read or write to inactive data buffers. If the ADSP-21161 processor operates on the inactive transmit or receive buffers while the SPORT is enabled, unpredictable results may occur.

Clock and Frame Sync Frequencies (DIV)

The DIVx registers contain divisor values that determine frequencies for internally generated clocks and frame syncs. These registers are shown Figure 10-8 and in "SPORT Divisor Registers (DIVx)" on page A-112

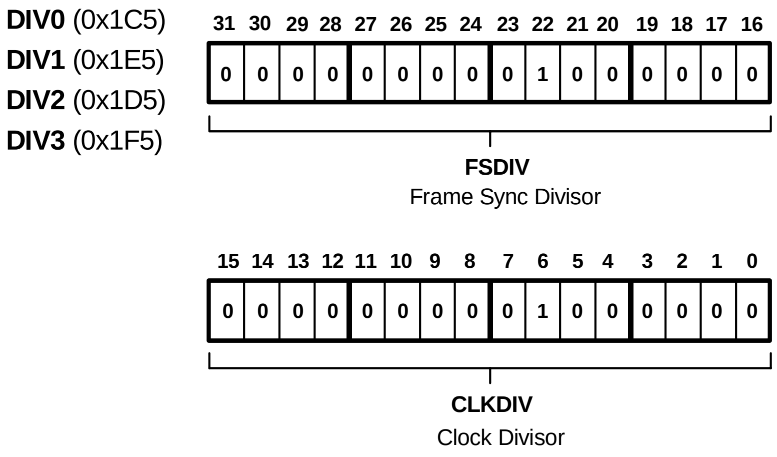


Figure 10-8. DIVx Register

The bit field CLKDIV specifies how many times the processor's internal clock (CCLK) is divided to generate the transmit and receive clocks. The frame sync FS is considered a receive frame sync if the data pins are configured as receivers. Likewise, the frame sync FS is considered a transmit frame sync if the data pins are configured as transmitters. The divisor is a 16-bit value, allowing a wide range of serial clock rates. Use the following equation to calculate the serial clock frequency:

$$\text{FSCLK} = \frac{f_{\text{CCLK}}}{2(\text{CLKDIV} + 1)}$$

The maximum serial clock frequency is equal to half the processor's internal clock (CCLK) frequency, which occurs when CLKDIV is set to zero. Use the following equation to determine the value of CLKDIV to use, given the CCLK frequency and desired serial clock frequency:

$$\text{CLKDIV} = \frac{f_{\text{CCLK}}}{2(f_{\text{SCLK}})} - 1$$

The processor's internal clock (CCLK) is the clock ratio determined by the $\overline{\text{CLKDBL}}$ pin and the CLK_CFG[1-0] pins.

The bit field FSDIV specifies how many transmit or receive clock cycles are counted before generating a FS pulse (when the frame sync is internally generated). In this way, a frame sync can initiate periodic transfers. The counting of serial clock cycles applies to internally or externally generated serial clocks. The formula for the number of cycles between frame sync pulses is:

# of serial clocks between frame syncs = FSDIV + 1

Use the following equation to determine the value of FSDIV, given the serial clock frequency and desired frame sync frequency:

$$\text{FSDIV} = \frac{f_{\text{SCLK}}}{f_{\text{SFS}}} - 1$$

The frame sync is continuously active when FSDIV = 0. The value of FSDIV should not be less than the serial word length minus one (the value of the SLEN field in the serial port control register), as this may cause an external device to abort the current operation or cause other unpredictable results. If the serial port is not being used, the FSDIV divisor can be used as a counter for dividing an external clock or for generating a periodic pulse or periodic interrupt. The serial port must be enabled for this mode of operation to work.

Exercise caution when operating with externally generated transmit clocks near the frequency of half the processor's internal clock. There is a delay between when the clock arrives at the SCLKx pin and when data is output—this delay may limit the receiver's speed of operation. Refer to the data sheet for exact timing specifications. For reliable operation, use full-speed serial clocks only when receiving with an externally generated clock and externally generated frame sync (ICLK = 0, IFS = 0).

Externally-generated late transmit frame syncs also experience a delay from when they arrive to when data is output. This can also limit the maximum serial clock speed. Refer to the *ADSP-21161N DSP Microcomputer Data Sheet* for exact timing specifications.

Data Word Formats

The format of the data words transmitted over the serial ports is configured by the DTYPE, SENDN, SLEN, and PACK bits of the SPCTLx control registers.

Word Length

Serial ports can process word lengths of 3 to 32 bits for serial and multi-channel modes and 8 to 32 bits for I^2S mode. Word length is configured using the 5-bit SLEN field in the SPCTLx control registers. The value of SLEN is given as follows:

SLEN = serial word length – 1

Do not set the SLEN value to zero or one. Words smaller than 32 bits are right-justified in the receive and transmit buffers, residing in the least significant bit positions.

🛇 Although serial ports process word lengths of 3 to 32 bits, transmitting or receiving words smaller than 7 bits at half the full clock rate of the processor may cause incorrect operation when DMA chaining is enabled. Chaining disables the processor's internal I/O bus for several cycles while the new TCB parameters are being loaded. Receive data may be lost (for example, overwritten) during this period.

🛇 Transmitting or receiving words smaller than five bits may cause incorrect operation when all the DMA channels are enabled with no DMA chaining.

Endian Format

Endian format determines whether serial words transmit MSB-first or LSB-first. Endian format is selected by the SENDN bit in the SPCTLx control registers. When SENDN = 0, serial words transmit (or receive) MSB-first. When SENDN = 1, serial words transmit (or receive) LSB-first.

Data Packing and Unpacking

Received data words of 16 bits or less may be packed into 32-bit words, and 32-bit words being transmitted may be unpacked into 16-bit words. Word packing and unpacking is selected by the PACK bit in the SPCTLx control registers.

When PACK = 1 in the control register, two successive words received are packed into a single 32-bit word, and each 32-bit word is unpacked and transmitted as two 16-bit words.

The first 16-bit (or smaller) word is right-justified in bits 15-0 of the packed word, and the second 16-bit (or smaller) word is right-justified in bits 31-16. This applies to both receive (packing) and transmit (unpacking) operations. Companding may be used when word packing or unpacking is being used.

When serial port data packing is enabled, the transmit and receive interrupts are generated for the 32-bit packed words, not for each 16-bit word.

When 16-bit received data is packed into 32-bit words and stored in normal word space in processor internal memory, the 16-bit words can be read or written with short word space addresses.

Data Type

The DTYPE field of the SPCTLx control registers specifies one of four data formats (for non-multichannel operation) shown in Table 10-4. This bit field is reserved for I^{2}S mode.

Table 10-4. DTYPE and Data Formatting (DSP Serial Mode)

DTYPE	Data Formatting
00	Right-justify, zero-fill unused MSBs
01	Right-justify, sign-extend into unused MSBs

Table 10-4. DTYPE and Data Formatting (DSP Serial Mode) (Cont'd)

DTYPE	Data Formatting
10	Compand using μ-law (primary A channels only)
11	Compand using A-law (primary A channels only)

These formats are applied to serial data words loaded into the receive and transmit buffers. Transmit data words are not zero-filled or sign-extended, because only the significant bits are transmitted.

For multichannel operation, the companding selection and MSB-fill selection is independent (Table 10-5).

Table 10-5. DTYPE and Data Formatting (Multichannel)

DTYPE	Data Formatting
x0	Right-justify, zero-fill unused MSBs
x1	Right-justify, sign-extend into unused MSBs
0x	Compand using μ-law (primary A channels only)
1x	Compand using A-law (primary A channels only)

Linear transfers occur if the channel is active and companding is not selected for that channel. Companded transfers occur if the channel is active and companding is selected for that channel. The multichannel compand select registers, MTzCCSx and MRzCCSx, specify the transmit and receive channels that are companded.

Transmit or receive sign extension is selected by bit 0 of DTYPE in the SPCTLx register and is common to all transmit or receive channels. If bit 0 of DTYPE is set, sign extension occurs on selected channels that do not have companding selected. If this bit is not set, the word contains zeros in the MSBs.

Companding

Companding (compressing/expanding) is the process of logarithmically encoding and decoding data to minimize the number of bits that must be sent. The serial ports support the two most widely used companding algorithms, A-law and μ-law, performed according to the CCITT G.711 specification. The type of companding can be selected independently for each SPORT. Companding is selected by the `DTYPE` field of the `SPCTLx` control register.

Companding is supported on the A channel only. SPORTs 2 and 3 primary channels are capable of compression, while SPORTs 0 and 1 primary channels are capable of expansion.

When companding is enabled, the data in the `RX0A` and `RX1A` buffers is the right-justified, sign-extended expanded value of the eight received LSBs. A write to `TX2A` and `TX3A` compresses the 32-bit value to eight LSBs (zero-filled to the width of the transmit word) before it is transmitted. If the 32-bit value is greater than the 13-bit A-law or 14-bit μ-law maximum, it is automatically compressed to the maximum value.

Since the values in the transmit and receive buffers are actually companded in-place, the companding hardware can be used without transmitting (or receiving) any data, for example during testing or debugging. This operation requires one cycle of overhead, as described below. For companding to execute properly, program the SPORT registers prior to loading data values into the SPORT buffers.

To compand data in-place, without transmitting:

1. Enable companding in the `DTYPE` field (bits 2–1) and enable the `DDIR` bit (bit 25) of the `SPCTLx` transmit control register.
2. Write a 32-bit data word to the transmit buffer. The companding is calculated in this cycle.

3. Wait one cycle. A NOP instruction can be used to do this; if a NOP is not inserted, the core is held off for one cycle anyway. This allows the serial port companding hardware to reload the transmit buffer with the companded value.

4. Read the 8-bit companded value from the transmit buffer. The following is an example for companding data in-place.

```
R0=0x2000004;
Dm(0x1f0)=r0;       //Set up SPCTL3
Nop;nop;nop;nop;
R0=0x1234;
Dm(0x1f1)=r0;       // Write 0x1234 to TX3A
Nop;
R0=dm(0x1f1);  // Read compressed value (0x8D) from TX3A
```

To expand data in-place, use the same sequence of operations with the receive buffer instead of the transmit buffer. When expanding data in this way, set the appropriate serial word length (SLEN) in the SPCTLx control register.

With companding enabled, interfacing the serial port to a codec requires little additional programming effort. If companding is not selected, two formats are available for received data words of fewer than 32 bits: one that fills unused MSBs with zeros, and another that sign-extends the MSB into the unused bits.

Clock Signal Options

Each serial port has a clock signal (SCLKx) for transmitting and receiving data on the two associated data pins. The clock signals are configured by the ICLK and CKRE bits of the SPCTLx control registers. The serial clock frequency is configured in the DIVx. A single clock pin clocks both data pins (either configured as inputs or outputs) to receive or transmit data at the same rate.

The serial clock can be independently generated internally or input from an external source. The ICLK bit of the SPCTLx control registers determines the clock source.

When ICLK is set (=1), the clock signal is generated internally by the processor and the SCLKx pins are outputs. The clock frequency is determined by the value of the serial clock divisor (CLKDIV) in the DIVx registers.

When ICLK is cleared (=0), the clock signal is accepted as an input on the SCLKx pins, and the serial clock divisors in the DIVx registers are ignored. The externally generated serial clock does not need to be synchronous with the system clock.

Frame Sync Options

Framing signals indicate the beginning of each serial word transfer. A variety of framing options are available on the SPORTs. The FSx signals are independent and are separately configured in the control register.

Framed Versus Unframed

The use of frame sync signals is optional in serial port communications. The FSR (transmit frame sync required) control bit determines whether frame sync signals are required. Active-low or active-high frame syncs are selected using the LFS bit in DSP serial mode and the LRFS bit in multichannel mode. These bits are located in the SPCTLx control registers.

When FSR is set (=1), a frame sync signal is required for every data word. To allow continuous transmission from the processor, each new data word must be loaded into the transmit buffer before the previous word is shifted out and transmitted.

When FSR is cleared (=0), the corresponding frame sync signal is not required. A single frame sync is required to initiate communications but it is ignored after the first bit is transferred. Data words are then transferred continuously in what is referred to as an unframed mode.

When DMA is enabled in a mode where frame syncs are not required, DMA requests may be held off by chaining or may not be serviced frequently enough to guarantee continuous unframed data flow.

Figure 10-9 illustrates framed serial transfers.

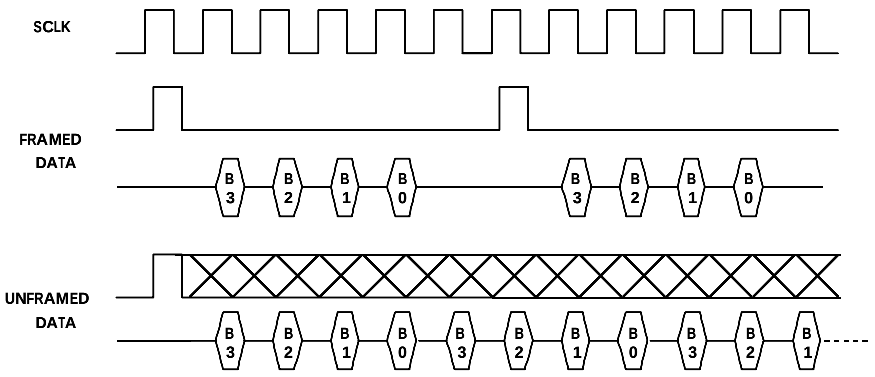


Figure 10-9. Framed Versus Unframed Data

Internal Versus External Frame Syncs

Both transmit and receive frame syncs can be generated internally or input from an external source. The IFS bit of the SPCTLx control register determines the frame sync source.

When IFS is set (=1), the corresponding frame sync signal is generated internally by the processor, and the FSx pin is an output. The frequency of the frame sync signal is determined by the value of the frame sync divisor (FSDIV) in the DIVx register.

When IFS is cleared (=0), the corresponding frame sync signal is accepted as an input on the FSx pins, and the frame sync divisors in the DIVx registers are ignored.

All of the frame sync options are available whether the signal is generated internally or externally.

Active Low Versus Active High Frame Syncs

Frame sync signals may be active high or active low (for example, inverted). Active-low or active-high frame syncs are selected using the LFS bit in DSP serial mode and the LRFS bit in multichannel mode. These bits are located in the SPCTLx control registers. LFS determines the frame sync's logic level:

- When LFS is cleared (=0), the corresponding frame sync signal is active high.
- When LFS is set (=1), the corresponding frame sync signal is active low.

Active high frame syncs are the default. The LFS bit is initialized to 0 after a processor reset.

Sampling Edge for Data and Frame Syncs

Data and frame syncs can be sampled on the rising or falling edges of the serial port clock signals. The CKRE bit of the SPCTLx control registers selects the sampling edge.

For receive/transmit data and frame syncs, setting CKRE to 1 in SPCTLx selects the rising edge of SCLKx. When CKRE is cleared (=0), the processor selects the falling edge. Note that data and frame sync signals change state on the clock edge that is not selected.

For example, the transmit and receive functions of any two serial ports connected together should always select the same value for CKRE so internally generated signals are driven on one edge and received signals are sampled on the opposite edge.

Early Versus Late Frame Syncs

Frame sync signals can be early or late. Frame sync signals can occur during the first bit of each data word or during the serial clock cycle immediately preceding the first bit. The LAFS bit of the SPCTLx control register configures this option.

When LAFS is cleared (=0), early frame syncs are configured. This is the normal mode of operation. In this mode, the first bit of the transmit data word is available (and the first bit of the receive data word is latched) in the serial clock cycle after the frame sync is asserted. The frame sync is not checked again until the entire word has been transmitted (or received). In multi-channel operation, this is the case when frame delay is 1.

If data transmission is continuous in early framing mode (for example, the last bit of each word is immediately followed by the first bit of the next word), the frame sync signal occurs during the last bit of each word. Internally generated frame syncs are asserted for one clock cycle in early framing mode.

When LAFS is set (=1), late frame syncs are configured. In this mode, the first bit of the transmit data word is available (and the first bit of the receive data word is latched) in the same serial clock cycle that the frame sync is asserted. In multichannel operation, this is the case when frame delay is zero. Receive data bits are latched by serial clock edges, but the frame sync signal is checked only during the first bit of each word. Internally generated frame syncs remain asserted for the entire length of the data word in late framing mode. Externally generated frame syncs are only checked during the first bit. They do not need to be asserted after that time period.

Figure 10-10 illustrates the two modes of frame signal timing.

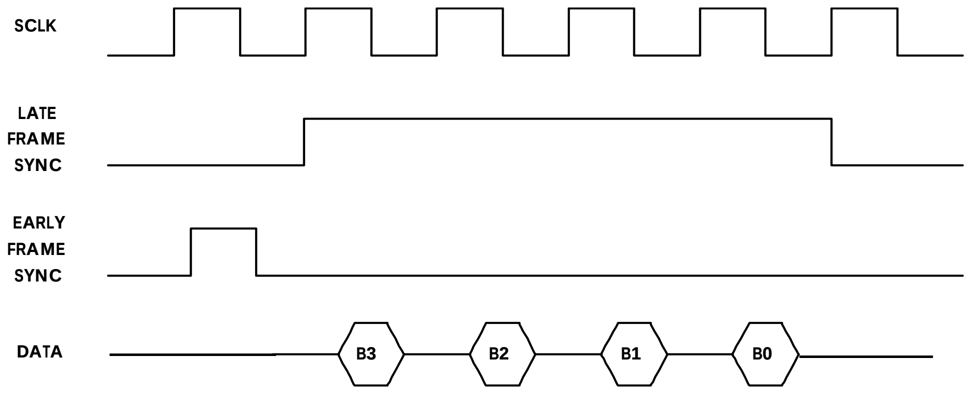


Figure 10-10. Normal vs. Alternate Framing

Data-Independent Transmit Frame Sync

Normally, the internally generated frame sync signal, in the case of transmitting data out of the SPORT (`DDIR` = 1), is output only when the transmit buffer has data ready to transmit. The Data-Independent Transmit Frame Sync (`DITFS`) mode allows the continuous generation of the `FSx` signal, with or without new data in the transmit register. The `DITFS` bit of the `SPCTLx` control register configures this option.

When `DITFS` is cleared (= 0), the internally generated (transmit) frame sync is only output when a new data word has been loaded into the transmit buffer. Once data is loaded into the transmit buffer, it is not transmitted until the next frame sync is generated. This mode of operation allows data to be transmitted only at specific times.

When DITFS is set (=1), the internally generated (transmit) frame sync is output at its programmed interval regardless of whether new data is available in the transmit buffer. Whatever data is present in the transmit buffer is retransmitted with each assertion of frame sync. Depending on the SPORT operating mode, the TUVF_A or DERR_A/DERR_B transmit underflow status bit is set when this occurs (for example, when old data is retransmitted). The TUVF_A or DERR_A/DERR_B status bit is also set if the transmit buffer does not have new data when an externally generated frame sync occurs. In this mode of operation, the first internally generated frame sync is delayed until data has been loaded into the transmit buffer.

If the internally generated frame sync is used, a single write to the transmit data register is required to start the transfer.

SPORT Loopback

When the SPORT loopback bit (SPL) is set in the SP02MCTL or SP13MCTL control register, the serial port is configured in an internal loopback connection as follows: SPORT0 and SPORT2 work as a pair for internal loopback, SPORT1 and SPORT3 work as a pair for internal loopback. The loopback configuration allows the serial ports to be tested internally.

When loopback is configured, the DxA, DxB, SCLKx, and FSx signals of the SPORT0 and SPORT1 are internally connected to the DyA, DyB, SCLKy, and FSy signals of SPORT2 and SPORT3 respectively where x = 0 or 1, and y = 2 or 3.

In loopback mode, either of the two paired SPORTS can be a transmitter or receiver. One SPORT in the loopback pair must be configured as a transmitter, and the other must be configured as a receiver. For example, SPORT0 can be a transmitter and SPORT2 can be a receiver for internal loopback. Or, SPORT0 can be a receiver and SPORT2 can be the transmitter when setting up internal loopback. The processor ignores external activity on the SCLKx, FSx, A and B channel data pins when the SPORT is configured as the receiver. This prevents contention with the internal loopback data transfer.

ⓘ Only transmit clock and transmit frame sync options may be used in loopback mode—programs must ensure that the serial port is set up correctly in the SPCTLx control registers. Multichannel mode is not allowed. Only standard DSP serial and I^{2}S modes support internal loopback.

SPORT Operation Modes

SPORTs operate in three modes: standard DSP serial mode, I^{2}S mode, and multichannel mode. Depending on the operation mode, the control bits are redefined. The operating mode bit (OPMODE) of SPCTLx register selects between I^{2}S mode and non I^{2}S mode (DSP serial port/multichannel mode). In the non I^{2}S mode, bit MCE in SPxyMCTL selects between the DSP serial port mode (standard mode) and multichannel mode. In addition to these bits, the data direction bit (DDIR) selects whether the port is a transmitter or receiver. The definition of all the control bits changes according to DDIR bit. The different operation modes are described in Table 10-6.

Table 10-6. SPORT Operation Modes

OPMODE	MCE	Mode
0	0	Standard DSP serial port
0	1	Multichannel
1	X	I^{2}S
1	1	Reserved

If DDIR bit is set (=1), the SPORT becomes a transmitter and all the other control bits are defined accordingly. Similarly for DDIR =0, the SPORT becomes a receiver. Multichannel mode and companding is not supported for I^{2}S mode.

I^{2}S Mode

I^{2}S is a three-wire serial bus standard protocol for transmission of two channel (stereo) Pulse Code Modulation (PCM) digital audio data, in which each sample is sent MSB-first. Many of today's analog and digital audio front-end devices support the I^{2}S protocol including: audio D/A and A/D converters, PC multimedia audio controllers, digital audio transmitters and receivers that support serial digital audio transmission standards such as AES/EBU, SP/DIF, IEC958, CP-340, and CP-1201, digital audio signal processors, dedicated digital filter chips, and sample rate converters.

The I^{2}S bus transmits audio data and control signals over separate lines. The data line carries two multiplexed data channels: the left channel and the right channel. In I^{2}S mode, if both channels on a SPORT are set up to transmit, then SPORT transmit channels (TXxA and TXxB) transmit simultaneously, each transmitting left and right I^{2}S channels. If both channels on a SPORT are set up to receive, the SPORT receive channels (RXxA and RXxB) receive simultaneously, each receiving left and right I^{2}S channels. Data is transmitted in MSB format.

Multichannel operation and companding are not supported in I^{2}S mode.

Each SPORT transmit or receive channel has channel enable, DMA enable, and chaining enable bits in its SPCTLx control register. The FSx signal is used as the transmit and/or receive word select signal. DMA-driven or interrupt-driven data transfers can also be selected using bits in the SPCTLx register.

Setting Internal Serial Clock and Frame Sync Rates

The serial clock rate (CLKDIV value) for internal clocks can be set using a bit field in the CLKDIV register. For details, see "Clock and Frame Sync Frequencies (DIV)" on page 10-33.

I^{2}S Control Bits

Several bits in the SPCTLx control register enable and configure I^{2}S operation: operation mode (OPMODE), word length (SLEN), I^{2}S channel transfer order (L_FIRST), frame sync (word select) generation (FS_BOTH), master mode enable (MSTR), DMA enable (SDEN), and DMA chaining enable (SCHEN).

Setting Word Length (SLEN)

SPORTs handle data words containing 8 to 32 bits in I^{2}S Mode. Set the bit length for transmit and receive data words. For details, see "Word Length" on page 10-36.

The transmitter sends the MSB of the next word one clock cycle after the word select (TFS) signal changes.

In I^{2}S mode, load the FSDIV register with the same value as SLEN to transmit or receive words continuously. For example, for 8-bit data words (SLEN = 7), set FSDIV = 7.

Selecting Transmit Receive Channel Order (L_FIRST)

In master and slave modes, it is possible to configure the I^{2}S channel that each SPORT channel transmits or receives first. By default, the SPORT channels transmit and receive on the right I^{2}S channel first. The left and right I^{2}S channels are time-duplexed data channels.

To select the channel order, set the L_FIRST bit (= 1) to transmit or receive on left channel first, or clear the L_FIRST bit (= 0) to transmit or receive on right channel first.

Selecting the Frame Sync Options (FS_BOTH)

The processor uses FSx as transmit or receive word select signals, depending on configured direction of the data pins. When the processor generates the transmit word select signal (based on the data in the transmit channels), set FS_BOTH (= 1) to generate the word select signal when both transmit channels contain data. Clear FS_BOTH (= 0) to generate word select signal if either transmit channel contains data.

The word select signal changes one clock cycle before the MSB of the data word transmits, enabling the slave transmitter to derive synchronous timing of the serial data and enabling the receiver to store the previous data word and clear its input for the next one.

When using both SPORT channels (DxA and DxB) as transmitters (FS_BOTH = 1) and MSTR = 1 and DITFS = 0, the processor generates a frame sync signal only when both transmit buffers contain data because both transmitters share the same CLKDIV and FS. For continuous transmission, both transmit buffers must contain new data.

When using both SPORT channels as transmitters and MSTR = 1 and DITFS = 1, the processor generates a frame sync signal at the frequency set by FSDIV = x whether or not the transmit buffers contain new data. In this case, the processor ignores the FS_BOTH bit. The DMA controller or the application is responsible for filling the transmit buffers with data.

Enabling SPORT Master Mode (MSTR)

The SPORTs transmit and receive channels can be configured for master or slave mode. In master mode, the processor generates the word select and serial clock signals for the transmitter or receiver. In slave mode, an external source generates the word select and serial clock signals for the transmitter or receiver. When MSTR is cleared (= 0), the processor uses an external word select and clock source. The SPORT transmitter or receiver is a slave. When MSTR is set (= 1), the processor uses the processor's internal clock for word select and clock source. The SPORT transmitter or receiver is the master.

Enabling SPORT DMA (SDEN)

DMA can be enabled or disabled independently on any of the SPORT's transmit and receive channels. Set SDEN (= 1) to enable DMA and set channel in DMA-driven data transfer mode. Clear SDEN (= 0) to disable DMA and set the channel in an interrupt-driven data transfer mode.

Interrupt-Driven Data Transfer Mode

In this mode, both the A and B channels share a common interrupt vector, regardless of being configured as a transmitter or receiver.

The SPORT generates an interrupt when the transmit buffer has a vacancy or the receive buffer has data. To determine the source of an interrupt, applications must check the TXSx or RXSx data buffer status bits, respectively.

DMA-Driven Data Transfer Mode

Each transmitter and receiver has its own DMA registers. For details, see "Serial Port DMA" on page 6-95. The same DMA channel drives the left and right I^2S channels for the transmitter or the receiver. The software application must de-multiplex the left and right channel data received by the receive buffer, because the left and right data is interleaved in the DMA buffers.

Channel A and B on each SPORT share a common interrupt vector. The DMA controller generates an interrupt at the end of DMA transfer only.

Figure 10-11 shows the relationship between FS (word select), serial clock, and I^2S data. Timing for word select is the same as for frame sync. Note that this example uses early frame sync.

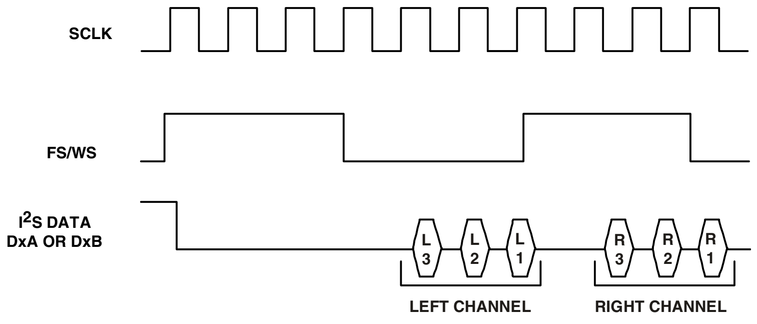


Figure 10-11. Word Select Timing in I^2S Mode

Multichannel Operation

The serial ports offer a multichannel mode of operation which allows the SPORT to communicate in a time-division-multiplexed (TDM) serial system. In multichannel communications, each data word of the serial bit stream occupies a separate channel. Each word belongs to the next consecutive channel. For example, a 24-word block of data contains one word for each of 24 channels.

The serial port can automatically select words for particular channels while ignoring the others. Up to 128 channels are available for transmitting or receiving or both. SPORT0 and SPORT1 receive and SPORT2 and SPORT3 transmit data selectively from any of the 128 channels.

Data companding and DMA transfers can also be used in multichannel mode on channel A. Channel B is not used in multichannel mode.

Although the four SPORTs are programmable for data direction in the standard mode of operation, their programmability is restricted for multichannel operations due to implementation and backward compatibility issues. See the configuration shown in Figure 10-12. The following points summarize these limitations:

1. The primary A channels of SPORT0 and SPORT1 are capable only of expansion, and the primary A channels of SPORT2 and SPORT3 are capable only of compression.
2. In multichannel mode, SPORT0 and SPORT2 work in pairs; SPORT0 is the receive channel, and SPORT2 is the transmit channel. The same is true for SPORT1 and SPORT3.
3. Receive comparison is not supported.

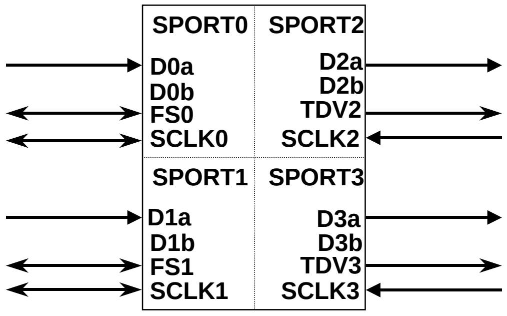


Figure 10-12. SPORT Multichannel Mode Pairings: SPORT0 and SPORT2, SPORT1 and SPORT3

In multichannel mode, the `SCLKx2` and `SCLKx3` pin is an input and is internally connected to its corresponding `SCLKx0` and `SCLKx1` pins. It is not necessary to externally connect `SCLKx2` to `SCLKx0` and `SCLKx1` to `SCLKx3`.

Figure 10-13 shows example timing for a multichannel transfer with SPORT pairing. The transfer has the following characteristics:

- Uses the TDM method in which serial data is sent or received on different channels sharing the same serial bus.
- `FS0` signals start of frame for each multichannel SPORT pairings.
- `FS2` and `FS3` are used as transmit data valid for external logic. These signals are active only during transmit channels. In a SPORT0/SPORT2 multichannel mode pairing, `FS2` is the transmit data valid signal. In a SPORT1/SPORT3 multichannel mode pairing, `FS3` is the transmit data valid signal.
- Receive on channels 0 and 2; transmit on channels 1 and 2.

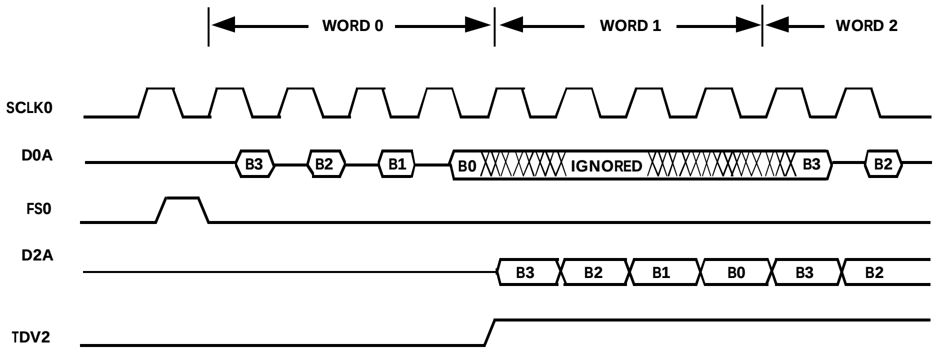


Figure 10-13. Multichannel Operation

Frame Syncs in Multichannel Mode

All receiving and transmitting devices in a multichannel system must have the same timing reference. The `FS0` or `FS1` signal is used for this reference, indicating the start of a block (or frame) of multichannel data words.

When multichannel mode is enabled on a SPORT0/2 or SPORT1/3 pair, both the transmitter and receiver use FS0/FS1 signal as a frame sync. This is true whether FS0 or FS1 is generated internally or externally. The FS0/FS1 signal synchronizes the channels and restarts each multichannel sequence. FS0/FS1 assertion occurs at the beginning of the channel 0 data word.

FS2 or FS3 is used as a transmit data valid signal, which is active during transmission of an enabled word. Because the serial port's D2A and D3A pins are three-stated when the time slot is not active, the FS2/FS3 signal specifies whether D2A/D3A is being driven by the ADSP-21161 processor. The processor drives FS2/FS3 in multichannel mode whether or not DITFS is cleared.

FS2 is renamed TDV2 and FS3 is renamed TDV3 in multichannel mode. These pins become outputs. Do not connect FS2 (TDV2) to FS0, and FS3 (TDV3) to FS1, in multichannel mode. Bus contention between the transmit data valid and multichannel frame sync pins will result.

After the TXxA transmit buffer is loaded, transmission begins and the FS2/FS3 signal is generated. When serial port DMA is used, this may happen several cycles after the multichannel transmission is enabled. If a deterministic start time is required, pre-load the transmit buffer.

Multichannel Control Bits in SPCTL

The SPCTLx control registers contain several bits that enable and configure multichannel operations. Multichannel mode is enabled by setting the MCE bit in the SP02MCTL or SP13MCTL control register:

- When MCE is set (= 1), multichannel operation is enabled.
- When MCE is cleared (= 0), all multichannel operations are disabled.

Multichannel operation is activated three cycles after MCE is set. Internally generated frame sync signals activate four cycles after MCE is set.

Setting the MCE bit enables multichannel operation for both receive and transmit sides of the SPORT0/2 or SPORT1/3 pair. A transmitting SPORT2 or SPORT3 must be in multichannel mode if the receiving SPORT0 or SPORT1 is in multichannel mode.

The number of channels used in multichannel operation is selected by the 7-bit NCH field in the SP02MCTL and SP13MCTL multichannel control register. Set NCH to the actual number of channels minus one:

NCH = Number of Channels – 1

The 7-bit CHNL field in the SP02MCTL and SP13MCTL multichannel control registers indicates the channel that is currently selected during multichannel operation. This field is a read-only status indicator. CHNL(6:0) increments modulo NCH(6:0) as each channel is serviced.

The 4-bit MFD field in the SP02MCTL and SP13MCTL multichannel control registers specifies a delay between the frame sync pulse and the first data bit in multichannel mode. The value of MFD is the number of serial clock cycles of the delay. Multichannel frame delay allows the processor to work with different types of T1 interface devices.

A value of zero for MFD causes the frame sync to be concurrent with the first data bit. The maximum value allowed for MFD is 15. A new frame sync may occur before data from the last frame has been received, because blocks of data occur back to back.

Use a multichannel frame delay of at least one pulse when the processor is generating frame syncs for the multichannel system and the serial clock of the system is equal to CLKIN (the processor clock). If MFD is not set to at least one, the master processor in a multiprocessing system does not recognize the first frame sync after multichannel operation is enabled. All succeeding frame syncs are recognized normally.

Channel Selection Registers

Specific channels can be individually enabled or disabled to select the words that are received and transmitted during multichannel communications. Data words from the enabled channels are received or transmitted, while disabled channel words are ignored. Up to 128 channels are available for transmitting and up to 128 channels for receiving.

The multichannel selection registers enable and disable individual channels. The registers for each serial port are as shown in Table 10-7.

Table 10-7. Multichannel Selection Registers

Register Names	Function

MR0CS(0-3)
MR1CS(0-3)
	Multichannel Receive Select-specifies the active receive channels (4x32-bit registers for 128 channels)

MT2CS(0-3)
MT3CS(0-3)
	Multichannel Transmit Select-specifies the active transmit channels (4x32-bit registers for 128 channels)

MR0CCS(0-3)
MR1CCS(0-3)
	Multichannel Receive Compand Select-specifies which active receive channels (out of 128 channels) are companded

MT2CCS(0-3)
MT3CCS(0-3)
	Multichannel Transmit Compand Select-specifies which active transmit channels (out of 128 channels) are companded

Each of the four multichannel enable and compand select registers are 32-bits in length. These registers provide channel selection for 128 (32 x 4 = 128) channel. Setting a bit enables that channel so that the serial port selects its word from the multiple-word block of data (for either receive or transmit). For example, setting bit 0 in `MR0CS0` or `MT2CS0` selects word 0, setting bit 12 selects word 12, and so on. Setting bit 0 in `MR0CS1` or `MT2CS1` selects word 32, setting bit 12 selects word 44, and so on.

Setting a particular bit to 1 in the `MT2CS(0-3)` or `MT3CS(0-3)` register causes SPORT2 or SPORT3 to transmit the word in that channel's position of the data stream. Clearing the bit in the `MT2CS(0-3)` or `MT3CS(0-2)` register causes SPORT2's `D2A` or SPORT3's `D3A` data transmit pin to three-state during the time slot of that channel.

Setting a particular bit to 1 in the `MR0CS(0-3)` or `MR1CS(0-3)` register causes the serial port to receive the word in that channel's position of the data stream; the received word is loaded into the receive buffer. Clearing the bit in the `MR0CS(0-3)/MR1CS(0-3)` register causes the serial port to ignore the data.

Companding may be selected on a per-channel basis. Setting a bit to 1 in any of the multichannel registers specifies that the data be companded for that channel. A-law or μ-law companding can be selected using the `DTYPE` bits in the `SPCTLx` control registers. SPORT0 and SPORT1 expand selected incoming time slot data, while SPORT2 and SPORT3 compress selected outgoing time slot data.

Transferring Data to Memory

Transmit and receive data can be transferred between the serial ports and on-chip memory with single-word transfers or with DMA block transfers. Both methods are interrupt-driven, using the same internally generated interrupts.

When serial port DMA is not enabled in the `SPCTLx` control registers, the SPORT generates an interrupt every time it receives a data word or starts to transmit a data word. SPORT DMA provides a mechanism for receiving or transmitting an entire block of serial data before the interrupt is generated. The ADSP-21161 processor's on-chip DMA controller handles the DMA transfer, allowing the processor core to continue running until the entire block of data is transmitted or received. Service routines can then operate on the block of data rather than on single words, significantly reducing overhead.

DMA Block Transfers

The ADSP-21161 processor's on-chip DMA controller allows automatic DMA transfers between internal memory and the two serial ports. There are eight DMA channels for serial port operations. Each SPORT has one channel for receiving data and one channel for transmitting data. The serial port DMA channels are numbered as shown in Table 10-8.

Table 10-8. Serial Port DMA Channels

Channel	Data Buffer	Description	Priority
0	RX0A/TX0A	SPORT0 A data	Highest
1	RX0B/TX0B	SPORT0 B data	↓
2	RX1A/TX1A	SPORT1 A data	
3	RX1B/TX1B	SPORT1 B data	
4	RX2A/TX2A	SPORT2 A data	
5	RX2B/TX2B	SPORT2 B data	
6	RX3A/TX3A	SPORT3 A data	
7	RX3B/TX3B	SPORT3 B data	Lowest

Data-direction programmability is supported in standard DSP serial mode and I^{2}S mode. The value of the `DDIR` bit in `SPCTL` (0=`RX`, 1=`TX`) in `SPCTLx` determines whether the receive or transmit register for the SPORT becomes active.

The SPORT DMA channels are assigned higher priority than all other DMA channels (for example, link ports and the external port) because of their relatively low service rate and their inability to hold off incoming data. Having higher priority causes the SPORT DMA transfers to be performed first when multiple DMA requests occur in the same cycle.

Although the DMA transfers are performed with 32-bit words, serial ports can handle word sizes from 3 to 32 bits (8 to 32-bits for I^{2}S mode). If serial words are 16 bits or smaller, they can be packed into 32-bit words for each DMA transfer; this is configured by the PACK bit of the SPCTLx control registers. When serial port data packing is enabled (PACK=1), the transmit and receive interrupts are generated for the 32-bit packed words, not for each 16-bit word.

The following sections present an overview of serial port DMA operations; additional details are covered in the DMA chapter of this manual.

- For information on SPORT DMA Channel Setup, see "Setting Up Serial Port DMA" on page 6-100.
- For information on SPORT DMA Parameter Registers, see "Serial Port DMA" on page 6-95.
- For information on SPORT DMA Chaining, see "Chaining DMA Processes" on page 6-25.

Setting Up DMA on SPORT Channels

Each SPORT DMA channel has an enable bit (SDEN) in its SPCTLx control register. When DMA is disabled for a particular channel, the SPORT generates an interrupt every time it receives a data word or whenever there is a vacancy in the transmit buffer. For more information, see "Single-Word Transfers" on page 10-65.

Each channel also has a DMA chaining enable bit (SCHEN) in its SPCTLx control register.

To set up a serial port DMA channel, write a set of memory buffer parameters to the SPORT DMA parameter registers shown in Table 10-9.

Load the II, IM, and C registers with a starting address for the buffer, an address modifier, and a word count, respectively. These registers can be written from the core processor or from an external processor.

Table 10-9. SPORT DMA Parameter Registers

Register[1]	Description
IIxY	DMA channel. x index; Start address for data buffer
IMxY	DMA channel. x modify; Address increment
CxY	DMA channel. x count; Number of words to transmit
CPxY	DMA channel. x chain pointer; Address next set of data buffer parameters
GPxY	DMA channel x general purpose

1 Y = A or B, and x = 0 - 3

Once serial port DMA is enabled, the processor's DMA controller automatically transfers received data words in the receive buffer to the buffer in internal memory. Likewise, when the serial port is ready to transmit data, the DMA controller automatically transfers a word from internal memory to the transmit buffer. The controller continues these transfers until the entire data buffer is received or transmitted.

When the count register of an active DMA channel reaches zero (0), the SPORT generates the corresponding interrupt.

SPORT DMA Parameter Registers

A DMA channel consists of a set of parameter registers that implements a data buffer in internal memory and the hardware that the serial port uses to request DMA service. The parameter registers for each SPORT DMA channel and their addresses are shown in Table 10-10. These registers are part of the processor's memory-mapped IOP register set.

The DMA channels operate similarly to the processor's data address generators (DAGs). Each channel has an index register (`IIx`) and a modify register (`IMx`) for setting up a data buffer in internal memory. It is necessary to initialize the index register with the starting address of the data buffer. After it transfers each serial I/O word to (or from) the SPORT, the DMA controller adds the modify value to the index register to generate the address for the next DMA transfer. The modify value in the `IM` register is a signed integer, which provides capability for both incrementing and decrementing the buffer pointer.

Each DMA channel has a count register `CxA/CxB`, which must be initialized with a word count that specifies the number of words to transfer. The count register decrements after each DMA transfer on the channel. When the word count reaches zero, the SPORT generates the interrupt for the channel and automatically disables the DMA channel.

Each SPORT DMA channel also has a chain pointer register (`CPxA/CPxB`) and a general-purpose register (`GPxA/GPxB`). The `CPx` register functions in chained DMA operations. The general-purpose registers can be used for any purpose. For more information on SPORT DMA chaining, see "Serial Port DMA" on page 6-95.

Table 10-10. SPORT DMA Parameter Registers Addresses

Register	Address	DMA Channel	SPORT Data Buffer
II0A	0x60	0	RX0A or TX0A
IM0A	0x61	0	RX0A or TX0A
C0A	0x62	0	RX0A or TX0A
CP0A	0x63	0	RX0A or TX0A
GP0A	0x64	0	RX0A or TX0A
Reserved 0x65- 0x67			
II0B	0x80	1	RX0B or TX0B
IM0B	0x81	1	RX0B or TX0B
C0B	0x82	1	RX0B or TX0B
CP0B	0x83	1	RX0B or TX0B
GP0B	0x84	1	RX0B or TX0B
Reserved 0x85 - 0x87			
II1A	0x68	2	RX1A or TX1A

Table 10-10. SPORT DMA Parameter Registers Addresses (Cont'd)

Register	Address	DMA Channel	SPORT Data Buffer
IM1A	0x69	2	RX1A or TX1A
C1A	0x6A	2	RX1A or TX1A
CP1A	0x6B	2	RX1A or TX1A
GP1A	0x6C	2	RX1A or TX1A
Reserved 0x6D - 0x6F			
II1B	0x88	3	RX1B or TX1B
IM1B	0x89	3	RX1B or TX1B
C1B	0x8A	3	RX1B or TX1B
CP1B	0x8B	3	RX1B or TX1B
GP1B	0x8C	3	RX1B or TX1B
Reserved 0x8D - 0x8F			
II2A	0x70	4	RX2A or TX2A
IM2A	0x71	4	RX2A or TX2A
C2A	0x72	4	RX2A or TX2A
CP2A	0x73	4	RX2A or TX2A
GP2A	0x74	4	RX2A or TX2A
Reserved 0x75 - 0x77			
II2B	0x90	5	RX2B or TX2B
IM2B	0x91	5	RX2B or TX2B
C2B	0x92	5	RX2B or TX2B
CP2B	0x93	5	RX2B or TX2B
GP2B	0x94	5	RX2B or TX2B
Reserved 0x95 - 0x97			
II3A	0x78	6	RX3A or TX3A

Table 10-10. SPORT DMA Parameter Registers Addresses (Cont'd)

Register	Address	DMA Channel	SPORT Data Buffer
IM3A	0x79	6	RX3A or TX3A
C3A	0x7A	6	RX3A or TX3A
CP3A	0x7B	6	RX3A or TX3A
GP3A	0x7C	6	RX3A or TX3A
Reserved 0x7D - 0x7F			
II3B	0x98	7	RX3B or TX3B
IM3B	0x99	7	RX3B or TX3B
C3B	0x9A	7	RX3B or TX3B
CP3B	0x9B	7	RX3B or TX3B
GP3B	0x9C	7	RX3B or TX3B

When programming the serial port channel (A or B) as a transmitter only the corresponding `TXxA` and `TXxB` become active, while the receive buffers (`RXxA` and `RXxB`) remain inactive. Similarly, when the SPORT channel A and B is programmed as receive, only the corresponding `RX0A` and `RX0B` is activated.

When performing core-driven transfers, write to the buffer designated by the `DDIR` bit setting in the `SPCTL` register. For DMA-driven transfers, the serial port logic performs the data transfer from internal memory to/from the appropriate buffer depending on `DDIR` bit setting. If the inactive SPORT data buffers are read or written to by core while the port is already being enabled, the core will hang. For example, if a SPORT is programmed to be a transmitter, while at the same time the core reads from the receive buffer of the same SPORT, the core hangs just as it would if it were reading an empty buffer that is currently active. This locks up the core permanently until the SPORT is reset.

Therefore, set the direction bit, the serial port enable bit, and DMA enable bits before initiating any operations on the SPORT data buffers. If the processor operates on the inactive transmit or receive buffers while the SPORT is enabled, it can cause unpredictable results.

SPORT DMA Chaining

In chained DMA operations, the processor's DMA controller automatically sets up another DMA transfer when the contents of the current buffer have been transmitted (or received). The chain pointer register (CPx) functions as a pointer to the next set of buffer parameters stored in memory. The DMA controller automatically downloads these buffer parameters to set up the next DMA sequence. For more information on SPORT DMA chaining, see "Serial Port DMA" on page 6-95.

DMA chaining occurs independently for the transmit and receive channels of each serial port. Each SPORT DMA channel has a chaining enable bit (SCHEN) that when set (= 1) enables DMA chaining and when cleared (= 0) disables DMA chaining. Writing all zeros to the address field of the chain pointer register (CPx) also disables chaining.

Single-Word Transfers

Individual data words may also be transmitted and received by the serial ports, with interrupts occurring as each 32-bit word is transmitted or received. When a serial port is enabled and DMA is disabled, the SPORT DMA interrupts are generated whenever a complete 32-bit word has been received in the receive buffer, or whenever the transmit buffer is not full. Single-word interrupts can be used to implement interrupt-driven I/O on the serial ports.

When the processor core's program reads a word from a serial port's receive buffer or writes a word to its transmit buffer, check the buffer's full/empty status to avoid hanging the processor core. (This can also happen to an external device, for example a host processor, when it is reading or writing a serial port buffer.) The full/empty status can be read in the DXS bits of the SPCTLx. Reading from an empty receive buffer or writing to a full transmit buffer causes the processor (or external device) to hang, waiting for the status to change.

To support debugging buffer transfers, the processor has a Buffer Hang Disable (BHD) bit. When set (= 1), this bit prevents the processor core from detecting a buffer-related stall condition, permitting debugging of this type of stall condition. For more information, see the BHD discussion on page 6-43.

Multiple interrupts can occur if both SPORTs transmit or receive data in the same cycle. Any interrupt can be masked in the IMASK register; if the interrupt is later enabled in IMASK, the corresponding interrupt latch bit in IRPTL must be cleared in case the interrupt has occurred in the meantime.

When serial port data packing is enabled (PACK = 1 in the SPCTLx control registers), the transmit and receive interrupts are generated for 32-bit packed words, not for each 16-bit word.

SPORT Pin/Line Terminations

The ADSP-21161 processor has very fast drivers on all output pins including the serial ports. The edge rate occurs at low-speed serial clock rates. Unlike previous SHARC processors, the ADSP-21161 processor contains internal series resistance equivalent to 50Ω on all input drivers. Therefore, for traces longer than six inches, external series resisters on control, SPORT data, clock or frame sync pins are not required on the processor side of the serial paths to dampen reflections from transmission line effects on point-to-point connections.

SPORT Programming Examples

This section provides two programming examples written for the ADSP-21161 processor. The example in Listing 10-1 demonstrates how the core directly reads from the SPORT receive buffer and writes to the SPORT transmit buffer. The example in Listing 10-2 demonstrates how the core directly writes to the SPORT transmit buffer and reads from the SPORT receive buffer after an interrupt.

Listing 10-1. Core-Driven Sport Loopback Example

```
/*_____________________________________________________________
ADSP-21161 Core-Driven SPORT Loopback Example
This example shows an internally looped-back SPORT 32-bit trans-
fer. The core directly writes to the transfer buffer (TX2A) and
reads from the receive buffer (RX0A).
_______________________________________________________________*/
#include "def21161.h"
#define N 8

.section/pm seg_rth;    /* Reset vector from ldf file */
nop;
jump start;

.section/dm seg_dmda;
.var source[N]= 0X11111111, 0X22222222, 0X33333333, 0X44444444,
0X55555555, 0X66666666, 0X77777777, 0X88888888;

.var dest[N];
/*---------------------Main Routine-------------------------*/
.section/pm seg_pmco;
start:
/* Pointers to source and dest, I0=B0 Automatically */
B0=source;
```

```
L0=@source;
B1=dest;
L1=@dest;

ustat3=dm(SYSCON);
bit clr ustat3 BHD;      /*Disable Core Buffer Hang*/
dm(SYSCON)=ustat3;

bit set mode1 CBUFEN;    /*Enable Circular Buffers*/
r0 = 0x00001000;
/*Set the SPL bit in the SPxxMCTL register to enable loopback*/
dm(SP02MCTL)=r0;

r0 = 0x0;            /*Externally generated clock and framesync*/
dm(DIV0) = r0;
r0 = 0x000021f1;
/* Set bits SPEN_A, SLEN = 32, FSR--enable the A channel, set the
word length to 32 bits, and require frame synch. */
dm(SPCTL0)=r0;

r0=0x00270004;
/*TCLKDIV=[FCCLK(96Mhz)/2xFSCLKx((19.2Mhz)]-1=0x0004 */
/*TFSDIV=[FSCLKx(9.6Mhz)/TFS(.24Mhz)]-1=0x0027 */
dm(DIV2)=r0;
r0=0x20065f1;
/* Set bits SPEN_A, SLEN=32, ICLK, IFS, FSR, DDIR--enable the A
channel, set the word length to 32 bits, generate internal frame-
synch and clock, require frame synch, and set for transmit. */
dm(SPCTL2)=r0;

lcntr = N, do send until LCE;
r1=dm(i0,1);       /*Test data to be transmitted*/
dm(TX2A)=r1;       /*Send data to buffer*/
r0=dm(RX0A);       /*Read data from buffer*/
```

```
send:dm(i1,1)=r0;      /*Store data*/

wait: idle;
jump wait;
```

Listing 10-2. Core-Driven Interrupt Sport Loopback Example

```
/*__________________________________________________________
ADSP-21161 Core-Driven Interrupt SPORT Loopback Example
This example shows an internally looped-back SPORT 32-bit trans-
fer. After receiving an interrupt, the core directly writes to
the transfer buffer (TX2A) and reads from the receive buffer
(RX0A).
____________________________________________________________*/
#include "def21161.h"
#define N 8

.section/pm seg_rth;    /*Reset vector from ldf file*/
nop;
          jump start;

.section/dm seg_dmda;
.var source[N]= 0X11111111, 0X22222222, 0X33333333, 0X44444444,
0X55555555, 0X66666666, 0X77777777, 0X88888888;

.var dest[N];

.section/pm sp0i_svc;     /*Sport 0 Interrupt*/
jump IRQ; rti; rti; rti;

/*----------------Main Routine----------------------------*/
.section/pm seg_pmco;
start:
/* Set pointers for source and dest, I0=B0 automatically */
```

```
B0=source;
L0=@source;
B1=dest;
L1=@dest;

ustat3=dm(SYSCON);
bit clr ustat3 BHD;              /*Disable Core Buffer Hang*/
dm(SYSCON)=ustat3;
bit set imask SP0I |SP2I;        /*Unmask SPORT0&2 Interrupts*/
bit set mode1 CBUFEN | IRPTEN;  /*Enable Circ. buffs & Global
                                   inters*/

r0 = 0x00001000;
/* Set the SPL bit in the SPxxMCTL register to enable loopback */
dm(SP02MCTL)=r0;

r0 = 0x0;     /* Externally generated clock and framesync */
dm(DIV0) = r0;
r0 = 0x000021f1;
/* Set bits SPEN_A, SLEN-32, FSR--enable the A channel, set the
word length to 32 bits, and require frame synch. */
dm(SPCTL0)=r0;

r0=0x00270004;  /* TCLKDIV=[FCCLK(96Mhz)/2xFSCLKx((19.2Mhz)]
                  -1=0x0004 */
               /* TFSDIV=[FSCLKx(9.6Mhz)/TFS(.24Mhz)]-1=0x0027 */
dm(DIV2)=r0;
r0=0x20065f1;
/* Set bits SPEN_A, SLEN=32, ICLK, IFS, FSR, DDIR--enable the A
channel, set the word length to 32 bits, generate internal frame-
synch and clock require frame synch, and set for transmit. */
dm(SPCTL2)=r0;

wait: idle;
```

```
jump wait;

IRQ:               /*Interrupt Service Routine*/
r1=dm(i0,1);       /*Test data to be transmitted*/
dm(TX2A)=r1;       /*Send data to buffer*/
r0=dm(RX0A);       /*Read data from buffer*/
dm(i1,1)=r0;       /*Store data*/
rti;
```

11 SERIAL PERIPHERAL INTERFACE (SPI)

The ADSP-21161 processor is equipped with a synchronous serial peripheral interface port that is compatible with the industry-standard Serial Peripheral Interface (SPI). The SPI port supports communication with a variety of different peripheral devices including CODECs, data converters, sample rate converters, SP/DIF or AES/EBU digital audio transmitters and receivers, LCDs, shift registers, microcontrollers, and FPGA devices with SPI emulation.

The processor's SPI port provides the following features and capabilities:

- A simple four wire interface consisting of two data pins, a device select pin, and a clock pin
- Full-duplex operation that allow the ADSP-21161 processor to transmit and receive data simultaneously on the same port
- Special data formats to accommodate little and big endian data, different word lengths, and packing modes
- Master and slave modes as well as multi-master mode in which the ADSP-21161processor can be connected to up to four other SPI devices
- Open drain outputs to avoid data contention and to support multi-master scenarios
- Programmable baud rates, clock polarities, and phases
- Slave booting from a master SPI device

Functional Description

The SPI interface has two shift registers: the transmit shift register (`TXSR`) and the receive shift register (`RXSR`). `TXSR` serially transmits data and `RXSR` receives data synchronously with the SPI clock signal (`SPICLK`).

Figure 11-1 provides a block diagram of the ADSP-21161 processor SPI interface. The data is shifted into or out of the shift registers on two separate pins: the Master In Slave Out (`MISO`) pin and the Master Out Slave In (`MOSI`) pin.

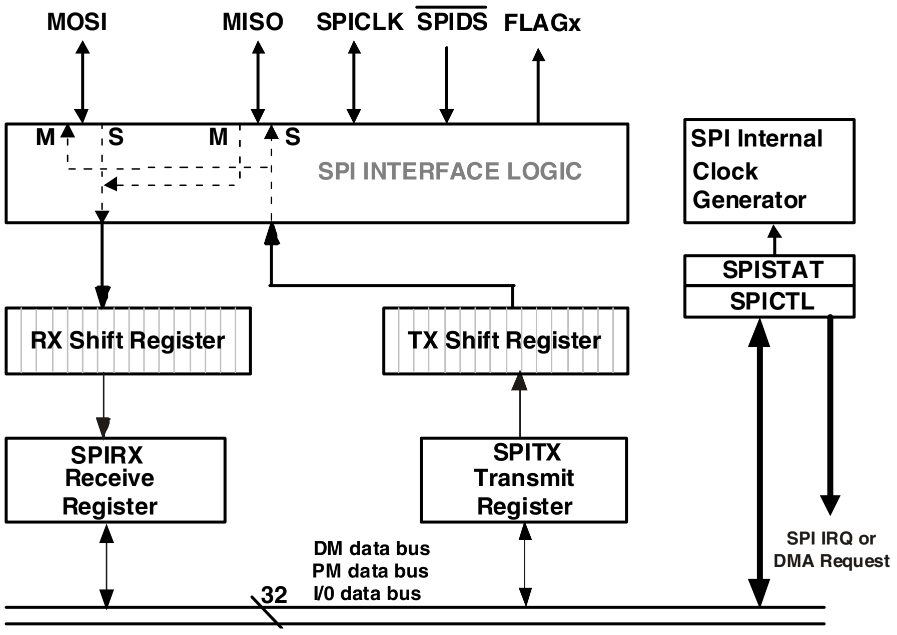


Figure 11-1. SPI Block Diagram

During data transfers one SPI device acts as the SPI master by controlling the data flow. It does this by generating the `SPICLK` and asserting the SPI device select signal ($\overline{\text{SPIDS}}$). The SPI master receives data using the `MISO` pin and transmits using the `MOSI` pin. The other SPI device acts as the SPI slave by receiving new data from the master into it's receive shift register using the MOSI pin. It transmits requested data out the transmit shift register using the MISO pin. The SPI has two 2-deep FIFOs: the transmit data buffer (SPITX) and the receive data buffer (SPIRX). Data to be transmitted is written to SPITX and then automatically transferred into the transmit shift register. Once a full data word has been received in the receive shift register, the data is automatically transferred into SPIRX from which the data can be read from. Programmable FLAGx pins provide slave selection. These pins are connected to the $\overline{\text{SPIDS}}$ of the slave devices.

In a multi-master or multi-device ADSP-21161 processor environment in which multiple ADSP-21161 processors are connected via their SPI ports, all MOSI pins are connected together, all MISO pins are connected together, and the SPICLK pins are connected together as well. The FLAGx pins are connected to each of the slave SPI devices in the system via the $\overline{\text{SPIDS}}$ pins.

SPI Interface Signals

This section describes the signals used to connect the SPI ports in a system that has multiple devices. Figure 11-2 shows the master-slave connections between two ADSP-21161 processors.

SPICLK

The Serial Peripheral Interface Clock (SPICLK) signal is driven by the master device and controls the data transfer rate. It is an output signal if the device is configured as a master and an input signal if the device is configured as a slave. The master transmits data at a variety of baud rates. The SPICLK signal cycles once for each bit transmitted.

The SPICLK signal is a gated clock that is active during data transfers, only for the length of the transferred word. SPICLK is configured with the BAUDR bits in the SPCTL register. The SPICLK clock rate (baud rate) can go as high as the rate given by the expression: f-core clock/8. The number of active

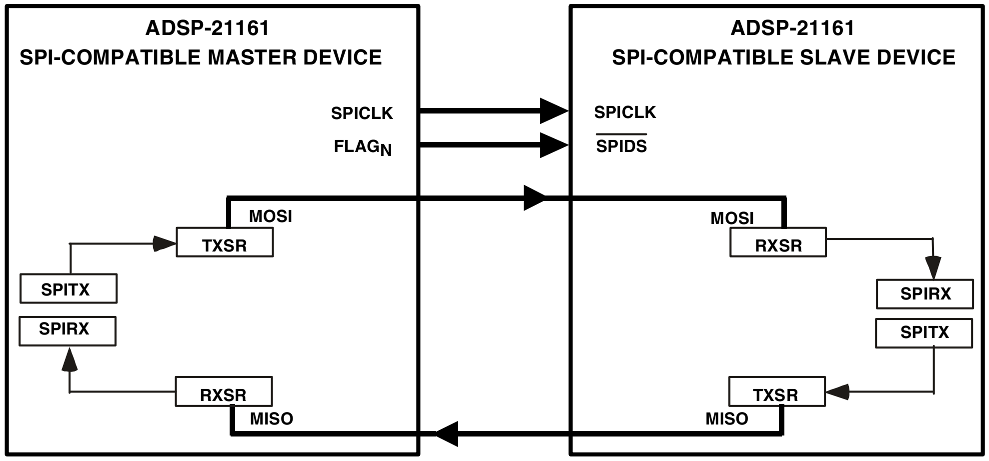


Figure 11-2. Master-Slave Interconnections

clock edges is equal to the number of bits driven on the data lines. Slave devices ignore the serial clock if the slave select input $\overline{\text{SPIDS}}$ is driven inactive.

The SPICLK signal shifts out and shifts in the data driven on the MISO and MOSI lines. The data is shifted out on one clock edge and sampled on the opposite clock edge. To define the transfer format, clock polarity and clock phase relative to data can be programmed into the SPICTL control register.

SPIDS

The Serial Peripheral Interface Slave Device Select ($\overline{\text{SPIDS}}$) signal is an active low signal used to enable an SPI port of an ADSP-21161 processor that is configured as a slave device. This input-only pin behaves like a chip select and is provided by the master device for the slave devices. For a master device, this signal can act as an error signal input in a multi-master environment. In multi-master mode, $\overline{\text{SPIDS}}$ can be asserted (driven low) to a master device to signal that another device is trying to be the master device. In this case, the ADSP-21161 processor's $\overline{\text{SPIDS}}$ signal is used as an input error signal from the slave device. If this signal is asserted low when the device is in master mode, it is considered a multi-master error. For a single-master, multiple-slave configuration in which FLAG0-3 are used as slave selects, $\overline{\text{SPIDS}}$ must be tied high to VDD. For ADSP-21161 processor to ADSP-21161 processor SPI interaction, any of the master processor's FLAG0-3 pins can be used to drive the $\overline{\text{SPIDS}}$ signal on the SPI slave device.

FLAG

The Flag (FLAGx) pins are general-purpose bidirectional I/O data pins. Each FLAG pin can be programmed as an input or output. For SPI, FLAG3-0 pins are used to select slaves in a system that has multiple SPI devices.

When FLAGS are used for SPI to select a slave using and the PSSE and FLS bits are enabled, SPI has higher priority than the core for use of the pins. If PSSE is set (= 1), all of the four flags become slave selects. If a particular GPIO is programmed as output, and the PSSE feature on that flag pin is enabled at the same time, the FLAG register bit is not reflected on the flag pin. However, if the pin is programmed as input, the status of the pin is reflected in the FLAG register. The SPI state machine drives this pin for the slave SPI device and the status is updated in the FLAG register. When using this pin to drive $\overline{\text{SPIDS}}$ while some other device is using it as GPIO, for example, the other device should not drive any data on this pin.

For related flag discussions, see the following sections:

- "Automatic Slave Selection" on page 11-26
- "Core-Based Flag Pins" on page 13-34

MOSI

The Master Out Slave In (MOSI) pin is one of the bidirectional I/O data pins. If the ADSP-21161 processor is configured as a master, the MOSI pin is a data transmit pin used to transmit output data. If the ADSP-21161 processor is configured as a slave, the MOSI pin is a data receive pin used to receive input data. In a system that has multiple SPI devices, data shifts out from the MOSI output pin of the master and into the MOSI input(s) of the slave(s).

MISO

The Master In Slave Out (MISO) pin is one of the bidirectional I/O data pins. If the ADSP-21161 processor is configured as a master, the MISO pin is a data receive pin used to receive input data. If the ADSP-21161 processor is configured as a slave, the MISO pin is a data transmit pin used to transmit output data. In a system that has multiple SPI devices, the data shifts out from the MISO output pin of the slave and into the MISO input pin of the master.

Only one slave may transmit data at any given time. The user application code must ensure that when multiple devices are selected to transmit data from the master, only one slave will respond with data to be transmitted back to the master during the active transfer. The DMISO bit in the SPICTL register can be programmed to accomplish this.

Figure 11-3 illustrates an example of an ADSP-21161 processor SPI interface where the processor is the SPI master. When using the SPI interface, the processor can be directed to alter the conversion resources, mute, modify the volume, and power down the AD1855 stereo DAC.

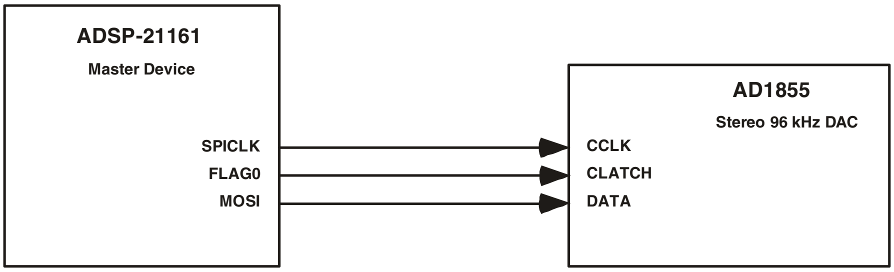


Figure 11-3. ADSP-21161 as SPI Master

Another SPI configuration example, shown in Figure 11-4, illustrates how the ADSP-21161 processor can be used as the SPI slave device. The 8-bit host microcontroller is the SPI master. The processor can be booted via its SPI interface to download user application code and data prior to runtime.

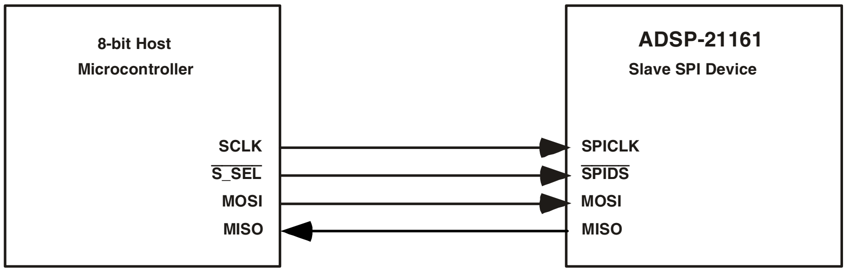


Figure 11-4. ADSP-21161 as Slave SPI Device

SPI Interrupts

The SPI port has two interrupts: a transmit interrupt and a receive interrupt.

- If DMA is enabled, a maskable interrupt occurs when the DMA block transfer has completed.
- If DMA is disabled, the core processor may read the SPIRX register from or write to the SPITX data buffer. To enable an interrupt, program the SPIRX interrupt enable (SPRINT) or the SPITX interrupt enable (SPTINT) in the SPICTL register. The SPIRX and SPITX buffers are memory mapped IOP registers. A maskable interrupt is generated when the receive buffer is not empty or the transmit buffer is not full.

In order for the SPI hardware to work properly, interrupts must always be enabled in the SPICTL register. If interrupts are not wanted or needed, they can be masked at a higher level in the LIRPTL register or the IMASK registers.

The transmit interrupt vector location (0x44) is used for both core driven transmit interrupts and DMA driven transmit interrupts. The receive interrupt vector location (0x40) is used for both core driven receive interrupts and DMA driven receive interrupts. In order to use SPI interrupts, unmask the IRPTEN bit in the MODE1 register, unmask the LPISUMI bit in the IRPTL register, and unmask the SPIRMSK bit or SPITMSK bit in the LIRPTL register.

- See "Interrupt Latch Register (IRPTL)" on page A-27 for IRPTL register bit descriptions.
- See "Link Port Interrupt Register (LIRPTL)" on page A-34 for LIRPTL register bit descriptions.

SPI IOP Registers

The SPI peripheral in the ADSP-21161 processor has two IOP registers and two data buffers: a control register (SPICTL), and a status register (SPISTAT), a receive data buffer (SPIRX) and a transmit data buffer (SPITX). The IOP addresses for the SPI registers are given in Table 11-1.

Table 11-1. IOP Addresses for SPI Registers

Register	IOP Address
SPICTL	0xB4
SPISTAT	0xB5
SPIRX	0xB7
SPITX	0xB6

SPI Control Register (SPICTL)

The SPI Control (SPICTL) register configures and operates the SPI system. It can be read or written to at any time. During active SPI transfers, writes to the SPICTL register are buffered and do not take effect until the current word transfer has completed in the SPI. This occurs prior to the start of the transfer of the next word on the SPI. The SPICTL register enables the SPI interface, selects the device as a master or slave, and determines the data transfer and word size.

The SPICTL register includes the SPI port enable (SPIEN) and SPI DMA enable bits (TDMAEN/RDMAEN). The SPIEN bit can be cleared (=0) to flush the SPI FIFO status. This clears the SPI FIFO status and any error status in the SPISTAT register. It can also be used to disable SPI transmission immediately. Table 11-2 and provides bit descriptions for the SPICTL register. See Figure 11-5 for the SPICTL register diagram.

For revisions 0.3, 1.0 and 1.1 silicon, the SPI transmit and receive FIFOs cannot be cleared by disabling the SPI port via `SPICTL`. In order to clear the SPI receive FIFO, the application program must execute up to two dummy core reads from the `SPIRX` register. The number of reads needed depends on the number of words in the FIFO as shown in the FIFO buffer status. To clear out the `SPITX` FIFOs, clear all the `FLS` bits and then poll the `SPITX` buffer status in the `SPISTAT` register. Note that when the `FLS` bits are not set, there are no slave devices selected. However, the data will still be driven on the appropriate data pin. This FIFO clear operation may be important if you need to reprogram the SPI port to communicate to a new slave, or to change from a master to a slave SPI device.

The default value of the `SPICTL` register at reset is 0x00000000. The value of the `SPICTL` register at slave boot is 0x0A001F81

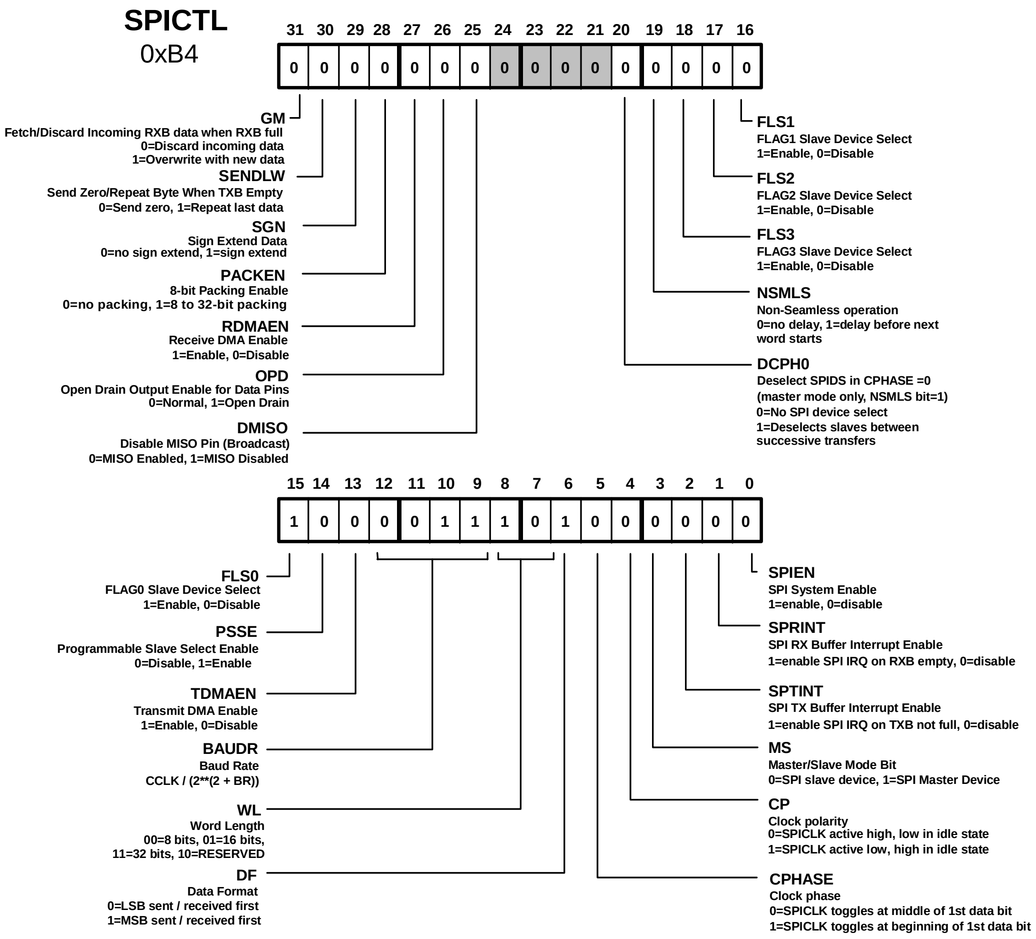

SPICTL
0xB4
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
GM
Fetch/Discard Incoming RXB data when RXB full
0=Discard incoming data
1=Overwrite with new data
SENDLW
Send Zero/Repeat Byte When TXB Empty
0=Send zero, 1=Repeat last data
SGN
Sign Extend Data
0=no sign extend, 1=sign extend
PACKEN
8-bit Packing Enable
0=no packing, 1=8 to 32-bit packing
RDMAEN
Receive DMA Enable
1=Enable, 0=Disable
OPD
Open Drain Output Enable for Data Pins
0=Normal, 1=Open Drain
DMISO
Disable MISO Pin (Broadcast)
0=MISO Enabled, 1=MISO Disabled
FLS1
FLAG1 Slave Device Select
1=Enable, 0=Disable
FLS2
FLAG2 Slave Device Select
1=Enable, 0=Disable
FLS3
FLAG3 Slave Device Select
1=Enable, 0=Disable
NSMLS
Non-Seamless operation
0=no delay, 1=delay before next word starts
DCPH0
Deselect SPIDS in CPHASE =0
(master mode only, NSMLS bit=1)
0=No SPI device select
1=Deselects slaves between successive transfers
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0
FLS0
FLAG0 Slave Device Select
1=Enable, 0=Disable
PSSE
Programmable Slave Select Enable
0=Disable, 1=Enable
TDMAEN
Transmit DMA Enable
1=Enable, 0=Disable
BAUDR
Baud Rate
CCLK / (2**(2 + BR))
WL
Word Length
00=8 bits, 01=16 bits,
11=32 bits, 10=RESERVED
DF
Data Format
0=LSB sent / received first
1=MSB sent / received first
SPIEN
SPI System Enable
1=enable, 0=disable
SPRINT
SPI RX Buffer Interrupt Enable
1=enable SPI IRQ on RXB empty, 0=disable
SPTINT
SPI TX Buffer Interrupt Enable
1=enable SPI IRQ on TXB not full, 0=disable
MS
Master/Slave Mode Bit
0=SPI slave device, 1=SPI Master Device
CP
Clock polarity
0=SPICLK active high, low in idle state
1=SPICLK active low, high in idle state
CPHASE
Clock phase
0=SPICLK toggles at middle of 1st data bit
1=SPICLK toggles at beginning of 1st data bit

Figure 11-5. SPICTL Register

Table 11-2. SPI Control Register Bit Descriptions

Bit(s)	Name	Function
0	SPIEN	**SPI Port Enable.** This bit enables (if set, =1) or disables (if cleared, =0) the SPI system.
1	SPRINT	**SPIRX Interrupt Enable.** This bit enables (if set, =1) or disables (if cleared, =0) an SPI interrupt. An interrupt is generated when the receive buffer is not empty.
2	SPTINT	**SPITX Interrupt Enable.** This bit enables (if set, =1) or disables (if cleared, =0) an SPI interrupt. An interrupt is generated when the transmit buffer is not full.
3	MS	**Master Select.** This bit selects the device as a master device (if set, =1) or a slave device (if cleared, =0).
4	CP	**Clock Polarity.** This bit selects the clock polarity. SPICLK high is the idle state (if set, =1), or SPICLK low is the idle state (if cleared, =0).
5	CPHASE	**Clock Phase.** This bit selects the clock phase transfer format. When set (=1), the SPICLK starts toggling at the beginning of the first data transfer bit. When cleared (=0), the SPICLK starts toggling at the middle of the first data transfer bit. For more information, see Figure 11-7 on page 11-22
6	DF	**Data Format.** This bit selects the data format. When set (=1), the MSB is sent/received first. When cleared (=0), the LSB is sent/received first.
7-8	WL	**Word Length.** These bits selects the word length as follows: 00 = 8 bits, 01 = 16 bits, 11 = 32 bits,10 = reserved.
9-12	BAUDR	Baud Rate. These bits define the SPICLK frequency per the following equation: SPICLK baud rate= Core clock / 2(2 + BR)
13	TDMAEN	**Transmit DMA Enable.** This bit enables (if set, =1) or disables (if cleared, =0) DMA transfers to the transmit buffer. At SPI boot this bit is 0.
Bits 14 to 24 are controlled during master mode.		

Table 11-2. SPI Control Register Bit Descriptions (Cont'd)

Bit(s)	Name	Function
14	PSSE	**Programmable Slave Select Enable.** This bit is used to program the controlled automatic generation of slave device select signals during SPI transfers. This bit enables (if set, =1) or disables (if cleared, =0) the programmable slave select mode. The slave selection is subsequently made using the FLS bits.
15-18	FLS	**Flag Select.** These bits select which flag pins are used when multiple slaves are used (0=disable, 1=enable) as follows: Bit 15= FLAG0 Bit 16= FLAG1 Bit 17= FLAG2 Bit 18= FLAG3 **Note:** O*nly Flag[0] to Flag[3] can be used this way.*
19	NSMLS	**Non-Seamless Operation.** This bit, if set (=1), indicates that after each word transfer there is a delay before the next word transfer starts. The delay is 2.5 SPICLK cycles. When cleared (=0), this bit indicates no delay before the next word starts, a seamless operation.
20	DCPH0	**Deselect SPIDS in CPHASE = 0.** This bit deselects when high (=1) the slaves between successive word transfers in CPhase 0. The slave is selected in master mode using PSSE functionality. This bit has no effect in slave mode for the SPI port. This functionality is valid only when NSMLS =1. It works for CPHASE =0 and CPHASE =1. The standard SPI peripherals use this mode only in CPHASE =0. This bit is cleared (=0) when not in use.
25	DMISO	**Disable MISO Pin**. This bit three-states, (if set, =1) the master in slave out (MISO) pin or (if cleared, =0) enables MISO. This is needed in an environment where master wishes to transmit to various slaves at one time (broadcast). Except for the slave from which it wishes to receive, all other slaves should have this bit set.
26	OPD	**Open Drain Output Enable**. This bit enables an open drain for SPICLK, MOSI and MISO pins if set (=1) or remains normal if cleared (=0). If enabled, the MISO, MOSI and SPICLK is driven only for logic low and pulled up by a 50kΩ resistance for a logic high.

Table 11-2. SPI Control Register Bit Descriptions (Cont'd)

Bit(s)	Name	Function
27	RDMAEN	**Receive DMA Enable.** This bit enables (if set, =1) or disables (if cleared, =0) DMA transfers from the receive buffer. At SPI boot this bit is set to 1 to enable the booting process via the SPI port.
28	PACKEN	**Packing Enable.** This bit enables, if set (=1), 8- to 32-bit packing or disables the packing, if cleared (=0). If this bit is enabled, the receiver packs the received byte whereas the transmitter unpacks the data before sending it. Fore more information on the packing, see "SPI Word Packing" on page 11-24. **Note:** *This bit should be 1 only for 8-bit data word length (WL = 00).*
29	SGN	**Sign Extend.** This bit sign extends the word if set (=1) or does not extend the sign if cleared (=0).
30	SENDLW	**Send Last Data.** When SPITX is empty, setting this bit(=1) re-transmits the last data. Clearing this bit (=0) sends zeros.
31	GM	**Get Data.** This bit fetches incoming data when set (=1) or discards incoming data when cleared (=0). The data that comes in overwrites the previous data in the SPIRX.

Baud Rate Example

The BAUDR bits of the SPICTL register set the baud rate using the following formula:

$$f_{\text{SPICLK}} = \frac{\text{coreclock}}{2^{2 + \text{BAUDR}}}$$

If the core clock is 100 MHz and the BAUDR bits are 0xD (13), the SPICLK frequency is determined as follows:

$$f_{\text{SPICLK}} = \frac{100\text{MHz}}{2^{2 + 13}} = 3052 \text{ Hz}$$

Seamless Operation

The SPI port can transmit words seamlessly without delay by clearing (=0) the NSMLS bit in the SPICTL register. When seamless operation is disabled (NSMLS=1), there is a delay between word transfers from the SPI master. During this delay, the state machine disables and enables the slaves for DCPHO = 1. The delay between words is 2.5 SPICLK cycles.

Some slower slaves need time between data transfers to receive data and move new data for transmitting into the shift register. Set the NSMLS bit in the master device in order to create enough delay for the slave to perform data transfers.

SPI Status Register (SPISTAT)

The SPISTAT register is a read-only register that detects when an SPI transfer is complete, if transmission/reception errors occur, and the status of the SPITX and SPIRX data buffers.

ⓘ For all revisions, a reset flushes the SPI FIFOS. For revisions 1.2 and higher, the SPITX and SPIRX buffers are flushed by disabling the SPI port via the SPIEN bit in the SPICTL register.

Figure 11-6 and Table 11-3 describe the eight status bits of the SPISTAT register.

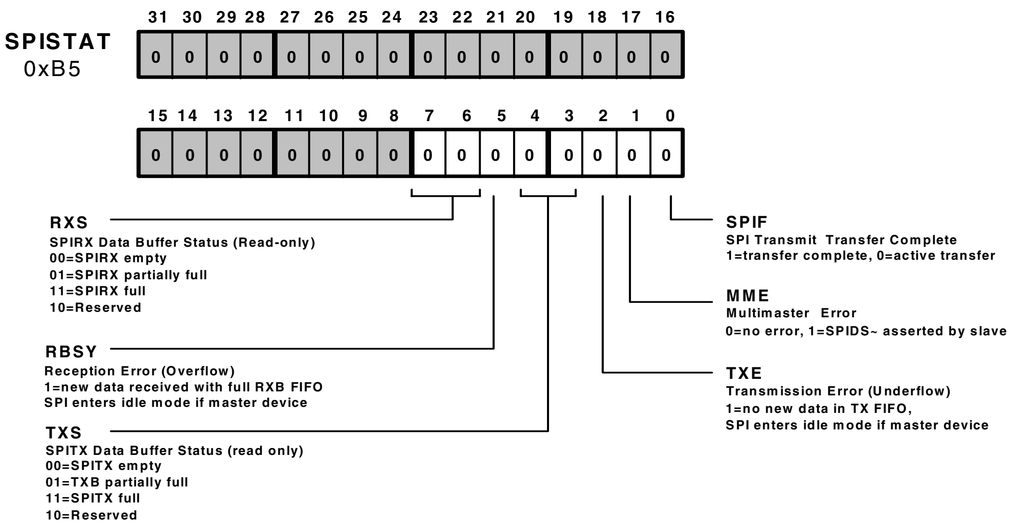

SPISTAT
0xB5
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RXS
SPIRX Data Buffer Status (Read-only)
00=SPIRX empty
01=SPIRX partially full
11=SPIRX full
10=Reserved
RBSY
Reception Error (Overflow)
1=new data received with full RXB FIFO
SPI enters idle mode if master device
TXS
SPITX Data Buffer Status (read only)
00=SPITX empty
01=TXB partially full
11=SPITX full
10=Reserved
SPIF
SPI Transmit Transfer Complete
1=transfer complete, 0=active transfer
MME
Multimaster Error
0=no error, 1=SPIDS~ asserted by slave
TXE
Transmission Error (Underflow)
1=no new data in TX FIFO,
SPI enters idle mode if master device

Figure 11-6. SPISTAT Register

Table 11-3. SPI Status Register Bit Descriptions

Bit(s)	Name	Revisions prior to 1.2 Definition	Revision 1.2 or greater Definition
0	SPIF	**SPI Transmit or Receive Transfer Complete.** This bit is set (=1) when an SPI transfer is complete. 1) This bit is updated only during interrupt or DMA driven SPI transfers. For example: SPRINT =1 or SPTINT=1 or TDMAEN=1 or RDMAEN =1 2) The bit is set when the transmit buffer status is empty **or** the receive buffer status is full. 3) This bit does not reflect the status of the transmit or receive shift register. 4) This bit is a sticky bit that can be reset only by disabling the SPI (SPIEN =0).	**SPI Transmit Transfer Complete.** This bit is set (=1) when an SPI transfer is complete. 1) This bit is updated during all SPI data transfers. 2) The bit is set when the transmit data buffer is empty and the data has been transmitted out of the transmit shift register. 3) This bit is not sticky.
1	MME	**Multimaster Error.** This bit is set when a device that is not currently the master device tries to become the master by driving a $\overline{\text{SPIDS}}$ signal while the current master device is communicating to SPI slave devices.	**Multimaster Error.** Same

Table 11-3. SPI Status Register Bit Descriptions (Cont'd)

Bit(s)	Name	Revisions prior to 1.2 Definition	Revision 1.2 or greater Definition
2	TXE	**Transmission Error.** This bit is set (=1) if there is a transmission error. 1) This bit is updated only during interrupt or DMA driven SPI transfers. For example: SPTINT=1 or TDMAEN=1 2) This bit is set (=1) when the transmit data buffer status is empty. 3) This bit does not reflect the status of the transmit shift register. This bit is set when the transmit buffer is empty and the data in the shift register is being transmitted. 4) This bit is a sticky bit. It can be reset only by disabling the SPI (SPIEN =0).	**Transmission Error.** This bit is set (=1) if there is a transmission error. 1) This bit is updated whenever there is a transmit error during all SPI data transfers. 2) This bit is set when the transmit buffer status is empty and the last data has been transmitted out of the transmit shift register. 3) This bit is a sticky bit. It can be reset only by disabling the SPI (SPIEN =0).
4-3	TXS	**Transmit Data Buffer Status.** These bits indicate the status (read only) of the SPITX data buffer. These bits are updated whenever an access (write by core/DMA or read by shift register) is made to the transmit data buffer. The status is as follows: 00 = empty 01 = partially full 11 = full	**Transmit Data Buffer Status.** Same

Table 11-3. SPI Status Register Bit Descriptions (Cont'd)

Bit(s)	Name	Revisions prior to 1.2 Definition	Revision 1.2 or greater Definition
5	RBSY	**Reception Error.** This bit is set (=1) if there is a reception error. 1) This bit is updated only during interrupt or DMA driven SPI transfers. For example SPRINT =1 or RDMAEN =1 2) This bit is set if the receive buffer status is full. 3) This bit does not reflect the status of the receive shift register. This bit will show high when the receive buffer is full, and the next data is being received by the shift register. 4) This bit is a sticky bit. It can be reset only by disabling the SPI (SPIEN =0).	**Reception Error.** This bit is set (=1) if there is a reception error. 1) This bit is updated whenever there is a receive error during all SPI transfers. 3) This bit is set if the receive buffer status is full and the last data has been received completely in the receive shift register. 2) This bit is a sticky bit. It can be reset only by disabling the SPI (SPIEN =0).
7-6	RXS	**Receive Data Buffer Status.** These bits indicate the status of the SPIRX data buffer (read only) as follows: 00 = empty 01 = partially full 11 = full 1) These bits are updated whenever an access (read by the core/DMA or write by shift register) is made to the receive data buffer.	**Receive Data Buffer Status.** Same
31-8		Reserved	Reserved

SPI Transmit Data Buffer (SPITX)

The SPITX transmit data buffer is a 32-bit data buffer which is part of the IOP register set. The buffer is 2-deep. The SPITX data buffer can be accessed by the core or the DMA controller. Data is loaded into SPITX before being transmitted. Once the SPI is enabled, data in SPITX is automatically loaded into the transmit shift register.

Consecutive writes to the SPITX may cause incorrect buffer status.

A write to SPITX instruction and a read from SPISTAT instruction must be separated by at least one instruction for TXS to be reflected properly in SPISTAT.

For interrupt based data transfers, the write to SPITX instruction and the RTI instruction of the ISR should be separated by at least one instruction.

SPI Receive Data Buffer (SPIRX)

The SPIRX receive data buffer is a 32-bit read-only data buffer accessible by the core or DMA controller. The buffer is 2-deep. After a word is received completely in the shift register RXSR, it is automatically transferred to the SPIRX.

Do not perform a normal core write of SPITX during DMA operation. A normal core read of SPITX can be done at any time and does not interfere with, or initiate, SPI transfers.

Do not perform a normal core read of SPIRX during DMA operation. A normal core write of SPIRX can be done at any time and does not interfere with, or initiate, SPI transfers.

A core hang results from writing to a full SPITX buffer or from reading from an empty SPITX buffer.

SPI Shift Registers

The SPI interface has two shift registers: one that serially transmits data (TXSR) and the other that receives data (RXSR) synchronously with the SPI clock signal (SPICLK). These registers are not directly accessible by the core or DMA controller. The registers shift right or left depending on the direction of the data flow (LSB first or MSB first) as defined by the DF bit in the SPICTL register. These shift registers include 32 shift cells that can be configured to transfer 8-, 16-, and 32-bit words.

SPI Data Word Formats

The ADSP-21161 processor SPI supports two transfer formats with respect to clock phases and clock polarities: CPHASE = 0 and CPHASE =1. The user application code can select one of the four combinations of serial clock phase and polarity using the CP and CPHASE bits in the SPICTL register. This section describes the transfer format and word packing for SPI transfers. See Table 11-2 on page 11-12 for SPICTL register bit description.

A master SPI transfer starts when the MS bit and the SPIEN bit are set (= 1) in the SPICTL register. If the CPHASE bit in the SPICTL register is cleared (=0), the SPICLK signal remains inactive for the first half of the first cycle of SPICLK. A slave SPI transfer starts as soon as the $\overline{\text{SPIDS}}$ signal from the master goes low without waiting for the SPICLK edge.

When CPHASE is set (=1), the transfer starts with the first edge of SPICLK going from its inactive state to the active state for both slave and the master devices. A transfer for a slave device is ended with $\overline{\text{SPIDS}}$ negated (inactive on rising edge). For a master device, transfer is considered complete after it transmits the last data word or receives the last data word.

Figure 11-7 and Figure 11-8 demonstrates the two basic transfer formats as defined by the CPHASE bit. Two waveforms are shown for SPICLK: one for CP = 0 and the other for CP=1. The diagram may be interpreted as a master or slave timing diagrams since the SPICLK, MISO, and MOSI pins are directly connected between the master and the slave. The MISO signal is the output from the slave (slave transmission), and the MOSI signal is the output from the master (master transmission). The SPICLK signal is generated by the master, and the $\overline{\text{SPIDS}}$ signal is the slave device select input to the slave from the master. The diagram represents an 8-bit transfer (WL=00) with MSB first (DF=1). Any combination of the WL and DF bits of the SPICTL register is allowed. For example, a 32-bit transfer with LSB first is also a possible configuration.

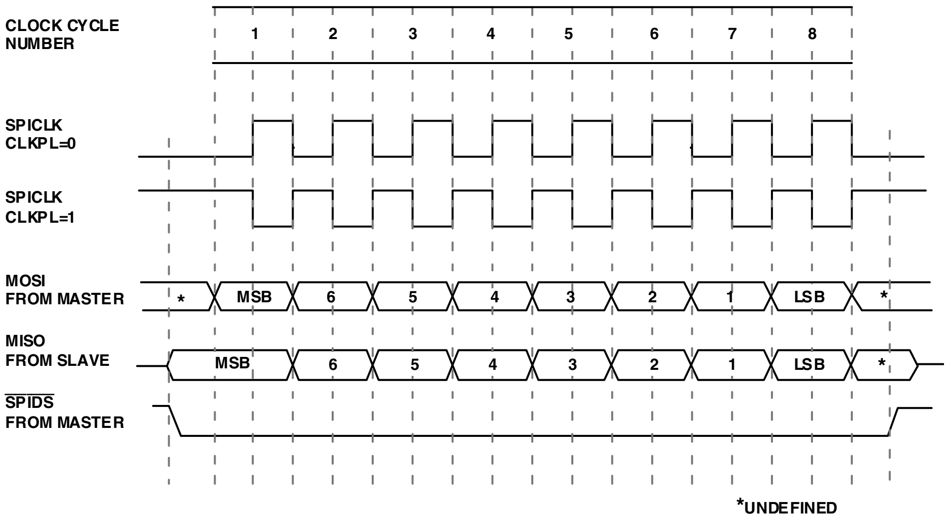


Figure 11-7. SPI Transfer Protocol for CPHASE = 0

The clock polarity and the clock phase must be identical for the master device and the slave device involved in the communication link. The transfer format from the master may be changed between transfers to adjust to various requirements of a slave device.

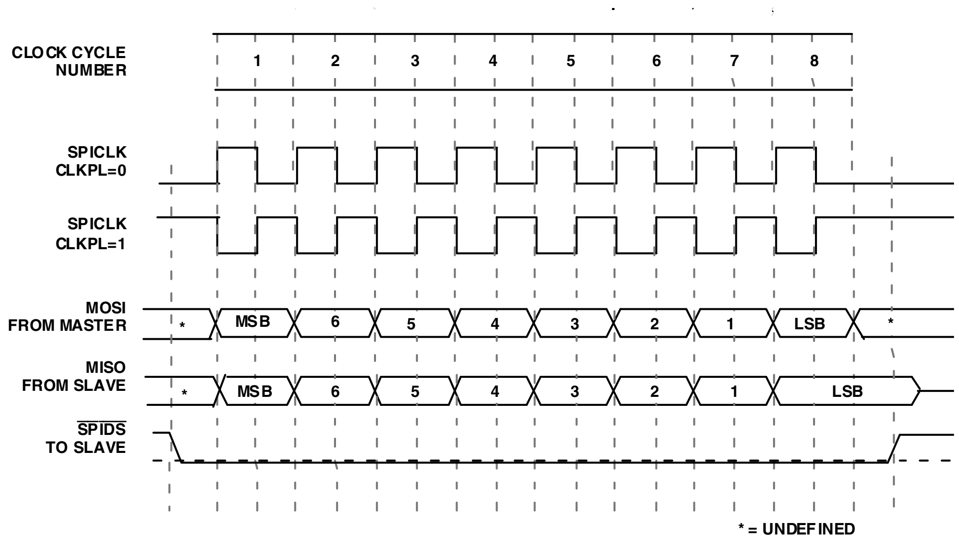


Figure 11-8. SPI Transfer Protocol for CPHASE = 1

Enable `DCPHO` (bit 20) (=1) to make the slave select line, $\overline{\texttt{SPIDS}}$, inactive (HIGH) between each serial transfer. This is controlled automatically by hardware logic. This feature is available in both `CPHASE`=0 and `CPHASE`=1. The standard SPI peripherals use this mode only in CPHASE=0. Clearing the `DCPHO` bit (=0) keeps $\overline{\texttt{SPIDS}}$ active low throughout the entire data transfer for both `CPHASE`=0 and `CPHASE`=1.

Table 11-4. SPICLK Driving and Latching Edges for SPI Data Transfers

Phase	Polarity	Driving Edge of SPICLK	Latching Edge of SPICLK
0	0	Falling Edge	Rising Edge
0	1	Rising Edge	Falling Edge
1	0	Rising Edge	Falling Edge
1	1	Falling Edge	Rising Edge

SPI Word Packing

The SPI packs two 8-bit words in a 32-bit word as shown below. This packing is set by the PACKEN bit in the SPICTL register. These words may be sign extended depending on the SGN bit in the SPICTL register as shown below.

31			0
S S S S S S S S [1]	Second 8-bit word	S S S S S S S S[1]	First 8-bit word

1 if SGN=1, S is the sign bit.
if SGN=0, S is 0.

SPI Operation Modes

The SPI in the ADSP-21161 processor can be used in a single master or multi-master mode. The MOSI, MISO, and the SPICLK signals of all SPI devices are connected together in both modes. The following sections describe the SPI operation as a master and as a slave. Figure 11-2 on page 11-4 shows the master-slave interconnections.

The ADSP-21161 processor is intended to be full-duplex. The SPICLK will stall on the transmit buffer becoming empty OR on the receive buffer becoming full. Dummy reads are necessary from the receive buffer even if programs are not interested in the transmitting data.

Master Mode Operation

When SPI is in master mode the interface operates in the following manner:

1. Configure the SPICTL register enabling the device as a master, specifying the appropriate word length, baud rate and any other options needed.

2. The SPI interface sends $\overline{\text{SPIDS}}$ signal to the desired slaves using one or more of the FLAG0-3 pins. For more information, see "Automatic Slave Selection" on page 11-26.

3. Write a data word to the SPITX data buffer using the core. This starts the SPICLK generation.

4. SPI generates the programmed clock pulses SPICLK and simultaneously shifts data from the transmit shift register (TXSR) out of the MOSI pin and into the receive shift register (RXSR) via the MISO pin. Before starting the shift, TXSR is loaded with the contents of the transmit data buffer register SPITX. At the end of the transfer, the receive data buffer register SPIRX is loaded with the contents of RXSR.

5. For interrupt driven core transfers to or from SPITX or SPIRX, enable bits SPTINT and SPRINT in the SPICTL register. An SPI interrupt occurs when SPITX is partially empty or when SPIRX is partially full. The interrupt service routine then transfers data to or from SPI data buffers.

6. For duplex DMA transfers, enable bits RDMAEN and TDAMEN in SPICTL. DMA requests are generated when SPITX is partially empty or when SPIRX is partially full. The DMA controller then transfers data between internal memory and the SPI data buffers.

Interrupt and DMA Driven Transfers

For interrupt driven transfers, the SPTINT or SPRINT bit should be set in the SPICTL. The interrupt routine in the user software is expected to perform the data transfer. For DMA driven transfers, the RDMAEN or TDMAEN bits must be set in the SPICTL. The DMA controller does the data transfer automatically. An interrupt is generated at the end of the DMA transfer. For more information on interrupts, see section "SPI Interrupts" on page 11-8.

Interrupts or DMA requests are automatically generated when the transmit buffer is partially empty or when the receive buffer is partially full. In the event that the SPITX and SPIRX interrupts are not serviced, or a higher priority DMA occurs, resulting in the transmit buffer becoming empty or the receive buffer becoming full, the SPI device will stall the SPI clock until all the data is read from the receive buffer or a piece of data is written to the transmit buffer.

Core Driven Transfers

For core driven SPI transfers, SPTINT and SPRINT are enabled in the SPI, and the corresponding interrupt masks SPIRMSK and SPITMSK are disabled in the LIRPTL register. The user software has to read from or write to SPIRX and SPITX in the transmit buffer becoming empty or the receive buffer becoming full, the SPI device will stall the SPI clock until all the data is read from the receive buffer or a piece of data is written to the transmit buffer.

Automatic Slave Selection

Multiple slaves are automatically controlled (selected and deselected) during the SPI transfer by enabling the PSSE bit in the SPICTL register. This bit locks all the four flag pins (FLAG0,FLAG1,FLAG2 and FLAG3) as SPI slave selects. By writing to the FLS bits (bits 15-18) in the SPICTL register, the corresponding FLAG bits are programmed as outputs for slave selection.

To enable the different slaves, connect the slave $\overline{\text{SPIDS}}$ pins to the programmable flag pins FLAG0-3 of the master ADSP-21161. Since these flags are NOT open drain, slave select pins (FLAGS) cannot be shorted together in multimaster environment. To control slave selects, an external glue logic is required in a multi-master environment. Enable the SPI port by setting the SPIEN bit in the SPICTL. The master's flag pins are asserted low and the $\overline{\text{SPIDS}}$ signals of the slaves are asserted. Upon completion of the transfer, the FLAG pins are de-asserted, and slave selection is subsequently disabled.

During data transfers, if the SPI clock is stalled, the slaves are automatically deselected by de-asserting the flags in the master. Once data transmission becomes possible, the slaves are automatically selected again by asserting the flags in the master.

When DCPH0 is set, the slaves are automatically deselected and selected again by de-asserting and asserting the flags in the master. This is done automatically in the SPI.

There is a one cycle latency for a flag output to change after writing to the SPICTL register (when PSSE is set and the flag is enabled). To use the PSSE feature, systems can have five SPI devices with ADSP-21161 as the master. The PSSE is programmed for slave selection of the other four devices. The ADSP-21161 processor can broadcast to all the four slaves at once or can write to individual slaves by appropriately programming the FLS bits.

User Controlled Slave Selection

The user can also control the slaves without enabling the PSSE bit in the SPI. The user can set or clear the I/O flags directly by writing a 1 or 0 into the FLAG register. The user can also emulate DCPH0 operation by setting or clearing the values in the FLAG register at the appropriate time.

When using this mode, the following sequence should be followed to ensure proper data transfer according to the SPI protocol.

1. Enable the SPI by writing into the `SPICTL` register.

2. Assert the required slave select by writing a zero into the appropriate bit in the `FLAG` register.

3. Load `SPITX` with the required data by enabling DMA's, interrupts, or by performing core writes to `SPITX`.

Slave Mode Operation

To prepare for the data transfer, a slave processor writes the data to be transmitted into the transmit data buffer. The following steps illustrate SPI operation in slave mode.

1. Configure the `SPICTL` register enabling the device as a slave and specifying the appropriate word length and any other options needed to be compatible with the master device.

2. Once the core receives the $\overline{\texttt{SPIDS}}$ signal from the master, it starts sending or receiving data on the proper `SPICLK` edge.

3. Reception/transmission continues until $\overline{\texttt{SPIDS}}$ is negated.

4. SPI receives the programmed clock pulses `SPICLK` and shifts data out of `MISO` and in from `MOSI`. Before starting the shift, the transmit shift register (`TXSR`) is loaded with the contents of the transmit data buffer register `SPITX`. At the end of the transfer, the receive data buffer register `SPIRX` is loaded with the contents of the receive shift register (`RXSR`).

5. For interrupt driven core transfers to or from SPITX or SPIRX, enable bits SPTINT and SPRINT in the SPICTL register. An SPI interrupt occurs when SPITX is partially empty or when the receive buffer SPIRX is partially full.

6. For duplex DMA transfers, enable RDMAEN and TDAMEN in SPICTL need. DMA requests are generated when SPITX is partially empty or when SPIRX is partially full. The DMA controller then transfers data between internal memory and the SPI data buffers.

Interrupts and DMA requests are automatically generated when the transmit buffer is partially empty or when the receive buffer is partially full. In case of DMA driven or core driven transfers, if the transmit buffer becomes empty or the receive buffer becomes full, the SPI device continues to operate based on the conditions of the SENDLW and GM bits.

If the SENDLW bit is cleared (=0) and the transmit buffer is empty, the device repeatedly transmits 0s out on the MISO pin. If the SENDLW is set (=1) and the transmit buffer is empty, the device continues to transmit the last word written to SPITX that was transmitted. Retransmission of the data in SPITX occurs after the transmit buffer becomes empty.

If the GM bit is set (=1) and the receive buffer is full, the device continues to receive new data from the MOSI pin, overwriting the previous (older) data in the SPIRX buffer. If the GM bit is reset (=0) and the receive buffer is full, the incoming data from the shift register is discarded and the SPIRX register is not updated. The register ignores the new data and retains the old information.

Error Signals and Flags

Please refer to the SPISTAT register definitions in Table 11-3 on page 11-17 for the following discussion. Note that the functionality of some bits differ in revisions prior to 1.2. These difference are described in Table 11-3.

Multi-Master Error (MME)

The MME bit is set (=1) in the SPISTAT register when the $\overline{\text{SPIDS}}$ pin of a master ADSP-21161 processor is driven low by another device in the system. This occurs when another device is trying to be the master. This can cause contention between two drivers and push-pull CMOS drivers. When this error is detected, the following actions take place:

1. The MS control bit in SPICTL is cleared (= 0), configuring the SPI interface as a slave.
2. The SPIEN bit in SPICTL is also cleared, disabling the SPI system.
3. The MME status bit in SPISTAT is set. This bit can be polled to test whether this error condition has occurred.

Transmission Error (TXE)

For revisions 1.1 and earlier, this error bit is updated only when SPTINT or TDAMEN is enabled. This bit is set in the SPISTAT register when SPI is enabled and there is no data in the transmit data buffer (SPITX). This is true in both master and slave SPI devices.

When the device is an SPI master, upon setting the error bit, the data in the transmit shift register (TXSR) is transmitted out. Then, the SPICLK is stalled automatically until new data is written into the SPITX data buffer.

When the device is an SPI slave, upon setting the error bit, the data is still transmitted as specified by the SENDLW bit in the SPICTL register.

The TXE bit is cleared (=0) only when SPIEN is disabled.

For revisions 1.2 and later, this error bit is updated during any SPI transfer. This bit is set in the SPISTAT register when SPI is enabled and there is no data in SPITX and in TXSR. This is true in both master and slave SPI devices.

When the device is an SPI master, the working of the bit depends on the mode of data transfer. For DMA or interrupt driven data transfer, the SPICLK will stall as soon as both the SPITX and the TXSR become empty. There is NO transmission error in this case. For core driven data transfers, the error bit is set as soon as both the SPITX and the TXSR become empty. The SPI continues to transmit the next data as specified by the SENDLW bit in the SPICTL register.

Reception Error (RBSY)

For revisions 1.1 and earlier, this error bit is updated only when SPRINT or RDAMEN is enabled. This bit is set in the SPISTAT register when SPI is enabled and there is no space in the receive data buffer (SPIRX). This is true in both master and slave SPI devices.

When the device is an SPI master, upon setting the error bit, one more data is fully received in the receive shift register (RXSR). Then, the SPICLK is stalled automatically until a data has been read out of the SPIRX data buffer.

When the device is an SPI slave, upon setting the error bit, the data is still received as specified by the GM bit in the SPICTL.

The RBSY bit is cleared (=0) only when SPIEN is disabled.

For revisions 1.2 and later, this error bit is updated during all SPI data transfers. This bit is set in the SPISTAT register when SPI is enabled and there is no space in the SPIRX and in RXSR. This is true in both master and slave SPI devices.

When the device is an SPI master, the working of the bit depends on upon the type of data transfer. For DMA or interrupt driven data transfer, the SPICLK is stalled as soon as the SPIRX becomes full. There is NO reception error in this case. For core driven data transfers, this error bit is set as soon as both the SPIRX and the RXSR become completely full. The next data is still received as specified by the GM bit in the SPICTL.

SPI/Link Port DMA

The SPI shares DMA channels 8 and 9 with the link port. The receive DMA is shared with link port 0 DMA, and the transmit DMA is shared with link port 1 DMA.

Do not enable SPI and link port DMA simultaneously. SPI and link port are mutually exclusive when one of the peripherals is enabled.

SPI DMAs have higher priority than link port DMAs. If SPI DMAs must be enabled, disable link port DMAs and pending link port DMA requests. For more information, see "SPI Port DMA" on page 6-108.

DMA Operation in SPI Master Mode

For transmit DMA operations, if the DMA controller is unable to keep up with the transmit stream, due perhaps to heavy DMA channel activity, the data in the transmit shift register (TXSR) is transmitted out. Then the SPI-CLK is stalled automatically until a new data is written into the SPITX data buffer.

For receive DMA operations, if the DMA controller is unable to keep up with the receive data stream, the receive buffer operates according to the state of the GM bit. If GM is set (=1) and the receive buffer is full, one more data is fully received in the receive shift register (RXSR). Then, the SPICLK is stalled automatically until data has been read out of the SPIRX data buffer.

Do not perform a normal core write of SPITX during DMA operation. A normal core read of SPITX can be done at any time and does not interfere with, or initiate, SPI transfers.

Do not perform a normal core read of SPIRX during DMA operation. A normal core write of SPIRX can be done at any time and does not interfere with, or initiate, SPI transfers.

Interrupts are generated based on DMA events that are configured in the SPICTL register.

DMA Operation in Slave Mode

When the DMA controller transmits or receives data in slave mode, the start of a transfer is triggered by a transition of the $\overline{\text{SPIDS}}$ signal to the active-low state or by the first active edge of SPICLK.

For transmit DMA operations, if the DMA engine is unable to keep up with the transmit stream, the transmit port operates according to the state of the SENDLW bit. Once the transmit buffer is empty and the last word has been transmitted completely out of the TXSR, if SENDLW is cleared (=0), the device repeatedly transmits 0s on the MISO pin. If SENDLW is set (=1), it repeatedly transmits the last word transmitted before the transmit buffer became empty.

For receive DMA operations, if the DMA controller is unable to keep up with the receive data stream, the receive buffer operates according to the state of the GM bit. If GM is set (=1) and the receive buffer is full, one more data is fully received in the receive shift register (RXSR). The SPICLK is stalled automatically until a data has been read out of the SPIRX data buffer.

Do not perform a normal core write of SPITX during DMA operation. A normal core read of SPITX can be done at any time and does not interfere with, or initiate, SPI transfers.

Do not perform a normal core read of SPIRX during DMA operation. A normal core write of SPIRX can be done at any time and does not interfere with, or initiate, SPI transfers.

SPI Booting

The ADSP-21161 processor allows a host SPI device to boot the processor on power-up $\overline{\text{RESET}}$ de-assertion. To enable the SPI booting mode, the EBOOT and $\overline{\text{BMS}}$ pins must be tied low, and the LBOOT pin must be tied high. When the processor comes out of reset, it starts the SPI boot process. The SPI is configured as a slave upon power-up. Therefore, after reset, the SPI waits for $\overline{\text{SPIDS}}$ and SPICLK from the SPI host to download the boot program.

The default value of the SPICTL register when the processor is configured for SPI boot is 0x0A00 1F81. The SPI port is enabled as a slave to receive 32-bit words in LSB-first format. DMA is enabled to facilitate loading the boot kernel. The DMISO bit is also enabled to avoid contention in the MISO pin in systems where multiple slave devices are to be booted simultaneously.

DMA channel 8 is used when downloading the boot kernel information to the processor. At reset, the DMA parameter registers for DMA channel 8 are initialized to their required values. Table 11-5 lists the initial values for these registers.

The ADSP-21161 SPI booting mode supports boots from 8-, 16-, or 32-bit host SPI devices. In SPI boot mode, the data word size in the shift register defaults to 32 bits. Therefore, for 8- or 16-bit hosts, data words are packed into the shift register to generate 32-bit words, which can be shifted into internal program memory.

Table 11-5. Parameter Initialization Value

Parameter Register	Initialization Value
IISRX	0x0004 0000
IMSRX	0x0000 0001
CSRX	0x0000 0180
GPSRX	uninitialized

The host initiates the booting operation by activating SPICLK and asserting the $\overline{\text{SPIDS}}$ signal to the active low state. The 256-word, boot-strapped instruction loader kernel is loaded 32 bits at a time, via the 32-bit SPI receive shift register (RXSR). To properly upload 256 instructions (48-bit words), the SPI DMA initially loads a DMA count of 0x180 (384) 32-bit words which is equivalent to 0x100 (256) 48-bit words. The relationship between the 32-bit words received into the SPIRX register and the instructions that need to be placed in internal memory is described in the Figure 11-9.

After the first 256 words are loaded the interrupt associated with the SPI receive is activated. The processor jumps to the location for SPIRI_svc (0x40040) and executes the code located there. Typically, the first instruction at the SPI receive interrupt vector (SPIRI) is an RTI instruction in which case the processor jumps to location 0x40005 where normal program execution continues. Because most applications require more than 256 words of instructions and initialization data, a loader and a 256 word loader kernel are supplied with the tools. Use these tools to create code that automatically loads the rest of the application code and then overwrites itself with application code and data. For more information on the loader, see the development tools documentation.

The boot loader kernel supplied with the tools loads a combination of instructions with DMA into scratch locations and then writes the instructions to internal memory using the core via the PX register. The 256-word, boot-strapped instruction loader kernel is loaded 32-bits at a time, via the

32-bit SPI receive shift register, using a normal-word addressing scheme with two-column memory addresses. Figure 11-9 shows how SPI data is packed in internal memory.

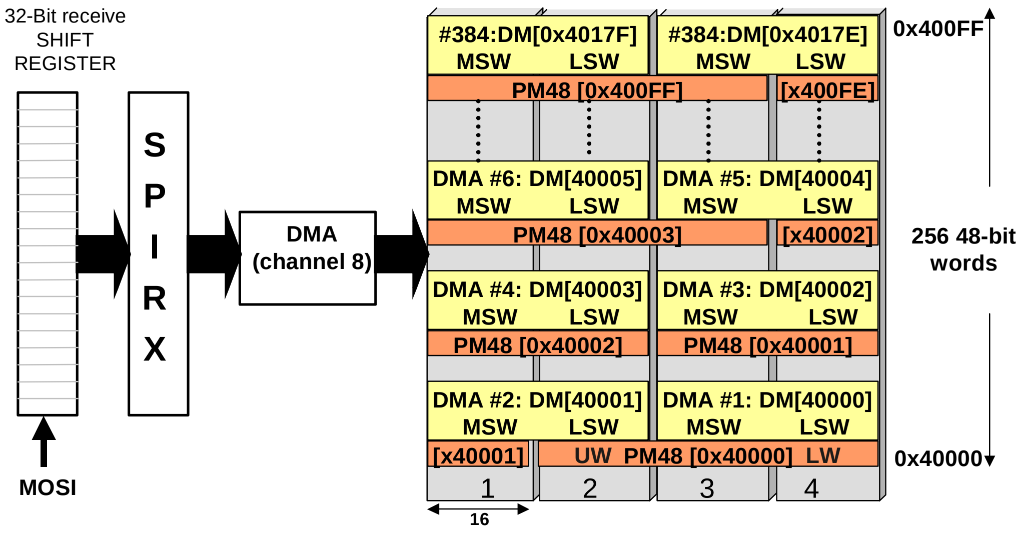


Figure 11-9. SPI Data Packing

The SPI Control Register (SPICTL) is configured to 0x0A00 1F81 upon reset during SPI boot. SPI transfers occur with the following default bit settings:

- SPIEN = 1, SPI enabled
- MS = 0, slave device
- DF = 0, LSB first
- WL = 11, 32-bit SPI receive shift register word length
- DMISO = 1, MISO disabled
- RDMAEN = 1, Receive DMA enabled

The SPIRX DMA channel 8 parameter registers are configured to DMA in 0x180 32-bit words into internal memory normal-word address space starting at 0x40000. Once the 32-bit DMA completes, the data is then accessed as 3-column 48-bit instruction accesses, for example, the processor executes a 256 (0x100) word loader kernel upon completion of the 32-bit, 0x180 word DMA.

For 16-bit SPI hosts, two words are shifted into the 32-bit receive shift register (RXSR) before a DMA transfer to internal memory occurs. For 8-bit SPI hosts, four words are shifted into the 32-bit receive shift register before a DMA transfer to internal memory occurs.

By default, the booting SPI expects to receive words into SPIRX seemlessly. This means that bits are received continuously without breaks. For different SPI host sizes, the processor expects to receive instructions and data packed in an LSW format.

Figure 11-10 shows a pair of instructions packed for SPI booting using a 32-, 16-, and an 8-bit host.

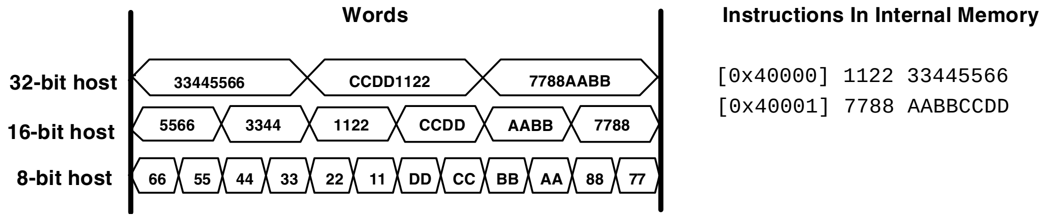


Figure 11-10. Instruction Packing for 32-, 16-, or 8-Bit SPI Host Booting

The following sections examine how data is packed into internal memory during SPI booting for SPI host word widths of 32-, 16-, or 8-bits.

32-Bit SPI Host Boot

Figure 11-11 shows 32-bit SPI host packing of 48-bit instructions. The 32-bit word is shifted to internal program memory during loading of the 256-word kernel.

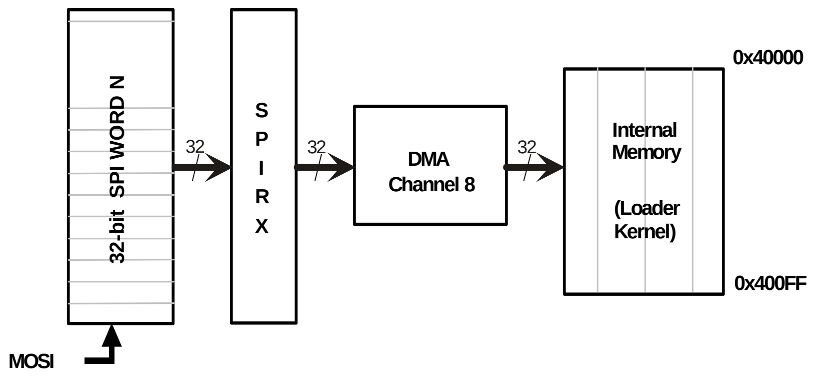


Figure 11-11. 32-Bit SPI Host Packing

The following is an example of 48-bit instructions to be executed at PM addresses 0x40000 and 0x40001:

> [0x40000] 1122 33445566
> [0x40001] 7788 AABBCCDD

The 32-bit SPI host would need to pack (prearrange data) as follows:

> SPI word 1 = 0x33445566
> SPI word 2 = 0xCCDD1122
> SPI word 3 = 0x7788AABB

16-Bit SPI Host Boot

Figure 11-12 shows 16-bit SPI host packing of 48-bit instructions. For 16-bit hosts, two 16-bit words are packed into the shift register to generate a 32-bit word. The 32-bit word is shifted to internal program memory during loading of the kernel.

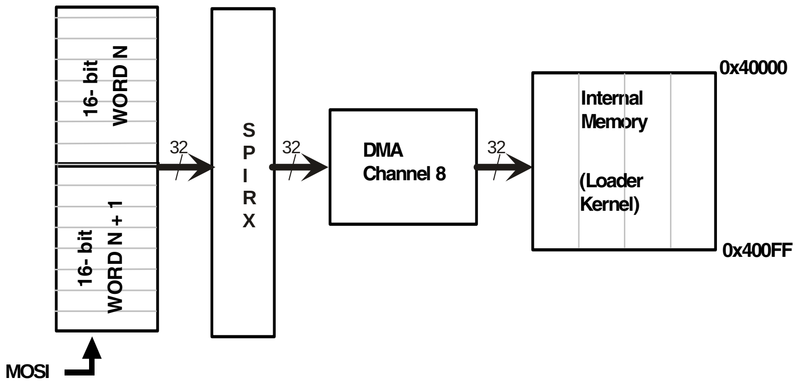


Figure 11-12. 16-Bit SPI Host Packing

The following is an example of 48-bit instructions to be executed at PM addresses 0x40000 and 0x40001:

```
[0x40000] 1122 33445566
[0x40001] 7788 AABBCCDD
```

The 16-bit SPI host would need to pack (prearrange data) as follows:

```
SPI word 1 = 0x5566
SPI word 2 = 0x3344
SPI word 3 = 0x1122
SPI word 4 = 0xCCDD
SPI word 5 = 0xAABB
SPI word 6 = 0x7788
```

The initial boot of the 256-word loader kernel requires a 16-bit host to transmit 768 16-bit words. One 32-bit word is created from two packed 16-bit words. The SPI DMA count value of 0x180 is equivalent to 384 words. Therefore, the total number of 16-bit words loaded is 768.

8-Bit SPI Host Boot

Figure 11-13 shows 8-bit SPI host packing. For 8-bit hosts, four 8-bit words are packed into the shift register to generate a 32-bit word. The 32-bit word is then shifted to internal program memory during loading of the 256-instruction word kernel.

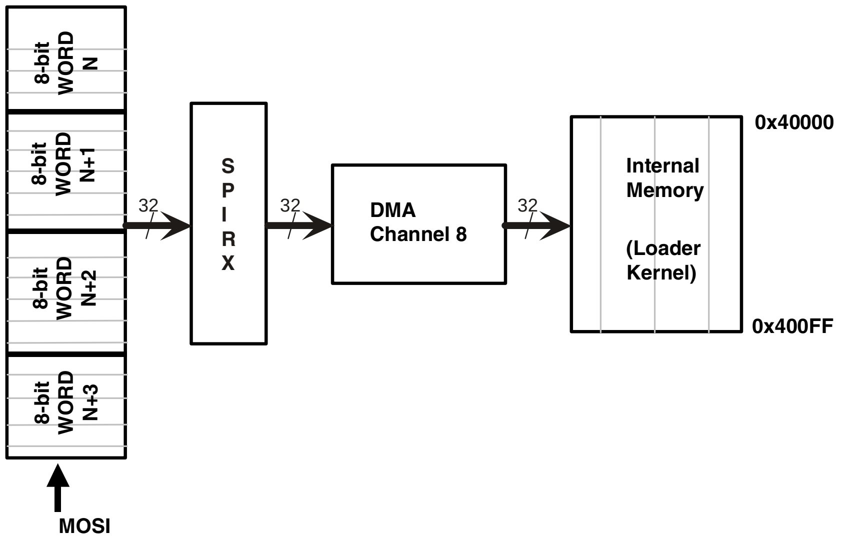


Figure 11-13. 8-Bit SPI Host Packing

The following is an example of 48-bit instructions to be executed at PM addresses 0x40000 and 0x40001:

```
[0x40000] 1122 33445566
[0x40001] 7788 AABBCCDD
```

The 8-bit SPI host would need to pack (prearrange data) as follows:

SPI word 1 = 0x66
SPI word 2 = 0x55
SPI word 3 = 0x44
SPI word 4 = 0x33
SPI word 5 = 0x22
SPI word 6 = 0x11
SPI word 7= 0xDD
SPI word 8 = 0xCC
SPI word 9 = 0xBB
SPI word 10 = 0xAA
SPI word 11 = 0x88
SPI word 12 = 0x77

The initial boot of the 256-word loader kernel requires an 8-bit host to transmit 1536 8-bit words. The SPI DMA count value of 0x180 is equal to 384 words. Since one 32-bit word is created from four packed 8-bit words, the total number of 8-bit words transmitted is 1536.

Multiprocessor SPI Port Booting

In systems where multiple ADSP-21161 processors are connected and configured for SPI booting, the master ADSP-21161 (or any SPI master device) can boot up to four processors configured as SPI slaves. The processor uses four programmable flags, `FLAG0-3`, as dedicated SPI device-select signals for the SPI slave devices. The `FLS` bits in the `SPICTL` register correspond to these flags.

- Figure 11-14 shows a single ADSP-21161 processor master with four slaves. The master processor selects each slave device using a dedicated `FLAG` pin. The master device communicates with one slave device at any given time, or it broadcasts data to multiple slaves by setting more than one `FLS` bit in `SPICTL`.

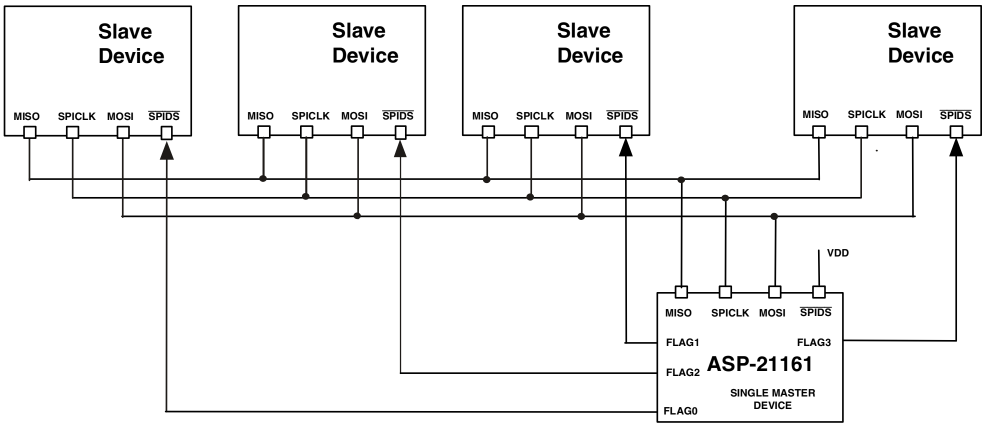


Figure 11-14. Single Master, Multiple Slaves Configuration – All ADSP-21161 Processors

The master ADSP-21161 processor can boot multiple slaves in the following ways:

- The ADSP-21161 processor transmits to all four SPI devices at the same time in a broadcast mode. Broadcast the 256-word loader kernel and identical application code simultaneously to all slaves. If the master is a ADSP-21161 processor, enable the `FLSx` bit in the `SPICTL` register, and disable the `MISO` pins. Otherwise, the master asserts the $\overline{\texttt{SPIDS}}$ pins of all the slaves to transmit the data.

  This feature can be enabled by setting the `DMISO` bit in the four slave processors. This `DMISO` feature may be available in some microcontrollers. Therefore, it is possible to use the `DMISO` feature with any SPI devices that include this functionality.

- Load the bootstrap kernel and processor instructions and data one-at-a time for each processor. In this case, enable only one FLSx bit at a time in the SPICTL register to drive the flag pin connected to a slave's device select. The master device will assert the $\overline{\text{SPIDS}}$ pin of the slave to load the data. This ensures that each processor boots one after the other.

- It is also possible to use a combination of broadcast and individual processor booting to boot a multiprocessor system. SPI hosts can broadcast boot application code that will reside on several slaves and then complete the booting process by booting the individual slaves with slave specific application code. In this situation, the host SPI device asserts the $\overline{\text{SPIDS}}$ pins of all slaves during the broadcast portion of the boot. The host then asserts the $\overline{\text{SPIDS}}$ pins of specific slaves. If the ADSP-21161 processor is the master as is shown in Figure 11-14, the master enables the FLSx bit in the SPICTL register for the slave currently booting.

Figure 11-14 shows one ADSP-21161 processor as a master and four ADSP-21161 processor (or other SPI-compatible devices) as slaves:

SPI Programming Example

This section provides two programming examples written for the ADSP-21161 processor. The core-driven interrupt SPI loopback example in Listing 11-1 demonstrates how the core reads from the SPI receive buffer and writes to the SPI transmit buffer after receiving an interrupt. The core-driven interrupt SPI loopback without interrupts example in Listing 11-2 demonstrates how the core reads from the SPI receive buffer and writes to the SPI transmit buffer after polling the buffer status. For an SPI DMA programming example, see Listing 6-7 on page 6-116.

Listing 11-1. Core-Driven Interrupt SPI Loopback

```
/*_____________________________________________________________
ADSP-21161 Core-Driven Interrupt SPI Loopback Example
This example shows looped-back SPI 32-bit transfer. On this
peripheral loop-back is preformed by externally connecting the
hardware MOSI and MISO pins on the processor. After receiving an
interrupt, the core directly writes to the transfer buffer
(SPITX) and reads from the receive buffer (SPIRX). Hardware
loop-back does not require the use of flags as device selects so
the FLS bits do not need to be used as they would in an SPI trans-
fer between two different SPI devices (non-loop-back.)
_____________________________________________________________*/
#include <def21161.h>
#define size 10

// reserved vector location
.section/pm seg_rsvd1;
Reserved_1:     rti; nop; nop; nop;

// vector code for reset vector from ldf file
.section/pm   seg_rth;
Chip_Reset:     idle; jump start; nop; nop;

// vector code for receive interrupt vector from ldf file
.section/pm spiri_svc;
nop; nop; jump receive; rti;

.section/dmseg_dmda;
.var spi_tx_buf[size] =0x11111111,0x22222222, 0x33333333,
0x44444444, 0x55555555,0x66666666, 0x77777777, 0x88888888,
0x99999999, 0xaaaaaaaa;
.var spi_rx_buf[size];
.section/pm seg_pmco;
```

```
start:

//Set pointers for source and dest, I0=B0 automatically
b0=spi_tx_buf;      // 32-bit SPI datawords
l0=@spi_tx_buf;
m0=1;
b1=spi_rx_buf;      // 32-bit SPI datawords
l1=@spi_rx_buf;
m1=1;

// set circular buffer enable and allow global interrupts
bit set MODE1 CBUFEN | IRPTEN;

bit set LIRPTL SPIRMSK ;  // enable SPI RX interrupts
bit set IMASK LPISUMI;    // unmask spi interrupts

r0=0x00000000;          // initially clear SPI control register
dm(SPICTL)=r0;

// prime SPITX register
r0=dm(i0,m0);
dm(SPITX)=r0;

ustat1=dm(SPICTL);    // set up options for the SPI port
bit set ustat1  SPIEN | SPRINT | SPTINT | MS | CPHASE | DF | WL32
| BAUDR5 | SGN | GM;

/* Enable spi port, spitx and spirx interrupts, master device
spiclk toggles at beginning of first data transfer bit, MSB first
format, 32 bit word length, baud rate sign extend and get more
new data even if receive buffer is full */

dm(SPICTL) = ustat1;   // start transfer by configuring SPICTL
```

```
wait: idle;jump wait;

receive: r0=dm(SPIRX);   //read SPIRX
dm(i1,m1)=r0;       //write value to internal memory buffer
r0=dm(i0,m0);       //get new value to transmit from internal
                    // transmit buffer
dm(SPITX)=r0;//write value to SPITX
rti;
```

Listing 11-2. Core-Driven Interrupt SPI Loopback Without Interrupts

```
/* ADSP-21161 Core-Driven Interrupt SPI Loopback without Inter-
ruptsThis examples shows a looped-back SPI 32-bit transfer.  On
this peripheral loop-back is performed by externally connecting
the hardware MOSI and MISO pins on the processor.  Hardware
loop-back doesnot require the use of flags as device selects so
the FLS bits do not need to be used as they would in an SPI trans-
fer between two different SPI devices (non-loop-back.)  In this
example, interruptsare not used to determine buffer status,
instead polling of the buffer status is implemented to allow the
code to know when it can  safely read from the SPIRX register.
Rev 1.1 1/22/02  */

#include <def21161.h>

#define size 10

// vector code for reset vector from ldf file
.section/pm   seg_rth;
Chip_Reset:     idle; jump start; nop; nop;

.SECTION/DM seg_dmda;
.var spi_tx_buf[size] =0x11111111,
0x22222222,
0x33333333,
0x44444444,
0x55555555,
0x66666666,
0x77777777,
0x88888888,
0x99999999,
0xaaaaaaaa;
.var spi_rx_buf[size];

.SECTION/PM seg_pmco;

start:

bit set MODE1 IRPTEN | CBUFEN;// set circular buffer enable and
allow global interrupts
b0=spi_tx_buf;// 32-bit SPI datawords
l0=@spi_tx_buf;
m0=1;

b1=spi_rx_buf;// 32-bit SPI datawords
l1=@spi_rx_buf;
m1=1;

r0=0x00000000;// initially clear SPI control register
dm(SPICTL)=r0;
ustat1=dm(SPICTL);

bit set ustat1  SPIEN | MS | DF | WL32 | BAUDR5 | SGN | GM;
/* The SPI transmit buffer must be fed with the first two data
words before enabling SPI if SPRINT/SPTINT will not be enabled
for interrupt usage */
r0=dm(i0,m0);
dm(SPITX)=r0;     //write to TX buffer
```

```
r0=dm(i0,m0);
dm(SPITX)=r0;      //write to TX buffer

dm(SPICTL) = ustat1;    //enable port

lcntr = 0x8, do looping until lce;
r0=dm(i0,m0);      //write to TX buffer
dm(SPITX)=r0;

// test receive buffer status to determine when it is ok to read
from SPIRX
test:ustat1=dm(SPISTAT);
bit tst USTAT1 RXS0;
if Not TF jump test;

r0=dm(SPIRX);      //read from RX buffer
dm(i1,m1)=r0;
looping: nop;

r0=dm(SPIRX);     //read from RX buffer
dm(i1,m1)=r0;

r0=dm(SPIRX);     //read from RX buffer
dm(i1,m1)=r0;

wait: jump wait;
idle;
```

12 JTAG TEST-EMULATION PORT

A boundary scan allows a system designer to test interconnections on a printed circuit board with minimal test-specific hardware. The scan is made possible by the ability to control and monitor each input and output pin on each chip through a set of serially scannable latches. Each input and output is connected to a latch, and the latches are connected as a long shift register so that data can be read from or written to them through a serial test access port (TAP). The ADSP-21161 processor contains a test access port compatible with the industry-standard IEEE 1149.1 (JTAG) specification. Only the IEEE 1149.1 features specific to the ADSP-21161 processor are described here. For more information, see the IEEE 1149.1 specification and other the documents listed in "References" on page 12-29.

The boundary scan allows a variety of functions to be performed on each input and output signal of the ADSP-21161 processor. Each input has a latch that monitors the value of the incoming signal and can also drive data into the chip in place of the incoming value. Similarly, each output has a latch that monitors the outgoing signal and can also drive the output in place of the outgoing value. For bidirectional pins, the combination of input and output functions is available.

Every latch associated with a pin is part of a single serial shift register path. Each latch is a master/slave type latch with the controlling clock provided externally. This clock (`TCK`) is asynchronous to the ADSP-21161 processor system clock (`CLKIN`).

The ADSP-21161 processor emulation features halt the processor at a pre-defined point to examine the state of the processor, execute arbitrary code, restore the original state, and continue execution.

The ADSP-21161 processor emulation features are a superset of the ADSP-21160 emulation features. All emulation features supported by previous SHARCs are supported on the ADSP-21161 processor, except the ICSA output signal and function. The set of features on which JTAG ICE designs rely are supported in an identical fashion on ADSP-21161 processor. The ADSP-21161 processor can be used with the ADSP-2106x SHARC JTAG ICE hardware.

There are several changes/extensions to the base functionality of the ADSP-2106x emulation capability, which require changes in the JTAG ICE software for ADSP-21161 processor support. These extensions include:

1. The emulation breakpoint address start/end registers have moved from UREG space to IOP register space. This change did not effect the TSTEMU block directly, only the address decodes to gain access to it.

2. EMU64PX has been added to the IR decode space. This shift register provides access to the full 64-bit wide PX register of ADSP-21161 processor.

3. A memory test shift register has been added to the IR decode space. This feature is for Analog Devices internal use ONLY.

Several on-chip facilities are directly accessed through the JTAG interface. These facilities are listed in Table 12-2 on page 12-4. Other emulation facilities are only indirectly accessible. To indirectly access the facilities that do not appear in Table 12-2 on page 12-4, scan the instruction which moves data of interest to/from the PX register, scan the PX data (if the instruction is a PX read), let the core execute the instruction, then scan the PX register out (if the instruction was a PX write).

The breakpoint start/end registers are mapped into the IOP register space of the ADSP-21161 processor. For specific addresses, see "Register and Bit #Defines (def21161.h)" on page A-121. The EMUN, EMUCLK, and EMUCLK2 registers occupy the same UREG address space as on the ADSP-2106x. These facilities are read-only by the ADSP-21161 processor core in normal operation.

JTAG Test Access Port

The emulator uses JTAG boundary scan logic for ADSP-21161 processor communications and control. This JTAG logic consists of a state machine, a five pin Test Access Port (TAP), and shift registers. The state machine and pins conform to the IEEE 1149.1 specification. The TAP pins appear in Table 12-1. A special pin ($\overline{\text{EMU}}$) is used by Analog Devices Inc. JTAG emulators. and is **not** defined in the IEEE-1149.1 specification. This signal notifies the JTAG ICE that the processor has completed an operation.

Table 12-1. JTAG Test Access Port (TAP) Pins

Pin	Function
TCK	(input) Test Clock: pin used to clock the TAP state machine.[1]
TMS	(input) Test Mode Select: pin used to control the TAP state machine sequence.[2]
TDI	(input) Test Data In: serial shift data input pin.
TDO	(output) Test Data Out: serial shift data output pin.
$\overline{\text{TRST}}$	(input) Test Logic Reset: resets the TAP state machine

1 Asynchronous with CLKIN
2 Synchronous to CLKIN

A Boundary Scan Description language (BSDL) file for the ADSP-21161 processor is available on Analog Devices' website. Set your browser to:

`http://www.analog.com/techsupt/documents/bsdl`

Refer to the IEEE 1149.1 JTAG specification for detailed information on the JTAG interface. The many sections of this appendix assume a working knowledge of the JTAG specification.

Instruction Register

The instruction register allows an instruction to be shifted into the processor. This instruction selects the test to be performed and/or the test data register to be accessed. The instruction register is 5 bits long with no parity bit. A value of 10000 binary is loaded (LSB nearest `TDO`) into the instruction register whenever the TAP reset state is entered.

Table 12-2 lists the binary code for each instruction. Bit 0 is nearest TDO and bit 4 is nearest TDI. No data registers are placed into test modes by any of the public instructions. The instructions affect the ADSP-21161 processor as defined in the 1149.1 specification. The optional instructions RUNBIST, IDCODE and USERCODE are not supported by the ADSP-21161 processor.

Table 12-2. JTAG Instruction Register Codes

43210	Register	Instruction	Comment	Type
11111	Bypass	BYPASS		Public
00000	Boundary	EXTEST		Public
10000	Boundary	SAMPLE		Public
01000	EMUPMD	EMULATION	48-bit scan length	Private
11000	Boundary	INTEST		Public
00100	EMUCTL	EMULATION		Private

Table 12-2. JTAG Instruction Register Codes (Cont'd)

43210	Register	Instruction	Comment	Type
10100	EMUPX	EMULATION	48-bit shift register	Private
10110	EMU64PX	EMULATION	64-bit shift register	Private
01100	EMUSTAT	EMULATION		Private
11100	BRKSTAT	EMULATION		Private
00010	EMUPC	EMULATION		Private
10101	MEMTST	TEST	Memory test	Private
All others	Reserved	Reserved		Private

The entry under "Register" is the serial scan path, either Boundary or Bypass in this case, enabled by the instruction. Figure 12-1 shows these register paths. The 1-bit Bypass register is fully defined in the 1149.1 specification. For more information on the Boundary register, see "Boundary Register" on page 12-17.

No special values need be written into any register prior to selection of any instruction. As Table 12-2 shows, certain instructions are reserved for emulator use. For more information, see Table 12-7.

EMUPMD Shift Register

The `EMUPMD` serial shift register is located in the system unit. `EMUPMD` is 48 bits wide and is accessed by the emulator through TAP. When the TAP enters the UPDATE state and `EMUPMD` is selected, a 48-bit slave register is updated from this register. The `EMUPMD` register is used to force the ADSP-21161 processor to execute emulator supplied instructions. The register accomplishes this by driving the instruction bus while in emulation space.

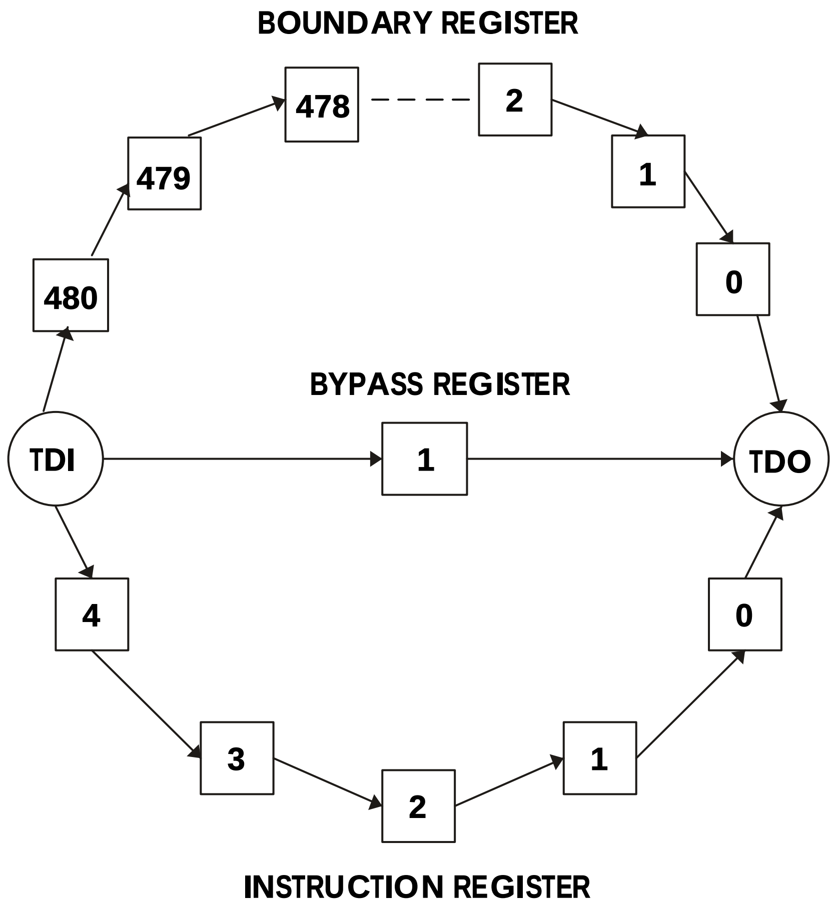


Figure 12-1. Serial Scan Paths

EMUPX Shift Register

The `EMUPX` serial shift register is located in the system unit. The `EMUPX` register is 48 bits wide and is accessed by the emulator through the TAP. When the TAP goes into the UPDATE state and `EMUPX` is selected, the most significant 48-bits of `PX` is updated from `EMUPX`. When the TAP goes into the CAPTURE state and EMUPX is selected, EMUPX is updated with the most significant 48-bits of PX. The EMUPX register is used to transfer data between the emulator and the target system.

The EMUPX register is provided for backwards compatibility with the SHARC ICE hardware and is 64 bits wide. To provide compatibility, only the most significant 48 bits of PX are mapped to EMUPX. 48-bit instructions, and 40-bit extended precision data, are always aligned to the most significant bit. When transferring 32-bit data to/from PX register, PX2 must be specified as the source/destination to ensure that the 32-bits is aligned to the most significant bit.

EMU64PX Shift Register

The EMU64PX serial shift register is located in the system unit. The EMU64PX register is 64 bits wide and is accessed by the emulator through the TAP. When the TAP goes into the UPDATE state and EMU64PX is selected, PX is updated from EMU64PX. When the TAP goes into the CAPTURE state and EMU64PX is selected, EMU64PX is updated from PX. The EMU64PX register transfers data between the emulator and the target system. The most significant 48-bits of EMU64PX are redundantly available in EMUPX.

EMUPC Shift Register

The EMUPC serial shift register is located in the system unit. The EMUPC register is 24 bits wide and is accessed by the emulator through the TAP. It captures addresses from the PC register. This data can be used to statistically profile the user's code. Addresses cannot be forced into the PC register from the EMUPC register.

EMUCTL Shift Register

The `EMUCTL` serial shift register is located in the system unit. The `EMUCTL` register is 40 bits wide and is accessed by the emulator through the TAP. It controls all of the ADSP-21161 processor emulation functionality. Table 12-3 lists this registers bits and describes their function.

Table 12-3. Emulation Control Register (EMUCTL) Definition

Bit #	Name	Function
0	EMUENA	**Emulator Function Enable.** The EMUENA bit enables ADSP-21161 processor emulation functions. (0=ignore breakpoints and emulator interrupts, 1=respond to breakpoints and emulator interrupts)
1	EIRQENA	**Emulator Interrupt Enable.** The EIRQENA bit enables the emulation logic to recognize external emulator interrupts. (0=disable, 1=enable)
2	BKSTOP	**Enable Autostop on Breakpoint.** The BKSTOP bit enables the ADSP-21161 processor to generate an external emulator interrupt when any breakpoint event occurs. (0=disable, 1=enable)
3	SS	**Enable Single Step Mode.** The SS bit enables single-step operation. (0=disable, 1=enable)
4	SYSRST	**Software Reset of the ADSP-21161 processor.** The SYSRST bit resets the ADSP-21161 processor in the same manner as the external $\overline{\text{RESET}}$ pin. The SYSRST bit must be cleared by the emulator. (0=normal operation, 1=reset)
5	ENBRKOUT	**Enable the BRKOUT pin.** The ENBRKOUT bit enables the $\overline{\text{BRKOUT}}$ pin operation. (0=$\overline{\text{BRKOUT}}$ pin at high-impedance state, 1=$\overline{\text{BRKOUT}}$ pin enabled)

Table 12-3. Emulation Control Register (EMUCTL) Definition (Cont'd)

Bit #	Name	Function
6	IOSTOP	**Stop IOP DMAs in EMU space.** The IOSTOP bit disables all DMA requests when the DSP is in emulation space. Data that is currently in the EP, LINK, or SPORT DMA buffers is held there unless the internal DMA request was already granted. IOSTOP causes incoming data to be held off and outgoing data to cease. Because SPORT receive data cannot be held off, it is lost and the overrun bit is set. The direct write buffer (internal memory write) and the EP pad buffer are allowed to flush any remaining data to internal memory. (0=IO continues, 1=IO Stops)
7	EPSTOP	**Stop I/O Processor EP operation in emulation space.** The EPSTOP bit disables all EP requests when the DSP is in emulation space. After an emulation interrupt is acknowledged, EPSTOP deasserts ACK (deasserts REDY if host access) to prevent further data from being accepted if the EP is accessed. The emulator may clear this bit—allowing I/O to continue and the bus to clear—so that the emulator may use the EP (through BR and bus lock). Note that the EP bus clears only if accesses are direct writes or IOP register writes, because all other IOP functions are halted. The EP bus does not clear if accesses to any of the DMA buffers are extended due to a buffer full or empty condition. (0=EP IO continues, 1=EP IO Stops)
8	NEGPA1	**Negate program memory data address breakpoint.** The NEG* bits enable breakpoint events if the address is greater than the end register value OR less than the start register value. This function is useful to detect index range violations in user code. (0=disable breakpoint, 1=enable breakpoint)
9	NEGDA1	**Negate data memory address breakpoint #1**. For more information, see NEGPA1 bit description.
10	NEGDA2	**Negate data memory address breakpoint #2.** For more information, see NEGPA1 bit description.
11	NEGIA1	**Negate instruction address breakpoint #1**. For more information, see NEGPA1 bit description.
12	NEGIA2	**Negate instruction address breakpoint #2.** For more information, see NEGPA1 bit description.

Table 12-3. Emulation Control Register (EMUCTL) Definition (Cont'd)

Bit #	Name	Function
13	NEGIA3	**Negate instruction address breakpoint #3.** For more information, see NEGPA1 bit description.
14	NEGIA4	**Negate instruction address breakpoint #4.** For more information, see NEGPA1 bit description.
15	NEGIO1	**Negate I/O address breakpoint.** For more information, see NEGPA1 bit description.
16	NEGEP1	**Negate EP address breakpoint.** For more information, see NEGPA1 bit description.
17	ENBPA	**Enable program memory data address breakpoints.** The ENB* bits enable each breakpoint group. Note that when the ANDBKP bit is set, breakpoint types not involved in the generation of the effective breakpoint must be disabled. (0=disable breakpoints, 1=enable breakpoints)
18	ENBDA	**Enable data memory address breakpoints.** For more information, see ENBPA bit description.
19	ENBIA	**Enable instruction address breakpoints.** For more information, see ENBPA bit description.
20	ENBIO	**Enable I/O address breakpoint.** For more information, see ENBPA bit description.
21	ENBEP	**Enable external port address breakpoint.** For more information, see ENBPA bit description.
22-23	PA1MODE	**PA1 breakpoint triggering mode.** The breakpoint triggering mode bits trigger on the following conditions: Mode — Triggering condition 00 — Breakpoint is disabled 01 — WRITE accesses only 10 — READ accesses only 11 — any access
24-25	DA1MODE	**DA1 breakpoint triggering mode.** For more information, see PA1MODE bit description.
26-27	DA2MODE	**DA2 breakpoint triggering mode.** For more information, see PA1MODE bit description.

Table 12-3. Emulation Control Register (EMUCTL) Definition (Cont'd)

Bit #	Name	Function
28-29	IO1MODE	**IO1 breakpoint triggering mode.** For more information, see PA1MODE bit description.
30-31	EP1MODE	**EP1 breakpoint triggering mode.** For more information, see PA1MODE bit description.
32	ANDBKP	**AND composite breakpoints.** The ANDBKP bit enables ANDing of each breakpoint type to generate an effective breakpoint from the composite breakpoint signals. (0=OR breakpoint types, 1=AND breakpoint types)
33		Reserved. The ICSA function and DMDSEL bit used by that function not supported on ADSP-21161 processor.
34	NOBOOT	**No power-up boot on reset.** The NOBOOT bit forces the ADSP-21161 processor into the No boot mode. In this mode, the processor does not boot load, but begins fetching instructions from 0x0080 0004 in external memory. (0=disable, 1=force No boot mode)
35	TMODE	**Test mode enable.** The TMODE bit is for Analog Devices' usage only. Do NOT set this bit. (0=normal operation)
36	BHO	**Buffer Hang Override bit.** The BHO control bit overrides the BHD bit in SYSCON, disabling BHD's control over core access of data buffer behavior. Note that the default (reset) state of BHD is now set for ADSP-21161 processor, a change from ADSP-2106x. (0=normal BHD operation, 1=override BHD operation)
37	MTST	**Memory Test Enable Bit.** The MTST bit enables scanning of data for to the latches used for memory test. (0=normal operation, 1=enable memory test)
38, 39		Reserved

EMUSTAT Shift Register

The `EMUSTAT` serial shift register is located in the system unit. It is 8 bits wide and is accessed by the emulator through the TAP. This register is updated by the ADSP-21161 processor when the TAP is in the CAP-

TURE state. The emulator reads EMUSTAT to determine the state of the ADSP-21161 processor. None of the bits in this register can be written by the emulator. All bits are active high. Table 12-4 lists the EMUSTAT register's bits.

Table 12-4. Emulation Status (EMUSTAT) Register Definition

Bit #	Name	Function (If bit=1...)
0	EMUSPACE	Indicates that the next instruction is to be fetched from the emulator.
1	EMUREADY	Indicates that the ADSP-21161 processor has finished executing the previous emulator instruction.
2	INIDLE	Indicates that the ADSP-21161 processor was in IDLE prior to the latest emulator interrupt.
3	COMHALT	Indicates a core access to a SPORT or a LINK is hung because of an external device.
4	EPHALT	Indicates a core access to a DMA buffer is hung because of the external port.
5-7		Reserved

BRKSTAT Shift Register

The BRKSTAT serial shift register is located in the system unit. It is 16 bits wide and is accessed by the emulator through the TAP. This register monitors the status of the emulation breakpoints and is updated on every clock cycle. None of the bits of this register can be written by the emulator.

Table 12-5 lists the BRKSTAT register's bits. A high bit indicates a breakpoint hit. When a breakpoint hit occurs, the register ceases updating. Stopping allows the emulator to see which breakpoint was triggered. When the ADSP-21161 processor leaves emulation space the BRKSTAT register is cleared and resumes updating. All status bits are synchronized to TCLK before being scanned out.

Table 12-5. BRKSTAT (Breakpoint Status) Register Definition

Bit #	Name	Function (If bit=1...)
0	STATPA	Program Memory Data breakpoint hit
1	STATDA0	Data Memory breakpoint hit
2	STATDA1	Data Memory breakpoint hit
3	STATIA0	Instruction Address breakpoint hit
4	STATIA1	Instruction Address breakpoint hit
5	STATIA2	Instruction Address breakpoint hit
6	STATIA2	Instruction Address breakpoint hit
7	STATIO	I/O Address breakpoint hit
8	STATEP	EP Address breakpoint hit
9-15	Reserved	

MEMTST Shift Register

The MEMTST serial shift register is for Analog Devices' usage only.

Do not attempt to use this register—incorrect usage of this feature can result in permanent damage to the ADSP-21161 processor being tested.

PSx, DMx, IOx, and EPx (Breakpoint) Registers

The PSx, DMx, IOx, and EPx (Breakpoint) registers are located in the I/O processor register set. The emulation breakpoint registers are not user accessible and can be written only when the ADSP-21161 processor is in emulation space or test mode. The breakpoint registers vary in size according to the address type: instruction (24-bit address), data (32-bit address), or I/O data (19-bit address)—Table 12-6 shows the sizes.

The ADSP-21161 processor contains nine sets of emulation breakpoint registers. Each set consists of a start and end register which describe an address range, with the start register setting the lower end of the address range. Each breakpoint set monitors a particular address bus. When a valid address is in the address range, than a breakpoint signal is generated. The address range includes the start and end addresses.

The nine breakpoint sets are grouped into five types: instruction (IA), DM data (DA), PM data (PA), IO data (IO), and EP data (EP). The individual breakpoint signals in each type are ORed together to create five composite breakpoint signals.

These composite signals can be optionally ANDed or ORed together to create the effective breakpoint event signal used to generate an emulator interrupt. The ANDBKP bit in the EMUCTL register selects the function used.

Each breakpoint type has an enable bit in the EMUCTL register. When set, these bits add the specified breakpoint type into the generation of the effective breakpoint signal. If cleared, the specified breakpoint type is not used in the generation of the effective breakpoint signal. This allows the user to trigger the effective breakpoint from a subset of the breakpoint types.

To provide further flexibility, each individual breakpoint can be programmed to trigger if the address is in range AND one of these conditions is met: READ access, WRITE access, ANY access, or NO access. The control bits for this feature are also located in EMUCTL. For more information, see PA1MODES bit description in Table 12-3 on page 12-8.

The address ranges of the emulation breakpoint registers are negated by setting the appropriate renege negation bits in the EMUCTL register. For more information, see NEGPA1 bit description Table 12-3 on page 12-8. Each breakpoint can be disabled by setting the start address larger than the end address.

Four of the breakpoints monitor the instruction address. Two monitor the data memory address. One monitors the program memory data address, one monitors the I/O address bus and one monitors the EP address bus.

The instruction address breakpoints monitor the address of the instruction being executed, not the address of the instruction being fetched. If the current execution is aborted, the breakpoint signal does not occur even if the address is in range. Data address breakpoints (DA and PA only) are also ignored during aborted instructions. The nine breakpoint sets appear in Table 12-6.

Table 12-6. PSx, DMx, IOx, and EPx (Breakpoint) Registers

Register	Function	Group[1]
PSA1S	Instruction Address Start #1	IA
PSA1E	Instruction Address End #1	IA
PSA2S	Instruction Address Start #2	IA
PSA2E	Instruction Address End #2	IA
PSA3S	Instruction Address Start #3	IA
PSA3E	Instruction Address End #3	IA
PSA4S	Instruction Address Start #4	IA
PSA4E	Instruction Address End #4	IA
DMA1S	Data Address Start #1	DA
DMA1E	Data Address End #1	DA
DMA2S	Data Address Start #2	DA
DMA2E	Data Address End #2	DA
PMDAS	Program Data Address Start	PA
PMDAE	Program Data Address End	PA
IOAS	I/O Address Start	IO
IOAE	I/O Address End	IO

Table 12-6. PSx, DMx, IOx, and EPx (Breakpoint) Registers (Cont'd)

Register	Function	Group[1]
EPAS	External Port Address Start	EP
EPAE	External Port Address End	EP

1 Group IA=24-bit addresses, Groups DA, PA, and EP=32-bit addresses, Group IO=19-bit addresses.

EMUN Register

The EMUN (Nth event counter) register is located in the I/O Processor register set. It is not user accessible and can be written only when the ADSP-21161 processor is in emulation space. The EMUN register is read-only from normal-space and can be written only when the ADSP-21161 processor is in emulation space. The Nth event counter allows an emulation breakpoint to occur on the Nth occurrence of the breakpoint event. This is accomplished by writing the desired Nth value to the EMUN register in UREG space. This register can be read from normal space, but it can be written only in emulation space. The counter decrements on each occurrence of the breakpoint event, asserting the interrupt when the counter is equal to zero and the hardware breakpoint event occurs.

EMUCLK and EMUCLK2 Registers

The EMUCLK (clock counter) and EMUCLK2 (clock counter scaling) registers are located in the universal (UREG) register set. EMUCLK and EMUCLK2 are not user accessible and can be written only when the ADSP-21161 processor is in emulation space. These registers are read-only from normal-space and can be written only when the ADSP-21161 processor is in emulation space. The Emulation Clock Counter consists of a 32-bit count register (EMUCLK) and a 32-bit scaling register (EMUCLK2). The EMUCLK counts clock cycles while the user has control of the ADSP-21161 processor and stops counting when the emulator gains control. These registers let you gauge the amount of time spent executing a particular section of code. The EMUCLK2 register extends the time EMUCLK can count by incrementing each time the EMUCLK value rolls over to zero. The combined emulation clock counter can count accurately for thousands of hours.

EMUIDLE Instruction

The EMUIDLE instruction places the ADSP-21161 processor in the idle state and triggers an emulator interrupt. This operation lets you use the EMUIDLE instruction to be used as a software breakpoint. When EMUIDLE is executed, the emulation clock counter immediately halts.

In Circuit Signal Analyzer (ICSA) Function

This function is NOT supported in the ADSP-21161 processor.

Boundary Register

The Boundary register is 481 bits long. This section defines the latch type and function of each position in the scan path. The positions are numbered with 0 being the first bit output (closest to TDO) and 480 being the last (closest to TDI). The following are some notes about boundary registers:

- Scan position 0 (NC_0) is the end is closest to TDO (scan in first)
- Scan position 480 (SPARE); this end is closest to TDI (scan in last)
- Output Enables:

  1 = Drive the associated signals during the EXTEST and INTEST instructions

  0 = Three-state the associated signals during the EXTEST and INTEST instructions

Table 12-7. JTAG Boundary Register

Scan #	Signal Name	Latch Type	Scan #	Signal Name	Latch Type
0	NC(I)	OUTP Closest to TDO scan in first	23	FLAG[8](I/O)	IN
1	NC(I)	OE	24	FLAG[7](I/O)	OUTP
2	NC(I)	IN	25	FLAG[7](I/O)	OE
3	BMSTR(O)	OUTP	26	FLAG[7](I/O)	IN
4	BMSTR(O)	OE	27	FLAG[6](I/O)	OUTP
5	BMSTR(O)	IN	28	FLAG[6](I/O)	OE
6	$\overline{\text{EMU}}$(O)	OUTP	29	FLAG[6](I/O)	IN
7	$\overline{\text{EMU}}$(O)	OE	30	FLAG[5](I/O)	OUTP
8	$\overline{\text{EMU}}$(O)	IN	31	FLAG[5](I/O)	OE
9	$\overline{\text{RESET}}$(I)	OUTP	32	FLAG[5](I/O)	IN
10	$\overline{\text{RESET}}$(I)	OE	33	FLAG[4](I/O)	OUTP
11	$\overline{\text{RESET}}$(I)	IN	34	FLAG[4](I/O)	OE
12	FLAG[11](I/O)	OUTP	35	FLAG[4](I/O)	IN
13	FLAG[11](I/O)	OE	36	FLAG[3](I/O)	OUTP
14	FLAG[11](I/O)	IN	37	FLAG[3](I/O)	OE
15	FLAG[10](I/O)	OUTP	38	FLAG[3](I/O)	IN
16	FLAG[10](I/O)	OE	39	FLAG[2](I/O)	OUTP
17	FLAG[10](I/O)	IN	40	FLAG[2](I/O)	OE
18	FLAG[9](I/O)	OUTP	41	FLAG[2](I/O)	IN
19	FLAG[9](I/O)	OE	42	FLAG[1](I/O)	OUTP
20	FLAG[9](I/O)	IN	43	FLAG[1](I/O)	OE
21	FLAG[8](I/O)	OUTP	44	FLAG[1](I/O)	IN
22	FLAG[8](I/O)	OE	45	FLAG[0](I/O)	OUTP
46	FLAG[0](I/O)	OE	70	ADDR[23](I/O)	OE

Table 12-7. JTAG Boundary Register (Cont'd)

Scan #	Signal Name	Latch Type	Scan #	Signal Name	Latch Type
47	FLAG[0](I/O)	IN	71	ADDR[23](I/O)	IN
48	IRQ0(I)	OUTP	72	ADDR[22](I/O)	OUTP
49	IRQ0(I)	OE	73	ADDR[22](I/O)	OE
50	IRQ0(I)	IN	74	ADDR[22](I/O)	IN
51	IRQ1(I)	OUTP	75	ADDR[21](I/O)	OUTP
52	IRQ1(I)	OE	76	ADDR[21](I/O)	OE
53	IRQ1(I)	IN	77	ADDR[21](I/O)	IN
54	IRQ2(I)	OUTP	78	ADDR[20](I/O)	OUTP
55	IRQ2(I)	OE	79	ADDR[20](I/O)	OE
56	IRQ2(I)	IN	80	ADDR[20](I/O)	IN
57	ID0(I)	OUTP	81	ADDR[19](I/O)	OUTP
58	ID0(I)	OE	82	ADDR[19](I/O)	OE
59	ID0(I)	IN	83	ADDR[19](I/O)	IN
60	ID1(I)	OUTP	84	ADDR[18](I/O)	OUTP
61	ID1(I)	OE	85	ADDR[18](I/O)	OE
62	ID1(I)	IN	86	ADDR[18](I/O)	IN
63	ID2(I)	OUTP	87	ADDR[17](I/O)	OUTP
64	ID2(I)	OE	88	ADDR[17](I/O)	OE
65	ID2(I)	IN	89	ADDR[17](I/O)	IN
66	TIMEXP(O)	OUTP	90	ADDR[16](I/O)	OUTP
67	TIMEXP(O)	OE	91	ADDR[16](I/O)	OE
68	TIMEXP(O)	IN	92	ADDR[16](I/O)	IN
69	ADDR[23](I/O)	OUTP	93	ADDR[15](I/O)	OUTP
94	ADDR[15](I/O)	OE	118	ADDR[7](I/O)	OE

Table 12-7. JTAG Boundary Register (Cont'd)

Scan #	Signal Name	Latch Type	Scan #	Signal Name	Latch Type
95	ADDR[15](I/O)	IN	119	ADDR[7](I/O)	IN
96	ADDR[14](I/O)	OUTP	120	ADDR[6](I/O)	OUTP
97	ADDR[14](I/O)	OE	121	ADDR[6](I/O)	OE
98	ADDR[14](I/O)	IN	122	ADDR[6](I/O)	IN
99	ADDR[13](I/O)	OUTP	123	ADDR[5](I/O)	OUTP
100	ADDR[13](I/O)	OE	124	ADDR[5](I/O)	OE
101	ADDR[13](I/O)	IN	125	ADDR[5](I/O)	IN
102	ADDR[12](I/O)	OUTP	126	ADDR[4](I/O)	OUTP
103	ADDR[12](I/O)	OE	127	ADDR[4](I/O)	OE
104	ADDR[12](I/O)	IN	128	ADDR[4](I/O)	IN
105	ADDR[11](I/O)	OUTP	129	ADDR[3](I/O)	OUTP
106	ADDR[11](I/O)	OE	130	ADDR[3](I/O)	OE
107	ADDR[11](I/O)	IN	131	ADDR[3](I/O)	IN
108	ADDR[10](I/O)	OUTP	132	ADDR[2](I/O)	OUTP
109	ADDR[10](I/O)	OE	133	ADDR[2](I/O)	OE
110	ADDR[10](I/O)	IN	134	ADDR[2](I/O)	IN
111	ADDR[9](I/O)	OUTP	135	ADDR[1](I/O)	OUTP
112	ADDR[9](I/O)	OE	136	ADDR[1](I/O)	OE
113	ADDR[9](I/O)	IN	137	ADDR[1](I/O)	IN
114	ADDR[8](I/O)	OUTP	138	ADDR[0](I/O)	OUTP
115	ADDR[8](I/O)	OE	139	ADDR[0](I/O)	OE
116	ADDR[8](I/O)	IN	140	ADDR[0](I/O)	IN
117	ADDR[7](I/O)	OUTP	141	$\overline{MS}$3(I/O)	OUTP
142	$\overline{MS}$3(I/O)	OE	165	BR4(I/O)	OUTP

Table 12-7. JTAG Boundary Register (Cont'd)

Scan #	Signal Name	Latch Type
143	$\overline{\text{MS}}$3(I/O)	IN
144	$\overline{\text{MS}}$2(I/O)	OUTP
145	$\overline{\text{MS}}$2(I/O)	OE
146	$\overline{\text{MS}}$2(I/O)	IN
147	$\overline{\text{MS}}$1(I/O)	OUTP
148	$\overline{\text{MS}}$1(I/O)	OE
149	$\overline{\text{MS}}$1(I/O)	IN
150	$\overline{\text{MS}}$0(I/O)	OUTP
151	$\overline{\text{MS}}$0(I/O)	OE
152	$\overline{\text{MS}}$0(I/O)	IN
153	$\overline{\text{SBTS}}$(I)	OUTP
154	$\overline{\text{SBTS}}$(I)	OE
155	$\overline{\text{SBTS}}$(I)	IN
156	$\overline{\text{PA}}$(I/O)	OUTP (Formerly CPA)
157	$\overline{\text{PA}}$(I/O)	OE (Formerly CPA)
158	$\overline{\text{PA}}$(I/O)	IN (Formerly CPA)
159	BR6(I/O)	OUTP
160	BR6(I/O)	OE
161	BR6(I/O)	IN
162	BR5(I/O)	OUTP
163	BR5(I/O)	OE
164	BR5(I/O)	IN
188	SDCLK1(O)	IN
166	BR4(I/O)	OE
167	BR4(I/O)	IN
168	BR3(I/O)	OUTP
169	BR3(I/O)	OE
170	BR3(I/O)	IN
171	BR2(I/O)	OUTP
172	BR2(I/O)	OE
173	BR2(I/O)	IN
174	BR1(I/O)	OUTP
175	BR1(I/O)	OE
176	BR1(I/O)	IN
177	$\overline{\text{WR}}$(I/O)	OUTP
178	$\overline{\text{WR}}$(I/O)	OE
179	$\overline{\text{WR}}$(I/O)	IN
180	$\overline{\text{RD}}$(I/O)	OUTP
181	$\overline{\text{RD}}$(I/O)	OE
182	$\overline{\text{RD}}$(I/O)	IN
183	BRST(I/O)	OUTP
184	BRST(I/O)	OE
185	BRST(I/O)	IN
186	SDCLK1(O)	OUTP
187	SDCLK1(O)	OE
212	$\overline{\text{HBG}}$(I/O)	IN

Table 12-7. JTAG Boundary Register (Cont'd)

Scan #	Signal Name	Latch Type	Scan #	Signal Name	Latch Type
189	SDA10(O)	OUTP	213	REDY(O)	OUTP
190	SDA10(O)	OE	214	REDY(O)	OE
191	SDA10(O)	IN	215	REDY(O)	IN
192	SDCKE(I/O)	OUTP	216	ACK(I/O)	OUTP
193	SDCKE(I/O)	OE	217	ACK(I/O)	OE
194	SDCKE(I/O)	IN	218	ACK(I/O)	IN
195	CLKOUT(O)	OUTP	219	$\overline{\text{CS}}$(I)	OUTP
196	CLKOUT(O)	OE	220	$\overline{\text{CS}}$(I)	OE
197	CLKOUT(O)	IN	221	$\overline{\text{CS}}$(I)	IN
198	SDCLK0(I/O)	OUTP	222	CLKDBL(I)	OUTP
199	SDCLK0(I/O)	OE	223	CLKDBL(I)	OE
200	SDCLK0 (I/O)	IN	224	CLKDBL(I)	IN
201	$\overline{\text{CAS}}$(I/O)	OUTP	225	DQM(O)	OUTP
202	$\overline{\text{CAS}}$(I/O)	OE	226	DQM(O)	OE
203	$\overline{\text{CAS}}$(I/O)	IN	227	DQM(O)	IN
204	$\overline{\text{RAS}}$(I/O)	OUTP	228	$\overline{\text{SDWE}}$(I/O)	OUTP
205	$\overline{\text{RAS}}$(I/O)	OE	229	$\overline{\text{SDWE}}$(I/O)	OE
206	$\overline{\text{RAS}}$(I/O)	IN	230	$\overline{\text{SDWE}}$(I/O)	IN
207	$\overline{\text{HBR}}$(I)	OUTP	231	CLK_CFG1(I)	OUTP
208	$\overline{\text{HBR}}$(I)	OE	232	CLK_CFG1(I)	OE
209	$\overline{\text{HBR}}$(I)	IN	233	CLK_CFG1(I)	IN
210	$\overline{\text{HBG}}$(I/O)	OUTP	234	CLK_CFG0(I)	OUTP
211	$\overline{\text{HBG}}$(I/O)	OE	235	CLK_CFG0(I)	OE
236	CLK_CFG0(I)	IN	260	DATA[19](I/O)	IN

Table 12-7. JTAG Boundary Register (Cont'd)

Scan #	Signal Name	Latch Type	Scan #	Signal Name	Latch Type
237	DMAR2(I)	OUTP	261	DATA[20](I/O)	OUTP
238	DMAR2(I)	OE	262	DATA[20](I/O)	OE
239	DMAR2(I)	IN	263	DATA[20](I/O)	IN
240	DMAG2(O)	OUTP	264	DATA[21](I/O)	OUTP
241	DMAG2(O)	OE	265	DATA[21](I/O)	OE
242	DMAG2(O)	IN	266	DATA[21](I/O)	IN
243	DMAR1(I)	OUTP	267	DATA[22](I/O)	OUTP
244	DMAR1(I)	OE	268	DATA[22](I/O)	OE
245	DMAR1(I)	IN	269	DATA[22](I/O)	IN
246	DMAG1(O)	OUTP	270	DATA[23](I/O)	OUTP
247	DMAG1(O)	OE	271	DATA[23](I/O)	OE
248	DMAG1(O)	IN	272	DATA[23](I/O)	IN
249	DATA[16](I/O)	OUTP	273	DATA[24](I/O)	OUTP
250	DATA[16](I/O)	OE	274	DATA[24](I/O)	OE
251	DATA[16](I/O)	IN	275	DATA[24](I/O)	IN
252	DATA[17](I/O)	OUTP	276	DATA[25](I/O)	OUTP
253	DATA[17](I/O)	OE	277	DATA[25](I/O)	OE
254	DATA[17](I/O)	IN	278	DATA[25](I/O)	IN
255	DATA[18](I/O)	OUTP	279	DATA[26](I/O)	OUTP
256	DATA[18](I/O)	OE	280	DATA[26](I/O)	OE
257	DATA[18](I/O)	IN	281	DATA[26](I/O)	IN
258	DATA[19](I/O)	OUTP	282	DATA[27](I/O)	OUTP
259	DATA[19](I/O)	OE	283	DATA[27](I/O)	OE
284	DATA[27](I/O)	IN	308	DATA[35](I/O)	IN

Table 12-7. JTAG Boundary Register (Cont'd)

Scan #	Signal Name	Latch Type
285	DATA[28](I/O)	OUTP
286	DATA[28](I/O)	OE
287	DATA[28](I/O)	IN
288	DATA[29](I/O)	OUTP
289	DATA[29](I/O)	OE
290	DATA[29](I/O)	IN
291	DATA[30](I/O)	OUTP
292	DATA[30](I/O)	OE
293	DATA[30](I/O)	IN
294	DATA[31](I/O)	OUTP
295	DATA[31](I/O)	OE
296	DATA[31](I/O)	IN
297	DATA[32](I/O)	OUTP
298	DATA[32](I/O)	OE
299	DATA[32](I/O)	IN
300	DATA[33](I/O)	OUTP
301	DATA[33](I/O)	OE
302	DATA[33](I/O)	IN
303	DATA[34](I/O)	OUTP
304	DATA[34](I/O)	OE
305	DATA[34](I/O)	IN
306	DATA[35](I/O)	OUTP
307	DATA[35](I/O)	OE
332	DATA[43](I/O)	IN

Scan #	Signal Name	Latch Type
309	DATA[36](I/O)	OUTP
310	DATA[36](I/O)	OE
311	DATA[36](I/O)	IN
312	DATA[37](I/O)	OUTP
313	DATA[37](I/O)	OE
314	DATA[37](I/O)	IN
315	DATA[38](I/O)	OUTP
316	DATA[38](I/O)	OE
317	DATA[38](I/O)	IN
318	DATA[39](I/O)	OUTP
319	DATA[39](I/O)	OE
320	DATA[39](I/O)	IN
321	DATA[40](I/O)	OUTP
322	DATA[40](I/O)	OE
323	DATA[40](I/O)	IN
324	DATA[41](I/O)	OUTP
325	DATA[41](I/O)	OE
326	DATA[41](I/O)	IN
327	DATA[42](I/O)	OUTP
328	DATA[42](I/O)	OE
329	DATA[42](I/O)	IN
330	DATA[43](I/O)	OUTP
331	DATA[43](I/O)	OE
356	L1DAT[2](I/O)	IN

Table 12-7. JTAG Boundary Register (Cont'd)

Scan #	Signal Name	Latch Type	Scan #	Signal Name	Latch Type
333	DATA[44](I/O)	OUTP	357	L1DAT[3](I/O)	OUTP
334	DATA[44](I/O)	OE	358	L1DAT[3](I/O)	OE
335	DATA[44](I/O)	IN	359	L1DAT[3](I/O)	IN
336	DATA[45](I/O)	OUTP	360	L1ACK(I/O)	OUTP
337	DATA[45](I/O)	OE	361	L1ACK(I/O)	OE
338	DATA[45](I/O)	IN	362	L1ACK(I/O)	IN
339	DATA[46](I/O)	OUTP	363	L1CLK(I/O)	OUTP
340	DATA[46](I/O)	OE	364	L1CLK(I/O)	OE
341	DATA[46](I/O)	IN	365	L1CLK(I/O)	IN
342	DATA[47](I/O)	OUTP	366	L1DAT[4](I/O)	OUTP
343	DATA[47](I/O)	OE	367	L1DAT[4](I/O)	OE
344	DATA[47](I/O)	IN	368	L1DAT[4](I/O)	IN
345	$\overline{\text{RSTOUT}}$(O)[1]	OUTP	369	L1DAT[5](I/O)	OUTP
346	$\overline{\text{RSTOUT}}$(O)	OE	370	L1DAT[5](I/O)	OE
347	$\overline{\text{RSTOUT}}$(O)	IN	371	L1DAT[5](I/O)	IN
348	L1DAT[0](I/O)	OUTP	372	L1DAT[6](I/O)	OUTP
349	L1DAT[0](I/O)	OE	373	L1DAT[6](I/O)	OE
350	L1DAT[0](I/O)	IN	374	L1DAT[6](I/O)	IN
351	L1DAT[1](I/O)	OUTP	375	L1DAT[7](I/O)	OUTP
352	L1DAT[1](I/O)	OE	376	L1DAT[7](I/O)	OE
353	L1DAT[1](I/O)	IN	377	L1DAT[7](I/O)	IN
354	L1DAT[2](I/O)	OUTP	378	L0DAT[0](I/O)	OUTP
355	L1DAT[2](I/O)	OE	379	L0DAT[0](I/O)	OE
380	L0DAT[0](I/O)	IN	404	L0DAT[6](I/O)	IN

Table 12-7. JTAG Boundary Register (Cont'd)

Scan #	Signal Name	Latch Type	Scan #	Signal Name	Latch Type
381	L0DAT[1](I/O)	OUTP	405	L0DAT[7](I/O)	OUTP
382	L0DAT[1](I/O)	OE	406	L0DAT[7](I/O)	OE
383	L0DAT[1](I/O)	IN	407	L0DAT[7](I/O)	IN
384	L0DAT[2](I/O)	OUTP	408	FS3(I/O)	OUTP
385	L0DAT[2](I/O)	OE	409	FS3(I/O)	OE
386	L0DAT[2](I/O)	IN	410	FS3(I/O)	IN
387	L0DAT[3](I/O)	OUTP	411	SCLK3(I/O)	OUTP
388	L0DAT[3](I/O)	OE	412	SCLK3(I/O)	OE
389	L0DAT[3](I/O)	IN	413	SCLK3(I/O)	IN
390	L0ACK(I/O)	OUTP	414	D3B(I/O)	OUTP
391	L0ACK(I/O)	OE	415	D3B(I/O)	OE
392	L0ACK(I/O)	IN	416	D3B(I/O)	IN
393	L0CLK(I/O)	OUTP	417	D3A(I/O)	OUTP
394	L0CLK(I/O)	OE	418	D3A(I/O)	OE
395	L0CLK(I/O)	IN	419	D3A(I/O)	IN
396	L0DAT[4](I/O)	OUTP	420	FS2(I/O)	OUTP
397	L0DAT[4](I/O)	OE	421	FS2(I/O)	OE
398	L0DAT[4](I/O)	IN	422	FS2(I/O)	IN
399	L0DAT[5](I/O)	OUTP	423	SCLK2(I/O)	OUTP
400	L0DAT[5](I/O)	OE	424	SCLK2(I/O)	OE
401	L0DAT[5](I/O)	IN	425	SCLK2(I/O)	IN
402	L0DAT[6](I/O)	OUTP	426	D2B(I/O)	OUTP
403	L0DAT[6](I/O)	OE	427	D2B(I/O)	OE
428	D2B(I/O)	IN	452	EBOOT(I)	IN

Table 12-7. JTAG Boundary Register (Cont'd)

Scan #	Signal Name	Latch Type
429	D2A(I/O)	OUTP
430	D2A(I/O)	OE
431	D2A(I/O)	IN
432	FS1(I/O)	OUTP
433	FS1(I/O)	OE
434	FS1(I/O)	IN
435	LBOOT(I)	OUTP
436	LBOOT(I)	OE
437	LBOOT(I)	IN
438	SCLK1(I/O)	OUTP
439	SCLK1(I/O)	OE
440	SCLK1(I/O)	IN
441	D1B(I/O)	OUTP
442	D1B(I/O)	OE
443	D1B(I/O)	IN
444	D1A(I/O)	OUTP
445	D1A(I/O)	OE
446	D1A(I/O)	IN
447	FS0(I/O)	OUTP
448	FS0(I/O)	OE
449	FS0(I/O)	IN
450	EBOOT(I)	OUTP
451	EBOOT(I)	OE
452	EBOOT(I)	IN
453	SCLK0(I/O)	OUTP
454	SCLK0(I/O)	OE
455	SCLK0(I/O)	IN
456	D0B(I/O)	OUTP
457	D0B(I/O)	OE
458	D0B(I/O)	IN
459	D0A(I/O)	OUTP
460	D0A(I/O)	OE
461	D0A(I/O)	IN
462	$\overline{\text{SPIDS}}$(I)	OUTP
463	$\overline{\text{SPIDS}}$(I)	OE
464	$\overline{\text{SPIDS}}$(I)	IN
465	SPICLK(I/O)	OUTP
466	SPICLK(I/O)	OE
467	SPICLK(I/O)	IN
468	MOSI(I/O)	OUTP
469	MOSI(I/O)	OE
470	MOSI(I/O)	IN
471	MISO(I/O)	OUTP
472	MISO(I/O)	OE
473	MISO(I/O)	IN
474	$\overline{\text{BMS}}$(I/O)	OUTP
475	$\overline{\text{BMS}}$(I/O)	OE
476	$\overline{\text{BMS}}$(I/O)	IN

Table 12-7. JTAG Boundary Register (Cont'd)

Scan #	Signal Name	Latch Type
477	RPBA(I)	OUTP
478	RPBA(I)	OE
479	RPBA(I)	IN
480	SPARE	Closest to TDI scan in last

1 $\overline{\text{RSTOUT}}$ only exists for silicon revisions 1.2 and greater.

Device Identification Register

No device identification register is included in the ADSP-21161 processor.

Built-In Self-Test Operation (BIST)

No self-test functions are supported by the ADSP-21161 processor.

Private Instructions

Table 12-2 on page 12-4 lists the private instructions that are reserved for emulation and memory test. The ADSP-21161 processor JTAG ICE emulator uses the TAP and boundary scan as a way to access the processor in the target system. The JTAG ICE emulator requires a target board connector for access to the TAP. For more information, see "Designing For JTAG Emulation" on page 13-49.

References

- IEEE Standard 1149.1-1990. Standard Test Access Port and Boundary-Scan Architecture.

  To order a copy, contact IEEE at 1-800-678-IEEE.

- Maunder, C.M. and R. Tulloss. Test Access Ports and Boundary Scan Architectures.

  IEEE Computer Society Press, 1991.

- Parker, Kenneth. The Boundary Scan Handbook.

  Kluwer Academic Press, 1992.

- Bleeker, Harry, P. van den Eijnden, and F. de Jong. Boundary-Scan Test—A Practical Approach.

  Kluwer Academic Press, 1993.

- Hewlett-Packard Co. HP Boundary-Scan Tutorial and BSDL Reference Guide.

  (HP part# E1017-90001.) 1992.

13 SYSTEM DESIGN

The ADSP-21161 processor supports many system design options. The options implemented in a system are influenced by cost, performance, and system requirements. This chapter provides the following system design information:

- "Pin Descriptions" on page 13-2
- "Dual-Voltage Power-up Sequencing" on page 13-41
- "Designing For JTAG Emulation" on page 13-49
- "Conditioning Input Signals" on page 13-60
- "Designing For High Frequency Operation" on page 13-62
- "Booting Single and Multiple Processors" on page 13-71

Other chapters also discuss system design issues. Some other locations for system design information include:

- "Setting External Port Modes" on page 7-3
- "Setting Link Port Modes" on page 9-5
- "SPORT Operation Modes" on page 10-47
- "SPI Operation Modes" on page 11-24

By following the guidelines described in this chapter, you can design the JTAG emulation interface for an Analog Devices target board. Development and testing of your application code and hardware can begin without debugging the debug port.

Pin Descriptions

This section describes the pins of the ADSP-21161 processor and shows how these signals can be used in a ADSP-21161 processor system. All I/O pins except CLKIN and XTAL have an internal 50kΩ resister that is enabled during reset. Figure 13-1 illustrates how the pins are used in a single-processor system. Figure 7-29 on page 7-91 shows a system diagram illustrating pin connections in an multiprocessor cluster.

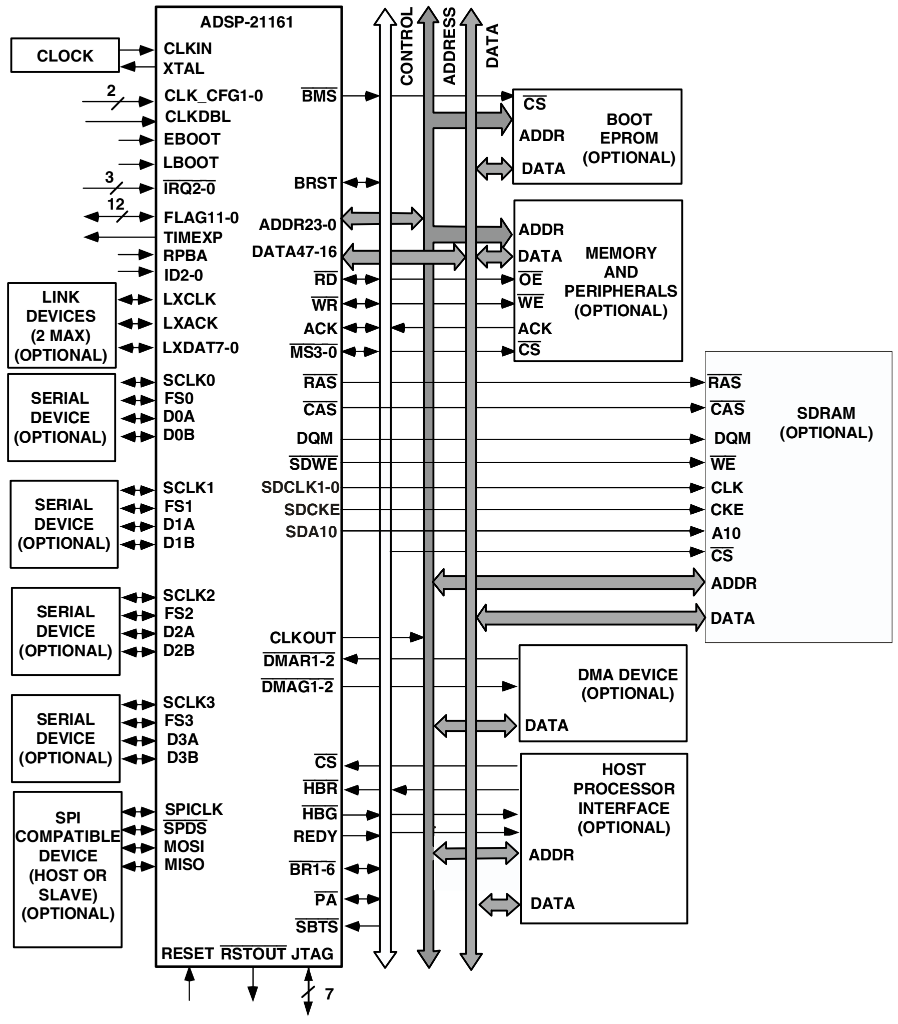


Figure 13-1. Single Processor System

ADSP-21161 processor pin definitions are listed in Table 13-1. The following symbols appear in the **Type** column of Table 13-1:

A	Asynchronous
G	Ground
I	Input
O	Output
P	Power Supply
S	Synchronous
(a/d)	Active Drive
(o/d)	Open Drain
T	Three-State (when $\overline{\text{SBTS}}$ is asserted or the processor is bus slave)

Table 13-1. Pin Descriptions

Pin	Type	Function
ACK	I/O/S	**Memory Acknowledge**. External devices can deassert ACK (low) to add wait states to an external memory access. ACK is used by I/O devices, memory controllers, or other peripherals to hold off completion of an external memory access. The ADSP-21161 processor deasserts ACK as an output to add wait states to a synchronous access of its IOP registers. ACK has a 20kΩ internal pull-up resistor that is enabled during reset or on processors with ID2-0=00x.
ADDR23-0	I/O/T	**External Bus Address**. The ADSP-21161 processor outputs addresses for external memory and peripherals on these pins. In a multiprocessor system the bus master outputs addresses for read/writes of the IOP registers of other ADSP-21161 processors while all other internal memory resources can be accessed indirectly via DMA control (that is, accessing IOP DMA parameter registers). The ADSP-21161 processor inputs addresses when a host processor or multiprocessing bus master is reading or writing its IOP registers. A keeper latch on the processor's ADDR23-0 pins maintains the input at the level it was last driven. This latch is only enabled on processors with ID2-0=00x.

Table 13-1. Pin Descriptions (Cont'd)

Pin	Type	Function
AVDD	P	**Analog Power Supply**. Nominally +1.8V dc and supplies the processor's internal PLL (clock generator). This pin has the same specifications as V_{DDINT}, except that added filtering circuitry is required. For more information, see "Power Supplies" in the ADSP-21161 Data Sheet.
AGND	G	**Analog Power Supply Return**.
$\overline{BR}$6-1	I/O/S	**Multiprocessing Bus Requests**. Used by multiprocessing ADSP-21161 processor's to arbitrate for bus mastership. An ADSP-21161 processor only drives its own $\overline{BR}$x line (corresponding to the value of its ID2-0 inputs) and monitors all others. In a multiprocessor system with less than six ADSP-21161 processors, the unused $\overline{BR}$x pins should be pulled high; the processor's own $\overline{BR}$x line must not be pulled high or low because it is an output.
$\overline{BMS}$	I/O/T	**Boot Memory Select**. Serves as an output or input as selected with the EBOOT and LBOOT pins; see Table 13-11 on page 13-72. This input is a system configuration selection that should be hardwired. For Host and PROM boot, DMA channel 10 (EPB0) is used. For Link Boot and SPI boot, DMA channel 8 is used. *Three-state only in EPROM boot mode (when $\overline{BMS}$ is an output).
BMSTR	O	**Bus Master Output.** In a multiprocessor system, indicates whether the ADSP-21161 processor is current bus master of the shared external bus. The ADSP-21161 processor drives BMSTR high only while it is the bus master. In a single-processor system (ID2-0 = 000), the processor drives this pin high.

Table 13-1. Pin Descriptions (Cont'd)

Pin	Type	Function
BRST	I/O/T	**Sequential Burst Access.** BRST is asserted by ADSP-21161 processor to indicate that data associated with consecutive addresses is being read or written. A slave device samples the initial address and increments an internal address counter after each transfer. The incremented address is not pipelined on the bus. A master ADSP-21161 processor in a multiprocessor environment can read slave external port buffers (EPBx) using the burst protocol. BRST is asserted after the initial access of a burst transfer. It is asserted for every cycle after that, except for the last data request cycle (denoted by $\overline{\text{RD}}$ or $\overline{\text{WR}}$ asserted and BRST negated). A keeper latch on the processor's BRST pin maintains the input at the level it was last driven. This latch is only enabled on processors with ID2-0=00x.
$\overline{\text{CAS}}$	I/O/T	**SDRAM Column Access Strobe.** In conjunction with $\overline{\text{RAS}}$, $\overline{\text{MSx}}$, $\overline{\text{SDWE}}$, $\overline{\text{SDCLKx}}$, and sometimes SDA10, defines the operation for the SDRAM to perform.
CLKIN	I	**Local Clock In.** Used in conjunction with XTAL. CLKIN is the ADSP-21161 processor clock input. It configures the ADSP-21161 processor to use either its internal clock generator or an external clock source. Connecting the necessary components to CLKIN and XTAL enables the internal clock generator. Connecting the external clock to CLKIN while leaving XTAL unconnected configures the ADSP-21161 processor to use the external clock source such as an external clock oscillator.The ADSP-21161 processor external port cycles at the frequency of CLKIN. The instruction cycle rate is a multiple of the CLKIN frequency; it is programmable at power-up via the CLK_CFG1-0 pins. CLKIN may not be halted, changed, or operated below the specified frequency.
CLK_CFG1-0	I	**Core/CLKIN Ratio Control.** ADSP-21161 processor core clock (instruction cycle) rate is equal to n x PLLICLK where n is user selectable to 2, 3, or 4, using the CLK_CFG1-0 inputs. These pins can also be used in combination with the $\overline{\text{CLKDBL}}$ pin to generate additional core clock rates of 6 x CLKIN and 8 x CLKIN (see Table 13-8 on page 13-29).

Table 13-1. Pin Descriptions (Cont'd)

Pin	Type	Function
$\overline{\text{CLKDBL}}$	I	**Crystal Double Mode Enable.** This pin is used to enable the 2x clock double circuitry, where CLKOUT can be configured as either 1x or 2x the rate of CLKIN. This CLKIN double circuit is primarily intended to be used for an external crystal in conjunction with the internal clock generator and the XTAL pin. The internal clock generator when used in conjunction with the XTAL pin and an external crystal is designed to support up to a maximum of 25 MHz external crystal frequency. $\overline{\text{CLKDBL}}$ can be used in XTAL mode to generate a 50 MHz input into the PLL. The 2x clock mode is enabled (during $\overline{\text{RESET}}$ low) by tying $\overline{\text{CLKDBL}}$ to GND, otherwise it is connected to V_{DDEXT} for 1x clock mode. For example, this allows the use of a 25 MHz crystal to enable 100MHz core clock rates and a 50 MHz CLKOUT operation when CLK_CFG1=0 and $\overline{\text{CLKDBL}}$=0. This pin can also be used to generate different clock rate ratios for external clock oscillators as well. The possible clock rate ratio options (up to 100 MHz) for either CLKIN (external clock oscillator) or XTAL (crystal input) are shown in Table 13-8 on page 13-29:
		Clock Rate Ratios. An 8:1 ratio allows the use of a 12.5 MHz crystal to generate a 100 MHz core (instruction clock) rate and a 25 MHz CLKIN (external port) clock rate. **Note:** When using an external crystal, the maximum crystal frequency cannot exceed 25 MHz. For all other external clock sources, the maximum CLKIN frequency is 50 MHz. For more information, see "Clock Rate Ratios" on page 13-29.

Table 13-1. Pin Descriptions (Cont'd)

Pin	Type	Function
CLKOUT	O/T	**Local Clock Out.** CLKOUT is 1x or 2x and is driven at either 1x or 2x the frequency of CLKIN frequency by the current bus master. The frequency is determined by the $\overline{\text{CLKDBL}}$ pin. The three programmable modes supported for CLKOUT by setting bit 22 (COD) and 23 (COPT) of the SYSCON register are: COPT COD CLKOUT Description 0 0 Free running 0 1 Disabled 1 x Driven by MMS master When the COPT bit is set, CLKOUT is driven by the master device. CLKOUT is three-stated during the bus transition cycle by the device giving up its bus master status. The new bus master then drives CLKOUT. During host accesses, the bus master that granted the bus to the host drives CLKOUT. A keeper latch on the processor's CLKOUT pin maintains the output at the level it was last driven. This latch is only enabled on processors with ID2-0=00x. If $\overline{\text{CLKDBL}}$ enabled, CLKOUT = 2xCLKIN period If $\overline{\text{CLKDBL}}$ disabled, CLKOUT = 1xCLKIN period **Note:** CLKOUT is only controlled by the $\overline{\text{CLKDBL}}$ pin and operates at either 1xCLKIN or 2xCLKIN. For more information, see "ADSP-21161 CLKOUT and CCLK Clock Generation Operation" on page 13-27.
$\overline{\text{CS}}$	I/A	**Chip Select.** Asserted by host processor to select the ADSP-21161 processor.

Table 13-1. Pin Descriptions (Cont'd)

Pin	Type	Function
DATA47-16	I/O/T	**External Bus Data.** The ADSP-21161 processor inputs and outputs data and instructions on these pins. Pull-up resistors on unused data pins are not necessary. A keeper latch on the processor's DATA47-16 pins maintains the input at the level it was last driven. This latch is only enabled on the processors with ID2-0=00x. **Note:** DATA[15:8] pins (multiplexed with L1DATA[7:0]) can also be used to extend the data bus if the link ports are disabled and not used. In addition, DATA[7:0] pins (multiplexed with L0DATA[7:0]) can also be used to extend the data bus if the link ports are not used. This allows execution of 48-bit instructions from external SBSRAM (system clock speed-external port), SRAM (system clock speed-external port) and SDRAM (core clock or one-half the core clock speed). The IPACKx Instruction Packing Mode Bits in SYSCON should be set correctly (IPACK1-0 = 0x1) to enable this full instruction Width/No-packing Mode of operation.
$\overline{\text{DMAR1}}$	I/A	**DMA Request 1** (DMA Channel 11). Asserted by external port devices to request DMA services. $\overline{\text{DMAR1}}$ has a 20kΩ internal pull-up resistor.
$\overline{\text{DMAR2}}$	I/A	**DMA Request 2** (DMA Channel 12). Asserted by external port devices to request DMA services. $\overline{\text{DMAR2}}$ has a 20kΩ internal pull-up resistor.
$\overline{\text{DMAG1}}$	O/T	**DMA Grant 1** (DMA Channel 11). Asserted by ADSP-21161 processor to indicate that the requested DMA starts on the next cycle. Driven by bus master only. $\overline{\text{DMAG1}}$ has a 20kΩ internal pull-up resistor that is enabled for processors with ID2-0=00x.
$\overline{\text{DMAG2}}$	O/T	**DMA Grant 2** (DMA Channel 12). Asserted by ADSP-21161 processor to indicate that the requested DMA starts on the next cycle. Driven by bus master only. $\overline{\text{DMAG2}}$ has a 20kΩ internal pull-up resistor that is enabled for processors with ID2-0=00x.
DQM	O/T	**SDRAM Data Mask.** In write mode, DQM has a latency of zero and is used during a precharge command and during SDRAM power-up initialization.

Table 13-1. Pin Descriptions (Cont'd)

Pin	Type	Function
DxA	I/O	**Data Transmit or Receive Channel A** (Serial Ports 0, 1, 2, 3). Each DxA pin has a 50kΩ internal pull-up resistor. Bidirectional data pin. This signal can be configured as an output to transmit serial data, or as an input to receive serial data.
DxB	I/O	**Data Transmit or Receive Channel B** (Serial Ports 0, 1, 2, 3). Each `DxB` pin has a 50kΩ internal pull-up resistor. Bidirectional data pin. This signal can be configured as an output to transmit serial data, or as an input to receive serial data.
EBOOT	I	**EPROM Boot Select**. For a description of how this pin operates, see the table in the $\overline{\text{BMS}}$ pin description. This signal is a system configuration selection that should be hardwired.
$\overline{\text{EMU}}$	O (O/D)	**Emulation Status**. Must be connected to the ADSP-21161 processor Analog Devices Tools product line of JTAG emulators target board connector only. $\overline{\text{EMU}}$ has a 50kΩ internal pullup resistor.
FLAG11-0	I/O/A	**Flag Pins**. Each is configured via control bits as either an input or output. As an input, it can be tested as a condition. As an output, it can be used to signal external peripherals.
FSx	I/O	**Transmit or Receive Frame Sync** (Serial Ports 0, 1, 2, 3). The frame sync pulse initiates shifting of serial data. This signal is either generated internally or externally. It can be active high or low or an early or a late frame sync, in reference to the shifting of serial data.
GND	G	**Power Supply Return** (26 pins).
$\overline{\text{HBR}}$	I/A	**Host Bus Request**. Must be asserted by a host processor to request control of the ADSP-21161 processor's external bus. When $\overline{\text{HBR}}$ is asserted in a multiprocessing system, the ADSP-21161 processor that is bus master relinquishes the bus and asserts $\overline{\text{HBG}}$. To relinquish the bus, the ADSP-21161 processor places the address, data, select, and strobe lines in a high impedance state. $\overline{\text{HBR}}$ has priority over all ADSP-21161 processor bus requests ($\overline{\text{BR6-1}}$) in a multiprocessing system.

Table 13-1. Pin Descriptions (Cont'd)

Pin	Type	Function
$\overline{\text{HBG}}$	I/O	**Host Bus Grant**. Acknowledges an $\overline{\text{HBR}}$ bus request, indicating that the host processor may take control of the external bus. $\overline{\text{HBG}}$ is asserted (held low) by the ADSP-21161 processor until $\overline{\text{HBR}}$ is released. In a multiprocessing system, $\overline{\text{HBG}}$ is output by the ADSP-21161 processor bus master and is monitored by all others. After $\overline{\text{HBR}}$ is asserted, and before $\overline{\text{HBG}}$ is given, $\overline{\text{HBG}}$ floats for 1 t_{CK} (1 CLKIN cycle). To avoid erroneous grants, $\overline{\text{HBG}}$ should be pulled up with a 20kΩ to 50kΩ ohm external resistor.
$\overline{\text{IRQ}}$2-0	I/A	**Interrupt Request Lines.** These are sampled on the rising edge of CLKIN and may be either edge-triggered or level-sensitive.
ID2-0	I	**Multiprocessing ID**. Determines which multiprocessing bus request ($\overline{\text{BR}}$1 - $\overline{\text{BR}}$6) is used by ADSP-21161 processor. ID2-0 = 001 corresponds to BR1, ID2-0 = 010 corresponds to BR2, and so on. Use ID2-0 = 000 or ID2-0 = 001 in single-processor systems. These lines are a system configuration selection that should be hardwired or only changed at reset.
LxDAT7-0 [DAT15-0]	I/O [I/O/T]	**Link Port Data** (Link Ports 0-1). Each LxDAT pin has a 20kΩ internal pull-down resistor that is enabled or disabled by the LxPDRDE bit of the LCTL register or a keeper latch when used as DATA pins. **Note:** L1DATA[7:0] are multiplexed with the DATA[15:8] pins L0DATA[7:0] are multiplexed with the DATA[7:0] pins. If link ports are disabled and are not be used, then these pins can be used as additional data lines for executing instructions at up to the full clock rate from external memory. For revisions 0.3, 1.0 and 1.1, LxDAT7-0 has a 50kΩ internal pull-down resistor that is enabled or disabled by the LxPDRDE bit of the LCTL register
LxCLK	I/O	**Link Port Clock** (Link Ports 0-1). Each LxCLK pin has a 50kΩ internal pull-down resistor that is enabled or disabled by the LxPDRDE bit of the LCTL register.

Table 13-1. Pin Descriptions (Cont'd)

Pin	Type	Function
LxACK	I/O	**Link Port Acknowledge** (Link Ports 0-1). Each LxACK pin has a 50kΩ internal pull-down resistor that is enabled or disabled by the LxPDRDE bit of the LCTL register.
LBOOT	I	**Link Boot**. For a description of how this pin operates, see the table in the $\overline{\text{BMS}}$ pin description. This signal is a system configuration selection that should be hardwired.
MOSI	I/O	**SPI Master Out Slave In.** If the ADSP-21161 processor is configured as a master, the MOSI pin becomes a data transmit (output) pin, transmitting output data. If the ADSP-21161 processor is configured as a slave, the MOSI pin becomes a data receive (input) pin, receiving input data. In an ADSP-21161 processor SPI interconnection, the data is shifted out from the MOSI output pin of the master and shifted into the MOSI input(s) of the slave(s). MOSI has a 50kΩ internal pull-up resistor.
MISO	I/O	**SPI Master In Slave Out.** If the ADSP-21161 processor is configured as a master, the MISO pin becomes a data receive (input) pin, receiving input data. If the ADSP-21161 processor is configured as a slave, the MISO pin becomes a data transmit (output) pin, transmitting output data. In an ADSP-21161 processor SPI interconnection, the data is shifted out from the MISO output pin of the slave and shifted into the MISO input pin of the master. MISO has a 50kΩ internal pull-up resistor. **Note:** Only one master is allowed to transmit data at any given time.
$\overline{\text{MS3-0}}$	I/O/T	**Memory Select Lines**. These outputs are asserted (low) as chip selects for the corresponding banks of external memory. Memory bank sizes are fixed to 16 Mwords for non-SDRAM and 64 Mwords for SDRAM. The $\overline{\text{MS3-0}}$ outputs are decoded memory address lines. In asynchronous access mode, the $\overline{\text{MS3-0}}$ outputs transition with the other address outputs. In synchronous access modes, the $\overline{\text{MS3-0}}$ outputs assert with the other address lines; however, they de-assert after the first CLKIN cycle in which ACK is sampled asserted. In a multiprocessor systems, the MSx signals are tracked by slave SHARCs. $\overline{\text{MS3-0}}$ has a keeper latched enabled for processor's with ID2-0=00x.

Table 13-1. Pin Descriptions (Cont'd)

Pin	Type	Function
NC		**Do Not Connect**. Reserved pins that must be left open and unconnected. (5 pins).
$\overline{\text{PA}}$	I/O/T	**Priority Access**. Asserting its $\overline{\text{PA}}$ pin allows an ADSP-21161 processor bus slave to interrupt background DMA transfers and gain access to the external bus. PA is connected to all ADSP-21161 processors in the system. If access priority is not required in a system, the $\overline{\text{PA}}$ pin should be left unconnected. $\overline{\text{PA}}$ has a 20kΩ internal pull-up resistor that is enabled for processors with ID2-0=00x.
$\overline{\text{RAS}}$	I/O/T	**SDRAM Row Access Strobe.** In conjunction with $\overline{\text{CAS}}$, $\overline{\text{MSx}}$, $\overline{\text{SDWE}}$, $\overline{\text{SDCLKx}}$, and sometimes SDA10, defines the operation for the SDRAM to perform.
$\overline{\text{RD}}$	I/O/T	**Memory Read Strobe.** $\overline{\text{RD}}$ is asserted whenever ADSP-21161 processor reads a word from external memory or from the IOP registers of other ADSP-21161 processors. External devices, including other ADSP-21161 processors, must assert $\overline{\text{RD}}$ for reading from a word of the ADSP-21161 processor IOP register memory. In a multiprocessing system, $\overline{\text{RD}}$ is driven by the bus master. $\overline{\text{RD}}$ has a 20kΩ internal pull-up resistor that is enabled for processors with ID2-0=00x.
REDY	O (O/D)	**Host Bus Acknowledge**. The ADSP-21161 processor de-asserts $\overline{\text{REDY}}$ (low) to add waitstates to a host access of its IOP registers when $\overline{\text{CS}}$ and $\overline{\text{HBR}}$ inputs are asserted.
$\overline{\text{RESET}}$	I/A	**Processor Reset**. Resets the ADSP-21161 processor to a known state and begins execution at the program memory location specified by the hardware reset vector address. The $\overline{\text{RESET}}$ input must be asserted (low) at power-up.
RPBA	I/S	**Rotating Priority Bus Arbitration Select**. When RPBA is high, rotating priority for multiprocessor bus arbitration is selected. When RPBA is low, fixed priority is selected. This signal is a system configuration selection that must be set to the same value on every ADSP-21161 processor. If the value of RPBA is changed during system operation, it must be changed in the same CLKIN cycle on every ADSP-21161 processor.

Table 13-1. Pin Descriptions (Cont'd)

Pin	Type	Function
$\overline{\text{RSTOUT}}$[1]	O	**Reset Out**. When $\overline{\text{RSTOUT}}$ is asserted, this pin is used to indicate to the external logic that the core blocks are in reset. It is deasserted 4096 cycles after $\overline{\text{RESET}}$ is deasserted allowing the PLL to stabilize and lock. For systems requiring a secondary reset for other devices needing to be simultaneously brought out of reset with the processor core reset, system designers can connect this pin to the reset pin of the other devices. This prevents other devices from driving data before the processor begins the booting process.
$\overline{\text{SDWE}}$	I/O/T	**SDRAM Write Enable.** In conjunction with $\overline{\text{CAS}}$, $\overline{\text{RAS}}$, $\overline{\text{MSx}}$, $\overline{\text{SDWE}}$, $\overline{\text{SDCLKx}}$, and sometimes SDA10, defines the operation for the SDRAM to perform.
SDCLK0	I/O/S/T	**SDRAM Clock Output 0.** Clock for SDRAM devices.
SDCLK1	O/S/T	**SDRAM Clock Output 1.** Additional clock for SDRAM devices. For systems with multiple SDRAM devices, handles the increased clock load requirements, eliminating need of off-chip clock buffers. Either SDCLK1 or both SDCLKx pins can be three-stated.
SDCKE	I/O/T	**SDRAM Clock Enable.** Enables and disables the CLK signal. For details, see the data sheet supplied with your SDRAM device.
SDA10	O/T	**SDRAM A10 Pin.** Enables applications to refresh an SDRAM in parallel with a non-SDRAM accesses or host accesses.
$\overline{\text{SBTS}}$	I/S	**Suspend Bus Three-State**. External devices can assert $\overline{\text{SBTS}}$ (low) to place the external bus address, data, selects, and strobes in a high impedance state for the following cycle. If the ADSP-21161processor attempts to access external memory while $\overline{\text{SBTS}}$ is asserted, the processor halts and the memory access is not completed until $\overline{\text{SBTS}}$ is de-asserted. $\overline{\text{SBTS}}$ should only be used to recover from host processor/ADSP-21161 processor deadlock.
SCLKx	I/O	**Transmit/Receive Serial Clock** (Serial Ports 0, 1, 2, 3). Each SCLK pin has a 50kΩ internal pull-up resistor. This signal can be either internally or externally generated.

Table 13-1. Pin Descriptions (Cont'd)

Pin	Type	Function
SPICLK	I/O	**Serial Peripheral Interface Clock Signal**. Driven by the master, controls the rate at which data is transferred. The master may transmit data at a variety of baud rates. SPICLK cycles once for each bit transmitted. SPICLK is a gated clock that is active during data transfers, only for the length of the transferred word. Slave devices ignore the serial clock if the slave select input is driven inactive (HIGH). SPICLK is used to shift out and shift in the data driven on the MISO and MOSI lines. The data is always shifted out on one clock edge of the clock and sampled on the opposite edge of the clock. Clock polarity and clock phase relative to data are programmable into the SPICTL control register and define the transfer format. SPICLK has a 50kΩ internal pull-up resistor.
$\overline{\text{SPIDS}}$	I	**Serial Peripheral Interface Slave Device Select.** An active low signal used to enable slave devices. This input signal behaves like a chip select, and is provided by the master device for the slave devices. In multi-master mode $\overline{\text{SPIDS}}$ signal can be asserted to a master device to signal that an error has occurred, as some other device is also trying to be the master device. If asserted low when the device is in master mode, it is considered a multi-master error. For a Single-Master, Multiple-Slave configuration where FLAG3-0 are used, this pin must be tied high to V_{DDINT}. For ADSP-21161 processor to ADSP-21161 processor SPI interaction, any of the master ADSP-21161 processor's FLAG3-0 pins can be used to drive the $\overline{\text{SPIDS}}$ signal on the ADSP-21161 processor SPI slave device.
TIMEXP	O	**Timer Expired**. Asserted for four core clock cycles when the timer is enabled and TCOUNT decrements to zero.
TCK	I	**Test Clock** (**JTAG**). Provides a clock for JTAG boundary scan.
TMS	I/S	**Test Mode Select** (**JTAG**). Used to control the test state machine. TMS has a 20kΩ internal pull-up resistor.
TDI	I/S	**Test Data Input** (**JTAG**). Provides serial data for the boundary scan logic. TDI has a 20kΩ internal pull-up resistor.
TDO	O	**Test Data Output** (**JTAG**). Serial scan output of the boundary scan path.

Table 13-1. Pin Descriptions (Cont'd)

Pin	Type	Function
$\overline{\text{TRST}}$	I/A	**Test Reset (JTAG).** Resets the test state machine. $\overline{\text{TRST}}$ must be asserted (pulsed low) after power-up or held low for proper operation of the ADSP-21161 processor. $\overline{\text{TRST}}$ has a 20kΩ internal pull-up resistor.
VDDINT	P	**Core Power Supply.** Nominally +1.8 V DC and supplies the processor's core processor (14 pins).
VDDEXT	P	**I/O Power Supply.** Nominally +3.3 V DC. (13 pins).
$\overline{\text{WR}}$	I/O/T	**Memory Write Low Strobe.** $\overline{\text{WR}}$ is asserted when ADSP-21161 processor writes a word to external memory or IOP registers of other ADSP-21161 processors. External devices must assert WR for writing to ADSP-21161 processor's IOP registers. In a multiprocessing system, WR is driven by the bus master. $\overline{\text{WR}}$ has a 20kΩ internal pull-up resistor that is enabled for processors with ID2-0=00x.
XTAL	O	**Crystal Oscillator Terminal 2.** Used in conjunction with CLKIN to enable the ADSP-21161 processor's internal clock generator or to disable it to use an external clock source. See CLKIN.

1 $\overline{\text{RSTOUT}}$ exists only for silicon revisions 1.2 and greater.

Inputs identified as synchronous (S) must meet timing requirements with respect to CLKIN (or with respect to TCK for TMS, TDI). Inputs identified as asynchronous (A) can be asserted asynchronously to CLKIN (or to TCK for $\overline{\text{TRST}}$).

Unused inputs should be tied or pulled to VDDEXT or GND, except for ADDR23-0, DATA47-16, FLAG11-0, and inputs that have internal pull-up or pull-down resistors ($\overline{\text{PA}}$, ACK, BRST, CLKOUT, $\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{DMAR}}$x, $\overline{\text{DMAG}}$x, DxA, DxB, SCLKx, LxDAT7-0, MISO, MOSI, SPICLK, LxCLK, LxACK, TMS, $\overline{\text{TRST}}$ and TDI)—these pins can be left floating. Some of these pins have a logic-level hold circuit (only enabled on the ADSP-21161 processor with ID2-0=00x) that prevents input from floating internally. See the pin list in Table 13-1.

The $\overline{\text{TRST}}$ input of the JTAG interface must be asserted (pulsed low) or held low after power-up for proper operation of the ADSP-21161 processor. Do not leave this pin unconnected.

Additional Notes:

- In single-processor systems, the processor owns the external bus during reset and does not perform bus arbitration to gain control of the bus.
- Operation of the $\overline{\text{RD}}$ and $\overline{\text{WR}}$ signals changes when $\overline{\text{CS}}$ is asserted by a host processor. For more information, see "Asynchronous Transfers" on page 7-48.
- Except during a Host Transition Cycle (HTC), the $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes should not be deasserted (low-to-high transition) while ACK or $\overline{\text{REDY}}$ are deasserted (low)—the processor hangs if this happens.
- In multiprocessor systems, the ACK signal is an input to the ADSP-21161 processor bus master and does not float when it is not being driven. It is not necessary to use an external pullup resistor on the ACK line during booting or at any other time. The ACK pin is pulled high internally with a 20kΩ equivalent resistor and is activated under the following three conditions:
    1. When the processor is in reset (regardless of the hardwired ID pin configuration)
    2. After reset, in a single processor system (ID2-0 =000)
    3. After reset, in a multiprocessor system, the processor having ID2-0 =001

Figure 13-2 shows how different data word sizes are transferred over the external port.

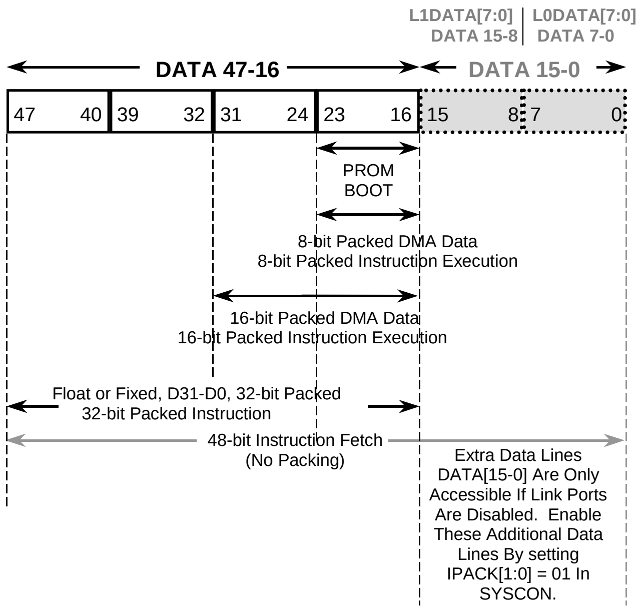

L1DATA[7:0]
DATA 15-8
L0DATA[7:0]
DATA 7-0
DATA 47-16
DATA 15-0
47
40
39
32
31
24
23
16
15
8
7
0
PROM
BOOT
8-bit Packed DMA Data
8-bit Packed Instruction Execution
16-bit Packed DMA Data
16-bit Packed Instruction Execution
Float or Fixed, D31-D0, 32-bit Packed
32-bit Packed Instruction
48-bit Instruction Fetch
(No Packing)
Extra Data Lines DATA[15-0] Are Only Accessible If Link Ports Are Disabled. Enable These Additional Data Lines By setting IPACK[1:0] = 01 In SYSCON.

Figure 13-2. External Port Data Alignment

Input Synchronization Delay

The ADSP-21161 processor has several asynchronous inputs: $\overline{\texttt{RESET}}$, $\overline{\texttt{TRST}}$, $\overline{\texttt{HBR}}$, $\overline{\texttt{CS}}$, $\overline{\texttt{DMAR1}}$, $\overline{\texttt{DMAR2}}$, $\overline{\texttt{IRQ2-0}}$, and FLAG11-0 (when configured as inputs). These inputs can be asserted in arbitrary phase to the processor clock, CLKIN. The processor synchronizes the inputs prior to recognizing them. The delay associated with recognition is called the synchronization delay.

Any asynchronous input must be valid prior to the recognition point in a particular cycle. If an input does not meet the setup time on a given cycle, it may be recognized in the current cycle or during the next cycle.

To ensure recognition of an asynchronous input, it must be asserted for at least one full processor cycle plus setup and hold time, except for $\overline{\text{RESET}}$, which must be asserted for at least four processor cycles. The minimum time prior to recognition (the setup and hold time) is specified in the *ADSP-21161N SHARC DSP Microcomputer Data Sheet*.

Pin States At Reset

Table 13-2 shows the processor pin states during and after reset.

Table 13-2. Pin States at Reset

Pin	Type	State During and After Reset
ACK	I/O/S/T	Pulled high by bus master (w/ 20kΩ internal pullup resistor)[1]
ADDR23-0	I/O/T	Driven[1]
$\overline{\text{BMS}}$	I/O/T	Input[2]
BMSTR	O	Driven high for ID1, driven low for ID2-6
$\overline{\text{BR6-1}}$	I/O	BR1 driven low if bus master, otherwise driven high[1]
BRST	I/O/T	Driven low[1]
$\overline{\text{CAS}}$	I/O/T	Driven high
CLK_CFG1-0	I	Input
$\overline{\text{CLKDBL}}$	I	Input
CLKIN	I	Input
CLKOUT	O/T	Driven
$\overline{\text{CS}}$	I	Input[2]
DATA47-16	I/O/T	Three-state[1]
$\overline{\text{DMAG1}}$	O/T	Driven high[1]
$\overline{\text{DMAG2}}$	O/T	Driven high[1]

Table 13-2. Pin States at Reset (Cont'd)

Pin	Type	State During and After Reset
$\overline{\text{DMAR1}}$	I	Input[2]
$\overline{\text{DMAR2}}$	I	Input[2]
DQM	I/O/T	Driven high until SDRAM power-up sequence starts
DxA	O	Three-state (for multichannel)
DxB	I	Input[4]
EBOOT	I	Input[2]
$\overline{\text{EMU}}$	O (o/d)	Three-state[3]
FLAG11-0	I/O/A	Input[2]
FSx	I/O	Three-state[4]
$\overline{\text{HBG}}$	I/O/S/T	Driven high[1]
$\overline{\text{HBR}}$	I/A	Input[2]
ID2-0	I	Input[2]
$\overline{\text{IRQ2-0}}$	I/A	Input[2]
LBOOT	I	Input[2]
LxACK	I/O	Three-state[4]
LxCLK	I/O	Three-state[4]
LxDAT7-0	I/O	Three-state[4]
MISO	I/O	Input[4]
MOSI	I/O	Input[4]
$\overline{\text{MS3-0}}$	I/O/T	Driven high[1]
$\overline{\text{PA}}$ (o/d)	I/O	Three-state[2]
$\overline{\text{RAS}}$	I/O/T	Driven high
$\overline{\text{RD}}$	I/O/T	Driven high[1]
REDY (o/d)	O	Three-state[2]

Table 13-2. Pin States at Reset (Cont'd)

Pin	Type	State During and After Reset
$\overline{\text{RESET}}$	I/A	Input[2]
$\overline{\text{RPBA}}$	I/S	Input[2]
$\overline{\text{RSTOUT}}$	O	Driven low[2]
$\overline{\text{SBTS}}$	I/S	Input
SCLK	I/O	Three-state[4]
SDA10	O/T	Driven
SDCKE	I/O/T	Driven high
SDCLK0	I/O/S/T	Driven
SDCLK1	O/S/T	Driven
$\overline{\text{SDWE}}$	I/O/T	Driven high
SPICLK	I/O	Three-state[4]
$\overline{\text{SPIDS}}$	I	Input[4]
TCK	I	Input[3]
TDI	I/S	Input[3]
TDO	O	Three-state[3]
TIMEXP	O	Driven low[2]
TMS	I/S	Input[3]
$\overline{\text{TRST}}$	I/A	Input[3]
$\overline{\text{WR}}$	I/O/T	Driven high[1]
XTAL	O/T	Driven

1 For ID =0 or 1, driven only by processor bus master, otherwise three-stated
2 Bus master independent
3 JTAG interface
4 Serial ports, SPI and link port

Pull-Up and Pull-Down Resistors

Table 13-3 shows the keeper latches, pull-up and pull-down resistor values associated with each pin.

Table 13-3. Keeper Latches and Resistor Values

Pin	Resistor va1ue
ACK	20kΩ pull-up enabled during reset or when ID2-0 = 00X
ADDR23-0	Keeper latch (only for ID2-0 = 00X)
$\overline{\text{BMS}}$	N/A
BMSTR	N/A
$\overline{\text{BR6-1}}$	N/A
BRST	Keeper latch (only for ID2-0 = 00X)
$\overline{\text{CAS}}$	N/A
CLK_CFG1-0	N/A
$\overline{\text{CLKDBL}}$	N/A
CLKIN	N/A
CLKOUT	Keeper latch (only for ID2-0 = 00X)
CLKx	N/A
$\overline{\text{CS}}$	N/A
DATA47-16	Keeper latch (only for ID2-0 = 00X)
$\overline{\text{DMAG1}}$	20kΩ Pull-up (only for ID2-0 = 00X)
$\overline{\text{DMAG2}}$	20kΩ Pull-up (only for ID2-0 = 00X)
$\overline{\text{DMAR1}}$	20kΩ Pull-up (only for ID2-0 = 00X)
$\overline{\text{DMAR2}}$	20kΩ Pull-up (only for ID2-0 = 00X)
DQM	N/A
DxA	50kΩ Pull-up
DxB	50kΩ Pull-up

Table 13-3. Keeper Latches and Resistor Values (Cont'd)

Pin	Resistor va1ue
EBOOT	N/A
$\overline{\text{EMU}}$	50kΩ Pull-up
FLAG11-0	N/A
FSx	N/A
$\overline{\text{HBG}}$	N/A
$\overline{\text{HBR}}$	N/A
ID2-0	N/A
$\overline{\text{IRQ2-0}}$	N/A
LBOOT	N/A
LxACK	50kΩ Pull-down that is enabled or disabled by the LxPDRDE bit of the LCTL register.
LxCLK	50kΩ Pull-down that is enabled or disabled by the LxPDRDE bit of the LCTL register.
LxDAT7-0	For Revisions 0.3. 1.0, 1.1, 50kΩ Pull-down For Revisions 1.2 and higher, a keeper latch is enabled when these pins are used as DATA lines or a 20kΩ Pull-down resistor is enabled or disabled based on the LxPDRDE bit setting.
MISO	50kΩ Pull-up
MOSI	50kΩ Pull-up
$\overline{\text{MS3-0}}$	Keeper latch (Only for ID2-0 = 00X)
$\overline{\text{PA}}$ (o/d)	20kΩ Pull-up (only for ID2-0 = 00X)
$\overline{\text{RAS}}$	N/A
$\overline{\text{RD}}$	20kΩ Pull-up (only for ID2-0 = 00X)
REDY (o/d)	N/A
$\overline{\text{RESET}}$	N/A
$\overline{\text{RPBA}}$	N/A

Table 13-3. Keeper Latches and Resistor Values (Cont'd)

Pin	Resistor va1ue
$\overline{\text{SBTS}}$	N/A
SCLK	50kΩ Pull-up
SDA10	N/A
SDCKE	N/A
SDCLK0	N/A
SDCLK1	N/A
$\overline{\text{SDWE}}$	N/A
SPICLK	50kΩ Pull-up
$\overline{\text{SPIDS}}$	N/A
TCK	N/A
TDI	20kΩ Pull-up
TDO	N/A
TIMEXP	N/A
TMS	20kΩ Pull-up
$\overline{\text{TRST}}$	20kΩ Pull-up
$\overline{\text{WR}}$	20kΩ Pull-up (only for ID2-0 = 00X)
XTAL	N/A

Clock Derivation

The ADSP-21161 processor employs a phase-locked loop on-chip, to provide clocks that switch at higher frequencies than the system clock (`CLKIN`). The PLL-based clocking methodology employed on the processor influences the clock frequencies and behavior for the serial, link, SDRAM, SPI, and external ports; in addition to the processor core and internal memory. In each case, the processor PLL provides a de-skewed clock to the port logic and I/O pins.

For the external port, this clock is fedback to the PLL, such that the external port clock always switches at the 1x or 2x frequency CLKIN frequency depending on if $\overline{\text{CLKDBL}}$ is enabled. The PLL provides internal clocks that switch at multiples of the CLKIN frequency for the internal memory, processor core, link and serial ports, and to modify the external port timing as required (for example, read/write strobes in asynchronous modes). The ratio of processor core clock frequency and CLKIN/external port clock frequency is determined by the CLK_CFG1-0 pins and $\overline{\text{CLKDBL}}$ pin (as shown in Table 13-8 on page 13-29), during reset.

The core clock ratio cannot be altered dynamically. The ADSP-21161 processor must be reset to alter the clock ratio.

The PLL provides a clock that switches at the processor core frequency to the serial and link ports. Each of the serial and link ports can be programmed to operate at clock frequencies derived from this clock. The four serial ports' transmit and receive clocks are divided down from the processor core clock frequency by setting the DIVx registers appropriately.

In addition to the PLL ratios generated by the CLK_CFG1-0 pins, an additional $\overline{\text{CLKDBL}}$ pin can be used for additional clock ratio options. The (1x/2x CLKIN) rate set by the $\overline{\text{CLKDBL}}$ pin determines the rate of the PLL input clock and the rate at which the synchronous external port operates. With the combination of CLK_CFG[1:0] and $\overline{\text{CLKDBL}}$, ratios of 2:1, 3:1, 4:1, 6:1, and 8:1 between the core and CLKIN are supported.

Timing Specifications

The ADSP-21161 processor's internal clock (a multiple of CLKIN) provides the clock signal for timing internal memory, processor core, link ports, serial ports, SPI, SDRAM, and external port (as required for read/write strobes in asynchronous access mode). During reset, program the ratio between the ADSP-21161 processor's internal clock frequency and external (CLKIN) clock frequency with the CLK_CFG1-0 and $\overline{\text{CLKDBL}}$ pins. Even though the internal clock is the clock source for the external port, it behaves as described in the Clock Rate Ratio chart ($\overline{\text{CLKDBL}}$ pin description).

To determine switching frequencies for the serial and link ports, divide down the internal clock, using the programmable divider control of each port (DIVx for the serial ports and LxCLKD1-0 for the link ports). For the SPI port, the BAUDR bit in the SPICTL register controls the SPICLK baud rate based on the core clock frequency. Each of the two link port clock frequencies are determined by programming the LxCLKDx parameters in the LCTL registers. For more information, see "Link Port Buffer Control Register (LCTL)" on page A-92.

Note the following definitions of various clock periods that are a function of CLKIN and the appropriate ratio control.

Figure 13-3 allows Core-to-CLKIN ratios of 2:1, 3:1, 4:1, 6:1, and 8:1 with external oscillator or crystal.

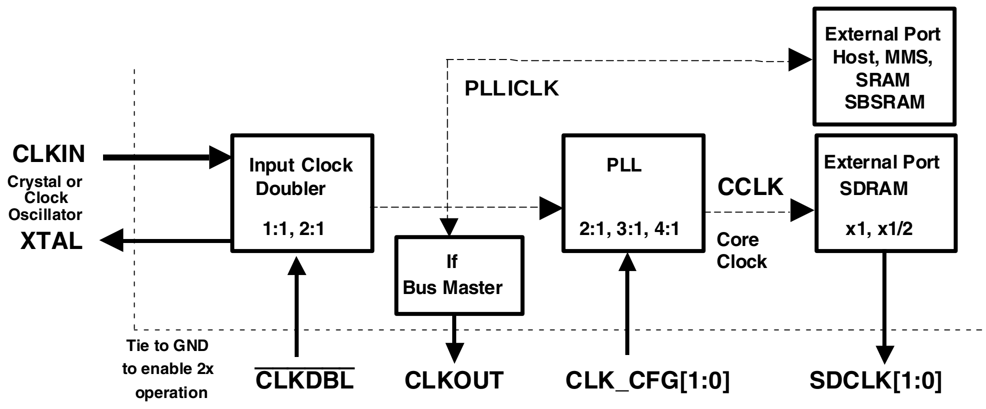


Figure 13-3. Core Clock and System Clock Relationship to CLKIN

Table 13-4 and Table 13-5 provide various definitions of clock inputs, outputs and uses in an ADSP-21161 processor system.

Table 13-4. ADSP-21161 CLKOUT and CCLK Clock Generation Operation

Timing Requirements		Calculation		Description
CLKIN	=	$1/t_{CKIN}$	=	Input Clock
CLKOUT	=	$1/t_{TCK}$	=	Local Clock Out
PLLICLK	=	$1/t_{PLLIN}$	=	PLL Input Clock
CCLK	=	$1/t_{CCLK}$	=	Core Clock

If $\overline{\texttt{CLKDBL}}$ is enabled (tied low at reset), then CLKOUT = PLLICLK = 2xCLKIN. Otherwise, CLKOUT = PLLICLK = CLKIN.

CCLK = Core Clock = PLLICLK x PLL Multiply Ratio (determined by CLK_CFG pins).

Table 13-5. Clock Relationships

Timing Requirements		Description[1]
t_{CK}	=	CLKOUT Clock Period
t_{PLLICK}	=	PLL Input Clock
t_{CCLK}	=	(Processor) Core Clock Period
t_{LCLK}	=	Link Port Clock Period = (t_{CCLK}) * LR
t_{SCLK}	=	Serial Port Clock Period = (t_{CCLK}) * SR
t_{SDK}	=	SDRAM Clock Period = (t_{CCLK}) * SDCKR
t_{SPICLK}	=	SPI Clock Period = (t_{CCLK}) * SPIR

1 where:
LR = link port-to-core clock ratio (1, 2, 3, or 1:4, determined by LxCLKD)
SR = serial port-to-core clock ratio (wide range, determined by CLKDIV)
SDCKR = SDRAM-to-Core Clock Ratio (1:1 or 1:2, determined by SDCTL register)
SPIR = SPI-to-Core Clock Ratio (wide range, determined by SPICTL register)
LCLK = Link Port Clock
SCLK = Serial Port Clock
SDK = SDRAM Clock
SPICLK = SPI Clock

Table 13-6 describes clock ratio requirements. Table 13-7 shows an example clock derivation.

Table 13-6. Clock Ratios

Timing Requirements		Description
c_{RTO}	=	Core:`CLKOUT` ratio, (2, 3, or 4:1, determined by `CLK_CFG`)
l_{RTO}	=	lport:core clock ratio (1:1, 1:2, 1:3, or 1:4, determined by `LxCLKD`)
s_{RTO}	=	Sport:core clock ratio (wide range determined by `xCLKDIV`)

Table 13-7. Determining Clock Period

Timing Requirements		Description
t_{CCLK}	=	(t_{CK}): `cRTO`
t_{LCLK}	=	(t_{CCLK}) * `lRTO`
t_{SCLK}	=	(t_{CCLK}) * `sRTO`

RESET and CLKIN

The ADSP-21161 processor receives its clock input on the `CLKIN` pin. The processor uses an on-chip phase-locked loop to generate its internal clock, which is a multiple of the `CLKIN` frequency. Because the phase-locked loop requires some time to achieve phase lock, `CLKIN` must be valid for a minimum time period during reset before the $\overline{\text{RESET}}$ signal can be deasserted. For information on minimum clock setup, see the *ADSP-21161N DSP Microcomputer Data Sheet*.

Table 13-8 describes the internal clock to CLKIN frequency ratios supported by the ADSP-21161 processor.

Table 13-8. Clock Rate Ratios

CLKDBL	CLK_CFG1	CLK_CFG0	Core Clock Ratio	CLKOUT Ratio
1	0	0	2:1	1x
1	0	1	3:1	1x
1	1	0	4:1	1x
0	0	0	4:1	2x
0	0	1	6:1	2x
0	1	0	8:1	2x

When using an external crystal, the maximum crystal frequency cannot exceed 25 MHz. The internal clock generator when used in conjunction with the XTAL pin and an external crystal is designed to support up to a maximum of 25 MHz external crystal frequency. For all other external clock sources, the maximum CLKIN frequency is 50 MHz.

Table 13-9 demonstrates the internal core clock switching frequency, across a range of CLKIN frequencies. The minimum operational range for any given frequency is constrained by the operating range of the phase lock loop. Note that the goal in selecting a particular clock ratio for the application is to provide the highest internal frequency, given a CLKIN frequency.

If an external master clock is used, it should not be driving the `CLKIN` pin when the processor is not powered. The clock must be driven immediately after powerup; otherwise, internal gates stay in an undefined (hot) state and can draw excess current. After powerup, there should be sufficient time for the oscillator to start up, reach full amplitude and deliver a stable `CLKIN` signal to the processor before the reset is released. This may take 100 ms depending on the choice of crystal, operating frequency, loop gain and capacitor ratios. For details on timing, refer to the *ADSP-21161N DSP Microcomputer Data Sheet*.

After the external $\overline{\texttt{RESET}}$ signal is deasserted, the PLL starts operating. The rest of the chip is held in reset for 4096 `CLKIN` cycles after $\overline{\texttt{RESET}}$ is deasserted by an internal (or core) reset ($\overline{\texttt{RSTOUT}}$[1]) signal. This sequence allows the PLL to lock and stabilize.

Table 13-9. Selecting Core to CLKIN Ratio

	Typical Crystal and Clock Oscillators Inputs					
	12.5	16.67	25	33.3	40	50
Clock Ratios	Core CLK (MHz)					
2:1	25	33.3	50	66.6	80	100
3:1	37.5	50	75	100	N/A	N/A
4:1	50	66.6	100	N/A	N/A	N/A
6:1	75	100	N/A	N/A	N/A	N/A

[1] $\overline{\texttt{RSTOUT}}$ exists only for silicon revisions 1.2 and greater.

Table 13-9. Selecting Core to CLKIN Ratio

	Typical Crystal and Clock Oscillators Inputs					
8:1	100	N/A	N/A	N/A	N/A	N/A

Reset Generators

It is important that an ADSP-21161 processor (or programmable device) have a reliable active $\overline{\text{RESET}}$ that is released once the power supplies and internal clock circuits have stabilized. The $\overline{\text{RESET}}$ signal should not only offer a suitable delay, but it should also have a clean monotonic edge. Analog Devices has a range of microprocessor supervisory ICs with different features. Features include one or more of the following:

- Powerup reset
- Optional manual reset input
- Power low monitor
- Back-up battery switching

Part number series for Analog Devices' supervisory circuits are as follows:

- ADM69x
- ADM70x
- ADM80x
- ADM1232
- ADM181x
- ADM869x

A simple powerup reset circuit is shown in Figure 13-4, using the ADM809-RART reset generator. The ADM809 provides an active low $\overline{\text{RESET}}$ signal whenever the supply voltage is below 2.63V. At powerup, a 240ms active reset delay is generated to give the power supplies and oscillators time to stabilize.

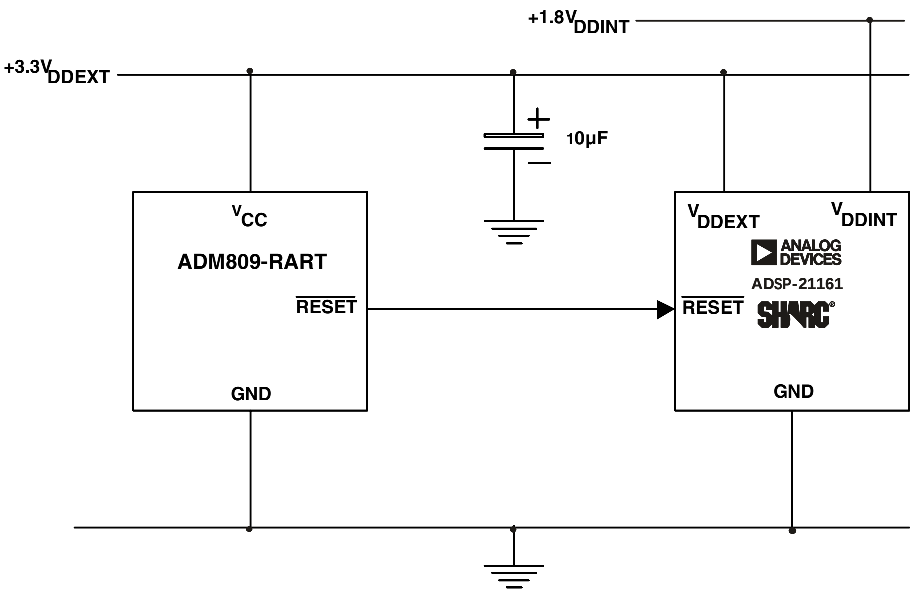


Figure 13-4. Simple Reset Generator

Another part, the ADM706TAR, provides power on $\overline{\text{RESET}}$ and optional manual $\overline{\text{RESET}}$. It allows designers to create a more complete supervisory circuit that monitors the supply voltage. Monitoring the supply voltage allows the system to initiate an orderly shutdown in the event of power failure. The ADM706TAR also allows designers to create a watchdog timer that monitors for software failure. This part is available in an eight lead SOIC package. Figure 13-5 shows a typical application circuit using the ADM706TAR.

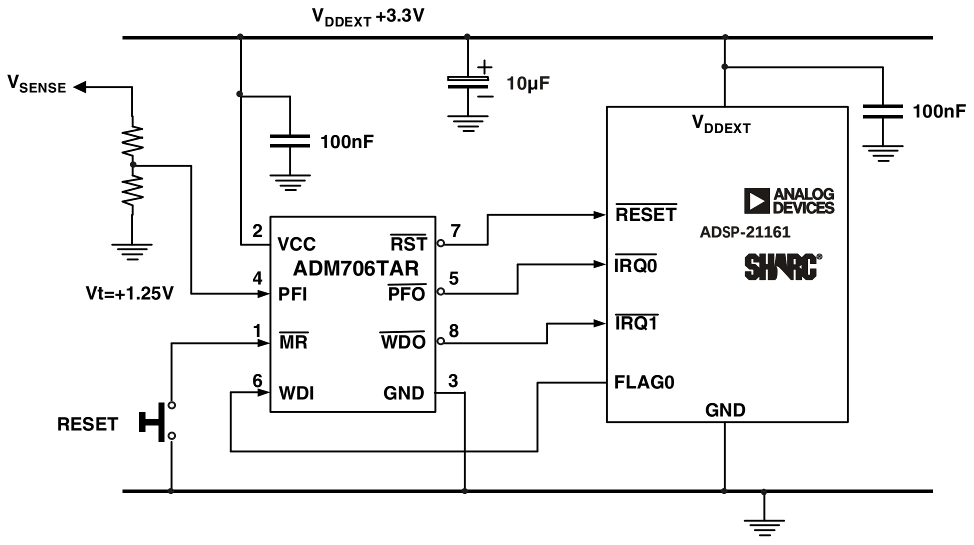


Figure 13-5. Reset Generator and Power Supply Monitor

Interrupt and Timer Pins

The ADSP-21161 processor's external interrupt pins, flag pins, and timer pin can be used to send and receive control signals to and from other devices in the system. Hardware interrupt signals are received on the $\overline{\text{IRQ2-0}}$ pins. Interrupts can come from devices that require the processor to perform some task on demand. A memory-mapped peripheral, for example, can use an interrupt to alert the processor that it has data available. For more information, see "Interrupts and Sequencing" on page 3-34.

The TIMEXP output is generated by the on-chip timer. It indicates to other devices that the programmed time period has expired. For more information, see "Timer and Sequencing" on page 3-50.

Core-Based Flag Pins

The FLAG3-0 pins allow single-bit signalling between the processor and other devices. For example, the ADSP-21161 processor can raise an output flag to interrupt a host processor. Each flag pin can be programmed to be either an input or output. In addition, many instructions can be conditioned on a flag's input value, enabling efficient communication and synchronization between multiple processors or other interfaces.

The flags are bidirectional pins, each with the same functionality. The FLGx0 bits in the MODE2 register program the direction of each flag pin. For more information, see "Mode Control 2 Register (MODE2)" on page A-10.

Flag Inputs

When a flag pin is programmed as an input, its value is stored in a bit in the FLAGS register. The bit is updated in each cycle with the input value from the pin. Flag inputs can be asynchronous to the processor clock, so there is a one-cycle delay before a change on the pin appears in FLAGS (if the rising edge of the input misses the setup requirement for that cycle). For more information, see "Flag Value Register (FLAGS)" on page A-37.

An flag bit is read-only if the flag is configured as an input. Otherwise, the bit is readable and writable. The flag bit states are conditions that code can specify in conditional instructions.

Flag Outputs

When a flag is configured as an output, the value on the pin follows the value of the corresponding bit in the FLAGS register. A program can set or clear the flag bit to provide a signal to another processor or peripheral.

The FLAG outputs transition on rising edge of CLKIN. Because the processor core operates at least twice the frequency of CLKIN, the programmer must hold the FLAG state stable for at least one full CLKIN period, to insure that the output changes state. Figure 13-6 describes the relationship between instruction execution and a Flag pin, when the processor core to bus clock ratio is set to 2:1. Figure 13-6 also describes the flag in/out process. Note that at least two instructions execute each `CLKIN` cycle.

```
BIT SET MODE2 FLG0;      /* 1st cycle: set FLAG0 to output in Mode2 */
BIT CLR FLAGS FLG0;      /* clear FLAG0 */
BIT SET FLAGS FLG0;      /* 1st cycle: set FLAG0 output high */
NOP;                     /* 2nd cycle: FLAG register updated here */
                         /* A NOP indicates a NOP or another instruction not related to FLAG. */
BIT CLR FLAGS FLG0;      /* 2nd cycle: clear FLAG0 output */
                         /* earliest assertion of FLAG0 output, depends on CLKOUT phase */
BIT CLR MODE2 FLG0;      /* 3rd cycle: set FLAG0 back to input */
NOP;                     /* 3rd cycle: */
NOP;                     /*4th cycle: earliest deassertion of FLAG0 output */
```

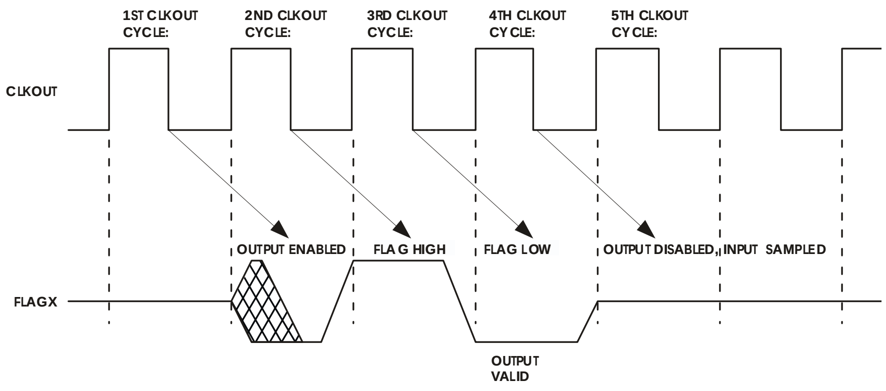


Figure 13-6. Flag Timing (At 2:1 Clock Ratio)

Programmable I/O Flags

The `IOFLAG` register is an IOP register created specifically for controlling the input/output flag pins. When a flag is configured as an output, the value on the pin follows the value of the corresponding bit in the `FLAGS` register. A program can set or clear the flag bit to provide a signal to another processor or peripheral. Some examples of assembly language instructions that demonstrate the use of macros to configure flag pins have been included at the end of this section.

The ADSP-21161 processor has an additional eight IOP based general-purpose programmable input/output flag pins - FLAG[11:4]. As outputs, these pins can signal peripheral devices; as inputs, these pins can provide the test for conditional branching. These pins correspond to the FLAG11-4 pins listed in the datasheet of the device.

All FLAG pins are configured as inputs on reset. When configuring IOFLAG register flag pins as outputs, do not set FLGx bits 0 to 7 in the same instruction cycle that the flag is configured as an output (setting the FLGxO bits 8 to 15 in the IOFLAG register). If your application requires that the flags be set after they are configured as outputs, two writes to the IOFLAG register are needed: one to configure the flag pin as an output, and another to set the flag pin high.

The functionality of the FLAG11-4 pins is similar to that of the FLAG3-0 except for both the status and control information are included in one register, IOFLAG. The control and status bits for the FLAG3-0 are in the MODE2 register and FLAGS register, respectively. Bits 0-7 of IOFLAG reflect the status of the FLAG pins while bits 15-8 control the direction (input or output) of these flags. A value of 0 programs the flag as an input and a value of 1 programs it as an output. Although you cannot execute bit wise operations such as BIT TST, BIT CLR, on these flags directly in memory, you can execute these operations by first writing to a system register such as the USTAT1 - USTAT4.

Figure 13-7 shows the IOFLAG register.

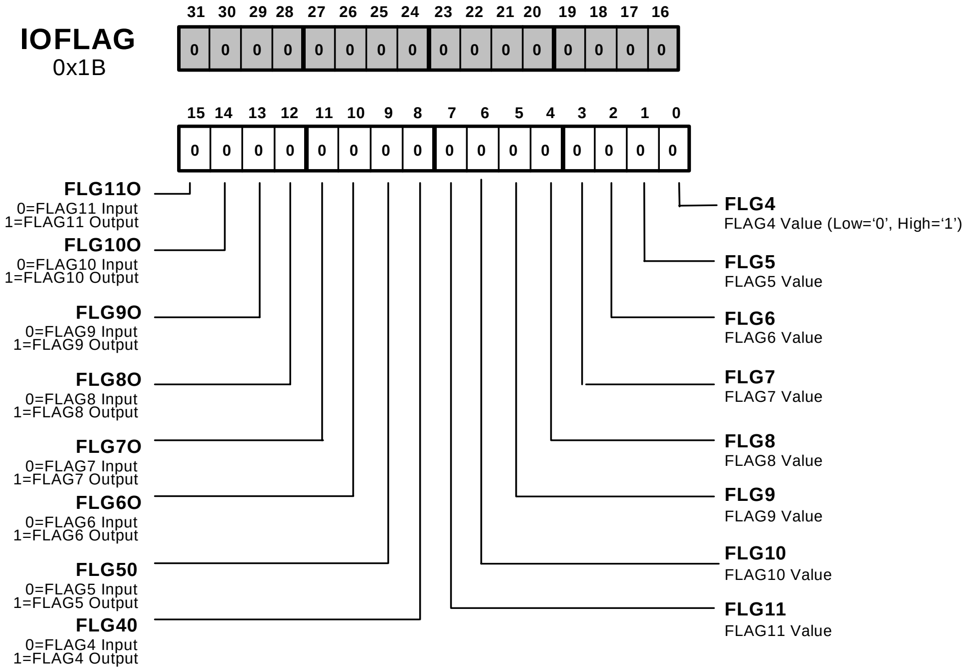


Figure 13-7. IOFLAG Register

Example #1: Configuring FLGx as Output Flags

The following example shows how to configure the flags as output flags, set the flag pins high and write the bits to the `IOFLAG` register:

```
ustat2 = 0x00000000;
bit set ustat2 FLG9O|FLG8O|FLG7O|FLG6O|FLG5O|FLG4O;
dm(IOFLAG) = ustat2;
```

After writing to the register, the flags can be toggled with the `bit tgl` command:

```
bit tgl ustat2 FLG9|FLG8|FLG7|FLG6|FLG5|FLG4;
dm(IOFLAG) = ustat2;
```

Example #2: Configuring FLGx as Input Flags

The following example shows how to configure the flags as input flags, clear the flag pins, and write the modified flag settings to the `IOFLAG` register:

```
ustat2 = 0x00000000;
bit clr ustat2 FLG90|FLG80|FLG70|FLG60|FLG50|FLG40;
dm(IOFLAG) = ustat2;
```

System Design Considerations for Flags

Normally, if a flag is sampled or driven periodically, latency issues with respect to when flag pin change occurs are not a concern to the programmer. However, since the flag pins are sampled or driven with respect to the rising edge of `CLKOUT` (or `CLKIN` if $\overline{\texttt{CLKDBL}}$ is disabled), it is important that the application program allows enough time in certain programming situations for the flag state to be driven or sampled in the `CLKOUT` cycle boundary. This is especially true if the flags states are driven at a (core-clock) rate faster than the completion of the `CLKOUT` cycle boundary. The same also applies for the external device driving the flag pin as an input. The external device must drive the flag pin for at least 1 `CLKOUT` cycle to guarantee that it is latched properly by the processor.

When setting (or toggling flag pins) in a loop, you must insert extra `NOP`s instructions to prevent an overrun of setting and clearing a flag pin every cycle (or every other cycle for the `IOFLAG` register flag pins). For example, if you are using 2:1 mode, the `CCLK` runs twice as fast as `CLKOUT`. Depending on where the `CLKOUT` cycle boundary is with respect to the instruction writing to I/O flag register, the processor can take up to two `CCLK` cycles before that change is received external to the processor based on the rising edge of CLKOUT. The same cycle effect applies to the 3:1 and 4:1 clock ratios. For the 3:1 clock ratio, the processor requires up to three CCLK cycles before the change is received external to the processor based on the rising edge of CLKOUT. For the 4:1 clock ratio, the processor requires up to four CCLK cycles.

Since a core stall does not occur when writing to or reading from FLAG pins synchronized to the slower ADSP-21161 processor system clock, NOP instructions are required. In this case, write extra NOPs to ensure overruns do not occur in the higher clock rates.

The ADSP-21161 processor samples FLAG inputs at the CLKIN frequency except when $\overline{\text{CLKDBL}}$ is enabled. When $\overline{\text{CLKDBL}}$ is enabled, the processor samples FLAG inputs at the CLKOUT frequency. FLAG outputs must be held stable for at least one full CLKIN cycle.

Figure 13-9 shows the delay in setting (or toggling the flag pins) for clock modes 2:1, 3:1, and 4:1.

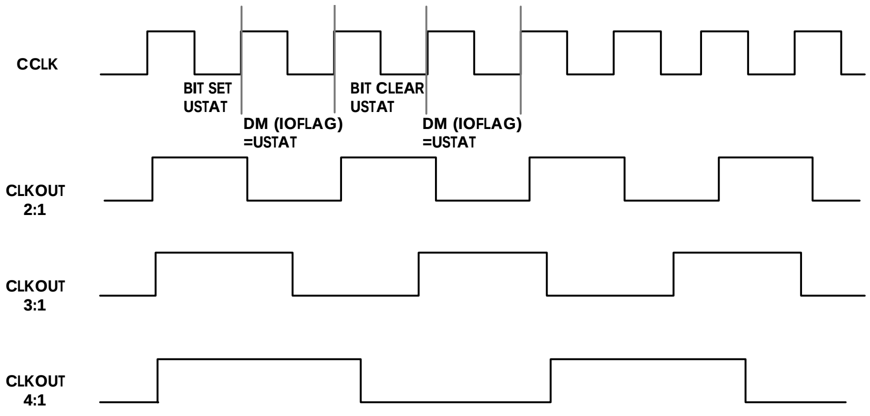


Figure 13-8. Delay in Setting Flag Pins for Clock Modes 2:1, 3:1 and 4:1

Example #3: Programming 2:1 Clock Ratio

The following example shows how to program an IOFLAG output with a 2:1 CCLK to CLKOUT ratio:

```
LCNTR = 100, DO flag_toggle UNTIL LCE;
bit tgl ustat1 FLG40;
flag_toggle: dm(IOFLAG) = ustat1;
```

Since a CLKOUT transition occurs every two CCLK instruction cycles, no additional NOP instructions are required.

Example #4: Programming 3:1 Clock Ratio

The following example shows how to set an IOFLAG output with 3:1 CCLK to CLKOUT ratio:

```
LCNTR = 100, DO flag_toggle UNTIL LCE;
bit tgl ustat1 FLG40;
dm(IOFLAG) = ustat1;
flag_toggle:nop;
```

Since a CLKOUT transition occurs every three CCLK instruction cycles, one NOP instruction is required to prevent the flag output overrun.

Example #5: Programming 4:1 Clock Ratio

The following example shows how to set an IOFLAG output with 4:1 CCLK to CLKOUT ratio:

```
LCNTR = 100, DO flag_toggle UNTIL LCE;
bit tgl ustat1 FLG40;
dm(IOFLAG) = ustat1;
nop;
flag_toggle:nop;
```

JTAG Interface Pins

The JTAG test access port consists of the TCK, TMS, TDI, TDO, and $\overline{\text{TRST}}$ pins. The JTAG port can be connected to a controller that performs a boundary scan for testing purposes. This port is also used by the Analog Devices Tools product line of JTAG emulator and development software to access on-chip emulation features. To allow the use of the emulator, a connector for its in-circuit probe must be included in the target system. For more information, see "Designing For JTAG Emulation" on page 13-49.

If $\overline{\text{TRST}}$ is not asserted (or held low) at power-up, the JTAG port is in an undefined state that may cause the processor to drive out on I/O pins that would normally be three-stated at reset. $\overline{\text{TRST}}$ can be held low with a jumper to ground on the target board connector.

Dual-Voltage Power-up Sequencing

The ADSP-21161 dual-voltage processor has special considerations related to power-up. Note that these are general recommendations, and specifics details on dual voltage power supply systems is beyond the scope of this book. When the system power is activated through the processor's dual power supply system, both supplies should be brought up as quickly as possible. Ideally, the two supplies, V_{DDEXT} and V_{DDINT} should be powered up simultaneously. Many commercially available dual supply regulators address simultaneous powerup requirements of the core and I/O. When designing a dual supply system, the designer should consider the relative voltage and ramp-up timing between the core and I/O voltages in order to avoid potential issues with long-term reliability.

The ADSP-21161 I/O pads have a network of internal diodes to protect the processor from damage by electrostatic discharge. These protection diodes connect the 1.8 V core and 3.3 V I/O supplies internally. Figure 13-9 shows how a network of protection diodes isolates the internal supplies and provides ESD protection for the I/O pins.

During the power-up sequence of the processor, differences in the ramp up rates and activation time between the two supplies can cause current to flow in the I/O ESD protection circuitry. When applying power separately to the V_{DDEXT} or V_{DDINT} pins, take precautions to prevent or limit the maximum current and duration conducted through the isolation diodes if the un powered pins are at ground potential. Since the ESD protection diodes connect the 1.8 V core and 3.3 V I/O supplies internally, these diodes can be damaged at any time the 1.8 V core supply voltage is present without the presence of the 3.3 V I/O supply.

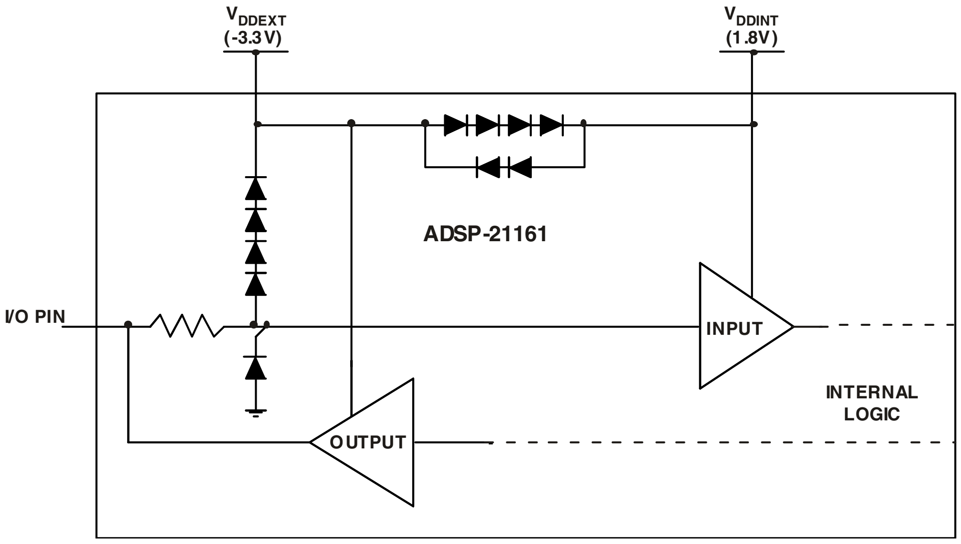


Figure 13-9. Protection Diodes and IO Pin ESD Protection

The ESD protection diodes connect the 1.8 V core and 3.3 V I/O supplies internally. Improper supply sequencing can cause damage to the ESD protection circuitry. If the 1.8 V supply is active for prolonged periods of time before the 3.3 V I/O supply is activated, there is a significant amount of loading on the I/O pins. Damage occurs because the I/O is powered from the 1.8 V supply through the ESD diodes.

To prevent this damage to the ESD diode protection circuitry, Analog Devices recommends including a bootstrap Schottky diode. The bootstrap Schottky diode connected between the 1.8 V and 3.3 V power supplies protects the ADSP-21161 from partially powering the 3.3 V supply. Including a Schottky diode shortens the delay between the supply ramps and thus prevent damage to the ESD diode protection circuitry. With this technique, of the 1.8 V rail rises ahead of the 3.3 V rail, the Schottky diode pulls the 3.3 V rail along with the 1.8 V rail.

For many power supply system designers, it may be easier to design the PLL clock gate workaround instead of shortening the V_{DDINT} ramp time. Moving between revisions does not require any hardware modifications to gate the clock. As long as the t_{CLKVDD} startup requirement is met then a reliable start-up reset of the PLL for revision 1.0/1.1 is assured. This requirement guarantees that the CLKIN source is present within 200 ms after the supplies are ramped. See the *ADSP-21161N DSP Microcomputer Data Sheet* for timing specifications. Holding off CLKIN up to a maximum of 200 ms is allowed.

Figure 13-10 shows a basic block diagram of the Schottky diode connected between the core and I/O voltage regulators and the processor. The anode of the diode must be connected to the 1.8 V supply. The diode must have a forward biased voltage of 0.6 V or less and must have a current rating sufficient to supply the 3.3 V rail of the system. The use of a Schottky diode is the recommended method suggested by Analog Devices.

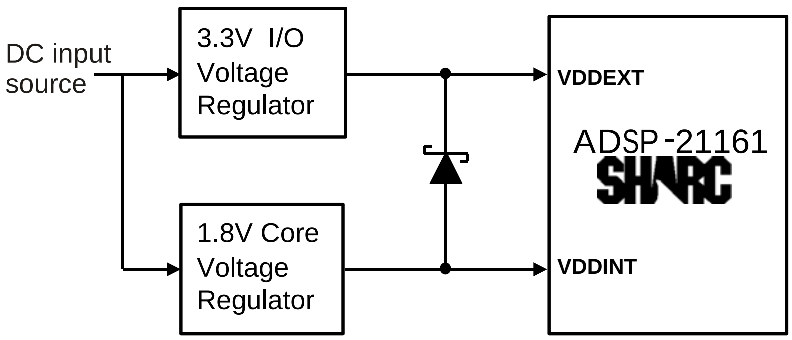


Figure 13-10. Dual 1.8 V/3.3 V Supplies With a Schottky Diode

For recommendations on managing power-up sequencing for the core I/O dual voltage supply, refer to the "Powerup Sequencing" specifications in the *ADSP-21161N SHARC DSP Microcomputer Data Sheet*.

PLL Start-Up (Revisions 1.0/1.1)

Two circuit blocks are included in the PLL start-up circuit to enable the PLL to lock effectively on start-up: a Power On Reset (POR) circuit and a 9-bit CLKIN counter. Figure 13-11 shows silicon revision 1.0/1.1 startup block diagram.

Power On Reset (POR) Circuit

The POR circuit monitors the voltage level on V_{DDINT} power supply and then generates a PLL pulse. This drives the PLL reset input circuit to reset the PLL to a default state. Figure 13-12 shows an example power-up and power-down waveform of V_{DDINT}. The POR circuit tracks this voltage internally to generate a PLL reset pulse. The actual POR output pulse,

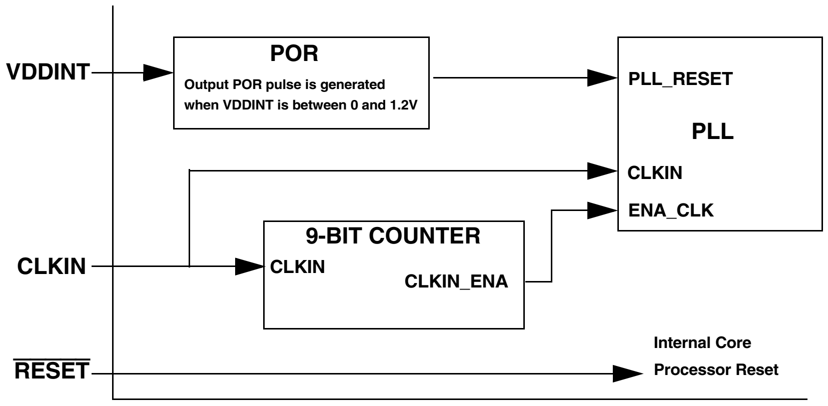

VDDINT
POR
Output POR pulse is generated when VDDINT is between 0 and 1.2V
PLL_RESET
PLL
CLKIN
ENA_CLK
9-BIT COUNTER
CLKIN
CLKIN_ENA
CLKIN
RESET
Internal Core Processor Reset

Figure 13-11. Power On Reset Circuit – Revisions 1.0 and 1.1

PLL_RESET, is generated as an active high pulse from the point at which VDDINT begins to ramp up from 0 V. It is deactivated when V_{DDINT} reaches 1.2 V.

For revisions 1.0 and 1.1, V_{DDINT} must ramp from 0 V to 1.8 V within 2 ms for the POR circuit to properly generate a PLL reset pulse

Figure 13-12 shows three PLL reset-related input signals: the top one is V_{DDINT}, the bottom two are derived from V_{DDINT} and are related to the POR circuit. The POR input tracks VDDINT up to 1.2 V before it drops down. This is used to generate the PLL reset pulse. As the input is rising to 1.2 V, the output of the POR generates the reset pulse for the PLL. After the POR input voltage reaches 1.2 V, the POR voltage drops off, which then deactivates the reset pulse connected to the PLL.

The duration of the POR circuit being driven active low is from 0 V to 1.2 V. If the system is powering down V_{DDINT} and coming back up again, there are a few requirements that must be met to properly generate a PLL reset pulse on the subsequent powerup. First, the POR circuit requires that V_{DDINT} voltage level is below 0.5 V. Secondly, re-ramp from 0.5 V to 1.2 V must occur within 1ms to guarantee another generated PLL_RESET pulse.

PLL CLKIN Enable Circuit

The 9-bit counter counts a certain number of CLKIN cycles before it allows the PLL to begin to lock to the incoming CLKIN frequency. This counter was added to allow the CLKIN source to amplify and oscillate to a stable fundamental frequency before the PLL begins to try to lock to the incoming frequency.

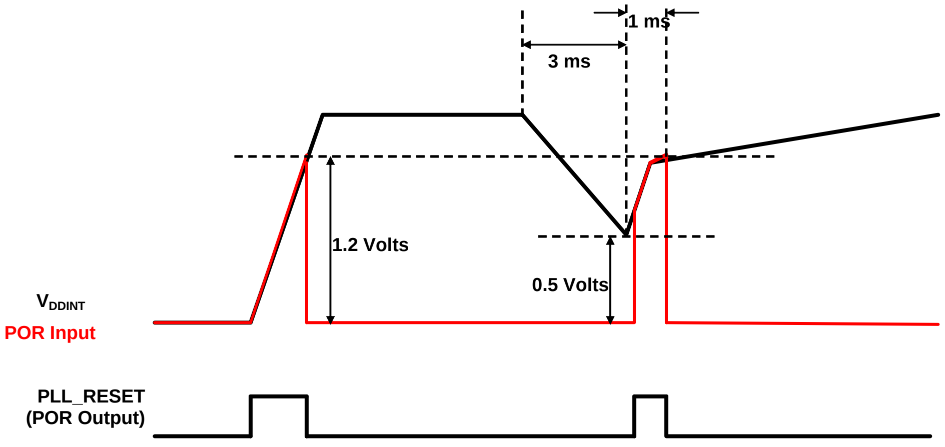


Figure 13-12. PLL Reset

Because oscillator or crystal startup times can range from 5 to 10 ms, the internal 512 cycle counter in some startup cases does not allow the CLKIN oscillator source to run at its locked oscillator fundamental frequency before the PLL clock input is enabled. Some oscillators might have a slow frequency ramp up time for 10 ms.

The revision 1.0 and 1.1 PLL can fail to lock or fail to continue to run if the CLKIN frequency goes below 15 MHz for more than 20 µs or when using $\overline{\text{CLKDBL}}$, the minimum CLKIN frequency cannot be less than 7.5 MHz.

There are two ways in which the PLL can be reset for revisions 1.0 and 1.1:

- Ensure that the V_{DDINT} ramp rate time is met (< 2 ms) with a stable CLKIN frequency applied when the POR circuit is enabled. When using an external clock oscillator powered by the V_{DDEXT} supply, bring up V_{DDEXT} for a recommended 25 ms before enabling V_{DDINT}. This allows the external CLKIN source to come up and stabilize before the V_{DDINT} power supply is brought up. The V_{DDINT} POR circuit then activates and generates a PLL pulse.
- Hold off or gate the CLKIN source until the V_{DDINT}/V_{DDEXT} supplies are known to be stable. This negates the V_{DDINT} ramp rate requirement if V_{DDINT} is exceeding 2 ms. Holding off CLKIN low or high until the supplies are stable also resets the internal PLL circuitry and allows the PLL to start reliably.

  Once, the processor is up and running, if you stop the CLKIN source, the PLL can lock up and not restart when CLKIN is reapplied. If there is a brown-out situation in your system, the watchdog circuit power-downs to at least 0.5 V and power-up of the V_{DDINT} supply within 1.0ms (to restart the POR circuit).

PLL Start-Up (Revision 1.2)

The PLL reset input and PLL CLKIN enable input directly to the $\overline{\text{RESET}}$ pin (Figure 13-13). This allows a PLL reset on the $\overline{\text{RESET}}$ rising edge. The $\overline{\text{RESET}}$ pin can be held low long enough to guarantee a stable CLKIN source and stable V_{DDINT}/V_{DDEXT} power supplies before the PLL is reset.

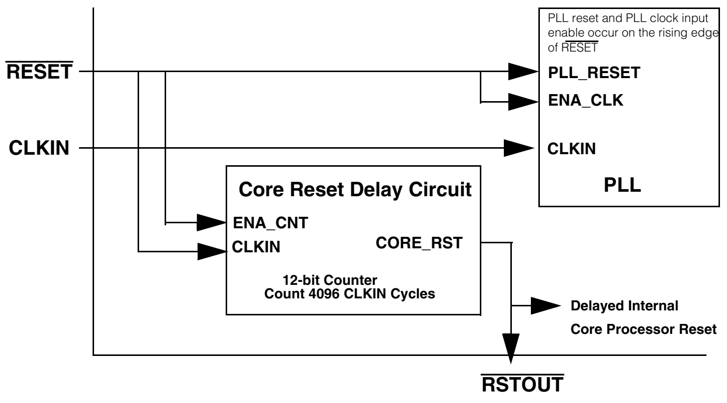


Figure 13-13. Power On Reset Circuit – Revisions 1.2

The PLL must lock to the CLKIN frequency (around 100 µs). Because the PLL resets on the rising edge of $\overline{\text{RESET}}$, the PLL needs time to lock to CLKIN before the core can execute or begin the boot process. A delayed core reset has been added via the delay circuit. There is a 12-bit counter that counts up to 4096 CLKIN cycles after $\overline{\text{RESET}}$ is transitioned from low to high. The delay circuit is activated at the same time the PLL is reset. A secondary $\overline{\text{RSTOUT}}$ pin (B15 which previously was a NC) has been added to allow system designers the option to have the ADSP-21161 processor reset another device after the core is reset. Note that as in previous silicon revisions, the CLKOUT is active during a reset. During reset the processor is in PLL BYPASS mode. CLKOUT frequency during reset depends upon $\overline{\text{CLKDBL}}$ pin. During reset if $\overline{\text{CLKDBL}}$ is HIGH then CLKOUT frequency = 1/4 of CLKIN frequency and if $\overline{\text{CLKDBL}}$ = LOW then CLKOUT frequency = 1/2 of CLKIN frequency.

The advantage of the delayed core reset is that the PLL can be reset any number of times without having to power-down the system. If there is a brown-out situation, the watchdog circuit only has to control the $\overline{\text{RESET}}$ pin to restart the PLL.

Designing For JTAG Emulation

The Analog Devices Tools product line of JTAG emulator is a development tool for debugging programs running in real time on target system hardware. The Analog Devices Tools product line of JTAG emulators provides a controlled environment for observing, debugging, and testing activities in a target system by connecting directly to the target processor through its JTAG interface.

Because the Analog Devices Tools product line of JTAG emulator controls the target system's processor through the processor's IEEE 1149.1 JTAG Test Access Port (TAP), non-intrusive in-circuit emulation is assured. The emulator uses the TAP to access the internal space of the processor, allowing the developer to load code, set breakpoints, observe variables, observe memory, examine registers, and so on. The processor must be halted to send data and commands, but once an operation is completed by the emulator, the system is set running at full speed with no impact on system timing.The emulator does not impact target loading or timing. The emulator's in-circuit probe connects to a variety of host computers (PCI bus, or USB) with plug-in boards.

Target systems must have a 14-pin connector in order to accept the Analog Devices Tools product line of JTAG emulator in-circuit probe, a 14-pin plug.

Designs must add this connector to the target board if the board is intended for use with the ADSP-21161 processor JTAG Emulator. The total trace length between the JTAG connector and the furthest device sharing the emulator's JTAG pins should be limited to 15 inches maximum for guaranteed operation. This length restriction must include the emulator's JTAG signals, which are routed to one or more ADSP-21161 processor devices, or a combination of ADSP-21161 processor devices and other JTAG devices on the chain.

Target Board Connector

The emulator interface to an Analog Devices JTAG processor is a 14-pin header, as shown in Figure 13-14. The customer must supply this header on their target board in order to communicate with the emulator. The interface consists of a standard dual row 0.025" square post header, set on 0.1" x 0.1" spacing, with a minimum post length of 0.235". Pin 3 is the key position used to prevent the pod from being inserted backwards. This pin must be clipped on the target board.

The clearance (length, width, and height) around the header must be as shown in Figure 13-19 on page 13-57. Maintain a minimum length of 0.15" and width of 0.10" for the target board header. The pod connector attaches the target board header in this area. Therefore, there must be clearance to attach and detach this connector. See the "JTAG Pod Connector" on page 13-57 for detailed drawings of the pod connector.

As can be seen in Figure 13-14, there are two sets of signals on the header, including the standard JTAG signals `TMS`, `TCK`, `TDI`, `TDO`, $\overline{\texttt{TRST}}$, $\overline{\texttt{EMU}}$ used for emulation purposes (via an emulator). Secondary JTAG signals `BTMS`, `BTCK`, `BTDI`, and $\overline{\texttt{BTRST}}$ are provided for optional use for board-level (boundary scan) testing. While they are rarely used, the "B" signals should be connected to a separate on-board JTAG boundary scan controller, if they are used. If the "B" signals are not used, tie them to ground as shown in Figure 13-15.

`BTCK` can alternately be activated (for some older silicon) to V_{CC} (+5 V, +3.3 V, or +2.5 V) using a 4.7 KΩ resistor, as described in previous documents. Tying the signal to ground is universal and works for all silicon.

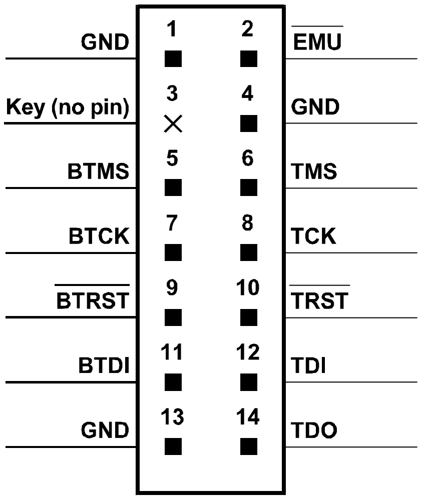


Figure 13-14. Emulator Interface for Analog Devices JTAG Processors

When the emulator is not connected to this header, jumpers should be placed across BTMS, BTCK, $\overline{\text{BTRST}}$, and BTDI as shown in Figure 13-15. This holds the JTAG signals in the correct state to allow the ADSP-21161 processor to run freely. All the jumpers should be removed when connecting the emulator to the JTAG header.

For a list of the state of each standard JTAG signal refer to Table 13-11. Use the following legend: O=Output, I=Input, and NU=Not Used.

The ADSP-21161 processor CLKIN signal is the clock signal line (typically 30 MHz or greater) that connects an oscillator to all processors in multiple processor systems requiring synchronization. In order for synchronous operations to work correctly the CLKIN signal on all the processors must be the same signal and the skew between them must be minimal (use clock drivers, or other means).

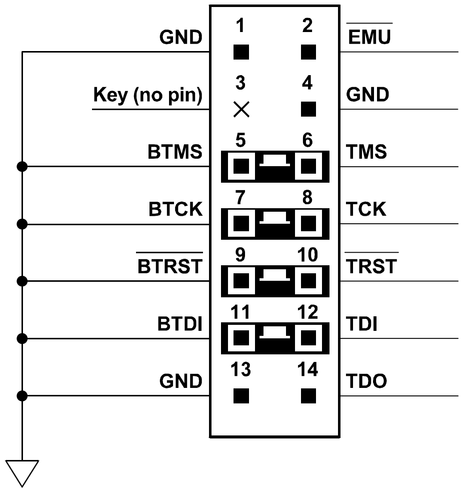


Figure 13-15. JTAG Target Board Connector With No Local Boundary Scan

Table 13-10. State of Standard JTAG Signals

Signal	Description	Emulator	ADSP-21161
TMS	Test Mode Select	O	I
TCK	Test Clock (10 MHz)	O	I
$\overline{\text{TRST}}$	Test Reset	O	I
TDI	Test Data In	O	I
TDO	Test Data Out	I	O
$\overline{\text{EMU}}$	Emulation Pin	I	O (Open Drain)
CLKIN	Processor Clock Input	NU	I

Note that the `CLKIN` signal is not used by the emulator and can cause noise problems if connected to the JTAG header. Legacy documents show it connected to pin 4 of the JTAG header. Pin 4 should be tied to ground on the 14-pin JTAG header (do not connect the JTAG header pin to the processor's CLKIN signal). If you have already connected it to the JTAG header pin, and are experiencing noise from this signal, simply clip this pin on the 14-pin JTAG header.

The final connections between a single processor target and the emulation header (within 6 inches) are shown in Figure 13-16. A 4.7 KΩ pull-up resistor has been added on TCK, TDI and TMS for increased noise resistance.

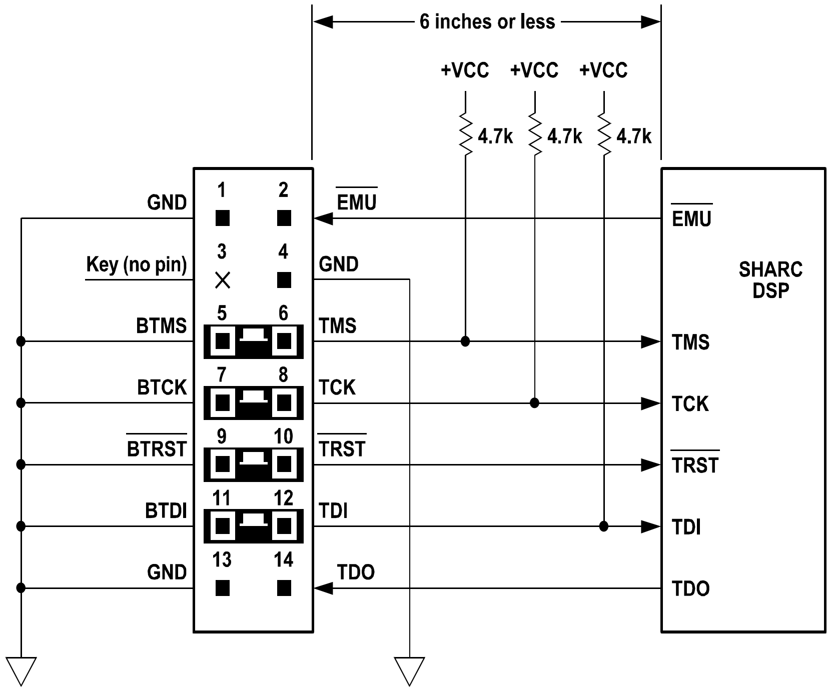


Figure 13-16. Single Connection to the JTAG Header

If a design uses more than one processor (or other JTAG device in the scan chain), or if the JTAG header is more than 6 inches from the processor, use a buffered connection scheme as shown in Figure 13-17 on page 13-55 (no local boundary scan mode shown). To keep signal skew to a minimum, be sure the buffers are all in the same physical package (typical chips have 6, 8, or 16 drivers). Using a buffer that includes a series of resistors such as the 74ABT2244 family can reduce ringing on the JTAG signal lines. For low voltage applications (3.3 V, 2.5 V, and 1.8 V I/O), the 74ALVT, and 74AVC logic families is useful. Also, note the position of the pull-up resistor on $\overline{\texttt{EMU}}$. This is required since the $\overline{\texttt{EMU}}$ line is an open drain signal.

If more than one processor (or JTAG device) is on the target (in the scan chain), you must buffer the JTAG header. This keeps the signals clean and avoids noise problems that occur with longer signal traces (ultimately resulting in reliable emulator operation).

Although the theoretical number of devices that can be supported (by the software) in one JTAG scan chain is large (50 devices or more) it is not recommended that you use more than eight physical devices in one scan chain. A physical device could however contain many JTAG devices such as inside a multi-chip module. The recommendation of not more than eight physical devices is mostly due to the transmission line effects that appear in long signal traces, and based on some field-collected empirical data. The best approach for large numbers of physical devices is to break the chain into several smaller independent chains, each with their own JTAG header and buffer. If this is not possible, at least add some jumpers that can reduce the number of devices in one chain for debug purposes, and pay special attention in the layout stage for transmission line effects.

Layout Requirements

All JTAG signals (TCK, TMS, TDI, TDO, $\overline{\texttt{EMU}}$, $\overline{\texttt{TRST}}$) should be treated as critical route signals. Specify a controlled impedance requirement for each route (value depends on your circuit board, typically 50-75 Ω). Keeping crosstalk and inductance to a minimum on these lines by using a good ground plane and by routing away from other high noise signals such as

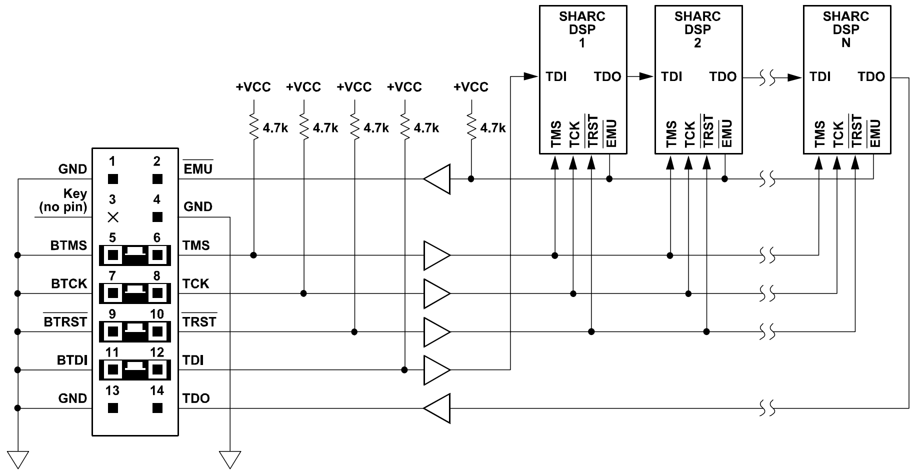


Figure 13-17. Multiple Connection to JTAG Header

clock lines is also important. Keep these routes as short and clean as possible, and keep the bused signals (TMS, TCK, $\overline{\text{TRST}}$, $\overline{\text{EMU}}$) as close to the same length as possible.

The JTAG TAP relies on the state of the TMS line and the TCK clock signal. If these signals have glitches (due to ground bounce, crosstalk, etc.) unreliable emulator operation results. When experiencing emulator problems, look at these signals using a high-speed digital oscilloscope. These lines must be clean, and may require special termination schemes. If you are buffering the JTAG header (most applications do) you must provide signal termination appropriate for your target board (series, parallel, R/C, etc.).

Power Sequence for Emulation

The power-on sequence for your target and emulation system is as follows:

1. Apply power to the emulator first, then to the target board. This ensures that the JTAG signals are in the correct state for the ADSP-21161 processor to run free.

2. Upon power-on, the emulator drives the $\overline{\text{TRST}}$ signal low, keeping the processor TAP in the test-logic-reset state, until the emulation software takes control.

   Removal of power should be done in reverse: Turn off power to the target board then to the emulator.

Additional JTAG Emulator References

The IEEE 1149.1 JTAG standard is sponsored by the Test Technology Standards Committee of the IEEE Computer Society, and published by the IEEE. The latest versions at the time of this publication are IEEE Standard. 1149.1-1990 and IEEE Standard 1149.1a-1993. To order a copy, call the IEEE at 1-800-678-4333 in the US and Canada, 1-908-981-1393 outside of the US and Canada. Visit the IEEE standards web site at http://standards.ieee.org/.

Pod Specifications

This section contains design details on various emulator pod designs by the Analog Devices Tools product line. The emulator pod is the device that connects directly to the target board 14-pin JTAG header. See also *Engineer-to-Engineer Notes EE-68*.

JTAG Pod Connector

Figure 13-18 details the dimensions of the JTAG pod connector at the 14-pin target end. Figure 13-19 displays the keep-out area for a target board header. The keep-out area allows the pod connector to properly seat onto the target board header. This board area should contain no components (chips, resistors, capacitors, and so on). The dimensions are referenced to the center of the 0.25” square post pin.

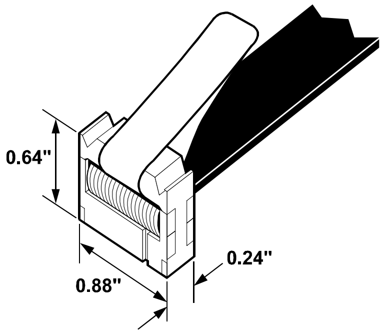


Figure 13-18. JTAG Pod Connector at the 14-pin Target End

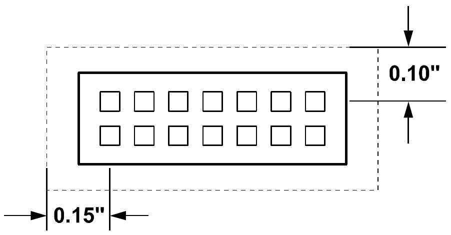


Figure 13-19. Keep-Out Area for a Target Board Header

3.3 V Pod Logic

A portion of the Analog Devices Tools product line 3.3 V emulator pod interface is shown in Figure 13-20. This figure describes the driver circuitry of the emulator pod. As can be seen, TMS, TCK and TDI are driven with a 33 Ω series resistor. $\overline{\text{TRST}}$ is driven with a 100 Ω series resistor. The TDO and CLKIN pins are terminated with an optional 91/120 Ω parallel terminator. The $\overline{\text{EMU}}$ signal is pulled up with a 4.7 KΩ resistor. The 74LVT244 chip drives the signals at 3.3 V, with a maximum current rating of ±32 mA.

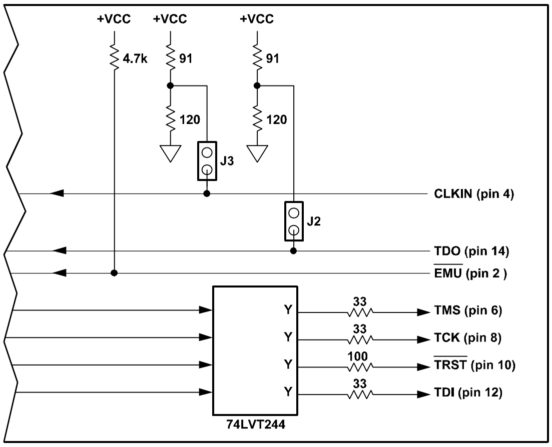


Figure 13-20. 3.3V JTAG Pod Driver Logic

Parallel terminate the TMS, TCK, $\overline{\text{TRST}}$, and TDI lines locally on your target board, if needed, since they are driven by the pod with sufficient current drive (±32 mA). In order to use the terminators on the TDO line (CLKIN is not used), you MUST have a buffer on your target board JTAG header.

The ADSP-21161 processor is not capable of driving the parallel terminator load directly with TDO. Assuming the proper buffers are included, use the optional parallel terminators by placing a jumper on J2.

2.5 V Pod Logic

A portion of the Analog Devices Tools product line 2.5 V emulator pod interface is shown in Figure 13-21. This figure describes the driver circuitry of the emulator pod. As can be seen, the TMS, TCK, and TDI liness are driven with a 33 Ω series resistor. The $\overline{\text{TRST}}$ signal is driven with a 100 Ω series resistor. The TDO line is pulled up with a 4.7 KΩ resistor and terminated with an optional parallel terminator that can be configured by the user. $\overline{\text{EMU}}$ is pulled up with a 4.7 KΩ resistor.

The CLKIN signal is not used and not connected inside the pod. The 74ALVT16244 chip drives the signals at 2.5 V, with a maximum current rating of ±8 mA.

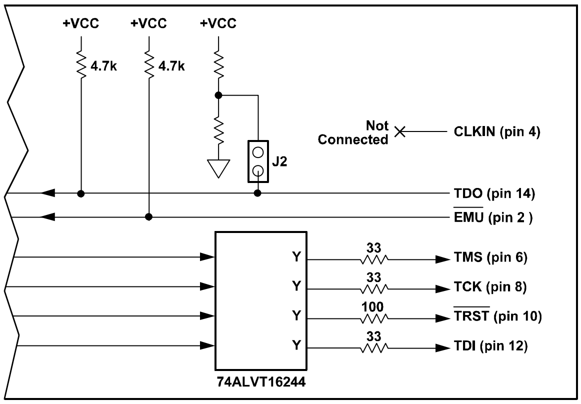


Figure 13-21. 2.5 V JTAG Pod Driver Logic

You can terminate the TMS, TCK, $\overline{\text{TRST}}$, and TDI lines locally on your target board, if needed, as long as the terminator's current use does not exceed the driver's maximum current supply (±8 mA). In order to use the terminator on the TDO line, include a buffer on your target board JTAG header. The ADSP-21161 processor is not capable of driving a parallel terminator load (typically 50-75 Ω) directly with TDO. Assuming you have the proper buffers, you may use the optional parallel terminator by adding the appropriate resistors and placing a jumper on J2.

Conditioning Input Signals

The ADSP-21161 processor is a CMOS device. It has input conditioning circuits which simplify system design by filtering or latching input signals to reduce susceptibility to glitches or reflections.

The following sections describe why these circuits are needed and their effect on input signals.

A typical CMOS input consists of an inverter with specific N and P device sizes that cause a switching point of approximately 1.4 V. This level is selected to be the midpoint of the standard TTL interface specification of V_{IL} =0.8 V and V_{IH} =2.0 V. This input inverter, unfortunately, has a fast response to input signals and external glitches wider than about 1 ns. Filter circuits and hysteresis are added after the input inverter on some processor inputs, as described in the following sections.

Link Port Input Filter Circuits

The ADSP-21161 processor's link port input signals have on-chip filter circuits rather than glitch rejection circuits. Filtering is not used on most signals because it delays the incoming signal and the timing specifications.

Filtering is implemented only on the link port data and clock inputs. This is possible because the link ports are self-synchronized. The clock and data are sent together. It is not the absolute delay but rather the relative delay between clock and data that determines performance margin.

By filtering both LxCLK and LxDAT7-0 with identical circuits, response to LxCLK glitches and reflections are reduced but relative delay is unaffected. The filter has the effect of ignoring a full strength pulse (a glitch) narrower than approximately 2 ns. Glitches that are not full strength can be somewhat wider. The link ports do not use glitch rejection circuits because they can be used with longer, series-terminated transmission lines where the reflections do not occur near the signal transitions.

RESET Input Hysteresis

Hysteresis is used only on the $\overline{\text{RESET}}$ input signal. Hysteresis causes the switching point of the input inverter to be slightly above 1.4 V for a rising edge and slightly below 1.4V for a falling edge. The value of the hysteresis is approximately ± 0.1 V. The hysteresis is intended to prevent multiple triggering of signals which are allowed to rise slowly, as might be expected on a reset line with a delay implemented by an RC input circuit. Hysteresis is not used to reduce the effect of ringing on input signals with fast edges, because the amount of hysteresis that can be used on a CMOS chip is too small to make much difference. The small amount of hysteresis allowable is due to the restrictions on the tolerance of the V_{IL} and V_{IH} TTL input levels under worst case conditions. Refer to the data sheet for exact specifications.

Designing For High Frequency Operation

Because the processor can operate at very fast clock frequencies, signal integrity and noise problems must be considered for circuit board design and layout. The following sections discuss these topics and suggest various techniques to use when designing and debugging ADSP-21161 processor systems.

All synchronous behavior is specified to `CLKIN`. System designers are encouraged to clock synchronous peripherals/memory (which are attached to the external port) with this same clock source (or a different low-skew output from the same clock driver). Alternatively, the clock output (`CLKOUT`) from the processor may be employed to clock synchronous peripherals/memory. Note the following behavior for `CLKOUT`:

1. The processor whose `ID2-0`=000 (uniprocessor), or 001 drives `CLKOUT` during reset.

2. `CLKOUT` is specified relative to `CLKIN` in the *ADSP-21161N DSP Microcomputer Data Sheet*. When using this output to clock system components, the phase and jitter terms associated with this output must be treated as additional derating factors in determining specs. The use of `CLKOUT` as a clock source to SBSRAMs can result in negative hold times and is not recommended.

3. For systems not needing `CLKOUT` as a clock source, `CLKOUT` may be used to identify the current bus master. This requires that the outputs not be tied together. If and when this debug feature is not needed, the `CLKOUT` output can be disabled by setting the `COD` bit in the `SYSCON` register. The bus master can be identified by checking the `BMSTR` pin.

Clock Specifications and Jitter

The clock signal must be free of ringing and jitter. Clock jitter can easily be introduced in a system where more than one clock frequency exists. High frequency jitter on the clock to the processor may result in abbreviated internal cycles.

As shown in Figure 13-22, keep the portions of the system that operate at different frequencies as physically separate as possible. The clock supplied to the ADSP-21161 processor must have a rise time of 3 ns or less and must meet or exceed a high and low voltage of 2.3 V and 0.8 V, respectively.

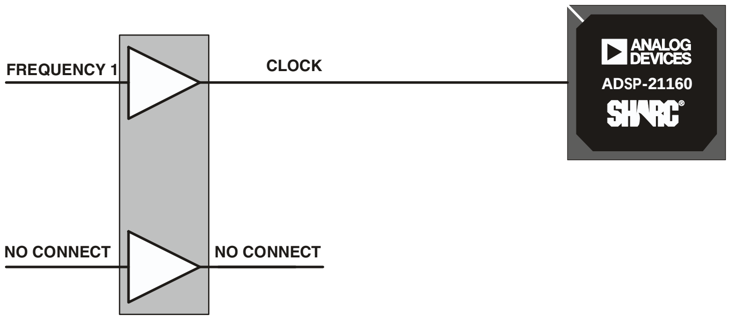


Figure 13-22. Reducing Clock Jitter and Ring

Never share a clock buffer IC with a signal of a different clock frequency. This introduces excessive jitter.

Clock Distribution

There must be low clock skew between processors in a multiprocessor cluster when communicating synchronously on the external bus. The clock must be routed in a controlled-impedance transmission line that can be properly terminated at either the end of the line or the source.

Figure 13-23 illustrates end-of-line termination for the clock. End-of-line termination is not usually recommended unless the distance between the processors is extremely small, because devices that are at a different wire distance from each other receive a skewed clock. This is due to the propagation delay of a PCB transmission line, which is typically 5 to 6 inches/ns.

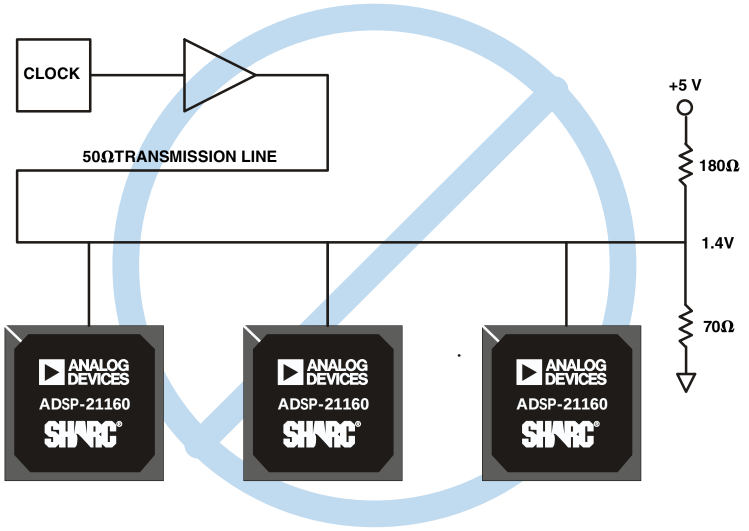


Figure 13-23. End-Of-Line Termination for the Clock Caution

Figure 13-24 illustrates source termination for the clock. Source termination allows delays in each path to be identical. Each device must be at the end of the transmission line because only there does the signal have a single transition. The traces must be routed so that the delay through each is matched to the others. Line impedance higher than 50 Ω may be used, but clock signal traces should be in the PCB layer closest to the ground plane to keep delays stable and crosstalk low. More than one device may be at the end of the line, but the wire length between them must be short and the impedance (capacitance) of these must be kept high. The matched

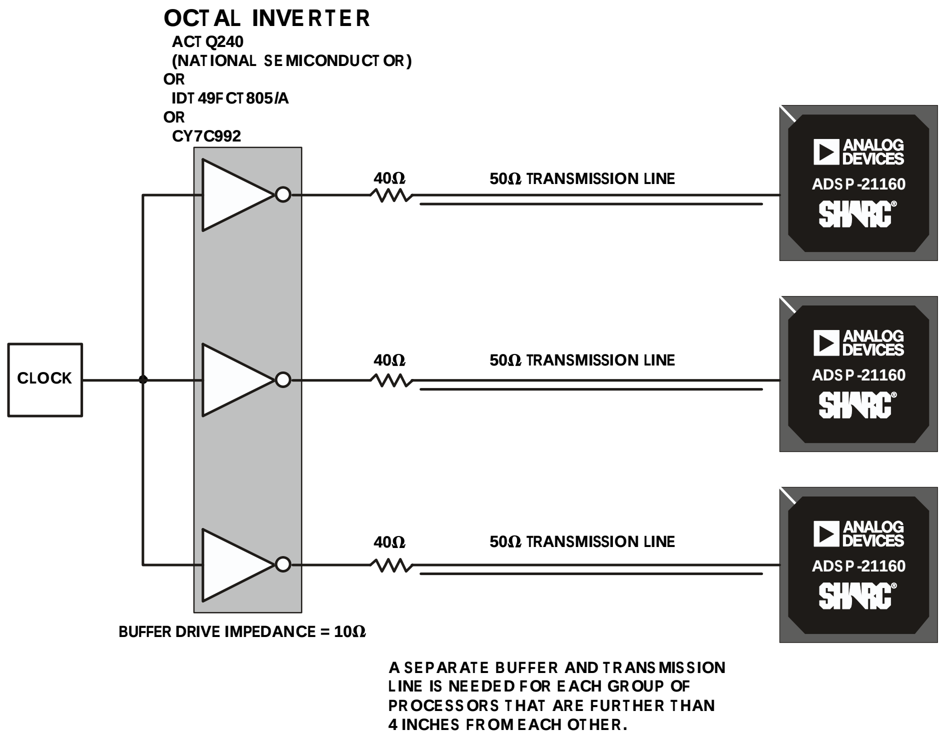


Figure 13-24. Use Source Termination to Distribute the Clock

inverters must be in the same IC and must be specified for a low skew (< 1 ns) with respect to each other. This skew should be as small as possible because it subtracts from the margin on most specifications.

Point-to-Point Connections

Unlike previous SHARC processors, the ADSP-21161 processor contains internal series resistance equivalent to 50 Ω on all drivers except the `CLKIN` and `XTAL` pins. Therefore, for traces longer than six inches, external series resisters on control, data, clock or frame sync pins are not required to dampen reflections from transmission line effects for point-to-point connections. However, for more complex networks such as a star configuration, a series termination is still recommended. Figure 13-26 shows an internal resistance in the driver of 10 Ω. The additional 40 Ω series resister at the driver pad results in a total resistance of 50 Ω. For more specific guidance on related issues, see the reference source in "Recommended Reading" on page 13-71 for suggestions on transmission line termination. Also, see the *ADSP-21161N DSP Microcomputer Data Sheet* for output drivers' rise and fall time data.

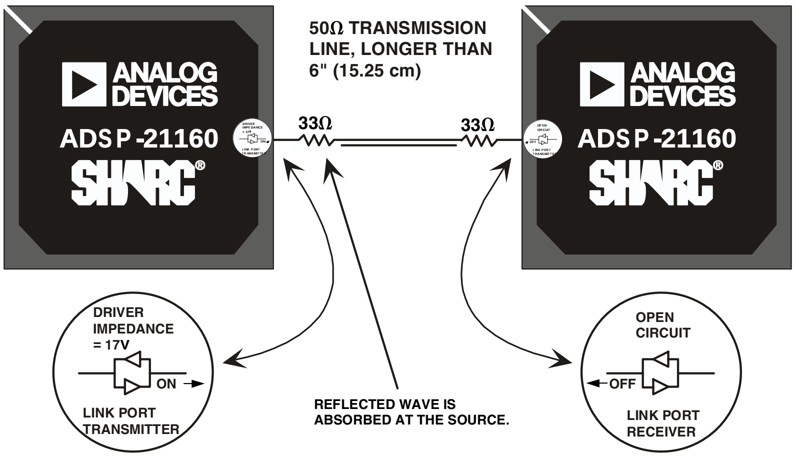


Figure 13-25. Source Termination for Point-to-Point Connectors

For link port operation at the full internal clock rate it is important to maintain low skew between the data (`LxDAT7-0`) and clock (`LxCLK`). For full speed operation with a 100 MHz internal clock, a skew of less than 0.5 ns is required.

Although the ADSP-21161 processor's serial ports may be operated at a slow rate, the output drivers still have fast edge rates.

Signal Integrity

The capacitive loading on high-speed signals should be reduced as much as possible. Loading of buses can be reduced by using a buffer for devices that operate with wait states, for example DRAMs. This reduces the capacitance on signals tied to the zero-wait-state devices, allowing these signals to switch faster and reducing noise-producing current spikes.

Signal run length (inductance) should also be minimized to reduce ringing. Extra care should be taken with certain signals such as the read and write strobes ($\overline{\text{RD}}$, $\overline{\text{WR}}$) and acknowledge (ACK). In a multiprocessor cluster, each processor can drive the read or write strobes. In this case, some damping resistance should be put in the signal path if the line length is greater than 6 inches (Figure 13-26). This is at the expense of additional signal delay. The time budget for these signals should be carefully analyzed.

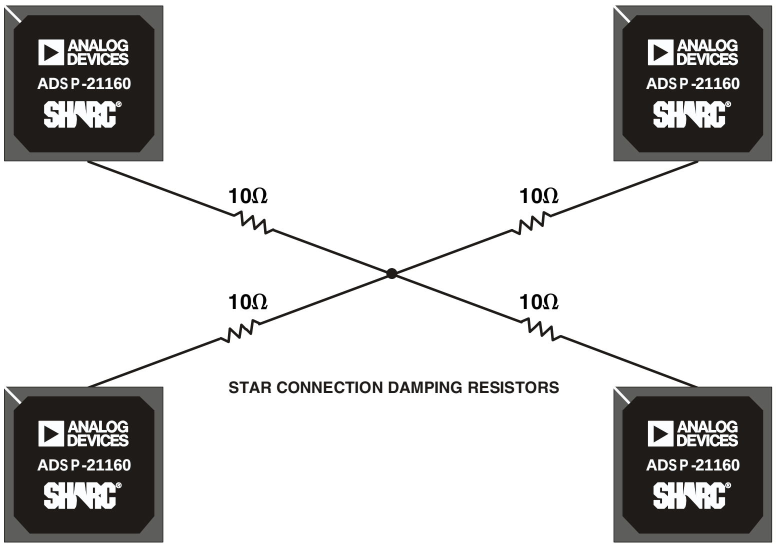


Figure 13-26. Star Connection Damping Resistors

Other Recommendations and Suggestions

The following are some additional suggestions for successfully designing an ADSP-21161 hardware platform.

- Use more than one ground plane on the PCB to reduce crosstalk. Be sure to use lots of vias between the ground planes. One V_{DD} plane for each supply is sufficient. These planes should be in the center of the PCB.

- Keep critical signals such as clocks, strobes, and bus requests on a signal layer next to a ground plane and away from or layout perpendicular to other non-critical signals to reduce crosstalk. For example, data outputs switch at the same time that $\overline{\text{BRx}}$ inputs are sampled; if the layout permits crosstalk between them, the system could have problems with bus arbitration.

- Position the processors on both sides of the board to reduce area and distances if possible.
- Design for lower transmission line impedances to reduce crosstalk and to allow better control of impedance and delay.
- Use of 3.3 V peripheral components and power supplies to help reduce transmission line problems, because the receiver switching voltage of 1.5 V is close to the middle of the voltage swing. In addition, ground bounce and noise coupling is less.
- Experiment with the board and isolate crosstalk and noise issues from reflection issues. This can be done by driving a signal wire from a pulse generator and studying the reflections while other components and signals are passive.

Decoupling Capacitors and Ground Planes

Ground planes must be used for the ground and power supplies. Designs should use a minimum of eight bypass capacitors (six 0.1 µF and two 0.01 µF ceramic) each for IO and core. The capacitors should be placed very close to the V_{DDEXT} and V_{DDINT} pins of the package as shown in Figure 13-27. Use short and fat traces for this. The ground end of the capacitors should be tied directly to the ground plane inside the package footprint of the processor (underneath it, on the bottom of the board), not outside the footprint. A surface-mount capacitor is recommended because of its lower series inductance. Connect the power plane to the power supply pins directly with minimum trace length. The ground planes must not be densely perforated with vias or traces as their effectiveness is reduced. In addition, there should be several large tantalum capacitors on the board.

Designs can use either bypass placement case shown in Figure 13-27 or combinations of the two. Designs should try to minimize signal feedthroughs that perforate the ground plane.

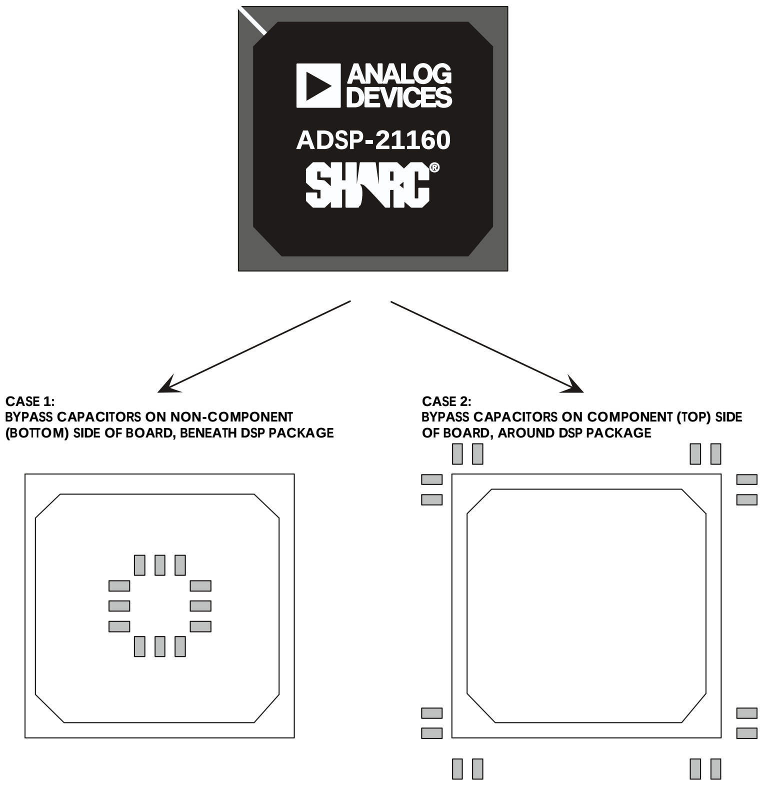

ANALOG DEVICES
ADSP-21160
SHARC®
CASE 1:
BYPASS CAPACITORS ON NON-COMPONENT (BOTTOM) SIDE OF BOARD, BENEATH DSP PACKAGE
CASE 2:
BYPASS CAPACITORS ON COMPONENT (TOP) SIDE OF BOARD, AROUND DSP PACKAGE

Figure 13-27. Bypass Capacitor Placement

Oscilloscope Probes

When making high-speed measurements, be sure to use a "bayonet" type or similarly short (< 0.5 inch) ground clip, attached to the tip of the oscilloscope probe. The probe should be a low-capacitance active probe with 3 pF or less of loading. The use of a standard ground clip with 4 inches of ground lead causes ringing to be seen on the displayed trace and makes the signal appear to have excessive overshoot and undershoot. A 1 GHz or better sampling oscilloscope is needed to see the signals accurately.

Recommended Reading

The text *High-Speed Digital Design: A Handbook of Black Magic* is recommended for further reading. This book is a technical reference that covers the problems encountered in state-of-the-art, high-frequency digital circuit design, and is an excellent source of information and practical ideas. Topics covered in the book include:

- High-Speed Properties of Logic Gates
- Measurement Techniques
- Transmission Lines
- Ground Planes and Layer Stacking
- Terminations and Vias
- Power Systems
- Ribbon Cables and Connectors
- Clock Distribution and Clock Oscillators
- *High-Speed Digital Design: A Handbook of Black Magic*, Johnson and Graham, Prentice Hall, Inc., ISBN 0-13-395724-1.

Booting Single and Multiple Processors

Programs can be automatically downloaded to the internal memory of an ADSP-21161 processor after power-up or after a software reset. This process is called booting. The processor supports four booting modes: EPROM, host, SPI, and link port. For cases when the processor must execute instructions from external memory without booting, a “No boot” mode may also be configured. For information on the setup and DMA processes for booting a single processor, see “Bootloading Through The External Port” on page 6-70 and “Bootloading Through The Link Port” on page 6-88 and “Bootloading Through the SPI Port” on page 6-113.

Multiprocessor systems can be booted from a host processor, from external EPROM, through a link port, SPI port, or from external memory.

Table 13-11. Booting Modes

Booting Mode	EBOOT	LBOOT	BMS
EPROM Connect BMS to EPROM chip select	1	0	Output
Host processor	0	0	1 (Input)
Serial boot via SPI	0	1	0 (Input)
Link port	0	1	1 (Input)
No booting Processor executes from external memory	0	0	0 (Input)
Reserved	1	1	x (Input)

Multiprocessor Host Booting

To boot multiple ADSP-21161 processors from a host, each processor must have its EBOOT, LBOOT, and $\overline{\text{BMS}}$ pins configured for host booting: EBOOT=0, LBOOT=0, and $\overline{\text{BMS}}$=1. After system powerup, each processor is in the idle state and the $\overline{\text{BR}}$x bus request lines are deasserted. The host must assert the $\overline{\text{HBR}}$ input and boot each processor by:

- Asserting its $\overline{\text{CS}}$ pin (for asynchronous). The $\overline{\text{CS}}$ pin of all processors can be asserted to select and boot them simultaneously by host broadcast writes.
- Downloading instructions as described in "Booting Another processor" on page 7-108

Multiprocessor EPROM Booting

There are two methods of booting a multiprocessor system from an EPROM. Processors perform the following steps in these methods:

- Arbitrate for the bus
- DMA the 256-word boot stream, after becoming bus master
- Release the bus
- Execute the loaded instructions

Booting From a Single EPROM

The $\overline{\text{BMS}}$ signals from each processor may be wire-ORed together to drive the chip select pin of the EPROM. Each processor can boot in turn, according to its priority. When the last one has finished booting, it must inform the others (which may be in the idle state) that program execution can begin (if all processors are to begin executing instructions simultaneously). An example system that uses this alternating technique appears in Figure 13-28. When multiple processors boot from one EPROM, the processors can boot either identical code or different code from the EPROM. If the processors load differing code, a jump table (based on processor ID) can be used to select the code for each processor.

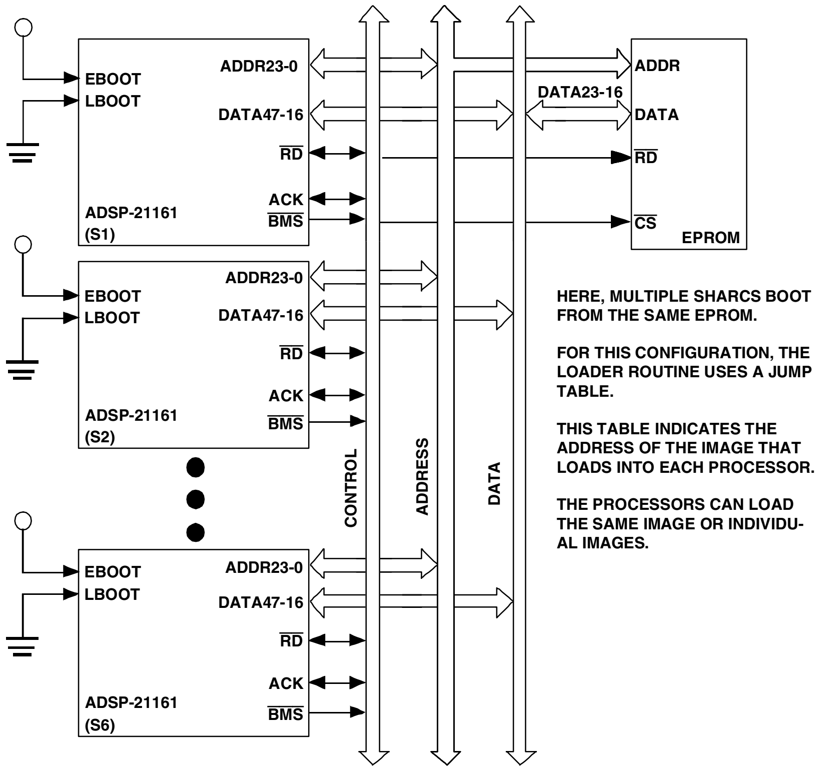

EBOOT
LBOOT
ADDR23-0
DATA47-16
RD
ACK
BMS
ADSP-21161
(S1)
ADSP-21161
(S2)
ADSP-21161
(S6)
CONTROL
ADDRESS
DATA
ADDR
DATA23-16
DATA
RD
CS
EPROM
HERE, MULTIPLE SHARCS BOOT FROM THE SAME EPROM.
FOR THIS CONFIGURATION, THE LOADER ROUTINE USES A JUMP TABLE.
THIS TABLE INDICATES THE ADDRESS OF THE IMAGE THAT LOADS INTO EACH PROCESSOR.
THE PROCESSORS CAN LOAD THE SAME IMAGE OR INDIVIDUAL IMAGES.

Figure 13-28. Alternating Booting From an EPROM

Sequential Booting

The `EBOOT` pin of the ADSP-21161 processor with `IDx`=1 must be set high for EPROM booting. All other processors should be configured for host booting (`EBOOT`=0, `LBOOT`=0, and $\overline{\texttt{BMS}}$=1), which leaves them in the idle state at startup and allows the processor with `IDx`=1 to become bus master and boot itself. Only the $\overline{\text{BMS}}$ pin of processor #1 is connected to the chip select of the EPROM. When processor #1 has finished booting, it can boot the remaining processors by writing to their external port DMA buffer 0 (EPB0) via multiprocessor memory space. An example system that uses this sequential technique appears in Figure 13-29.

Multiprocessor Link Port Booting

In systems where multiple processors are not connected by the parallel external bus, booting can be accomplished from a single source through the link ports. To sequentially boot all of the processors, a parallel common connection should be made to link port buffer 0 (LBUF0) on each of the processors. If only a daisy chain connection exists between the processors' link ports, then each processor can boot the next one in turn. Link buffer 0 must always be used for booting.

If you want to boot multiple processors simultaneously, you must add glue logic to handle multiple LxACK signals.

Multiprocessor Booting From External Memory

If external memory contains a program after reset, then the processor with IDx=1 should be set up for no boot mode. It begins executing from address 0x0020 0004 in external memory. When booting has completed, the other processors may be booted by processor #1 if they are set up for host booting, or they can begin executing out of external memory if they are set up for no boot mode. Multiprocessor bus arbitration allows this booting to occur in an orderly manner. The bus arbitration sequence after reset is described in "Multiprocessor Bus Arbitration" on page 7-93.

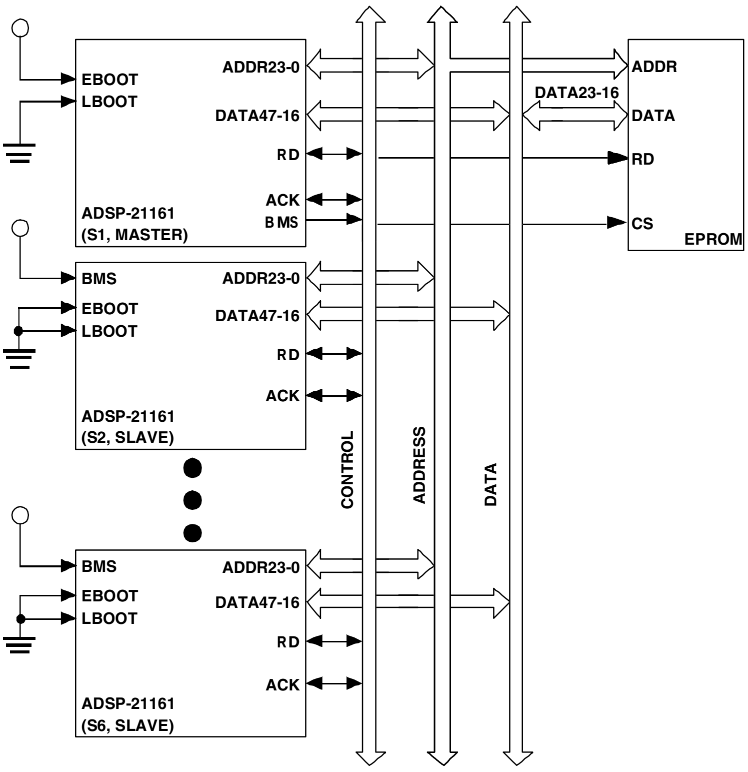


Figure 13-29. Sequential Booting from an EPROM

Data Delays, Latencies, and Throughput

Table 13-13 specifies data delays, latencies and throughput for the ADSP-21161 processor. *Data delay* and *latency* are defined as the number cycles after the first cycle required to complete the operation. A zero wait-state memory has a data delay of zero. A single waitstate memory has a data delay of one. *Throughput* is the maximum rate at which the operation is performed. Data delay and throughput are the same whether the access is from a host processor or from another ADSP-21161 processor.

Execution Stalls

The following events can cause an execution stall for the ADSP-21161 processor:

- 1 cycle on a program memory data access with instruction cache miss
- 2 cycles on non-delayed branches
- 2 cycles on normal interrupts
- 5 cycles on vector interrupts
- 1-2 cycles on short loops with small iterations
- n cycles on an `IDLE` instruction

DAG Stalls

1 cycle hold on register conflict

Memory Stalls

- 1 cycle on PM and DM bus access to the same block of internal memory
- n cycles if conflicting accesses to external memory
- n cycles if access to external memory (until I/O buffers are cleared out)

- n cycles if external access and ADSP-21161 processor does not control the external bus
- n cycles until external access is complete (for example, waitstates, idle cycles, and so on.)

IOP Register Stalls

- n cycles if PM and DM bus access IOP registers (both must complete)
- n cycles if conflict occurs with slave access

DMA Stalls

- 1 cycle if an access to a DMA parameter register conflicts with the DMA address generation (for example, writing to the register while a register update is taking place) or reading while a DMA register conflicts with DMA chaining
- 1 cycle if an access to a DMA parameter register or the `DMASTAT` register conflicts with DMA address generation. For example, one cycle stall occurs when writing to a DMA register while a register update is taking place. Similarly, a one cycle stall occurs when reading from a DMA register while DMA chaining is taking place.
- n cycles if writing (or reading) to a DMA buffer when the buffer is full (or empty)

Link Port and Serial Port Stalls

- 1 cycle if two link buffer reads back-to-back
- n cycles if write to a full buffer or read from an empty buffer

Table 13-12. Data Delay and Throughput

Operation	**Minimum Data Delay (cycles)**	**Maximum Throughput (cycles/ transfer)**
Core processor access to external memory	0	1
Synchronous access to slave's IOP registers[1]		
Read (Transfer out)	0	2
Write (Transfer in)	2	1
Slave mode DMA		
Read (Transfer out)	-	2[2]
Write (Transfer in)	-	1
Master mode DMA		
Transfer out	-	1
Transfer in	-	1
Handshake mode DMA[3]		
Read/Write (Transfer in/out)	3	1
External-Handshake mode DMA[4]		
Read/Write (Transfer in/out)	3	1

1 The delay is between data in the IOP register and at the external port. For example, an IOP register is written in the second cycle after a write completes at the external port.
2 These transfer rates are limited by the speed of the read of the DMA FIFO buffer. When bursting is enabled, the first read requires three cycles. The maximum burst read throughput is 3-2-2-2.
3 The delay is between DMA and $\overline{\text{DMAR}}$x.
4 The delay is between $\overline{\text{DMAR}}$x and the external transfer.

Table 13-13. Latencies and Throughput

Operation	**Minimum Data Delay (cycles)**	**Maximum Throughput (cycles/ transfer)**
Interrupts ($\overline{\text{IRQ}}$2-0)	3	-
Multiprocessor bus requests (BR1-6)	1	-
Host bus request	2	-
SYSCON effect latency	1	-

Table 13-13. Latencies and Throughput (Cont'd)

Operation	Minimum Data Delay (cycles)	Maximum Throughput (cycles/ transfer)
Host packing status update in SYSTAT register	0	-
DMA packing status update in DMACx register	1	-
DMA chain initialization	7-11	-
Vector interrupt	6	-
Serial ports[1]	35	32
Link ports[1]		
1x CCLK speed	7	4
1/2x CCLK speed	11	8
1/3x CCLK speed	15	12
1/4x CCLK speed	19	16

1 ADSP-21161 processor to ADSP-21161 processor transfers using 32-bit words. Link port throughput is decreased and cycle time increased when the link port clock divisor bits are set in the LCTL register.

The link port control register `LCTL` and the serial port control register `SPCTLx` share the same internal bus for reads and writes. Therefore, when a read of one of these registers followed by a write occurs, the write will require two processor cycles to complete.

A REGISTERS

The ADSP-21161 processor has general-purpose and dedicated registers in each of its functional blocks. The register reference information for each functional block includes bit definitions, initialization values, and memory-mapped addresses (for I/O processor registers). Information on each type of register is available at the following locations:

- "Control and Status System Registers" on page A-2
- "Processing Element Registers" on page A-23
- "Program Sequencer Registers" on page A-25
- "Data Address Generator Registers" on page A-46
- "I/O Processor Registers" on page A-47

When writing programs, it is often necessary to set, clear, or test bits in the processor's registers. While these bit operations can all be done by referring to the bit's location within a register or (for some operations) the register's address with a hexadecimal number, it is much easier to use symbols that correspond to the bit's or register's name. For convenience and consistency, Analog Devices provides a header file that provides these bit and registers definitions. For more information, see the "Register and Bit #Defines (def21161.h)" on page A-121.

Many registers have reserved bits. When writing to a register, programs may only clear (write zero to) the register's reserved bits.

Control and Status System Registers

The processor's control and status system registers determine how the processor core operates and indicate the status of many processor core operations. In the *ADSP-21160 SHARC DSP Instruction Set Reference*, these registers are referred to as System Registers (SREG), which are a subset of the processor's Universal Registers (UREG). Not all registers are valid in all assembly language instructions. In the assembly syntax descriptions, the register group name (UREG, SREG, and others) indicates which type of register is valid within the instruction's context. Table A-1 lists the processor core's control and status registers with their initialization values. Descriptions of each register follow. Other system registers (SREG) are in the I/O processor. For more information, see "I/O Processor Registers" on page A-47.

Table A-1. Control and Status System Registers (SREG and UREG)

Register Name and Page Reference	Initialization After Reset
"Mode Control 1 Register (MODE1)" on page A-3	0x0000 0000[1]
"Mode Mask Register (MMASK)" on page A-8	0x0020 0000
"Mode Control 2 Register (MODE2)" on page A-10	0xXX00 0000[2]
"Arithmetic Status Registers (ASTATx and ASTATy)" on page A-13	0x0000 0000
"Sticky Status Registers (STKYx and STKYy)" on page A-18	0x0540 0000
"User-Defined Status Registers (USTATx)" on page A-22	0x0000 0000

1 MODE 1 register initialization value is 0x0000 0000 for revisions less than 1.0. For revisions greater than or equal to 1.0, the initialization value is 0x0100 0000 because circular buffering (CBUFEN) is enabled.

2 MODE2_SHDW bits 31-25 are the processor ID and silicon revision number, so the initialization value varies with the processor's ID2-0 pins' input and the silicon revision.

Mode Control 1 Register (MODE1)

The Mode Control 1 register is a non-memory-mapped, universal, system register (UREG and SREG). The reset value for this register is 0x0000 0000[1]. Table A-2 and Figure A-1 provide bit information for the `MODE1` register.

Table A-2. Mode Control 1 Register (MODE1) Bit Definitions

Bit(s)	Name	Definition
0	BR8	**Bit Reverse Addressing For Index I8 Enable.** This bit enables (bit reversed if set, =1) or disables (normal if cleared, =0) bit reversed addressing for accesses that are indexed with DAG2 register I8.
1	BR0	**Bit Reverse Addressing For Index I0 Enable.** This bit enables (bit reversed if set, =1) or disables (normal if cleared, =0) bit reversed addressing for accesses that are indexed with DAG1 register I0.
2	SRCU	**Secondary Registers For Computational Units Enable.** This bit enables (use secondary if set, =1) or disables (use primary if cleared, =0) secondary result (MR) registers in the computational units.
3	SRD1H	**Secondary Registers For DAG1 High Enable.** This bit enables (use secondary if set, =1) or disables (use primary if cleared, =0) secondary DAG1 registers for the upper half (I, M, L, B7-4) of the address generator.
4	SRD1L	**Secondary Registers For DAG1 Low Enable.** This bit enables (use secondary if set, =1) or disables (use primary if cleared, =0) secondary DAG1 registers for the lower half (I, M, L, B3-0) of the address generator.
5	SRD2H	**Secondary Registers For DAG2 High Enable.** This bit enables (use secondary if set, =1) or disables (use primary if cleared, =0) secondary DAG2 registers for the upper half (I, M, L, B15-12) of the address generator.

[1] MODE 1 register initialization value is 0x0000 0000 for revisions less than 1.0. For revisions greater than or equal to 1.0, the initialization value is 0x0100 0000 because circular buffering (CBUFEN) is enabled.

Table A-2. Mode Control 1 Register (MODE1) Bit Definitions (Cont'd)

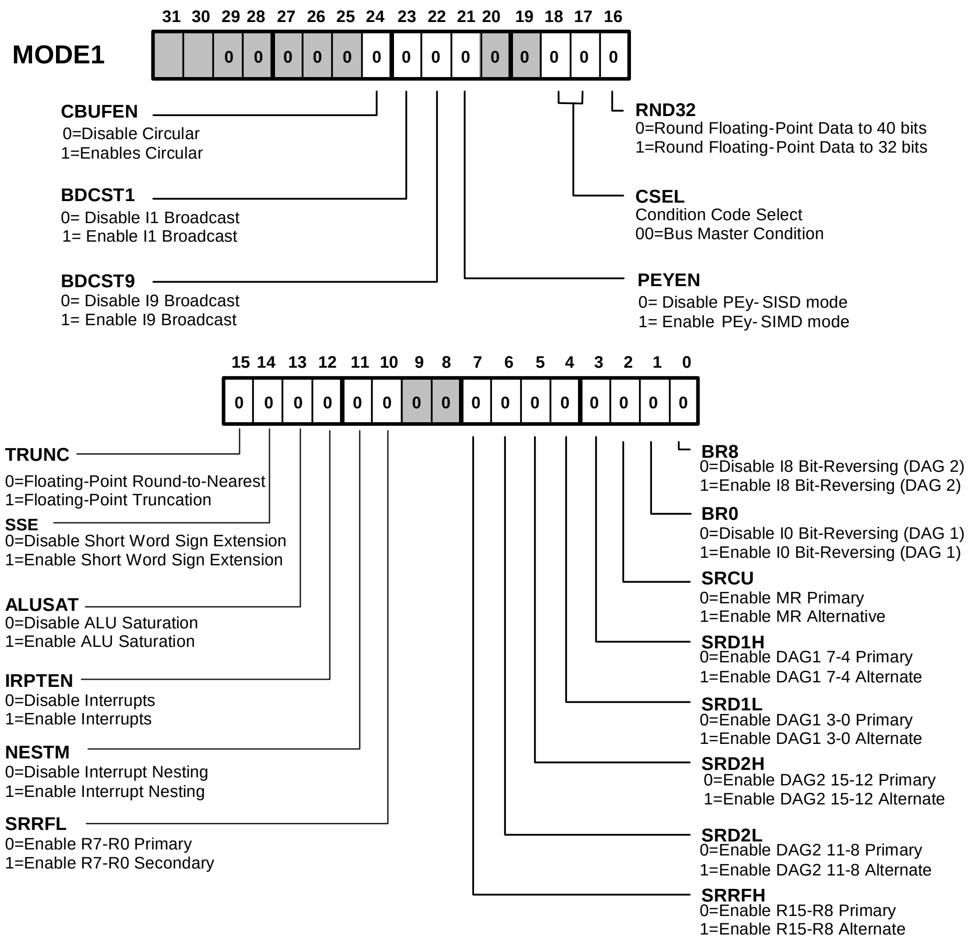
Bit(s)	Name	Definition
6	SRD2L	**Secondary Registers For DAG2 Low Enable.** This bit enables (use secondary if set, =1) or disables (use primary if cleared, =0) secondary DAG2 registers for the lower half (I, M, L, B11-8) of the address generator.
7	SRRFH	**Secondary Registers For Register File High Enable.** This bit enables (use secondary if set, =1) or disables (use primary if cleared, =0) secondary data registers for the upper half (R15-8) of the computational units.
9-8		Reserved
10	SRRFL	**Secondary Registers For Register File Low Enable.** This bit enables (use secondary if set, =1) or disables (use primary if cleared, =0) secondary data registers for the lower half (R7-0) of the computational units.
11	NESTM	**Nesting Multiple Interrupts Enable.** This bit enables (nest if set, =1) or disables (no nesting if cleared, =0) interrupt nesting in the interrupt controller. When interrupt nesting is disabled, a higher priority interrupt can not interrupt a lower priority interrupt's service routine. Other interrupts are latched as they occur, but they are processed after the active routine finishes. When interrupt nesting is enabled, a higher priority interrupt can interrupt a lower priority interrupt's service routine. Lower interrupts are latched as they occur, but they are processed after the nested routines finish.
12	IRPTEN	**Global Interrupt Enable.** This bit enables (if set, =1) or disables (if cleared, =0) all maskable interrupts.
13	ALUSAT	**ALU Saturation Select.** This bit selects whether the computational units saturate results on positive or negative fixed-point overflows (if 1) or return unsaturated results (if 0).
14	SSE	**Fixed-Point Sign Extension Select.** This bit selects whether the computational units sign extend short-word, 16-bit data (if 1) or zero-fill the upper 32 bits (if 0).

Table A-2. Mode Control 1 Register (MODE1) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
15	TRUNC	**Truncation Rounding Mode Select.** This bit selects whether the computational units round results with round-to-zero (if 1) or round-to-nearest (if 0).
16	RND32	**Rounding For 32-bit Floating-Point Data Select.** This bit selects whether the computational units round floating-point data to 32 bits (if 1) or round to 40 bits (if 0).
18-17	CSEL	**Bus Master Code Selection.** These bits indicate whether the processor has control of the external bus as follows: 00=processor is bus master or 01, 10, 11=processor is not bus master.
20-19		Reserved
21	PEYEN	**Processor Element Y Enable.** This bit enables computations in PEy—SIMD mode—(if 1) or disables PEy—SISD mode—(if 0). When set, Processing Element Y (computation units and register files) accepts instruction dispatches. When cleared, Processing Element Y goes into a low power mode.
22	BDCST9	**Broadcast Register Loads Indexed With I9 Enable.** This bit enables (broadcast I9 if set, =1) or disables (no I9 broadcast if cleared, =0) broadcast register loads for loads that use the data address generator I9 index. When the BDCST9 bit is set, data register loads from the PM data bus that use the I9 DAG2 index register are "broadcast" to a register or register pair in each PE.
23	BDCST1	**Broadcast Register Loads Indexed With I1 Enable.** This bit enables (broadcast I1 if set, =1) or disables (no I1 broadcast if cleared, =0) broadcast register loads for loads that use the data address generator I1 index. When the BDCST1 bit is set, data register loads from the DM data bus that use the I1 DAG1 index register are "broadcast" to a register or register pair in each PE.

Table A-2. Mode Control 1 Register (MODE1) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
24	CBUFEN	**Circular Buffer Addressing Enable.** This bit enables (circular if set, =1) or disables (linear if cleared, =0) circular buffer addressing for buffers with loaded I, M, B, and L data address generator register.
31-25		Reserved



MODE1
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0
CBUFEN
0=Disable Circular
1=Enables Circular
BDCST1
0= Disable I1 Broadcast
1= Enable I1 Broadcast
BDCST9
0= Disable I9 Broadcast
1= Enable I9 Broadcast
RND32
0=Round Floating-Point Data to 40 bits
1=Round Floating-Point Data to 32 bits
CSEL
Condition Code Select
00=Bus Master Condition
PEYEN
0= Disable PEy- SISD mode
1= Enable PEy- SIMD mode
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
TRUNC
0=Floating-Point Round-to-Nearest
1=Floating-Point Truncation
SSE
0=Disable Short Word Sign Extension
1=Enable Short Word Sign Extension
ALUSAT
0=Disable ALU Saturation
1=Enable ALU Saturation
IRPTEN
0=Disable Interrupts
1=Enable Interrupts
NESTM
0=Disable Interrupt Nesting
1=Enable Interrupt Nesting
SRRFL
0=Enable R7-R0 Primary
1=Enable R7-R0 Secondary
BR8
0=Disable I8 Bit-Reversing (DAG 2)
1=Enable I8 Bit-Reversing (DAG 2)
BR0
0=Disable I0 Bit-Reversing (DAG 1)
1=Enable I0 Bit-Reversing (DAG 1)
SRCU
0=Enable MR Primary
1=Enable MR Alternative
SRD1H
0=Enable DAG1 7-4 Primary
1=Enable DAG1 7-4 Alternate
SRD1L
0=Enable DAG1 3-0 Primary
1=Enable DAG1 3-0 Alternate
SRD2H
0=Enable DAG2 15-12 Primary
1=Enable DAG2 15-12 Alternate
SRD2L
0=Enable DAG2 11-8 Primary
1=Enable DAG2 11-8 Alternate
SRRFH
0=Enable R15-R8 Primary
1=Enable R15-R8 Alternate

Figure A-1. MODE1 Register

Mode Mask Register (MMASK)

This is a non-memory-mapped, universal, system register (UREG and SREG). The reset value for this register is 0x0020 0000. Each bit in the MMASK register corresponds to a bit in the MODE1 register. Bits that are set in MMASK are used to clear bits in MODE1 when the processor's status stack is pushed. This effectively disables different modes upon servicing an interrupt, or when executing a PUSH STS instruction.

The processor's status stack is pushed in two cases:

1. When you execute a PUSH STS instruction explicitly in your code.
2. When an $\overline{\text{IRQ}}$2-0 timer expires or a VIRPT interrupt occurs.

Example

Before the PUSH STS instruction, MODE1 is set to 0x01202811. This MODE1 value corresponds to the following settings being enabled:

- Bit Reversing for I8
- Secondary Registers for DAG2 (high)
- Interrupt Nesting, ALU Saturation
- Processor Element Y (SIMD)
- Circular Buffering

The MMASK register (Figure A-2) is set to 0x0020 2001 indicating that you want to disable ALU Saturation, SIMD, and bit reversing for I8 after pushing the status stack. The value in MODE1 after PUSH STS is 0x0100 0810. The other settings that were previously in MODE1 remain the same. The only bits that are affected are those that are set both in MMASK and in MODE1. These bits are cleared after the status stack is pushed.

Note also that the reset value of `MMASK` is 0x0020 0000. If you do not make any changes to the `MMASK` register, the default setting automatically disables SIMD when servicing any of the hardware interrupts mentioned above, or during any push of the status stack.

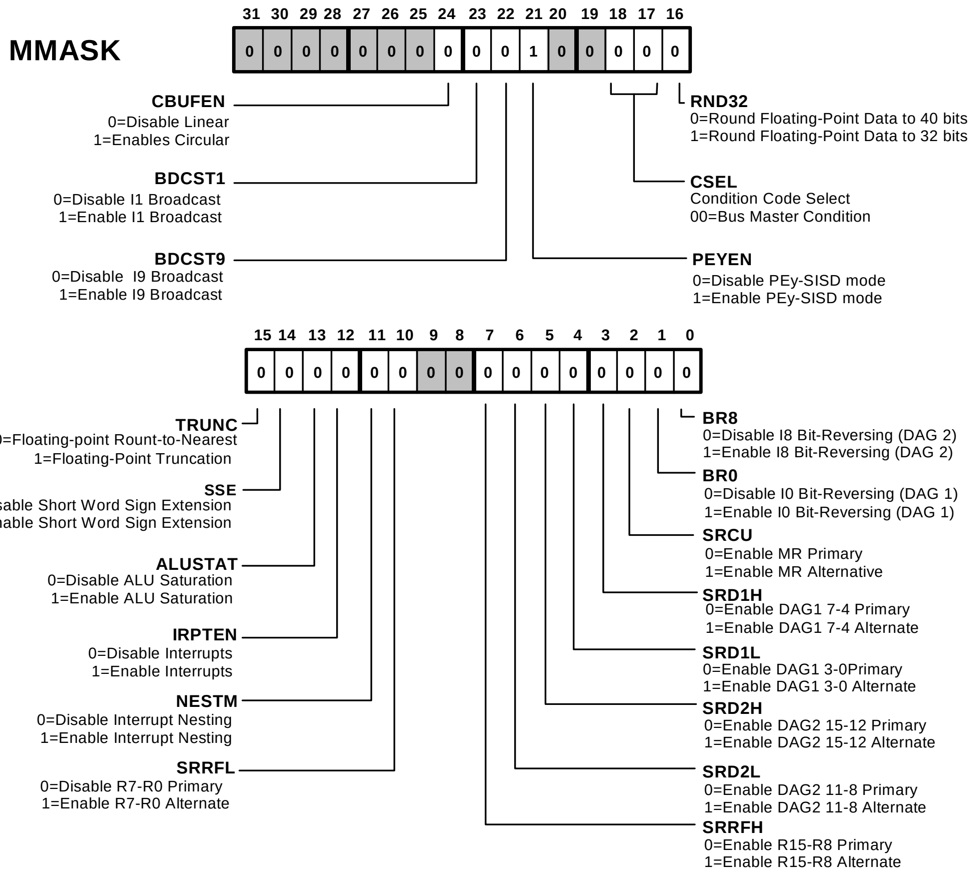


Figure A-2. MMASK Register

Mode Control 2 Register (MODE2)

This is a non-memory-mapped, universal, system register (UREG and SREG). The reset value for this register is 0x0000 0000. Table A-3 and Figure A-3 provide bit information for the MODE2 register.

The MODE2_SHDW register contains silicon revision information in bits 31-25. The corresponding bits in the MODE2 register are reserved. For more information, see "MODE2 Shadow Register (MODE2_SHDW)" on page A-78.

Table A-3. Mode Control 2 Register (MODE2) Bit Definitions

Bit	Name	Definition
0	IRQ0E	$\overline{\text{IRQ0}}$ **Sensitivity Select.** This bit selects sensitivity for $\overline{\text{IRQ0}}$ as edge-sensitive (if set, =1) or level-sensitive (if cleared, =0).
1	IRQ1E	$\overline{\text{IRQ1}}$ **Sensitivity Select.** This bit selects sensitivity for $\overline{\text{IRQ1}}$ as edge-sensitive (if set, =1) or level-sensitive (if cleared, =0).
2	IRQ2E	$\overline{\text{IRQ2}}$ **Sensitivity Select.** This bit selects sensitivity for $\overline{\text{IRQ2}}$ as edge-sensitive (if set, =1) or level-sensitive (if cleared, =0).
3		Reserved
4	CADIS	**Cache Disable.** This bit disables the instruction cache (if set, =1) or enables the cache (if cleared, =0).
5	TIMEN	**Timer Enable.** This bit enables the timer (starts, if set, =1) or disables the timer (stops, if cleared, =0).
6	BUSLK	**Bus Lock Request.** This bit requests bus lock (ADSP-21161 processor retains bus master control, if set, =1) or does not request bus lock (normal bus master control, if cleared, =0).
14-7		Reserved
15	FLG0O	**FLAG0 Output Select.** This bit selects the I/O direction for FLAG0 as an output (if set, =1) or an input (if cleared, =0).
16	FLG1O	**FLAG1 Output Select.** This bit selects the I/O direction for FLAG1 as an output (if set, =1) or an input (if cleared, =0).

Table A-3. Mode Control 2 Register (MODE2) Bit Definitions (Cont'd)

Bit	Name	Definition
17	FLG2O	**FLAG2 Output Select.** This bit selects the I/O direction for FLAG2 as an output (if set, =1) or an input (if cleared, =0).
18	FLG3O	**FLAG3 Output Select.** This bit selects the I/O direction for FLAG3 as an output (if set, =1) or an input (if cleared, =0).
19	CAFRZ	**Cache Freeze.** This bit freezes the instruction cache (retains contents, if set, =1) or thaws the cache (allows new input, if cleared, =0).
20	IIRAE	**Illegal I/O Processor Register Access Enable.** This bit enables detection of I/O processor register accesses (if set, =1) or disables detection of I/O processor register accesses (if cleared, =0). If IIRAE is set, the processor flags an illegal access by setting the IIRA bit in the STKYx register.
21	U64MAE	**Unaligned 64-bit Memory Access Enable.** This bit enables detection of unaligned long word accesses (if set, =1) or disables detection of unaligned long word accesses (if cleared, =0). If U64MAE is set, the processor flags an unaligned long word accesses by setting the U64MA bit in the STKYx register.
31-22		Reserved

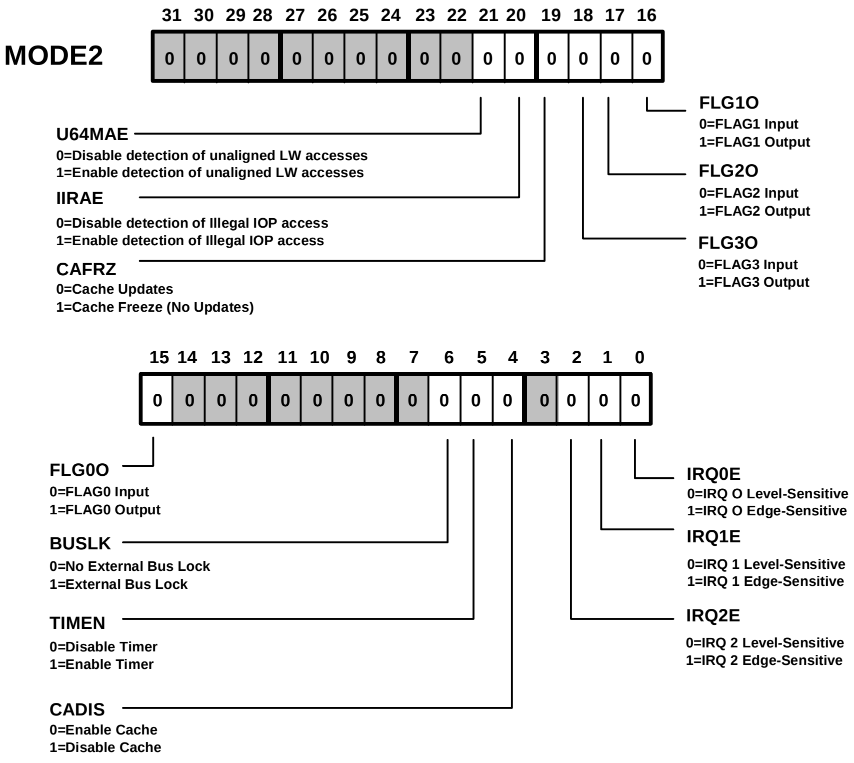


Figure A-3. MODE2 Register

Arithmetic Status Registers (ASTATx and ASTATy)

ASTATx and ASTATy are non-memory-mapped, universal, system registers (UREG and SREG). The reset value for these registers is 0x0000 0000. Each processing element has its own ASTAT register. ASTATx indicates status for PEx operations while ASTATy indicates status for PEy operations. Table A-4 and Figure A-4 provide bit information for the ASTAT register.

If a program loads the ASTATx register manually, there is a one cycle effect latency before the new value in ASTATx can be used in a conditional instruction.

Table A-4. Arithmetic Status Registers (ASTATx/y) Bit Definitions

Bit(s)	Name	Definition
0	AZ	**ALU Zero/Floating-Point Underflow.** Indicates whether the last ALU operation's result was zero (if set, =1) or non-zero (if cleared, =0). The ALU updates AZ for all fixed-point and floating-point ALU operations. AZ can also indicate a floating-point underflow. During an ALU underflow (indicated by a set (=1) AUS bit in the STKYx/y register), the processor sets AZ if the floating-point result is smaller than can be represented in the output format.
1	AV	**ALU Overflow.** Indicates whether the last ALU operation's result overflowed (if set, =1) or did not overflow (if cleared, =0). The ALU updates AV for all fixed-point and floating-point ALU operations. For fixed-point results, the processor sets AV and the AOS bit in the STKYx/y register when the XOR of the two most significant bits is a 1. For floating-point results, the processor sets AV and the AVS bit in the STKYx/y register when the rounded result overflows (unbiased exponent > 127).
2	AN	**ALU Negative.** Indicates whether the last ALU operation's result was negative (if set, =1) or positive (if cleared, =0). The ALU updates AN for all fixed-point and floating-point ALU operations.

Table A-4. Arithmetic Status Registers (ASTATx/y) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
3	AC	**ALU Fixed-Point Carry.** Indicates whether the last ALU operation had a carry out of most significant bit of the result (if set, =1) or had no carry (if cleared, =0). The ALU updates AC for all fixed-point operations. The processor clears AC during fixed-point logic operations: PASS, MIN, MAX, COMP, ABS, and CLIP. The ALU reads the AC flag for fixed-point accumulate operations: addition with carry and fixed-point subtraction with carry.
4	AS	**ALU X-Input Sign (for ABS and MANT).** Indicates whether the last ALU ABS or MANT operation's input was negative (if set, =1) or positive (if cleared, =0). The ALU updates AS only for fixed-point and floating-point ABS and the MANT operations. The ALU clears AS for all operations other than ABS and MANT.
5	AI	**ALU Floating-Point Invalid Operation.** Indicates whether the last ALU operation's input was invalid (if set, =1) or valid (if cleared, =0). The ALU updates AI for all fixed-point and floating-point ALU operations. The processor sets AI and the AIS bit in the STKYx/y register if the ALU operation: • Receives a NAN input operand • Adds opposite-signed infinities • Subtracts like-signed infinities • Overflows during a floating-point to fixed-point conversion when saturation mode is not set • Operates on an infinity when the saturation mode is not set
6	MN	**Multiplier Negative.** Indicates whether the last multiplier operation's result was negative (if set, =1) or positive (if cleared, =0). The multiplier updates MN for all fixed-point and floating-point multiplier operations.

Table A-4. Arithmetic Status Registers (ASTATx/y) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
7	MV	**Multiplier Overflow.** Indicates whether the last multiplier operation's result overflowed (if set, =1) or did not overflow (if cleared, =0). The multiplier updates MV for all fixed-point and floating-point multiplier operations. For floating-point results, the processor sets MV and the MVS bit in the STKYx/y register if the rounded result overflows (unbiased exponent > 127). For fixed-point results, the processor sets MV and the MOS bit in the STKYx/y register if the result of the multiplier operation is: • Twos-complement, fractional with the upper 17 bits of MR not all zeros or all ones • Twos-complement, integer with the upper 49 bits of MR not all zeros or all ones • Unsigned, fractional with the upper 16 bits of MR not all zeros • Unsigned, integer with the upper 48 bits of MR not all zeros If the multiplier operation directs a fixed-point result to an MR register, the processor places the overflowed portion of the result in MR1 and MR2 for an integer result or places it in MR2 only for a fractional result.

Table A-4. Arithmetic Status Registers (ASTATx/y) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
8	MU	**Multiplier Floating-Point Underflow.** Indicates whether the last multiplier operation's result underflowed (if set, =1) or did not underflow (if cleared, =0). The multiplier updates MU for all fixed-point and floating-point multiplier operations. For floating-point results, the processor sets MU and the MUS bit in the STKYx/y register if the floating-point result underflows (unbiased exponent < –126). Denormal operands are treated as Zeros, therefore they never cause underflows. For fixed-point results, the processor sets MU and the MUS bit in the STKYx/y register if the result of the multiplier operation is: • Twos-complement, fractional: upper 48 bits all zeros or all ones, lower 32 bits not all zeros • Unsigned, fractional: upper 48 bits all zeros, lower 32 bits not all zeros If the multiplier operation directs a fixed-point, fractional result to an MR register, the processor places the underflowed portion of the result in MR0.
9	MI	**Multiplier Floating-Point Invalid Operation.** Indicates whether the last multiplier operation's input was invalid (if set, =1) or valid (if cleared, =0). The multiplier updates MI for floating-point multiplier operations. The processor sets MI and the MIS bit in the STKYx/y register if the ALU operation: • Receives a NAN input operand • Receives an Infinity and Zero as input operands
10	AF	**ALU Floating-Point Operation.** Indicates whether the last ALU operation was floating-point (if set, =1) or fixed-point (if cleared, =0). The ALU updates AF for all fixed-point and floating-point ALU operations.

Table A-4. Arithmetic Status Registers (ASTATx/y) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
11	SV	**Shifter Overflow.** Indicates whether the last shifter operation's result overflowed (if set, =1) or did not overflow (if cleared, =0). The shifter updates SV for all shifter operations. The processor sets SV if the shifter operation: • Shifts the significant bits to the left of the 32-bit fixed-point field • Tests, sets, or clears a bit outside of the 32-bit fixed-point field • Extracts a field that is past or crosses the left edge of the 32-bit fixed-point field • Performs a LEFTZ or LEFTO operation that returns a result of 32
12	SZ	**Shifter Zero.** Indicates whether the last shifter operation's result was zero (if set, =1) or non-zero (if cleared, =0). The shifter updates SZ for all shifter operations. The processor also sets SZ if the shifter operation performs a bit test on a bit outside of the 32-bit fixed-point field.
13	SS	**Shifter Input Sign.** Indicates whether the last shifter operation's input was negative (if set, =1) or positive (if cleared, =0). The shifter updates SS for all shifter operations.
17-14		Reserved
18	BTF	**Bit Test Flag for System Registers.** Indicates whether the system register bit is true (if set, =1) or false (if cleared, =0). The processor sets BTF when the bit(s) in a system register and value in the Bit Tst instruction match. The processor also sets BTF when the bit(s) in a system register and value in the Bit Xor instruction match.
23-19		Reserved
31-24	CACC	**Compare Accumulation Shift Register.** Bit 31 of CACC indicates which operand was greater during the last ALU compare operation: X input (if set, =1) or Y input (if cleared, =0). The other seven bits in CACC form a right-shift register, each storing a previous compare accumulation result. With each new compare, the processor right shifts the values of CACC, storing the newest value in bit 31 and the oldest value in bit 24.

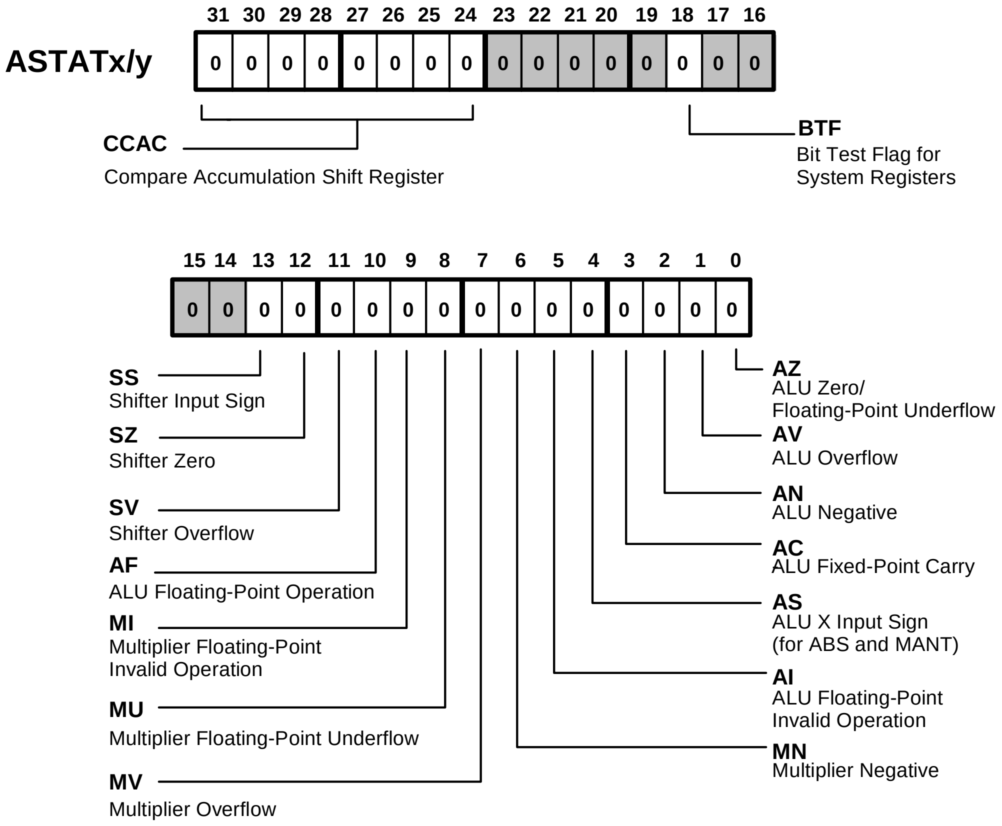


Figure A-4. ASTAT Register

Sticky Status Registers (STKYx and STKYy)

These are non-memory-mapped, universal, system registers (UREG and SREG). The reset value for these registers is 0x0000 0000. Each processing element has its own `STKY` register. The `STKYx` register indicates status for PEx operations and some program sequencer stacks. The `STKYy` register only indicates status for PEy operations. Table A-5, Figure A-5 and Figure A-6 lists bits for both STKYx and STKYy, noting with an √ the bits that apply only to STKYx.

STKY bits do not clear themselves after the condition they flag is no longer true. They remain "sticky" until cleared by the program.

The ADSP-21161 processor sets a STKY bit in response to a condition. For example, the processor sets the AUS bit in the STKY register when an ALU underflow set AZ in the ASTAT register. The processor clears AZ if the next ALU operation does not cause an underflow. The bit AUS remains set until a program clears the STKY bit. Interrupt service routines should clear their interrupt's corresponding STKY bit so the processor can detect a re-occurrence of the condition. For example, an interrupt service routine for the floating-point underflow exception interrupt (FLTUI) would clear the AUS bit in the STKY register near the beginning of the routine.

Table A-5. Sticky Status Registers (STKYx/y) Bit Definitions

Bit(s)	Name	Definition At right: √ shows bits in both STKYx/y × shows bits in STKYx only	↓ ↓
0	AUS	**ALU Floating-Point Underflow.** A sticky indicator for the ALU AS bit. For more information, see "AZ" on page A-13.	√
1	AVS	**ALU Floating-Point Overflow.** A sticky indicator for the ALU AV bit. For more information, see "AV" on page A-13.	√
2	AOS	**ALU Fixed-Point Overflow.** A sticky indicator for the ALU AV bit. For more information, see "AV" on page A-13.	√
4-3		Reserved	
5	AIS	**ALU Floating-Point Invalid Operation.** A sticky indicator for the ALU AI bit. For more information, see "AI" on page A-14.	√
6	MOS	**Multiplier Fixed-Point Overflow.** A sticky indicator for the multiplier MV bit. For more information, see "MV" on page A-15.	√
7	MVS	**Multiplier Floating-Point Overflow.** A sticky indicator for the multiplier MV bit. For more information, see "MV" on page A-15.	√

Table A-5. Sticky Status Registers (STKYx/y) Bit Definitions (Cont'd)

Bit(s)	Name	Definition At right: √ shows bits in both STKYx/y × shows bits in STKYx only	↓ ↓
8	MUS	**Multiplier Floating-Point Underflow.** A sticky indicator for the multiplier MU bit. For more information, see "MU" on page A-16.	√
9	MIS	**Multiplier Floating-Point Invalid Operation.** A sticky indicator for the multiplier MI bit. For more information, see "MI" on page A-16.	√
16-10		Reserved	
17	CB7S	**DAG1 Circular Buffer 7 Overflow.** Indicates whether a circular buffer being addressed with DAG1 register I7 has overflowed (if set, =1) or has not overflowed (if cleared, =0). A circular buffer overflow occurs when DAG circular buffering operation increments the I register past the end of buffer.	×
18	CB15S	**DAG2 Circular Buffer 15 Overflow.** Indicates whether a circular buffer being addressed with DAG2 register I15 has overflowed (if set, =1) or has not overflowed (if cleared, =0). A circular buffer overflow occurs when DAG circular buffering operation increments the I register past the end of buffer.	×
19	IIRA	**Illegal IOP Register Access.** Indicates if set (=1) whether a core, host, or multiprocessor access to I/O processor registers has occurred or has not occurred (if 0).	×
20	U64MA	**Unaligned 64-Bit Memory Access.** Indicates if set (=1) whether a Normal word access with the LW mnemonic addressing an uneven memory address has occurred or has not occurred (if 0).	×
21	PCFL	**PC Stack Full.** Indicates whether the PC stack is full (if 1) or not full (if 0)—Not a sticky bit, cleared by a Pop.	×
22	PCEM	**PC Stack Empty.** Indicates whether the PC stack is empty (if 1) or not empty (if 0)—Not sticky, cleared by a Push.	×
23	SSOV	**Status Stack Overflow.** Indicates whether the status stack is overflowed (if 1) or not overflowed (if 0)—A sticky bit.	×
24	SSEM	**Status Stack Empty.** Indicates whether the status stack is empty (if 1) or not empty (if 0)—Not sticky, cleared by a Push.	×

Table A-5. Sticky Status Registers (STKYx/y) Bit Definitions (Cont'd)

Bit(s)	Name	Definition At right: √ shows bits in both STKYx/y × shows bits in STKYx only	↓ ↓
25	LSOV	**Loop Stack Overflow.** Indicates whether the loop counter stack and loop stack are overflowed (if 1) or not overflowed (if 0)—A sticky bit.	×
26	LSEM	**Loop Stack Empty.** Indicates whether the loop counter stack and loop stack are empty (if 1) or not empty (if 0)—Not sticky, cleared by a Push.	×
31-27		Reserved	

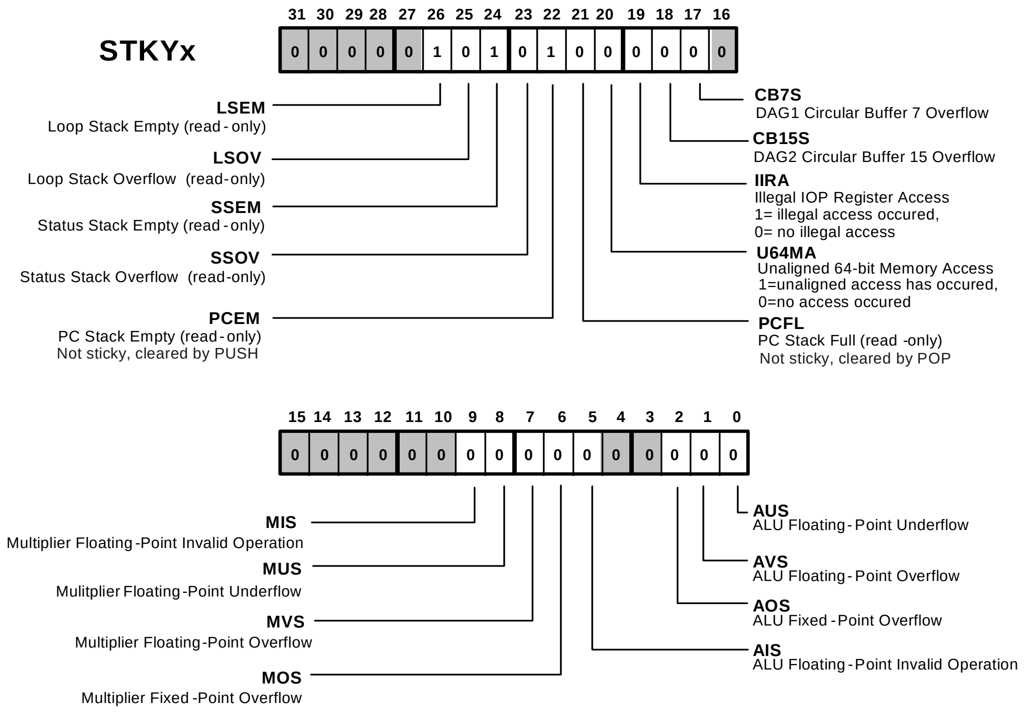

STKYx
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0
LSEM
Loop Stack Empty (read-only)
LSOV
Loop Stack Overflow (read-only)
SSEM
Status Stack Empty (read-only)
SSOV
Status Stack Overflow (read-only)
PCEM
PC Stack Empty (read-only)
Not sticky, cleared by PUSH
CB7S
DAG1 Circular Buffer 7 Overflow
CB15S
DAG2 Circular Buffer 15 Overflow
IIRA
Illegal IOP Register Access
1= illegal access occured,
0= no illegal access
U64MA
Unaligned 64-bit Memory Access
1=unaligned access has occured,
0=no access occured
PCFL
PC Stack Full (read -only)
Not sticky, cleared by POP
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
MIS
Multiplier Floating-Point Invalid Operation
MUS
Mulitplier Floating-Point Underflow
MVS
Multiplier Floating-Point Overflow
MOS
Multiplier Fixed-Point Overflow
AUS
ALU Floating-Point Underflow
AVS
ALU Floating-Point Overflow
AOS
ALU Fixed-Point Overflow
AIS
ALU Floating-Point Invalid Operation

Figure A-5. STKYx Registers

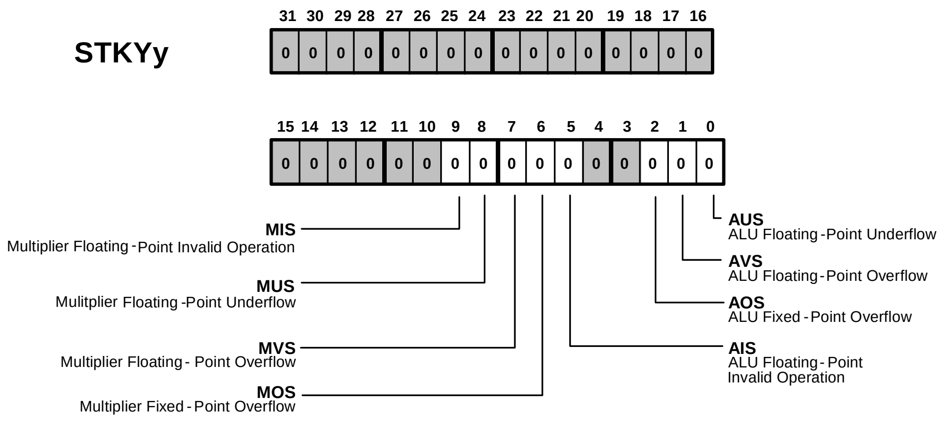

STKYy
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
MIS
Multiplier Floating-Point Invalid Operation
MUS
Mulitplier Floating-Point Underflow
MVS
Multiplier Floating- Point Overflow
MOS
Multiplier Fixed-Point Overflow
AUS
ALU Floating-Point Underflow
AVS
ALU Floating-Point Overflow
AOS
ALU Fixed-Point Overflow
AIS
ALU Floating-Point Invalid Operation

Figure A-6. STKYy Register

User-Defined Status Registers (USTATx)

These are non-memory-mapped, universal, system registers (UREG and SREG). The reset value for these registers is 0x0000 0000. The USTATx registers (Figure A-7) are user-defined, general-purpose status registers. Programs can use these 32-bit registers with bitwise instructions (SET, CLEAR, TEST, and others). Often, programs use these registers for low-overhead, general-purpose flags or for temporary 32-bit storage of data.

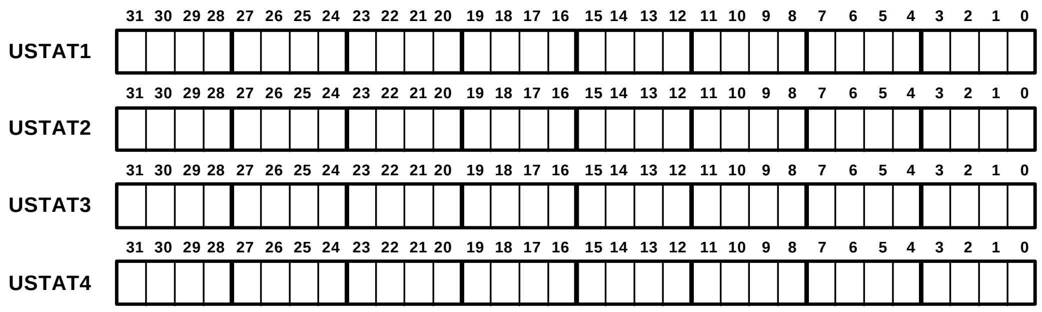


Figure A-7. USTAT Registers

Processing Element Registers

Except for the PX register, the processor's processing element registers store data for each element's ALU, multiplier, and shifter. The inputs and outputs for processing element operations go through these registers. The PX register lets programs transfer data between the data buses, but cannot be an input or output in a calculation.

Table A-6. Processing Element Universal Registers (UREG)

Register Name and Page Reference	Initialization After Reset
"Data File Data Registers (Rx, Fx, Sx)" on page A-23	Undefined
"Multiplier Results Registers (MRFx, MRBx)" on page A-24	Undefined
"Program Memory Bus Exchange Register (PX)" on page A-25	Undefined

Data File Data Registers (Rx, Fx, Sx)

The Data File Data Registers are non-memory-mapped, universal, data registers (UREG and DREG). Each of the ADSP-21161's processing elements has a data register file—a set of 40-bit data registers that transfer data between the data buses and the computation units. These registers also provide local storage for operands and results.

The R, F, and S prefixes on register names do not effect the 32-bit or 40-bit data transfer; the naming convention determines how the ALU, multiplier, and shifter treat the data and determines which processing element's data registers are being used.

For more information on how to use these registers, see "Data Register File" on page 2-30.

Multiplier Results Registers (MRFx, MRBx)

The `MRFx` and `MRBx` registers are non-memory-mapped, universal, data registers (UREG and DREG). Each of the processor's multipliers has a primary or foreground (`MRF`) register and alternate or background (`MRB`) results register. Fixed-point operations place 80-bit results in the multiplier's foreground `MRF` register or background `MRB` register, depending on which is active. For more information on selecting the result register, see "Alternate (Secondary) Data Registers" on page 2-32. For more information on result register fields, see "Data Register File" on page 2-30.

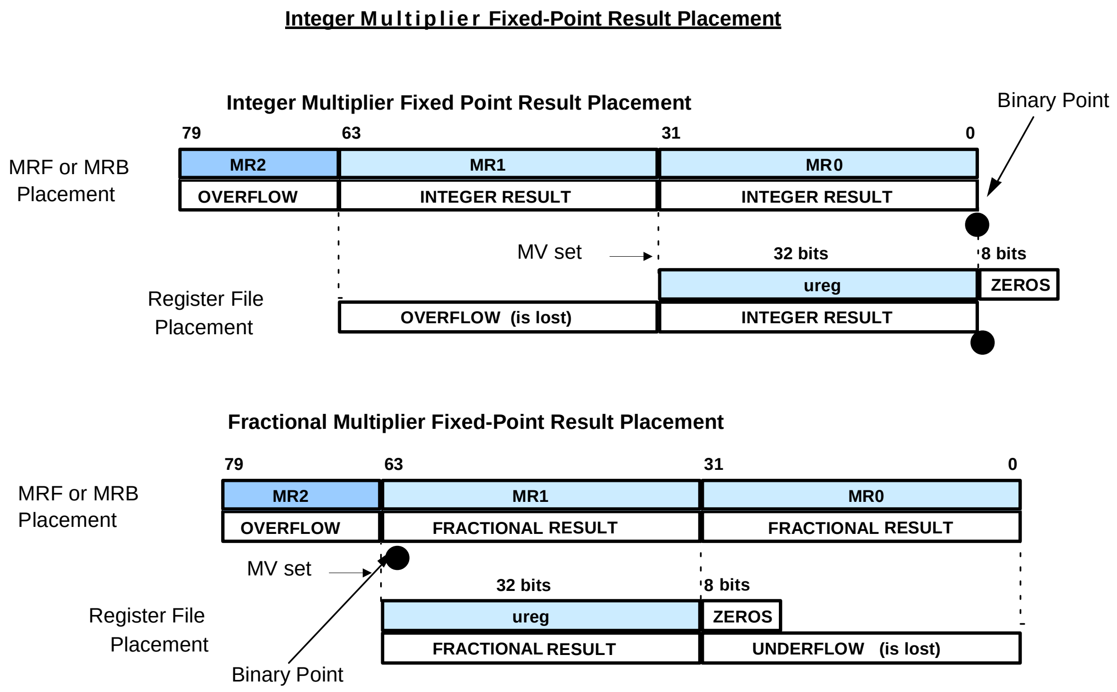


Figure A-8. MRFx and MRBx Registers

Program Memory Bus Exchange Register (PX)

The PX register (Figure A-9) is a non-memory-mapped, universal registers (UREG only). The PM Bus Exchange (PX) register permits data to flow between the PM and DM data buses. The PX register can work as one 64-bit register or as two 32-bit registers (PX1 and PX2). PX1 is the lower 32 bits of the PX register and PX2 is the upper 32 bits of PX. See the section "Internal Data Bus Exchange" on page 5-10 for more information about the PX register.

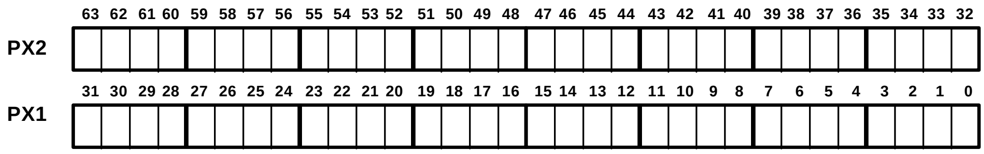


Figure A-9. PX Register

Program Sequencer Registers

The ADSP-21161 processor's program sequencer registers direct the execution of instructions. These registers include support for the:

- Instruction pipeline
- Program and loop stacks
- Timer
- Interrupt mask and latch

Table A-7. Program Sequencer System Registers (UREG and SREG)

Register	Initialization After Reset
"Interrupt Latch Register (IRPTL)" on page A-27	0x0000 0000 (cleared)
"Interrupt Mask Register (IMASK)" on page A-31	0x0000 0003
"Interrupt Mask Pointer Register (IMASKP)" on page A-32	0x0000 0000 (cleared)
"Link Port Interrupt Register (LIRPTL)" on page A-34	0x0000 0000 (cleared)
"Flag Value Register (FLAGS)" on page A-37	0x0000 000n[1]

1 FLAGS bits 0-3 are equal to the values of the FLAG0-3 input pins after reset; the flag pins are configured as inputs after reset.

Table A-8. Program Sequencer Universal Registers (UREG only)

Register	Initialization After Reset
"Program Counter Register (PC)" on page A-41	Undefined
"Program Counter Stack Register (PCSTK)" on page A-44	Undefined
"Program Counter Stack Pointer Register (PCSTKP)" on page A-44	Undefined
"Fetch Address Register (FADDR)" on page A-44	Undefined
"Decode Address Register (DADDR)" on page A-44	Undefined
"Loop Address Stack Register (LADDR)" on page A-45	Undefined
"Current Loop Counter Register (CURLCNTR)" on page A-45	Undefined
"Loop Counter Register (LCNTR)" on page A-45	Undefined
"Timer Period Register (TPERIOD)" on page A-46	Undefined
"Timer Count Register (TCOUNT)" on page A-46	Undefined

Interrupt Latch Register (IRPTL)

The `IRPTL` register is a non-memory-mapped, universal, system register (UREG and SREG). The reset value for this register is 0x0000 0000. The `IRPTL` register indicates latch status for interrupts. Table A-9 and Figure A-10 provide bit definitions for the `IRPTL` register.

Table A-9. Interrupt Latch Register (IRPTL) Bit Definitions

Bit(s)	Name	Definition
0	EMUI	**Emulator Interrupt.** Indicates whether an EMUI is latched and is pending (if set, =1) or no EMUI is pending (if cleared, =0). An EMUI occurs on reset and when an external device asserts the $\overline{\text{EMU}}$ pin.
1	RSTI	**Reset Interrupt.** Indicates whether an RSTI is latched and is pending (if set, =1) or no RSTI is pending (if cleared, =0). An RSTI occurs on reset as an external device asserts the $\overline{\text{RESET}}$ pin.
2	IICDI	**Illegal Input Condition Detected Interrupt.** Indicates whether an IICDI is latched and is pending (if set, =1) or no IICDI is pending (if cleared, =0). An IICDI occurs when a TRUE results from the logical Oring of the Illegal I/O Processor Register Access (IIRA) and Unaligned 64-bit Memory Access bits in the STKYx registers.
3	SOVFI	**Stack Overflow/Full Interrupt.** Indicates whether a SOVFI is latched and is pending (if set, =1) or no SOVFI is pending (if cleared, =0). An SOVFI occurs when a stack in the program sequencer overflows or is full. For more information, see "PCFL" on page A-20, "SSOV" on page A-20, and "LSOV" on page A-21.

Table A-9. Interrupt Latch Register (IRPTL) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
4	TMZHI	**Timer Expired High Priority.** Indicates whether a TMZHI is latched and is pending (if set, =1) or TMZHI is not pending (if cleared, =0). A TMZHI occurs when the timer decrements to zero. Note that this event also triggers a TMZLI. The timer operations are controlled as follows: • The TCOUNT register contains the timer counter. The timer decrements the TCOUNT register each clock cycle. • The TPERIOD value specifies the frequency of timer interrupts. The number of cycles between interrupts is TPERIOD + 1. The maximum value of TPERIOD is $2^{32} - 1$. • The TIMEN bit in the MODE2 register starts and stops the timer. Since the timer expired event (TCOUNT decrements to zero) generates two interrupts, TMZHI and TMZLI, programs should unmask the timer interrupt with the desired priority and leave the other one masked.
5	VIRPTI	**Multiprocessor Vector Interrupt.** Indicates whether a VIRPTI is latched and is pending (if set, =1) or no VIRPTI is pending (if cleared, =0). A VIRPTI occurs when one of the DSPs in a multiprocessor system writes an address (the vector) to the processor's VIRPT register.
6	IRQ2I	$\overline{\text{IRQ2}}$ **Hardware Interrupt.** Indicates whether an IRQ2I is latched and is pending (if set, =1) or no IRQ2I is pending (if cleared, =0). An IRQ2I occurs when an external device asserts the $\overline{\text{IRQ2}}$ pin.
7	IRQ1I	$\overline{\text{IRQ1}}$ **Hardware Interrupt.** Indicates whether an IRQ1I is latched and is pending (if set, =1) or no IRQ1I is pending (if cleared, =0). An IRQ1I occurs when an external device asserts the $\overline{\text{IRQ1}}$ pin.
8	IRQ0I	$\overline{\text{IRQ0}}$ **Hardware Interrupt.** Indicates whether an IRQ0I is latched and is pending (if set, =1) or no IRQ0I is pending (if cleared, =0). An IRQ0I occurs when an external device asserts the $\overline{\text{IRQ0}}$ pin.
9		Reserved

Table A-9. Interrupt Latch Register (IRPTL) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
10	SP0I	**SPORT0 DMA Interrupt.** Indicates whether a SP0I is latched and is pending (if set, =1) or no SP0I is pending (if cleared, =0). A SP0I occurs two cycles after the last bit of an input the serial word is latched into RX0A or RX0B *or* two cycles after data is shifted out of TX0A or TX0B.
11	SP1I	**SPORT1 DMA Interrupt.** Indicates whether a SP1I is latched and is pending (if set, =1) or no SP1I is pending (if cleared, =0). A SP1I occurs two cycles after the last bit of an input the serial word is latched into RX1A or RX1B *or* two cycles after data is shifted out of TX1A or TX1B.
12	SP2I	**SPORT2 DMA Interrupt.** Indicates whether a SP2I is latched and is pending (if set, =1) or no SP2I is pending (if cleared, =0). An SP2I occurs two cycles after the last bit of an output the serial word is latched from RX2A or RX2B *or* two cycles after data is shifted out of TX2A or TX2B.
13	SP3I	**SPORT3 DMA Interrupt.** Indicates whether a SP3I is latched and is pending (if set, =1) or no SP3I is pending (if cleared, =0). An SP3I occurs two cycles after the last bit of an output the serial word is latched from RX3A or RX3B *or* two cycles after data is shifted out of TX3A or TX3B.
14	LPISUMI	**Link or SPI Buffer DMA Summary Interrupt.** Indicates whether an LPISUMI is latched and is pending (if set, =1) or no LPISUMI is pending (if cleared, =0). An LPISUMI occurs when a TRUE results from the logical Or'ing of unmasked link port and SPI interrupts, which are configured in the LIRPTL register. Indicates whether at least one unmasked link port (LBUF0 or LBUF1) or SPI port (SPIRX or SPITX) interrupt is latched. To enable link or SPI interrupts this bit must be unmasked in addition to unmasking the individual interrupts.

Table A-9. Interrupt Latch Register (IRPTL) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
15	EP0I	**External Port Buffer 0 DMA Interrupt.** Indicates whether an EP0I is latched and is pending (if set, =1) or no EP0I is pending (if cleared, =0). An EP0I occurs when the external port buffer's DMA is disabled (DEN=0) and either: • The buffer is set to receive (TRAN=0), and the buffer is not empty • The buffer is set to transmit (TRAN=1), and the buffer is not full
16	EP1I	**External Port Buffer 1 DMA Interrupt.** Indicates whether an EP1I is latched and is pending (if set, =1) or no EP1I is pending (if cleared, =0). For more information, see "EP0I" on page A-30.
17	EP2I	**External Port Buffer 2 DMA Interrupt.** Indicates whether an EP2I is latched and is pending (if set, =1) or no EP2I is pending (if cleared, =0). For more information, see "EP0I" on page A-30.
18	EP3I	**External Port Buffer 3 DMA Interrupt.** Indicates whether an EP3I is latched and is pending (if set, =1) or no EP3I is pending (if cleared, =0). For more information, see "EP0I" on page A-30.
19	LSRQI	**Link Port Service Request Interrupt.** Indicates whether an LSRQI is latched and is pending (if set, =1) or no LSRQI is pending (if cleared, =0). An LSRQI occurs when an external source accesses an unassigned link port or accesses an assigned link port that has its link buffer disabled.
20	CB7I	**DAG1 Circular Buffer 7 Overflow Interrupt.** Indicates whether a CB7I is latched and is pending (if set, =1) or no CB7I interrupt is pending (if cleared, =0). For more information, see "CB7S" on page A-20.
21	CB15I	**DAG2 Circular Buffer 15 Overflow Interrupt.** Indicates whether a CB15I is latched and is pending (if set, =1) or no CB15I is pending (if cleared, =0). For more information, see "CB15S" on page A-20.
22	TMZLI	**Timer Expired (Low Priority) Interrupt.** Indicates whether a TMZLI is latched and is pending (if set, =1) or no TMZLI is pending (if cleared, =0). For more information, see "TMZHI" on page A-28.

Table A-9. Interrupt Latch Register (IRPTL) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
23	FIXI	**Fixed-Point Overflow Interrupt.** Indicates whether a FIXI is latched and is pending (if set, =1) or no FIXI is pending (if cleared, =0). For more information, see "AOS" on page A-19.
24	FLTOI	**Floating-Point Overflow Interrupt.** Indicates whether a FLTOI is latched and is pending (if set, =1) or no FLTOI is pending (if cleared, =0).
25	FLTUI	**Floating-Point Underflow Interrupt.** Indicates whether a FLTUI is latched and is pending (if set, =1) or no FLTUI is pending (if cleared, =0).
26	FLTII	**Floating-Point Invalid Operation Interrupt.** Indicates whether a FLTII is latched and is pending (if set, =1) or no FLTII is pending (if cleared, =0). For more information, see "AIS" on page A-19.
27	SFT0I	**User Software Interrupt 0.** Indicates whether a SFT0I is latched and is pending (if set, =1) or no SFT0I is pending (if cleared, =0). An SFT0I occurs when a program sets (=1) this bit.
28	SFT1I	**User Software Interrupt 1.** Indicates whether a SFT1I is latched and is pending (if set, =1) or no SFT1I is pending (if cleared, =0). For details, see SFT0I bit description.
29	SFT2I	**User Software Interrupt 2.** Indicates whether a SFT2I is latched and is pending (if set, =1) or no SFT2I is pending (if cleared, =0). For details, see SFT0I bit description.
30	SFT3I	**User Software Interrupt 3.** Indicates whether a SFT3I is latched and is pending (if set, =1) or no SFT3I is pending (if cleared, =0). For details, see SFT0I bit description.
31		Reserved

Interrupt Mask Register (IMASK)

The `IMASK` register is a non-memory-mapped, universal, system register (UREG and SREG). The reset value for this register is 0x0000 0003. Each bit in the `IMASK` register corresponds to a bit with the same name in the

IRPTL registers. The bits in IMASK unmask (enable if set, =1) or mask (disable if cleared, =0) the interrupts that are latched in the IRPTL register. Except for $\overline{\text{RESET}}$, all interrupts are maskable.

When IMASK masks an interrupt, the masking disables the processor's response to the interrupt. The IRPTL register still latches an interrupt even when masked, and the processor responds to that latched interrupt if it is later unmasked. Table A-10 and Figure A-10 provide bit definitions for the IMASK register.

Interrupt Mask Pointer Register (IMASKP)

The IMASKP register is a non-memory-mapped, universal, system register (UREG and SREG). The reset value for this register is 0x0000 0000. Each bit in the IMASKP register corresponds to a bit with the same name in the IRPTL registers. This register supports an interrupt nesting scheme that lets higher priority events interrupt an interrupt service routine and keeps lower priority events from interrupting. Table A-9 and Figure A-10 provide bit definitions for the IMASKP register.

When interrupt nesting is enabled, the bits in the IMASKP register mask interrupts having lower priority than the interrupt that is currently being serviced. Other bits in this register unmask interrupts having higher priority than the interrupt that is currently being serviced. Interrupt nesting is enabled using NESTM in the MODE1 register. The IRPTL register latches a lower priority interrupt even when masked, and the processor responds to that latched interrupt if it is later unmasked.

When interrupt nesting is disabled (NESTM=0 in the MODE1 register), the bits in IMASKP mask all interrupts while an interrupt is currently being serviced. The IRPTL register still latches these interrupts even when masked, and the processor responds to the highest priority latched interrupt after servicing the current interrupt.

For more information, see "NESTM" on page A-4.

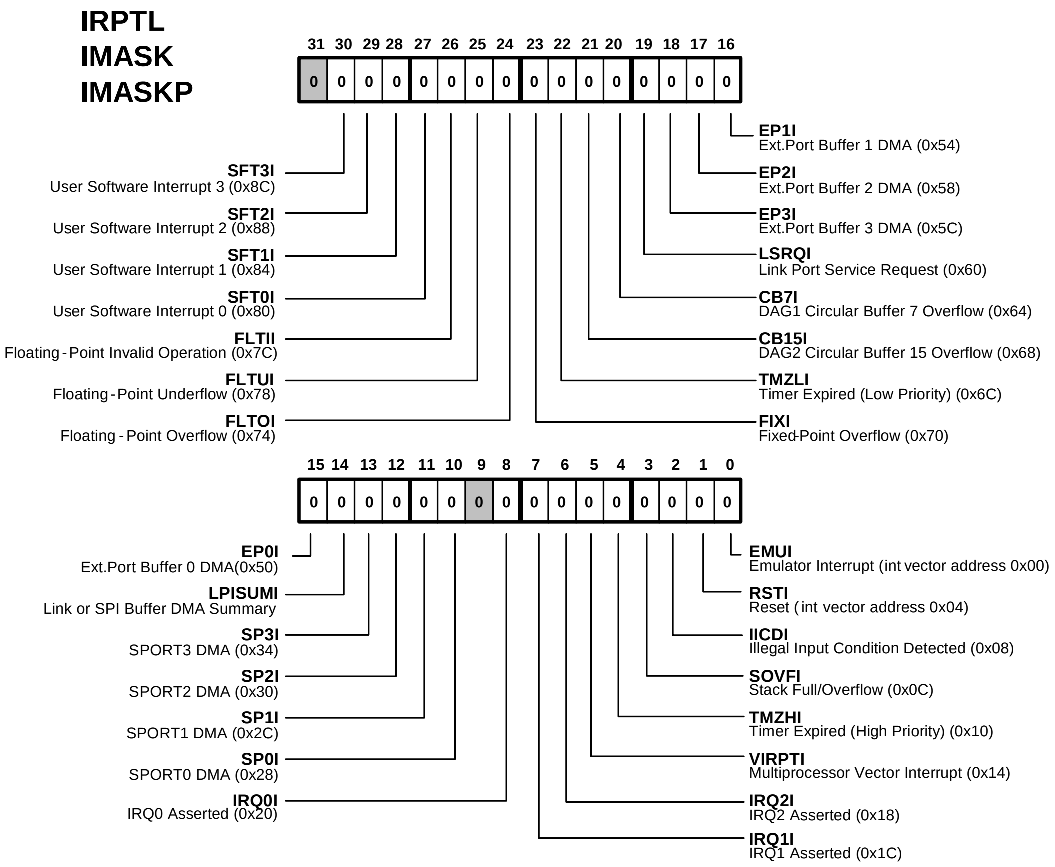

IRPTL
IMASK
IMASKP
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
EP1I
Ext.Port Buffer 1 DMA (0x54)
SFT3I
User Software Interrupt 3 (0x8C)
EP2I
Ext.Port Buffer 2 DMA (0x58)
SFT2I
User Software Interrupt 2 (0x88)
EP3I
Ext.Port Buffer 3 DMA (0x5C)
SFT1I
User Software Interrupt 1 (0x84)
LSRQI
Link Port Service Request (0x60)
SFT0I
User Software Interrupt 0 (0x80)
CB7I
DAG1 Circular Buffer 7 Overflow (0x64)
FLTII
Floating-Point Invalid Operation (0x7C)
CB15I
DAG2 Circular Buffer 15 Overflow (0x68)
FLTUI
Floating-Point Underflow (0x78)
TMZLI
Timer Expired (Low Priority) (0x6C)
FLTOI
Floating-Point Overflow (0x74)
FIXI
Fixed-Point Overflow (0x70)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
EP0I
Ext.Port Buffer 0 DMA(0x50)
EMUI
Emulator Interrupt (int vector address 0x00)
LPISUMI
Link or SPI Buffer DMA Summary
RSTI
Reset (int vector address 0x04)
SP3I
SPORT3 DMA (0x34)
IICDI
Illegal Input Condition Detected (0x08)
SP2I
SPORT2 DMA (0x30)
SOVFI
Stack Full/Overflow (0x0C)
SP1I
SPORT1 DMA (0x2C)
TMZHI
Timer Expired (High Priority) (0x10)
SP0I
SPORT0 DMA (0x28)
VIRPTI
Multiprocessor Vector Interrupt (0x14)
IRQ0I
IRQ0 Asserted (0x20)
IRQ2I
IRQ2 Asserted (0x18)
IRQ1I
IRQ1 Asserted (0x1C)

Figure A-10. IMASK, IMASKP, IRPTL Registers

Link Port Interrupt Register (LIRPTL)

The LIRPTL register is a non-memory-mapped, universal, system register (UREG and SREG). The reset value for these registers is 0x0000 0000. The LIRPTL register indicates latch status, select masking, and displays mask pointers for link port interrupts. Table A-10 and Figure A-11 provide bit definitions for the LIRPTL register.

Note that the LPISUM bit in the IRPTL register contains a logical Oring of the link port and SPI port latch bits in the LIRPTL register. For more information, see "LPISUMI" on page A-29.

Table A-10. Link Port Interrupt Latch, Mask, and Mask Pointer Register (LIRPTL) Bit Definitions

Bit	Name	Definition
0	LP0I	**Link Port Buffer 0 DMA Interrupt.** Indicates whether an LP0 interrupt is latched and is pending (if set, =1) or no LP0 interrupt is pending (if cleared, =0). An LP0 interrupt occurs when the link port buffer's DMA is disabled (DEN=0) and either: • The buffer is set to receive (TRAN=0), and the buffer is not empty • The buffer is set to transmit (TRAN=1), and the buffer is not full Note that LP0 is set irrespective of whether the link port is enabled in core mode or DMA mode.
1	LP1I	**Link Port Buffer 1 DMA Interrupt.** Indicates whether an LP1 interrupt is latched and is pending (if set, =1) or no LP1 interrupt is pending (if cleared, =0).
2	SPIRI	**SPI Receive DMA Interrupt Latch.** Indicates whether an SPIRI is latched and is pending (if set, =1) or no SPIRI is pending (if cleared, =0).
3	SPITI	**SPI Transmit DMA Interrupt Latch.** Indicates whether an SPITI is latched and is pending (if set, =1) or no SPITI is pending (if cleared, =0).

Table A-10. Link Port Interrupt Latch, Mask, and Mask Pointer Register (LIRPTL) Bit Definitions (Cont'd)

Bit	Name	Definition
15-4		Reserved
16	LP0MSK	**Link Buffer 0 DMA Interrupt Mask.** This bit unmasks the LP0 interrupt (if set, =1) or masks the LP0 interrupt (if cleared, =0). For more information on how interrupt masking works, see "Interrupt Latch Register (IRPTL)" on page A-27.
17	LP1MSK	**Link Buffer 1 DMA Interrupt Mask.** This bit unmasks the LP1 interrupt (if set, =1) or masks the LP1 interrupt (if cleared, =0). For more information on how interrupt masking works, see "Interrupt Latch Register (IRPTL)" on page A-27.
18	SPIRMSK	**SPI Receive DMA Interrupt Mask.** This bit unmasks the SPIR interrupt (if set, =1) or masks the SPIR interrupt (if cleared, =0). For more information on how interrupt masking works, see "Interrupt Latch Register (IRPTL)" on page A-27.
19	SPITMSK	**SPI Transmit DMA Interrupt Mask.** This bit unmasks the SPIT interrupt (if set, =1) or masks the SPIT interrupt (if cleared, =0). For more information on how interrupt masking works, see "Interrupt Latch Register (IRPTL)" on page A-27.
23-20		Reserved
24	LP0MSKP	**Link Buffer 0 DMA Interrupt Mask Pointer.** When the ADSP-21161 processor is servicing another interrupt, indicates whether the LP0 interrupt is unmasked (if set, =1) or the LP0 interrupt is masked (if cleared, =0). For more information on how interrupt mask pointers works, see "Interrupt Mask Pointer Register (IMASKP)" on page A-32.
25	LP1MSKP	**Link Buffer 1 DMA Interrupt Mask Pointer.** When the processor is servicing another interrupt, this bit indicates whether the LP1 interrupt is unmasked (if set, =1) or the LP1 interrupt is masked (if cleared, =0). For more information on how interrupt mask pointers works, see "Interrupt Mask Pointer Register (IMASKP)" on page A-32.

Table A-10. Link Port Interrupt Latch, Mask, and Mask Pointer Register (LIRPTL) Bit Definitions (Cont'd)

Bit	Name	Definition
26	SPIRMSKP	**SPI Receive DMA Interrupt Mask Pointer.** When the processor is servicing another interrupt, this bit indicates whether the SPIR interrupt is unmasked (if set, =1) or the SPIR interrupt is masked (if cleared, =0). For more information on how interrupt mask pointers works, see "Interrupt Mask Pointer Register (IMASKP)" on page A-32.
27	SPITMSKP	**SPI Transmit DMA Interrupt Mask Pointer.** When the processor is servicing another interrupt, this bit indicates whether the SPIT interrupt is unmasked (if set, =1) or the SPIT interrupt is masked (if cleared, =0). For more information on how interrupt mask pointers works, see "Interrupt Mask Pointer Register (IMASKP)" on page A-32.
31-28		Reserved

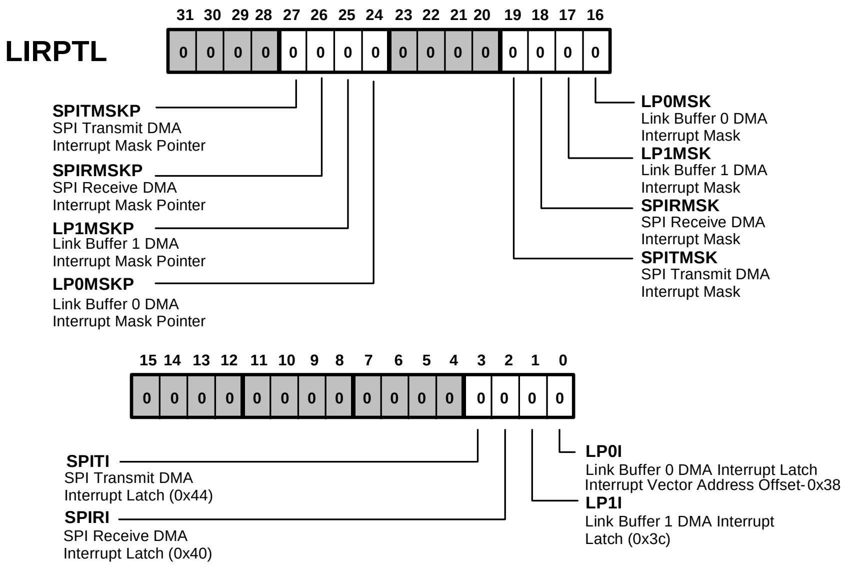

LIRPTL
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
SPITMSKP
SPI Transmit DMA Interrupt Mask Pointer
SPIRMSKP
SPI Receive DMA Interrupt Mask Pointer
LP1MSKP
Link Buffer 1 DMA Interrupt Mask Pointer
LP0MSKP
Link Buffer 0 DMA Interrupt Mask Pointer
LP0MSK
Link Buffer 0 DMA Interrupt Mask
LP1MSK
Link Buffer 1 DMA Interrupt Mask
SPIRMSK
SPI Receive DMA Interrupt Mask
SPITMSK
SPI Transmit DMA Interrupt Mask
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
SPITI
SPI Transmit DMA Interrupt Latch (0x44)
SPIRI
SPI Receive DMA Interrupt Latch (0x40)
LP0I
Link Buffer 0 DMA Interrupt Latch Interrupt Vector Address Offset-0x38
LP1I
Link Buffer 1 DMA Interrupt Latch (0x3c)

Figure A-11. LIRPTL Register

Flag Value Register (FLAGS)

The FLAGS register is a non-memory-mapped, universal, system register (UREG and SREG). The reset value for these registers is 0x0000 0000. The FLAGS register indicates the state of the FLG[3:0] pins. When a FLG[3:0] pin is an output, the processor outputs a high in response to a program setting the pin's bit in FLAGS. The I/O direction (input or output) selection of each bit is controlled by its FLG[3:0] bit in the MODE2 register. For more information, see "FLG0O" on page A-10. The FLAG register bit definitions are given in Table A-11 and Figure A-12.

Table A-11. FLAGS Register (FLAGS) Bit Definitions

Bit	Name	Definition
0	FLG0	**FLAG0 Value.** Indicates the state of the FLAG0 pin, whether the pin is high (if set, =1) or low (if cleared, =0).
1	FLG1	**FLAG1 Value.** Indicates the state of the FLAG1 pin, whether the pin is high (if set, =1) or low (if cleared, =0).
2	FLG2	**FLAG2 Value.** Indicates the state of the FLAG2 pin, whether the pin is high (if set, =1) or low (if cleared, =0).
3	FLG3	**FLAG3 Value.** Indicates the state of the FLAG3 pin, whether the pin is high (if set, =1) or low (if cleared, =0).
31-4		Reserved

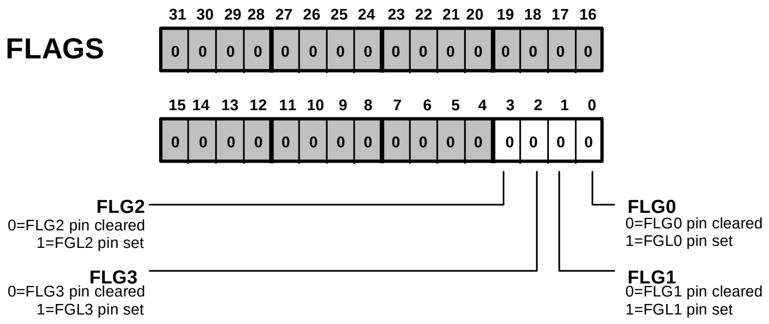


Figure A-12. FLAGS Register

IOFLAG Value Register

The IOFLAG register (Figure A-13 and Table A-12) is a memory-mapped, IO processor register. The reset value for this register is 0x0000 0000. The IOFLAG register indicates status and control information for the FLG pins. When a FLG[11:4] pin is an output, the processor outputs a high when a program sets the pin's bit in IOFLAG.

Bits 7-0 of the IOFLAG register reflect the status of FLG[11:4] pins and bits 15-8 control the direction (input or output) of these flags. A value of 0 programs the flag as an input while a value of 1 programs it as an output. You cannot directly execute bit wise operations such as BIT TST or BIT CLR on these flags. However, it is possible to execute these operations indirectly by writing to system registers such as USTAT1, USTAT2, USTAT3 or USTAT4.

Table A-12. IOFLAG Register (IOFLAG) Bit Definitions

Bit	Name	Definition
0	FLG4	**FLAG4 Value.** Indicates the state of the FLAG4 pin, whether the pin is high (if set, =1) or low (if cleared, =0).
1	FLG5	**FLAG5 Value.** Indicates the state of the FLAG5 pin, whether the pin is high (if set, =1) or low (if cleared, =0).
2	FLG6	**FLAG6 Value.** Indicates the state of the FLAG6 pin, whether the pin is high (if set, =1) or low (if cleared, =0).
3	FLG7	**FLAG7 Value.** Indicates the state of the FLAG7 pin, whether the pin is high (if set, =1) or low (if cleared, =0).
4	FLG8	**FLAG8 Value.** Indicates the state of the FLAG8 pin, whether the pin is high (if set, =1) or low (if cleared, =0).
5	FLG9	**FLAG9 Value.** Indicates the state of the FLAG9 pin, whether the pin is high (if set, =1) or low (if cleared, =0).
6	FLG10	**FLAG10 Value.** Indicates the state of the FLAG10 pin, whether the pin is high (if set, =1) or low (if cleared, =0).
7	FLG11	**FLAG11 Value.** Indicates the state of the FLAG11 pin, whether the pin is high (if set, =1) or low (if cleared, =0).
8	FLG4O	**FLAG4 Output Select.** This bit selects the I/O direction for the FLAG4 pin; the flag is programmed as an output (if set, =1) or input (if cleared, =0).
9	FLG5O	**FLAG5 Output Select.** This bit selects the I/O direction for the FLAG5 pin; the flag is programmed as an output (if set, =1) or input (if cleared, =0).
10	FLG6O	**FLAG6 Output Select.** This bit selects the I/O direction for the FLAG6 pin; the flag is programmed as an output (if set, =1) or input (if cleared, =0).
11	FLG7O	**FLAG7 Output Select.** This bit selects the I/O direction for the FLAG7 pin; the flag is programmed as an output (if set, =1) or input (if cleared, =0).

Table A-12. IOFLAG Register (IOFLAG) Bit Definitions (Cont'd)

Bit	Name	Definition
12	FLG8O	**FLAG8 Output Select.** This bit selects the I/O direction for the FLAG8, the flag is programmed as an output (if set, =1) or input (if cleared, =0).
13	FLG9O	**FLAG9 Output Select.** This bit selects the I/O direction for the FLAG9, the flag is programmed as an output (if set, =1) or input (if cleared, =0).
14	FLG10O	**FLAG10 Output Select.** This bit selects the I/O direction for the FLAG10, the flag is programmed as an output (if set, =1) or input (if cleared, =0).
15	FLG11O	**FLAG11 Output Select.** This bit selects the I/O direction for the FLAG11pin, the flag is programmed as an output (if set, =1) or input (if cleared, =0).
31-16		Reserved

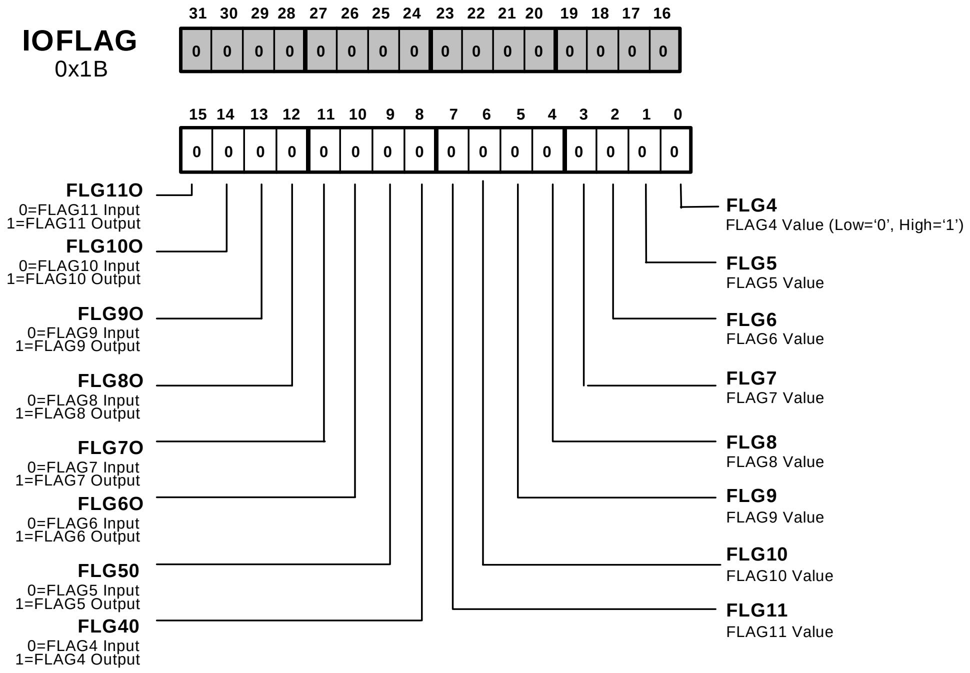


Figure A-13. IOFLAG Register

Program Counter Register (PC)

The PC register is a non-memory-mapped, universal register (UREG only). The Program Counter register is the last stage in the fetch-decode-execute instruction pipeline and contains the 24-bit address of the instruction that the ADSP-21161 processor executes on the next cycle. The PC couples with the Program Counter Stack, PCSTK, which stores return addresses and top-of-loop addresses. All addresses generated by the sequencer are 24-bit program memory instruction addresses. The amount of addressable space is 62.5 Mwords for Non-SDRAM and 254 Mwords for SDRAM.

As shown in Figure A-14, the address buses can handle 32-bit addresses, but the program sequencer only generates 24-bit addresses over the PM bus. Since the sequencer generates 24-bit addresses, sequencing is limited to the low 64 Mwords of the processor's 254 Mword memory map.

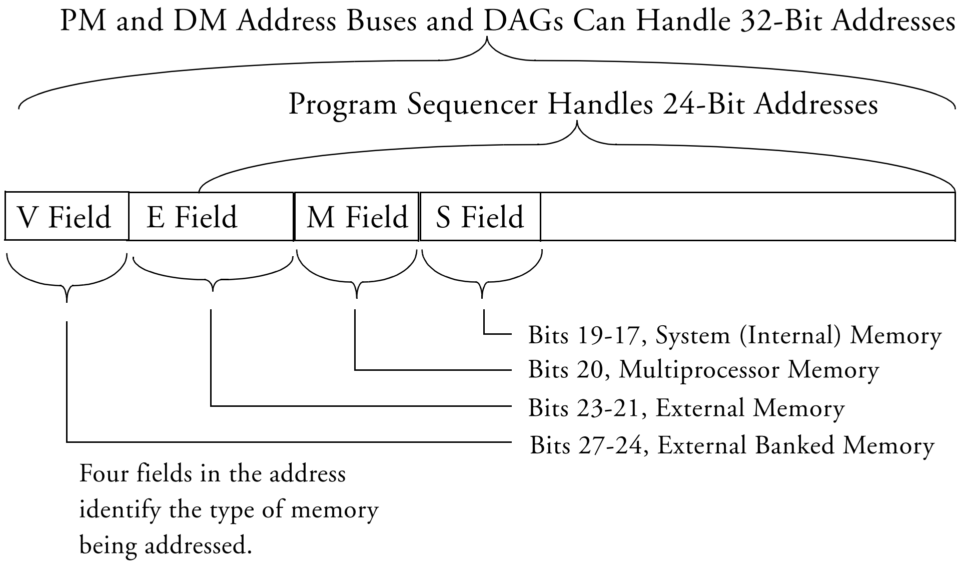

PM and DM Address Buses and DAGs Can Handle 32-Bit Addresses
Program Sequencer Handles 24-Bit Addresses
V Field
E Field
M Field
S Field
Bits 19-17, System (Internal) Memory
Bits 20, Multiprocessor Memory
Bits 23-21, External Memory
Bits 27-24, External Banked Memory
Four fields in the address identify the type of memory being addressed.

Figure A-14. PM and DM Bus Addresses Versus Sequencing Addresses

Table A-13 describes the three fields that appear in Figure A-14. The content of the External (E), Multiprocessor (M), and System (S) fields in the address route the data or instruction access to the memory space.

Table A-13. PM and DM Address Bus E, M, and S Fields

Bit Field	Description	
E	External Address — Values in this field have the following meaning:	
	all zeros	The address is in the IOP registers of another ADSP-21161 processor (M and S activated)
	non-zero	The address is in external memory; with the E bits active, remaining bits [20-0] are a valid address
M	Multiprocessor — Values in this field have the following meaning:	
	non-zero	ID of another ADSP-21161 processor
	1	Write to IOP register of an ADSP-21161 processor. This field is only set for accesses between ADSP-21161 processors.
	0	Address in the processor's own internal memory
S	System — Values in this field have the following meaning:	
	000	Address of an IOP register
	001	Address in Long Word Addressing space
	01x	Address in Normal Word Addressing space
	1xx	Address in Short Word Addressing space
V	Virtual — Values in this field have the following meaning:	
	00	Depends on E, S1-0, and M bits; address corresponds to locals internal or external (bank 0) memory or to remote processor's IOP space.
	01	External memory bank 1, local processor
	10	External memory bank 2, local processor
	11	External memory bank 3, local processor

Program Counter Stack Register (PCSTK)

This is a non-memory-mapped, universal register (UREG only). The Program Counter Stack register contains the address of the top of the PC stack. This register is a readable and writable register.

Program Counter Stack Pointer Register (PCSTKP)

The PCSTKP register is a non-memory-mapped, universal register (UREG only). The Program Counter Stack Pointer register contains the value of PCSTKP. This value is given as follows: 0 when the PC stack is empty, 1...30 when the stack contains data, and 31 when the stack overflows. This register is readable and writable. A write to PCSTKP takes effect after a one-cycle delay. If the PC stack is overflowed, a write to PCSTKP has no effect.

Fetch Address Register (FADDR)

The FADDR register is a non-memory-mapped, universal register (UREG only). The Fetch Address register is the first stage in the fetch-decode-execute instruction pipeline and contains the 24-bit address of the instruction that the processor fetches from memory on the next cycle.

Decode Address Register (DADDR)

The DADDR register is a non-memory-mapped, universal register (UREG only). The Decode Address register is the second stage in the fetch-decode-execute instruction pipeline and contains the 24-bit address of the instruction that the processor decodes on the next cycle.

Loop Address Stack Register (LADDR)

The LADDR register is a non-memory-mapped, universal register (UREG only). The Loop Address Stack is six levels deep by 32 bits wide. The 32-bit word of each level consists of a 24-bit loop termination address, a 5-bit termination code, and a 2-bit loop type code.

Table A-14. Loop Address Stack Register (LADDR)

Bits	Value
0-23	Lloop termination address
24-28	Termination code
29	Reserved (always reads 0)
30-31	Loop type code 00 = arithmetic condition-based (not LCE) 01 = counter-based, length 1 10 = counter-based, length 2 11 = counter-based, length > 2

Current Loop Counter Register (CURLCNTR)

The CURLCNTR register is a non-memory-mapped, universal register (UREG only). The Current Loop Counter register provides access to the loop counter stack and tracks iterations for the DO UNTIL LCE loop being executed. For more information on how to use CURLCNTR, see "Loop Counter Stack" on page 3-30.

Loop Counter Register (LCNTR)

The LCNTR register is a non-memory-mapped, universal register (UREG only). The Loop Counter register provides access to the loop counter stack and holds the count value before the DO UNTIL LCE loop is executed. For more information on how to use LCNTR, see "Loop Counter Stack" on page 3-30.

Timer Period Register (TPERIOD)

The `TPERIOD` register is a non-memory-mapped, universal register (UREG only). The Timer Period register contains the decrementing timer count value, counting down the cycles between timer interrupts. For more information on how to use the timer, see “Timer and Sequencing” on page 3-50.

Timer Count Register (TCOUNT)

The `TCOUNT` register is a non-memory-mapped, universal register (UREG only). The Timer Count register contains the timer period, indicating the number of cycles between timer interrupts. For more information on how to use the timer, see “Timer and Sequencing” on page 3-50.

Data Address Generator Registers

The ADSP-21161 processor’s Data Address Generator (DAG) registers hold data addresses, modify values, and circular buffer configurations. Using these registers, the DAGs can automatically increment addressing for ranges of data locations (a buffer).

Table A-15. Data Address Generator Universal Registers (UREG only)

Register	Initialization After Reset
“Index Registers (Ix)” on page A-47	Undefined
“Modify Registers (Mx)” on page A-47	Undefined
“Length and Base Registers (Lx,Bx)” on page A-47	Undefined

Index Registers (Ix)

The Ix registers are non-memory-mapped, universal registers (UREG only). The Data Address Generators store addresses in Index registers (I0-I7 for DAG1 and I8-I15 for DAG2). An index register holds an address and acts as a pointer to a memory location. For more information, see "Data Address Generator" on page 4-1.

Modify Registers (Mx)

The Mx register are non-memory-mapped, universal registers (UREG only). The Data Address Generators update stored addresses using Modify registers (M0-M7 for DAG1 and M8-M15 for DAG2). A modify register provides the increment or step size by which an index register is pre- or post-modified during a register move. For more information, see "Data Address Generator" on page 4-1.

Length and Base Registers (Lx,Bx)

The Lx and Bx registers are non-memory-mapped, universal registers (UREG only). The Data Address Generators control circular buffering operations with Length and Base registers (L0-L7 and B0-B7 for DAG1 and L8-L15 and B8-B15 for DAG2). Length and base registers setup the range of addresses and the starting address for a circular buffer. For more information, see "Data Address Generator" on page 4-1.

I/O Processor Registers

The I/O processor's registers are accessible as part of the processor's memory map. Table A-17 on page A-51 lists the I/O processor's memory-mapped registers in address order and provides a cross reference to a description of each register. These registers occupy addresses 0x0000

0000 through 0x0000 01FF of the memory map. The I/O registers control the following operations: External port DMA, Link port DMA, Serial port DMA and SPI port DMA.

I/O processor registers have a one cycle effect latency (changes take effect on the second cycle after the change).

Since the I/O processor's registers are part of the processor's memory map, buses access these registers as locations in memory. While these registers act as memory-mapped locations, they are separate from the processor's internal memory and have different bus access. One bus can access one I/O processor register from one I/O processor register group at a time. Table A-16 lists the I/O processor register groups.

When there is contention among the buses for access to registers in the same I/O processor register group, the processor arbitrates register access as follows:

- External Port (EP) bus accesses (highest priority)
- Data Memory (DM) bus accesses
- Program Memory (PM) bus accesses
- I/O processor (I/O) bus (lowest priority) accesses
- DMA parameter register or `DMASTAT` register conflicts
  There is a one cycle DMA stall if an access to a DMA parameter register or the `DMASTAT` register conflicts with DMA address generation. For example, one cycle stall occurs when writing to a DMA register while a register update is taking place. Similarly, a one cycle stall occurs when reading from a DMA register while DMA chaining is taking place.

The bus with highest priority gets access to the I/O processor register group, and the other buses are held off from accessing that I/O processor register group until that access been completed.

There is one exception to this access contention rule. The I/O bus and EP bus can simultaneously access the EP (External Port) group of registers, allowing DMA transfers to internal memory at full speed.

Table A-16. I/O Processor Register Groups

Register Group	I/O Processor Registers In This Group
System Control (SC) Registers	SYSCON, VIRPT, WAIT, SYSTAT, MSGR0, MSGR1, MSGR2, MSGR3, MSGR4, MSGR5, MSGR6, MSGR7, BMAX, BCNT, PC_SHDW, IOFLAG, MODE2_SHDW, DMASTAT
DMA Address (DA) Registers	II0A, II0B, IM0A, IM0B, C0A, C0B, CP0A, CP0B, GP0A, GP0B, II1A, II1B, IM1A, IM1B, C1A, C1B, CP1A, CP1B, GP1A, GP1B, II2A, II2B, IM2A, IM2B, C2A, C2B, CP2A, CP2B, GP2A, GP2B, II3A, II3B, IM3A, IM3B, C3A, C3B, CP3A, CP3B, GP3A, GP3B, IILB0 (IISRX), IMLB0 (IMSRX), CLB0 (CSRX), GPLB0 (GPSRX), IILB1(IISTX), IMLB1 (IMSTX), CLB1, GPLB0 (GPSTX), IIEP0, IMEP0, CEP0, CPEP0, GPEP0, EIEP0, EMEP0, ECEP0, IIEP1, IMEP1, CEP1, CPEP1, GPEP1, EIEP1, EMEP1, ECEP1, IIEP2, IMEP2, CEP2, CPEP2, GPEP2, EIEP2, EMEP2, ECEP2, IIEP3, IMEP3, CEP3, CPEP3, GPEP3, EIEP3, EMEP3, ECEP3, EI13, EM13, EC13
External Port (EP) Registers	EPB0, EPB1, EPB2, EPB3, DMAC10,DMAC11,DMAC12,DMAC13

Table A-16. I/O Processor Register Groups (Cont'd)

Register Group	I/O Processor Registers In This Group
SDRAM Controller (SD)	SDCTL, SDRDIV
Link, SPI & Serial Port (LSP) Registers	LBUF0, LBUF1, LCTL, LSRQ SPIRX, SPITX, SPICTL, SPISTAT RX0A, RX0B, TX0A, TX0B, SPCTL0, DIV0, CNT0, MR0CS0, MR0CCS0, MR0CS1, MR0CCS1, MR0CS2, MR0CCS2, MR0CS3, MR0CCS3 RX1A, RX1B, TX1A, TX1B, SPCTL1, DIV1, CNT1, MT1CS0, MT1CCS0, MT1CS1, MT1CCS1, MT1CS2, MT1CCS2, MT1CS3, MT1CCS3 RX2A, RX2B, TX2A, TX2B, SPCTL2, DIV2, CNT2, MR2CS0, MR2CCS0, MR2CS1, MR2CCS1, MR2CS2, MR2CCS2, MR2CS3, MR2CCS3 RX3A, RX3B, TX3A, TX3B, SPCTL3, DIV3, CNT3, MT3CS0, MT3CCS0, MT3CS1, MT3CCS1, MT3CS2, MT3CCS2, MT3CS3, MT3CCS3 SP02MCTL, SP13MCTL

Since the I/O processor registers are memory-mapped, the ADSP-21161 processor's architecture does not allow programs to directly transfer data between these registers and other memory locations, except as part of a DMA operation. To read or write I/O processor registers, programs must use the processor core registers. The following example code shows a value being transferred from memory to the `USTAT1` register, then the value is transferred to the I/O processor `WAIT` registers.

```
USTAT2= 0x108421; /* 1st instr. to be executed after reset */
DM(WAIT)=USTAT2;   /* Set external memory waitstates to 0 */
```

The register names for I/O processor registers are not part of the processor's assembly syntax. To ease access to these registers, programs should use the `#include` command to incorporate a file containing the registers' symbolic names and addresses. An example `#include` file appears in the "Register and Bit #Defines (def21161.h)" on page A-121.

Table A-17. I/O Processor Registers Memory Map

Register Address	Register Name	Initialization After Reset	Register Group	Reference
0x000	SYSCON	0x0001 0020	SC	page A-60
0x001	VIRPT	0x0004 0014	SC	page A-63
0x002	WAIT	0x01ce 739c	SC	page A-65
0x003	SYSTAT	0x000n 0nn0	SC	page A-69
0x004	EPB0	ni	EP	page A-76
0x006	EPB1	ni	EP	page A-76
0x008	MSGR0	ni	SC	page A-77
0x009	MSGR1	ni	SC	page A-77
0x00A	MSGR2	ni	SC	page A-77
0x00B	MSGR3	ni	SC	page A-77
0x00C	MSGR4	ni	SC	page A-77
0x00D	MSGR5	ni	SC	page A-77
0x00E	MSGR6	ni	SC	page A-77
0x00F	MSGR7	ni	SC	page A-77
0x010	PC_SHDW	ni	SC	page A-77
0x011	MODE2_ SHDW	0xnn00 0000	SC	page A-78
0x014	EPB2	ni	EP	page A-76

Notes: An "ni" in the Initialization column indicates that the register is Not Initialized. For information on Register Groups, see Table A-16 on page A-49.

Table A-17. I/O Processor Registers Memory Map (Cont'd)

Register Address	Register Name	Initialization After Reset	Register Group	Reference
0x016	EPB3	ni	EP	page A-76
0x018	BMAX	0x0000 0000	SC	page A-79
0x019	BCNT	0x0000 0000	SC	page A-79
0x01B	IOFLAG	0x0000 0000	SC	page A-38
0x01C	DMAC10	ni[1]	EP	page A-80
0x01D	DMAC11	0x0000 0000	EP	page A-80
0x01E	DMAC12	0x0000 0000	EP	page A-80
0x01F	DMAC13	0x0000 0000	EP	page A-80
0x030	IILB0/IISRX	ni	DA	-
0x031	IMLB0/IMSRX	ni	DA	-
0x032	CLB0/CSRX	ni	DA	-
0x033	CPLB0	ni	DA	-
0x034	GPLB0/GPSRX	ni	DA	-
0x037	DMASTAT	ni	SC	page A-90
0x038	IILB1/IISTX	ni	DA	-
0x039	IMLB1/IMSTX	ni	DA	-
0x03A	CLB1/CSTX	ni	DA	-
0x03B	CPLB1	ni	DA	-
0x03C	GPLB1/GPSTX	ni	DA	-
0x040	IIEP0	ni[1]	DA	-
0x041	IMEP0	ni[1]	DA	-
0x042	CEP0	ni[1]	DA	-

Notes: An "ni" in the Initialization column indicates that the register is Not Initialized. For information on Register Groups, see Table A-16 on page A-49.

Table A-17. I/O Processor Registers Memory Map (Cont'd)

Register Address	Register Name	Initialization After Reset	Register Group	Reference
0x043	CPEP0	ni[1]	DA	-
0x044	GPEP0	ni[1]	DA	-
0x045	EIEP0	ni[1]	DA	-
0x046	EMEP0	ni[1]	DA	-
0x047	ECEP0	ni[1]	DA	-
0x048	IIEP1	ni	DA	-
0x049	IMEP1	ni	DA	-
0x04A	CEP1	ni	DA	-
0x04B	CPEP1	ni	DA	-
0x04C	GPEP1	ni	DA	-
0x04D	EIEP1	ni	DA	-
0x04E	EMEP1	ni	DA	-
0x04F	ECEP1	ni	DA	-
0x050	IIEP2	ni	DA	-
0x051	IMEP2	ni	DA	-
0x052	CEP2	ni	DA	-
0x053	CPEP2	ni	DA	-
0x054	GPEP2	ni	DA	-
0x055	EIEP2	ni	DA	-
0x056	EMEP2	ni	DA	-
0x057	ECEP2	ni	DA	-
0x058	IIEP3	ni	DA	-
Notes: An "ni" in the Initialization column indicates that the register is Not Initialized. For information on Register Groups, see Table A-16 on page A-49.				

Table A-17. I/O Processor Registers Memory Map (Cont'd)

Register Address	Register Name	Initialization After Reset	Register Group	Reference
0x059	IMEP3	ni	DA	-
0x05A	CEP3	ni	DA	-
0x05B	CPEP3	ni	DA	-
0x05C	GPEP3	ni	DA	-
0x05D	EIEP3	ni	DA	-
0x05E	EMEP3	ni	DA	-
0x05F	ECEP3	ni	DA	-
0x060	II0A	ni	DA	-
0x061	IM0A	ni	DA	-
0x062	C0A	ni	DA	-
0x063	CP0A	ni	DA	-
0x064	GP0A	ni	DA	-
0x067-65	Reserved			
0x068	II1A	ni	DA	-
0x069	IM1A	ni	DA	-
0x06A	C1A	ni	DA	-
0x06B	CP1A	ni	DA	-
0x06C	GP1A	ni	DA	-
0x06F-6D	Reserved			
0x070	II2A	ni	DA	-
0x071	IM2A	ni	DA	-
0x072	C2A	ni	DA	-

Notes: An "ni" in the Initialization column indicates that the register is Not Initialized. For information on Register Groups, see Table A-16 on page A-49.

Table A-17. I/O Processor Registers Memory Map (Cont'd)

Register Address	Register Name	Initialization After Reset	Register Group	Reference
0x073	CP2A	ni	DA	-
0x074	GP2A	ni	DA	-
0x078	II3A	ni	DA	-
0x079	IM3A	ni	DA	-
0x07A	C3A	ni	DA	-
0x07B	CP3A	ni	DA	-
0x07C	GP3A	ni	DA	-
0x080	II0B	ni	DA	-
0x081	IM0B	ni	DA	-
0x082	C0B	ni	DA	-
0x083	CP0B	ni	DA	-
0x084	GP0B	ni	DA	-
0x088	II1B	ni	DA	-
0x089	IM1B	ni	DA	-
0x08A	C1B	ni	DA	-
0x08B	CP1B	ni	DA	-
0x08C	GP1B	ni	DA	-
0x090	II2B	ni	DA	-
0x091	IM2B	ni	DA	-
0x092	C2B	ni	DA	-
0x093	CP2B	ni	DA	-
0x094	GP2B	ni	DA	-

Notes: An "ni" in the Initialization column indicates that the register is Not Initialized. For information on Register Groups, see Table A-16 on page A-49.

Table A-17. I/O Processor Registers Memory Map (Cont'd)

Register Address	Register Name	Initialization After Reset	Register Group	Reference
0x098	II3B	ni	DA	-
0x099	IM3B	ni	DA	-
0x09A	C3B	ni	DA	-
0x09B	CP3B	ni	DA	-
0x09C	GP3B	ni	DA	-
0x0B4	SPICTL	0x0000 0000	LSP	page A-117
0x0B5	SPISTAT	ni	LSP	page A-115
0x0B6	SPITX	ni	LSP	page A-121
0x0B7	SPIRX	ni	LSP	page A-121
0x0B8	SDCTL	ni	SD	page A-73
0x0B9	SDRDIV	ni	SD	page A-72
0x0C0	LBUF0	ni	LSP	page A-92
0x0C2	LBUF1	ni	LSP	page A-92
0x0CC	LCTL	0x0000 0000	LSP	page A-92
0x0D0	LSRQ	0x0000 0000	LSP	page A-98
0x0D7 – 0x0DF	Reserved			
0x1C0	SPCTL0	0x0000 0000	SC	page A-114
0x1C1	TX0A	ni	SC	page A-112
0x1C2	TX0B	ni	SC	page A-112
0x1C3	RX0A	ni	SC	page A-112
0x1C4	RX0B	ni	SC	page A-112
0x1C5	DIV0	ni	SC	page A-112

Notes: An "ni" in the Initialization column indicates that the register is Not Initialized. For information on Register Groups, see Table A-16 on page A-49.

Table A-17. I/O Processor Registers Memory Map (Cont'd)

Register Address	Register Name	Initialization After Reset	Register Group	Reference
0x1C6	CNT0	ni	SC	-
0x1C7	MR0CS0	ni	SC	page A-114
0x1C8	MR0CCS0	ni	SC	page A-114
0x1C9	MR0CS1	ni	SC	page A-114
0x1CA	MR0CCS1	ni	SC	page A-114
0x1CB	MR0CS2	ni	SC	page A-114
0x1CC	MR0CCS2	ni	SC	page A-114
0x1CD	MR0CS3	ni	SC	page A-114
0x1CE	MR0CCS3	ni	SC	page A-114
0x1D0	SPCTL2	0x0000 0000	SC	page A-100
0x1D1	TX2A	ni	SC	page A-112
0x1D2	TX2B	ni	SC	page A-112
0x1D3	RX2A	ni	SC	page A-112
0x1D4	RX2B	ni	SC	page A-112
0x1D5	DIV2	ni	SC	page A-112
0x1D6	CNT2	ni	SC	
0x1D7	MT2CS0	ni	SC	page A-113
0x1D8	MT2CCS0	ni	SC	page A-113
0x1D9	MT2CS1	ni	SC	page A-113
0x1DA	MT2CCS1	ni	SC	page A-113
0x1DB	MT2CS2	ni	SC	page A-113
0x1DC	MT2CCS2	ni	SC	page A-113

Notes: An "ni" in the Initialization column indicates that the register is Not Initialized. For information on Register Groups, see Table A-16 on page A-49.

Table A-17. I/O Processor Registers Memory Map (Cont'd)

Register Address	Register Name	Initialization After Reset	Register Group	Reference
0x1DD	MT2CS3	ni	SC	page A-113
0x1DE	MT2CCS3	ni	SC	page A-113
0x1DF	SP02MCTL	0x0000 0000	SC	page A-109
0x1E0	SPCTL1	0x0000 0000	LSP	page A-109
0x1E1	TX1A	ni	LSP	page A-112
0x1E2	TX1B	ni	LSP	page A-112
0x1E3	RX1A	ni	LSP	page A-112
0x1E4	RX1B	ni	LSP	page A-112
0x1E5	DIV1	ni	LSP	page A-112
0x1E6	CNT1	ni	LSP	-
0x1E7	MR1CS0	ni	LSP	page A-114
0x1E8	MR1CCS0	ni	LSP	page A-114
0x1E9	MR1CS1	ni	LSP	page A-114
0x1EA	MR1CCS1	ni	LSP	page A-114
0x1EB	MR1CS2	ni	LSP	page A-114
0x1EC	MR1CCS2	ni	LSP	page A-114
0x1ED	MR1CS3	ni	LSP	page A-114
0x1EE	MR1CCS3	ni	LSP	page A-114
0x1F0	SPCTL3	0x0000 0000	SC	page A-100
0x1F1	TX3A	ni	LSP	page A-112
0x1F2	TX3B	ni	LSP	page A-112
0x1F3	RX3A	ni	LSP	page A-112
Notes: An "ni" in the Initialization column indicates that the register is Not Initialized. For information on Register Groups, see Table A-16 on page A-49.				

Table A-17. I/O Processor Registers Memory Map (Cont'd)

Register Address	Register Name	Initialization After Reset	Register Group	Reference
0x1F4	RX3B	ni	LSP	page A-112
0x1F5	DIV3	ni	LSP	page A-112
0x1F6	CNT3	ni	LSP	-
0x1F7	MT3CS0	ni	LSP	page A-113
0x1F8	MT3CCS0	ni	LSP	page A-113
0x1F9	MT3CS1	ni	LSP	page A-113
0x1FA	MT3CCS1	ni	LSP	page A-113
0x1FB	MT3CS2	ni	LSP	page A-113
0x1FC	MT3CCS2	ni	LSP	page A-113
0x1FD	MT3CS3	ni	LSP	page A-113
0x1FE	MT3CCS3	ni	LSP	page A-113
0x1FF	SP13MCTL	0x0000 0000	SC	page A-109
Notes: An "ni" in the Initialization column indicates that the register is Not Initialized. For information on Register Groups, see Table A-16 on page A-49.				

1 Initialization depends on the booting mode.

System Configuration Register (SYSCON)

The SYSCON register, described in Table A-18 and Figure A-15, is used to set up system configuration selections. This register's address is 0x00. The reset value for this register is 0x0001 0020, based on HBW configured for an 8-bit host.

Table A-18. System Configuration Register (SYSCON) Bit Definitions

Bit(s)	Name	Definition
0	SRST	**Software Reset.** This bit resets (when set, =1) the processor. When a program sets (=1) SRST, the processor responds to the non-maskable RSTI interrupt and clears (=0) SRST.
1	BSO	**Boot Select Override.** This bit enables (if set, =1) or disables (if cleared, =0) access to Boot Memory Space. When BSO is set, the processor uses the $\overline{\text{BMS}}$ select line (instead of $\overline{\text{MS3-0}}$) to perform DMA channel 10 accesses of external memory. The processor uses 8-to 48-bit packing when reading from 8-bit boot memory space, but does no packing on writes to this space.
2	IIVT	**Internal Interrupt Vector Table.** This bit forces placement of the interrupt vector table at address 0x0004 0000 regardless of booting mode (if 1) or allows placement of the interrupt vector table as selected by the booting mode (if 0).
3		Reserved
5-4	HBW	**Host Bit Width.** These bits select the bit width for host access as follows: 00= 32-bit, 01= 16-bit, 10=8-bit (default), 11=reserved.
6		Reserved
7	HMSWF	**Host Most Significant Word First Packing Select.** This bit selects the word packing order for host accesses as most-significant-word first (if set, =1) or least-significant-word first (if cleared, =0).
8		Reserved
9	IMDW0	**Internal Memory Block 0 Data Width.** This bit selects the normal word data access size for internal memory Block 0 as 40-bit data (if set, =1) or 32-bit data (if cleared, =0).

Table A-18. System Configuration Register (SYSCON) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
10	IMDW1	**Internal Memory Block 1 Data Width.** This bit selects the normal word data access size for internal memory Block 1 as 40-bit data (if set, =1) or 32-bit data (if cleared, =0).
11	ADREDY	**Active Drive REDY.** This bit selects line driver type for the processor's REDY pin as active drive (a/d) (if set, =1) or open drain (o/d) (if cleared, =0).
15-12		Reserved
16	BHD	**Buffer Hang Disable.** This bit controls whether the processor core proceeds (hang disabled if set, =1) or is held-off (hang enabled if cleared, =0) when the core tries to read from an empty EPBx, RXx, SPIRX, or LBUFx buffer or tries to write to a full EPBx, TXx, SPITX, or LBUFx buffer. Is cleared by default at reset.
18-17	EBPR	**External Bus Priority.** These bits select the priority for the I/O processor's EP bus when arbitrating access to the processor's external port as follows: 00—priority rotates between DM or PM and IO buses, 01—the winning DM or PM bus has priority over the IO bus, 10—the IO bus has priority over the winning DM or PM bus.
19	DCPR	**External Port DMA Channel Priority Rotation Enable.** This bit enables (rotates if set, =1) or disables (fixed if cleared, =0) priority rotation among external port DMA channels (channel 10-13).
20	LDCPR	**Link Port DMA Channel Priority Rotation Enable.** This bit enables (rotates if set, =1) or disables (fixed if cleared, =0) priority rotation among link port DMA channels (channel 8-9).
21	PRROT	**Link–External Port DMA Channel Priority Rotation Enable.** This bit enables (rotates if set, =1) or disables (fixed if cleared, =0) priority rotation between link port DMA channels (channel 8-9) and external port DMA channels (channel10-13).
22	COD	**CLKOUT Disable.** This bit disables (if set, =1) or enables (if cleared, =0) the processor clock output on the CLKOUT pin. If enabled, the processor outputs the clock signal on CLKOUT. If disabled, the processor three-states the CLKOUT pin. **Note:** Is ignored if bit 23 COPT is set.

Table A-18. System Configuration Register (SYSCON) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
23	COPT	**CLKOUT Option.** This bit enables (if set, =1) the master device in a multiprocessor system to drive the CLKOUT pin. If cleared (=0), CLKOUT is controlled by bit 22 COD.
29-24		Reserved
31-30	IPACK	**External Packed Instruction Execution Mode.** This bit sets the packing of instructions as follows: 00=32- to 48-bit packing, 01=no packing, 10=16- to 48-bit packing, 11=8- to 48-bit packing.

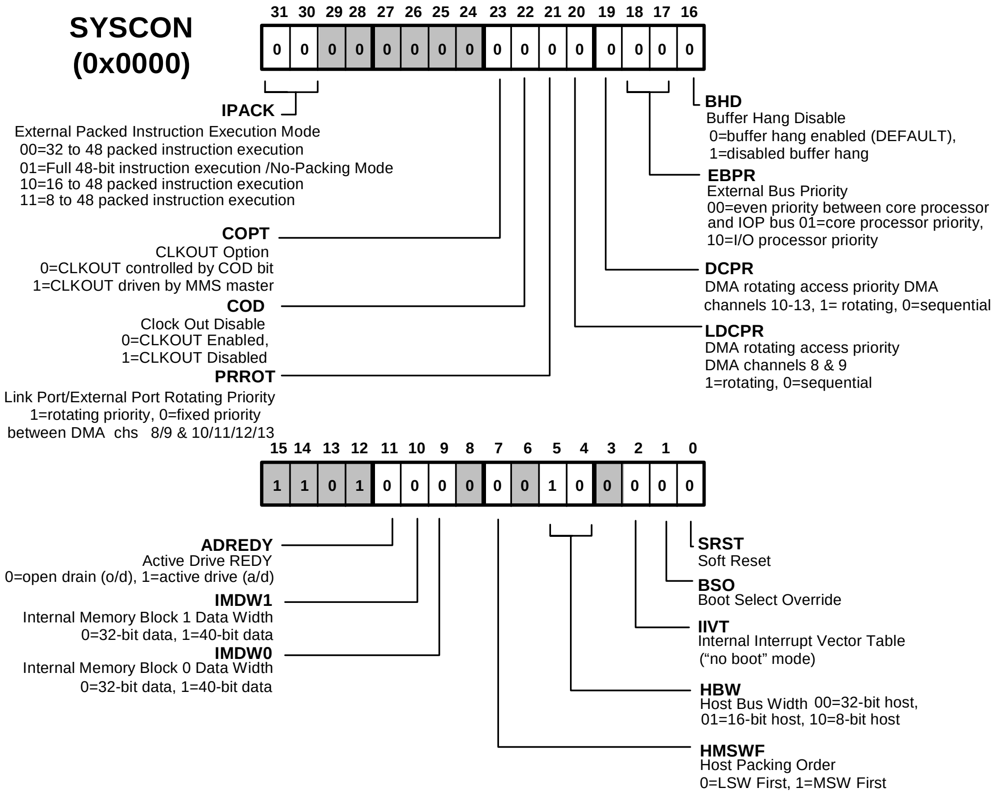

SYSCON
(0x0000)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
IPACK
External Packed Instruction Execution Mode
00=32 to 48 packed instruction execution
01=Full 48-bit instruction execution /No-Packing Mode
10=16 to 48 packed instruction execution
11=8 to 48 packed instruction execution
COPT
CLKOUT Option
0=CLKOUT controlled by COD bit
1=CLKOUT driven by MMS master
COD
Clock Out Disable
0=CLKOUT Enabled,
1=CLKOUT Disabled
PRROT
Link Port/External Port Rotating Priority
1=rotating priority, 0=fixed priority
between DMA chs 8/9 & 10/11/12/13
BHD
Buffer Hang Disable
0=buffer hang enabled (DEFAULT),
1=disabled buffer hang
EBPR
External Bus Priority
00=even priority between core processor and IOP bus 01=core processor priority,
10=I/O processor priority
DCPR
DMA rotating access priority DMA channels 10-13, 1= rotating, 0=sequential
LDCPR
DMA rotating access priority
DMA channels 8 & 9
1=rotating, 0=sequential
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 0 0 0 0 0 1 0 0 0 0 0
ADREDY
Active Drive REDY
0=open drain (o/d), 1=active drive (a/d)
IMDW1
Internal Memory Block 1 Data Width
0=32-bit data, 1=40-bit data
IMDW0
Internal Memory Block 0 Data Width
0=32-bit data, 1=40-bit data
SRST
Soft Reset
BSO
Boot Select Override
IIVT
Internal Interrupt Vector Table
("no boot" mode)
HBW
Host Bus Width 00=32-bit host,
01=16-bit host, 10=8-bit host
HMSWF
Host Packing Order
0=LSW First, 1=MSW First

Figure A-15. SYSCON Register

Vector Interrupt Address Register (VIRPT)

The `VIRPT` register's address is 0x01. The reset value for this register is 0x0004 0014. In no boot mode, the reset value is 0x0020 0014 because the interrupt resides in external memory. The sequencer uses the `VIRPT` register (Table A-19 and Figure A-16)to support multiprocessor vector interrupts. The vector interrupt (`VIRPTI`) permits passing interprocessor commands in multiple-processor systems. This interrupt occurs when an external processor (a host or another processor) writes an address to the `VIRPT` register, inserting a new vector address for `VIRPT`.

Table A-19. Vector Interrupt Address Register (VIRPT) Bit Definitions

Bit(s)	Name	Definition
23-0	VIRPTA	**Vector Interrupt Address.** These bits contain the multiprocessor interrupt's vector (address). When an external processor loads an address into this register, the processor pushes the status stack and starts executing the routine at the vector address.
31-24	VIRPTD	**Vector Interrupt (optional) Data.** These bits contain optional data that the external processor may pass to the interrupt service routine.

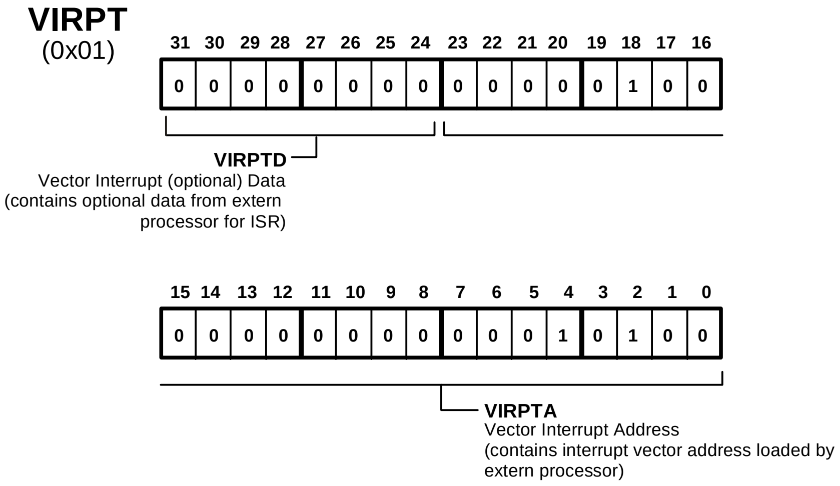


Figure A-16. VIRPT Register

External Memory Waitstate and Access Mode Register (WAIT)

The WAIT register, shown in Table A-20 and Figure A-17, has an address of 0x02. The reset value for this register is 0x01ce 739c, which equates to the following processor external memory settings: **asynchronous access mode** for all external memory banks, **seven waitstates** with a hold cycle for all accesses to external memory banks, external DRAM page size of 256 words (if installed), and **disable idle cycle** for DMA handshake.

Table A-20. External Memory Setup Register (WAIT) Bit Definitions

Bit(s)	Name	Definition
1-0	EB0AM	**External Bank 0 Access Mode.** These bits select the access mode for external memory Bank 0 as follows: EBxAM External Bank Access Mode 00 Asynchronous—processor $\overline{RD}$ and $\overline{WR}$ strobes change before CLKOUT's edge—accesses use the waitstate count setting from EBxWS and require external acknowledge (ACK), allowing a de-asserted ACK to extend the access time. 01 Synchronous—processor $\overline{RD}$ and $\overline{WR}$ strobes change on CLKOUT's edge—reads use the waitstate count setting from EBxWS (minimum EBxWS=001); writes are 0-wait state. 10 Synchronous—processor $\overline{RD}$ and $\overline{WR}$ strobes change on CLKOUT's edge—reads use the waitstate count setting from EBxWS (minimum EBxWS=001); writes are 1-wait state. 11 Reserved
4-2	EB0WS	**External Bank 0 Waitstates.** These bit fields select the waitstates for external memory Bank 0 as follows: EBxWS # of Waitstates Hold Time Cycle? 000 0 no 001 1 no 010 2 yes 011 3 yes 100 4 yes 101 5 yes 110 6 yes 111 7 yes Note that Hold Cycles applies to asynchronous mode only.
6-5	EB1AM	**External Bank 1 Access Mode.** (see EB0AM definition)
9-7	EB1WS	**External Bank 1 Waitstates.** (see EB0WS definition)
11-10	EB2AM	**External Bank 2 Access Mode.** (see EB0AM definition)

Table A-20. External Memory Setup Register (WAIT) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
14-12	EB2WS	**External Bank 2 Waitstates.** (see EB0WS definition)
16-15	EB3AM	**External Bank 3 Access Mode.** (see EB0AM definition)
19-17	EB3WS	**External Bank 3 Waitstates.** (see EB0WS definition)
21-20	RBAM	**ROM Boot Access Mode.** (see EB0AM definition)
24-22	RBWS	**ROM Boot Waitstates.** (see EB0WS definition)
29-25		Reserved
30	HIDMA	**Handshake and Idle for DMA Enable.** This bit enables (if set, =1) or disables (if cleared, =0) adding an idle cycle after every memory access for DMAs with handshaking ($\overline{\text{DMAR}}$-$\overline{\text{DMAG}}$). The added cycle reduces bus contention by accommodating devices with a slow three-state time. Also, the added cycle accommodates long write recovery time by de-asserting $\overline{\text{DMAG}}$ longer.
31		Reserved

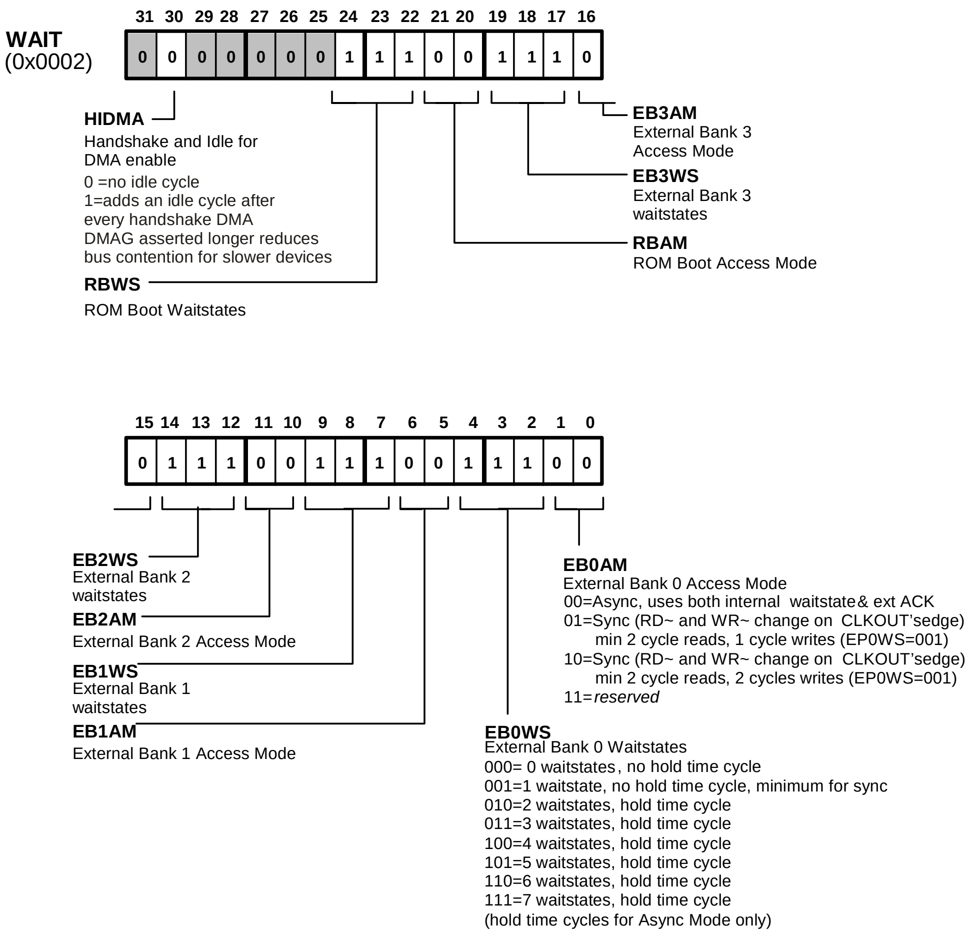

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
WAIT
(0x0002)
0 0 0 0 0 0 0 1 1 1 0 0 1 1 1 0
HIDMA
Handshake and Idle for
DMA enable
0 =no idle cycle
1=adds an idle cycle after
every handshake DMA
DMAG asserted longer reduces
bus contention for slower devices
RBWS
ROM Boot Waitstates
EB3AM
External Bank 3
Access Mode
EB3WS
External Bank 3
waitstates
RBAM
ROM Boot Access Mode
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 0 0 1 1 1 0 0 1 1 1 0 0
EB2WS
External Bank 2
waitstates
EB2AM
External Bank 2 Access Mode
EB1WS
External Bank 1
waitstates
EB1AM
External Bank 1 Access Mode
EB0AM
External Bank 0 Access Mode
00=Async, uses both internal waitstate& ext ACK
01=Sync (RD~ and WR~ change on CLKOUT'sedge)
min 2 cycle reads, 1 cycle writes (EP0WS=001)
10=Sync (RD~ and WR~ change on CLKOUT'sedge)
min 2 cycle reads, 2 cycles writes (EP0WS=001)
11=reserved
EB0WS
External Bank 0 Waitstates
000= 0 waitstates, no hold time cycle
001=1 waitstate, no hold time cycle, minimum for sync
010=2 waitstates, hold time cycle
011=3 waitstates, hold time cycle
100=4 waitstates, hold time cycle
101=5 waitstates, hold time cycle
110=6 waitstates, hold time cycle
111=7 waitstates, hold time cycle
(hold time cycles for Async Mode only)

Figure A-17. WAIT Register

System Status Register (SYSTAT)

The `SYSTAT` register's address is 0x03. The reset value has all bits initialized to zero, except for the `IDC`, `CRBM`, `CRAT` fields, which are set from values on the ADSP-21161 processor's pins. This register is described in Table A-21 and Figure A-18.

Table A-21. System Status Register (SYSTAT) Bit Definitions

Bit(s)	Name	Definition
0	HSTM	**Host Bus Master.** Indicates whether the Host processor has control of the external bus (host bus master if set, =1) or does not have control of the bus (host not bus master if cleared, =0, reset value).
1	BSYN	**Bus Synchronized.** Indicates whether the processor's bus arbitration logic is synchronized (if set, =1) or is not synchronized (if cleared, =0, reset value).
3-2		Reserved (reset value =0)
6-4	CRBM	**Current Bus Master.** These bits indicate the ID of the processor that currently is the bus master in a multiprocessor system. Because CRBM is only valid for DSPs with ID inputs other than zero (e.g. a multiprocessor system), the processor keeps CRBM set to 001 when ID equals 000. The reset value of CRBM is undefined.
7		Reserved (reset value =0)
10-8	IDC	**ID Code.** These bits indicate the state of the ID pins on the processor. The reset value of IDCID is undefined.
12-11		Reserved (reset value =0)
13	VIPD	**Vector Interrupt Pending.** Indicates whether a vector interrupt is pending (if set, =1) or is not pending (if cleared, =0, reset value). A vector interrupt occurs when an address is written to the VIRPT register. The processor clears VIPD on return from the VIRPT interrupt service routine. Systems using vector interrupts should monitor VIPD to determine that the processor has serviced the VIRPT interrupt and is ready for another vector interrupt.
15-14		Reserved

Table A-21. System Status Register (SYSTAT) Bit Definitions

Bit(s)	Name	Definition
18-16	CRAT	**Core Clock-to-CLKIN ratio.** These bits indicate the state of the CLK_CFG[1:0] pins (clock ratio) on the processor as follows 010 = 2:1 ratio, 011 = 3:1 ratio, 100 = 4:1 ratio. The reset value of CRAT is undefined.
19		Reserved
20	SSWPD	**Synchronous Slave Write FIFO Data Pending.** Indicates if set (=1) that a synchronous slave IOP register write is pending. This bit indicates, if cleared (=0), that no slave write is pending.
21	SWPD	**Slave Write FIFO Pending.** Indicates whether a direct write (synchronous or asynchronous) to processor's IOP register is pending (if set, =1) or is not pending (if cleared, =0, reset value). The processor clears SWPD when the direct write is complete. If an external device attempts a direct write during a DMA chaining operation or if higher priority DMA request occurs, the direct IOP register write may be delayed for several cycles. The maximum delay for a pending direct write is 12 cycles.
24-22	HPS	**Host Packing Status.** These bits indicate the host's packing status as follows: 000 = pack complete (reset value) and 6th stage of 8- to 48-bit packing, 4th stage of 8- to 32-bit packing etc. 001 = 1st stage pack/unpack 010 = 2nd stage multi-stage pack/unpack 011 = 3rd stage multi-stage pack/unpack 100 = 5th stage multi-stage pack/unpack 101 = 110 = 111 = reserved These bits are read-only. The processor clears these bits when DEN is cleared (changes from 1 to 0).
31-25		Reserved (reset value =0)

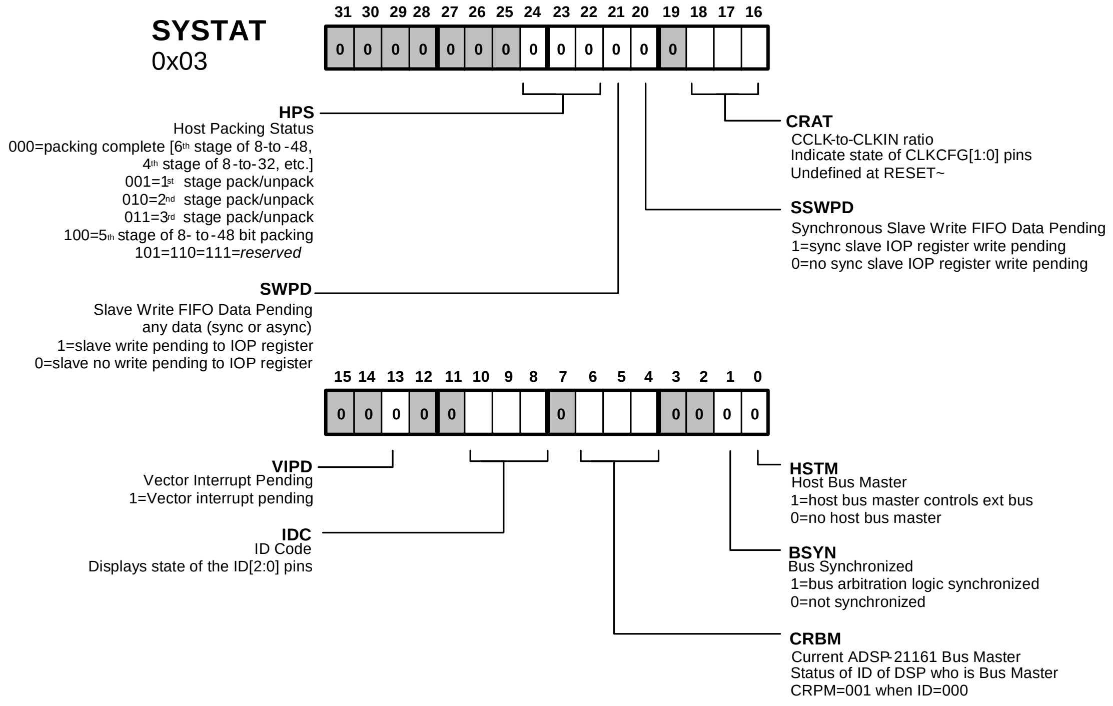
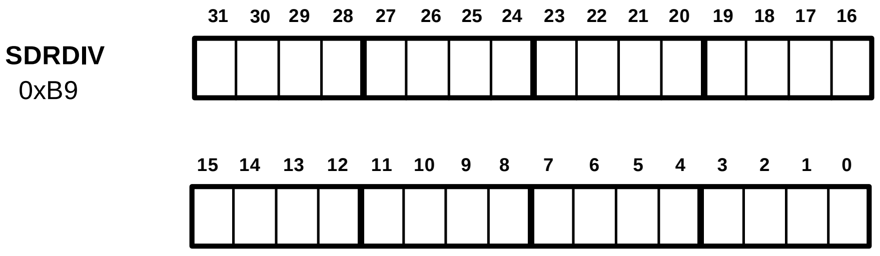

SYSTAT
0x03
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0
HPS
Host Packing Status
000=packing complete [6th stage of 8-to-48, 4th stage of 8-to-32, etc.]
001=1st stage pack/unpack
010=2nd stage pack/unpack
011=3rd stage pack/unpack
100=5th stage of 8- to-48 bit packing
101=110=111=reserved
SWPD
Slave Write FIFO Data Pending
any data (sync or async)
1=slave write pending to IOP register
0=slave no write pending to IOP register
CRAT
CCLK-to-CLKIN ratio
Indicate state of CLKCFG[1:0] pins
Undefined at RESET~
SSWPD
Synchronous Slave Write FIFO Data Pending
1=sync slave IOP register write pending
0=no sync slave IOP register write pending
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0
VIPD
Vector Interrupt Pending
1=Vector interrupt pending
IDC
ID Code
Displays state of the ID[2:0] pins
HSTM
Host Bus Master
1=host bus master controls ext bus
0=no host bus master
BSYN
Bus Synchronized
1=bus arbitration logic synchronized
0=not synchronized
CRBM
Current ADSP-21161 Bus Master
Status of ID of DSP who is Bus Master
CRPM=001 when ID=000

Figure A-18. SYSTAT Register

SDRDIV Register (SDRDIV)

This register's address is 0xb9. The reset value for this register is undefined. The SDRDIV register is a programmable refresh counter used to coordinate the supplied clock rate with the SDRAM device's required refresh rate. This register is described in Figure A-19.



$$\mathrm{SDRDIV} = \frac{f_{CCLK}}{\text{SDRAM refresh rate cycle}} - CL - t_{RP} - 5$$

Figure A-19. SDRDIV Register

SDRAM Control Register (SDCTL)

This register's addresses is 0xb8. The reset value for this register is undefined. At reset, all the SDCTL bits are cleared to zero. The SDCTL register is an IOP register for the SDRAM controller. This register is described in Table A-22 and Figure A-20. This register does not support bitwise operations.

Table A-22. SDCTL Control Register Bits

Bit Number	Name	Description
1-0	SDCL	**SDRAM CAS Latency.** Set the CAS delay as follows: 01= 1 latency, 10 = 2 latency, 11 = 3 latency, 00 = reserved. The CAS latency is the delay, in clock cycles, between when the SDRAM detects the read command and when it provides the data at its output pins.
2	DSDCTL	**Disable SDCLK0 and Control Signals.** This bit disables if set (=1) or enables if cleared (=0) the following signals: SDCLK0, RAS, CAS, SDWE, SDCKE.
3	DSDCK1	**Disable SDCLK1.** This bit disables if set (=1) or enables if cleared (=0), the SDCK1 signals.
7-4	SDTRAS	**SDRAM TRAS Specification**. Set the SDRAM t_{RAS} specification in number of clock cycle (0-15 cycles). T_{RAS} is the required delay between issuing an activate command and issuing a precharge command.
10-8	SDTRP	**SDRAM TRP Specification**. Set the SDRAM t_{RAS} specification in number of clock cycle (0-7 cycles). T_{RP} is the required delay between issuing a precharge command and issuing an activate command.

Table A-22. SDCTL Control Register Bits (Cont'd)

Bit Number	Name	Description
11	SDPM	**SDRAM Power-Up Mode.** This bit enables if set (=1) or disables if cleared (=0) the following commands as generated by the SDRAM controller: 0 = PRE/ 8 CBR /MRS, 1= PRE/ MRS/ 8 CBR. PRE - Precharge- closes an active bank. CBR - Automatic Refresh - the SDRAM drives its own refresh cycle with no external control input. MRS- Mode Register Set - initializes the SDRAM operation parameters during the power-up sequence.
13-12	SDPGS	**SDRAM Page Size.** Set the SDRAM page size as follows: 00= 256 words, 01 = 512 words, 10 = 1k words, 11 = 2k words.
14	SDPSS	**SDRAM Power-Up Sequence**. This bit enables if set (=1) or disables if cleared (=0) the SDRAM power up sequence start.
15	SDSRF	**SDRAM Self Refresh Mode**. This bit enables if set (=1) or disables if cleared (=0) SDRAM self refresh mode. In this mode, the SDRAM drives its own refresh cycle with no external control input. At cycle end, both SDRAM banks are precharged (idle). This control bit always reads zero.
16[1]	SDEM0	**External Memory Bank 0**. Indicates, if set (=1), that external memory bank 0 has SDRAM or, if cleared (=0), that external memory bank 0 does not have SDRAM.
17[1]	SDEM1	**External Memory Bank 1**. Indicates, if set (=1), that external memory bank 1 has SDRAM or, if cleared (=0), that external memory bank 1 does not have SDRAM.
18[1]	SDEM2	**External Memory Bank 2**. Indicates, if set (=1), that external memory bank 2 has SDRAM or, if cleared (=0), that external memory bank 2 does not have SDRAM.
19[1]	SDEM3	**External Memory Bank 3**. Indicates, if set (=1), that external memory bank 3 has SDRAM or, if cleared (=0), that external memory bank 3 does not have SDRAM.

Table A-22. SDCTL Control Register Bits (Cont'd)

Bit Number	Name	Description
20	SDBN	**SDRAM Bank Number.** Indicates the number of banks your SDRAM device contains. If set (=1), there are four bank in SDRAM. If cleared (=0), there are two banks in SDRAM.
21	SDCKR	**SDRAM Clock to Core Clock Ratio.** Indicates the SDRAM clock to core clock ratio as follows: 1 = full core clock, 0 = half core clock.
22		Reserved
23	SDBUF	**External Register Buffer**. Indicates, if set (=1), the existence of an external register buffer for address and control of SDRAM. If this bit is cleared (=0), there is no external register buffer.
26-24	SDTRCD	**SDRAM TRCD Specification**. Set the SDRAM t_{RCD} specification in number of clock cycle (0-7 cycles). T_{RCD} is the required delay between an ACT command and the start of the first read or write operation.
31-27		Reserved

1 The $\overline{\text{CS}}$ pin of a SDRAM chip should be connected to $\overline{\text{MSx}}$ pin of the ADSP-21161 processor for the corresponding memory bank in which you want to map the SDRAM device. All four memory banks can have SDRAM simultaneously.

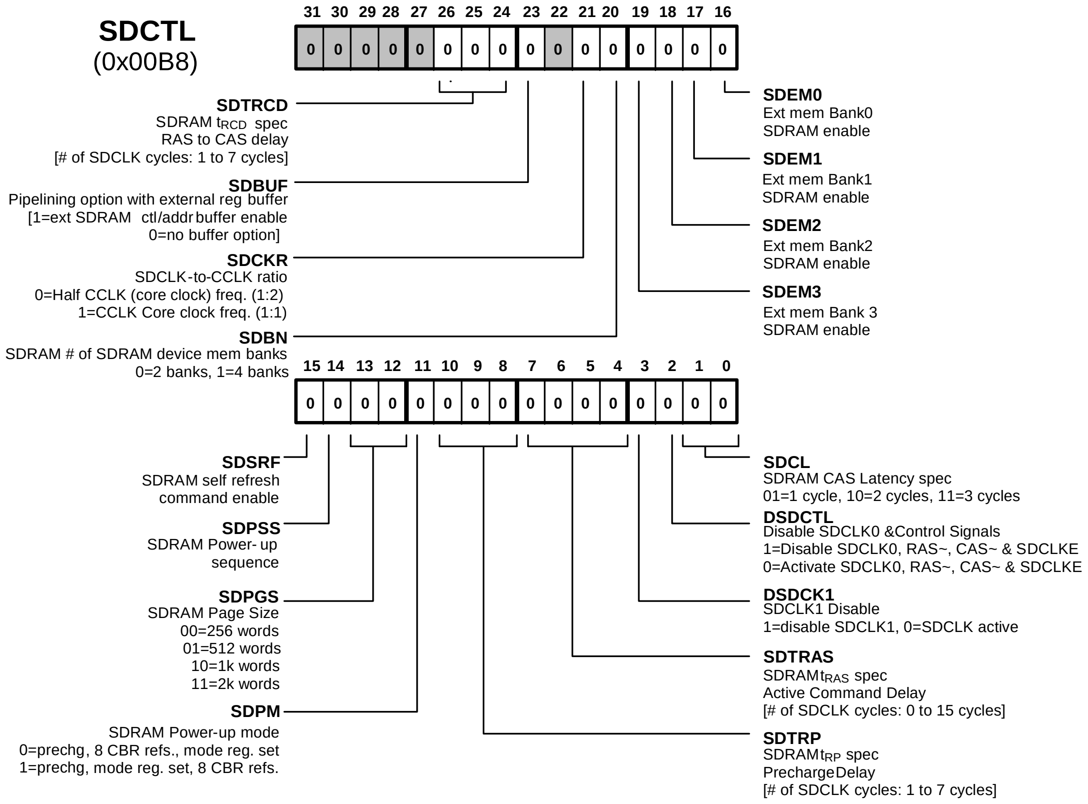


Figure A-20. SDCTL Register

External Port DMA Buffer Registers (EPBx)

The EPBx registers' addresses are: EPB0–0x04, EPB1–0x06, EPB2–0x14, and EPB3–0x16. The reset value for these registers is undefined.

External port buffers are 8 levels deep and 64 bits wide. The buffers contain 40/48- or 32/64-bit words, depending on the external port buffer's data type selected with the DTYPE bit in the port's DMACx register. If the buffer contains 32-, 40- or 48-bit words, the port aligns the data with the lower bits of the buffer and zero fills the upper 32, 24 or 16 bits.

Normally, a DMA process automatically accesses the buffer register for memory transfer. Programs can also access these buffers as registers. However, programs must use the PX register to access the full width of the buffer. A PX register move can access the entire 64 bits of an external port buffer using the full width PX.

Message Registers (MSGRx)

The MSGRx registers' addresses are: MSGR0–0x08, MSGR1–0x09, MSGR2–0x0a, MSGR3–0x0b, MSGR4–0x0c, MSGR5–0x0d, MSGR6–0x0e, and MSGR7–0x0f. The reset value for these registers is undefined.

PC Shadow Register (PC_SHDW)

The PCSHDW register's address is 0x10. The reset value for this register matches the PC register. PC_SHDW contains a read-only mirror of the 24-bit address in the Program Counter (PC) register. External devices can poll this PC_SHDW for the contents of PC. Note that the value in PC_SHDW may lag behind the current PC by one or more core clock cycles. This register is shown in Figure A-21.

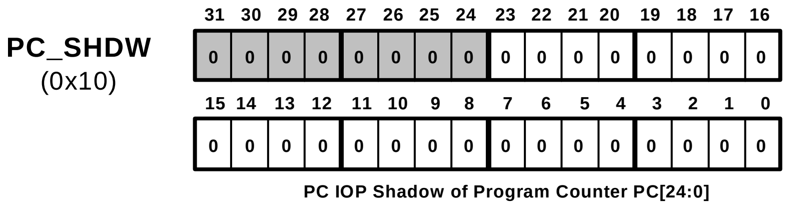


Figure A-21. PC Shadow Register

MODE2 Shadow Register (MODE2_SHDW)

This register's address is 0x11. Because `MODE2_SHDW` register bits 31-25 are the ADSP-21161 processor ID and silicon revision, the reset value varies with the system setting and silicon revision. External devices can poll this `MODE2_SHDW` for the processor's processor ID and silicon revision. This register is described in Table A-23 and Figure A-22.

Table A-23. Mode2 Shadow Register (MODE2_SHDW) Bit Definitions

Bit	Name	Definition
24-0		Reserved
27-25	PID2-0	**Processor Identification (Read only) PID2-0.**
29-28		**Silicon Revision number.** Silicon revision 1.0 and 1.1 are both 01. Silicon revision 0.3 is 00.
31-30	PID4-3	**Processor Identification (Read only) PID4-3.**

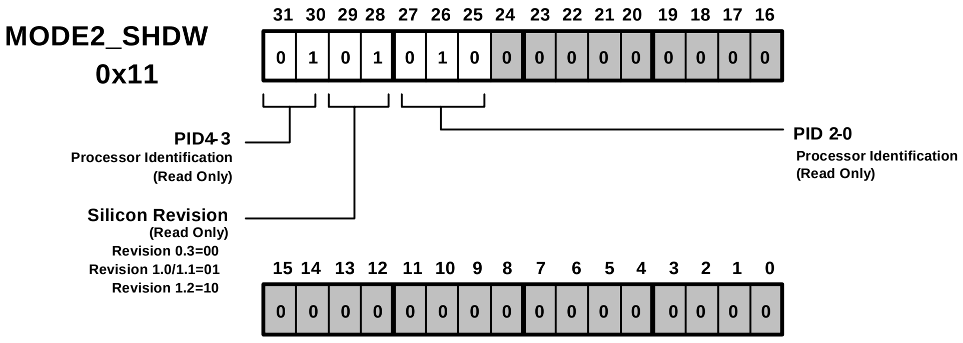


Figure A-22. MODE2 Shadow Register

Bus Time-Out Maximum Register (BMAX)

This register's address is 0x18 and it is shown in Figure A-23. The reset value for this register is 0x0000 0000. The lower 16 bits of this register hold the value for the maximum number of cycles -2 that the processor can retain bus mastership. The upper 16 bits of this register are reserved.

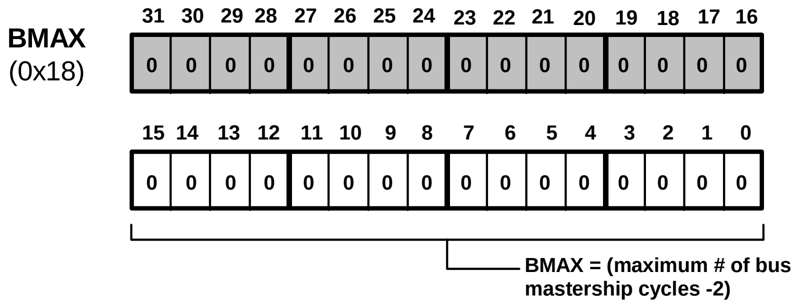


Figure A-23. BMAX Register

For more information describing how BMAX and BCNT work, see "Bus Mastership Timeout" on page 7-101.

Bus (Time-Out) Counter Register (BCNT)

This register's address is 0x19 and it is shown in Figure A-24. The reset value for this register is 0x0000 0000. The lower 16 bits of this register hold the count of the number of cycles remaining for the processor to retain bus mastership. The upper 16 bits of this register are reserved.

For more information describing how BMAX and BCNT work, see "Bus Mastership Timeout" on page 7-101.

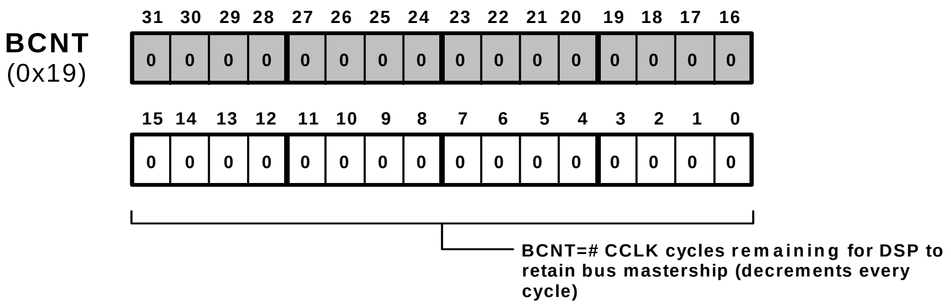


Figure A-24. BCNT Register

External Port DMA Control Registers (DMACx)

These registers' addresses are: DMAC10–0x1C, DMAC11–0x1D, DMAC12–0x1E, DMAC13–0x1F. The reset value for these registers is 0x0000 0000 unless you are booting from a host processor or PROM booting.

Each external port DMA channel has its own control register. The registers, DMAC10, DMAC11, DMAC12, and DMAC13 correspond to DMA channels 10, 11, 12, and 13. Table A-24 and Figure A-25 provide bit definitions for the DMACx registers.

Except for the FLSH bit, the control bits in the DMACx registers have a one cycle effect latency. The FLSH bit has a two cycle effect latency.

Table A-24. External Port DMA Control Registers (DMACx) Bit Definitions

Bit(s)	Name	Definition
0	DEN	**External Port DMA Enable.** This bit enables (if set, =1) or disables (if cleared, =0) DMA for the corresponding external port FIFO buffer (EPBx).
1	CHEN	**External Port DMA Chaining Enable.** This bit enables (if set, =1) or disables (if cleared, =0) DMA chaining for the corresponding external port FIFO buffer (EPBx).

Table A-24. External Port DMA Control Registers (DMACx) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
2	TRAN	**External Port Transmit/Receive Select.** This bit selects the transfer direction (transmit if set, =1) (receive if cleared, =0) for the corresponding external port FIFO buffer (EPBx).
4-3		Reserved
5	DTYPE	**External Port Data Type Select.** This bit selects the transfer data type (40/48=bit, 3-column if set, =1) (32/64-bit, 4-column if cleared, =0) for the corresponding external port FIFO buffer (EPBx). Programs must not change a buffer's DTYPE setting while the buffer is enabled. The buffer's DTYPE setting overrides the internal memory block's setting IMDWx for Normal word width. Whether buffer is set for 48- or 64- bit words, programs must index (IIx) the corresponding DMA channel with a Normal word address; always an even address 64-bit.
8-6	PMODE	**External Port Packing Mode.** These bits select the packing mode for the corresponding external port FIFO buffer (EPBx) as follows: 001=16 external to 32/64 internal packing, 010=16 external to 48 internal packing, 011=32 external to 48 internal packing, 101=8 external to 48 internal packing, 100=32 external to 32/64 internal packing (No pack), 110=8 external to 32/64 internal packing, 000 =111=reserved. Programs must not change a buffer's PMODE setting while the buffer is enabled. For host processor accesses through the external port, the buffer's PMODE setting must match the Host Bus Width (HBW) setting in the SYSCON registers.
9	MSWF	**Most Significant 16-bit Word First During Packing.** When the buffer's PMODE is 001 or 010, this bit selects the packing order of 16-bit words (most significant first set, =1) (least significant first cleared, =0) for the corresponding external port FIFO buffer (EPBx). Programs must not change a buffer's MSWF setting while the buffer is enabled.

Table A-24. External Port DMA Control Registers (DMACx) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
10	MASTER	**Master Mode Enable.** This bit enables (if set, =1) or disables (if cleared, =0) master mode for the corresponding external port FIFO buffer (EPBx). Programs must not change a buffer's MASTER setting while the buffer is enabled. The MASTER, HSHAKE, and EXTERN bits work together to select the external port buffer's mode.
11	HSHAKE	**Handshake Mode Enable.** This bit enables (if set, =1) or disables (if cleared, =0) handshake mode for the corresponding external port FIFO buffer (EPBx). Programs must not change a buffer's HSHAKE setting while the buffer is enabled. The MASTER, HSHAKE, and EXTERN bits work together to select the external port buffer's mode.
12	INTIO	**Single-Word Interrupt Enable.** This bit enables (if set, =1) or disables (if cleared, =0) single-word, non-DMA, interrupt-driven transfers for the corresponding external port FIFO buffer (EPBx). To avoid spurious interrupts, programs must not change a buffer's INTIO setting while the buffer is enabled.
13	EXTERN	**External Handshake Mode Enable.** This bit enables (if set, =1) or disables (if cleared, =0) external handshake mode for the corresponding external port FIFO buffer (EPBx). Programs must not change a buffer's EXTERN setting while the buffer is enabled. The MASTER, HSHAKE, and EXTERN bits work together to select the external port buffer's mode.

Table A-24. External Port DMA Control Registers (DMACx) Bit Definitions (Cont'd)

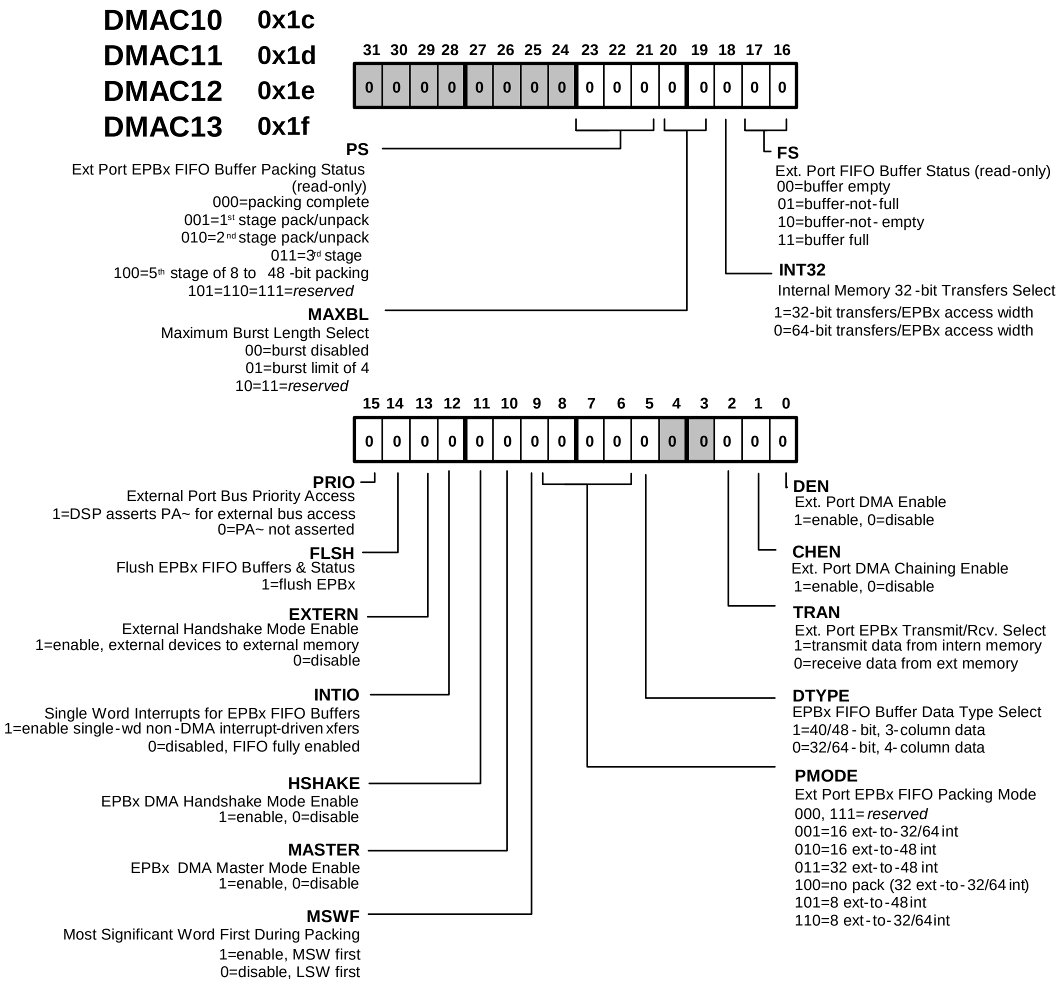
Bit(s)	Name	Definition
14	FLSH	**Flush DMA Buffers & Status.** This bit flushes (when set, =1) settings for the corresponding external port FIFO buffer (EPBx). Flushing these settings does the following: • Clears (=0) the FS and PS status bits • Clears (=0) the FIFO buffer and DMA request counter • Clears (=0) any partially packed words When a program sets (=1) FLSH, the processor flushes the settings and clears (=0) FLSH. There is a two-cycle effect latency in completing the flush operation. Programs must not set a buffer's FLSH during the same write that enables the buffer. Also, programs must not set a buffer's FLSH bit while the DMA channel is active. Programs should determine the channel's active status by reading the corresponding bit in the DMASTAT register.
15	PRIO	**External Port Bus Priority.** This bit selects the external bus access priority level (high if set, =1) (low if cleared, =0) for the corresponding external port FIFO buffer (EPBx). Programs must not change a buffer's PRIO setting while the buffer is enabled. When PRIO is set, the processor asserts the $\overline{\text{PA}}$ pin as part of external bus arbitration for DMA accesses using this buffer. The PRIO bit does not effect internal DMA priority arbitration.

Table A-24. External Port DMA Control Registers (DMACx) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
17-16	FS	**External Port FIFO Buffer Status.** These bits indicate the corresponding FIFO buffer's status as 00=buffer empty, 01=buffer-not-full, 10=buffer-not-empty, 11=buffer full. For transmit (TRAN=1), buffer-not-full means that the buffer has space for one Normal word, and buffer-not-empty means that the buffer has space for two-or-more Normal words. For receive (TRAN=0), buffer-not-full means that the buffer contains one Normal word, and buffer-not-empty means that the buffer contains two-or-more Normal words. Any type of full status (01, 10, or 11) in receive mode indicates that new (unread) data is in the buffer. These bits are read-only. The processor clears these bits when DEN is cleared (changes from 1 to 0).
18	INT32	**Internal Memory 32-bit Transfers Select.** This bit selects the external bus access width (32-bit transfers only if set, =1) (64-bit transfers when possible if cleared, =0) for the corresponding external port FIFO buffer (EPBx). Programs must not change a buffer's INT32 setting while the buffer is enabled. Note that the buffer's DTYPE and internal memory block's IMDWx setting (either can select 40/48-bit transfers) overrides a 32-bit transfers only (INT32 =1) setting.
20-19	MAXBL	**Maximum Burst Length Select.** These bits select the maximum burst transfer length for the corresponding external port FIFO buffer (EPBx) as follows: 00=burst disabled, 01=burst limit of 4, 10=11=reserved. Processors may perform burst accesses to external memory banks only when the bank is configured for synchronous access (EBxAM field in WAIT register). For burst writes, the memory bank's EBxAM must be configured for the one-wait state write, synchronous access mode.

Table A-24. External Port DMA Control Registers (DMACx) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
23-21	PS	**External Port Packing Status.** These bits indicate the corresponding FIFO buffer's packing status as 000=pack complete, 001=1st stage pack/unpack, 010=2nd stage multi-stage pack/unpack, 011= 3rd stage, 100=5th stage of 8 to 48-bit packing, 101=110=111=reserved. These bits are read-only. The processor clears these bits when DEN is cleared (changes from 1 to 0).
31-24		Reserved



DMAC10 0x1c
DMAC11 0x1d
DMAC12 0x1e
DMAC13 0x1f
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
PS
Ext Port EPBx FIFO Buffer Packing Status (read-only)
000=packing complete
001=1st stage pack/unpack
010=2nd stage pack/unpack
011=3rd stage
100=5th stage of 8 to 48 -bit packing
101=110=111=reserved
MAXBL
Maximum Burst Length Select
00=burst disabled
01=burst limit of 4
10=11=reserved
FS
Ext. Port FIFO Buffer Status (read-only)
00=buffer empty
01=buffer-not-full
10=buffer-not- empty
11=buffer full
INT32
Internal Memory 32-bit Transfers Select
1=32-bit transfers/EPBx access width
0=64-bit transfers/EPBx access width
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
PRIO
External Port Bus Priority Access
1=DSP asserts PA~ for external bus access
0=PA~ not asserted
FLSH
Flush EPBx FIFO Buffers & Status
1=flush EPBx
EXTERN
External Handshake Mode Enable
1=enable, external devices to external memory
0=disable
INTIO
Single Word Interrupts for EPBx FIFO Buffers
1=enable single-wd non-DMA interrupt-driven xfers
0=disabled, FIFO fully enabled
HSHAKE
EPBx DMA Handshake Mode Enable
1=enable, 0=disable
MASTER
EPBx DMA Master Mode Enable
1=enable, 0=disable
MSWF
Most Significant Word First During Packing
1=enable, MSW first
0=disable, LSW first
DEN
Ext. Port DMA Enable
1=enable, 0=disable
CHEN
Ext. Port DMA Chaining Enable
1=enable, 0=disable
TRAN
Ext. Port EPBx Transmit/Rcv. Select
1=transmit data from intern memory
0=receive data from ext memory
DTYPE
EPBx FIFO Buffer Data Type Select
1=40/48-bit, 3-column data
0=32/64-bit, 4-column data
PMODE
Ext Port EPBx FIFO Packing Mode
000, 111=reserved
001=16 ext-to-32/64 int
010=16 ext-to-48 int
011=32 ext-to-48 int
100=no pack (32 ext-to-32/64 int)
101=8 ext-to-48 int
110=8 ext-to-32/64 int

Figure A-25. DMAC Register

Internal Memory DMA Index Registers (IIx)

The IIx registers' addresses are: II0A–0x60, II0B–0x80, II1A–0x68, II1B–0x88, II2A–0x70, II2B–0x90, II3A–0x78, II3B–0x98, IILB0–0x30, IISRX–0x30, IILB1–0x38, IISTX–0x38, IIEP0–0x40, IIEP1–0x48, IIEP2–0x50, IIEP3–0x58. The reset value for these registers is undefined. The IIx register is an 18-bit wide register that holds an address and acts as a pointer to memory for a DMA transfer. For more information, see "I/O Processor" on page 6-1.

Internal Memory DMA Modifier Registers (IMx)

The IMx registers' addresses are: IM0A–0x61, IM0B–0x81, IM1A–0x69, IM1B–0x89,IM2A–0x71, IM2B–0x91,IM3A–0x79, IM3B–0x99, IMLB0–0x31, IMSRX–0x31, IMLB1–0x39, IMSTX–0x39, IMEP0–0x41, IMEP1–0x49, IMEP2–0x51, IMEP3–0x59. The reset value for these registers is undefined. The IMx register is a 16-bit wide register that provides the increment or step size by which an IIx register is post-modified during a DMA operation. For more information, see "I/O Processor" on page 6-1.

Internal Memory DMA Count Registers (Cx)

The Cx registers' addresses are: C0A–0x62, C0B–0x82, C1A–0x6a, C1B–0x8a, C2A–0x72, C2B–0x92, C3A–0x7a, C3B–0x9a, CLB0–0x32, CSRX–0x32, CLB1–0x3a, CSTX–0x3a, CEP0–0x42, CEP1–0x4a, CEP2–0x52, CEP3–0x5a. The reset value for these registers is undefined. The Cx registers are 16 bits wide and hold the word count for a DMA transfer. For more information, see "I/O Processor" on page 6-1.

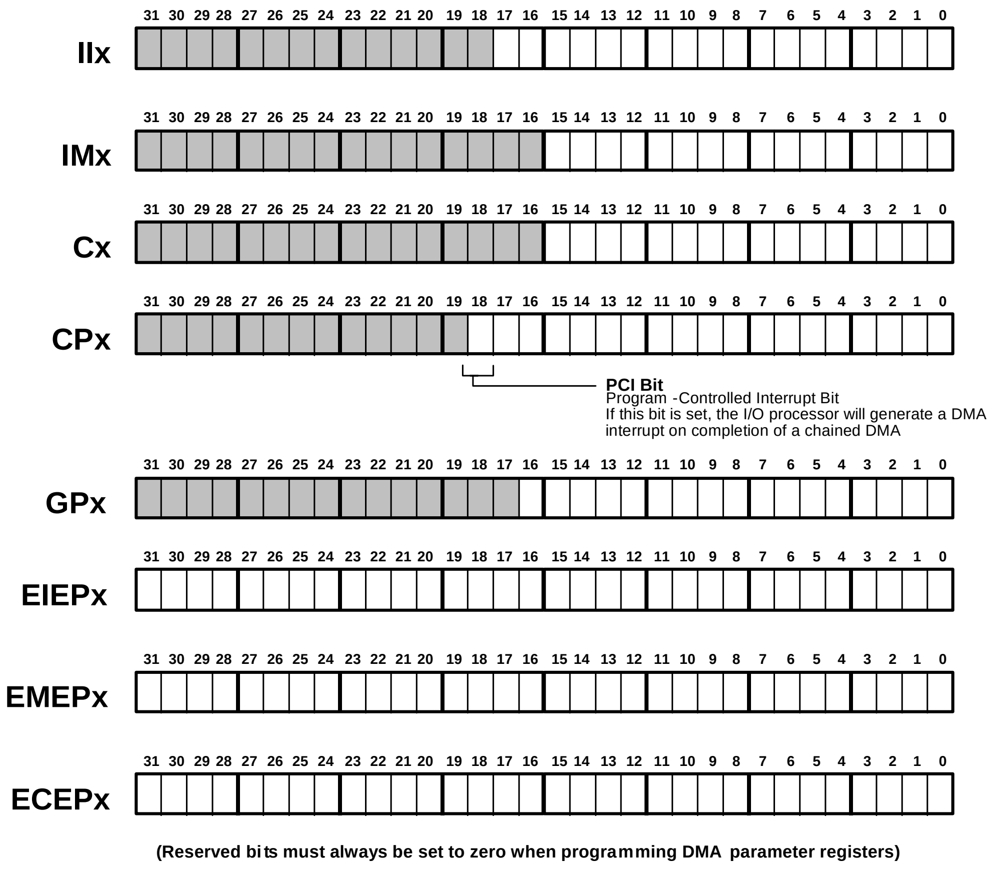

IIx
IMx
Cx
CPx
PCI Bit
Program -Controlled Interrupt Bit
If this bit is set, the I/O processor will generate a DMA interrupt on completion of a chained DMA
GPx
EIEPx
EMEPx
ECEPx
(Reserved bits must always be set to zero when programming DMA parameter registers)

Figure A-26. IOP Parameter Registers

Chain Pointer For Next DMA TCB Registers (CPx)

These registers' addresses are CP0A–0x63, CP0B–0x83, CP1A–0x6B, CP1B–0x8B, CP2A–0x73, CP2B–0x93, CP3A–0x7B, CP3B–0x9B, CPLB0–0x33, CPLB1–0x3B, CPEP0–0x43, CPEP1–0x4B CPEP2–0x53, CPEP3–0x5B. The reset value for these registers is undefined. The CPx registers are 19 bits wide and hold the address for the next transfer control block in a chained DMA operation. For more information, see "I/O Processor" on page 6-1.

General Purpose DMA Registers (GPx)

The GPx registers' addresses are GP0A–0x64, GP0B–0x84, GP1A–0x6C, GP1B–0x8C, GP2A–0x74, GP2B–0x94, GP3A–0x7C, GP3B–0x9C, GPLB0–0x34, GPSRX–0x34, GPLB1–0x3C, GPSTX–0x3C, GPEP0–0x44, GPEP1–0x4C, GPEP2–0x54, GPEP3–0x5C. The GPx registers are 17 bits wide. The reset value for these registers is undefined.

External Memory DMA Index Registers (EIEPx)

These registers' addresses are: EIEP0–0x45, EIEP1–0x4D, EIEP2–0x55, EIEP3–0x5D. The reset value for these registers is undefined. The EIEPx registers hold an external memory address and acts as a pointer to memory for an external port DMA transfer. The 32-bit wide EIEPx registers have more bit space than required to generate external memory addresses. When programming these registers, write zeros to the upper address bits ADDR28 through ADDR31. The lower 28 bits contain a valid address field while the upper MSBs are never generated off-chip in the processor's 254 Mword address space. For more information, see "I/O Processor" on page 6-1.

Only External Port DMA channels have EIEPx registers, because these channels exclusively address ADSP-21161 processor external memory.

External Memory DMA Modifier Registers (EMEPx)

The EMEPx registers' addresses are: EMEP0–0x46, EMEP1–0x4E, EMEP2–0x56, EMEP3–0x5E. The reset value for these registers is undefined. The EMEPx registers provide the increment or step size by which an EIEPx register is post-modified during an external port DMA operation.

The value of EMEPx should be such that after being modified with EMEPx, the value of EIEPx does not fall outside the valid memory range. For more information, see "I/O Processor" on page 6-1.

ⓘ Only External Port DMA channels have EMEPx registers, because these channels exclusively address processor external memory.

External Memory DMA Count Registers (ECEPx)

The ECEPx registers' addresses are: ECEP0–0x47, ECEP1–0x4F, ECEP2–0x57, ECEP3–0x5F. The reset value for these registers is undefined. The ECEPx registers hold the word count for an external port DMA transfer.

When doing multiple transfers, the word count indicated by ECEPx should be such that the value of EIEPx doesn't go beyond the valid memory range. For more information, see "I/O Processor" on page 6-1.

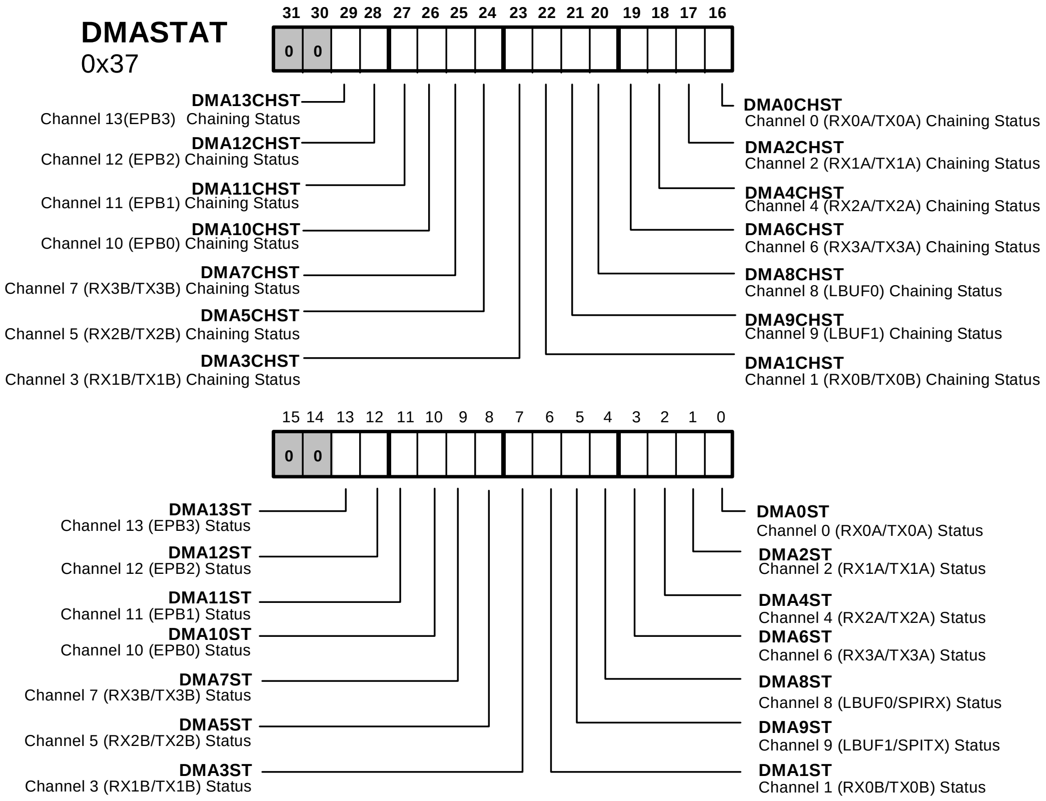
ⓘ Only External Port DMA channels have ECx registers, because these channels exclusively address processor external memory.

DMA Channel Status Register (DMASTAT)

The DMASTAT register's address is 0x37. The reset value for this register is undefined.

The lower bits in the DMASTAT register indicate DMA channel activity. Bits 0 through 13 correspond to channels 0 through 13 and indicate DMA status for each channel as active (if set, =1) or inactive (if cleared, =0). The upper bits in the DMASTAT register indicate DMA chaining status. Bits 16 through 29 correspond to channels 0 through 13 and indicate DMA chaining status for each channel as enabled/pending (if set, =1) or disabled (if cleared, =0). This register is shown in Figure A-27.

Note that there is a single cycle of read latency between a change in a DMA channel's status and the update of its `DMASTAT` bit(s).



DMASTAT
0x37
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0
DMA13CHST Channel 13(EPB3) Chaining Status
DMA12CHST Channel 12 (EPB2) Chaining Status
DMA11CHST Channel 11 (EPB1) Chaining Status
DMA10CHST Channel 10 (EPB0) Chaining Status
DMA7CHST Channel 7 (RX3B/TX3B) Chaining Status
DMA5CHST Channel 5 (RX2B/TX2B) Chaining Status
DMA3CHST Channel 3 (RX1B/TX1B) Chaining Status
DMA0CHST Channel 0 (RX0A/TX0A) Chaining Status
DMA2CHST Channel 2 (RX1A/TX1A) Chaining Status
DMA4CHST Channel 4 (RX2A/TX2A) Chaining Status
DMA6CHST Channel 6 (RX3A/TX3A) Chaining Status
DMA8CHST Channel 8 (LBUF0) Chaining Status
DMA9CHST Channel 9 (LBUF1) Chaining Status
DMA1CHST Channel 1 (RX0B/TX0B) Chaining Status
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0
DMA13ST Channel 13 (EPB3) Status
DMA12ST Channel 12 (EPB2) Status
DMA11ST Channel 11 (EPB1) Status
DMA10ST Channel 10 (EPB0) Status
DMA7ST Channel 7 (RX3B/TX3B) Status
DMA5ST Channel 5 (RX2B/TX2B) Status
DMA3ST Channel 3 (RX1B/TX1B) Status
DMA0ST Channel 0 (RX0A/TX0A) Status
DMA2ST Channel 2 (RX1A/TX1A) Status
DMA4ST Channel 4 (RX2A/TX2A) Status
DMA6ST Channel 6 (RX3A/TX3A) Status
DMA8ST Channel 8 (LBUF0/SPIRX) Status
DMA9ST Channel 9 (LBUF1/SPITX) Status
DMA1ST Channel 1 (RX0B/TX0B) Status

* Channel Active Status: 1=Active [transferring data or waiting to transfer current block, and not transferring TCB]
0= Inactive [DMA transter complete, or in TCB chain loading]

** Channel Chaining Status: 1=Chaining is *Enabled* and currently transferring TCB, or is *Pending* to transfer TCB,
0 = Chaining Disabled

Status does not change on the master ADSP-21161 processor during external port DMA until the external portion is completed (for example, the EPBx buffers are emptied).

If in chain insertion mode (DEN=0, CHEN=1), then *channel chaining status* will never go to a 1. Therefore, test *channel status* to see if it is ready so that your program can rewrite the chain pointer (CPx) register.

Figure A-27. DMASTAT Register

Link Port Buffer Registers (LBUFx)

These registers' addresses are: LBUF0–0xc0, LBUF1–0xc2. The reset value for these registers is undefined. These registers are shown in Figure A-28.

Link port buffers are two levels deep and 48 bits wide. The buffers contain 32- or 48-bit words, depending on the link port's extended word size selected with the LxEXT bit in the port's LCTL register. If the buffer contains 32-bit words, the port aligns the data with the lower 32 bits of the buffer and zero fills the upper 16 bits.

Normally, a DMA process automatically accesses the buffer register for memory transfer. Programs can also access these buffers as registers. However, programs must use the PX register to access the full width of the buffer. A PX register move can access the entire 48 bits of a link buffer using the lower 48 bits of PX.

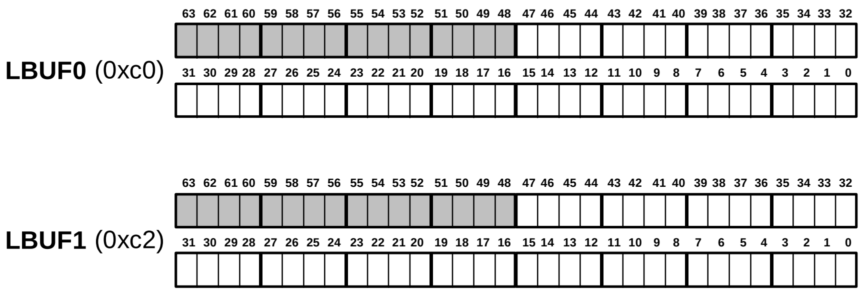


Figure A-28. LBUFx Register

Link Port Buffer Control Register (LCTL)

This register's address is 0xCC. The reset value for this register is 0x0020 0000. Table A-25 and Figure A-29 on page A-97 describe the bit fields within this register. To avoid spurious interrupts, programs should mask Link Service Requests (LSRQ) before modifying the LCTL register. For more information, see "Link Port Service Request & Mask Register (LSRQ)" on page A-98.

Table A-25. Link Port Buffer Control Registers (LCTL) Bit Definitions

Bit(s)	Name	Definition
0	L0EN	**Link Buffer Enable.** This bit enables (if set, =1) or disables (if cleared, =0) link buffer 0 (LBUF0). When the processor disables the buffer (L0EN transitions from high to low), the processor clears the corresponding L0STAT and L0RERR bits.
1	L0DEN	**Link Buffer DMA Enable.** This bit enables (if set, =1) or disables (if cleared, =0) DMA transfers link buffer 0 (LBUF0).
2	L0CHEN	**Link Buffer DMA Chaining Enable.** This bit enables (if set, =1) or disables (if cleared, =0) DMA chaining link buffer 0 (LBUF0)
3	L0TRAN	**Link Buffer Transfer Direction.** This bit selects the transfer direction (transmit if set, =1) (receive if cleared, =0) for link buffer 0 (LBUF0).
4	L0EXT	**Link Buffer Extended Word Size.** This bit selects the transfer extended word size (48-bit if set, =1) (32-bit if cleared, =0) for link buffer 0(LBUF0). Programs must not change a buffer's L0EXT setting while the buffer is enabled. The buffer's L0EXT setting overrides the internal memory block's setting IMDWx for Normal word width. Whether buffer is set for 48- or 32- bit words, programs must index (IIx) the corresponding DMA channel with a Normal word address.
6-5	L0CLKD	**Link Port Clock Divisor.** These bits select the transfer clock divisor for link buffer 0 (LBUF0). The transfer clock equals the processor core clock divided by L0CLKD, where L0CLKD[6-5] is: 01=1, 10=2, 11=3, or 00=4.
7		Reserved

Table A-25. Link Port Buffer Control Registers (LCTL) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
8	L0PDRDE	**Link Port Pulldown Resistor Disable.** This bit disables (if set, =1) or enables (if cleared, =0) the internal pulldown resistors on the L0CLK, L0ACK, and L0DAT7-0 pins of the corresponding unassigned or disabled link port for silicon revisions 0.3, 1.0 and 1.1 and L0CLK and L0ACK for silicon revisions 1.2 and higher. this bit applies to the port which is not necessarily the port assigned to link buffer 0 (LBUF0). For revisions 0.3, 1.0 and 1.1 systems should not leave link port pins (L0CLK, L0ACK, and L0DAT7-0) unconnected without clearing the corresponding L0PDRDE bit or applying an external pulldown. For silicon revisions 1.2 or higher, this applies to L0CLK and L0ACK pins only. In systems where several processors share a link port, only one processor should have this bit cleared. For complete pin descriptions, see Table 13-1 on page 13-4.
9	L0DPWID	**Link Port Data Path Width.** This bit selects the link port data path width (8-bit if set, =1) (4-bit if cleared, =0) for link buffer 0 (LBUF0). Systems using a 4-bit width should connect the lower link port data pins (L0DAT3-0) for data transfers and leave the upper pins (L0DAT7-4) unconnected. In the 4-bit mode, the processor applies pulldowns to the upper pins.
10	L1EN	**Link Buffer Enable.** This bit enables (if set, =1) or disables (if cleared, =0) link buffer 1 (LBUF1). When the processor disables the buffer (L1EN transitions from high to low), the processor clears the corresponding L1STAT and L1RERR bits.
11	L1DEN	**Link Buffer DMA Enable.** This bit enables (if set, =1) or disables (if cleared, =0) DMA transfers link buffer 1(LBUF1).
12	L1CHEN	**Link Buffer DMA Chaining Enable.** This bit enables (if set, =1) or disables (if cleared, =0) DMA chaining link buffer 1(LBUF1).
13	L1TRAN	**Link Buffer Transfer Direction.** This bit selects the transfer direction (transmit if set, =1) (receive if cleared, =0) link buffer 1(LBUF1).

Table A-25. Link Port Buffer Control Registers (LCTL) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
14	L1EXT	**Link Buffer Extended Word Size.** This bit selects the transfer extended word size (48-bit if set, =1) (32-bit if cleared, =0) for link buffer 1(LBUF1). Programs must not change a buffer's L1EXT setting while the buffer is enabled. The buffer's L1EXT setting overrides the internal memory block's setting IMDWx for Normal word width. Whether buffer is set for 48- or 32- bit words, programs must index (IIx) the corresponding DMA channel with a Normal word address.
16-15	L1CLKD	**Link Port Clock Divisor.** These bits select the transfer clock divisor for link buffer 1(LBUF1). The transfer clock equals the processor core clock divided by L1CLKD, where L1CLKD[16-15] is: 01=1, 10=2, 11=3, or 00=4.
17		Reserved
18	L1PDRDE	**Link Port Pulldown Resistor Disable.** This bit disables (if set, =1) or enables (if cleared, =0) the internal pulldown resistors on the L1CLK, L1ACK, and L1DAT7-0 pins of the corresponding unassigned or disabled link port for silicon revisions 0.3, 1.0 and 1.1 and L1CLK and L1ACK for silicon revisions 1.2 and higher. This bit applies to the port, which is not necessarily the port assigned to link buffer 1 (LBUF1). For revisions 0.3, 1.0 and 1.1 systems should not leave link port pins (L1CLK, L1ACK, and L1DAT7-0) unconnected without clearing the corresponding L1PDRDE bit or applying an external pulldown. For silicon revisions 1.2 or higher, this applies to L1CLK and L1ACK pins only. In systems where several DSPs share a link port, only one processor should have this bit cleared. For complete pin descriptions, see Table 13-1 on page 13-4.

Table A-25. Link Port Buffer Control Registers (LCTL) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
19	L1DPWID	**Link Port Data Path Width.** This bit selects the link port data path width (8-bit if set, =1) (4-bit if cleared, =0) for link buffer 1 (LBUF1). Systems using a 4-bit width should connect the lower link port data pins (L1DAT3-0) for data transfers and leave the upper pins (L1DAT7-4) unconnected. In the 4-bit mode, the processor applies pulldowns to the upper pins.
20	LAB0	**Link Port Assignments for LBUF0.** This bit assigns link buffer 0 to link port 1 if set (=1) or link port 0 if cleared (=0).
21	LAB1	**Link Port Assignments for LBUF1.** This bit assigns link buffer 1 to link port 1 if set (=1) or link port 0 if cleared (=0).
23-22	L0STAT	**Link Buffer 0 Status.** These bits identify the status of link buffer 0 as follows: 11=full, 00=empty, 10=one word.
25-24	L1STAT	**Link Buffer 1 Status.** These bits identify the status of link buffer 1 as follows: 11=full, 00=empty, 10=one word.
26	LRERR0	**Receive Packing Error Status for Link Buffer 0.** Indicates if the packed bits in link buffer 0 were receive completely (=0), without error, or incompletely (=1).
27	LRERR1	**Receive Packing Error Status for Link Buffer 1.** Indicates if the packed bits in link buffer 1 were received completely (=0), without error, or incompletely (=1).
31-28		Reserved

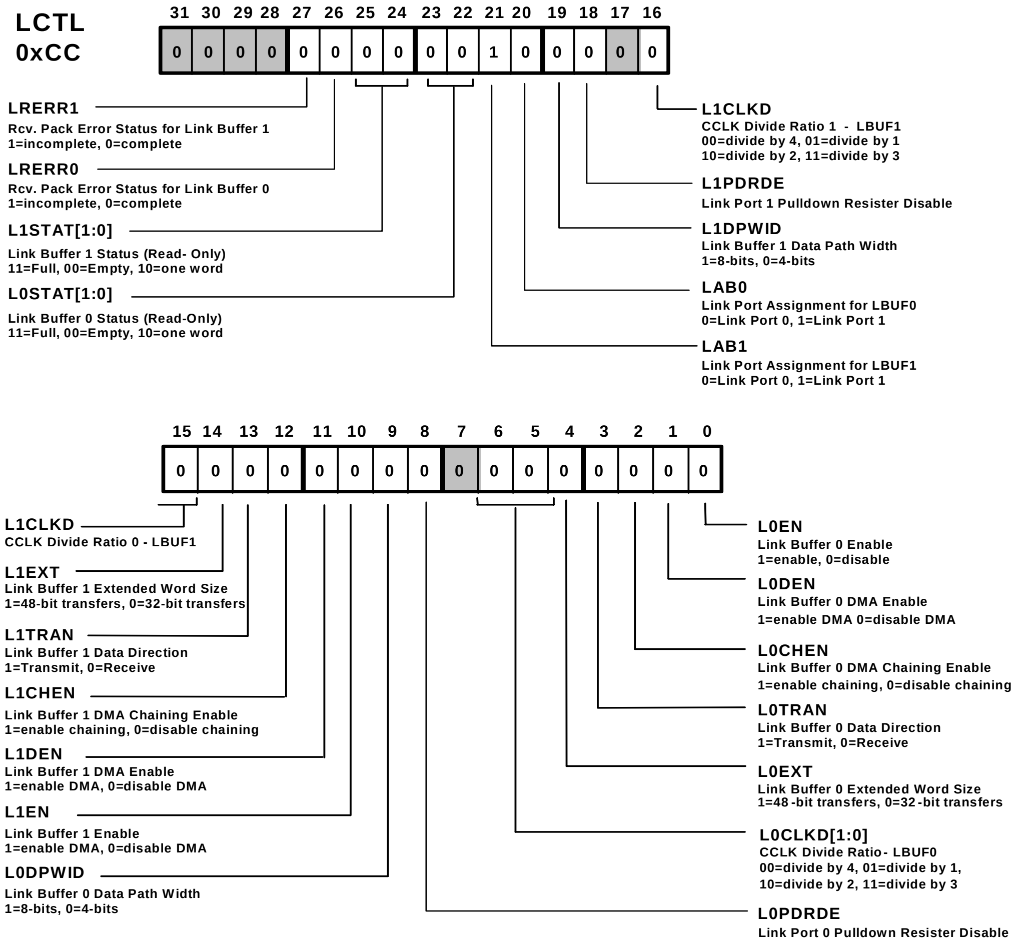

LCTL
0xCC
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
LRERR1
Rcv. Pack Error Status for Link Buffer 1
1=incomplete, 0=complete
LRERR0
Rcv. Pack Error Status for Link Buffer 0
1=incomplete, 0=complete
L1STAT[1:0]
Link Buffer 1 Status (Read- Only)
11=Full, 00=Empty, 10=one word
L0STAT[1:0]
Link Buffer 0 Status (Read-Only)
11=Full, 00=Empty, 10=one word
L1CLKD
CCLK Divide Ratio 1 - LBUF1
00=divide by 4, 01=divide by 1
10=divide by 2, 11=divide by 3
L1PDRDE
Link Port 1 Pulldown Resister Disable
L1DPWID
Link Buffer 1 Data Path Width
1=8-bits, 0=4-bits
LAB0
Link Port Assignment for LBUF0
0=Link Port 0, 1=Link Port 1
LAB1
Link Port Assignment for LBUF1
0=Link Port 0, 1=Link Port 1
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
L1CLKD
CCLK Divide Ratio 0 - LBUF1
L1EXT
Link Buffer 1 Extended Word Size
1=48-bit transfers, 0=32-bit transfers
L1TRAN
Link Buffer 1 Data Direction
1=Transmit, 0=Receive
L1CHEN
Link Buffer 1 DMA Chaining Enable
1=enable chaining, 0=disable chaining
L1DEN
Link Buffer 1 DMA Enable
1=enable DMA, 0=disable DMA
L1EN
Link Buffer 1 Enable
1=enable DMA, 0=disable DMA
L0DPWID
Link Buffer 0 Data Path Width
1=8-bits, 0=4-bits
L0EN
Link Buffer 0 Enable
1=enable, 0=disable
L0DEN
Link Buffer 0 DMA Enable
1=enable DMA 0=disable DMA
L0CHEN
Link Buffer 0 DMA Chaining Enable
1=enable chaining, 0=disable chaining
L0TRAN
Link Buffer 0 Data Direction
1=Transmit, 0=Receive
L0EXT
Link Buffer 0 Extended Word Size
1=48 -bit transfers, 0=32 -bit transfers
L0CLKD[1:0]
CCLK Divide Ratio- LBUF0
00=divide by 4, 01=divide by 1,
10=divide by 2, 11=divide by 3
L0PDRDE
Link Port 0 Pulldown Resister Disable

Figure A-29. LCTL Register

Link Port Service Request & Mask Register (LSRQ)

The LSRQ register's address is 0xD0. This register is described in Table A-26 and Figure A-30. The reset value for this register is 0x0000 0000. The LSRQ register contains transmit and receive mask and status bits for each link port. The mask bits in LSRQ mask (disable if set, =1) or unmask (enable if cleared, =0) the status bits in LSRQ register.

The status bits indicate whether a disabled link port (DEN=0) has a pending service request to receive or transmit data. When an LSRQ receive request status bit is set (LxRRQ=1), another ADSP-21161 processor has requested to send data by setting the link port's clock (LxCLK=1). When an LSRQ transmit request status bit is set (LxTRQ=1), another ADSP-21161 processor has requested more data by setting the link port's acknowledge (LxACK=1).

Table A-26. Link Port Service Request Register (LSRQ) Bit Definitions

Bit(s)	Name	Definition
3-0		Reserved
4	L0TM	**Link Port 0 Transmit Mask.** This bit masks (if set, =1) or unmasks (if cleared, =0) the L0TRQ status bit.
5	L0RM	**Link Port 0 receive mask.** This bit masks (if set, =1) or unmasks (if cleared, =0) the L0RRQ status bit.
6	L1TM	**Link Port 1 Transmit Mask.** This bit masks (if set, =1) or unmasks (if cleared, =0) the L1TRQ status bit.
7	L1RM	**Link Port 1 Receive Mask.** This bit masks (if set, =1) or unmasks (if cleared, =0) the L1RRQ status bit.
19-8		Reserved
20	L0TRQ	**Link Port 0 Transmit Request Status (Read-Only).** If set (=1), indicates that link port 0 is disabled, but L0ACK is set (indicating an external transmit request).

Table A-26. Link Port Service Request Register (LSRQ) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
21	L0RRQ	**Link Port 0 Receive Request Status (Read-Only).** If set (=1), indicates that link port 0 is disabled, but L0CLK is set (indicating an external receive request).
22	L1TRQ	**Link Port 1 Transmit Request Status (Read-Only).** If set (=1), indicates that link port 1 is disabled, but L1ACK is set (indicating an external transmit request).
23	L1RRQ	**Link Port 1 Receive Request Status (Read-Only).** If set (=1), indicates that link port 1 is disabled, but L1CLK is set (indicating an external receive request).
31-24		Reserved

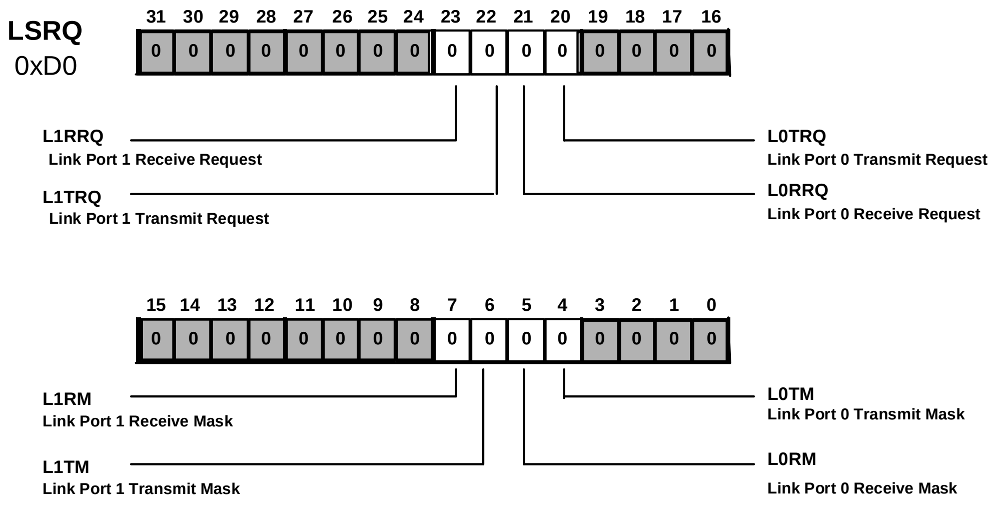


Figure A-30. LSRQ Register

Serial Port Registers

This section provides bit descriptions for all ADSP-21161 SPORT registers.

SPORT Serial Control Registers (SPCTLx)

These registers' addresses are: `SPCTL0`–0x1C0, `SPCTL1`–0x1E0, `SPCTL2`–0x1D0, `SPCTL3`–0x1F0. The reset value for these registers is 0x0000 0000. The `SPCTLx` registers are transmit and receive control registers for the corresponding serial port (SPORT 0, 1, 2 and 3). Table A-27 provides bit descriptions for the SPORT registers. Some these bits are reserved or have different names when the SPORT is in multichannel or I^{2}S mode. The table notes these difference.

- Figure A-31 on page A-105 provides bit definitions for the `SPCTLx` register in serial mode.
- Figure A-32 on page A-106 provides bit definitions for the `SPCTLx` register in I^{2}S mode.
- Figure A-33 on page A-107 provides bit definitions for SPORTS 0 and 1 (receive) in multichannel mode.
- Figure A-34 on page A-108 provides bit definitions for SPORTS 2 and 3 (transmit) in multichannel mode.

When changing SPORT operating modes, programs should clear a serial port's control register before writing new settings to the control register.

Table A-27. Serial Port Control Registers (SPCTLx) Bit Definitions

Bit(s)	Name	Definition
0	SPEN_A	**Serial Port Enable A.** This bit enables (if set, =1) or disables (if cleared, =0) the corresponding serial port A channel. This bit is reserved when the SPORT is in multichannel mode.
2-1	DTYPE	**Data Type Select.** These bits select the data type formatting for normal and multi-channel transmissions as follows: Normal / Multi / Data Type Formatting 00 / x0 / Right-justify, zero-fill unused MSBs 01 / x1 / Right-justify, sign-extend unused MSBs 10 / 0x / Compand using μ-law 11 / 1x / Compand using A-law
3	SENDN	**Serial Word Endian Select.** This bit selects little endian words (LSB first, if set, =1) or big endian words (MSB first, if cleared, =0).
8-4	SLEN	**Serial Word Length Select.** These bits select the word length in bits. Word sizes can be from 3-bit (SLEN=2) to 32-bit (SLEN=31).
9	PACK	**16-bit to 32-Bit Word Packing Enable.** This bit enables (if set, =1) or disables (if cleared, =0) 16- to 32-bit word packing.
10	ICLK	**Internal Transmit Clock Select.** This bit selects the internal transmit clock (if set, =1) or external transmit clock (if cleared, =0). This bit applies to processor serial and multichannel modes for SPCTL0 and SPCTL1 registers.
	MSTR (I^2S mode only)	In I^2S mode, this bit selects the word source and internal transmit clock (if set, =1) or external transmit clock (if cleared, =0)
11	OPMODE	**Sport Operation Mode.** This bit selects the I^2S mode if set (=1) or processor Serial mode/Multichannel mode if cleared (=0).
12	CKRE	**Clock Rising Edge Select.** This bit selects whether the serial port uses the rising edge (if set, =1) or falling edge (if cleared, =0) of the clock signal for sampling data and the frame sync. This bit is reserved when the SPORT is in I^2S mode.

Table A-27. Serial Port Control Registers (SPCTLx) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
13	FSR	**Frame Sync Required Select.** This bit selects whether the serial port requires (if set, =1) or does not require (if cleared, =0) a transfer frame sync. This bit is reserved when the SPORT is in I^{2}S mode and multichannel mode.
14	IFS (IRFS)	**Internally Frame Sync Select.** This bit selects whether the serial port uses an internal generated FS (if set, =1) or uses an external FS (if cleared, =0). This bit is reserved when the SPORT is in I^{2}S mode and multichannel transmit mode.
15	DITFS	**Data Independent Transmit Frame Sync Select.** This bit selects whether the serial port uses a data-independent transmit FS (sync at selected interval, if set, =1) or uses a data-dependent TFS (sync when data in TX, if cleared, =0) when DDIR=1. This bit is reserved when the SPORT is in multichannel mode.
16	LFS (LRFS, LTDV)	**Low Active Frame Sync Select.** This bit selects an active low FS (if set, =1) or active high FS (if cleared, =0).
17	LAFS	**Late Transmit Frame Sync Select.** This bit selects a late FS (FS during first bit, if set, =1) or an early FS (FS before first bit, if cleared, =0). This bit is reserved when the SPORT is in I^{2}S mode and multichannel mode.
18	SDEN_A	**Serial Port DMA Enable A.** This bit enables (if set, =1) or disables (if cleared, =0) the serial port's A channel DMA.
19	SCHEN_A	**Serial Port DMA Chaining Enable A.** This bit enables (if set, =1) or disables (if cleared, =0) serial port's channel A DMA chaining.

Table A-27. Serial Port Control Registers (SPCTLx) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
20	SDEN_B	**Serial Port DMA Enable B.** This bit enables (if set, =1) or disables (if cleared, =0) the serial port's channel B DMA. This bit is reserved when the SPORT is in multichannel mode.
21	SCHEN_B	**Serial Port DMA Chaining Enable B.** This bit enables (if set, =1) or disables (if cleared, =0) serial port's channel B DMA chaining. This bit is reserved when the SPORT is in multichannel mode.
22	FS_BOTH	**FS Both Enable.** This bit issues WS if data is present in **both** transmit buffers if set (=1). If cleared (=0), WS is issued if data is present in either transmit buffers. This bit is reserved when the SPORT is in multichannel mode.
23		Reserved
24	SPEN_B	**Serial Port Enable B.** This bit enables (if set, =1) or disables (if cleared, =0) the corresponding serial port B channel. This bit is reserved when the SPORT is in multichannel mode.
25	DDIR	**Data Direction Control.** This bit activates transmit buffers TXnA or TXnB if set (=1) or enables receive buffers RXnA or RXnB if cleared (=0). This bit is reserved when the SPORT is in multichannel mode.
26	DERR_B	**DXB Error Status (Sticky, Read-Only).** This bit indicates whether the serial transmit operation has underflowed (if set, =1 and DDIR=1) or a receive operation has overflowed (if cleared, =0 and DDIR=0) in the DXB data buffer. This bit is reserved when the SPORT is in multichannel mode.

Table A-27. Serial Port Control Registers (SPCTLx) Bit Definitions (Cont'd)

Bit(s)	Name	Definition
28-27	DXS_B	**DXB Data Buffer Status (Read-Only).** These bits indicate the status of the serial port's DXB data buffer as follows: 11=full, 00=empty, 10=partially full. This bit is reserved when the SPORT is in multichannel mode.
29	DERR_A (ROVF_A, TUVF_A)	**DXA Error Status (Sticky, Read-Only).** This bit indicates whether the serial transmit operation has underflowed (if set, =1 and DDIR=1) or a receive operation has overflowed (if cleared, =0 and DDIR=0) in the DXA data buffer.
31-30	DXS_A (RXS_A, TXS_A)	**DXA Data Buffer Status (Read-Only).** These bits indicate the status of the serial port's DXA data buffer as follows: 11=full, 00=empty, 10=partially full.

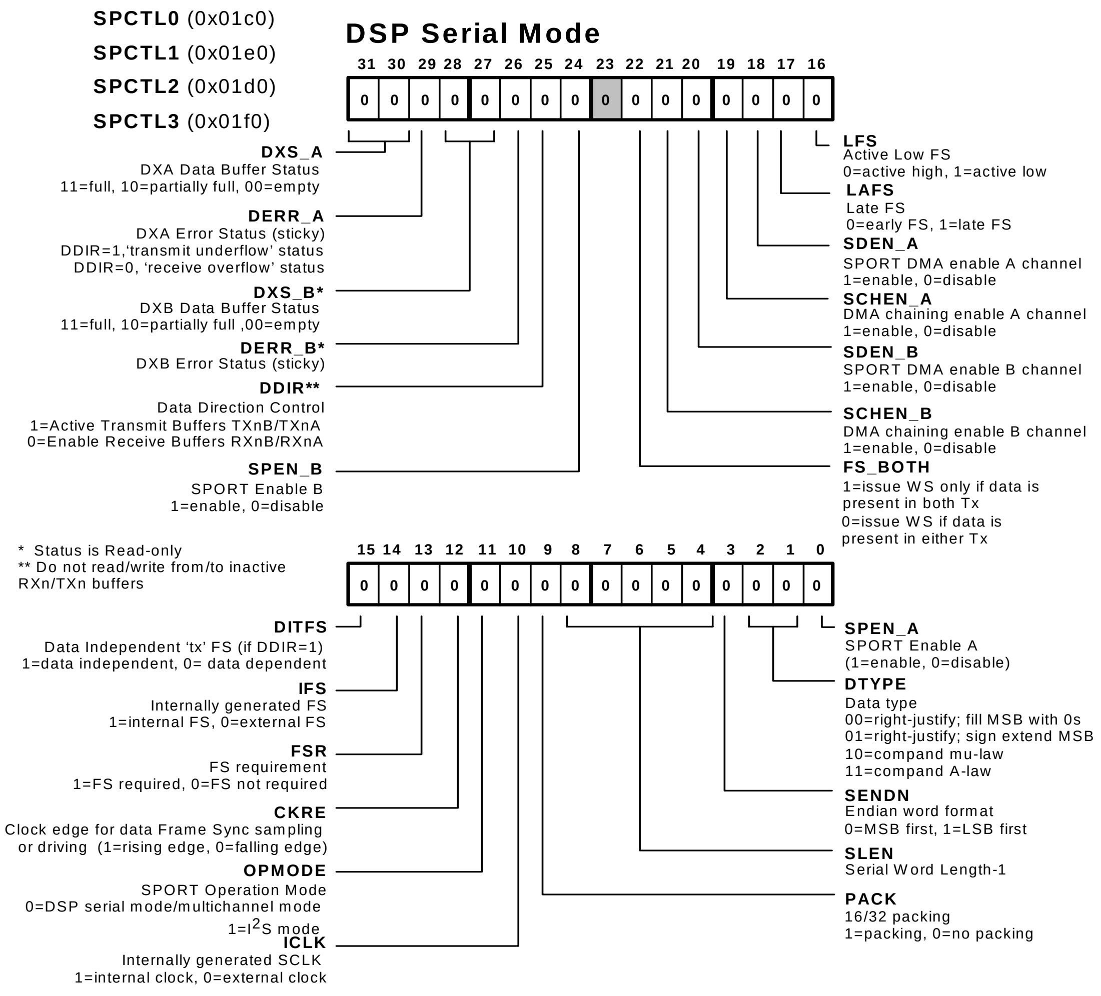
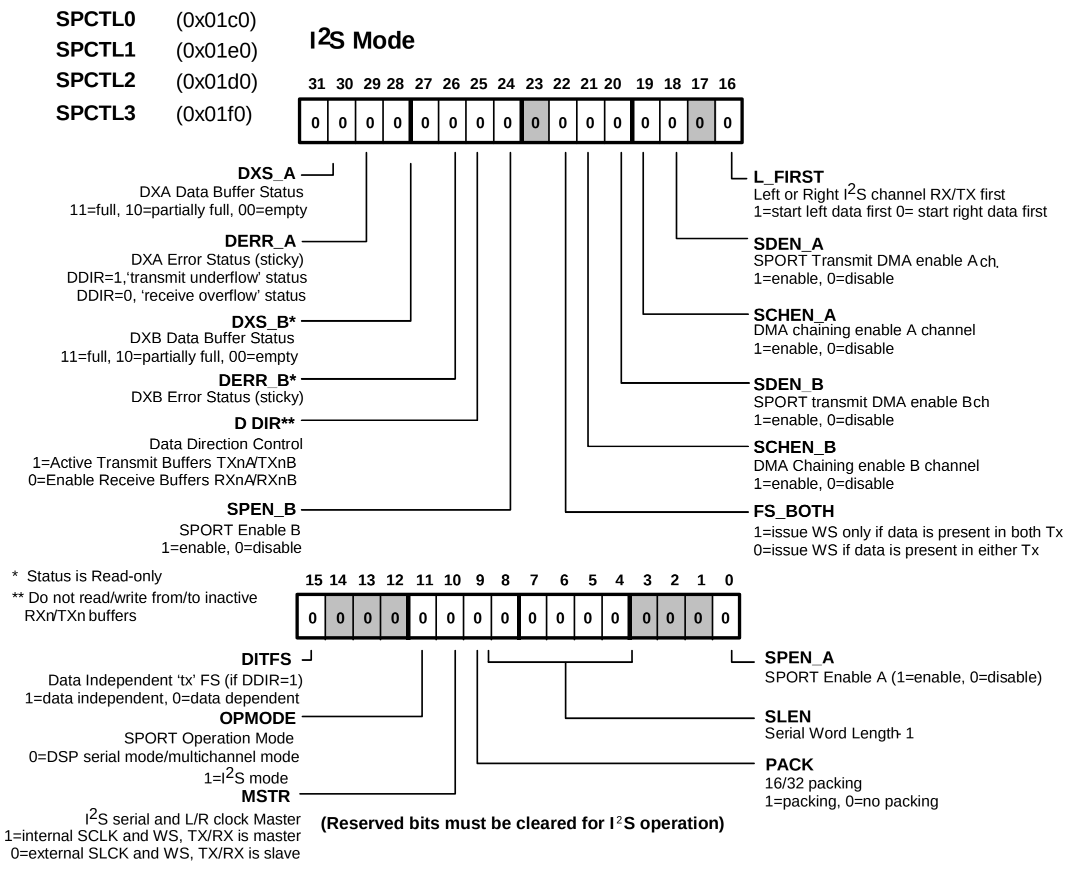

SPCTL0 (0x01c0)
SPCTL1 (0x01e0)
SPCTL2 (0x01d0)
SPCTL3 (0x01f0)
DSP Serial Mode
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
DXS_A
DXA Data Buffer Status
11=full, 10=partially full, 00=empty
DERR_A
DXA Error Status (sticky)
DDIR=1,'transmit underflow' status
DDIR=0, 'receive overflow' status
DXS_B*
DXB Data Buffer Status
11=full, 10=partially full ,00=empty
DERR_B*
DXB Error Status (sticky)
DDIR**
Data Direction Control
1=Active Transmit Buffers TXnB/TXnA
0=Enable Receive Buffers RXnB/RXnA
SPEN_B
SPORT Enable B
1=enable, 0=disable
LFS
Active Low FS
0=active high, 1=active low
LAFS
Late FS
0=early FS, 1=late FS
SDEN_A
SPORT DMA enable A channel
1=enable, 0=disable
SCHEN_A
DMA chaining enable A channel
1=enable, 0=disable
SDEN_B
SPORT DMA enable B channel
1=enable, 0=disable
SCHEN_B
DMA chaining enable B channel
1=enable, 0=disable
FS_BOTH
1=issue WS only if data is present in both Tx
0=issue WS if data is present in either Tx
* Status is Read-only
** Do not read/write from/to inactive RXn/TXn buffers
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
DITFS
Data Independent 'tx' FS (if DDIR=1)
1=data independent, 0= data dependent
IFS
Internally generated FS
1=internal FS, 0=external FS
FSR
FS requirement
1=FS required, 0=FS not required
CKRE
Clock edge for data Frame Sync sampling or driving (1=rising edge, 0=falling edge)
OPMODE
SPORT Operation Mode
0=DSP serial mode/multichannel mode
1=I²S mode
ICLK
Internally generated SCLK
1=internal clock, 0=external clock
SPEN_A
SPORT Enable A
(1=enable, 0=disable)
DTYPE
Data type
00=right-justify; fill MSB with 0s
01=right-justify; sign extend MSB
10=compand mu-law
11=compand A-law
SENDN
Endian word format
0=MSB first, 1=LSB first
SLEN
Serial Word Length-1
PACK
16/32 packing
1=packing, 0=no packing

Figure A-31. SPCTL Register – DSP Serial Mode



SPCTL0 (0x01c0)
SPCTL1 (0x01e0)
SPCTL2 (0x01d0)
SPCTL3 (0x01f0)
I2S Mode
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
DXS_A
DXA Data Buffer Status
11=full, 10=partially full, 00=empty
DERR_A
DXA Error Status (sticky)
DDIR=1,'transmit underflow' status
DDIR=0, 'receive overflow' status
DXS_B*
DXB Data Buffer Status
11=full, 10=partially full, 00=empty
DERR_B*
DXB Error Status (sticky)
D DIR**
Data Direction Control
1=Active Transmit Buffers TXnA/TXnB
0=Enable Receive Buffers RXnA/RXnB
SPEN_B
SPORT Enable B
1=enable, 0=disable
L_FIRST
Left or Right I2S channel RX/TX first
1=start left data first 0= start right data first
SDEN_A
SPORT Transmit DMA enable A ch.
1=enable, 0=disable
SCHEN_A
DMA chaining enable A channel
1=enable, 0=disable
SDEN_B
SPORT transmit DMA enable B ch
1=enable, 0=disable
SCHEN_B
DMA Chaining enable B channel
1=enable, 0=disable
FS_BOTH
1=issue WS only if data is present in both Tx
0=issue WS if data is present in either Tx
* Status is Read-only
** Do not read/write from/to inactive RXn/TXn buffers
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
DITFS
Data Independent 'tx' FS (if DDIR=1)
1=data independent, 0=data dependent
OPMODE
SPORT Operation Mode
0=DSP serial mode/multichannel mode
1=I2S mode
MSTR
I2S serial and L/R clock Master
1=internal SCLK and WS, TX/RX is master
0=external SLCK and WS, TX/RX is slave
SPEN_A
SPORT Enable A (1=enable, 0=disable)
SLEN
Serial Word Length- 1
PACK
16/32 packing
1=packing, 0=no packing
(Reserved bits must be cleared for I2S operation)

Figure A-32. SPCTLx Register–I^{2}S Mode

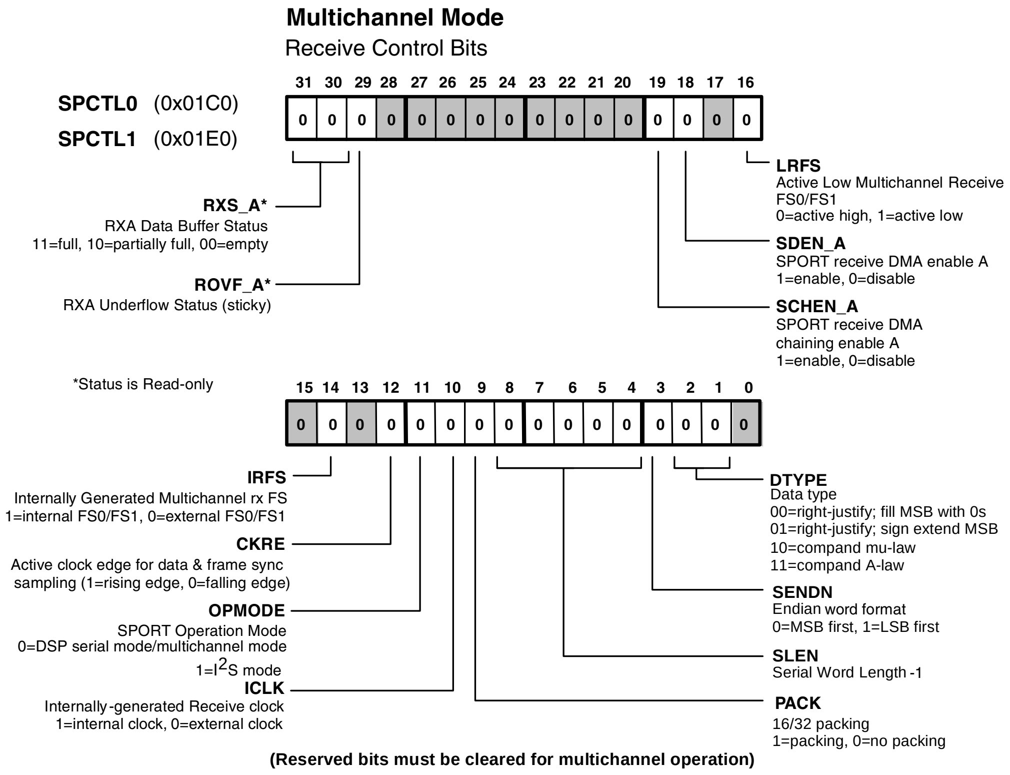

Multichannel Mode
Receive Control Bits
SPCTL0 (0x01C0)
SPCTL1 (0x01E0)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RXS_A*
RXA Data Buffer Status
11=full, 10=partially full, 00=empty
ROVF_A*
RXA Underflow Status (sticky)
LRFS
Active Low Multichannel Receive FS0/FS1
0=active high, 1=active low
SDEN_A
SPORT receive DMA enable A
1=enable, 0=disable
SCHEN_A
SPORT receive DMA chaining enable A
1=enable, 0=disable
*Status is Read-only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
IRFS
Internally Generated Multichannel rx FS
1=internal FS0/FS1, 0=external FS0/FS1
CKRE
Active clock edge for data & frame sync sampling (1=rising edge, 0=falling edge)
OPMODE
SPORT Operation Mode
0=DSP serial mode/multichannel mode
1=I2S mode
ICLK
Internally-generated Receive clock
1=internal clock, 0=external clock
DTYPE
Data type
00=right-justify; fill MSB with 0s
01=right-justify; sign extend MSB
10=compand mu-law
11=compand A-law
SENDN
Endian word format
0=MSB first, 1=LSB first
SLEN
Serial Word Length -1
PACK
16/32 packing
1=packing, 0=no packing
(Reserved bits must be cleared for multichannel operation)

Figure A-33. SPCTL0 and SPCTL1 Registers

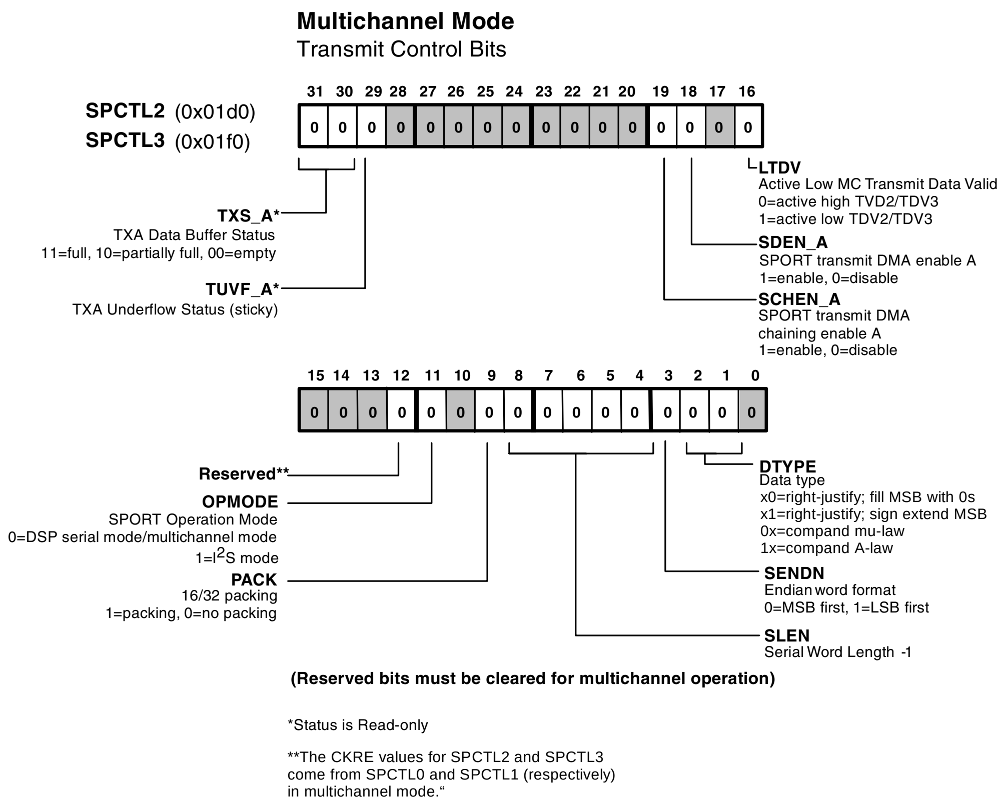


Figure A-34. SPCTL2 and SPCTL3 Registers

SPORT Multichannel Control Registers (SPxyMCTL)

These registers' addresses are `SP02MCTL`–0x1DF, `SP13MCTL`–0x1FF. The `SP02MCTL` register is the multichannel control register for SPORTs 0 and 2. The `SP13MCTL` register is the multichannel control register for SPORTs 1 and 3. The reset value for these registers is undefined. These registers are described in Table A-28 and Figure A-35.

Table A-28. SPORT Multichannel Control Register Bit Definitions

Bit(s)	Name	Definition
0	MCE	**Multichannel Mode Enable.** Standard and multichannel modes only. Bit 0 in the SP02MCTL and SP13MCTL registers.One of two configuration bits that enable and disable multichannel mode on both the receive or transmit serial port channels. See also, OPMODE. 0 = Disable multichannel operation. 1 = Enable multichannel operation if OPMODE=0.
4-1	MFD	**Multichannel Frame Delay.** These bits set the interval, in number of serial clock cycles, between the multichannel frame sync pulse and the first data bit. These bits provide support for different types of T1 interface devices. Valid values range are from 0 to 15 with bits SP02MCTL[4:1] or SP13MCTL[4:1]. Values of 1 to15 correspond to the number of intervening serial clock cycles. A value of 0 corresponds to no delay. The multichannel frame sync pulse is concurrent with first data bit.
11-5	NCH	**Number of Multichannel Slots** (minus one).These bits select the number of channel slots (maximum of 128) to use for multichannel operation.Valid values for actual number of channel slots range from 1 to 128. Use this formula to calculate the value for NCH: NCH = Actual number of channel slots -1.

Table A-28. SPORT Multichannel Control Register Bit Definitions

Bit(s)	Name	Definition
12	SPL	**SPORT Loopback Mode.** This bit enables if set (=1) or disables if cleared (=0) the channel loopback mode. Loopback mode enables developers to run internal tests and to debug applications. Loopback only works under the following SPORT configurations: SPORT0 (configured as a receiver or transmitter) together with SPORT2 (configured as a transmitter or receiver). SPORT0 can only be paired with SPORT2, controlled via the `SPL` bit in the SP02MCTL register. SPORT1 (configured as a receiver or transmitter) together with SPORT3 (configured as a transmitter or receiver). SPORT1 can only be paired with SPORT3, controlled via the SPL bit in the SP13MCTL register. Either of the two paired SPORTs can be set up to transmit or receive, depending on their DDIR bit configurations.
15-13		Reserved
22-16	CHNL	**Current Channel Selected** (Read-Only, Sticky). These bits identify the currently selected transmit channel slot (0 to 127).
31-23		Reserved

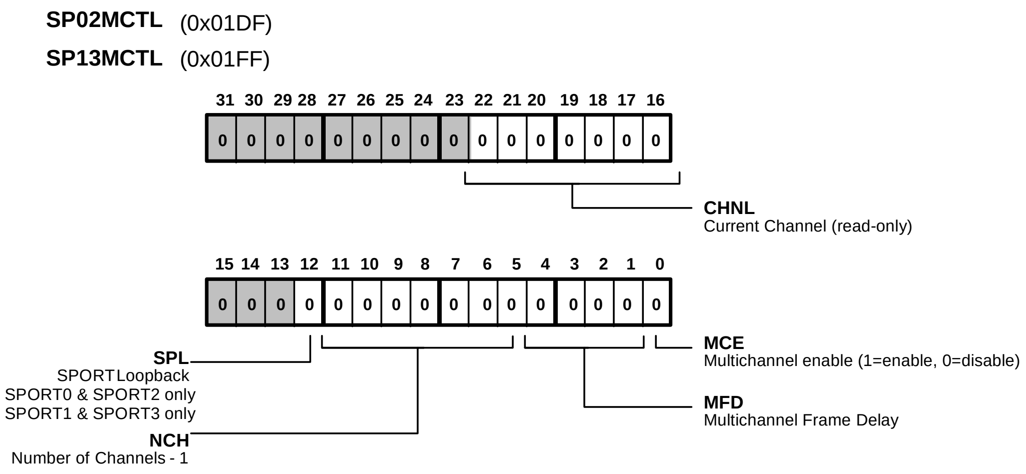


Figure A-35. SP02MCTL and SP13MCTL Registers

SPORT Transmit Buffer Registers (TXx)

The `TXx` registers' addresses are: `TX0A`–0x1C1, `Tx0B`–0x1C2, `Tx1A`–0x1E1, `Tx1B`–0x1E2, `Tx2A`–0x1D1, `Tx2B`–0x1D2, `Tx3A`–0x1F1, `Tx3B`–0x1F2. The reset value for these registers is undefined. The 32-bit `TXx` registers hold the output data for serial port transmit operations. For more information on how transmit buffers work, see "Transmit and Receive Data Buffers" on page 10-30.

SPORT Receive Buffer Registers (RXx)

The `RXx` registers' addresses are: `Rx0A`–0x1C3, `Rx0B`–0x1C4, `Rx1A`–0x1E3, `Rx1B`–0x1E4, `Rx2A`–0x1D3, `Rx2B`–0x1D4, `Rx3A`–0x1F3, `Rx3B`–0x1F4. The reset value for these registers is undefined. The 32-bit `RXx` registers hold the input data from serial port receive operations. For more information on how receive buffers work, see "Transmit and Receive Data Buffers" on page 10-30.

SPORT Divisor Registers (DIVx)

The DIVx registers' addresses are: DIV0–0x1C5, DIV1–0x1E5, DIV2–0x1D5, DIV3–0x1F5 (shown in Figure A-36). The reset value for these registers is undefined. These registers contain two fields:

- Bits 15-0 are CLKDIV. These bits select the Serial Clock Divisor for internally generated SCLK as follows:

$$\mathrm{CLKDIV} = \frac{f_{\mathrm{CCLK}}}{2(f_{\mathrm{SCLK}})} - 1$$

- Bits 31-16 are FSDIV. These bits select the Frame Sync Divisor for internally generated TFS as follows:

$$\mathrm{FSDIV} = \frac{f_{\mathrm{SCLK}}}{f_{\mathrm{SFS}}} - 1$$

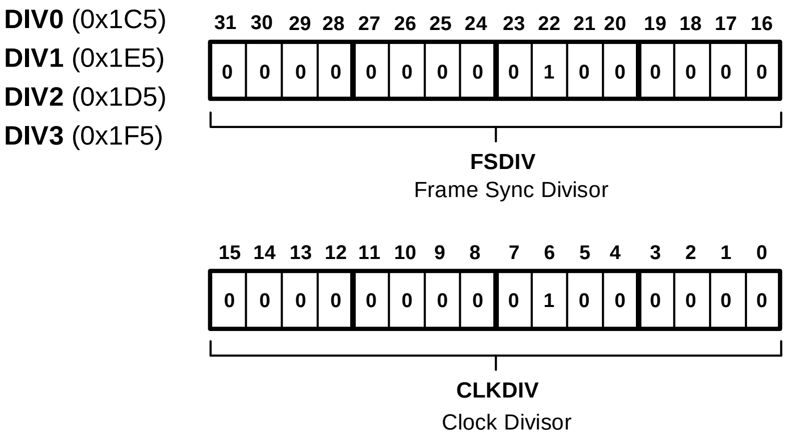


Figure A-36. DIVx Register

SPORT Count Registers (CNTx)

The CNTx registers' addresses are: CNT0–0x1C6, CNT1–0x1E6, CNT2–0x1D6, CNT3–0x1F6. The reset value for these registers is undefined. The CNTx registers provides status information for the internal clock and frame sync.

SPORT Transmit Select Registers (MT2CSx and MT3CSx)

The MT2CSx and MT3CSx registers' addresses are: MT2CS0–0x1D7, MT2CS1–0x1D9, MT2CS2–0x1DB, MT2CS3–0x1DD, MT3CS0–0x1F7, MT3CS1–0x1F9, MT3CS2–0x1FB, MT3CS3–0x1FD. The reset value for these registers is undefined.

Each bit, 31-0, set (=1) in one of four MTxCSx registers correspond to an active transmit channel, 127-0, on a multichannel mode serial port. When the MT2CSx and MT3CSx registers activate a channel, the serial port transmits the word in that channel's position of the data stream. When a channel's bit in the MTCSx register is cleared (=0), the serial port's DT (data transmit) pin three-states during the channel's transmit time slot.

SPORT Transmit Compand Registers (MT2CCSx and MT3CCSx)

The MT2CCSx and MT3CCSx registers' addresses are: MT2CCS0–0x1D8, MT2CCS1–0x1DA, MT2CCS2–0x1DC, MT2CCS3–0x1DE, MT3CCS0–0x1F8, MT3CCS1–0x1FA, MT3CCS2–0x1FC, MT3CCS3–0x1FE. The reset value for these registers is undefined.

Each bit, 31-0, set (=1) in one of four MTxCCSx registers correspond to an companded transmit channel, 127-0, on a multichannel mode serial port. When the MTCCSx register activates companding for a channel, the serial port applies the companding from the serial port's DTYPE selection to the transmitted word in that channel's position of the data stream. When a channel's bit in the MTCCSx register is cleared (=0), the serial port does not compand the output during the channel's receive time slot.

SPORT Receive Select Registers

The MRCSx registers' addresses are: MR0CS0–0x1C7, MR0CS1–0x1C9, MR0CS2–0x1CB, MR0CS3–0x1CD, MR1CS0–0x1E7, MR1CS1–0x1E9, MR1CS2–0x1EB, MR1CS3–0x1ED. The reset value for these registers is undefined.

Each bit, 31-0, set (=1) in one of the four MRCSx registers corresponds to an active receive channel, 127-0, on a multichannel mode serial port. When the MRCSx register activates a channel, the serial port receives the word in that channel's position of the data stream and loads the word into the RXx buffer. When a channel's bit in the MRCSx register is cleared (=0), the serial port ignores any input during the channel's receive time slot.

SPORT Receive Compand Registers

These registers' addresses are: MR0CCS0–0x1C8, MR0CCS1–0x1CA, MR0CCS2–0x1CC, MR0CCS3–0x1CE, MR1CCS0–0x1E8, MR1CCS1–0x1EA, MR1CCS2–0x1EC, MR1CCS3–0x1EE. The reset value for these registers is undefined.

Each bit, 31-0, set (=1) in the MR0CCSx and MR1CCSx registers correspond to an companded receive channel, 127-0, on a multichannel mode serial port. When one of the four MR0CCSx and MR1CCSx registers activate companding for a channel, the serial port applies the companding from the serial port's DTYPE selection to the received word in that channel's position of the data stream. When a channel's bit in the MR0CCSx and MR1CCSx registers are cleared (=0), the serial port does not compand the input during the channel's receive time slot.

Serial Peripheral Interface Registers

The following sections provide descriptions of the registers used in setting up the processor's SPI interface.

SPI Port Status Register

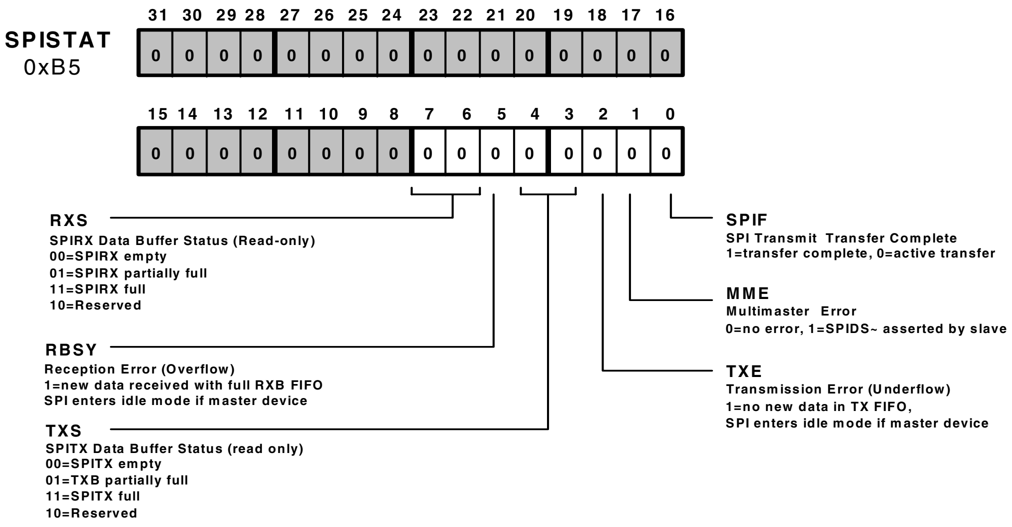
This register's address is 0xB5 and it is described in Table A-29 and Figure A-37. The reset value for this register is undefined.The `SPISTAT` register is a read-only register used to detect when an SPI transfer is complete, if transmission or reception errors occur, and the status of the `SPITX` and `SPIRX` FIFOs.

Table A-29. SPI Status Register Bit Descriptions

Bit(s)	Name	Definition
0	SPIF	**SPI Transmit or Receive Transfer Complete.** This bit is set (=1) when the SPI transfer is complete and one of the following conditions is met: SPRINT = 1 and receive buffer full —or— RDMAEN = 1 and receive buffer full —or— SPTINT = 1 and transmit buffer empty —or— TDMAEN = 1 and transmit buffer empty
1	MME	**Multimaster Error.** This bit is set when a device that is not currently the master device tries to become the master by driving a $\overline{\text{SPIDS}}$ signal while the current master device is communicating to SPI slave devices.
2	TXE	**Transmission Error.** This bit is set when SPI is Slave/Master, SPTINT = 1 or TDMAEN = 1, but there is no data in SPITX FIFO. If you are not servicing the interrupt quickly enough and not updating the contents of SPITX, this bit is set. In master mode, this means an end of operation and SPI going into idle mode.
4-3	TXS	**Transmit Data Buffer Status.** These bits indicate the status of the SPITX data buffer status (read only) as follows: 00 = empty, 01 = partially full, 11 = full.

Table A-29. SPI Status Register Bit Descriptions (Cont'd)

Bit(s)	Name	Definition
5	RBSY	**Reception Error**. This bit is set when a data is received with receive buffer full. Either RDMAEN = 1 or SPRINT = 1 and the receive buffer is full. In master mode, this means an end of operation and SPI going into idle mode.
7-6	RXS	**Receive Data Buffer Status.** These bits indicate the status of the SPIRX data buffer status (read only) as follows: 00 = empty, 01 = partially full, 11 = full.
31-8		Reserved



SPISTAT
0xB5
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RXS
SPIRX Data Buffer Status (Read-only)
00=SPIRX empty
01=SPIRX partially full
11=SPIRX full
10=Reserved
RBSY
Reception Error (Overflow)
1=new data received with full RXB FIFO
SPI enters idle mode if master device
TXS
SPITX Data Buffer Status (read only)
00=SPITX empty
01=TXB partially full
11=SPITX full
10=Reserved
SPIF
SPI Transmit Transfer Complete
1=transfer complete, 0=active transfer
MME
Multimaster Error
0=no error, 1=SPIDS~ asserted by slave
TXE
Transmission Error (Underflow)
1=no new data in TX FIFO,
SPI enters idle mode if master device

Figure A-37. SPISTAT Register

SPI Control Register (SPICTL)

This register's address is 0xB4 and it is described in Table A-30 and Figure A-38. The reset value for these registers is undefined. The SPI Control Register (SPICTL) register is used to configure and enable the SPI system. This register is used to set up SPI configurations such as selecting the device as a master or slave or determining the data transfer rate and word size.

Table A-30. SPI Control Register Bit Descriptions

Bit(s)	Name	Function
0	SPIEN	**SPI Port Enable.** This bit enables (if set, =1) or disables (if cleared, =0) the SPI system.
1	SPRINT	**SPIRX Interrupt Enable.** This bit enables (if set, =1) or disables (if cleared, =0) an SPI interrupt. An interrupt is generated when the receive buffer is not empty.
2	SPTINT	**SPITX Interrupt Enable.** This bit enables (if set, =1) or disables (if cleared, =0) an SPI interrupt. An interrupt is generated when the transmit buffer is not full.
3	MS	**Master Select.** This bit selects the device as a master device (if set, =1) or a slave device (if cleared, =0).
4	CP	**Clock Polarity.** This bit selects the clock polarity. SPICLK high is the idle state (if set, =1), or SPICLK low is the idle state (if cleared, =0).
5	CPHASE	**Clock Phase.** This bit selects the clock phase transfer format. When set (=1), the SPICLK starts toggling at the beginning of the first data transfer bit. When cleared (=0), the SPICLK starts toggling at the middle of the first data transfer bit. For more information, see Figure 11-7 on page 11-22.
6	DF	**Data Format.** This bit selects the data format. When set (=1), the MSB is sent/received first. When cleared (=0), the LSB is sent/received first.
7-8	WL	**Word Length.** This bit selects the word length as follows: 00 = 8 bits, 01 = 16 bits, 11 = 32 bits,10 = reserved.

Table A-30. SPI Control Register Bit Descriptions (Cont'd)

Bit(s)	Name	Function
9-12	BAUDR	**Baud Rate.** These bits define the SPICLK frequency per the following equation: SPICLK baud rate= Core clock / $2^{(2 + BR)}$
13	TDMAEN	**Transmit DMA Enable.** This bit enables (if set, =1) or disables (if cleared, =0) DMA transfers to the transmit buffer. At SPI boot this bit is 0.
Bits 14 to 24 are controlled during master mode.		
14	PSSE	**Programmable Slave Select Enable.** This bit is used to program the controlled automatic generation of slave device select signals during SPI transfers. This bit enables (if set, =1) or disables (if cleared, =0) the programmable slave select mode. The slave selection is subsequently made using the FLS bit.
15-18	FLS	**Flag Select.** These bits select which flag pins are asserted when multiple slaves are used (0=Disable, 1=Enable) as follows: Bit 15= FLAG0 Bit 16= FLAG1 Bit 17= FLAG2 Bit 18= FLAG3 **Note:** Only Flag[0] to Flag[3] can be used this way.
19	NSMLS	**Non-Seamless Operation.** This bit, if set (=1), indicates that after each word transfer there is a delay before the next word transfer starts. When cleared (=0), indicates no delay before the next word starts, a seamless operation.
20	DCPH0	**Deselect SPIDS in CPHASE = 0.** This bit deselects when high (=1) the slaves between successive word transfers in CPhase 0. The slave is selected in master mode using PSSE functionality. This bit has no effect in slave mode for the SPI port. This functionality is valid only when NSMLS =1 and CPHASE =0. This bit is cleared (=0) when not in use.

Table A-30. SPI Control Register Bit Descriptions (Cont'd)

Bit(s)	Name	Function
25	DMISO	**Disable MISO Pin**. This bit three-states, (if set, =1) the master in slave out (MISO) pin or (if cleared, =0) enables MISO. This is needed in an environment where master wishes to transmit to various slaves at one time (broadcast). Except for the slave from which it wishes to receive, all other slaves should have this bit set.
26	OPD	**Open Drain Output Enable**. This bit enables an open drain for data pins if set (=1) or remains normal if cleared (=0). If enabled, the MISO, MOSI and SPICLK is driven only for logic low and pulled up by a 50 kΩ resistance for a logic high.
27	RDMAEN	**Receive DMA Enable.** This bit enables (if set, =1) or disables (if cleared, =0) DMA transfers from the receive buffer. At SPI boot this bit is set to 1 to enable the booting process via the SPI port.
28	PACKEN	**Packing Enable.** This bit enables, if set (=1), 8- to 32-bit packing or disables the packing, if cleared (=0). If this bit is enabled, the receiver packs the received byte whereas the transmitter unpacks the data before sending it. Fore more information on the packing, see "SPI Word Packing" on page 11-24. **Note:** This bit should be 1 only for 8-bit data word length (WL = 00).
29	SGN	**Sign Extend.** This bit sign extends the word if set (=1) or does not extend the sign if cleared (=0).
30	SENDLW	**Send Last Data.** When SPITX is empty, setting this bit(=1) re transmits the last data. Clearing this bit (=0) sends zeros.
31	GM	**Get Data.** This bit fetches incoming data when set (=1) or discards incoming data when cleared (=0). The data that comes in overwrites the previous data in the SPIRX.

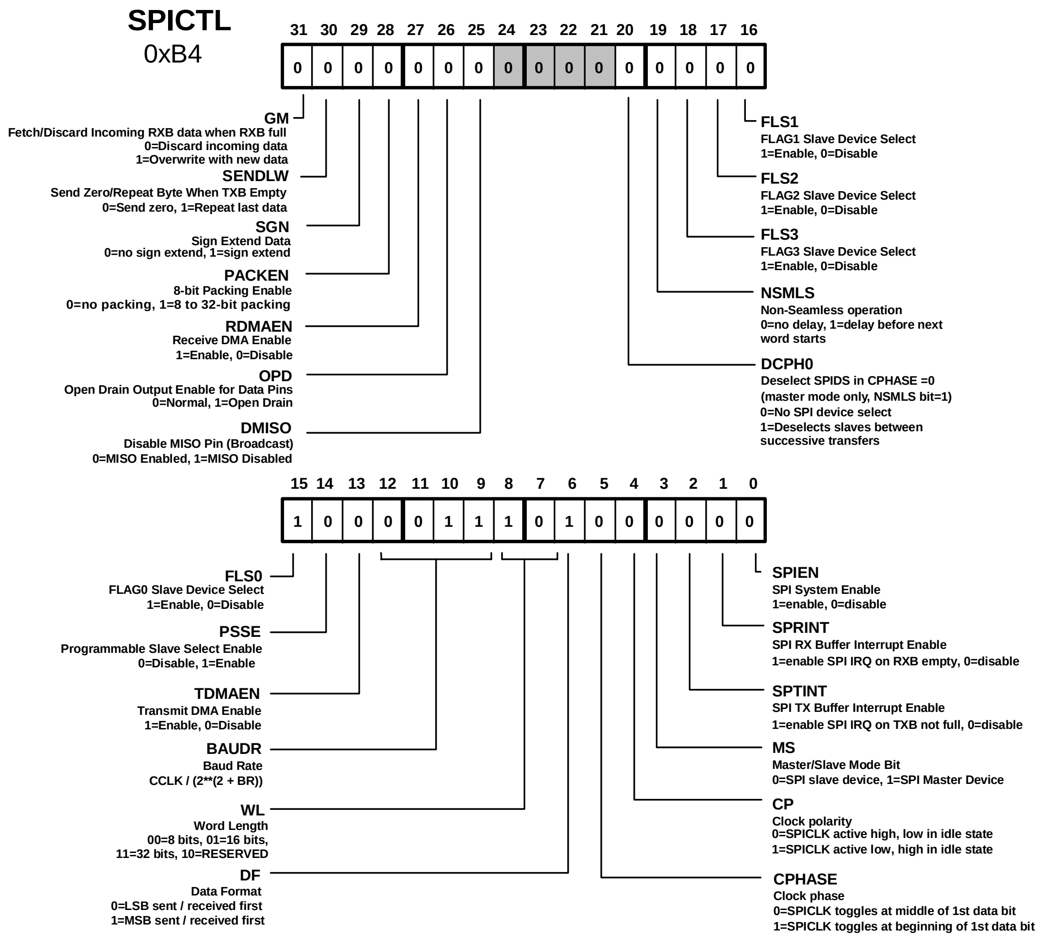

SPICTL
0xB4
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
GM
Fetch/Discard Incoming RXB data when RXB full
0=Discard incoming data
1=Overwrite with new data
SENDLW
Send Zero/Repeat Byte When TXB Empty
0=Send zero, 1=Repeat last data
SGN
Sign Extend Data
0=no sign extend, 1=sign extend
PACKEN
8-bit Packing Enable
0=no packing, 1=8 to 32-bit packing
RDMAEN
Receive DMA Enable
1=Enable, 0=Disable
OPD
Open Drain Output Enable for Data Pins
0=Normal, 1=Open Drain
DMISO
Disable MISO Pin (Broadcast)
0=MISO Enabled, 1=MISO Disabled
FLS1
FLAG1 Slave Device Select
1=Enable, 0=Disable
FLS2
FLAG2 Slave Device Select
1=Enable, 0=Disable
FLS3
FLAG3 Slave Device Select
1=Enable, 0=Disable
NSMLS
Non-Seamless operation
0=no delay, 1=delay before next word starts
DCPH0
Deselect SPIDS in CPHASE =0
(master mode only, NSMLS bit=1)
0=No SPI device select
1=Deselects slaves between successive transfers
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0
FLS0
FLAG0 Slave Device Select
1=Enable, 0=Disable
PSSE
Programmable Slave Select Enable
0=Disable, 1=Enable
TDMAEN
Transmit DMA Enable
1=Enable, 0=Disable
BAUDR
Baud Rate
CCLK / (2**(2 + BR))
WL
Word Length
00=8 bits, 01=16 bits,
11=32 bits, 10=RESERVED
DF
Data Format
0=LSB sent / received first
1=MSB sent / received first
SPIEN
SPI System Enable
1=enable, 0=disable
SPRINT
SPI RX Buffer Interrupt Enable
1=enable SPI IRQ on RXB empty, 0=disable
SPTINT
SPI TX Buffer Interrupt Enable
1=enable SPI IRQ on TXB not full, 0=disable
MS
Master/Slave Mode Bit
0=SPI slave device, 1=SPI Master Device
CP
Clock polarity
0=SPICLK active high, low in idle state
1=SPICLK active low, high in idle state
CPHASE
Clock phase
0=SPICLK toggles at middle of 1st data bit
1=SPICLK toggles at beginning of 1st data bit

Figure A-38. SPICTL Register

SPI Receive Buffer Register (SPIRX)

This register's address is 0xB7. The reset value for this register is undefined. This is a 32-bit read-only register accessible by the core or DMA controller. At the end of a data transfer, `SPIRX` is loaded with the data in the shift register. During a DMA receive operation, the data in `SPIRX` is automatically loaded into the internal memory. For core or interrupt driven transfer, you can also use the RXS status bits in the SPISTAT register to determine if the receive buffer is full. Reading from an empty SPIRX buffer causes a core hang if the buffer hang disable bit is cleared in the SYSCON register.

SPI Transmit Buffer Register (SPITX)

This register's address is 0xB6.The reset value for this register is undefined. This SPI transmit data register is a 32-bit register which is part of the IOP register set and can be accessed by the core or the DMA controller. Data is loaded into this register before being transmitted. Prior to the beginning of a data transfer, data in SPITX is automatically loaded into the transmit shift register. During a DMA transmit operation, the data in SPITX is automatically loaded from internal memory.

Register and Bit #Defines (def21161.h)

The following example definitions file is for the ADSP-21161 processor. For the most current definitions file, programs should use the version of this file included with the software development tools. The version of the file that appears here is provided as a guide only.

```
/*****************************************************************/
 *
 * def21161.h
 *
 * (c) Copyright 2001 Analog Devices, Inc. All rights reserved.
 *
 /*****************************************************************/

/*--------------------------------------------------------------

def21161.h - SYSTEM & IOP REGISTER BIT & ADDRESS DEFINITIONS FOR ADSP-21161
Last updated 5/14/01

This include file contains a list of macro "defines" to enable the programmer
to use symbolic names for the following ADSP-21161 facilities:
 - instruction condition codes
 - system register bit definitions
 - IOP register address memory map
 - *most* IOP control/status register bit definitions

Changes from def21160 include new I/O flags, SDRAM and SPI interfaces, changes to SPORT, Link Port, and
DMA.
```

```
Here are some example uses:

    bit set mode1 BR0|IRPTEN|ALUSAT;
    ustat1=BSO|HPM01|HMSWF;
    DM(SYSCON)=ustat1;

---------------------------------------------------------------------------- */
#ifndef __DEF21161_H_
#define __DEF21161_H_

/*--------------------------------------------------------------------------*/
/*                System Register bit definitions                           */
/*--------------------------------------------------------------------------*/
/* MODE1 and MMASK registers */
#define BR8     0x00000001 /* Bit  0: Bit-reverse for I8                    */
#define BR0     0x00000002 /* Bit  1: Bit-reverse for I0 (uses DMS0- only )   */
#define SRCU    0x00000004 /* Bit  2: Alt. register select for comp. units    */
#define SRD1H   0x00000008 /* Bit  3: DAG1 alt. register select (7-4)         */
#define SRD1L   0x00000010 /* Bit  4: DAG1 alt. register select (3-0)         */
#define SRD2H   0x00000020 /* Bit  5: DAG2 alt. register select (15-12)       */
#define SRD2L   0x00000040 /* Bit  6: DAG2 alt. register select (11-8)        */
#define SRRFH   0x00000080 /* Bit  7: Register file alt. select for R(15-8)   */
#define SRRFL   0x00000400 /* Bit 10: Register file alt. select for R(7-0)    */
#define NESTM   0x00000800 /* Bit 11: Interrupt nesting enable                */
#define IRPTEN  0x00001000 /* Bit 12: Global interrupt enable                 */
#define ALUSAT  0x00002000 /* Bit 13: Enable ALU fixed-pt. saturation         */
#define SSE     0x00004000 /* Bit 14: Enable short word sign extension        */
#define TRUNCATE 0x00008000 /* Bit 15: 1=fltg-pt. truncation 0=Rnd to nearest */
#define RND32   0x00010000 /* Bit 16: 1=32-bit fltg-pt.rounding 0=40-bit rnd  */
#define CSEL 0x00060000  /* Bit 17-18: CSelect: Bus Mastership */
#define PEYEN   0x00200000  /* Bit 21: Processing Element Y enable */
#define SIMD    0x00200000  /* Bit 21: Enable SIMD Mode */
#define BDCST9  0x00400000  /* Bit 22: Load Broadcast for I9 */
#define BDCST1  0x00800000  /* Bit 23: Load Broadcast for I1 *
#define CBUFEN  0x01000000  /* Bit 23: Circular Buffer Enable */

/* MODE2 register */
#define IRQ0E   0x00000001  /* Bit  0: IRQ0- 1=edge sens. 0=level sens. */
#define IRQ1E   0x00000002  /* Bit  1: IRQ1- 1=edge sens. 0=level sens. */
#define IRQ2E   0x00000004  /* Bit  2: IRQ2- 1=edge sens. 0=level sens. */
#define CADIS   0x00000010  /* Bit  4: Cache disable */
#define TIMEN   0x00000020  /* Bit  5: Timer enable */
#define BUSLK   0x00000040  /* Bit  6: External bus lock */
#define FLG0O   0x00008000  /* Bit 15: FLAG0 1=output 0=input */
#define FLG1O   0x00010000  /* Bit 16: FLAG1 1=output 0=input */
#define FLG2O   0x00020000  /* Bit 17: FLAG2 1=output 0=input */
#define FLG3O   0x00040000  /* Bit 18: FLAG3 1=output 0=input */
#define CAFRZ   0x00080000  /* Bit 19: Cache freeze */
#define IIRAE   0x00100000  /* Bit 20: Illegal IOP Register Access Enable */
#define U64MAE  0x00200000  /* Bit 21: Unaligned 64-bit Memory Access Enable */
/* bits 31-30, 27-25 are Processor ID[4:0], read only, value: 0b01001
   bits 29-28    are silicon revision[1:0], read only, value: 0

These bits (only) are routed to Mode2 Shadow register (IOP register 0x11)*/

/* FLAGS register */
#define FLG0    0x00000001  /* Bit 0: FLAG0 value */
#define FLG1    0x00000002  /* Bit 1: FLAG1 value */
#define FLG2    0x00000004  /* Bit 2: FLAG2 value */
#define FLG3    0x00000008  /* Bit 3: FLAG3 value */

/* ASTATx and ASTATy registers */

#ifdef SUPPORT_DEPRECATED_USAGE
/* Several of these (AV, AC, MV, SV, SZ) are assembler-reserved keywords,
   so this style is now deprecated.  If these are defined, the assembler-
   reserved keywords are still available in lowercase, e.g.,
           IF sz JUMP LABEL1.*/
```

```
#  define AZ    0x00000001 /* Bit  0: ALU result zero or fltg-pt. underflow */
#  define AV    0x00000002 /* Bit  1: ALU overflow                          */
#  define AN    0x00000004 /* Bit  2: ALU result negative                   */
#  define AC    0x00000008 /* Bit  3: ALU fixed-pt. carry                   */
#  define AS    0x00000010 /* Bit  4: ALU X input sign (ABS and MANT ops)   */
#  define AI    0x00000020 /* Bit  5: ALU fltg-pt. invalid operation        */
#  define MN    0x00000040 /* Bit  6: Multiplier result negative            */
#  define MV    0x00000080 /* Bit  7: Multiplier overflow                   */
#  define MU    0x00000100 /* Bit  8: Multiplier fltg-pt. underflow         */
#  define MI    0x00000200 /* Bit  9: Multiplier fltg-pt. invalid operation */
#  define AF    0x00000400 /* Bit 10: ALU fltg-pt. operation                */
#  define SV    0x00000800 /* Bit 11: Shifter overflow                      */
#  define SZ    0x00001000 /* Bit 12: Shifter result zero                   */
#  define SS    0x00002000 /* Bit 13: Shifter input sign                    */
#  define BTF   0x00040000 /* Bit 18: Bit test flag for system registers    */
#  define CACC0 0x01000000 /* Bit 24: Compare Accumulation Bit 0            */
#  define CACC1 0x02000000 /* Bit 25: Compare Accumulation Bit 1            */
#  define CACC2 0x04000000 /* Bit 26: Compare Accumulation Bit 2            */
#  define CACC3 0x08000000 /* Bit 27: Compare Accumulation Bit 3            */
#  define CACC4 0x10000000 /* Bit 28: Compare Accumulation Bit 4            */
#  define CACC5 0x20000000 /* Bit 29: Compare Accumulation Bit 5            */
#  define CACC6 0x40000000 /* Bit 30: Compare Accumulation Bit 6            */
#  define CACC7 0x80000000 /* Bit 31: Compare Accumulation Bit 7            */

#endif

#define ASTAT_AZ      0x00000001 /* Bit  0: ALU result zero or fltg-pt. u'flow*/
#define ASTAT_AV      0x00000002 /* Bit  1: ALU overflow                      */
#define ASTAT_AN      0x00000004 /* Bit  2: ALU result negative               */
#define ASTAT_AC      0x00000008 /* Bit  3: ALU fixed-pt. carry               */
#define ASTAT_AS      0x00000010 /* Bit  4: ALU X input sign(ABS and MANT ops)*/
#define ASTAT_AI      0x00000020 /* Bit  5: ALU fltg-pt. invalid operation    */
#define ASTAT_MN      0x00000040 /* Bit  6: Multiplier result negative        */
#define ASTAT_MV      0x00000080 /* Bit  7: Multiplier overflow               */
#define ASTAT_MU      0x00000100 /* Bit  8: Multiplier fltg-pt. underflow     */
#define ASTAT_MI      0x00000200 /* Bit  9: Multiplier fltg-pt. invalid op.   */
#define ASTAT_AF      0x00000400 /* Bit 10: ALU fltg-pt. operation            */
#define ASTAT_SV      0x00000800 /* Bit 11: Shifter overflow                  */
#define ASTAT_SZ      0x00001000 /* Bit 12: Shifter result zero               */
#define ASTAT_SS      0x00002000 /* Bit 13: Shifter input sign                */
#define ASTAT_BTF     0x00040000 /* Bit 18: Bit test flag for system registers*/
#define ASTAT_CACC0   0x01000000 /* Bit 24: Compare Accumulation Bit 0        */
#define ASTAT_CACC1   0x02000000 /* Bit 25: Compare Accumulation Bit 1        */
#define ASTAT_CACC2   0x04000000 /* Bit 26: Compare Accumulation Bit 2        */
#define ASTAT_CACC3   0x08000000 /* Bit 27: Compare Accumulation Bit 3        */
#define ASTAT_CACC4   0x10000000 /* Bit 28: Compare Accumulation Bit 4        */
#define ASTAT_CACC5   0x20000000 /* Bit 29: Compare Accumulation Bit 5        */
#define ASTAT_CACC6   0x40000000 /* Bit 30: Compare Accumulation Bit 6        */
#define ASTAT_CACC7   0x80000000 /* Bit 31: Compare Accumulation Bit 7        */

/* STKYx and STKYy registers */
/* bits 0 to 9 in both STKYx and STKYY, bits 17 to 26 in STKYx only */
#define AUS     0x00000001 /* Bit  0: ALU fltg-pt. underflow                  */
#define AVS     0x00000002 /* Bit  1: ALU fltg-pt. overflow                   */
#define AOS     0x00000004 /* Bit  2: ALU fixed-pt. overflow                  */
#define AIS     0x00000020 /* Bit  5: ALU fltg-pt. invalid operation          */
#define MOS     0x00000040 /* Bit  6: Multiplier fixed-pt. overflow           */
#define MVS     0x00000080 /* Bit  7: Multiplier fltg-pt. overflow            */
#define MUS     0x00000100 /* Bit  8: Multiplier fltg-pt. underflow           */
#define MIS     0x00000200 /* Bit  9: Multiplier fltg-pt. invalid operation   */
/* STKYx register *ONLY* */
#define CB7S    0x00020000 /* Bit 17: DAG1 circular buffer 7 overflow         */
#define CB15S   0x00040000 /* Bit 18: DAG2 circular buffer 15 overflow        */
#define IIRA    0x00080000 /* Bit 19: Illegal IOP Register Access             */
#define U64MA   0x00100000 /* Bit 20: Unaligned 64-bit Memory Access          */
#define PCFL    0x00200000 /* Bit 21: PC stack full */
#define PCEM    0x00400000 /* Bit 22: PC stack empty  */
#define SSOV    0x00800000 /* Bit 23: Status stack overflow (MODE1 and ASTAT) */
```

```
#define SSEM    0x01000000 /* Bit 24: Status stack empty */
#define LSOV    0x02000000 /* Bit 25: Loop stack overflow   */
#define LSEM    0x04000000 /* Bit 26: Loop stack empty   */

/* IRPTL and IMASK and IMASKP registers */
#define EMUI    0x00000001 /* Bit  0: Offset: 00: Emulator Interrupt */
#define RSTI    0x00000002 /* Bit  1: Offset: 04: Reset */
#define IICDI   0x00000004 /* Bit  2: Offset: 08: Illegal Input Condition Detected */
#define SOVFI   0x00000008 /* Bit  3: Offset: 0c: Stack overflow */
#define TMZHI   0x00000010 /* Bit  4: Offset: 10: Timer = 0 (high priority) */
#define VIRPTI  0x00000020 /* Bit  5: Offset: 14: Vector interrupt */
#define IRQ2I   0x00000040 /* Bit  6: Offset: 18: IRQ2- asserted */
#define IRQ1I   0x00000080 /* Bit  7: Offset: 1c: IRQ1- asserted */
#define IRQ0I   0x00000100 /* Bit  8: Offset: 20: IRQ0- asserted */
#define SP0I0x00000400 /* Bit 10: Offset: 28: SPORT0 DMA channel */
#define SP1I0x00000800 /* Bit 11: Offset: 2c: SPORT1 DMA channel */
#define SP2I0x00001000 /* Bit 12: Offset: 30: SPORT2 DMA channel */
#define SP3I0x00002000 /* Bit 13: Offset: 34: SPORT3 DMA channel */
#define LPISUMI0x00004000 /* Bit 14: Offset: na: LPort Interrupt Summary */
#define EP0I0x00008000 /* Bit 15: Offset: 50: External port channel 0 DMA */
#define EP1I0x00010000 /* Bit 16: Offset: 54: External port channel 1 DMA */
#define EP2I0x00020000 /* Bit 17: Offset: 58: External port channel 2 DMA */
#define EP3I0x00040000 /* Bit 18: Offset: 5c: External port channel 3 DMA */
#define LSRQI  0x00080000      /* Bit 19: Offset: 60: Link service request */
#define CB7I    0x00100000   /* Bit 20: Offset: 64: Circ. buffer 7 overflow */
#define CB15I   0x00200000   /* Bit 21: Offset: 68: Circ. buffer 15 overflow */
#define TMZLI   0x00400000    /* Bit 22: Offset: 6c: Timer = 0 (low priority) */
#define FIXI    0x00800000 /* Bit 23: Offset: 70: Fixed-pt. overflow          */
#define FLTOI   0x01000000 /* Bit 24: Offset: 74: fltg-pt. overflow           */
#define FLTUI   0x02000000 /* Bit 25: Offset: 78: fltg-pt. underflow          */
#define FLTII   0x04000000 /* Bit 26: Offset: 7c: fltg-pt. invalid            */
#define SFT0I   0x08000000 /* Bit 27: Offset: 80: user software int 0         */
#define SFT1I   0x10000000 /* Bit 28: Offset: 84: user software int 1         */
#define SFT2I   0x20000000 /* Bit 39: Offset: 88: user software int 2         */
#define SFT3I   0x40000000 /* Bit 30: Offset: 8c: user software int 3         */

/* LIRPTL register */
#define LP0I 0x00000001  /* Bit  0: Offset: 38: Link port channel 0 DMA */
#define LP1I 0x00000002  /* Bit  1: Offset: 3C: Link port channel 1 DMA */
#define SPIRI 0x00000004  /* Bit  2: Offset: 40: SPI Receive  DMA */
#define SPITI 0x00000008  /* Bit  3: Offset: 44: SPI Transmit DMA */
#define LP0MSK 0x00010000  /* Bit 16: Link port channel 0 Interrupt Mask */
#define LP1MSK 0x00020000  /* Bit 17: Link port channel 1 Interrupt Mask */
#define SPIRMSK 0x00040000  /* Bit 18:  SPI Receive Interrupt Mask */
#define SPITMSK 0x00080000  /* Bit 19:  SPI Transmit Interrupt Mask */
#define LP0MSKP 0x01000000  /* Bit 24: Link port channel 0 Interrupt Mask Pointer */
#define LP1MSKP 0x02000000  /* Bit 25: Link port channel 1 Interrupt Mask Pointer */
#define SPIRMSKP  0x04000000  /* Bit 26: SPI Receive Interrupt Mask Pointer */
#define SPITMSKP  0x08000000  /* Bit 27: SPI Transmit Interrupt Mask Pointer */

/* LSRQ register */
#define L0TM 0x00000010  /* Link Port 0 Transmit Mask      */
#define L0RM 0x00000020  /* Link Port 0 Receive Mask       */
#define L1TM 0x00000040  /* Link Port 1 Transmit Mask      */
#define L1RM 0x00000080  /* Link Port 1 Receive Mask       */
#define L0TRQ 0x00100000  /* Link Port 0 Transmit Request      */
#define L1TRQ 0x00200000  /* Link Port 1 Receive Request      */
#define L0RRQ 0x00400000  /* Link Port 0 Transmit Request      */
#define L1RRQ 0x00800000  /* Link Port 1 Receive Request      */


/*--------------------------------------------------------------------------------*/
/*                                                                                */
/*                 I/O Processor Register Address Memory Map                      */
/*                                                                                */
/*--------------------------------------------------------------------------------*/
#define SYSCON 0x00      /* System configuration register                     */
#define VIRPT  0x01      /* Vector interrupt register                         */
#define WAIT   0x02       /* External Port Wait register - renamed to EPCON  */
```

```
#define EPCON  0x02      /* External Port configuration register             */
#define SYSTAT 0x03      /* System status register                           */
/* the upper 32-bits of the 64-bit epbxs are only accessible as 64-bit reference*/
#define EPB0   0x04      /* External port DMA buffer 0                       */
#define EPB1   0x06      /* External port DMA buffer 1                       */
#define MSGR0  0x08      /* Message register 0                               */
#define MSGR1  0x09      /* Message register 1                               */
#define MSGR2  0x0a      /* Message register 2                               */
#define MSGR3  0x0b      /* Message register 3                               */
#define MSGR4  0x0c      /* Message register 4                               */
#define MSGR5  0x0d      /* Message register 5                               */
#define MSGR6  0x0e      /* Message register 6                               */
#define MSGR7  0x0f      /* Message register 7                               */

/* IOP shadow registers of the core control regs                                */
#define PC_SHDW    0x10     /* PC IOP shadow register (PC[23-0])                */
#define MODE2_SHDW 0x11     /* Mode2 IOP shadow register (MODE2[31-25])         */
#define EPB2   0x14         /* External port DMA buffer 2                       */
#define EPB3   0x16         /* External port DMA buffer 3                       */
#define BMAX   0x18         /* Bus time-out maximum                */
#define BCNT   0x19         /* Bus time-out counter                */
#define DMAC10 0x1c      /* EP DMA10 control register           */
#define DMAC11 0x1d      /* EP DMA11 control register           */
#define DMAC12 0x1e      /* EP DMA12 control register           */
#define DMAC13 0x1f      /* EP DMA13 control register           */
#define DMASTAT 0x37     /* DMA channel status register           */

/* SPI Registers  IOP Register Addresses*/
#define SPICTL    0xb4  /* Serial peripheral-compatible interface control register */
#define SPISTAT   0xb5  /* Serial periipheral-compatible interface status register */
#define SPIRX     0xb7  /* SPI receive data buffer */
#define SPITX     0xb6  /* SPI transmit data buffer */

/* IOFLAG Register Address */
#define IOFLAG   0x1b  /* Address of programmable I/O flags 4-11 */

/* IOP registers for SDRAM controller.     */
#define SDCTL    0xb8       /* SDRAM control reg.                 */
#define SDRDIV   0xb9       /* Refresh counter div reg.           */

/* Link Port Registers */
#define LBUF0   0xc0     /* Link buffer 0          */
#define LBUF1   0xc2     /* Link buffer 1          */
#define LCTL    0xcc     /* Link buffer control         */
#define LSRQ    0xd0     /* Link service request and mask registers    */

/* SPORT0 */
#define SPCTL0 0x1c0       /* SPORT0 serial port control register      */
#define TX0A    0x1c1      /* SPORT0 serial port control register      */
#define TX0B    0x1c2      /* SPORT0 transmit secondary B channel data buffer   */
#define RX0A    0x1c3      /* SPORT0 receive primary A channel data buffer   */
#define RX0B    0x1c4      /* SPORT0 receive secondary B channel data buffer   */
#define DIV0    0x1c5      /* SPORT0 divisor for transmit/receive SLCK0 and FS0 */
#define CNT0    0x1c6      /* SPORT0 count register */

/* SPORT2 */
#define SPCTL2 0x1d0     /* SPORT2 serial port control register      */
#define TX2A    0x1d1    /* SPORT2 serial port control register      */
#define TX2B    0x1d2    /* SPORT2 transmit secondary B channel data buffer   */
#define RX2A    0x1d3    /* SPORT2 receive primary A channel data buffer   */
#define RX2B    0x1d4    /* SPORT2 receive secondary B channel data buffer   */
#define DIV2    0x1d5    /* SPORT2 divisor for transmit/receive SLCK2 and FS2  */
#define CNT2    0x1d6    /* SPORT2 count register         */

/* SPORT1 */
#define SPCTL1 0x1e0     /* SPORT1 serial port control register      */
#define TX1A    0x1e1    /* SPORT1 serial port control register      */
#define TX1B    0x1e2    /* SPORT1 transmit secondary B channel data buffer   */
#define RX1A    0x1e3    /* SPORT1 receive primary A channel data buffer   */
```

```
#define RX1B    0x1e4    /* SPORT1 receive secondary B channel data buffer   */
#define DIV1    0x1e5    /* SPORT1 divisor for transmit/receive SLCK1 and FS1  */
#define CNT1    0x1e6    /* SPORT1 count register         */

/* SPORT3 */
#define SPCTL3 0x1f0  /* SPORT3 serial port control register      */
#define TX3A    0x1f1    /* SPORT3 serial port control register      */
#define TX3B    0x1f2    /* SPORT3 transmit secondary B channel data buffer   */
#define RX3A    0x1f3    /* SPORT3 receive primary A channel data buffer   */
#define RX3B    0x1f4    /* SPORT3 receive secondary B channel data buffer   */
#define DIV3  0x1f5   /* SPORT3 divisor for transmit/receive SLCK3 and FS3  */
#define CNT3  0x1f6  /* SPORT3 count register         */

/* SPORT0 - MCM Receive (Works in pair with SPORT2) */
#define MR0CS0 0x1c7  /* SPORT0 multichannel rx select, channels 31 - 0    */
#define MR0CCS0 0x1c8  /* SPORT0 multichannel rx compand select, channels 31 - 0  */
#define MR0CS1  0x1c9  /* SPORT0 multichannel rx select, channels 63 - 32    */
#define MR0CCS1  0x1ca  /* SPORT0 multichannel rx compand select, channels 63 - 32  */
#define MR0CS2  0x1cb  /* SPORT0 multichannel rx select, channels 95 - 64    */
#define MR0CCS2  0x1cc  /* SPORT0 multichannel rx compand select, channels 95 - 64  */
#define MR0CS3  0x1cd  /* SPORT0 multichannel rx select, channels 127 - 96   */
#define MR0CCS3  0x1ce /* SPORT0 multichannel rx compand select, channels 127 - 96 */

/* SPORT2 - MCM Transmit (Works in pair with SPORT0) */
#define MT2CS0      0x1d7  /* SPORT2 multichannel tx select, channels 31 - 0    */
#define MT2CCS0     0x1d8  /* SPORT2 multichannel tx compand select, channels 31 - 0  */
#define MT2CS1      0x1d9  /* SPORT2 multichannel tx select, channels 63 - 32    */
#define MT2CCS1     0x1da  /* SPORT2 multichannel tx compand select, channels 63 - 32  */
#define MT2CS2      0x1db  /* SPORT2 multichannel tx select, channels 95 - 64    */
#define MT2CCS2     0x1dc  /* SPORT2 multichannel tx compand select, channels 95 - 64  */
#define MT2CS3      0x1dd  /* SPORT2 multichannel tx select, channels 127 - 96   */
#define MT2CCS3     0x1de  /* SPORT2 multichannel tx compand select, channels 127 - 96 */

#define SP02MCTL    0x1df  /* SPORTs 0 & 2 Multichannel Control Register */

/* SPORT1 - MCM Receive (Works in pair with SPORT3) */
#define MR1CS0      0x1e7  /* SPORT1 multichannel rx select, channels 31 - 0    */
#define MR1CCS0     0x1e8  /* SPORT1 multichannel rx compand select, channels 31 - 0  */
#define MR1CS1      0x1e9  /* SPORT1 multichannel rx select, channels 63 - 32    */
#define MR1CCS1     0x1ea  /* SPORT1 multichannel rx compand select, channels 63 - 32  */
#define MR1CS2      0x1eb  /* SPORT1 multichannel rx select, channels 95 - 64    */
#define MR1CCS2     0x1ec  /* SPORT1 multichannel rx compand select, channels 95 - 64  */
#define MR1CS3      0x1ed  /* SPORT1 multichannel rx select, channels 127 - 96   */
#define MR1CCS3     0x1ee /* SPORT1 multichannel rx compand select, channels 127 - 96 */

/* SPORT3 - MCM Transmit (Works in pair with SPORT1) */
#define MT3CS0      0x1f7  /* SPORT3 multichannel tx select, channels 31 - 0    */
#define MT3CCS0     0x1f8  /* SPORT3 multichannel tx compand select, channels 31 - 0  */
#define MT3CS1      0x1f9  /* SPORT3 multichannel tx select, channels 63 - 32    */
#define MT3CCS1     0x1fa  /* SPORT3 multichannel tx compand select, channels 63 - 32  */
#define MT3CS2      0x1fb  /* SPORT3 multichannel tx select, channels 95 - 64    */
#define MT3CCS2     0x1fc  /* SPORT3 multichannel tx compand select, channels 95 - 64  */
#define MT3CS3      0x1fd  /* SPORT3 multichannel tx select, channels 127 - 96   */
#define MT3CCS3     0x1fe  /* SPORT3 multichannel tx compand select, channels 127 - 96 */

#define SP13MCTL    0x1ff /* SPORTs 1 & 3 Multichannel Control Register */


/*------ DMA Parameter Register Assignments - New Naming Conventions -------*/

/* DMA Channel 0 - Serial Port 0, A channel data */
#define II0A    0x60    /* Internal DMA0 memory address          */
#define IM0A    0x61    /* Internal DMA0 memory access modifier        */
#define C0A     0x62    /* Contains number of DMA0 transfers remaining   */
#define CP0A    0x63    /* Points to next DMA0 parameters          */
#define GP0A    0x64    /* DMA0 General purpose            */

/* DMA Channel 1 - Serial Port 0, B channel data */
#define II0B    0x80    /* Internal DMA1 memory address          */
```

```
#define IM0B    0x81    /* Internal DMA1 memory access modifier        */
#define C0B     0x82    /* Contains number of DMA1 transfers remaining   */
#define CP0B    0x83    /* Points to next DMA1 parameters         */
#define GP0B    0x84    /* DMA1 General purpose            */

/* DMA Channel 2 - Serial Port 1, A channel data */
#define II1A    0x68    /* Internal DMA2 memory address          */
#define IM1A    0x69    /* Internal DMA2 memory access modifier        */
#define C1A     0x6a    /* Contains number of DMA2 transfers remaining   */
#define CP1A    0x6b    /* Points to next DMA2 parameters         */
#define GP1A    0x6c    /* DMA2 General purpose             */

/* DMA Channel 3 - Serial Port 1, B channel data */
#define II1B    0x88    /* Internal DMA3 memory address          */
#define IM1B    0x89    /* Internal DMA3 memory access modifier        */
#define C1B     0x8a    /* Contains number of DMA3 transfers remaining   */
#define CP1B    0x8b    /* Points to next DMA3 parameters         */
#define GP1B    0x8c    /* DMA3 General purpose            */

/* DMA Channel 4 - Serial Port 2, A channel data        */
#define II2A    0x70    /* Internal DMA4 memory address          */
#define IM2A    0x71    /* Internal DMA4 memory access modifier        */
#define C2A     0x72    /* Contains number of DMA4 transfers remaining   */
#define CP2A    0x73    /* Points to next DMA4 parameters         */
#define GP2A    0x74    /* DMA4 General purpose            */

/* DMA Channel 5 - Serial Port 2, B channel data */
#define II2B    0x90    /* Internal DMA5 memory address          */
#define IM2B    0x91    /* Internal DMA5 memory access modifier        */
#define C2B     0x92    /* Contains number of DMA5 transfers remaining   */
#define CP2B    0x93    /* Points to next DMA5 parameters         */
#define GP2B    0x94    /* DMA5 General purpose            */

/* DMA Channel 6 - Serial Port 3, A channel data */
#define II3A    0x78    /* Internal DMA6 memory address          */
#define IM3A    0x79    /* Internal DMA6 memory access modifier        */
#define C3A     0x7a    /* Contains number of DMA6 transfers remaining   */
#define CP3A    0x7b    /* Points to next DMA6 parameters         */
#define GP3A    0x7c    /* DMA6 General purpose            */

/* DMA Channel 7 - Serial Port 3, B channel data */
#define II3B    0x98    /* Internal DMA7 memory address          */
#define IM3B    0x99    /* Internal DMA7 memory access modifier         */
#define C3B     0x9a    /* Contains number of DMA7 transfers remaining   */
#define CP3B    0x9b    /* Points to next DMA7 parameters         */
#define GP3B    0x9c    /* DMA7 General purpose            */

/* DMA Channel 8 - Link Buffer 0 (or SPI Receive) */
#define IILB0   0x30    /* Internal DMA8 memory address          */
#define IMLB0   0x31    /* Internal DMA8 memory access modifier        */
#define CLB0    0x32    /* Contains number of DMA8 transfers remaining   */
#define CPLB0   0x33    /* Points to next DMA8 parameters         */
#define GPLB0   0x34    /* DMA8 General purpose            */

/* DMA Channel 8 -  SPI Receive (or Link Buffer 0) - No DMA Chain Pointer reg */
#define IISRX   0x30    /* Internal DMA8 memory address          */
#define IMSRX   0x31    /* Internal DMA8 memory access modifier        */
#define CSRX    0x32    /* Contains number of DMA8 transfers remaining   */
#define GPSRX   0x34    /* DMA8 General purpose            */

/* DMA Channel 9 - Link Buffer 1 (or SPI Transmit) */
#define IILB1   0x38    /* Internal DMA9 memory address          */
#define IMLB1   0x39    /* Internal DMA9 memory access modifier        */
#define CLB1    0x3a    /* Contains number of DMA9 transfers remaining   */
#define CPLB1   0x3b    /* Points to next DMA9 parameters         */
#define GPLB1   0x3c    /* DMA9 General purpose            */

/* DMA Channel 9 -  SPI Transmit (or Link Buffer 1) - No DMA Chain Pointer reg */
#define IISTX    0x38    /* Internal DMA9 memory address          */
```

```
#define IMSTX    0x39    /* Internal DMA9 memory access modifier        */
#define CSTX     0x3a    /* Contains number of DMA9 transfers remainnig   */
#define GPSTX    0x3c    /* DMA9 General purpose */

/* DMA Channel 10 - External Port FIFO Buffer 0 */
#define IIEP0   0x40    /* Internal DMA10 memory address     */
#define IMEP0   0x41    /* Internal DMA10 memory access modifier        */
#define CEP0    0x42     /* Contains number of DMA10 transfers remaining  */
#define CPEP0   0x43    /* Points to next DMA10 parameters         */
#define GPEP0   0x44    /* DMA10 General purpose            */
#define EIEP0   0x45    /* External DMA10 address           */
#define EMEP0   0x46    /* External DMA10 address modifier         */
#define ECEP0   0x47    /* External DMA10 counter           */

/* DMA Channel 11 - External Port FIFO Buffer 1 */
#define IIEP1   0x48    /* Internal DMA11 memory address          */
#define IMEP1   0x49    /* Internal DMA11 memory access modifier        */
#define CEP1    0x4a    /* Contains number of DMA11 transfers remaining  */
#define CPEP1   0x4b    /* Points to next DMA11 parameters         */
#define GPEP1   0x4c    /* DMA11 General purpose            */
#define EIEP1   0x4d    /* External DMA11 address           */
#define EMEP1   0x4e    /* External DMA11 address modifier         */
#define ECEP1   0x4f    /* External DMA counter            */

/* DMA Channel 12 - External Port FIFO Buffer 2 */
#define IIEP2   0x50    /* Internal DMA12 memory address          */
#define IMEP2   0x51    /* Internal DMA12 memory access modifier        */
#define CEP2    0x52    /* Contains number of DMA12 transfers remaining  */
#define CPEP2   0x53    /* Points to next DMA12 parameters         */
#define GPEP2   0x54    /* DMA12 General purpose          */
#define EIEP2   0x55    /* External DMA12 address           */
#define EMEP2   0x56    /* External DMA12 address modifier         */
#define ECEP2   0x57    /* External DMA12 counter           */

/* DMA Channel 13 - External Port FIFO Buffer 3 */
#define IIEP3   0x58    /* Internal DMA13 memory address          */
#define IMEP3   0x59    /* Internal DMA13 memory access modifier        */
#define CEP3    0x5a    /* Contains number of DMA13 transfers remaining  */
#define CPEP3   0x5b    /* Points to next DMA13 parameters         */
#define GPEP3   0x5c    /* DMA13 General purpose            */
#define EIEP3   0x5d    /* External DMA13 address           */
#define EMEP3   0x5e    /* External DMA13 address modifier        */
#define ECEP3   0x5f    /* External DMA13 counter           */


/*---- DMA Parameter Register Assignments - Old Legacy ADSP-21160 Naming Conventions ---- */
/* NOTE: For backwards compatibility, we can retain the old DMA parameter
register names used in the ADSP-21160.  However, the naming conventions used for
DMA channels of the ADSP-21160 do not necessarily correspond to the actual DMA channel
priority assigment for the ADSP-21160

Ex) DMA Channel 4 IOP addresses on the ADSP-21160 are now DMA channel 8 on the ADSP-21161
    DMA Channel 5 IOP addresses on the ADSP-21160 are now DMA channel 9 on the ADSP-21161

To clear any confusion, we recommend using the new IOP naming conventions for the
DMA parameter registers as defined above */

#define II0    0x60    /* Internal DMA0 memory address           */
#define IM0    0x61    /* Internal DMA0 memory access modifier         */
#define C0     0x62    /* Contains number of DMA0 transfers remaining      */
#define CP0    0x63    /* Points to next DMA0 parameters          */
#define GP0    0x64    /* DMA0 General purpose              */

#define II1    0x68    /* Internal DMA1 memory address           */
#define IM1    0x69    /* Internal DMA1 memory access modifier         */
#define C1     0x6a    /* Contains number of DMA1 transfers remaining      */
#define CP1    0x6b    /* Points to next DMA1 parameters          */
#define GP1    0x6c    /* DMA1 General purpose              */
```

```
#define II2    0x70     /* Internal DMA2 memory address           */
#define IM2    0x71     /* Internal DMA2 memory access modifier         */
#define C2     0x72     /* Contains number of DMA2 transfers remaining      */
#define CP2    0x73     /* Points to next DMA2 parameters          */
#define GP2    0x74     /* DMA2 General purpose              */

#define II3    0x78    /* Internal DMA3 memory address           */
#define IM3    0x79    /* Internal DMA3 memory access modifier         */
#define C3     0x7a    /* Contains number of DMA3 transfers remaining      */
#define CP3    0x7b    /* Points to next DMA3 parameters          */
#define GP3    0x7c    /* DMA3 General purpose              */

#define II6    0x80    /* Internal DMA6 memory address           */
#define IM6    0x81    /* Internal DMA6 memory access modifier         */
#define C6     0x82    /* Contains number of DMA6 transfers remaining      */
#define CP6    0x83    /* Points to next DMA6 parameters          */
#define GP6    0x84    /* DMA6 General purpose              */

#define II7    0x88    /* Internal DMA7 memory address           */
#define IM7    0x89    /* Internal DMA7 memory access modifier         */
#define C7     0x8a    /* Contains number of DMA7 transfers remaining      */
#define CP7    0x8b    /* Points to next DMA7 parameters */
#define GP7    0x8c    /* DMA7 General purpose              */

#define II8    0x90    /* Internal DMA8 memory address           */
#define IM8    0x91    /* Internal DMA8 memory access modifier         */
#define C8     0x92    /* Contains number of DMA8 transfers remaining      */
#define CP8    0x93    /* Points to next DMA8 parameters          */
#define GP8    0x94    /* DMA8 General Purpose        */

#define II9    0x98    /* Internal DMA9 memory address           */
#define IM9    0x99    /* Internal DMA9 memory access modifier         */
#define C9     0x9a    /* Contains number of DMA9 transfers remaining      */
#define CP9    0x9b    /* Points to next DMA9 parameters          */
#define GP9    0x9c    /* DMA9 General purpose              */

#define II4    0x30    /* Internal DMA4 memory address           */
#define IM4    0x31    /* Internal DMA4 memory access modifier         */
#define C4     0x32    /* Contains number of DMA4 transfers remaining      */
#define CP4    0x33    /* Points to next DMA4 parameters          */
#define GP4    0x34    /* DMA4 General purpose              */

#define II5    0x38    /* Internal DMA5 memory address           */
#define IM5    0x39    /* Internal DMA5 memory access modifier         */
#define C5     0x3a    /* Contains number of DMA5 transfers remaining      */
#define CP5    0x3b    /* Points to next DMA5 parameters          */
#define GP5    0x3c    /* DMA5 General purpose              */

#define II10   0x40    /* Internal DMA10 memory address            */
#define IM10   0x41    /* Internal DMA10 memory access modifier         */
#define C10    0x42    /* Contains number of DMA10 transfers remaining     */
#define CP10   0x43    /* Points to next DMA10 parameters          */
#define GP10   0x44    /* DMA10 General purpose              */
#define EI10   0x45    /* External DMA10 address             */
#define EM10   0x46    /* External DMA10 address modifier          */
#define EC10   0x47    /* External DMA10 counter             */

#define II11   0x48    /* Internal DMA11 memory address            */
#define IM11   0x49    /* Internal DMA11 memory access modifier         */
#define C11    0x4a    /* Contains number of DMA11 transfers remaining     */
#define CP11   0x4b    /* Points to next DMA11 parameters          */
#define GP11   0x4c    /* DMA11 General purpose              */
#define EI11   0x4d    /* External DMA11 address             */
#define EM11   0x4e    /* External DMA11 address modifier          */
#define EC11   0x4f    /* External DMA counter               */

#define II12   0x50    /* Internal DMA12 memory address          */
#define IM12   0x51    /* Internal DMA12 memory access modifier         */
#define C12    0x52    /* Contains number of DMA12 transfers remaining     */
```

```
#define CP12   0x53    /* Points to next DMA12 parameters          */
#define GP12   0x54    /* DMA12 General purpose             */
#define EI12   0x55    /* External DMA12 address            */
#define EM12   0x56    /* External DMA12 address modifier          */
#define EC12   0x57    /* External DMA12 counter            */

#define II13   0x58    /* Internal DMA13 memory address           */
#define IM13   0x59    /* Internal DMA13 memory access modifier         */
#define C13    0x5a    /* Contains number of DMA13 transfers remaining     */
#define CP13   0x5b    /* Points to next DMA13 parameters          */
#define GP13   0x5c    /* DMA13 General purpose             */
#define EI13   0x5d    /* External DMA13 address            */
#define EM13   0x5e    /* External DMA13 address modifier          */
#define EC13   0x5f    /* External DMA13 counter            */


/* Emulation/Breakpoint Registers (remapped from UREG space) */
/*  NOTES:
       - These registers are ONLY accessible by the core
       - It is *highly* recommended that these facilities be accessed only
         through the ADI emulator routines
*/
/* Core Emulation HWBD Registers */
#define PSA1S  0xa0        /* Instruction address start #1                  */
#define PSA1E  0xa1        /* Instruction address end   #1                  */
#define PSA2S  0xa2        /* Instruction address start #2                  */
#define PSA2E  0xa3        /* Instruction address end   #2                  */
#define PSA3S  0xa4        /* Instruction address start #3                  */
#define PSA3E  0xa5        /* Instruction address end   #3                  */
#define PSA4S  0xa6        /* Instruction address start #4                  */
#define PSA4E  0xa7        /* Instruction address end   #4                  */
#define PMDAS  0xa8        /* Program Data address start                    */
#define PMDAE  0xa9        /* Program Data address end                      */
#define DMA1S  0xaa        /* Data address start #1                         */
#define DMA1E  0xab        /* Data address end   #1                         */
#define DMA2S  0xac        /* Data address start #2                         */
#define DMA2E  0xad        /* Data address end   #2                         */
#define EMUN   0xae        /* hwbp hit-count register                       */

/* IOP Emulation HWBP Bounds Registers */
#define IOAS   0xb0        /* IOA Upper Bounds Register                     */
#define IOAE   0xb1        /* IOA Lower Bounds Register                     */
#define EPAS   0xb2        /* EPA Upper Bounds Register                     */
#define EPAE   0xb3        /* EPA Lower Bounds Register                     */


/*----------------------------------------------------------------------------*/
/*                                                                            */
/*                   IOP Control/Status Register Bit Definitions              */
/*                                                                            */
/*----------------------------------------------------------------------------*/

/* SYSCON Register */
#define SRST   0x00000001  /* Soft Reset */
#define BSO    0x00000002  /* Boot Select Override*/
#define IIVT   0x00000004  /* Internal Interrupt Vector Table*/
#define IWT    0x00000008  /* Instruction word transfer (0 = data, 1 = inst)  */
#define HBW32  0x00000000  /* Host bus width: 32  */
#define HBW16  0x00000010  /* Host bus width: 16  */
#define HBW8   0x00000020  /* Host bus width: 8     */
#define HMSWF  0x00000080  /* Host packing order (0 = LSW first, 1 = MSW)     */
#define HPFLSH 0x00000100  /* Host pack flush*/
#define IMDWOX 0x00000200  /* Internal memory block 0, extended data (40 bit) */
#define IMDW1X 0x00000400  /* Internal memory block 1, extended data (40 bit) */
#define ADREDY 0x00000800  /* Active Drive Ready */
#define BHD    0x00010000  /* Buffer Hand Disable*/
#define EBPR00 0x00000000  /* External bus priority: Even*/
#define EBPR01 0x00020000  /* External bus priority: Core has priority      */
#define EBPR10 0x00040000  /* External bus priority: IO has priority      */
```

```
#define DCPR   0x00080000  /* Select rotating access priority on DMA10 - DMA13*/
#define LDCPR  0x00100000  /* Select rotating access priority on DMA8 - DMA9  */
#define PRROT  0x00200000  /* Select rotating prio between LPort and EPort    */
#define COD    0x00400000  /* Clock Out Disable         */
#define IPACK0 0x40000000  /* External instruction execution packing mode bit 0 */
#define IPACK1 0x80000000  /* External instruction execution packing mode bit 1 */


/* SYSTAT Register */
#define HSTM   0x00000001  /* Host is the Bus Master*/
#define BSYN   0x00000002  /* Bus arbitration logic is synchronized     */
#define CRBM   0x00000070  /* Current ADSP211xx Bus Master*/
#define IDC    0x00000700  /* ADSP211xx ID Code*/
#define VIPD   0x00002000  /* Vector interrupt pending (1 = pending) */
#define CRAT   0x00070000  /* CLK_CFG(3-0), Core:CLKIN clock ratio  */
#define SSWPD  0x00100000  /* Sync slave write pending... SSWPD bit added for 21161 */
#define SWPD   0x00200000  /* Any (sync + Async) slave write pending     */
#define HPS    0x01c00000  /* Host pack status... HPS modified for 21161 */


/* MODE2_SHDW Register - IOP register adrees 0x11 */
/* bits 31-30, 27-25 are Processor ID[4:0], read only, value: 01010
   bits 29-28    are silicon revision[1:0], read only, value: 01
   These former MODE2 register bitfields (only) are now routed to the MODE2
   Shadow register (IOP register 0x11).  Bits 25-31 now reserved in MODE2. */
#define PID20       0x0E000000 /* PID[2:0] Processor Identification (read-only)*/
#define SIREV       0x30000000 /* Silicon Revision (read-only) */
#define PID43       0xC0000000 /* PID[4:3] Processor Identification (read-only) */


/* WAIT Register */
/* generic WAIT bitfields */
#define EB0AM       0x00000003     /* External Bank 0 Access Mode                  */
#define EB0WS       0x0000001C     /* External Bank 0 Waitstate Configuration      */
#define EB1AM       0x00000060     /* External Bank 1 Access Mode                  */
#define EB1WS       0x00000380     /* External Bank 1 Waitstate Configuration      */
#define EB2AM       0x00000C00     /* External Bank 1 Access Mode                  */
#define EB2WS       0x00007000     /* External Bank 2 Waitstate Configuration      */
#define EB3AM       0x00018000     /* External Bank 1 Access Mode                  */
#define EB3WS       0x000E0000     /* External Bank 3 Waitstate Configuration      */
#define RBAM        0x00300000     /* ROM Boot Access Mode                         */
#define RBWS        0x01C00000     /* ROM Boot Waitstate Configuration             */
#define HIDMA       0x80000000     /* Single idle cycle for DMA handshake */
/* specific wait access mode settings */
#define EB0A0       0x00000000     /* Ext Bank 0 Async, internal AND external ACK      */
#define EB0S1       0x00000001     /* Ext Bank 0 Sync, 2-cycle reads, 1-cycle writes */
#define EB0S2       0x00000002     /* Ext Bank 0 Sync, 2-cycle reads, 2-cycle writes */
#define EB1A0       0x00000000     /* Ext Bank 1 Async, internal AND external ACK      */
#define EB1S1       0x00000020     /* Ext Bank 1 Sync, 2-cycle reads, 1-cycle writes */
#define EB1S2       0x00000040     /* Ext Bank 1 Sync, 2-cycle reads, 2-cycle writes */
#define EB2A0       0x00000000     /* Ext Bank 2 Async, internal AND external ACK      */
#define EB2S1       0x00000400     /* Ext Bank 2 Sync, 2-cycle reads, 1-cycle writes */
#define EB2S2       0x00000800     /* Ext Bank 2 Sync, 2-cycle reads, 2-cycle writes */
#define EB3A0       0x00000000     /* Ext Bank 3 Async, internal AND external ACK      */
#define EB3S1       0x00008000     /* Ext Bank 3 Sync, 2-cycle reads, 1-cycle writes */
#define EB3S2       0x00010000     /* Ext Bank 3 Sync, 2-cycle reads, 2-cycle writes */
#define RBWA0       0x00000000     /* ROM boot: Async, internal AND external ACK       */
#define RBWS1       0x00100000     /* ROM boot: Sync, 2-cycle reads, 1-cycle writes    */
#define RBWS2       0x00200000     /* ROM boot: Sync, 2-cycle reads, 2-cycle writes    */
/* individual waitstate combinations */
#define EB0WS0  0x00000000  /* External Bank 0: 0 waitstates, no hold cycle */
#define EB0WS1  0x00000004  /* External Bank 0: 1 waitstates, no hold cycle */
#define EB0WS2  0x00000008  /* External Bank 0: 2 waitstates, hold cycle    */
#define EB0WS3  0x0000000C  /* External Bank 0: 3 waitstates, hold cycle    */
#define EB0WS4  0x00000010  /* External Bank 0: 4 waitstates, hold cycle    */
#define EB0WS5  0x00000014  /* External Bank 0: 5 waitstates, hold cycle    */
#define EB0WS6  0x00000018  /* External Bank 0: 6 waitstates, hold cycle    */
#define EB0WS7  0x0000001C  /* External Bank 0: 7 waitstates, hold cycle    */
#define EB1WS0  0x00000000  /* External Bank 1: 0 waitstates, no hold cycle */
```

```
#define EB1WS1  0x00000080  /* External Bank 1: 1 waitstates, no hold cycle */
#define EB1WS2  0x00000100  /* External Bank 1: 2 waitstates, hold cycle    */
#define EB1WS3  0x00000180  /* External Bank 1: 3 waitstates, hold cycle    */
#define EB1WS4  0x00000200  /* External Bank 1: 4 waitstates, hold cycle    */
#define EB1WS5  0x00000280  /* External Bank 1: 5 waitstates, hold cycle    */
#define EB1WS6  0x00000300  /* External Bank 1: 6 waitstates, hold cycle    */
#define EB1WS7  0x00000380  /* External Bank 1: 7 waitstates, hold cycle    */
#define EB2WS0  0x00000000  /* External Bank 2: 0 waitstates, no hold cycle */
#define EB2WS1  0x00001000  /* External Bank 2: 1 waitstates, no hold cycle */
#define EB2WS2  0x00002000  /* External Bank 2: 2 waitstates, hold cycle    */
#define EB2WS3  0x00003000  /* External Bank 2: 3 waitstates, hold cycle    */
#define EB2WS4  0x00004000  /* External Bank 2: 4 waitstates, hold cycle    */
#define EB2WS5  0x00005000  /* External Bank 2: 5 waitstates, hold cycle    */
#define EB2WS6  0x00006000  /* External Bank 2: 6 waitstates, hold cycle    */
#define EB2WS7  0x00007000  /* External Bank 2: 7 waitstates, hold cycle    */
#define EB3WS0  0x00000000  /* External Bank 3: 0 waitstates, no hold cycle */
#define EB3WS1  0x00020000  /* External Bank 3: 1 waitstates, no hold cycle */
#define EB3WS2  0x00040000  /* External Bank 3: 2 waitstates, hold cycle    */
#define EB3WS3  0x00060000  /* External Bank 3: 3 waitstates, hold cycle    */
#define EB3WS4  0x00080000  /* External Bank 3: 4 waitstates, hold cycle    */
#define EB3WS5  0x000A0000  /* External Bank 3: 5 waitstates, hold cycle    */
#define EB3WS6  0x000C0000  /* External Bank 3: 6 waitstates, hold cycle    */
#define EB3WS7  0x000E0000  /* External Bank 3: 7 waitstates, hold cycle    */
#define RBWST0 0x00000000  /* ROM boot wait state 0, no hold cycle*/
#define RBWST1 0x00400000  /* ROM boot wait state 1, no hold cycle*/
#define RBWST2 0x00800000  /* ROM boot wait state 2, hold cycle*/
#define RBWST3 0x00C00000  /* ROM boot wait state 3, hold cycle*/
#define RBWST4 0x01000000  /* ROM boot wait state 4, hold cycle*/
#define RBWST5 0x01400000  /* ROM boot wait state 5, hold cycle*/
#define RBWST6 0x01800000  /* ROM boot wait state 6, hold cycle*/
#define RBWST7 0x01C00000     /* ROM boot wait state 7, hold cycle*/


/* DMAC10, DMAC11, DMAC12, DMAC13 Register Bitfield Definitions */
#define        DEN       0x00000001/* External Port DMA Enable */
#define        CHEN      0x00000002/* External Port DMA Chaining Enable */
#define        TRAN      0x00000004/* External Port EPBx Transmit/Receive Select */
#define        DTYPE     0x00000020/* EPBx FIFO Buffer Data Type Select */
#define     PMODE1 0x00000040/* EPBx FIFO Pack Modes.16-bit ext to 32/64-bit int packing */
#define        PMODE2    0x00000080/* 16-bit external to 48-bit internal packing */
#define        PMODE3    0x000000C0/* 32-bit external to 48-bit internal packing */
#define        PMODE4    0x00000100/* No Pack Mode-32-bit external to 32/64-bit internal packing */
#define        PMODE5    0x00000140/* 8-bit external to 48-bit internal packing */
#define        PMODE6    0x00000180/* 8-bit external to 32/64-bit internal packing */
#define        MSWF      0x00000200/* Most Significant Word First During Packing */
#define        MASTER    0x00000400/* EPBx DMA Master Mode Enable */
#define        HSHAKE    0x00000800/* EPBx DMA Handshake Mode Enable */
#define        INTIO     0x00001000/* Single Word Interrupts for EPBx FIFO Buffers */
#define        EXTERN    0x00002000/* External Handshake Mode Enable */
#define        FLSH      0x00004000/* Flush EPBx FIFO Buffers and Status */
#define        PRIO      0x00008000/* External Port Bus Priority Access */
#define        FS        0x00030000/* External Port FIFO Buffer Status (read-only) */
#define        INT32     0x00040000/* Internal Memory 32-bit Transfer Select */
#define        MAXBL0  0x00000000/* Maximum Burst Length Select Disabled */
#define        MAXBL1  0x00080000/* Bit 19 set; Maximum Burst Length Limit of 4 Enabled*/
#define        PS      0x00E00000/* Ext. Port EPBx FIFO Buffer Pack Status (read-only)*/

/* DMASTAT Register (read-only) */
#define DMA0ST   0x00000001   /* DMA channel 0 (RX0A/TX0A) Active Status */
#define DMA2ST  0x00000002   /* DMA channel 2 (RX1A/TX1A) Active Status */
#define DMA4ST   0x00000004   /* DMA channel 4 (RX2A/TX2A) Active Status */
#define DMA6ST   0x00000008   /* DMA channel 6 (RX3A/TX3A) Active Status */
#define DMA8ST   0x00000010   /* DMA channel 8 (LBUF0) Active Status */
#define DMA9ST   0x00000020   /* DMA channel 9 (LBUF1) Active Status */
#define DMA1ST   0x00000040   /* DMA channel 1 (RX0B/TX0B) Active Status */
#define DMA3ST   0x00000080   /* DMA channel 3 (RX1B/TX1B) Active Status */
#define DMA5ST   0x00000100   /* DMA channel 5 (RX2B/TX2B) Active Status */
#define DMA7ST   0x00000200   /* DMA channel 7 (RX3B/TX3B) Active Status */
#define DMA10ST   0x00000400/* DMA channel 10 (EPB0) Active Status */
```

```
#define DMA11ST  0x00000800  /* DMA channel 11 (EPB1) Active Status */
#define DMA12ST  0x00001000  /* DMA channel 12 (EPB2) Active Status */
#define DMA13ST  0x00002000  /* DMA channel 13 (EPB3) Active Status */
#define DMA0CHST  0x00010000/* DMA channel 0 (RX0A/TX0A) Chaining Status */
#define DMA2CHST  0x00020000/* DMA channel 2 (RX1A/TX1A) Chaining Status */
#define DMA4CHST  0x00040000/* DMA channel 4 (RX2A/TX2A) Chaining Status */
#define DMA6CHST  0x00080000/* DMA channel 6 (RX3A/TX3A) Chaining Status */
#define DMA8CHST  0x00100000/* DMA channel 8 (LBUF0) Chaining Status */
#define DMA9CHST  0x00200000/* DMA channel 9 (LBUF1) Chaining Status */
#define DMA1CHST  0x00400000/* DMA channel 1 (RX0B/TX0B) Chaining Status */
#define DMA3CHST  0x00800000/* DMA channel 3 (RX1B/TX1B) Chaining Status */
#define DMA5CHST  0x01000000/* DMA channel 5 (RX2B/TX2B) Chaining Status */
#define DMA7CHST  0x02000000/* DMA channel 7 (RX3B/TX3B) Chaining Status */
#define DMA10CHST 0x04000000/* DMA channel 10 (EPB0) Chaining Status */
#define DMA11CHST 0x08000000/* DMA channel 11 (EPB1) Chaining Status */
#define DMA12CHST 0x10000000/* DMA channel 12 (EPB2) Chaining Status */
#define DMA13CHST 0x20000000/* DMA channel 13 (EPB3) Chaining Status */


/* SDCTL - SDRAM Control Register bit definitions */
#define SDCL1       0x00000001/* SDCL[1:0] - CAS Latency field */
#define SDCL2       0x00000002/*  (delay between RD cmd and data at o/p pins) */
#define SDCL3       0x00000003/* configurable between 1 and 3 SDCLK cycles */

#define DSDCTL      0x00000004 /* disable SDCLK0, /RAS, /CAS & SDCKE pins   */
#define DSDCK1      0x00000008/* disable SDCLK1 pin   */

#define       SDTRAS0     0x00000000/* SDTRAS[3:0] - tRAS spec (active command delay)*/
#define       SDTRAS1     0x00000010  /*  (required delay between a Bank Activate     */
#define       SDTRAS2     0x00000020/*   command to a Precharge command)            */
#define       SDTRAS3     0x00000030  /* configurable between 0 to 15 SDCLK cycles */
#define       SDTRAS4     0x00000040
#define       SDTRAS5     0x00000050
#define       SDTRAS6     0x00000060
#define       SDTRAS7     0x00000070
#define       SDTRAS8     0x00000080
#define       SDTRAS9     0x00000090
#define       SDTRAS10    0x000000a0
#define       SDTRAS11    0x000000b0
#define       SDTRAS12    0x000000c0
#define       SDTRAS13    0x000000d0
#define       SDTRAS14    0x000000e0
#define       SDTRAS15    0x000000f0

#define       SDTRP0      0x00000000/* SDTRP[2:0] - tRP spec (precharge delay) */
#define       SDTRP1      0x00000100/*  (required delay between a precharge command */
#define       SDTRP2      0x00000200/*   to a Bank Activate command)                */
#define       SDTRP3      0x00000300  /* configurable between 1 to 7 cycles */
#define       SDTRP4      0x00000400
#define       SDTRP5      0x00000500
#define       SDTRP6      0x00000600
#define       SDTRP7      0x00000700

#define SDPM        0x00000800/* SDRAM power-up mode bit    */
#define SDPGS256    0x00000000/* SDRAM Page Size - 256 words */
#define SDPGS512    0x00001000/* SDRAM Page Size - 512 words */
#define SDPGS1024   0x00002000/* SDRAM Page Size - 1024 words */
#define SDPGS2048   0x00003000/* SDRAM Page Size - 2048 words */
#define SDPSS       0x00004000/* SDRAM power-up sequence start command    */
#define SDSRF       0x00008000/* Self refresh command     */
#define SDEM0       0x00010000/* Memory Bank 0 SDRAM Enable*/
#define SDEM1       0x00020000/* Memory Bank 1 SDRAM Enable*/
#define SDEM2       0x00040000/* Memory Bank 2 SDRAM Enable*/
#define SDEM3       0x00080000/* Memory Bank 3 SDRAM Enable*/
#define SDBN2       0x00000000/* SDRAM contains 2 memory banks    */
#define SDBN4       0x00100000/* SDRAM contains 4 memory banks*/
#define SDCKRx1     0x00200000/* 1:1 (full) SDCLK-to-CCLK (core-clock) ratio */
#define SDCKR_DIV2  0x00000000/* 1:2 (one-half) SDCLK-to-CCLK ratio */
#define SDBUF       0x00800000/* Pipeline (reg. buf) option*/
```

```
#define SDTRCD0     0x00000000/* SDTRCD[2:0] - tRCD spec. (RAS-to-CAS delay)*/
#define SDTRCD1     0x01000000/*  (required delay between a Bank Activate cmd */
#define SDTRCD2     0x02000000/*   and the start of the first RD or WR)       */
#define SDTRCD3     0x03000000/* configurable between 1 to 7 SDCLK cycles*/
#define SDTRCD4     0x04000000
#define SDTRCD5     0x05000000
#define SDTRCD6     0x06000000
#define SDTRCD7     0x07000000


/* IOFLAG - programmable I/O status macro definitions */
#define FLG4   0x00000001     /* FLAG4 value (Low = '0', High = '1') */
#define FLG5   0x00000002     /* FLAG5 value (Low = '0', High = '1') */
#define FLG6   0x00000004     /* FLAG6 value (Low = '0', High = '1') */
#define FLG7   0x00000008     /* FLAG7 value (Low = '0', High = '1') */
#define FLG8   0x00000010     /* FLAG8 value (Low = '0', High = '1') */
#define FLG9   0x00000020     /* FLAG9 value (Low = '0', High = '1') */
#define FLG10  0x00000040     /* FLAG10 value (Low = '0', High = '1') */
#define FLG11  0x00000080     /* FLAG11 value (Low = '0', High = '1') */
/* IOFLAG - programmable I/O control macro definitions */
#define FLG4O  0x00000100     /* FLAG4 control ('0' = flag input, '1' = flag output) */
#define FLG5O  0x00000200     /* FLAG5 control ('0' = flag input, '1' = flag output) */
#define FLG6O  0x00000400     /* FLAG6 control ('0' = flag input, '1' = flag output) */
#define FLG7O  0x00000800     /* FLAG7 control ('0' = flag input, '1' = flag output) */
#define FLG8O  0x00001000     /* FLAG8 control ('0' = flag input, '1' = flag output) */
#define FLG9O  0x00002000     /* FLAG9 control ('0' = flag input, '1' = flag output) */
#define FLG10O 0x00004000     /* FLAG10 control ('0' = flag input, '1' = flag output) */
#define FLG11O 0x00008000     /* FLAG11 control ('0' = flag input, '1' = flag output) */


/*SPICTL register */
#define SPIEN  0x00000001     /* SPI system enable */
#define SPRINT 0x00000002     /* SPIRX buffer interrupt enable */
#define SPTINT 0x00000004     /* SPITX buffer interrupt enable */
#define MS     0x00000008     /* Master/Slave Mode bit */
#define CP     0x00000010     /* SPICLK Polarity */
#define CPHASE 0x00000020     /* SPICLK Phase */
#define DF     0x00000040     /* Data Format */
#define WL8    0x00000000     /* SPI Word Length = 8 */
#define WL16   0x00000080     /* SPI Word Length = 16 */
#define WL32   0x00000180     /* SPI Word Length = 32 */
#define BAUDR1 0x00000200     /* BAUDRATE = CCLK / 2**(2 + 1) = CCLK/8 */
#define BAUDR2 0x00000400     /* BAUDRATE = CCLK / 2**(2 + 2) = CCLK/16 */
#define BAUDR3 0x00000600     /* BAUDRATE = CCLK / 2**(2 + 3) = CCLK/32 */
#define BAUDR4 0x00000800     /* BAUDRATE = CCLK / 2**(2 + 4) = CCLK/64 */
#define BAUDR5 0x00000A00     /* BAUDRATE = CCLK / 2**(2 + 5) = CCLK/128 */
#define BAUDR6 0x00000C00     /* BAUDRATE = CCLK / 2**(2 + 6) = CCLK/512 */
#define BAUDR7 0x00000E00     /* BAUDRATE = CCLK / 2**(2 + 7) = CCLK/1024 */
#define BAUDR8 0x00001000     /* BAUDRATE = CCLK / 2**(2 + 8) = CCLK/2048 */
#define BAUDR9 0x00001200     /* BAUDRATE = CCLK / 2**(2 + 9) = CCLK/4096 */
#define BAUDR100x00001400     /* BAUDRATE = CCLK / 2**(2 + 10) = CCLK/8192 */
#define BAUDR110x00001600     /* BAUDRATE = CCLK / 2**(2 + 11) = CCLK/16384 */
#define BAUDR120x00001800     /* BAUDRATE = CCLK / 2**(2 + 12) = CCLK/32768 */
#define BAUDR130x00001A00     /* BAUDRATE = CCLK / 2**(2 + 13) = CCLK/65536 */
#define BAUDR140x00001C00     /* BAUDRATE = CCLK / 2**(2 + 14) = CCLK/131072 */
#define BAUDR150x00001E00     /* BAUDRATE = CCLK / 2**(2 + 15) = CCLK/262144 */
#define TDMAEN 0x00002000     /* SPITX transmit buffer DMA enable, DMA channel 9 */
#define PSSE   0x00004000     /* Programmable slave device select */
#define FLS0   0x00008000     /* FLAG0 slave device select enable */
#define FLS1   0x00010000     /* FLAG1 slave device select enable */
#define FLS2   0x00020000     /* FLAG2 slave device select enable */
#define FLS3   0x00040000     /* FLAG3 slave device select enable */
#define SMLS   0x00080000     /* Seamless operation
#define NSMLS  0x00080000     /* Seamless operation */
#define DCPH0  0x00100000     /* Select or deselect SPIDS~ between transfers */
#define DMISO  0x02000000     /* Disable MISO Pin for Broadcast Mode */
#define OPD    0x04000000     /* Open drain output enable for data pins */
#define RDMAEN 0x08000000     /* SPIRX recevie buffer DMA enable, DMA channel 8 */
```

```
#define PACKEN 0x10000000     /* 8-to-16 Bit Packing Enable */
#define SGN    0x20000000     /* Sign-extend SPIRX/SPITX data */
#define SENDZ  0x40000000     /* Send zero or repeat previous data when SPITX empty */
#define SENDLW 0x40000000     /* Send zero or repeat previous data when SPITX empty */
#define GM     0x80000000     /* Retrieve or discard incoming data when SPIRX full */


/* SPISTAT register */
#define SPIF   0x00000001     /* SPI transmit or receive transfer complete (in pre 1.2 Si)*/
#define SRS    0x00000001     /* SPI shift register status (in 1.2 Si and above)*/
#define MME    0x00000002     /* Multimaster error */
#define TXE    0x00000004     /* SPITX transmission error (underflow) */
#define TXS0   0x00000008     /* TXS[0] - SPITX data buffer status */
#define TXS1   0x00000010     /* TXS[1] - SPITX data buffer status */
#define RBSY   0x00000020     /* SPIRX reception error (overflow) */
#define RXS0   0x00000040     /* RXS[0] - SPIRX data buffer status */
#define RXS1   0x00000080     /* RXS[1] - SPIRX data buffer status */


/* LCTL register - 0xcc */
#define L0EN   0x00000001     /* Link buffer 0 enable */
#define L0DEN  0x00000002     /* Link buffer 0 DMA enable */
#define L0CHEN 0x00000004     /* Link buffer 0 DMA chaining enable */
#define L0TRAN 0x00000008     /* Link buffer 0 data direction */
#define L0EXT  0x00000010     /* Link buffer 0 extended word size */
#define L0CLKD00x00000020     /* L0CLKD[0] Link buffer 0 CCLK divide ratio */
#define L0CLKD10x00000040     /* L0CLKD[1] Link buffer 0 CCLK divide ratio */
#define L0PDRDE0x00000100     /* Link Port 0 pulldown resister disable */
#define L0DPWID0x00000200     /* Link buffer 0 data path width */
#define L1EN   0x00000400     /* Link buffer 1 enable */
#define L1DEN  0x00000800     /* Link buffer 1 DMA enable */
#define L1CHEN 0x00001000     /* Link buffer 1 DMA chaining enable */
#define L1TRAN 0x00002000     /* Link buffer 1 data direction */
#define L1EXT  0x00004000     /* Link buffer 1 extended word size */
#define L1CLKD00x00008000     /* L1CLKD[0] Link buffer 1 CCLK divide ratio */
#define L1CLKD10x00010000     /* L1CLKD[1] Link buffer 1 CCLK divide ratio */
#define L1PDRDE0x00040000     /* Link Port 1 pulldown resister disable */
#define L1DPWID0x00080000     /* Link buffer 1 data path width */
#define A0LB   0x00100000     /* Link Port Assignment for LBUF0 - 2106x/21160 compatibility */
#define A1LB   0x00200000     /* Link Port Assignment for LBUF1 - 2106x/21160 compatibility */
#define LAB0   0x00100000     /* Link Port Assignment for LBUF0 -new naming conventions    */
#define LAB1   0x00200000     /* Link Port Assignment for LBUF1 - new naming conventions    */
#define L0STAT00x00400000     /* L0STAT[0] - link buffer 0 status (read-only) */
#define L0STAT10x00800000     /* L0STAT[1] - link buffer 0 status (read-only) */
#define L1STAT00x01000000     /* L1STAT[0] - link buffer 1 status (read-only) */
#define L1STAT10x02000000     /* L1STAT[1] - link buffer 1 status (read-only) */
#define LRERR0 0x04000000     /* Link Buffer 0 receive pack error status */
#define LRERR1 0x08000000     /* Link Buffer 1 receive pack error status */


/* SP02MCTL & SP13MCTL registers */
#define MCE    0x00000001     /* Multichannel Mode Enable */
#define MFD0   0x00000000     /* no frame delay, multichannel FS pulse in same SCLK cycle as first data
bit */
#define MFD1   0x00000002     /* multichannel mode 1 cycle frame sync delay */
#define MFD2   0x00000004     /* multichannel mode 2 cycle frame sync delay */
#define MFD3   0x00000006     /* multichannel mode 3 cycle frame sync delay */
#define MFD4   0x00000008     /* multichannel mode 4 cycle frame sync delay */
#define MFD5   0x0000000A     /* multichannel mode 5 cycle frame sync delay */
#define MFD6   0x0000000C     /* multichannel mode 6 cycle frame sync delay */
#define MFD7   0x0000000E     /* multichannel mode 7 cycle frame sync delay */
#define MFD8   0x00000010     /* multichannel mode 8 cycle frame sync delay */
#define MFD9   0x00000012     /* multichannel mode 9 cycle frame sync delay */
#define MFD10  0x00000014     /* multichannel mode 10 cycle frame sync delay */
#define MFD11  0x00000016     /* multichannel mode 11 cycle frame sync delay */
#define MFD12  0x00000018     /* multichannel mode 12 cycle frame sync delay */
#define MFD13  0x0000001A     /* multichannel mode 13 cycle frame sync delay */
#define MFD14  0x0000001C     /* multichannel mode 14 cycle frame sync delay */
#define NCH    0x00000FE0     /* Number of MCM channels - 1 */
```

```
#define SPL    0x00001000     /* SPORT 0&2 or SPORT 1&3 Internal Loopback Mode */
#define CHNL   0x007F0000     /* Current Channel Status (read-only) */


/* SPCTL0, SPCTL1, SPCTL2 and SPCTL3 registers */
#define       SPEN_A         0x00000001/* SPORT enable primary A channel */
#define       DTYPE0         0x00000000/* right justify, fill unused MSBs with 0s */
#define        DTYPE1        0x00000002/* right justify, sign-extend into unused MSBs */
#define        DTYPE2          0x00000004/* compand using mu law */
#define        DTYPE3          0x00000006/* compand using a law */
#define       SENDN            0x00000008/* MSB or LSB first */
#define       SLEN3            0x00000020/* serial length 3 */
#define       SLEN4            0x00000030/* serial length 4 */
#define       SLEN5            0x00000040/* serial length 5 */
#define       SLEN6            0x00000050/* serial length 6 */
#define       SLEN7            0x00000060/* serial length 7 */
#define       SLEN8            0x00000070/* serial length 8 */
#define       SLEN9            0x00000080/* serial length 9 */
#define       SLEN10           0x00000090/* serial length 10 */
#define       SLEN11           0x000000A0/* serial length 11 */
#define       SLEN12           0x000000B0/* serial length 12 */
#define       SLEN13           0x000000C0/* serial length 13 */
#define       SLEN14           0x000000D0/* serial length 14 */
#define       SLEN15           0x000000E0/* serial length 15 */
#define       SLEN16           0x000000F0/* serial length 16 */
#define       SLEN17           0x00000100/* serial length 17 */
#define       SLEN18           0x00000110/* serial length 18 */
#define       SLEN19           0x00000120/* serial length 19 */
#define       SLEN20           0x00000130/* serial length 20 */
#define       SLEN21           0x00000140/* serial length 21 */
#define       SLEN22           0x00000150/* serial length 22 */
#define       SLEN23           0x00000160/* serial length 23 */
#define       SLEN24         0x00000170/* serial length 24 */
#define       SLEN25         0x00000180/* serial length 25 */
#define       SLEN26         0x00000190/* serial length 26 */
#define       SLEN27         0x000001A0/* serial length 27 */
#define       SLEN28         0x000001B0/* serial length 28 */
#define       SLEN29         0x000001C0/* serial length 29 */
#define       SLEN30         0x000001D0/* serial length 30 */
#define       SLEN31         0x000001E0/* serial length 31 */
#define       SLEN32         0x000001F0/* serial length 32 */
#define        PACK          0x00000200/* 16-to-32 data packing */
#define        MSTR          0x00000400/* I2S Mode only... TX/RX is master or slave */
#define        ICLK          0x00000400/* internally 1 or externally 0 generated transmit or recieve
SCLKx */
#define        OPMODE        0x00000800/* I2S mode enable ('1') or DSP Serial Mode/Multichannel mode
('0') */
#define        CKRE          0x00001000/* Clock edge for data and frame sync sampling (rx) or driving
(tx) */
#define        FSR           0x00002000/* transmit or receive frame sync (FSx) required */
#define        IFS           0x00004000/* internally generated transmit or receive frame sync (FSx) */
#define        IRFS          0x00004000/* internally generated receive FS0 or FS1 in multichannel mode
*/
#define        DITFS         0x00008000/* (I2S/DSP serial mode only) Data Independent tx FSx when DDIR
bit=1 */
#define       LFS            0x00010000/* Active Low transmit or receive frame sync (FSx) */
#define        LRFS          0x00010000/* SPORT0 and SPORT1 active low TDM frame sync FS0/FS1 in MC
mode */
#define        LTDV          0x00010000/* (MC Mode only) SPORT2/SPORT3 tx data valid ena in TDM
mode-TDV2/TDV3 alternate pin config */
#define        LFIRST        0x00010000/* (I2S Mode Only) transmit left channel first 1, or right
channel first 0 */
#define        LAFS          0x00020000/* (DSP Serial Mode only) Late (vs early) frame sync FSx */
#define        SDEN_A        0x00040000/* SPORT TXnA/RXnA DMA enable primary A channel */
#define        SCHEN_A       0x00080000/* SPORT TXnA/RXnA DMA chaining enable primary A channel */
#define        SDEN_B        0x00100000/* SPORT TXnB/RXnB DMA enable primary B channel */
#define        SCHEN_B       0x00200000/* SPORT TXnB/RXnB DMA chaining enable primary B channel */
#define        FS_BOTH       0x00400000/* (DSP Serial & I2S modes only) Issue FSx only if data is in
both TXnA & TXnB regs */
```

```
#define         SPEN_B        0x01000000/* SPORTx secondary B channel enable */
#define         DDIR          0x02000000/* SPORT data buffer data dirrection 1 = transmitter, 0 = 
receiver */
#define         DERR_B        0x04000000/* SPORTx secondary B overflow/underflow error status in DSP 
serial & I2S modes (read-only) */
#define         DXS0_B        0x08000000/* SPORTx secondary B data buffer status in DSP Serial & I2S 
modes

read-only)*/
#define         DXS1_B        0x10000000/* SPORTx secondary B data buffer status in DSP Serial & I2S 
modes (read-only)*/
#define         DERR_A        0x20000000/* SPORTx primary A over/underflow error status in DSP Serial & 
I2S modes (read-only) */
#define         TUVF_A        0x20000000/* SPORT2/SPORT3 TX2A/TX3A underflow status in MC mode 
(read-only, sticky)*/
#define         ROVF_A        0x20000000/* SPORT0/SPORT1 RX0A/RX1A overflow status in MC mode 
(read-only, sticky)*/
#define         DXS0_A        0x40000000/* SPORTx primary A data buffer status in DSP serial and I2S 
modes (read-only)*/
#define         DXS1_A        0x80000000/* SPORTx primary A data buffer status in DSP serial and I2S 
modes (read-only)*/
#define         RXS0_A        0x40000000/* SPORT0/SPORT1 RX0A/RX1A data buffer status in MC mode 
(read-only)*/
#define         RXS1_A        0x80000000/* SPORT0/SPORT1 RX0A/RX1A data buffer status in MC mode 
(read-only)*/
#define         TXS0_A        0x40000000/* SPORT2/SPORT3 TX2A/TX3A data buffer status in MC mode 
(read-only)*/
#define         TXS1_A        0x80000000/* SPORT2/SPORT3 TX2A/TX3A data buffer status in MC mode 
(read-only) */

#endif
```

B INTERRUPT VECTOR ADDRESSES

Table B-1 shows all ADSP-21161 processor interrupts, listed according their bit position in the IRPTL, LIRPTL, and IMASK registers. For more information, see "Interrupt Latch Register (IRPTL)" on page A-27 and "Interrupt Mask Register (IMASK)" on page A-31. Also shown is the address of the interrupt vector. Each vector is separated by four memory locations. The addresses in the vector table represent offsets from a base address. For an interrupt vector table in internal memory, the base address is 0x0004 0000. For an interrupt vector table in external memory, the base address is 0x0020 0000. The interrupt name column in Table B-1 lists a mnemonic name for each interrupt as they are defined by the def21161.h file that comes with the software development tools. For more information, see "Register and Bit #Defines (def21161.h)" on page A-121.

Table B-1. Interrupt Vector Addresses

Register	**IRPTL/ IMASK, LIRPTL Bit#**	**Vector Address**	**Interrupt Name**	**Function**
IRPTL	0	0x00	EMUI	Emulator (read-only, non-maskable) HIGHEST PRIORITY
IRPTL	1	0x04	RSTI	Reset (read-only, non-maskable)
IRPTL	2	0x08	IICDI	Illegal Input Condition Detected
IRPTL	3	0x0C	SOVFI	Status, loop, or mode stack overflow; or PC stack full
IRPTL	4	0x10	TMZHI	Timer=0 (high priority option)

Table B-1. Interrupt Vector Addresses (Cont'd)

Register	IRPTL/ IMASK, LIRPTL Bit#	Vector Address	Interrupt Name	Function
IRPTL	5	0x14	VIRPTI	Multiprocessor Vector Interrupt
IRPTL	6	0x18	IRQ2I	$\overline{\text{IRQ2}}$ asserted
IRPTL	7	0x1C	IRQ1I	$\overline{\text{IRQ1}}$ asserted
IRPTL	8	0x20	IRQ0I	$\overline{\text{IRQ0}}$ asserted
IRPTL	9	0x24	-	Reserved
IRPTL	10	0x28	SP0I	SPORT0 DMA
IRPTL	11	0x2C	SP1I	SPORT1 DMA
IRPTL	12	0x30	SP2I	SPORT2 DMA
IRPTL	13	0x34	SP3I	SPORT3 DMA
LIRPTL	0/16	0x38	LP0I	Link Buffer 0 DMA Interrupt
LIRPTL	1/17	0x3C	LP1I	Link Buffer 1 DMA Interrupt
LIRPTL	2/18	0x40	SPIRI	SPI Receive DMA Interrupt
LIRPTL	3/19	0x44	SPITI	SPI Transmit DMA Interrupt
LIRPTL	-	0x48	-	Reserved
LIRPTL	-	0x4c	-	Reserved
IRPTL	15	0x50	EP0I	DMA Channel 10 - Ext. Port Buffer 0
IRPTL	16	0x54	EP1I	DMA Channel 11 - Ext. Port Buffer 1
IRPTL	17	0x58	EP2I	DMA Channel 12 - Ext. Port Buffer 2
IRPTL	18	0x5C	EP3I	DMA Channel 13 - Ext. Port Buffer 3
IRPTL	19	0x60	LSRQI	Link Port Service Request
IRPTL	20	0x64	CB7I	Circular Buffer 7 overflow
IRPTL	21	0x68	CB15I	Circular Buffer 15 overflow
IRPTL	22	0x6C	TMZLI	Timer=0 (low priority option)

Table B-1. Interrupt Vector Addresses (Cont'd)

Register	IRPTL/ IMASK, LIRPTL Bit#	Vector Address	Interrupt Name	Function
IRPTL	23	0x70	FIXI	Fixed-point overflow
IRPTL	24	0x74	FLTOI	Floating-point overflow exception
IRPTL	25	0x78	FLTUI	Floating-point underflow exception
IRPTL	26	0x7C	FLTII	Floating-point invalid exception
IRPTL	27	0x80	SFT0I	User software interrupt 0
IRPTL	28	0x84	SFT1I	User software interrupt 1
IRPTL	29	0x88	SFT2I	User software interrupt 2
IRPTL	30	0x8C	SFT3I	User software interrupt 3
IRPTL	31	0x90	-	Reserved - lowest priority

C NUMERIC FORMATS

The ADSP-21161 processor supports the 32-bit single-precision floating-point data format defined in the IEEE Standard 754/854. In addition, the processor supports an extended-precision version of the same format with eight additional bits in the mantissa (40 bits total). The processor also supports 32-bit fixed-point formats—fractional and integer—which can be signed (twos-complement) or unsigned.

IEEE Single-Precision Floating-Point Data

IEEE Standard 754/854 specifies a 32-bit single-precision floating-point format, shown in Figure C-1. A number in this format consists of a sign bit `s`, a 24-bit significand, and an 8-bit unsigned-magnitude exponent `e`.

For normalized numbers, the significand consists of a 23-bit fraction f and a “hidden” bit of 1 that is implicitly presumed to precede f_{22} in the significand. The binary point is presumed to lie between this hidden bit and f_{22}. The least significant bit (LSB) of the fraction is f_0; the LSB of the exponent is e_0.

The hidden bit effectively increases the precision of the floating-point significand to 24 bits from the 23 bits actually stored in the data format. It also insures that the significand of any number in the IEEE normalized number format is always greater than or equal to 1 and less than 2.

The unsigned exponent e can range between 1 ≤ e ≤ 254 for normal numbers in the single-precision format. This exponent is biased by +127 (254 ¸ 2). To calculate the true unbiased exponent, 127 must be subtracted from e.

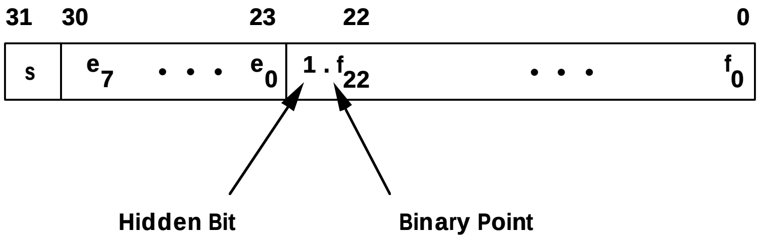


Figure C-1. IEEE 32-Bit Single-Precision Floating-Point

The IEEE Standard also provides for several special data types in the single-precision floating-point format:

- An exponent value of 255 (all ones) with a nonzero fraction is a Not-A-Number (NAN). NANs are usually used as flags for data flow control, for the values of uninitialized variables, and for the results of invalid operations such as 0 * ∞.
- Infinity is represented as an exponent of 255 and a zero fraction. Note that because the fraction is signed, both positive and negative Infinity can be represented.
- Zero is represented by a zero exponent and a zero fraction. As with Infinity, both positive Zero and negative Zero can be represented.

The IEEE single-precision floating-point data types supported by the processor and their interpretations are summarized in Table C-1.

Table C-1. IEEE Single-Precision Floating-Point Data Types

Type	Exponent	Fraction	Value
NAN	255	Nonzero	Undefined
Infinity	255	0	$(-1)^s$ Infinity
Normal	$1 \leq e \leq 254$	Any	$(-1)^s (1.f_{22-0}) 2^{e-127}$
Zero	0	0 $(-1)^s$ Zero	

Extended-Precision Floating-Point

The extended-precision floating-point format is 40 bits wide, with the same 8-bit exponent as in the standard format but a 32-bit significand. This format is shown in Figure C-2. In all other respects, the extended floating-point format is the same as the IEEE standard format.

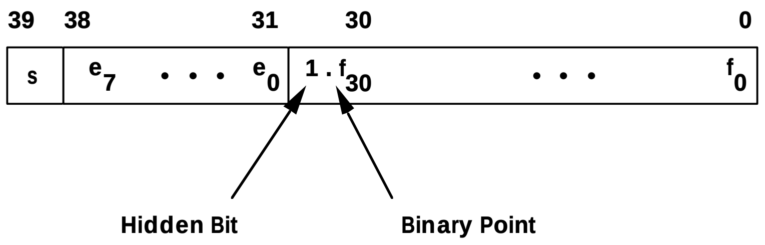


Figure C-2. 40-Bit Extended-Precision Floating-Point Format

Short Word Floating-Point Format

The ADSP-21161 processor supports a 16-bit floating-point data type and provides conversion instructions for it. The short float data format has an 11-bit mantissa with a four-bit exponent plus sign bit, as shown in Figure C-3. The 16-bit floating-point numbers reside in the lower 16 bits of the 32-bit floating-point field.

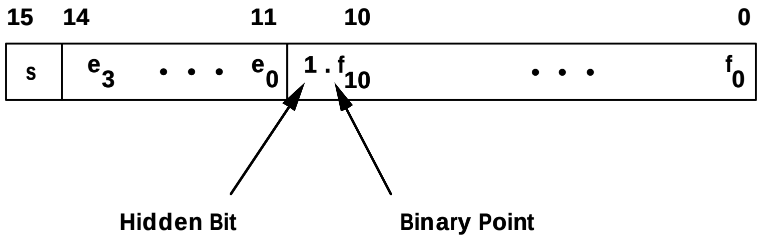


Figure C-3. 16-Bit Floating-Point Format

Packing for Floating-Point Data

Two shifter instructions, `FPACK` and `FUNPACK`, perform the packing and unpacking conversions between 32-bit floating-point words and 16-bit floating-point words. The `FPACK` instruction converts a 32-bit IEEE floating-point number to a 16-bit floating-point number. `FUNPACK` converts the 16-bit floating-point numbers back to 32-bit IEEE floating-point. Each instruction executes in a single cycle. The results of the `FPACK` and `FUNPACK` operations appear in Table C-2 and Table C-3.

The short float type supports gradual underflow. This method sacrifices precision for dynamic range. When packing a number which would have underflowed, the exponent is set to zero and the mantissa (including "hidden" 1) is right-shifted the appropriate amount. The packed result is a denormal which can be unpacked into a normal IEEE floating-point number.

During the `FPACK` operation, an overflow sets the `SV` condition and non-overflow will clear it. During the `FUNPACK` operation, the `SV` condition is cleared. The `SZ` and `SS` conditions are cleared by both instructions.

Table C-2. FPACK Operations

Condition	Result
135 < exp	Largest magnitude representation.
120 < exp ≤ 135	Exponent is MSB of source exponent concatenated with the three LSBs of source exponent. The packed fraction is the rounded upper 11 bits of the source fraction.
109 < exp ≤ 120	Exponent=0. Packed fraction is the upper bits (source exponent – 110) of the source fraction prefixed by zeros and the "hidden" 1. The packed fraction is rounded.
exp < 110	Packed word is all zeros.
exp = source exponent **sign bit remains the same in all cases**	

Table C-3. FUNPACK Operations

Condition	Result
0 < exp ≤ 15	Exponent is the 3 LSBs of the source exponent prefixed by the MSB of the source exponent and four copies of the complement of the MSB. The unpacked fraction is the source fraction with 12 zeros appended.
exp = 0	Exponent is (120 – N) where N is the number of leading zeros in the source fraction. The unpacked fraction is the remainder of the source fraction with zeros appended to pad it and the "hidden" 1 stripped away.
exp = source exponent **sign bit remains the same in all cases**	

Fixed-Point Formats

The ADSP-21161 processor supports two 32-bit fixed-point formats: fractional and integer. In both formats, numbers can be signed (twos-complement) or unsigned. The four possible combinations are shown in Figure C-4. In the fractional format, there is an implied binary point to the left of the most significant magnitude bit. In integer format, the binary point is understood to be to the right of the LSB. Note that the sign bit is negatively weighted in a twos-complement format.

ALU outputs always have the same width and data format as the inputs. The multiplier, however, produces a 64-bit product from two 32-bit inputs. If both operands are unsigned integers, the result is a 64-bit unsigned integer. If both operands are unsigned fractions, the result is a 64-bit unsigned fraction. These formats are shown in Figure C-5.

If one operand is signed and the other unsigned, the result is signed. If both inputs are signed, the result is signed and automatically shifted left one bit. The LSB becomes zero and bit 62 moves into the sign bit position. Normally bit 63 and bit 62 are identical when both operands are signed. (The only exception is full-scale negative multiplied by itself.) Thus, the left shift normally removes a redundant sign bit, increasing the precision of the most significant product. Also, if the data format is fractional, a single-bit left shift renormalizes the MSP to a fractional format. The signed formats with and without left shifting are shown in Figure C-6.

The multiplier has an 80-bit accumulator to allow the accumulation of 64- bit products. For more information on the multiplier and accumulator, see "Multiply—Accumulator (Multiplier)" on page 2-15.

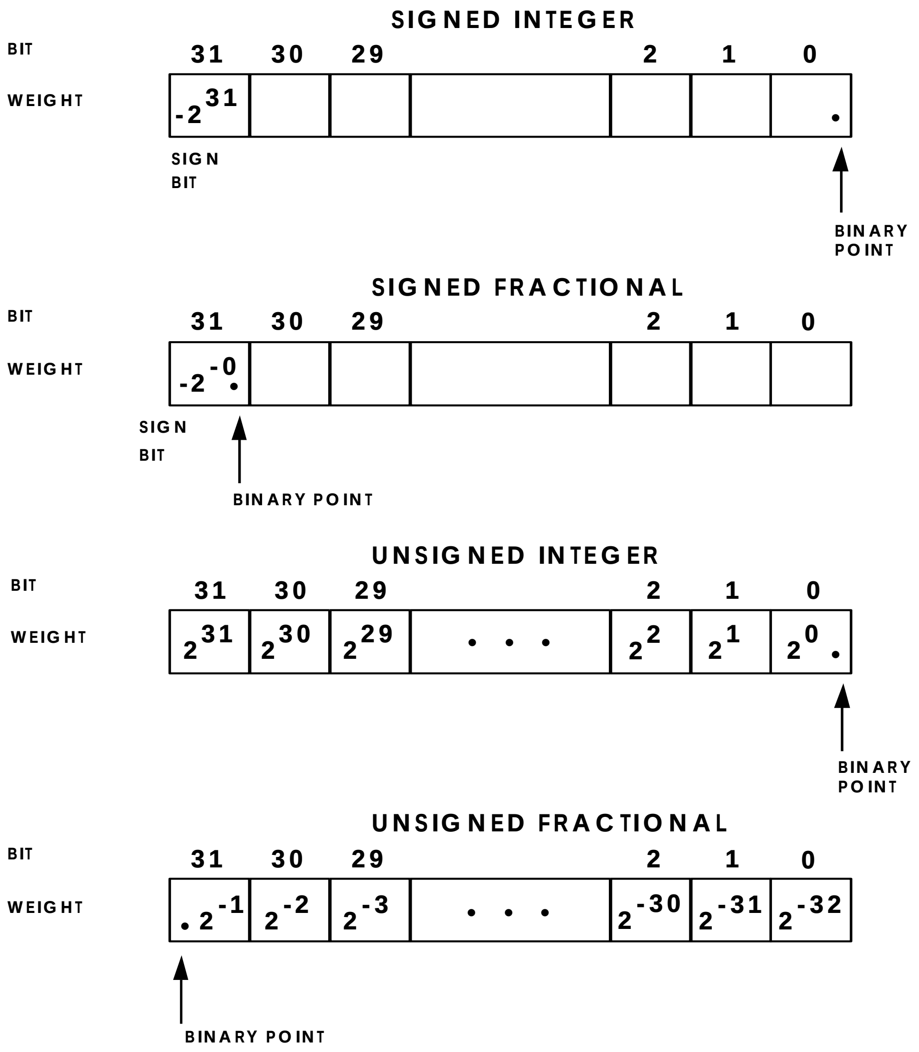


Figure C-4. 32-Bit Fixed-Point Formats

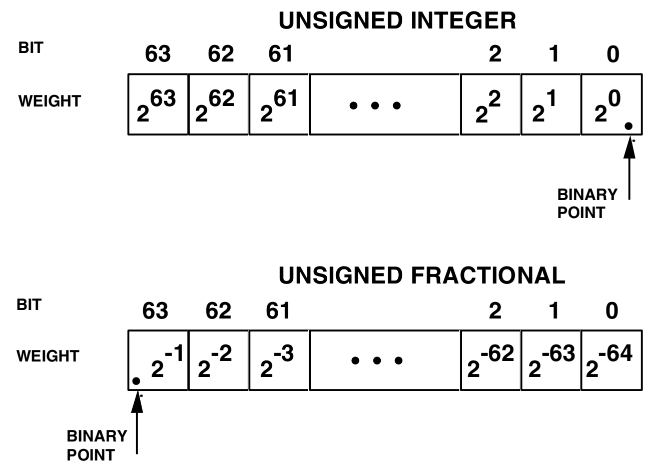


Figure C-5. 64-Bit Unsigned Fixed-Point Product

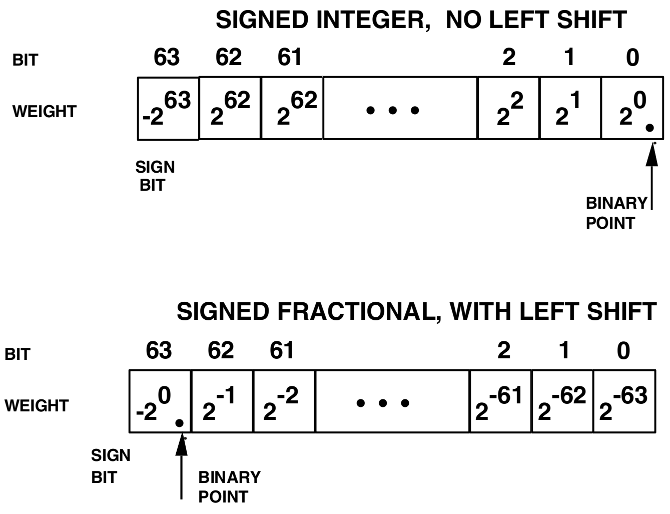


Figure C-6. 64-Bit Signed Fixed-Point Product

G GLOSSARY

Autobuffering Unit (ABU). (See I/O processor and DMA)

Arithmetic Logic Unit (ALU). This part of a processing element performs arithmetic and logic operations on fixed-point and floating-point data.

Asynchronous transfers. Asynchronous host accesses of the processor. After acquiring control of the processor's external bus, the host must assert the $\overline{\texttt{CS}}$ pin of the processor it wants to access.

Auxiliary registers. (See Index Registers)

Base address. The starting address of a circular buffer to which the DAG wraps around. This address is stored in a DAG `Bx` register.

Base registers. A base (`Bx`) register is a Data Address Generator (DAG) register that sets up the starting address for a circular buffer.

Bit-reverse addressing. The Data Address Generator (DAG) provides a bit-reversed address during a data move without reversing the stored address.

Block repeat. (See Do/Until instructions in the *ADSP-21160 DSP Instruction Set Reference*)

Block size register. (See Length Registers)

Boot Memory Space. The processor supports an external boot EPROM mapped to external memory and selected with the $\overline{\texttt{BMS}}$ pin. The boot EPROM provides one of the methods for automatically loading a program into the internal memory of the processor after power-up or after a software reset.

Broadcast data moves. The Data Address Generator (DAG) performs dual data moves to complementary registers in each processing element to support SIMD mode.

Buffered serial port. (See Serial ports)

Burst transfers. Multi-cycle synchronous transfers that contains a packet of at least two 64-bit transfers. For a master, only a DMA channel can master a burst transaction. As a slave, supports burst read transfers from internal memory, or the EPBx data buffers.

Bus slave or slave mode. A processor can be a bus slave to another processor or to a host processor. The processor becomes a host bus slave when the $\overline{\text{HBG}}$ signal is returned.

Bus Transition Cycle (BTC). A cycle in which control of the external bus is passed from one processor to another (in a multiprocessor system).

Circular buffer addressing. The DAG uses the Ix, Mx and Lx register settings to constrain addressing to a range of addresses. This range contains data that the DAG steps through repeatedly, "wrapping around" to repeat stepping through the range of addresses in a circular pattern.

Cluster multiprocessing. This is a multiprocessing system architecture in which the processor uses its link ports and external port for inter-processor communication.

Companding (compressing/expanding). This is the process of logarithmically encoding and decoding data to minimize the number of bits that must be sent.

Conditional branches. These are JUMP or CALL/return instructions whose execution is based on testing an IF condition.

Conflict resolution ratio. Because the external port must arbitrate accesses over three internal buses to one external bus, there is a 3:1 conflict resolution ratio at the external port interface. This ratio plus the 2:1 or greater clock ratio between the processor's internal clock and the external system clock forces systems that fetch instructions or data through the external port must tolerate at least one cycle—and usually many additional cycles—of latency.

DAGEN, Data address generator (See DAGs)

Data Address Generator (DAG). The Data Address Generators (DAGs) provide memory addresses when data is transferred between memory and registers.

Data flow multiprocessing. This is a multiprocessor system architecture in which the processor uses its link ports for inter-processor communication.

Data register file. This is the set of data registers that transfer data between the data buses and the computation units. These registers also provide local storage for operands and results.

Data registers (Dreg). These are registers in the PEx and PEy processing elements. These registers are hold operands for multiplier, ALU, or shifter operations and are denoted as `Rx` when used for fixed point operations or `Fx` when used for floating-point operations.

Deadlock resolution. When both theprocessor subsystem and the system try to access each other's bus in the same cycle, a deadlock may occur in which neither access can complete. Techniques for resolving deadlock vary with the interface: DRAM, host, or multiprocessor system.

Delayed branches. These are `JUMPS` and `CALL`/return instructions with the delayed branches `(DB)` modifier. In delayed branches, no instruction cycles are lost in the pipeline, because the processor executes the two instructions after the branch while the pipeline fills with instructions from the new branch.

Direct branches. These are `JUMP` or `CALL`/return instructions that use an absolute—not changing at runtime—address (such as a program label) or use a PC-relative address.

Direct reads and writes. A direct access of the processor's internal memory or I/O processor registers by another processor or by a host processor.

DMA (Direct Memory Accessing). The processor's I/O processor supports DMA of data between processor memory and external memory, host, or peripherals through the external, link, and serial ports. Each DMA operation transfers an entire block of data.

DMA chaining. The processor supports chaining together multiple DMA sequences. In chained DMA, the I/O processor loads the next Transfer Control Block (DMA parameters) into the DMA parameter registers when the current DMA finishes and auto-initializes the next DMA sequence.

DMA parameter registers. These registers function similarly to data address generator registers, setting up a memory access process. These registers include Internal Index registers (`IIx`), Internal Modify registers (`IMx`), Count registers (`Cx`), Chain Pointer registers (`CPx`), General Purpose registers (`GPx`), External Index registers (`EIEPx`), External Modify registers (`EMEPx`), and External Count registers (`ECEPx`).

DMA TCB chain loading. This is the process that the I/O processor uses for loading the TCB of the next DMA sequence into the parameter registers during chained DMA.

DMACx control registers. The DMA control registers for the `EPBx` external port buffers: `DMAC10`, `DMAC11`, `DMAC12`, and `DMAC13`. These correspond respectively to `EPB0`, `EPB1`, `EPB2`, and `EPB3`.

Edge-sensitive interrupt. The processor detects this type of interrupt if the input signal is high (inactive) on one cycle and low (active) on the next cycle when sampled on the rising edge of `CLKIN`.

Endian format, little versus big. The processor uses big-endian format—moves data starting with most-significant-bit and finishing with least significant bit—in almost all instances. The two exceptions are bit order for data transfer through the serial port and word order for packing through the external port. For compatibility with little-endian (least significant-first) peripherals, the processor supports both big- and little-endian bit order data transfers. Also for compatibility little-endian hosts, the processor supports both big- and little endian word order data transfers.

Explicit Versus Implicit operations. In SIMD mode, identical instructions execute on the PEx and PEy computational units; the difference is the data. The data registers for PEy operations are identified (implicitly) from the PEx registers in the instruction. This implicit relation between PEx and PEy data registers corresponds to complementary register pairs.

External bus. The processor extends the following signals off-chip as an external bus: DATA47-16, ADDR23-0, $\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{MS}}$3-0, $\overline{\text{BMS}}$, CLKOUT, BRST, ACK, and $\overline{\text{SBTS}}$.

External memory space. This space ranges from address 0x0200 0000 through 0x0CFF FFFF (Normal word) for Non-SDRAM and from address 0x0020 0000 through 0x0FFF FFFF (Normal word) for SDRAM. External memory space refers to the off-chip memory or memory mapped peripherals that are attached to the processor's external address (ADDR23-0) and data (DATA47-16) buses.

External port FIFO buffers (EPB0, EPB1, EPB2, and EPB3). The I/O processor registers used for external port DMA transfers and single-word data transfers (from other processors or from a host processor). These buffers are eight-deep FIFOs.

External port. This port extends the internal address and data buses off chip, providing the processor's interface to off-chip memory and peripherals.

Field deposit (Fdep) instructions. These shifter instructions take a group of bits from the input register (starting at the LSB of the 32-bit integer field) and deposit the bits as directed anywhere within the result register.

Field extract (Fext) instructions. These shifter extract a group of bits as directed from anywhere within the input register and place them in the result register (aligned with the LSB of the 32-bit integer field).

Programmable Flag pins. These pins (FLGx) can be programmed as input or output pins using bit settings in the MODE2 register. The status of the flag pins is given in the FLAGS or IOFLAG register.

General-purpose input/output pins. (See Programmable Flag pins)

Flag update. The processor's update to status flags occurs at the end of the cycle in which the status is generated and is available on the next cycle.

Harvard architecture. A memory architectures that has separate buses for program and data storage. The two buses allow a data word and an instruction simultaneously.

Hold time cycle. This is an inactive bus cycle that is automatically generates at the end of a read or write (depending on the external port access mode) to allow a longer hold time for address and data. The address—and data, if a write—remains unchanged and is driven for one cycle after the read or write strobes are deasserted.

Host transition cycle (HTC). A cycle in which control of the external bus is passed from the ADSP-21161 processor to the host processor. During this cycle the processor stops driving the $\overline{\text{RD}}$, $\overline{\text{WR}}$, ADDR23-0, $\overline{\text{MS}}$3-0, CLKOUT, $\overline{\text{PA}}$, and $\overline{\text{DMAGx}}$ signals, which must then be driven by the host.

I/O processor register. One of the control, status, or data buffer registers of the processor's on-chip I/O processor.

Idle cycle. This is an inactive bus cycle that is automatically generated (depending on the external port access mode) to avoid data bus driver conflicts. Such a conflict can occur when a device with a long output disable time continues to drive after $\overline{\text{RD}}$ is deasserted while another device begins driving on the following cycle.

IDLE. An instruction that causes the processor to cease operations, holding its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

Index registers. An index register is a Data Address Generator (DAG) register that holds an address and acts as a pointer to memory.

Indirect branches. These are JUMP or CALL/return instructions that use a dynamic—changes at runtime—address that comes from the PM data address generator.

Interleaved data. To take advantage of the processor's data accesses to 4- and 3-column locations, programs must adjust the interleaving of data into (not necessarily sequential) memory locations to accommodate the memory access mode.

Internal memory space. This space ranges from address 0x0000 0000 through 0x0005 3FFF (Normal word). Internal memory space refers to the ADSP-21161 processor's on-chip SRAM and memory mapped registers.

Interrupts. Subroutines in which a runtime event (not an instruction) triggers the execution of the routine.

JTAG port. This port supports the IEEE standard 1149.1 Joint Test Action Group (JTAG) standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system.

Jumps. Program flow transfers permanently to another part of program memory.

Link ports. The processor has two 8-bit wide link ports, which can connect to other processors' or peripherals' link ports. These bidirectional ports have eight data lines, an acknowledge, and a clock line.

Length registers. A length registers is a Data Address Generator (DAG) register that sets up the range of addresses a circular buffer.

Level-sensitive interrupts. This type of interrupt is detected if the signal input is low (active) when sampled on the rising edge of CLKIN.

Loops. One sequence of instructions executes several times with zero overhead.

McBSP, multichannel buffered serial port. (See Serial port)

MCM, multichannel mode. (See Multichannel mode)

Memory access modes. The processor supports Asynchronous and Synchronous modes for accessing external memory space. In asynchronous access mode, the processor's $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes change before `CLKIN`'s edge. In synchronous access mode, the processor's $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes change on `CLKIN`'s edge.

Memory blocks and banks. Memory is divided into **blocks** that are each associated with different data address generators. The processor's external memory spaces is divided into **banks**, which may be addressed by either data address generator.

Modified addressing. The DAG generates an address that is incremented by a value or a register.

Modify address. The Data Address Generator (DAG) increments the stored address without performing a data move.

Modify registers. A modify register is a Data Address Generator (DAG) register that provides the increment or step size by which an index register is pre- or post-modified during a register move.

Multichannel mode. In this mode, each data word of the serial bit stream occupies a separate channel.

Multifunction computations. Using the many parallel data paths within its computational units, the processor supports parallel execution of multiple computational instructions. These instructions complete in a single cycle, and they combine parallel operation of the multiplier and the ALU or dual ALU functions. The multiple operations perform the same as if they were in corresponding single-function computations.

Multiplier. This part of a processing element does floating-point and fixed-point multiplication and executes fixed-point multiply/add and multiply/subtract operations.

Multiprocessor memory space. The portion of the memory map that includes the I/O processor registers of in a multiprocessing system. This address space is mapped into the unified address space of the ADSP-21161 processor.

Multiprocessor system. A system with multiple processors, with or without a host processor. The processors are connected by the external bus and/or link ports.

Multiprocessor vector interrupt. The vector interrupt (`VIRPT`) permits passing interprocessor commands in multiple-processor systems. One processor writes a vector address to another processors `VIRPT` register. Writing the address initiates the vector interrupt on the processor that receives the write. The ADSP-21161 processor executes (vectors to) the interrupt service routine at that address.

Neighbor registers. In Long word addressed accesses, data is moved to or from two neighboring data registers. The least-significant-32-bits moves to or from the explicit (named) register in the neighbor register pair. In forced Long word accesses (Normal word address with `LW` mnemonic), the the Normal word address is converted to Long word, placing the even Normal word location in the explicit register and the odd Normal word location in the other register in the neighbor pair.

PAGEN, Program address generation logic. (See the Program Sequencer chapter)

Peripherals. This refers to everything outside the processor core. The ADSP-21161 processor's peripherals include internal memory, external port, I/O processor, JTAG port, and any external devices that connect to the ADSP-21161.

Precision. The precision of a floating-point number depends on the number of bits after the binary point in the storage format for the number. The processor supports two high precision floating-point formats: 32-bit IEEE single-precision floating-point (which uses 8 bits for the exponent and 24 bits for the mantissa) and a 40-bit extended precision version of the IEEE format.

Post-modify addressing. The Data Address Generator (DAG) provides an address during a data move and auto-increments the stored address for the next move.

Pre-modify addressing. The Data Address Generator (DAG) provides a modified address during a data move without incrementing the stored address.

Registers swaps. This special type of register-to-register move instruction uses the special swap operator, <->. A register-to-register swap occurs when registers in different processing elements exchange values.

Saturation (ALU saturation mode). In this mode, all positive fixed-point overflows return the maximum positive fixed-point number (0x7FFF FFFF), and all negative overflows return the maximum negative number (0x8000 0000).

Semaphore. This is a flag that can be read and written by any of the processors sharing the resource. Semaphores can be used in multiprocessor systems to allow the processors to share resources such as memory or I/O. The value of the semaphore tells the processor when it can access the resource. Semaphores are also useful for synchronizing the tasks being performed by different processors in a multiprocessing system.

Serial ports. The ADSP-21161 processor has four synchronous serial ports that provide an inexpensive interface to a wide variety of digital and mixed-signal peripheral devices.

SHARC. This is an acronym for Super Harvard Architecture. This architecture balances a high performance processor core with high performance buses (PM, DM, IO).

Shifter. This part of a processing element completes logical shifts, arithmetic shifts, bit manipulation, field deposit, and field extraction operations on 32-bit operands. Also, the Shifter can derive exponents.

SMUL, Saturation on multiplication. (See Multiplier Saturation modes)

SST, Saturation on store. (See Multiplier Saturation modes)

Subroutines. The processor temporarily interrupts sequential flow to execute instructions from another part of program memory.

Single-word data transfers. Reads and writes to the `EPBx` external port buffers, performed externally by the bus master (or host) or internally by the slave's core. These occur when DMA is disabled in the `DMACx` control register.

Synchronous transfers. Synchronous host accesses of the ADSP-21161 processor. When $\overline{\texttt{CS}}$ is not asserted, the host must act like another processor in a multiprocessor system, by generating an address in multiprocessor memory space, asserting $\overline{\texttt{PA}}$ and $\overline{\texttt{WR}}$ or $\overline{\texttt{RD}}$, and driving out or latching in the data.

TADD, TDM address. (See the section "Multichannel Mode")

TCB chain loading. The process in which the DMA controller downloads a Transfer Control Block from memory and autoinitializes the DMA parameter registers.

Time Division Multiplexed (TDM) mode. The serial ports support TDM or multichannel operations. In multichannel mode, each data word of the serial bit stream occupies a separate channel— each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of 24 channels.

Transfer control block (TCB). A set of DMA parameter register values stored in memory that are downloaded by the DMA controller for chained DMA operations.

Tristate versus three-state. Analog Devices documentation uses the term "three-state" instead of "tristate" because Tristate™ is a trademarked term, which is owned by National Semiconductor.

Universal registers (Ureg). These are any processing element registers (data registers), any Data Address Generator (DAG) registers, any program sequencer registers, and any I/O processor registers.

Von Neumann architecture. This is the architecture used by most (non-DSP) microprocessors. This architecture uses a single address and data bus for memory access.

Waitstates. Waitstates are applied to each external memory access depending on the bank's external memory access mode (`EBxAM`). The External Bank Waitstate (`EBxWS`) field in the `WAIT` register sets the number of waitstates for each bank.

I INDEX