Features

- **Fully Autonomous DSP System**
- **16-bit Fixed-point OakDSPCore®**
- **24K x 16 of Uploadable Program RAM**
- **16K x 16 of Data RAM**
- **2K x 16 of X-RAM**
- **2K x 16 of Y-RAM**
- **X-RAM and Y-RAM Accessible within the Same Cycle**
- **JTAG Interface Available on AT75C220 and AT75C320**
- **On-chip Emulation Module**
- **Flexible Codec Interface**
- **Communication with External Processor through Dual-port Mailbox**

Description

The AT75C DSP subsystem is an autonomous DSP-based computation block that co-exists on-chip with other processors and functions. It is built around a 16-bit, fixed-point, industry-standard OakDSPCore. Additionally, the DSP subsystem embeds all elements necessary to run complex DSP algorithms independently without any external resources.

The self-contained subsystem contains the OakDSPCore itself, program memory, data memory, an on-chip emulation module and a flexible codec interface. These resources allow the subsystem to run complex DSP routines, such as V.34 modem emulation or state-of-the-art voice compression.

The codec interface permits connection of any external industry-standard codec device, allowing the DSP subsystem to handle directly external analog signals such as telephone line or handset signals.

Communication between the DSP subsystem and the on-chip ARM7TDMI™ core is achieved through a semaphore-operated dual-port mailbox.

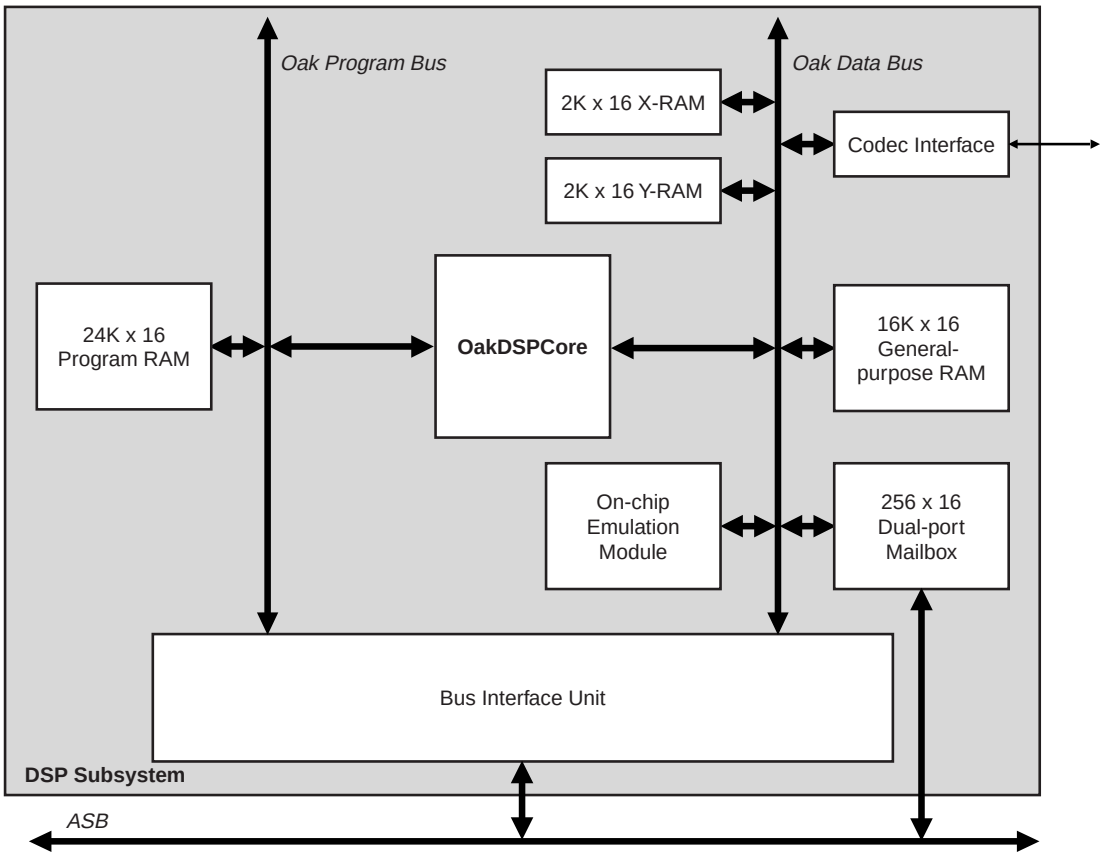
Smart Internet Appliance Processor (SIAP™)

AT75C DSP Subsystem

Rev. 1368C–INTAP–08/02

Architectural Overview

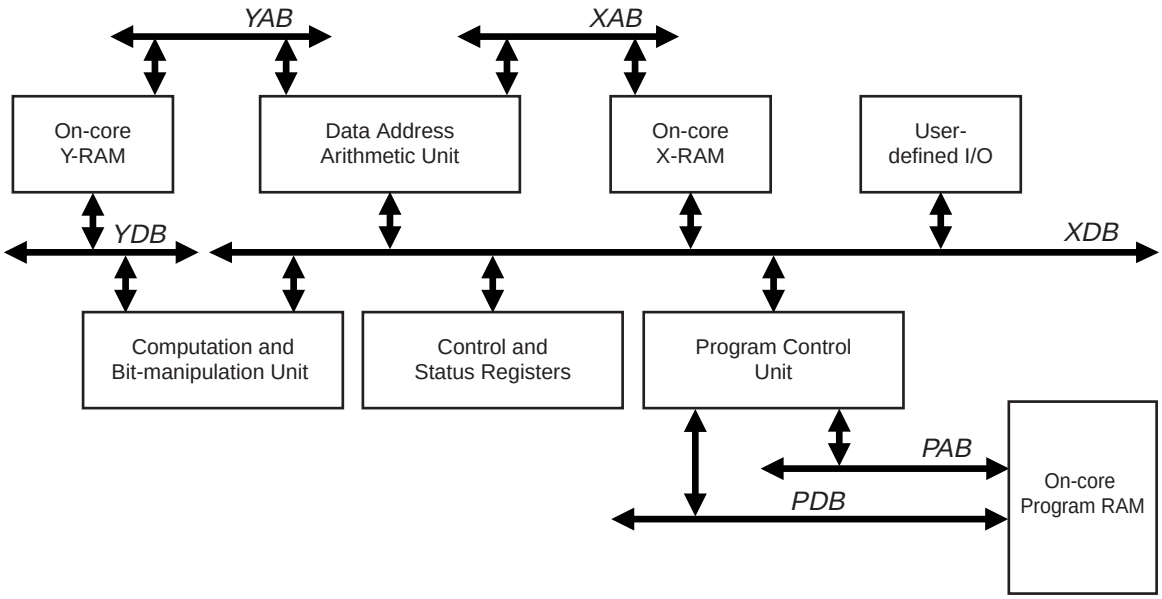
Figure 1. AT75C DSP Subsystem Block Diagram



Processing Unit

A block diagram of the OakDSPCore architecture is shown in Figure 2.

Figure 2. AT75C OakDSPCore Block Diagram



The major components of the OakDSPCore are:

- Computation and Bit-manipulation Unit (CBU)
  - Accumulators (A0, A1, B0, B1)
  - Saturation Logic
  - Arithmetic and Logical Unit (ALU)
  - Bit-field Operations (BFO)
  - Shift Value Register (SV)
  - Barrel Shifter
  - Exponent Logic (EXP)
  - Multiplier
  - Input Registers (X, Y)
  - Output Register (P)
  - Output Shifter
- Data Address Arithmetic Unit (DAAU)
  - DAAU Registers (R0...R5)
  - DAAU Configuration Registers (CFGI, CFGJ)
  - Software Stack Pointer (SP)
  - Base Register (RB)
  - Alternative bank of registers (R0B, R1B, R4B, CFGIB)
  - Minimal/Maximal Pointer Register (MIXP)
- Data buses (XDB, YDB, PDB)
- Address buses (XAB, YAB, PAB)

- Program Control Unit (PCU)
  Loop Counter (LC)
  Internal Repeat Counter (REPC)
- Memories
  On-core Data Memories (X-RAM, Y-RAM)
  Program Memory
- Control Registers
  Status Registers (ST0, ST1, ST2)
  Interrupt Context Switching Register (shadow and swap)
  Internal Configuration Register (ICR)
  Data Value Match Register (DVM)
- User-defined I/Os

Bus Architecture

Data Buses

Data is transferred via the X-data bus (XDB) and the program data bus (PDB), 16-bit bi-directional buses, and the Y-data bus (YDB), a 16-bit unidirectional bus. The XDB is the main data bus, where most of the data transfers occur. Data transfer between the Y-data memory (Y-RAM) and the multiplier (Y-register) occurs over the YDB when a multiply instruction uses two data memory locations simultaneously. Instruction word fetches take place in parallel over PDB.

The bus structure supports the following movements:

- Register to register
- Register to memory
- Memory to register
- Program to data
- Program to register
- Data to program

The bus structure can transfer up to two 16-bit words within one cycle.

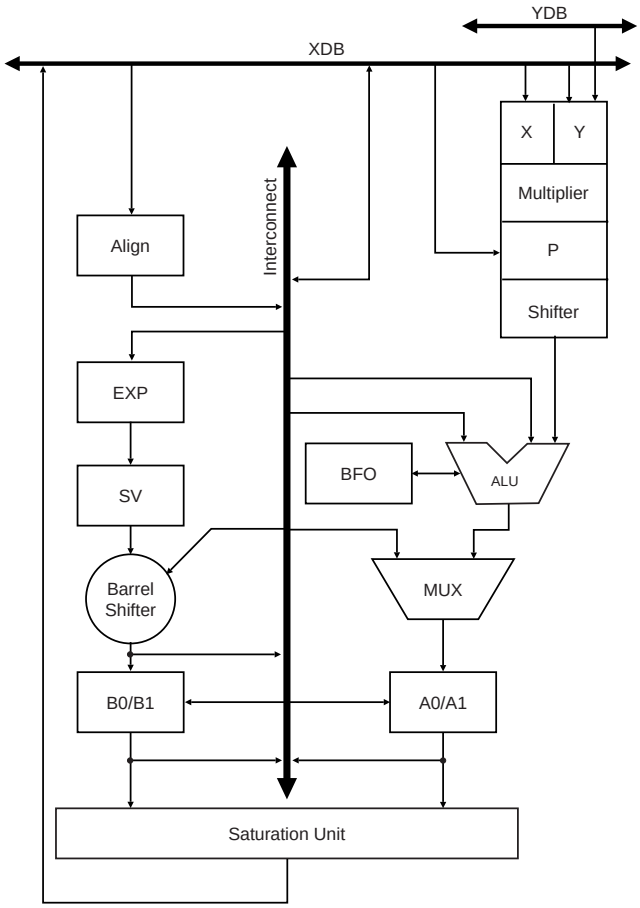
Address Buses

Addresses are specified for the on-core X- and Y-RAM on the 16-bit unidirectional X-address bus (XAB) and the 11-bit unidirectional Y-address bus (YAB). Program memory addresses are specified on the 16-bit unidirectional program address bus (PAB).

Computation and Bit-manipulation Unit

The computation and bit-manipulation unit (CBU) contains two main units, the computation unit (CU) and the bit-manipulation unit (BMU). The saturation unit is shared by the CU and the BMU units. A detailed block diagram of the computation and bit-manipulation unit is shown below in Figure 3.

Figure 3. Computation Unit and Bit-manipulation Unit Block Diagram



Computation Unit

The computation unit (CU) consists of the multiplier unit, the ALU and two 36-bit accumulator registers (A0 and A1), as shown in Figure 3.

Multiplier Unit

The multiplier unit consists of a 16-by-16 to 32-bit parallel 2's complement, single-cycle, non-pipelined multiplier, two 16-bit input registers (X and Y), a 32-bit output register (P), and a product output shifter. The multiplier performs signed-by-signed, signed-by-unsigned or unsigned-by-unsigned multiplication. Together with the ALU, the OakDSPCore can perform a single-cycle multiply-accumulate (MAC) instruction. The register P is updated only after a multiply instruction and not after a change in the input registers.

The X- and Y-registers are read or written via the XDB, and the Y-register is written via YDB, as 16-bit operands. The 16-bit Most Significant Portion (MSP) of the P register, PH, can be written by the XDB as an operand. This enables a single-cycle restore of PH during interrupt service routine. The complete 32-bit P register, sign-extended into 36 bits and shifted by the output shifter, can be used only by the ALU and can be moved only to the A0 and A1 accumulators.

The X- and Y-registers can be also used as general-purpose temporary data registers.

Product Output Shifter

The register P is sign-extended into 36 bits and then shifted. In addition to passing the data unshifted, the output shifter is capable of shifting the data from the register P into the ALU unit by one bit to the right or by one and two bits to the left. In right shift, the sign is extended, whereas in left shift, a zero is appended to the LSBs. Shift operation is controlled by two bits (PS) in the status register ST1. The shifter also includes alignment (a 16-bit right shift) for supporting double-precision multiplication.

Double-precision Multiplication

The OakDSPCore supports double-precision multiplication by several multiplication instructions and by alignment option of the register P. The register P can be aligned (shifting 16 bits to the right) before accumulating the partial multiplication results, in multiply-accumulate instructions (MAA and MAASU instructions). An example of different multiplication operations is in the multiplication of 32-bit by 16-bit fractional numbers, where two multiplication operations are needed: multiplying the 16-bit number with the lower or upper portion of a 32-bit (double-precision) number. The signed-by-unsigned operation is used to multiply or multiply-accumulate the 16-bit signed number with the lower, unsigned portion of the 32-bit number. The signed-by-signed operation is used to multiply the 16-bit signed number with the upper, signed portion of the 32-bit number. While the signed-by-signed operation is executed, it is recommended to accumulate the aligned (using MAA instruction) result of the previous signed-by-unsigned operation. For the multiplication of two double-precision (32-bit) numbers, the unsigned operation can be used. If this operation requires a 64-bit result, the unsigned-by-unsigned operation should be used. For details, on the various multiply instructions (MPY, MPYSU, MACUS, MACUU, MAA, MAASU, MSU and MPYI), refer to “Instruction Set” on page 31.

Ax-accumulators

Each Ax-accumulator is organized as two regular 16-bit registers (A0H, A0L, A1H and A1L) and a 4-bit extension nibble (A0E and A1E). The two portions of each accumulator can be accessed as any other 16-bit data register and can be used as 16-bit source or destination registers in all relevant instructions. The Ax-accumulators can serve as the source operand and the destination operand of the ALU, barrel shifter and exponent units. The extension nibbles of the A0 and A1 accumulators are the MSB’s part of status registers ST0 and ST1, respectively. The Ax-accumulators can be swapped with the Bx-accumulators in a single cycle. Saturation arithmetic is provided to selectively limit overflow from the high portion of an accumulator to the extension bits when performing a move instruction from one of the accumulators through the XDB, or when using the LIM instruction, which performs saturation on the 36-bit accumulator. For more details, refer to “Saturation” on page 12.

Registers AxH and AxL can also be used as general-purpose, temporary 16-bit data registers.

Extension Nibbles

Extension nibbles A0E and A1E offer protection against 32-bit overflows. When the result of an ALU output crosses bit 31, it sets the extension flag (E) in ST0, representing crosses of the MSB in AxH. Up to 15 overflows or underflows are possible using the extension nibble, after which the sign is lost beyond the MSB of the ALU output and/or beyond the MSB of the extension nibble, setting the overflow flag (V) in ST0, and also latching the Limit flag (L) in ST0. Refer to “Status Registers” on page 23 for more detail.

Sign Extension

Sign extension of the 36-bit Ax-accumulators is provided when the Ax or AxH is written with a smaller size operand. This occurs when these registers are written from XDB, from the ALU or from the exponent unit in certain CBU operations. Sign extension can be suppressed by specific instructions. For details, refer to “Instruction Set” on page 31.

Loading of Ax-accumulators

AxL is cleared while loading data into AxH, and AxH is cleared while loading AxL. Loading a full 32-bit value is accomplished via the ADDL or ADDH instructions (refer to

"Instruction Set" on page 31). The full 36-bit accumulators can also be loaded using the shift instructions or with another 36-bit accumulator in a single cycle using the SWAP instruction. For details, refer to the sections "Swapping the Accumulators" on page 13 and "Instruction Set" on page 31.

Arithmetic and Logic Unit

The Arithmetic and Logic Unit (ALU) performs all arithmetic and logical operations on data operands. It is a 36-bit, single-cycle, non-pipelined ALU.

The ALU receives one operand from Ax (x = 0,1), and another operand from either the output shifter of the multiplier, the XDB (through bus alignment logic), or from Ax. The source operands can be 8, 16 or 36 bits. Operations between the two Ax-accumulators are also possible. The source and destination Ax-accumulator of an ALU instruction is always the same. The XDB input is used for transferring one of the register's contents, an immediate operand or the contents of a data memory location addressed in direct memory addressing mode, indirect addressing mode, index addressing mode or pointed to by the stack pointer, as a source operand. The ALU results are stored in one of the Ax-accumulators or transferred through the XDB to one of the registers or to a data memory location. The latter is used for addition, subtraction and compare operations between a 16-bit immediate operand and a data memory location or one of the registers, without effecting the accumulators, in two cycles. The add and subtract are part of the read-modify-write instructions. Refer to ADDV, SUBV, CMPV instructions in "Instruction Set" on page 31. A bit-field operation (BFO) unit is attached to the ALU and described in detail in "Bit-field Operations".

The ALU can perform positive or negative accumulate, add, subtract, compare, logical and several other operations, most of them in a single cycle. It uses 2's complement arithmetic.

Unless otherwise specified, in all operations between an 8-bit or 16-bit operand and a 36-bit Ax, the 16-bit operand will be regarded as the LSP of a 36-bit operand with sign extension for arithmetic operations and zero extension for logical operations. The ADDH, SUBH, ADDL and SUBL instructions are used when this convention is not adequate in arithmetic operations. For details, refer to these instructions in "Instruction Set" on page 31.

The flags are affected as a result of the ALU output, as well as a result of the BFO or the barrel shifter operation. In most of the instructions where the ALU result is transferred to one of the Ax-accumulators, the flags represent the Ax-accumulator status.

Rounding

Rounding (by adding 0x8000 to the LSP of the accumulator) can be performed by special instructions, in a single cycle or in parallel to other operations. Refer to MOVR and MODA instructions in "Instruction Set" on page 31.

Division Step

A single-cycle division step is supported. For details, refer to the DIVS instruction in "Instruction Set" on page 31.

Logical Operations

The logical operations performed by the ALU are AND, OR, and XOR. All logical operations are 36 bits wide. 16-bit operands are zero extended when used in logical operations. The source and destination Ax-accumulator of these instructions is always the same. Operations between the two Ax-accumulators are also possible. For details, refer to AND, OR and XOR instructions in "Instruction Set" on page 31.

Other logical operations are set, reset, change and test, executed on one of the registers or on data memory contents. Refer to "Bit-field Operations" on page 11.

Maximum/Minimum Operations

A single-cycle maximum/minimum operation is available, with pointer latching and modification. One of the Ax-accumulators, defined in the instruction, holds the maximal value in a MAX instruction, or the minimal value in a MIN instruction. In one cycle the two accumulators are compared and when a new maximal or minimal number is found, this value is copied to the above-defined accumulator. In the same instruction, the register R0 can be used, for example, as a buffer pointer. This register can be post-modified according to the specified mode in the instruction. When the new maximal or minimal number is found, the R0 pointer is also latched into the 16-bit dedicated minimum/maximum pointer latching (MIXP) register – one of the DAAU registers. The maximum operation can also be performed directly on a data memory location pointed to by the register R0 (MAXD instruction), saving the maximal number in the defined Ax-accumulator and latching the R0 value into MIXP in a single cycle. For more details, refer to MAX, MAXD and MIN instructions in “Instruction Set” on page 31. For more details on registers R0 and MIXP, refer to “Data Address Arithmetic Unit (DAAU)” on page 14.

When finding the maximum/minimum value, a few buffer elements can have the same value. The accumulator will save the same value; the latched pointer, however, depends on the condition used in the instruction. In finding the maximum value, the pointer of the first element or the last element will be latched, using greater than (>), or greater-than-or-equal to (≥) conditions, respectively. In finding the minimum value, the pointer of the first element or the last element will be latched, using less than (<), or less- than-or-equal to (≤) conditions, respectively. All these cases are supported by the MAX, MAXD and MIN instructions.

Bit-manipulation Unit

The bit-manipulation unit (BMU) consists of a full 36-bit barrel shifter, an exponent unit (EXP), a bit-field operation unit (BFO), two 36-bit accumulator registers (B0 and B1), and the shift value (SV) register. Refer to Figure 3 on page 6.

Barrel Shifter

The barrel shifter performs arithmetic shift, logical shift and rotate operations. It is a 36-bit left and right, single-cycle, non-pipelined barrel shifter.

The barrel shifter receives the source operand from either one of the four accumulators (A0, A1, B0, B1) or from the XDB (through bus alignment logic). The XDB input is used for transferring the contents of one of the registers or a data memory location, addressed in direct memory addressing mode or in indirect addressing mode. The source operands may be 16 or 36 bits. The destination of the shifted value is one of the four accumulators. The amount of shifts is determined by a constant embedded in the instruction opcode or by a value in the SV register.

The flags are effected as a result of the barrel shifter output, as well as a result of the ALU and BFO outputs. When this result is transferred into one of the accumulators, the flags represent the accumulator status.

Shifting Operations

A few options of shifting are available using the barrel shifter, all of them performed in a single cycle. Each of the four accumulators can be shifted according to a 6-bit signed number representing +31, -32 shifts (shift left is a positive number, while shift right is a negative number) embedded in the instruction opcode, into each of the four accumulators. The accumulators can also be shifted conditionally, according to the SV register content. In this case the accumulators can be shifted by +36, -36. This supports calculating the amount of shifts at run-time as, for example, in normalization operations. Refer to “Normalization” on page 11.

The source and the destination accumulators can be the same or different. If the accumulators are different, the source accumulator is unaffected. For details, refer to SHFC and SHFI instructions in “Instruction Set” on page 31.

In addition to the conditional shifting operations performed when the accumulators are shifted according to the SV register (by +36, -36), the shift and rotate operations included in the MODA and MODB instructions are also performed conditionally. MODA and MODB include:

- 1-bit right and left arithmetic shift and rotate
- 4-bit right and left arithmetic shift

For details, refer to MODA and MODB instructions in “Instruction Set” on page 31.

Arithmetic left shift and logical left shift perform the same operation, filling the LSBs with zeros. During an arithmetic right shift, the MSBs are sign extended and during a logical right shift, the MSBs are filled with zeros. Whether the shift mode is arithmetic or logic is determined according to the Status mode bit (S) in the register ST2. It affects all shift instructions.

Figure 4. Arithmetic Shift Right

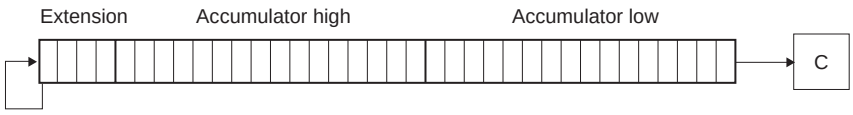


Figure 5. Logical Shift Right

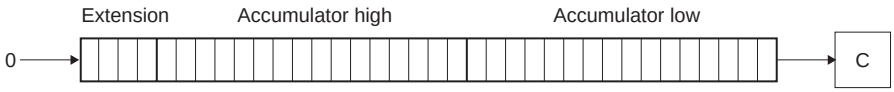


Figure 6. Logical Shift Left

Move and Shift Operations

The four accumulators can be loaded by a shifted value, according to the SV register content, in a single cycle. The accumulators can be loaded from one of the registers or from a data memory location, addressed in direct addressing mode or indirect addressing mode, according to the SV shift value. The shifting capability is +36, -36, (shift left is a positive number, while shift right is a negative number). The accumulators can also be loaded by one of the DAAU registers (Rn registers), shifted according to a constant embedded in the instruction opcode. The shifting capability in this case is 15 to -16. Whether arithmetic shift mode or logic shift mode is selected is determined by the Status mode bit (S) in the register ST2. Refer to MOVS and MOVSI instructions in “Instruction Set” on page 31.

The register content or a data memory location content can be moved and shifted according to the SV shift value into one of the four accumulators in a single cycle. One of the four accumulators can also be loaded by a shifted value of the DAAU registers (Rn registers), according to a constant embedded in the instruction opcode, also in a single cycle. The shifting capability in this case is 16 to -15. Refer to “Instruction Set” on page 31 for MOVS and MOVSI instructions.

Exponent Unit

The exponent unit (EXP) performs exponent evaluation of one of the four accumulators, of a data memory location or of one of the registers. The result of this operation is a signed 6-bit value sign-extended into 16 bits and transferred into the shift value register (SV). Optionally, it can also be transferred, sign-extended into 36 bits, into one of the Ax-accumulators. The source operand is unaffected by this calculation. The source

operand can be 36 bits when it is one of the accumulators or 16 bits when it is a data memory location or one of the registers.

The algorithm for determining the exponent result for a 36-bit number is as follows: Let N be the number of the sign bits (i.e., the number of MSBs equal to bit 36) found in the evaluated number. This means that the exponent is evaluated with respect to bit 32. For a 16-bit operand, the 16 bits are regarded as bits 16 31, sign-extended into bits 32 to 35 and then treated as a 36-bit operand. Therefore, in this case, the exponent result is always greater than or equal to zero. For examples, refer to Table 1 on page 11.

A negative result represents a number for which the extension bits are not identical. The value of this negative result stands for how many right shifts should be executed in order to normalize the number, i.e., bit 31 (representing the sign) and bit 30 (representing part of the magnitude) will be different. A positive result represents an un-normalized number for which at least the four extension bits and the sign bits are the same. When evaluating the exponent value of one of the accumulators, the positive number is the amount of left shifts that should be executed in order to normalize the source operand. An exponent result equal to zero represents a normalized number.

Examples including an application in a normalization operation can be found in Table 1 below.

Table 1. Normalization Operation Examples

Evaluated Number [35:...]	N	Exponent Result N-5	Normalized Number [35:...]
0000 0000101...	8	3 (shift left by 3)	0000 0101...
1100 10101...	2	-3 (shift right by 3)	1111...
0000 0111000...	5	0 (no shift)	0000 0111000...

Full normalization can be achieved in two cycles using the EXP instruction followed by a shift instruction. For more details, refer to the section below on normalization, or to the EXP, SHFC and MOVS instructions in “Instruction Set” on page 31.

The exponent unit can also be used in floating-point calculations, where it is useful to transfer the exponent result into both the SV register and one of the Ax-accumulators.

Normalization

Two techniques for normalization are provided. Using the first technique, normalization can be done by two cycles using two instructions. The first instruction evaluates the exponent value of one of the registers including the accumulators, or the value of a data memory location. The second instruction is shifting the evaluated number according to the exponent result stored at SV register. The second technique is using a normalization step (NORM instruction). For details, refer to “Instruction Set” on page 31.

Bit-field Operations

The bit-field operation unit (BFO) is used for set, reset, change or test a set of up to 16 bits within a data memory location or within one of the registers. The data memory location can be addressed using a direct memory or an indirect address. The SET, RST and CHNG instructions are read-modify-write and require two cycles and two words. The 16-bit immediate mask value is embedded in the instruction opcode. Various testing instructions can be used. Testing for zeros (TST0) or for ones (TST1), of up to 16 bits in a single cycle, can be achieved when the mask is in one of the AxL or in two cycles when the mask value is embedded in the instruction opcode.

Testing instruction (TSTB) for a specific bit, 1 out of 16, in a data memory location or in one of the registers in a single cycle, is also available. For details, refer to SET, RST, CHNG, TST0, TST1, TSTB instructions in the “Instruction Set” on page 31.

The bit-field operation unit (BFO) is attached to the ALU. Flags are affected as a result of the bit-field operations as well as a result of the ALU and the barrel shifter operation.

Bx-accumulators

Each Bx-accumulator is organized as two regular 16-bit registers (B0H, B0L, B1H and B1L) and a 4-bit extension nibble. The two portions of each accumulator can be accessed as 16-bit data registers using the XDB bus and can be used as 16-bit source or destination data registers in relevant instructions. The Bx-accumulators can be swapped with the Ax-accumulators in a single cycle. Saturation arithmetic is provided to selectively limit overflow from the high portion of an accumulator to the extension bits when performing a move instruction from one of the accumulators through the XDB. For more details, refer to “Saturation” on page 12.

Each of the Bx-accumulators can be a source operand of the exponent unit and can be a source operand or a destination operand of the barrel shifter.

Extension Nibbles

Extension nibbles of B0 and B1 offer protection against 32-bit overflows. When the result of the barrel shifter crosses bit 31, it sets the extension flag (E) in ST0, representing crosses of the MSB at BxH. When the sign is lost beyond the MSB of the barrel shifter and/or beyond the MSB of the extension nibble, the overflow flag (V) in ST0 is set and latched in the Limit flag (L) in ST0. Refer to “Status Registers” on page 23 for more details. The extension bits can be accessed with the aid of a single-cycle shift instruction or by swapping to the Ax-accumulator.

Sign Extension

Sign extension of the 36-bit Bx-accumulators is provided when the Bx or BxH is written with a smaller size operand. This occurs when these registers are written from XDB or from the barrel shifter in shift operations.

Loading Bx-accumulators

BxL is cleared while loading data into BxH and BxH is cleared while loading BxL. The full 36-bit accumulator can be loaded in a single cycle, using the shift instructions or by another 36-bit accumulator, using the SWAP instruction (refer to “Swapping the Accumulators” on page 13 and “Instruction Set” on page 31).

Shift Value Register

The shift value (SV) register is a 16-bit register used for shifting operations and exponent calculation. In shift operations it determines the amount of shifts and therefore enables calculating the amount of shifts at run-time. The exponent result is transferred to the SV register. This register can be used for full normalization by serving as the destination of the exponent calculation and as the control for the shift (see “Normalization” on page 11 and “Instruction Set” on page 31).

The SV register can also be used as a general-purpose temporary data register.

Saturation

Saturation arithmetic is provided to selectively limit overflow from the high portion of an accumulator to the extension bits. Saturation is performed when moving from the high portion or low portion of one of the accumulators through the XDB, or when using the LIM instruction, which performs saturation on the 36-bit accumulator. The saturation logic will substitute a “limited” data value having maximum magnitude and the same sign as the source accumulator.

In case saturation occurs when performing a move instruction (MOV or PUSH) from one of the accumulators (AxH, AxL, BxL or BxH) through the XDB, the value of the accumulator is not changed. Only the value transferred over the XDB is limited to a full-scale, 16-bit positive (0x7FFF for AxH or BxH; 0xFFFF for AxL or BxL) or negative (0x8000 for AxH or BxH; 0x0000 for AxL or BxL) value. Limiting will be correctly performed even if the transfer to the XDB does not immediately follow the accumulator overflow. When an accumulator is swapped by the SWAP instruction, limitation will be correctly performed

when the value is transferred to the XDB. The saturation in move instructions can be disabled by the SAT bit in the register ST0. When limiting occurs, the L flag in ST0 is set. Refer to “Status Registers” on page 23 for more details.

The LIM instruction activates saturation on a 36-bit Ax-accumulator. When there is an overflow from the high portion of an Ax-accumulator to the extension bits and a LIM instruction is executed, the accumulator is limited to a full-scale, 32-bit positive (0 X 7FFFFFFF) or negative (0 X 8000000) value. Limiting will be correctly performed when the value is operated on by the LIM instruction. The LIM instruction can use the same accumulator for both source and destination or it can use one source Ax-accumulator, which will not change, and transfer the limited result into the other Ax-accumulator. For more details, refer to the LIM instruction in “Instruction Set” on page 31. When limiting occurs, the L flag in ST0 is set. The SAT bit in ST0 has no effect on this instruction. Refer to “Status Registers” on page 23 for more details.

Swapping the Accumulators

The Ax-accumulators can be swapped with the Bx-accumulators in a single cycle. It is possible to swap two 36-bit registers or all four 36-bits registers. Swapping is also enabled between a specific Ax-accumulator into one of the Bx-accumulators, and in the same cycle, from that Bx-accumulator into the other Ax-accumulator. Similarly, swapping is enabled between a specific Bx-accumulator into one of the Ax-accumulators, and in the same cycle, from that Ax-accumulator into the other Bx-accumulator. For a summary of the 14 swap options and other details, refer to the SWAP instruction in “Instruction Set” on page 31.

Note that during interrupt context switching, the A1 and B1 accumulators are automatically swapped. For more details, refer to “Interrupt Context Switching” on page 27.

Data Address Arithmetic Unit (DAAU)

The DAAU performs all address storage and effective address calculations necessary to address data operands in data and program memories and supports the software stack pointer. In addition, it supports latching of the modified register in maximum/minimum operations (see “Maximum/Minimum Operations” on page 9) and loop counter operations in conjunction with the MODR instruction (see “Instruction Set” on page 31) and the R flag (see “Status Registers” on page 23). This unit operates in parallel with other core resources to minimize address generation overhead. The DAAU performs two types of arithmetics: linear and modulo. The DAAU contains six 16-bit address registers (R0, R3 and R4, R5, also referred to as Rn) for indirect addressing, two 16-bit configuration registers (CFGI and CFGJ) for modulo and increment/decrement step control and a base register (RB) for supporting index addressing. In addition, it contains a 16-bit stack pointer register (SP), alternative bank registers (R0B, R1B, R4B, CFGIB) supported by an individual bank exchange and a 16-bit minimum-maximum pointer latching register (MIXP, see “Maximum/Minimum Operations” on page 9). The Rn and configuration registers are divided into two groups for simultaneous addressing over XAB and YAB (or PAB): R0, R3 with CFGI, and R4, R5 with CFGJ. Registers from both groups, in addition to RB and SP, can be used for both XAB and YAB (or PAB) for instructions that use only one address register. In addition, in these instructions the X-RAM and Y-RAM can be viewed as a single continuous data memory space.

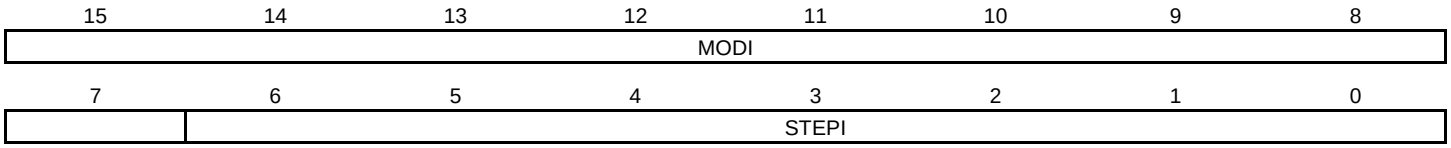
The R0, R1, R2, R3, R4, R5, CFGI, CFGJ, SP, RB and MIXP registers may be read from or written to by the XDB as 16-bit operands, and thus can serve as general-purpose registers.

Address Modification

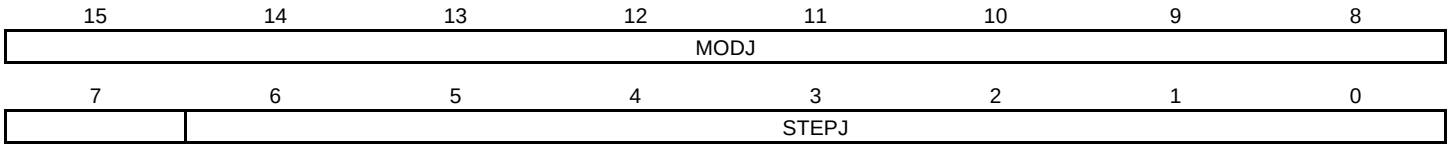
The DAAU can generate two 16-bit addresses every instruction cycle, which can be post-modified by two modifiers: linear and modulo. The address modifiers allow the creation of data structures in memory for circular buffers, delay lines, FIFOs, another pointer to the software stack, etc. The Rn registers can also be used, in addition to the block-repeat nesting, as loop counters in conjunction with the MODR instruction (see “Instruction Set” on page 31) and the R flag of ST0 (see “Status Registers” on page 23). Address modification is performed using 16-bit (modulo 65,536) 2’s complement linear arithmetic. The range of values of the registers can be considered as signed (from -32,768 to +32,767) or unsigned (from 0 to +65,535). This is also true for the data space memory map. Index addressing capability is also available. For details, see “Index Modifier”.

Configuration Registers

CFGI



CFGJ



Linear (Step) Modifier

During one instruction cycle, one or two (from different groups) of the address register, Rn, can be post-incremented or post-decremented by 1 or added with a 2's complement 7-bit step (from -64 to +63). The selection of linear modifier type (one out of four) is included in the relevant instructions (see "Conventions" on page 34). Step values STEPI and STEPJ are stored as the seven LSBs of the configuration registers, CFGI and CFGJ, respectively.

Modulo Modifier

The two modulo arithmetic units can update one or two address registers from different groups within one instruction cycle. They are capable of performing modulo calculations of up to 2^9. Each register can be set independently to be affected or unaffected by the modulo calculation using the six Mn status bits in the ST2 register. Modulo setting values MODI and MODJ are stored in nine MSBs of configuration registers CFGI and CFGJ, respectively.

For proper modulo calculation, the following constraints must be satisfied (M = modulo factor; q = STEPx, +1 or -1).

1. Rn should be initiated with a number whose p LSBs are less than M, where p is the minimal integer that satisfies $2^p \leq M$.
2. The constraints when the modulo M to the power of 2 (full modulo operation):

   a. The lower boundary (base address) must have zeros in at least the k LSBs, where k is the minimal integer that satisfies $2^k \geq M-1$.

   b. MODx (x denotes I or J) must be loaded with M - |q|.

   c. $M \geq q$
3. The constraints when the modulo M is not a power of 2:

   a. The lower boundary (base address) must have zeros in at least the k LSBs, where k is the minimal integer that satisfies $2^k \geq M - |q|$.

   b. MODx (x denotes I or J) must be loaded with M - |q|.

   c. M must be an integer multiple of q (this is always true for q = ±1).

   d. Rn should be initiated with a number that contains an integer multiple of |q| or zeros in its k LSBs.

Note: |q| denotes the absolute value of q.

The modulo modifier operation, a post-modification of the Rn register, is defined as follows:

Rn <- 0 in k LSB; if Rn is equal to MODx in k LSB and $q \geq 0$,

Rn <- MODx in k LSB; if Rn is equal to 0 in k LSB and
q < 0,

Rn (k LSBs) <- Rn+q (k LSBs); Otherwise

When M = |q| (i.e., MODx = 0), modulo operation is:
Rn <- Rn.

Notes: 1. R0 ≥ R3 can only work with STEPI and MODI, while R4 • R5 can work only with STEPJ and MODJ.
2. The modulo operation can work for modulo values greater than 512 when the M - |STEPx| ≥ 511 and constraints 3a, 3b, 3c and 3d are met.

Examples:

1. M = 7 with STEPx = 1 (or +1 selected in instruction), MODx = 7 - 1 = 6, Rn = 0x0010 (hexa). The sequence of Rn values will be:
0x0010,0x0011,0x0012,0x0013,0x0014,0x0015,0x0016,0x0010,0x0011,...
2. M = 8 with STEPx = 2, MODx = 8 - 2 = 6, Rn = 0x0010. The sequence of Rn values will be: 0x0010,0x0012,0x0014,0x0016,0x0010,0x0012,...
3. M = 9 with STEPx = -3, MODx = 9 - |-3| = 6, Rn = 0x0016. The sequence of Rn values will be:
0x0016,0x0013,0x0010,0x0016,0x0013,...
4. M = 8 with STEPx = 3, (2^3 = 8 - full modulo support), MODx = 8 - 3 = 5, Rn = 0x0010.
The sequence of Rn values will be:
0x0010,0x0013,0x0016,0x0011,0x0014,0x0017,0x0012,0x0015,0x0010,0x0013
,...

Index Modifier

The OakDSPCore has short and long index addressing modes. The base register is RB, one of the DAAU registers. In the short index addressing mode, the base register, together with a 7-bit signed short immediate value (-64 to +63) embedded in the instruction opcode, is used to point to a data memory location in a single cycle. In the long index addressing mode, the base register, together with a signed 16-bit offset, given as the second word of the instruction, is used to access the memory in two cycles. Unlike the linear and modulo addressing modes, in both index addressing modes, address pre-modification is performed prior to accessing the memory. The base register is unaffected. Indexed addition, subtraction, compare, AND, OR, XOR and move from/into the pointed data memory location can be performed in either one or two cycles, using the short or long mode, respectively.

The base register can be used as an array pointer or in conjunction with the stack pointer (SP) register. When the stack is used for transferring routine parameters, initializing the base register by the SP value enables quick access to routine parameters transferred using the stack. The index addressing mode is useful for supporting C-compiler.

The RB register is part of the global register set and can be used as a general-purpose register.

Software Stack

The OakDSPCore contains a software stack, pointed to by a dedicated 16-bit register, the stack pointer (SP). The SP contains the address of the top value in the stack; therefore, it points to the last value pushed onto the stack. The stack is filled from high memory address to low memory address. A POP instruction performs a post-increment; a PUSH instruction performs a pre-decrement. The Program Counter (PC) is automatically pushed to the stack whenever a subroutine call or an interrupt occurs and popped back on return from subroutine or interrupt. Other values can be pushed and popped

using the PUSH and POP instructions. The top of stack can be read without affecting SP using a dedicated MOV instruction.

The software stack can reside anywhere in the data space (X-RAM and Y-RAM) and can be accessed by any other pointer (R0 to R5 and RB).

The software stack is useful for supporting the C-compiler. The stack can be used for transferring routine parameters (e.g., C-automatic variables). Thus, after initializing the base register (RB) by the SP value, the routine parameters can be referenced by the index mode with the MOV, ADD, SUB, CMP, AND, OR and XOR instructions. Another support for transfer of routine parameters is the RETS instruction, which returns from a subroutine and updates the SP by a short immediate value.

The SP register is part of the global register set. Refer to “Programming Model and Registers” on page 23.

Alternative Bank of Registers

The DAAU contains an alternative bank of four registers: R0B, R1B, R4B, CFGIB. For each of the R0/R0B, R1/R1B, R4/R4B or CFGI/CFGIB, only one register is accessible at a time. The selection between the current or the alternative register is controlled by a special BANKE instruction, which exchanges (swaps) the contents between the current register with the alternative bank register.

The bank exchanging is selected individually, meaning that the BANKE instruction includes a list of registers to be exchanged in a single cycle. For the four registers there are 15 different options. Refer to BANKE instruction in “Instruction Set” on page 31.

The individual selectivity of the bank registers contributes to flexibility of the bank registers. The user can decide where each of the alternative registers will be utilized in interrupts, routines, etc.

Program Control Unit (PCU)

The Program Control Unit (PCU) performs instruction fetch, instruction decoding, exception handling, hardware loop control, wait state support and on-chip emulation support. In addition, it controls the internal program memory protection. Refer to “Program Memory”.

The PCU contains the Repeat/Block-repeat unit and two 16-bit directly accessible registers: the Program Counter (PC) and the Loop Counter (LC).

The PCU selects and/or calculates the next address from several possible sources:

- the incremented PC in sequential program flow
- jump address in branch or call operations
- short PC-relative address of seven bits in relative branch or call operations
- start address and exit address of hardware loops
- interrupts vector handling
- user write to PC or the top value on the stack, pointed to by the SP register upon returning from subroutines and interrupts

The PCU also writes the PC to the top of stack in subroutines and interrupts.

The PC always contains the address of the next instruction.

For more information on the pipeline method, refer to “Pipeline Method” on page 140.

Repeat and Block-repeat Unit

The Repeat/Block-repeat unit performs the hardware-loop calculations and controls execution without overhead (other than the one-time execution of set up instructions REP or BKREP for initialization of repeat or block-repeat mechanism, respectively). Four nested levels of block-repeat can be performed and the REP instruction can be performed inside each one of the these levels.

The number of repetitions can be a fixed value embedded in the instruction code or a value transferred from one of the processor's 16-bit registers. This option supports calculating the number of repetitions in run-time.

For the repeat operation, the unit contains an internal 16-bit repeat counter (REPC) for repeating a single-word instruction from 1 to 65536 repetitions. REPC counter is readable by the programmer.

In block-repeat operation, the last and first addresses of a loop are stored in 16-bit dedicated registers. A 16-bit dedicated counter, LC, counts the number of loop repetitions (1 - 65536). In case of nested block-repeats, it saves these values in internal registers. The LC of each level can be accessed by the user; the start-address and end-address registers and the internal shadow registers cannot be accessed as registers by the programmer. An indication of the block-repeat nesting level is a read-only block-repeat nesting counter (BC2, BC1, BC0) in the internal configuration register (ICR). See also “Internal Configuration Register” on page 29. The 16-bit block-repeat loop counter (LC) is one of the global registers. The LC register can be used as an index inside the block-repeat loop or for determining the value of the block-repeat counter when a jump out of the block-repeat loop occurs.

The single instruction repeat can reside in each of the block-repeat levels. Both the repeat and the block-repeat mechanisms are interruptible. For details, on specific limitations, refer to REP and BKREP instructions in “Instruction Set” on page 31.

A BREAK instruction can be used for stopping each of the four nested levels of a block-repeat. Refer to the BREAK instruction in “Instruction Set” on page 31.

The in-loop (LP) bit in the ICR is set when a block-repeat is executed and reset upon normal completion of the outer block-repeat loop. When the user resets this bit, it stops the execution of all four levels of block-repeat. For more details on the LP bit, refer to “Internal Configuration Register” on page 29. If the LP bit is cleared in the current block-repeat loop, the processor is no longer in any of the block loop levels (BC2, BC1, BC0 bits in ICR register are cleared). Therefore, when the last address is reached there are no jumps to the first address of the loop, the counter is not decremented and the processor continues to the sequential instruction. An exception is when LP is cleared at one of the last three addresses of the block-repeat. In these cases the effect of clearing LP takes place only in the next loop. An instruction that reads ICR and starts at last address of the outer block-repeat loop results in the LP bit equal to zero when the last repetition of this outer loop is reached.

The LC register may also serve as a 16-bit general-purpose register for temporary storage.

Memory Spaces and Organization

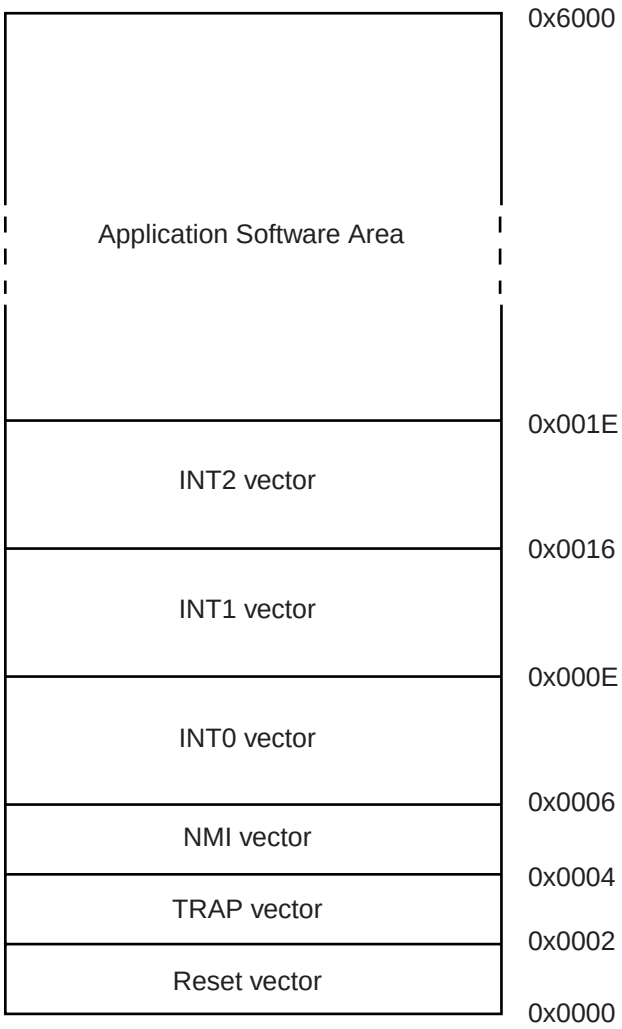
Two independent memory spaces are available: the data space (X-RAM and Y-RAM) and the program space.

Program Memory

Addresses 0x0000 to 0x0016 are used as interrupt vectors for Reset, TRAP (software interrupt with option for hardware activation), NMI (non-maskable interrupt) and three maskable interrupts (INT0, INT1, INT2). The RESET, TRAP and NMI vectors have been separated by two locations so that branch instructions can be accommodated in those locations if desired. The maskable interrupts have been separated by eight locations so that branch instructions, or small and fast interrupt service routines, can be accommodated in those locations.

The program memory addresses are generated by the PCU.

Figure 7. Program Memory Diagram



Note that in the AT75C the interrupts have been affected as shown in Table 2.

Table 2. Affectation of the Interrupt Request Lines

Interrupt	Affectation
TRAP	Used in conjunction with the OCEM for debug purposes
NMI	Not used
INTerrupt 0	Indicates that the dual-port mailbox requires service
INTerrupt 1	Indicates that the codec interface requires service
INTerrupt 2	Not used

Data Memory

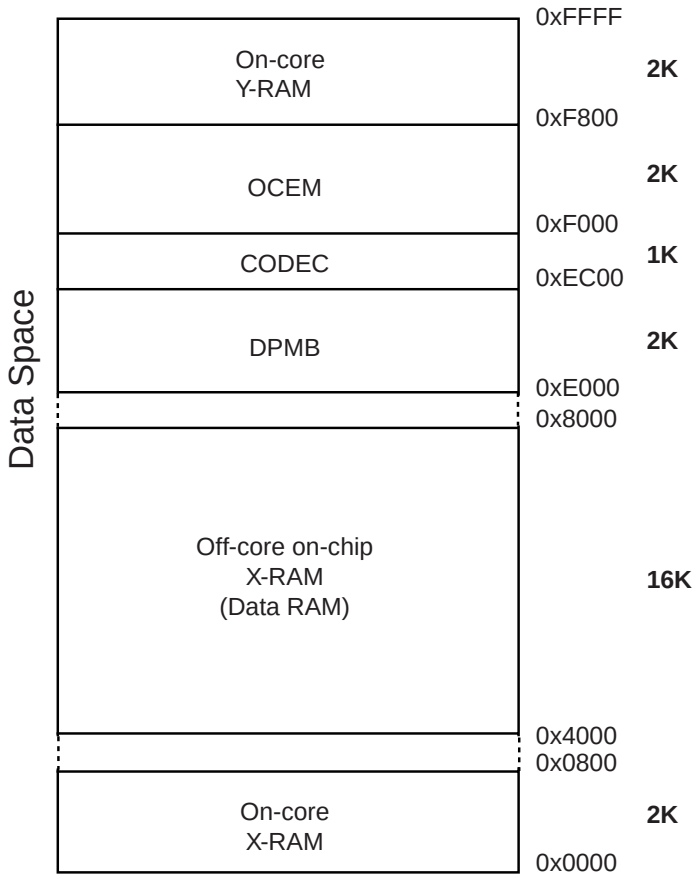
The data space is divided as follows:

- a Y-data space for the on-core Y-RAM (2K x 16)
- an X-data space for the on-core X-RAM (2K x 16)
- an additional X-data space for an on-chip but off-core X-RAM (16K x 16)
- the dual-port mailbox (refer to “Dual-port Mailbox” on page 152 and following sections)
- memory-mapped I/Os for peripherals connection

The on-core Y-RAM space and the on-core X-RAM space are mapped to allow a continuous data space. The data space partition allows expansion of the X-RAM, which has been grown off-core with an additional 16K x 16 block. This off-core block is used as general-purpose working memory and can be accessed with no wait state cycles. The dual-port mailbox is also seen in the X-data space.

Memory-mapped I/Os are used to connect to on-chip peripherals (OCEM, codec interface). They are seen by the OakDSPCore in the X-data space. Refer to Figure 8.

Figure 8. Data Memory Diagram



Memory Addressing Modes

There are five data memory addressing modes.

Short Direct Addressing Mode

Eight bits embedded in the instruction opcode as LSBs plus eight bits from status register ST1 as MSB (see “Status Registers” on page 23) compose the 16-bit address to the data memory. The pages are thus of 256 words each. For example, page 0 corresponds to addresses 0 to 255 in X-RAM, page 1 from 250 to 511 in X-RAM and page 255 from -256 to -1 in Y-RAM. Any memory location in the 64K-word data space can be directly accessed in a single cycle.

Long Direct Addressing Mode

Sixteen bits embedded in the instruction opcode as the second word of the instruction are used as the 16-bit address of the data memory. Any memory location in the 64K-word data space can be directly accessed in two cycles.

Indirect Addressing Mode

The Rn registers of the DAAU are used as 16-bit addresses for indirect addressing the X-RAM and the Y-RAM. Some instructions use two registers, simultaneously addressing a memory location in X-RAM and another in Y-RAM, both addressed in indirect addressing mode.

Short Index Addressing Mode

The base register RB plus an index value (offset7, a 7-bit immediate value embedded in the instruction opcode) are used for index-based indirect addressing the X-RAM or Y-RAM. The index value can be from -64 to +63. The actual address is RB + offset7, but leaving the contents of RB unaffected. For details, refer to “Index Modifier” on page 16.

Long Index Addressing Mode

The base register RB plus a 16-bit immediate index value (embedded in the second instruction opcode word) are used for index-based indirect addressing of the X-RAM or Y-RAM. The index value can vary between -32768 to +32767. The contents of RB remain unaffected. For details, refer to “Index Modifier” on page 16.

The software stack located in the data memory is addressed using the stack pointer (SP) register.

Program memory is addressed by:

- Indirect Addressing Mode
  The Rn registers of the DAAU and the accumulator can be used for addressing the program memory in specific instructions.
- Special Relative Addressing Mode
  Special Branch Relative (BRR) and Call Relative (CALLR) instructions support jumping relative to the PC (from PC - 63 to PC + 64).

Programming Model and Registers

Most of the OakDSPCore's visible registers are arranged as a global register set of 34 registers that can be accessed by most data moves and core operations.

The registers are listed below, organized according to the units' partition. Additional details on each register can be found in the description of each unit and in the following paragraphs.

Status Registers

Three status registers are available to hold the flags, status bits, control bits, user I/O bits and paging bits for direct addressing. The contents of each register and their field definitions are described below.

Status Register 0

15	14	13	12	11	10	9	8
A0E				Z	M	N	V

7	6	5	4	3	2	1	0
C	E	L	R	IM1	IM0	IE	SAT

- **Z: Zero**

Set if the ALU/BFO/Barrel Shifter output used at the last instruction equals zero; cleared otherwise. This flag is also used to indicate the result of the test bit/s instructions (TST0, TST1, TSTB).

The zero flag is cleared during processor reset.

The zero flag can be modified by writing to ST0.

- **M: Minus**

Set if ALU/BFO/Barrel Shifter output used at the last instruction is a negative number; cleared otherwise. The minus flag is the same as the MSB of the output (bit 35).

The minus flag is cleared during processor reset.

The minus flag can be modified by writing to ST0.

- **N: Normalized**

Set if the 32 LSBs of the ALU/Barrel Shifter output used at the last instruction are normalized; cleared otherwise, i.e., set if $Z \cup (\text{bit } 31 \oplus \text{bit } 30) \cap \overline{E}$.

The normalized flag is cleared during processor reset. The normalized flag can be modified by writing to ST0.

- **V: Overflow**

Set if an arithmetic overflow (36-bit overflow) occurs after an arithmetic operation; cleared otherwise. It indicates that the result of an operation cannot be represented in 36 bits.

- **C: Carry**

Set if an addition operation generates a carry or if a subtract generates a borrow; cleared otherwise. It also accepts the rotated bit or the last bit shifted out of the 36-bit result.

The carry flag is cleared during processor reset.

The carry flag can be modified by writing to ST0.

- **E: Extension**

Set if bits 35 to 31 of the ALU/Barrel Shifter output used at the last instruction are not identical; cleared otherwise. If the E flag is cleared, it indicates that the 4 MSBs of the output are the sign-extension of bit 31 and can be ignored.

The extension flag is cleared during processor reset.

The extension flag can be modified by writing to ST0.

- **L: Limit**

The L flag has two functions: to latch the overflow (V) flag and to indicate limitation during accumulator move or LIM operations.

Set if the overflow flag was set (overflow latch) or if a limitation occurred when performing a move instruction (MOV or PUSH) from one of the accumulators (AxH, AxL, BxL or BxH) through the data bus or if a limitation occurred when the LIM (TBD) instruction was executed. Otherwise, it is not affected.

The limit flag is cleared during processor reset.

The limit flag can be modified by writing to ST0.

- **R: Rn Register is Zero**

This flag is affected by the MODR and NORM instructions. The R flag is set if the result of the Rn modification operation (Rn; Rn + 1; Rn - 1; Rn + S) is zero; cleared otherwise. The R flag status is latched until one of the above instructions is used.

The R flag is cleared during processor reset.

The R flag can be modified by writing to ST0.

- **IM1, IM0: Interrupt 0 Mask, Interrupt 1 Mask**

IM0 – Interrupt mask for INT0

IM1 – Interrupt mask for INT1

Clear – Disable the specific interrupt.

Set – Enable the specific interrupt.

The interrupt mask bits are cleared during processor reset. The interrupt mask bits can be modified by writing to ST0.

- **IE: Interrupt Enable**

Clear – Disable all maskable interrupts.

Set – Enable all maskable interrupts.

The interrupt enable bit is cleared during processor reset. The interrupt enable bit can be modified by the instructions EINT (enable interrupts) and DINT (disable interrupts) by using RETI/RETID for returning from one of the maskable interrupt service routines or by writing to ST0.

- **SAT: Saturation Mode**

Clear – Enable the saturation when transferring the contents of the accumulator onto the data bus.

Set – Disable the saturation mode.

Note that this bit has no effect on the LIM instruction.

The saturation enable bit is cleared during processor reset.

The saturation enable bit can be modified by writing to ST0.

Status Register 1

15	14	13	12	11	10	9	8
A1E				PS		–	–

7	6	5	4	3	2	1	0
PAGE							

- **PS: Product Shifter Control**

The product shifter control bits control the scaling shifter at the output of register P as follows:

PS Bit		
Bit 11	Bit 10	**Number of Shifts**
0	0	No shift
0	1	Shift right by one
1	0	Shift left by one
1	1	Shift left by two

The PS bits are cleared during processor reset.

The PS bits can be modified by writing to ST1.

- **PAGE: Data Memory Space Page**

Used for direct address. Refer to “Memory Addressing Modes” on page 21.

The PAGE bits can be modified by the LOAD instruction, the LPG instruction or by writing to ST1.

Status Register 2

15	14	13	12	11	10	9	8
IP1	IP0	IP2	---	IU1	IU0	OU1	OU0

7	6	5	4	3	2	1	0
S	IM2	M5	M4	M3	M2	M1	M0

- **IP1, IP0, IP2: Interrupt Pending**

IP0 – Interrupt pending for INT0

IP1 – Interrupt pending for INT1

IP2 – Interrupt pending for INT2

The interrupt pending bit is set when the corresponding interrupt is active. The bit reflects the interrupt level regardless of the mask bits.

The IPx bits are read-only.

- **IU1, IU0: IUSER0, IUSER1**

The IUSERx bits reflect the logic state of the corresponding user input pins.

The IUSERx bits are read-only bits.

- **OU1, OU0: OUSER0, OUSER1**

The OUSERx bits define the logic state of the corresponding user output pins.

The OUSERx bits are cleared during processor reset.

The OUSERx bits can be modified by writing to ST2.

- **S: Shift Mode**

The shift mode bit defines the shift method. Affects all shift instructions: SHFC, SHFI, MODA, MODB, MOVS and MOVSI. Refer to “Shifting Operations” on page 9.

Clear – The shift instruction performs an arithmetic shift.

Set – The shift instruction performs a logical shift.

The shift mode bit is cleared during processor reset.

The shift mode bit can be modified by writing to ST2.

- **IM2: Interrupt 2 Mask**

Interrupt mask for INT2

Clear – Disable interrupt 2

Set – Enable interrupt 2

The interrupt mask bit is cleared during processor reset.

The interrupt mask bit can be modified by writing to ST2.

- **M5, M4, M3, M2, M1, M0: Modulo Enable**

Cleared Mn bit – When using the corresponding Rn register, the Rn register will be modified as specified by the instruction, regardless of the modulo option.

Set Mn bit – When using the corresponding Rn register, the Rn register will be modified as specified by the instruction, using the suitable modulo.

Note that the MODR instruction is the only instruction that can use one of the Rn registers without being affected by the corresponding Mn bit, using a special option field.

The Mn bits are cleared during processor reset.

The Mn bits can be modified by writing to ST2.

Interrupt Context Switching

When a program is interrupted by an interrupt service routine, it is necessary to save those registers used by the service routine so that the interrupted program resumes execution correctly. To reduce the involved overhead, a context-switching mechanism can be used for each of the following interrupts: NMI, INT0, INT1 and/or INT2. Whether a specific interrupt should use the context-switching mechanism is determined by the corresponding bit in the internal configuration register (ICR). Refer to the comments on ICR contents on page 29 and to the MOV instruction in “Instruction Set” on page 31.

When an interrupt that uses context switching is accepted, context switching occurs automatically without any impact on interrupt latency. When returning from this interrupt service routine, context switching should be used to restore the original register values automatically. Refer to the RETI and CNTX instructions in “Instruction Set” on page 31.

Context switching involves three parallel mechanisms: push/pop to/from dedicated shadow bits, swap of a dedicated page register and swap between two specific accumulators.

The following register bits are saved automatically as shadow bits, i.e., one stack level register: ST0[11:2], ST0[0], ST1[11:10], ST2[7:0]. This means that the data bits can be pushed to or popped from the shadow registers.

The page bits ST1[7:0] are swapped to an alternative register. This means that when an interrupt is accepted, the current page is saved into the alternative register while the previous (stored) value of the page is restored so that it can be used without additional initialization. When returning from the interrupt, the interrupt page is saved again into the alternative register for the next interrupt, and the page used before entering the interrupt service routine is swapped back to ST1.

The A1- and B1-accumulators are automatically swapped. Therefore, it is very convenient to use B1 to store data needed for interrupt routines. This data will be transferred automatically into the A1-accumulator on interrupt service for interrupts using the context-switching mechanism and transferred back while returning from the interrupt service routine.

A context-switching activation instruction is also available. For details refer to the CNTX instruction in “Instruction Set” on page 31.

Status Register 0 (ST0)

15	14	13	12	11	10	9	8
A0E				Z	M	N	V

7	6	5	4	3	2	1	0
C	E	L	R	IM1	IM0	IE	SAT

15	14	13	12	11	10	9	8
				Z	M	N	V

7	6	5	4	3	2	1	0
C	E	L	R	IM1	IM0		SAT

Status Register 1 (ST1)

15	14	13	12	11	10	9	8
A1E				PS		–	–

7	6	5	4	3	2	1	0
PAGE							

15	14	13	12	11	10	9	8
				PS			

7	6	5	4	3	2	1	0
PAGE							

Status Register 2 (ST2)

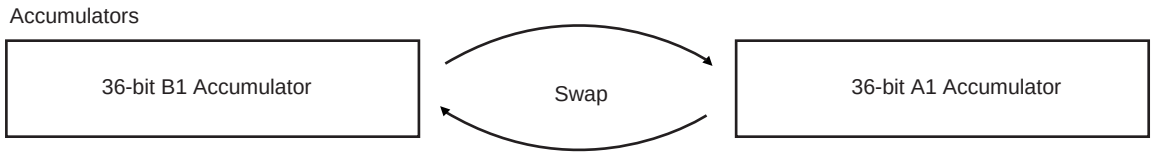
15	14	13	12	11	10	9	8
IP1	IP0	IP2	–	IU1	IU0	OU1	OU0

7	6	5	4	3	2	1	0
S	IM2	M5	M3	M3	M2	M1	M0

15	14	13	12	11	10	9	8

7	6	5	4	3	2	1	0
S	IM2	M5	M3	M3	M2	M1	M0

Figure 9. Accumulator Swapping



Internal Configuration Register

The internal configuration register includes context-switching bits, the block-repeat indication and processor status bits.

Internal Configuration Register

15	14	13	12	11	10	9	8
Reserved							

7	6	5	4	3	2	1	0
BC2	BC1	BC0	LP	IC2	IC1	IC0	NMIC

- **BC2, BC1, BC0: Block-repeat Nesting Counter**

Holds the current block-repeat loop nesting level as in the following:

BC2	BC1	BC0	Block-repeat Counter State Description
0	0	0	Not within a block-repeat loop
0	0	1	Within first block-repeat level (outer loop)
0	1	0	Within second block-repeat level
0	1	1	Within third block-repeat level
1	0	0	Within fourth block-repeat level (inner loop)

The BCx bits are cleared during processor reset and due to disabling the block-repeat mechanism, by clearing the LP bit.

The BCx bits are read-only.

- **LP: INLOOP**

Set if a block-repeat is executed; cleared otherwise.

When transferring data into ICR, the LP bit will be influenced as follows:

1 – The LP bit and the block-repeat nesting counter are cleared.

0 – The LP bit is unaffected.

Clearing this bit causes a break from the four levels of block-repeat, hence clearing the block repeat nesting counter (BCx) bits.

The inloop bit is cleared during processor reset.

The inloop bit can be cleared by writing to ICR.

In addition, refer to “Repeat and Block-repeat Unit” on page 18.

For breaking out from one block-repeat level, refer to the BREAK instruction in “Instruction Set” on page 31.

- **IC2, IC1, IC0, NMIC: Context Switching Enable**

IC2 – INT2 Context switching enable

IC1 – INT1 Context switching enable

IC0 – INT0 Context switching enable

NMIC – NMI Context switching enable

Set – Enable context switching during the beginning of the corresponding interrupt.

Clear – Disable context switching during the beginning of the corresponding interrupt.

The ICx bits and NMIC are cleared during processor reset.

The ICx bits and NMIC can be modified by writing to ICR.

Data Value Match Register

The data value match (DVM) register is part of on-core support for on-chip emulation. This register can be used by an on-chip emulation module, residing off-core, for generating a breakpoint on a data value match. A data value match occurs when the DVM register content is the same as the data on XDB. In order to enable comparison for any transaction, and since the data on XDB is not always transferred off-core, the DVM register is implemented as part of the core.

This register is also used upon servicing the TRAP routine: The PC content is transferred into the DVM in addition to the software stack. The DVM content can be transferred from/into the accumulators. Refer to the MOV instruction in the “Instruction Set” on page 31.

Instruction Set

This section provides an overview and detailed description of the OakDSPCore instruction set definition and coding, as well as complete information on the function of each instruction. The pipeline method is covered briefly. The section gives sufficient information to understand the nature of OakDSPCore programming and the capability of the instruction set itself.

Notations and Conventions

Notations

The following notations are used in this section:

Registers

rN = Address registers: r0, r1, r2, r3, r4, r5

rI = Address registers: r0, r1, r2, r3

rJ = Address registers: r4, r5

aX = a0 or a1

aXl = ax-accumulator-low (LSP), X = 0, 1

aXh = ax-accumulator-high (MSP), X = 0, 1

aXe = ax-accumulator extension, X = 0, 1

bX = b0 or b1

bXl = bx-accumulator-low (LSP), X = 0, 1

bXh = bx-accumulator-high (MSP), X = 0, 1

ac = a0, a1, a0h, a1h, a0l, a1l

bc = b0, b1, b0h, b1h, b0l, b1l

ab = a0, a1, b0, b1

cfgX = Configuration registers of DAAU (MODI or MODJ, STEPI or STEPJ), x = i, j

sv = Shift value register

sp = Stack pointer

pc = Program counter

lc = Loop counter

ext = External registers, X = 0, 1, 2, 3

REG = a0, a1, a0h, a1h, a0l, a1l, b0, b1, b0h, b1h, b0l, b1l, rN, rb, y, p or ph, sv, sp, pc, lc, st0, st1, st2, cfgi, cfgj, ext

x = x (multiplier input) register

mixp = Minimum/maximum pointer

icr = Internal configuration register

repc = Repeat counter

dvm = Data value match register

Number Representation

___ decimal

0b___ , 0B___ binary

0x___ , 0X___ hexadecimal

Data and Program Operands

Table 3 lists the data and the program operands: number of bits, operand range including the assembler mnemonics and an example for each operand. See also “Memory Addressing Modes” on page 21.

Table 3. Data Operands

Data Operand	Bit Count	Assembler Syntax			Example
		Decimal	Hexadecimal	Binary	
#signed short immediate	2’s complement 8 bits	#-128..127	#-0x80..0x7F	#-0b100000000.. 0b01111111	mov #-12, r0
#signed 6-bit immediate	2’s complement 6 bits	#-32..31	#-0x20..0x1F	#-0b100000.. 0b011111	shfi b0, a0, #-4
#signed 5-bit immediate	2’s complement 5 bits	#-16..15	#-0x10..0x0F	#0b10000.. 0b01111	movsi r1, a0, #3
#unsigned 9-bit immediate	unsigned 9 bits	#0..511	#0x000..0x1FF	#0b000000000.. 0b111111111	load #270, modi
#unsigned short immediate	unsigned 8 bits	#0..255	#0x00..0xFF	#0b00000000.. 0b11111111	add #0b10, a0
#unsigned 7-bit immediate	unsigned 7 bits	#0..127	#0x00..0x7F	#0b0000000.. 0b1111111	load #3, stepj
#unsigned 5-bit immediate	unsigned 5 bits	#0..31	#0x00..0x1F	#0b00000.. 0b11111	mov #0x5, icr
#unsigned 2-bit immediate	unsigned 2 bits	#0..3	#0x0..0x3	#0b00..0b11	load #0b11, ps
#bit number	unsigned 4 bits	#0..15	#0x0..0xF	#0b0000..0b1111	tstb r0, #12
##long immediate, ##offset	2’s complement 16 bits	##-32768..32767	##0x8000..0x7FFF		mov ##-0x9000, a0
	unsigned 16 bits	##0..65535	##0x0000..0xFFFF		mov ##0xF000, r0
offset7	2’s complement 7 bits	-64..63	-0x40..0x3F	-0b1000000.. 0b0111111	add (rb-5), a1

Table 4. Program Operands

Program Operand	Bit Count	Assembler Syntax			Example
		Decimal	Hexadecimal	Binary	
direct address	unsigned 8 bits	#0..255	#-0x00..0xFF	#0b00000000.. 0b11111111	add 120, a1
address	unsigned 16 bits	#0..65535	#-0x0000..0xFFFF		call 0x5000

Negative numbers can also be written as four hexadecimal digits. For example: -0x80 can be written as 0xFF80; -0x20 can be written as 0xFFe0.

Option Fields

Table 5. Option Field Table

eu	Extension unaffected. Optional field in the mov direct address, axh [eu] instruction. When mentioned, the data is transferred into aXh without affecting aXe. When not mentioned, the data is transferred into Xh with sign-extension into aXe.
context	Context switching. Optional field in the reti instruction. When mentioned, it means automatic context switching. When not mentioned, it means without context switching.
dmod	Disable modulo. This is an option field in the modr instruction. When mentioned, the rN is post-modified with modulo modifier disable. When not mentioned, the post-modification of rN is influenced by the Mn bit.

Condition Fields

Table 6. Condition Field Table (cond)

Mnemonic	Description	Condition
true	Always	
eq	Equal to zero	Z = 1
neq	Not equal to zero	Z = 0
gt	Greater than zero	M = 0 and Z = 0
ge	Greater than or equal to zero	M = 0
lt	Less than zero	M =1
le	Less than or equal to zero	M = 1 or Z = 1
nn	Normalized flag is cleared	N = 0
v	Overflow flag is set	V = 1
c	Carry flag is set	C = 1
e	Extension flag is set	E = 1
l	Limit flag is set	L = 1
nr	R flag is cleared	R = 0
niu0	Input user pin 0 is cleared	
iu0	Input user pin 0 is set	
iu1	Input user pin 1 is set	

Other Tokens

(x)	The contents of x
|	One of the options should be included
[ ]	Optional field at the instruction
–>	Is assigned to
>>	Shift right

<<	Shift left
exp(x)	Exponent of x
.	Not
»	Or
«	And

Flags Notation

The effect of each instruction on the flags is described by the following table:

x	The flag is affected by the execution of the instruction.
–	The flag is not affected by the instruction.
1 or 0	The flag is unconditionally set or cleared by the instruction.

For flag definitions, refer to “Status Registers” on page 23.

Conventions

1. The arithmetic operations are performed in 2’s complement.
2. The post-modification of rN registers is supported with the following instructions:
   – instructions that use an indirect addressing mode
   – modr
   – norm
   – max, maxd, min (in r0 only)

   In these instructions the contents of rN register are post-modified as follows:
   – rN, rN + 1, rN - 1, rN + step
   – Options controlled by configuration registers cfgX:
   Step size: STEPI, STEPJ – 2’s complement 7 bits (-64 to 63)
   Modulo size: MODI, MODJ – unsigned 9 bits (1 to 512)
   – Options controlled by st2:
   For each rN register, it should be defined if MODULO is enabled or disabled. In order to use MODI or MODJ the relative Mn bit must be set (the only exception for this is at modr instruction, when there is an optional field for disabling the modulo). For more details on the modulo arithmetic unit, refer to “Modulo Modifier” on page 15.
   Whenever the operand field in the instruction includes the option of (rN), it means that the rN can be post-modified in one of the four options.
   Assembler syntax: (rN), (rN)+, (rN)-, (rN) + s
   For example: mov(r0)-, r1
   mac(r4)+, (r0) + s, a0
   add(r2), al
   modr(r5)-
3. Direct addressing mode assembler syntax:
   The syntax when a one-word instruction is used is either direct address or [direct address].
4. The MSP of the P register (ph) is a write-only register. The 32-bit P register is updated after a multiply operation and can be read only by transferring it to the ALU; that is, it can be moved into aX or be an operand for arithmetic and logic operations. When transferring it into the ALU, it is sign-extended to 36 bits. This enables the user to store and restore the P register.
5. The P register is used as a source operand for different instructions: as one of the REG registers; at moda instruction – pacr function; at multiply instructions where the P register is added or subtracted from one of the accumulators. When using the P register as a source operand, it always means using the “shifted P register”. Shifted P register means that the P register is sign-extended into 36 bits and then shifted as defined at the PS field, status register st1. In shift right, the sign is extended, whereas in shift left a zero is appended into the LSB. The

contents of the P register remain unchanged. At two multiply instructions, maa and maasu, the P register is also aligned, i.e., after the P register is sign-extended and shifted according to the PS field, it is also shifted by 16 to the right.

6. All move instructions using the accumulator (aX or bX) as a destination are sign-extended.

   All instructions that use the accumulator-low (aXl or bXl) as a destination will clear the accumulator-high and the accumulator extension. Therefore, they are sign extension-suppressed.

   All instructions using the accumulator-high (aXh or bXh) as a destination will clear the accumulator-low and are sign- extended. An exception is mov direct address, axh [eu], when moving data into accumulator-high can be controlled with sign extension or with sign extension unaffected (the accumulator extension aXe is unaffected).

7. In all arithmetic operations between 16-bit registers and aX (36 bits), the 16-bit register will be regarded as the 16 low-order bits of a 36-bit operand with a sign extension in the MSBs.

8. It is recommended that the flags are used immediately after the instruction that updated them. Otherwise, very careful programming is required (some flags may be changed in the meantime).

9. The condition field is almost always an optional field, except when the condition is followed by another optional field as in reti instruction. The condition field is the last field of the instruction. When the condition field is missing, the condition is true.

   Examples: *shr4 true* is the same as *shr4*, but in *reti true, context* the true cannot be omitted.

10. General Restrictions:

    a) Arithmetic and logical operations (but not bit-manipulation operations) must not be performed with the same accumulator as the source (soperand) and the destination (doperand).

    Example: add a0, a0 is not allowed (shfc a0, a0 is allowed)

    b) An instruction immediately following an instruction that modifies the rb register may not use the index addressing mode. The only exception is when rb is modified using a long immediate operand (mov ## long immediate, rb).

11. Two nop instructions should follow instructions that use the pc as a destination register except after move ## long immediate, pc, where only one nop is needed.

ADD – Add

Syntax: add operand, aX

Operation: aX + operand –> aX

Operand: REG
[##direct address]
#unsigned short immediate
##long immediate
(rb + offset7)
(rb + ##offset)
(rN)

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1
2 when the instruction is two words long

Words: 1
2 when the operand is *##long immediate* or *(rb + ##offset)* or *[##direct address]*

Notes: The REG cannot be bX.

ADDH – Add to High Accumulator

Syntax: addh operand, aX

Operation: aX + operand • 2^{16} –> aX
aXl remains unaffected

Operand: REG
(rN)
direct address

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1

Words: 1

Notes: The REG cannot be aX, bX, p.

ADDL – Add to Low Accumulator

Syntax: addl operand, aX

Operation: aX + operand –> aX
The operation is sign-extension suppressed.

Operand: REG
(rN)
direct address

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1

Words: 1

Notes: The REG cannot be aX, bX, p.

ADDV – Add Long Immediate Value or Data Memory Location

Syntax: addv ##long immediate, operand

Operation: operand + ##long immediate –> operand
The operand and the long immediate values are sign-extended. If the operand is not part of an accumulator (aXl, aXh, aXe, bXl, bXh), then the accumulators are unaffected. If the operand is a part of an accumulator, only the addressed part is affected.

Operand: REG
(rN)
direct address

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	–	-x	x	–	–	–

Z, M, C are a result of the 16-bit operation. M is affected by bit 15.
When the operand is st0, st0 (including the flags) accepts the addition result regardless of a0e bits.

Cycles: 2

Words: 2

Notes: The REG cannot be aX, bX, p, pc. Note that aX can be used in *add ##long immediate, aX* instruction.

When adding a long immediate value to st0, st0 (including the flags) accepts the ALU output result. When adding a long immediate value to st1, the flags are affected by the ALU output, as usual.

Note that when the operand is part of an accumulator, only the addressed part is affected. For example, if the instruction *addv ##long immediate, a01* generates a carry. The carry flag is set. However, a0h is unchanged. On the other hand, the instruction *addl ##long immediate, a0l* (with same a0 and immediate values) changes the a0h and affects the carry flag according to bit 36 of the ALU result.

AND – And

Syntax: and operand, aX

Operation: If operand is aX or P:
aX[35:0] AND operand –> aX[35:0]

If operand is unsigned short immediate:
aX[7:0] AND operand –> aX[7:0]
aX[15:8] –> aX[15:8]
0 –> aX[35:16]

if operand is REG, (rN), long immediate:
aX[15:0] AND operand –> aX[15:0]
0 –> aX[35:16]
Note: If the operand is one of the a accumulators or the P register, it is ANDed with the destination accumulator.

If the operand is short immediate, the operand is zero-extended to form a 36-bit operand, then ANDed with the destination accumulator. Bits 15 to 8 are unaffected; other bits of the accumulator are cleared.

If the operand is a 16-bit register or a long immediate value, the operand is zero-extended to form a 36-bit operand, then ANDed with the accumulator. Therefore, the upper bits of the accumulator are cleared by this instruction.

Operand: REG
(rN)
direct address
[##direct address]
#unsigned short immediate
##long immediate
(rb + offset7)
(rb + ##offset)

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	–	–	–	x	–	–

Z flag is set if all the bits at the ALU output are zeroed, cleared otherwise. Note that when the operand is unsigned short immediate, ALU output bits 35 to 8 are 0.

Cycles: 1
2 when the instruction is two words long

Words: 1
2 when the operand is *##long immediate* or *(rb+offset)* or *[##direct*

address]

Notes: The instruction *and #unsigned short immediate, aX* can be used for clearing some of the low-order bits at a 16-bit destination.

For example:

mov ram, aX
and #unsigned short immediate, aX
mov aX, ram

Using the *and* instruction, bits 15:8 are unaffected. Therefore, the high-order bits at the destination do not change. See also *rst* instruction.

In addition, this instruction can be used for bit test, test one of the low-order bits of a destination (e.g., at accumulator-low).
For example:

and #unsigned short immediate, aX
br address, eq

See also the *tstb* instruction.

The REG cannot be bX.

BANKE – Bank Exchange

Syntax: banke [r0], [r1], [r4], [cfgi]

Operation: Exchange the registers appearing in the exchange list with their corresponding swap registers.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 1

Words: 1

Notes: The number and the order of the registers appearing in the exchange list may vary. Here are some valid examples:

banke r0
banke r1, cfgi
banke r1, r0
banke cfgi, r1, r4

For more details, refer to “Alternative Bank of Registers” on page 17.

BKREP – Block-Repeat

Syntax: bkrep operand, address

Operation: operand –> lc
1 –> LP status bit
BCx + 1 –> BCx

Begin a block-repeat that is to be repeated operand + 1 times.
The repetition range is from 1 to 65536.
The first block address is the address after the *bkrep* instruction and the last block address is the address specified in the *address* field. The operand is inserted into the loop counter register (lc). The inloop status bit LP is set – indicating a block-repeat loop. The block-repeat nesting level counter is incremented by one.

The repeated block is interruptible.

Operand: #unsigned short immediate
REG

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 2

Words: 2

Notes: This instruction can be nested. Four levels of block-repeat can be used.

When using an unsigned short immediate operand, the number of repetitions is between 1 and 256. When transferring the *#unsigned short immediate* number into the lc register, it is copied to the low-order 8 bits of lc. The high-order 8 bits are zero-extended.

In case the last instruction at the *block-repeat* is:
a. a one-word instruction, the *address* field should contain the address of the instruction;
b. a two-word instruction, the *address* field should contain the address of the second word.

In the outer block-repeat level, the REG cannot be aX, bX, p.
In other nested levels, the REG cannot be aX, bX, p, lc. Note that the assembler cannot check the restriction on lc register in a nested block-repeat.
The data read while reading lc during a block repeat loop execution is from the loop counter. If the outer block repeat loop has normally completed its turn with the contents of lc of 0; if it was completed using

break, the contents of lc will be the value of the loop counter at the break point.

The minimum length of the repeated block is two words.

Restrictions:
• Break cannot start at the last address of the block repeat loop.
• The following instructions cannot start at *address* – 1 in a *block-repeat* loop: *break* (1 word), *mov soperand* (2 words)*, icr*, *mov icr, ab* (2 words).
• The following instructions cannot start at the two last addresses of the block-repeat loop: *br*, *brr*, *call*, *callr*, *calla*, *ret*, *reti*, *rets*, *retd*, *retid*, *bkrep*, *rep*, instructions with pc or lc as destination, instructions with lc or icr as source.
• The following instructions cannot start at *address* – 2 of a *block-repeat* loop: instructions with lc as destination, *mov* soperand, icr.
• The following instructions cannot start at *address* – 3 of a *block-repeat* loop: *set*, *rst*, *chng*, *addv*, *subv,* with lc as destination.
• Note that illegal instruction sequences are also restricted at the last and first instructions of a *block-repeat* loop.
• Two *block-repeat* loops cannot have the same last address.

BR – Conditional Branch

Syntax: br address[, cond]

Operation: If condition, then address –> pc
If the condition is met, branch to the program memory location specified by the *address* field.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 2 if the branch is not to occur
3 if the branch is to occur

Words: 2

Notes: If the condition is met, *address* is the address of the new program memory location. The *address* is the second word of the instruction.

BREAK – Break from Block-repeat

Syntax: break

Operation: Used for breaking out of the current block repeat loop. The internal registers that contain the first address, last address and loop counter are popped.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 1

Words: 1

Notes: The break instruction cannot be the last of a block-repeat loop.

A break at the outer level does not change lc and resets the LP bit.

BRR – Relative Conditional Branch

Syntax: brr relative address[, cond]

Operation: if condition, then (pc + relative address + 1) –> pc
If the condition is met, a branch is executed to an address relative to the current program memory location. The offset range is - 63 to +64.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 2

Words: 1

CALL – Conditional Call Subroutine

Syntax: call address[, cond]

Operation: if condition, then
sp - 1 –> sp
pc –> (sp)
address –> pc

If the condition is met, the stack pointer is pre-decremented, the program counter is pushed into the software stack and a branch is performed to the program memory location specified by the *address* field.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 2 if the branch is not to occur
3 if the branch is to occur

Words: 2

Notes: If the condition is met, *address* is the address of the new program memory location. The *address* is the second word of the instruction.

CALLA – Call Subroutine at Location Specified by Accumulator

Syntax: calla aX

Operation: sp - 1 –> sp
pc –> (sp)
aXl –> pc

Call subroutine indirect (address from aXl). The stack pointer (sp) is pre-decremented. The program counter (pc) is pushed into the software stack and a branch is executed to the address pointed by accumulator-low. This instruction can be used to perform computed subroutine calls.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 3

Words: 1

CALLR – Relative Conditional Call Subroutine

Syntax: callr relative address[, cond]

Operation: if condition, then
sp - 1 –> sp
pc –> (sp)
pc + relative address + 1 –> pc

If the condition is met, the stack pointer (sp) is pre-decremented, the program counter (pc) is pushed into the software stack and a branch is executed to an address relative to the current program memory location. The offset range is - 63 to +64.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 2

Words: 1

CHNG – Change Bit-field

Syntax: chng ##long immediate, operand

Operation: operand XOR ##long immediate –> operand

Change specific bit-field in a 16-bit operand according to a long immediate value that contains ones in the bit-field location.
If the operand is not part of an accumulator (aXl, aXh, aXe, bXl, bXh), then the accumulators are un-affected. If the operand is part of an accumulator, only the addressed part is affected.
The operand and the long immediate values are sign-extension suppressed.

Operand: REG
(rN)
direct address

Affected Flags:

When the operand is not st0:

Z	M	N	V	C	E	L	R
x	x	–	–	–	–	–	–

When the operand is st0, the specified bits are changed according to the bit-field in the long immediate value regardless of whether the a0e bits have changed.

Cycles: 2

Words: 2

Notes: The REG cannot be aX, bX, p.

When changing the a0e bits (chng ##long immediate, st0), the flags are affected according to the long immediate value. When changing the a1e bits (chng ##long immediate, st1), the flags are affected according to the ALU output.

CLR – Clear Accumulator

Syntax: clr aX[, cond] See *moda* instructions.
or
clr bX[,cond] See *modb* instructions.

Operation: If the condition is met,
0 –> aX
or
0 –> bX

CLRR – Clear and Round aX-accumulator

Syntax: clrr aX[, cond]

Operation: If the condition is met, 0x8000 –> aX.

See *moda* instructions.

CMP – Compare

Syntax: cmp operand, aX

Operation: aX - operand

The subtraction result is not stored, but the status flags are set correspondingly.

Operand: REG
(rN)
direct address
[##direct address]
#unsigned short immediate
##long immediate
(rb + offset7)
(rb + offset)

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1
2 when the instruction is two words long

Words: 1
2 when the operand is *##long immediate* or *(rb + offset)* or *[##direct address]*

Notes: The REG cannot be bX.

CMPU – Compare Unsigned

Syntax: cmpu operand, aX

Operation: aX - operand

The subtraction result is not stored, but the status flags are set correspondingly. The operand is sign-extension suppressed.

Operand: REG
(rN)
direct address

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1

Words: 1

Notes: The REG cannot be aX, bX, p.

In order to compare aX with an unsigned 16-bit operand, bits 35 to 16 of the accumulator should be cleared.

CMPV – Compare Long Immediate Value to Register or Data Memory Location

Syntax: cmpv ##long immediate, operand

Operation: operand - ##long immediate

The subtraction result is not stored, but the status flags are set correspondingly. The operand and the long immediate values are sign-extended.

Operand: REG
(rN)
direct address

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	–	–	x	–	–	–

Z, M, C reflect the result of the 16-bit operation. M is affected by bit 15.

Cycles: 2

Words: 2

Notes: The REG cannot be aX, bX, p, pc. Note that aX can be used in the *cmp ##long immediate, aX* instruction.

Note that when using *subv ##long immediate, st0* and *cmpv ##long immediate, st0*, the flags are set differently.

CNTX – Context Switching Store or Restore

Syntax: cntx s|r

Operation: This instruction triggers the context-switching mechanism.

s: Store the shadow/swap bits and swap a1 and b1 accumulators' contents.
The following bits: st0[0], st0[11..2], st1[11:10], st2[7:0] are pushed to their shadow bits.
The page bits st1[7:0] are swapped with their alternative register.

r: Restore the shadow/swap bits and swap a1 and b1 accumulators' contents.
The following bits: st0[0], st0[11..2], st1[11:10], st2[7:0] are popped from their shadow bits.
The page bits st1[7:0] are swapped with their alternative register.

Affected Flags: In store, flags represent the data transferred into a1.

Z	M	N	V	C	E	L	R
x	x	x	–	–	x	–	–

In restore, flags are written from their shadow bits.

Z	M	N	V	C	E	L	R
x	x	x	–	–	x	–	–

Cycles: 1

Words: 1

COPY – Copy aX-accumulator

Syntax: copy aX[, cond]

Operation: If the condition is met, aX –> aX.

See *moda* instructions.

DEC – Decrement aX-accumulator by One

Syntax: dec aX[, cond]

Operation: If the condition is met, aX - 1 –> aX.

See *moda* instructions.

DINT – Disable Interrupt

Syntax: dint

Operation: 0 –> IE
IE bit is cleared. Disable the interrupts.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 1

Words: 1

DIVS – Division Step

Syntax: divs direct address, aX

Operation: aX - (*direct address* • 2^{15}) –> ALU output
if ALU output < 0
then aX = aX • 2
else aX = ALU output • 2 + 1

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	–	–	x	–	–

Cycles: 1

Words: 1

Notes: The 16-bit dividend is placed at accumulator-low while the accumulator-high and the accumulator-extension are cleared. The divisor is placed at the *direct address*. For a 16-bit division, DIVS should be executed 16 times. After 16 times, the quotient is in the accumulator-low and the remainder is in the accumulator-high. The dividend and the divisor should both be positive.

EINT – Enable Interrupt

Syntax: eint

Operation: 1 –> IE

IE bit is set. Enable the interrupts.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 1

Words: 1

EXP – Evaluate the Exponent Value

Syntax: exp soperand[, aX]

Operation: When using *exp soperand*:
exponent (soperand) –> sv
The soperand remains unaffected.

When using *exp soperand, aX*:
exponent (soperand) –> sv and aX
The soperand remains unaffected.

Operand: REG
(rN)

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 1

Words: 1

Notes: The REG cannot be p.

The instruction following an *exp* intruction cannot move to/from the SV register. The SV register can be used only in *shfc* and *movs* instructions.

INC – Increment Accumulator by One

Syntax: inc aX[, aX]

Operation: If the condition is met, aX + 1 –> aX.

See *moda* instructions.

LIM – Limit Accumulator

Syntax: lim aX[, aX]

Operation: When using *lim aX*:
if aX > 0x7FFFFFFF, then aX = 0x7FFFFFFF
else
if aX < 0x80000000, then aX = 0x80000000
else
aX is unaffected

When using lim aX, aX:
if aX > 0x7FFFFFFF, then aX = 0x7FFFFFFF
else
if aX < 0x80000000, then aX = 0x80000000
else
aX = aX

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	–	–	0	x	–

L flag is set when limitation occurs

Cycles: 1

Words: 1

LOAD – Load Specific Fields into Registers

Syntax: load #unsigned immediate 8 bits, page
load #unsigned immediate 9 bits, modi
load #unsigned immediate 9 bits, modj
load #unsigned immediate 7 bits, stepi
load #unsigned immediate 7 bits, stepj
load #unsigned immediate 2 bits, ps

Operation: Load a specific field (second operand) with a constant (first operand).

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 1

Words: 1

Notes: The assembler syntax permits use of *lpg #unsigned short immediate*, which is equivalent to *load #unsigned short immediate, page*.

LPG – Load the Page Bits

Syntax: lpg #unsigned short immediate

Operation: The low-order bits of st1 (page bits) are loaded with an 8-bit constant (0 to 255).

See *load* instruction.

MAA – Multiply and Accumulate Aligned Previous Product

Syntax: maa operand1, operand2, aX

Operation: aX + aligned and shifted p –> aX
operand1 –> y
operand2 –> x
signed y • signed x –> p

Operands: y, direct address
y, (rN)
y, REG
(rJ), (rI)
(rN), ##long immediate

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1
2 when the second operand is ##long immediate

Words: 1
2 when the second operand is ##long immediate

Notes: Aligned and shifted p means that the previous product is sign-extended into 36 bits, then shifted as defined by the PS field of status register st1, and then aligned with sign-extension, 16 bits to the right.

y –> y means that y retains its value.

The REG cannot be aX, bX, p.

The multiplication in *maa (rJ), (rI), aX* is between X-RAM and Y-RAM only, where rJ points to Y-RAM and rI points to X-RAM.

MAASU – Multiply Signed by Unsigned and Accumulate Aligned Previous Product

Syntax: maasu operand1, operand2, aX

Operation: aX + aligned and shifted p –> aX
operand1 –> y
operand2 –> x
signed y • unsigned x –> p

Operands: y, (rN)
y, REG
(rJ), (rI)
(rN), ##long immediate

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1
2 when the second operand is ##long immediate

Words: 1
2 when the second operand is ##long immediate

Notes: Aligned and shifted p means that the previous product is sign-extended into 36 bits, then shifted as defined by the PS field of status register st1, and then aligned with sign-extension, 16 bits to the right.

y –> y means that y retains its value.

The REG cannot be aX, bX, p.

The multiplication in *maasu (rJ), (rI), aX* is between X-RAM and Y-RAM only, where rJ points to Y-RAM and rI points to X-RAM.

MAC – Multiply and Accumulate Previous Product

Syntax: mac operand1, operand2, aX

Operation: aX + shifted p –> aX
operand1 –> y
operand2 –> x
signed y • signed x –> p

Operands: y, direct address
y, (rN)
y, REG
(rJ), (rI)
(rN), ##long immediate

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1
2 when the second operand is ##long immediate

Words: 1
2 when the second operand is ##long immediate

Notes: Shifted p means that the previous product is sign-extended into 36 bits, then shifted as defined by the PS field of status register st1.

y –> y means that y retains its value.

The REG cannot be aX, bX, p.

The multiplication in *mac (rJ), (rI), aX* is between X-RAM and Y-RAM only, where rJ points to Y-RAM and rI points to X-RAM.

MACSU – Multiply Signed by Unsigned and Accumulate Previous Product

Syntax: macsu operand1, operand2, aX

Operation: aX + shifted p –> aX
operand1 –> y
operand2 –> x
signed y • unsigned x –> p

Operands: y, direct address
y, (rN)
y, REG
(rJ), (rI)
(rN), ##long immediate

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1
2 when the second operand is ##long immediate

Words: 1
2 when the second operand is ##long immediate

Notes: Shifted p means that the previous product is sign-extended into 36 bits, then shifted as defined by the PS field of status register st1.

y –> y means that y retains its value.

The REG cannot be aX, bX, p.

The multiplication in *macsu (rJ), (rI), aX* is between X-RAM and Y-RAM only, where rJ points to Y-RAM and rI points to X-RAM.

MACUS – Multiply Unsigned by Signed and Accumulate Previous Product

Syntax: macus operand1, operand2, aX

Operation: aX + shifted p –> aX
operand1 –> y
operand2 –> x
unsigned y • signed x –> p

Operands: y, direct address
y, (rN)
y, REG
(rJ), (rI)
(rN), ##long immediate

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1
2 when the second operand is ##long immediate

Words: 1
2 when the second operand is ##long immediate

Notes: Shifted p means that the previous product is sign-extended into 36 bits, then shifted as defined by the PS field of status register st1.

y –> y means that y retains its value.

The REG cannot be aX, bX, p.

The multiplication in *macus (rJ), (rI), aX* is between X-RAM and Y-RAM only, where rJ points to Y-RAM and rI points to X-RAM.

MACUU – Multiply Unsigned by Unsigned and Accumulate Previous Product

Syntax: macuu operand1, operand2, aX

Operation: aX + shifted p –> aX
operand1 –> y
operand2 –> x
unsigned y • unsigned x –> p

Operands: y, (rN)
y, REG
(rJ), (rI)
(rN), ##long immediate

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1
2 when the second operand is ##long immediate

Words: 1
2 when the second operand is ##long immediate

Notes: Shifted p means that the previous product is sign-extended into 36 bits, then shifted as defined by the PS field of status register st1.

y –> y means that y retains its value.

The REG cannot be aX, bX, p.

The multiplication in *macus (rJ), (rI), aX* is between X-RAM and Y-RAM only, where rJ points to Y-RAM and rI points to X-RAM.

After using this instruction, the P register cannot be reconstructed. During an interrupt service routine that uses the P register, the P register should be saved before it is used, and restored before returning.
It is also recommended to disable the interrupts before a *macuu* instruction and to enable the interrupts after the instruction using the result of the unsigned product.

The instruction that uses the P register or the shifted P register as a source operand after a macuu instruction uses the unsigned result in the P register is zero extended into 36 bits and then shifted as defined by the PS field. This behavior will be in effect until a new signed product is generated or a new value is written in ph.

MAX – Maximum between Two Accumulators

Syntax: max aX, (r0), ge|gt

Operation: When using *ge*:
If aX ≥ the other a-accumulator, then
the other a-accumulator –> aX
r0 –> mixp
r0 is post-modified as specified.

When using *gt*:
If aX > the other a-accumulator, then
the other a-accumulator –> aX
r0 –> mixp
r0 is post-modified as specified.

This instruction is used to to find the maximal value between the two a-accumulators. In case the maximal value should be updated, it saves the new maximal value in the specified accumulator (aX) and saves the r0 pointer value in the mixp register. The r0 register is post-modified as specified in the instruction, regardless of whether the new maximal value is updated.

Affected Flags:

Z	M	N	V	C	E	L	R
–	x	–	–	–	–	–	–

M is set when the maximum value is found and the accumulator and mixp register are updated. Cleared otherwise.

Cycles: 1

Words: 1

Notes: mixp cannot be read in the instruction following the *max* instruction.

MAXD – Maximum between Data Memory Location and Accumulator

Syntax: maxd aX, (r0), ge|gt

Operation: When using *ge*:
If (r0) ≥ aX, then
(r0) –> aX
r0 –> mixp
r0 is post-modified as specified.

When using *gt*:
If (r0) > aX, then
(r0) –> aX
r0 –> mixp
r0 is post-modified as specified.

This instruction is used to find the maximal value between a data memory location pointed to by r0 and one of the aX-accumulators. In case r0 points to a larger (or larger or equal) value than the accumulator, the new maximal is transferred in the specified accumulator (aX) and the r0 pointer is transferred in the mixp register. The r0 register is post-modified as specified in the instruction, regardless of whether the new maximal value is updated.

Affected Flags:

Z	M	N	V	C	E	L	R
–	x	–	–	–	–	–	–

M is set when the maximum value is found and the accumulator and mixp register are updated. Cleared otherwise.

Cycles: 1

Words: 1

Notes: mixp cannot be read in the instruction following the *maxd* instruction.

MIN – Minimum between Two Accumulators

Syntax: min aX, (r0), le|lt

Operation: When using *le*:
If aX ≤ the other a-accumulator, then
the other a-accumulator –> aX
r0 –> mixp
r0 is post-modified as specified.

When using *gt*:
If aX < the other a-accumulator, then
the other a-accumulator –> aX
r0 –> mixp
r0 is post-modified as specified.

This instruction is used to find the minimal value between the two a-accumulators. In case the minimal value should be updated, it saves the new minimal value in the specified accumulator (aX) and saves the r0 pointer value in the mixp register. The r0 register is post-modified as specified in the instruction, regardless of whether the new maximal value is updated.

Affected Flags:

Z	M	N	V	C	E	L	R
–	x	–	–	–	–	–	–

M is set when the minimal value is found and the accumulator and mixp register are updated.
Cleared otherwise.

Cycles: 1

Words: 1

Notes: mixp cannot be read in the instruction following the *min* instruction.

MODA – Modify A-accumulator Conditionally

Syntax: moda func, aX[, cond]
func aX[, cond]

Operation: If the condition is met, then aX is modified by func.
The accumulator and the flags are modified according to the function field only when the condition is met.

func:	shr	aX >> 1–> aX
	shl	aX << 1 –> aX
	shr4	aX >> 4 –> aX
	shl4	aX << 4 –> aX
	ror	rotate aX right through carry
	rol	rotate aX left through carry
	clr	0 –> aX
	copy	aX –> aX
	neg	-aX –> aX
	not	not(aX) –> aX
	rnd	aX + 0x8000 –> aX
	pacr	shifted p + 0x8000 –> aX
	clrr	0x8000 –> aX
	inc	aX + 1 –> aX
	dec	aX - 1 –> aX

Affected Flags:

Arithmetic Shift:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

C is set according to the last bit shifted out of the operand (*shr*: bit 0; *shr4*: bit 3; *shl*: bit 35; *shl4*: bit 32).
V: with *shl* and *shl4*, cleared if the operand being shifted could be represented in 35/32 bits for *shl*/*shl4,* respectively. Set otherwise.

Logical Shift:

Z	M	N	V	C	E	L	R
x	x	x	–	x	x	–	–

C is set according to the last bit shifted out of the operand (*shr*: bit 0; *shr4*: bit 3; *shl*: bit 35; *shl4*: bit 32).

Rotate Right:

Z	M	N	V	C	E	L	R
x	x	x	–	x	x	–	–

C is set according to the last bit (bit 0) shifted out of the operand.

Rotate left:

Z	M	N	V	C	E	L	R
x	x	x	–	x	x	–	–

C is set according to the last bit (bit 35) shifted out of the operand.

Not, copy, clr, clrr:

Z	M	N	V	C	E	L	R
x	x	x	–	–	x	–	–

Neg, rnd, pacr:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Inc, dec:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1

Words: 1

Notes: The assembler syntax permits omission of the moda, e.g., *shr a0* is equivalent to *moda shr, a0.*

Shifted P register means that the P register is sign-extended to 36 bits and then shifted as defined by the PS field in status register st1.

Arithmetic shift is performed when the S status bit is cleared. Logical shift is performed when the S status bit is set. See “Shifting Operations” on page 9 and status register field definitions on page 23.

MODB – Modify B-accumulator Conditionally

Syntax: modb func, bX[, cond]
func bX[, cond]

Operation: If the condition is met, then bX is modified by func.
The accumulator and the flags are modified according to the function field only when the condition is met.

func:	shr	bX >> 1 –> bX
	shl	bX << 1 –> bX
	shr4	bX >> 4 –> bX
	shl4	bX << 4 –> bX
	ror	rotate bX right through carry
	rol	rotate bX left through carry
	clr	0 –> bX

Affected Flags:

Arithmetic Shift:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

C is set according to the last bit shifted out of the operand (*shr*: bit 0; *shr4*: bit 3; *shl*: bit 35; *shl4*: bit 32).
V: with *shl* and *shl4*, cleared if the operand being shifted could be represented in 35/32 bits for *shl*/*shl4,* respectively. Set otherwise.

Logical Shift:

Z	M	N	V	C	E	L	R
x	x	x	–	x	x	–	–

C is set according to the last bit shifted out of the operand (*shr*: bit 0; *shr4*: bit 3; *shl*: bit 35; *shl4*: bit 32).

Rotate Right:

Z	M	N	V	C	E	L	R
x	x	x	–	x	x	–	–

C is set according to the last bit (bit 0) shifted out of the operand.

Rotate Left:

Z	M	N	V	C	E	L	R
x	x	x	–	x	x	–	–

C is set according to the last bit (bit 35) shifted out of the operand.

Clr:

Z	M	N	V	C	E	L	R
x	x	x	–	–	x	–	–

Cycles: 1

Words: 1

Notes: The assembler syntax permits omission of the moda, e.g., *shr b0* is equivalent to *moda shr, b0.*

Shifted P register means that the P register is sign-extended to 36 bits and then shifted as defined by the PS field in status register st1.

Arithmetic shift is performed when the S status bit is cleared. Logical shift is performed when the S status bit is set. See “Shifting Operations” on page 9 and status register field definitions on page 23.

MODR – Modify rN

Syntax: modr (rN)[, dmod]

Operation: When using *modr (rN)*:
rN is modified as specified, and influenced by the corresponding Mn bit.

When using *modr (rN), dmod*:
rN is modified as specified, with modulo disabled.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	X

R is set if the 16-bit rN becomes zero after the post-modifications; cleared otherwise.

Cycles: 1

Words: 1

Notes: This instruction can also be used for loop control.
Example: *modr (r0)-*
brr address, nr

MOV – Move Data

Syntax: mov soperand, doperand

Operation: soperand –> doperand

The list below gives all the possible combinations for the operands.

Operands: soperand, doperand:

REG,	REG	<1, 2, 3, 4>
REG,	(rN)	<1, 2, 5>
(rN),	REG	<4, 5>
mixp,	REG	<6>
REG,	mixp	<1, 2, 6>
icr,	AB	
x,	AB	
dvm,	AB	
repc,	AB	
aXl,	x	
bXl,	x	
aXl,	dvm	
bXl,	dvm	
REG,	icr	<7, 8>
rN,	direct address	
aXl,	direct address	
aXh,	direct address	
bXl,	direct address	
bXh,	direct address	
y,	direct address	
rb,	direct address	
sv,	direct address	
direct address,	rN	
direct address,	aX	
direct address,	aXl	
direct address,	aXh[,eu]	<10>
direct address,	bX	
direct address,	bXl	
direct address,	bXh	
direct address,	y	
direct address,	rb	
direct address,	sv	
[##direct address],	aX	
aXl,	[##direct address]	
(sp),	REG	<4, 6>
(rb+#offset7),	aX	
(rb+##offset),	aX	
aXl,	(rb+#offset7)	
aXl,	(rb+##offset)	

##long immediate,	REG	<4>
#unsigned short immediate,	aXl	
#signed short immediate,	aXh	
#signed short immediate,	rN	<9>
#signed short immediate,	y	<9>
#signed short immediate,	b	<9>
#signed short immediate,	extX	<9>
#signed short immediate,	sv	<9>
#unsigned short immediate,	icr	<7, 8>

Affected Flags: No effect when doperand is not ac, bc, st0, or when soperand is not aXl, aXh, bXl, bXh.

When soperand is aXl, aXh, bXl, or bXh:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	x	–

When doperand is ac or bc:

Z	M	N	V	C	E	L	R
X	x	x	–	–	x	–	–

If doperand is st0, st0 (including the flags) accepts the transferred data.

Cycles: 1
2 when the instruction is a two-word instruction

Words: 1
2 when the operand is *33 long immediate* or *(rb+##offset)* or *[##direct address]*

Notes: The 32-bit P register can be transferred (through the product output shifter) only to aX (mov p, aX). ph is write only. Therefore, soperand cannot be ph.

The 36-bit accumulators can be a soperand only with the *mov ab, ab* instruction.

With *mov reg, reg* the soperand cannot be the same as the doperand.

When the doperand is the pc register, two nop instructions must be placed after the *mov soperand, pc* instruction, except for the *mov ##long immediate, pc*, where only one nop is needed.

It is not permitted to move a data from a location pointed to by one of the registers to the same rN register (and vice versa) with post-modification.

The reg cannot be bX.

Enable or disable of context switching (by a write to icr) takes effect after the next sequential instruction. For example, when the user enables context switching for a specific interrupt, if the same interrupt is accepted immediately after the write to icr, it will not activate the context-switching mechanism.

a *mov soperand, icr* cannot be followed by a *bkrep* instruction.

Loading the doperand by a short immediate number causes sign-extension.

The eu field is an optional field. When the eu field is specified, the accumulator extension remains
unaffected.

MOVD – Move from Data Memory into Program Memory

Syntax: movd (rI), (rJ)

Operation: rI poins to data memory location
rJ points to program memory location
(rI) –> (rJ)
rI and rJ are post-modified as specified

Move a word from data memory location pointed to by rI into a program memory location pointed to by rJ.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 4

Words: 1

Notes: It is forbidden for the rI register to point to the *movd* instruction address or to (*movd* address) + 1.

MOVP – Move from Program Memory into Data Memory

Syntax: movp soperand, doperand

Operation: soperand points to a program memory location.
soperand –> doperand

Move a word from program memory location pointed to by soperand into a data memory location pointed to by doperand or into REG. When using aX as a soperand, the address is defined by the aX-accumulator low.

Operands: soperand, doperand:
(aXl), REG
(rN), (rI)

Affected flags: No effect when doperand is not ac, st0.

When doperand is ac:

Z	M	N	V	C	E	L	R
x	x	x	–	–	x	–	–

If the operand is st0, st0 (including the flags) will accept the pointed program memory contents.

Cycles: 3

Words: 1

Notes: When the REG operand is the pc register, two *nop* instructions must be placed after the *movp (aX)*, pc instruction.

The REG operand cannot be bX.

MOVR – Move and Round

Syntax: movr operand, aX

Operation: operand + 0x8000 –> aX

Operand: REG
(rN)

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1

Words: 1

Notes: The REG cannot be bX.

MOVS – Move and Shift According to Shift Value Register

Syntax: movs operand, ab

Operation: The operand is sign-extended to 36 bits.
If 0 < sv ≤ 36, then operand << sv –> ab
If -36 ≤ sv < 0, then operand >> -sv –> ab
If sv = 0, then operand –> ab.

Operand: REG
(rN)
direct address

Affected Flags: When arithmetic shift is performed:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

When logical shift is performed:

Z	M	N	V	C	E	L	R
x	x	x	–	x	x	–	–

Cycles: 1

Words: 1

Notes: The REG cannot be p.

When operand is ab, the assembler translates it into an shfc instruction.

MOVSI – Move and Shift According to an Immediate Shift Value

Syntax: movsi operand, ab, #signed 5-bit immediate

Operation: The operand is sign-extended to 36 bits.
If 0 < #immediate ≤ 15, then operand << #immediate –> ab
If -16 ≤ #immediate < 0, then operand >> - #immediate –> ab
If #immediate = 0, then operand –> ab.

Operand: rN
y
rb

Affected Flags: When arithmetic shift is performed:

Z	M	N	V	C	E	L	R
x	x	x	0	x	x	x	–

When logical shift is performed:

Z	M	N	V	C	E	L	R
x	x	x	–	x	x	–	–

Note: If #immediate = 0, the C flag is cleared.

Cycles: 1

Words: 1

MPY – Multiply

Syntax: mpy operand1, operand2

Operation: operand1 –> x
operand2 –> y
signed y • signed x –> p

Operands: y, direct address
y, (rN)
y, REG
(rJ), (rI)
(rN), ##long immediate

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

Notes: y –> y means that y retains its value.

The REG cannot be aX, bX, p.

The multiplication in mpy (rJ), (rI) is between X-RAM and Y-RAM only, where rJ points to Y-RAM and rI points to X-RAM.

MPYI – Multiply Signed Short Immediate

Syntax: mpyi y, #signed short immediate

Operation: #signed short immediate –> x
signed y • signed x –> p

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 1

Words: 1

MPYSU – Multiply Signed by Unsigned

Syntax: mpysu operand1, operand2

Operation: operand1 –> x
operand2 –> y
signed y • unsigned x –> p

Operands: y, (rN)
y, REG
(rJ), (rI)
(rN), ##long immediate

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

Notes: y –> y means that y retains its value.

The REG cannot be aX, bX, p.

The multiplication in mpy (rJ), (rI) is between X-RAM and Y-RAM only, where rJ points to Y-RAM and rI points to X-RAM.

MSU – Multiply and Subtract Previous Product

Syntax: msu operand1, operand2, aX

Operation: aX - shifted p –> aX
operand1 –> x
operand2 –> y
signed y • signed x –> p

Operands: y, direct address
y, (rN)
y, REG
(rJ), (rI)
(rN), ##long immediate

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

Notes: Shifted P register means that the previous product is sign-extended into 36 bits, then shifted as defined in the PS field, status register 1.

y –> y means that y retains its value.

The REG cannot be aX, bX, p.

The multiplication in msu (rJ), (rI), aX is between X-RAM and Y-RAM only, where rJ points to Y-RAM and rI points to X-RAM.

NEG – 2’s Complement of aX-accumulator

Syntax: neg aX[, cond]

Operation: -aX –> aX

See *moda* instruction.

NORM – Normalize

Syntax: norm aX, (rN)

Operation: If N = 0 (aX is not normalized), then aX • 2 –> aX and rN is modified as specified
else
nop
nop

This instruction is used to normalize the signed number in the accumulator. It affects the rN register.

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	x

The R flag is updated in the *norm* instruction **only** when the rN pointer is modified.
C is set or cleared as in *shl* (*moda*).

Cycles: 2

Words: 1

Notes: The *norm* instruction uses the N flag to decide between shift or *nop*. Therefore, when using *norm* in the first iteration of a loop, the flag must be updated according to aX.

To normalize a number with the *norm* instruction, the *norm* instruction can be used together with a *rep* instruction.

```
Example:    rep #n
            norm a0, (r0)+
```

Another method is to use the N flag for conditional branch.

```
Example:    nrm:    norm a0, (r0)+
                    brr nrm, nn
```

Normalization can also be performed using the *exp* and *shift* instructions. For more details, refer to “Normalization” on page 11.

NOT – Logical Not

Syntax: not aX[,cond]

Operation: not(aX) –> aX

See *moda* instruction.

OR – Logical Or

Syntax: or operand, aX

Operation: If the operand is aX or p:
aX[35:0] or operand –> aX[35:0]

If operand is REG, (rN), unsigned short immediate, long immediate:
aX[15:0] or operand –> aX[15:0]
aX[35:16] –> aX[35:16]

If the operand is one of the aX-accumulators or the shifted p register, it is ORed with the destination accumulator.

If the operand is a 16-bit register or an immediate value, the operand is zero-extended to form a 36-bit operand, then ORed with the accumulator. Consequently, the upper bits of the accumulator are unaffected by this instruction.

Operands: REG
(rN)
direct address
[##direct address]
#unsigned short immediate
##long immediate
(rb + #offset7)
(rb + ##offset)

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	–	–	x	–	–

Cycles: 1
2 when the instruction is coded on two words

Words: 1
2 when the operand is *##long immediate*, *(rb + ##offset),* or *[##direct address].*

Notes: The REG cannot be bX.

PACR – Product Move and Round to aX-accumulator

Syntax: pacr aX[,cond]

Operation: shifted p + 0x8000 –> aX

See *moda* instruction.

POP – Pop from Stack into Register

Syntax: pop REG

Operation: (sp) –> REG
sp + 1 –> sp

The top of the stack is popped into one of the registers and the stack pointer is post-incremented.

Affected Flags: When REG is ac:

Z	M	N	V	C	E	L	R
x	x	x	–	–	x	–	–

If the REG is st0, st0 (including the flags) accepts the popped data.

Cycles: 1

Words: 1

Notes: When popping to p, the data is transferred into p-high (ph).

The Reg cannot be sp, bX.

PUSH – Push Register or Long Immediate Value onto Stack

Syntax: push operand

Operation: sp -1 –> sp
operand –> (sp)

The stack pointer (sp) is pre-decremented and the operand is pushed onto the software stack.

Operands: REG
##long immediate

Affected Flags: When the operand is aXl, aXh, bXl or bXh:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	x	–

In other cases, the flags remain unaffected.

Cycles: 1
2 when the operand is ##long immediate

Words: 1
2 when the operand is ##long immediate

Notes: The REG cannot be aX, bX, p, sp.

The push instruction cannot follow instuctions that have sp as the destination operand.

REP – Repeat Next Instruction

Syntax: rep operand

Operation: Begins a single-word instruction loop that is to be repeated operand + 1 times. The repetition range is from 1 to 65536.
The repeat mechanism is interruptible and the interrupt service routine can use another repeat (i.e., nested repeat). The nested repeat is uninterruptible.

Operands: #unsigned short immediate
REG

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 1

Words: 1

Notes: Interrupts are not accepted within *rep* loops in the following places:
• Between the *rep* instruction and the first execution of the repeated instruction
• Between the last instruction repetition and the next sequential instruction

When using an unsigned short immediate operand, the number of repetitions is between 1 and 256. When transferring the #unsigned short immediate number into the lc register, it is copied to the low-order 8 bits of lc. The higher-order bits are zero-extended.

The REG cannot be aX, bX, p.

The instructions that break the pipeline cannot be repeated: brr; callr; trap; ret; reti; retid; rets; rep; calla; mov operand, pc; pop pc; movp (aX), pc; mov repc, ab.

Rep can be performed inside a block-repeat (bkrep).

RET – Return Conditionally

Syntax: ret [cond]

Operation: If the condition is true, then
(sp) –> pc
sp + 1 –> sp

If the condition is met, the program counter (pc) is pulled from the software stack, while the previous program counter is lost; the stack pointer (sp) is post-incremented. This instruction is used to return from subroutines or interrupts.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 2 if the return is not performed
3 if the return is performed

Words: 1

Notes: This instruction can also be used to return from the maskable interrupt service routines (INT0 or INT1 or INT2) to enable additional interrupts, the IE bit in st0 must be set by the user.
The *ret* instruction cannot follow the following instructions:
• *mov soperand, sp* (except for *mov ##long immediate, sp*)
• *movp (aXl), sp; addv/subv/set/rst/chng ##long immediate, sp*

RETD – Delayed Return

Syntax: retd

Operation: (sp) –> temporary storage
sp + 1 –> sp
One two-cycle instruction or two one-cycle instructions are executed.
Temporary storage –> pc

Delayed return. The two single-cycle instructions, or one two-cycle instruction (*brr*; *callr; rep*; *trap*; *retd*; *retid*; *mov operand*, *pc*; *pop pc*) are/is fetched and executed before executing the return. When returned, the program counter (pc) is pulled from the software stack, while the previous program counter is lost; the stack pointer (sp) is post-incremented. This instruction is used to obtain a delayed return from subroutines or interrupts.
The *retd* instruction and the instruction(s) that follow the *retd* (two one-cycle instructions or one two-cycle instruction) cannot be interrupted.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 1

Words: 1

Notes: The *retd* instruction and the two following cycles are uninterruptible.

The two cycles following a retd cannot be instructions that break the pipeline: *brr*; *callr; rep*; *trap*; *retd*; *retid*; *mov operand*, *pc*; *pop pc*; *addv*/*subv*/*set*/*rst*/*chng*/ *##long immediate*, *pc*.

This instruction can also be used as return from the maskable interrupts service routines (INT0 or INT1 or INT2) to enable additional interrupts, the IE bit at st0 must be set by the user.

The retd instruction cannot follow the following instructions:
- *mov soperand, sp* (except for *mov ##long immediate, sp*)
- *movp (aXl), sp; addv/subv/set/rst/chng ##long immediate, sp*

RETI – Return from Interrupt Conditionally

Syntax: reti [cond [, context]]

Operation: If the condition is met, then
(sp) –> temporary storage
sp + 1 –> sp
1 –> IE

This instruction is used to return from interrupt service routines with or without interrupt context switching.

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 2 in case the return is not performed
3 in case the return is performed

Words: 1

Notes: IE is set only when returning from INT0, INT1 or INT2 service routine.

When the context field is specified, the interrupt is returned with context switching. See “Interrupt Context Switching” on page 27.

RETID – Delayed Return from Interrupt

Syntax: retid

Operation: (sp) –> temporary storage
sp + 1 –> sp
1 –> IE
One two-cycle instruction or two one-cycle instructions are executed.
Temporary storage –> sp

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 1

Words: 1

Notes: The IE bit is set only when returning from INT0, INT1 or INT2 service routine.

The two cycles following a *retid* cannot be instructions that break the pipeline: *brr*; *callr*; *rep*; *trap*; *retd*; *retid*; *mov operand*, *pc*; *pop pc*; *addv/subv/set/rst/chng/ ##long immediate*, *pc*.

RETS – Return with Short Immediate Parameter

Syntax: rets #unsigned short immediate

Operation: (sp) –> pc
sp + 1 + #immediate –> sp

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 3

Words: 1

Notes: This instruction is used to return from subroutines or interrupts and delete unnecessary parameters from the stack.

This instruction can also be used to return from the maskable interrupt service routines when the IE bit must be left unaffected (with *reti*, IE is set to 1).

This instruction can also be used as return from the maskable interrupts service routines (INT0 or INT1 or INT2) to enable additional interrupts, the IE bit at st0 must be set by the user.

The retd instruction cannot follow the following instructions:
- *mov soperand, sp* (except for *mov ##long immediate, sp*)
- *movp (aXl), sp; addv/subv/set/rst/chng ##long immediate, sp*

RND – Round Upper 20 Bits of aX-accumulator

Syntax: rnd aX[, cond]

Operation: aX + 0x8000 –> aX

See *moda* instruction.

ROL – Rotate Accumulator Left through Carry

Syntax: rol aX[, cond]
rol bX[, cond]

Operation: Rotate the specified accumulator left through carry.

See *moda* and *modb* instructions.

ROR – Rotate Accumulator Right through Carry

Syntax: ror aX[, cond]
ror bX[, cond]

Operation: Rotate the specified accumulator right through carry.

See *moda* and *modb* instructions.

RST – Reset Bit-field

Syntax: rst ##long immediate, operand

Operation: operand and not (##long immediate) –> operand

Reset a specific bit-field in a 16-bit operand according to a long immediate value. The long immediate value contains ones in the bit-field locations.

The operand and the long immediate value are sign-extension suppressed.

Operands: REG
(rN)
direct adddress;

Affected Flags: When the operand is not st0:

Z	M	N	V	C	E	L	R
x	x	–	–	–	–	–	–

When the operand is st0, the specified bits are reset.

Cycles: 2

Words: 2

Notes: The REG cannot be aX, bX, p, pc.

If the operand is not a part of an accumulator, then the accumulators are unaffected. If the operand is a part of an accumulator, then only the addressed part is affected.

When resetting the a0e bits (*rst ##long immediate, st0*), the flags are reset according to the long immediate value. When resetting the a1e bits (*rst ##long immediate, st1*), the flags are reset according to the ALU output.

SET – Set Bit-field

Syntax: set ##long immediate, operand

Operation: operand or ##long immediate –> operand

Set a specific bit-field in a 16-bit operand according to a long immediate value. The long immediate value contains ones in the bit-field locations.

The operand and the long immediate value are sign-extension suppressed.

Operands: REG
(rN)
direct adddress;

Affected Flags: When the operand is not st0:

Z	M	N	V	C	E	L	R
x	x	–	–	–	–	–	–

When the operand is st0, the specified bits are set.

Cycles: 2

Words: 2

Notes: The REG cannot be aX, bX, p, pc.

If the operand is not a part of an accumulator, then the accumulators are unaffected. If the operand is a part of an accumulator, then only the addressed part is affected.

When setting the a0e bits (*set ##long immediate, st0*), the flags are set according to the long immediate value. When resetting the a1e bits (*set ##long immediate, st1*), the flags are set according to the ALU output.

SHFC – Shift Accumulators according to Shift Value Register

Syntax: shfc soperand, doperand[, cond]

Operation: When the condition is met:
soperand << sv –> doperand

If 0 < sv <= 36 then
soperand << sv –> doperand

If -36 <= sv < 0 then
soperand >> |sv| –> doperand

If sv = 0 then
soperand –> doperand

Operands: ab, ab

Affected Flags: When arithmetic shift is performed:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

V: - if sv is negative or zero (shift right), then V is cleared.
- if sv is positive, less than 36 (shift left) and the soperand can be represented in (36 sv) bits, then V is cleared. V is set otherwise.
- if sv = 36 and the operand is not zero, then V is set. V is cleared otherwise.

C: cleared if sv = 0.

When logical shift is performed:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	–	–

C: cleared if sv = 0.

Cycles: 1

Words: 1

Notes: In case that the sv content is zero, this instruction is a conditional move between the two accumulators.

If soperand and doperand are different, then soperand is unaffected.

SHFI – Shift Accumulators by an Immediate Shift Value

Syntax: shfi soperand, doperand, #signed 6-bit immediate

Operation: If 0 < #immediate <= 31 then
soperand << #immediate –> doperand

If -32 <= #immediate < 0 then
soperand >> #|immediate –> doperand

If #immediate = 0 then
soperand -> doperand

If soperand is not equal to doperand then
soperand is unaffected.

Operands: ab, ab

Affected flags: When arithmetic shift is performed:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

V: If -32 <= #immediate <= (shift right) then V is cleared.
If 0 < #immediate <= 31 (shift left) and the operand before being shifted can be represented in (36 - #immmediate) bits, then V is cleared; set otherwise.

When logical shift is performed:

Z	M	N	V	C	E	L	R
x	x	x	–	x	x	–	–

C: cleared if #immediate = 0.

Cycles: 1

Words: 1

Notes: In case the immediate shift value is zero, this instruction can be used as a move instruction between the 36-bit accumulators.

SHL – Shift Accumulator Left

Syntax: shl aX[, cond]
shl bX[, cond]

Operation: Shift the specified accumulator left by one bit.

See *moda* and *modb* instructions.

SHL4 – Shift Accumulator Left by 4 Bits

Syntax: shl4 aX[, cond]
shl4 bX[, cond]

Operation: Shift the specified accumulator left by four bits.

See *moda* and *modb* instructions.

SHR – Shift Accumulator Right

Syntax: shr aX[, cond]
shr bX[, cond]

Operation: Shift the specified accumulator right by one bit.

See *moda* and *modb* instructions.

SHR4 – Shift Accumulator Right by 4 Bits

Syntax: shr4 aX[, cond]
shr4 bX[, cond]

Operation: Shift the specified accumulator right by four bits.

See *moda* and *modb* instructions.

SQR – Square

Syntax: sqr operand

Operation: operand –> x
operand –> y
signed y • signed x –> p

Operands: direct address
(rN)
REG

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 1

Words: 1

Notes: The REG cannot be aX, bX, p.

SQRA – Square and Accumulate Previous Product

Syntax: sqra operand, aX

Operation: aX + shifted p –> aX
operand –> x
operand –> y
signed y • signed x –> p

Operands: direct address
(rN)
REG

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1

Words: 1

Notes: Shifted P register means that the previous product is sign-extended to 36 bits, then shifted as defined by the ps field, status register st1.

The REG cannot be aX, bX, p.

SUB – Subtract

Syntax: sub operand, aX

Operation: aX - operand –> aX

Operands: direct address
(rN)
REG
[##direct address]
#unsigned short immediate
##long immediate
(rb + #offset7)
(rb + ##offset)

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1
2 when the instruction is a two-word instruction

Words: 1
2 when the operand is *##long immediate* or *(rb + ##offset)* or *[##direct address]*

Notes: The REG cannot be bX.

SUBH – Subtract from High Accumulator

Syntax: subh operand, aX

Operation: aX - operand • 2^{16} –> aX
The aXl remains unaffected.

Operands: direct address
(rN)
REG

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1

Words: 1

Notes: The REG cannot be aX, bX, p.

SUBL – Subtract from Low Accumulator

Syntax: subl operand, aX

Operation: aX - operand –> aX
The operand is sign-extension suppressed.

Operands: direct address
(rN)
REG

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	x	x	x	x	–

Cycles: 1

Words: 1

Notes: The REG cannot be aX, bX, p.

SUBV – Subtract Long Immediate Value from a Register or a Data Memory Location

Syntax: subv ##long immediate, operand

Operation: operand - ##long immediate –> operand

The operand and the long immediate values are sign-extended. If the operand is not part of an accumulator (aXl, aXh, aXe, bXl, bXh), then the accumulators are unaffected. If the operand is a part of an accumulator, only the addressed part is affected.

Operand: REG
(rN)
direct address

Affected Flags:

When operand is **NOT** st0:

Z	M	N	V	C	E	L	R
x	x	–	–	x	–	–	–

Z, M, C are a result of the 16-bit operation. M is affected by bit 15.
When the operand is st0, st0 (including the flags) accepts the subtraction result, regardless of a0e bits.

Cycles: 2

Words: 2

Notes: The REG cannot be aX, bX, p, pc. Note that aX can be used in *sub ##long immediate*, aX instruction.

When subtracting a long immediate value from st0, st0 (including the flags) accepts the ALU output result. When subtracting a long immediate value from st1, the flags are affected by the ALU output, as usual.

Note that when the operand is part of an accumulator, only the addessed part is affected. For example, if the instruction *subv ##long immediate, a0l* generates a borrow, the carry flag is set. However, a0h is unchanged. On the other hand, the instruction *subl ##long immediate, a0l* (with same a0 and immediate values) changes the a0h and affects the carry flag according to bit 36 of the ALU result.

Note that when using *subv ##long immediate, st0* and *cmpv ##long immediate, st0* the flags are set differently.

SWAP – Swap aX- and bX-accumulators

Syntax: swap option

Operation: swap between aX- and bX-accumulators according to the following options:

Assembler Mnemonic	Operation
swap (a0, b0), (a1, b1)	a0 <—> b0, a1<—> b1
swap (a0, b1), (a1, b0)	a0 <—> b1, a1 <—> b0
swap (a0, b0)	a0 <—> b0
swap (a0, b1)	a0 <—> b1
swap (a1, b0)	a1 <—> b0
swap (a1, b1)	a1 <—> b1
swap (a0, b0, a1)	a0 –> b0 –> a1
swap (a0, b1, a1)	a0 –> b1 –> a1
swap (a1, b0, a0)	a1 –> b0 –> a0
swap (a1, b1, a0)	a1 –> b1 –> a0
swap (b0, a0, b1)	b0 –> a0 –> b1
swap (b0, a1, b1)	b0 –> a1 –> b1
swap (b1, a0, b0)	b1 –> a0 –> b0
swap (b1, a1, b0)	b1 –> a1 –> b0

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	–	–	x	–	–

In case of *swap (a0, b0), (a1, b1)* and *swap (a0, b1), (a1, b0)*, the flags represent the data transferred into a0.
In other cases, the flags represent the data transferred into aX.

Cycles: 1

Words: 1

Notes: When the operation is x -> y -> z,
this means that
y-> z
then x -> y

TRAP – Software Interrupt

Syntax: trap

Operation:
sp - 1 –> sp
pc –> (sp)
pc –> dvm
0x0002 –> pc
Disable interrupts (INT0, INT1, INT2, NMI, BI)

Affected Flags:

Z	M	N	V	C	E	L	R
–	–	–	–	–	–	–	–

Cycles: 2

Words: 1

Notes: The software interrupt (TRAP) and the breakpoint interrupt (BI) share the same vector address. For more details on TRAP/BI, refer to “TRAP/BI Operation” on page 142.

The *trap* instruction should not be used in a TRAP/BI service routine.

To return from TRAP/BI service routine, it is advisable to use only the *reti* or *retid* instruction.

TST0 – Test Bit-field for Zeros

Syntax: tst0 mask, operand

Operation: If (operand and mask) = 0, then Z = 1; else Z = 0.
The operand and the mask are sign-extension suppressed.

Operands:

mask:	aXl
	##long immediate
operand	REG
	(rN)
	direct address

Affected Flags:

Z	M	N	V	C	E	L	R
x	–	–	–	–	–	–	–

Cycles: 1
2 when the mask is ##long immediate

Words: 1
2 when the mask is ##long immediate

Notes: The instructions *tst0 a0l, a0l* and *tst0 a1l, a1l* are forbidden.

The REG cannot be aX, bX, p.

TST1 – Test Bit-field for Ones

Syntax: tst1 mask, operand

Operation: If ($\overline{\text{operand}}$ and mask) = 0, then Z = 1; else Z = 0.
The operand and the mask are sign-extension suppressed.

Operands:

mask:	aXl
	##long immediate
operand	REG
	(rN)
	direct address

Affected Flags:

Z	M	N	V	C	E	L	R
x	–	–	–	–	–	–	–

Cycles: 1
2 when the mask is ##long immediate

Words: 1
2 when the mask is ##long immediate

Notes: The instructions *tst1 a0l, a0l* and *tst1 a1l, a1l* are forbidden.

The REG cannot be aX, bX, p.

TSTB – Test Specific Bit

Syntax: tstb operand, #bit number

Operation: If operand[#bit number] = 1, then Z = 1; else Z = 0.

Operand: REG
(rN)
direct address

#bit number is between 0 and 15

Affected Flags:

Z	M	N	V	C	E	L	R
x	–	–	–	–	–	–	–

Cycles: 1

Words: 1

Notes: The REG cannot be aX, bX, p.

XOR – Exclusive Or

Syntax: xor operand, aX

Operation: if the operand is aX or p:
aX[35:0] xor operand –> aX[35:0]

if the operand is REG, (rN):
aX[15:0] xor operand –> aX[15:0]
aX[35:16] –> aX[35:16]

Operand: REG
(rN)
direct address
[##direct address]
#unsigned short immediate
##long immediate
(rb + #offset7)
(rb + ##offset)

Affected Flags:

Z	M	N	V	C	E	L	R
x	x	x	–	–	x	–	–

Cycles: 1
2 when the instruction is two words long

Words: 1
2 when the operand is *##long immediate* or *(rb + ##offset)* or *[##direct address]*.

Notes: The REG cannot be bX.

Instruction Coding

This section provides a condensed overview of the coding of the OakDSPCore instruction set. It lists all the codes and number cycles and words for all instructions.

The first section provides the definition of the fields' abbreviations and their coding.

The second section tabulates the detailed coding of all OakDSPCore instructions, including the number of cycles and the number of words.

Abbreviation Definition and Encoding

- The "." letter anywhere in the code means do not care. It is translated as "0" by the assembler.
- Bolded opcodes are coding families that are translated to several opcodes according to the table below the code.

The coding families are:

ALU – ALU opcodes

ALM – ALU and MULTIPLY opcodes

ALB – ALU and BMU opcodes

A(aX) =	0	Accumulator a0
	1	Accumulator a1
B(bX) =	0	Accumulator b0
	1	Accumulator b1
L =	0	Low
	1	High
AB =	00	bo
	01	b1
	10	a0
	11	a1
AB1 =	00	b01
	001	b0h
	010	b11
	011	b1h
	100	a01
	101	a0h
	110	a11
	111	a1h

dddddddd = direct address bits

vvvvvvvv = 8-bit short immediate

0000000 = 7-bit offset (offset7) of relative and index addressing modes

BBBB = bit number (one of 16 bits of a register)

nnn (rN) =	000	r0
	001	r1
	010	r2
	011	r3
	100	r4
	101	r5
nnn (rN*) =	000	r0
	001	r1
	010	r2
	011	r3
	100	r4
	101	r5
	110	rb
	111	y
rrrr (register) =	0000	r0
	00001	r1
	00010	r2
	00011	r3
	00100	r4
	00101	r5
	00110	rb
	00111	y
	01000	st0
	01001	st1
	01010	st2
	01011	p/ph
	01100	pc
	01101	sp
	01110	cfgi
	01111	cfgj
	10000	b0h
	10001	b1h
	10010	b01
	10011	b1l
	10100	ext 0
	10101	ext1

	10110	ext2
	10111	ext3
	11000	a0
	11001	a1
	11010	a0l
	11011	a1l
	11100	a0h
	11101	a1h
	11110	1c
	11111	sv

Modification of rN:

mm =	00	No modification
	01	+1
	10	-1
	11	+ step

Modification of rI:

ii =	00	No modification
	01	+1
	10	-1
	11	+ step

Modification of rJ:

jj =	00	No modification
	01	+1
	10	-1
	11	+ step
w (fJ) =	0	r4
	1	r5
qq (rJ) =	00	r0
	01	r1
	10	r2
	11	r3
cccc =	0000	true
	0001	eq
	0010	neg
	0011	gt
	0100	ge

0101	1t
0110	le
0111	nn
1000	c
1001	v
1010	e
1011	l
1100	nr
1101	niu0
1110	iu0
1111	iu1

Instruction Coding Table

This section tabulates the detailed coding of all OakDSPCore instructions, including the number of cycles and the number of words.

The instructions are ordered according to the instruction group. Some of the codes are organized in subgroups. Following each subgroup appears a list of instructions that use the code of this subgroup and the encoding of the XX..X field for each instruction. Notice that the same instruction may appear in several subgroups, depending on the addressing mode or operand usage. See Table 7.

Table 7. Instruction Coding Table

Opcode		Code	Cycles	Words
ALM direct		101XXXXAddddddddd	1	1
ALM (rN)		100XXXXA100mmnnn	1	1
ALM register		101XXXXA101rrrrr	1	1
XXX =	0000	or		
	0001	and		
	0010	xor		
	0011	add		
	0100	tst0_a (mask in aX1)		
	0101	tst1_a (mask in aX1)		
	0110	cmp		
	0111	sub		
	1000	msu		
	1001	addh		
	1010	addl		
	1011	subh		
	1100	subl		
	1101	sqr		
	1110	sqra		
	1111	cmpu		
ALU #short immediate		1100XXXAvvvvvvv	1	1
ALU ##long immediate		1000XXXA11000000	2	2
ALU (rb + #offset7), ax		0100XXXA0ooooooo	1	1
ALU (rb + ##offset),ax		1101010A11011XXX	2	2
ALU [##direct add.],ax		1101010A11111XXX	2	2
XXX =	000	or		
	001	and		
	010	xor		
	011	add		
	110	cmp		
	111	sub		

Table 7. Instruction Coding Table (Continued)

Opcode		Code	Cycles	Words
ffff =	0000	shr		
	0001	shr4		
	0010	shl		
	0011	shl4		
	0100	ror		
	0101	rol		
	0110	clr		
	0111	reserved		
	1000	not		
	1001	neg		
	1010	rnd		
	1011	pacr		
	1100	clrr		
	1101	inc		
	1110	dec		
	1111	copy		
norm		10001010A110mmnn	2	1
divs		0000111Adddddddd	1	1
ALB (rN)		1000XXX0111mmnnn	2	2
ALB register		1000XXX1111rrrrr	2	2
ALB direct		1110XXX1ddddddddd	2	2
XXX =	000	set		
	001	rst		
	010	chng		
	011	addv		
	100	tst0_(mask in ##long immediate)		
	101	tst0_(mask in ##long immediate)		
	110	cmpv		
	111	subv		
maxd		100000fA011mm000	1	1
f =	0	ge		
	1	gt		
max		1000001fA011mm000	1	1
f =	0	ge		
	1	gt		
lim		01000010000111XX	1	1
XX =	00	lim a0		
	01	lim a0, a1		
	10	lim a1, a0		
	11	lim a1		
MUL y, (rN)		10000AXXX011mmnnn	1	1
MUL y, register		1000AXXX010rrrrr	1	1

Table 7. Instruction Coding Table (Continued)

Opcode		Code	Cycles	Words
MUL (rJ), (rI)		1101AXXX0jjiiwqq	1	1
MUL (rN), ##long immediate		1000AXXX000mmnnn	2	2
XXX =	000 001 010 011 100 101 110 111	mpy mpysu mac macus maa macuu macsu maasu		
MUL y, direct address		1110AXX0ddddddddd	1	1
XX =	00 01 10 11	mpy mac maa macsu		
mpyi		00001000vvvvvv	1	1
msu (rN), ##long immediate		1001000A110mmnnn	2	2
msu (rJ), (rI)		1101000Aljjiiwqq	1	1
tstb (rN)		1001bbbb001mmnnn	1	1
tstb register		1001bbb000rrrr	1	1
tstb direct address		1111bbbbddddddd	1	1
shfc		1101ab101AB0cccc	1	1
ab is the source				
shfi		1001ab1AB1vvvvvv	1	1
ab is the source vvvvvv = 6-bit immediate				
modb		011B11110fffcccc	1	1
fff =	000 001 010 011 100 101 110 111	shr shr4 shl shl4 ror rol clr reserved		
exp (rN), aX		1001100A010mmnnn	1	1
exp register, aX		10001000A010rrrrr	1	1
exp bX, aX		1001000A011000B	1	1
exp (rN), sv		10011100010mmnnn	1	1
exp register, sv		10010100010rrrrr	1	1

Table 7. Instruction Coding Table (Continued)

Opcode		Code	Cycles	Words
exp bx, sv		100101000110000B	1	1
mov register, register		010110RRRRRrrrrr	1	1
RRRRR is the destination				
mov ab, AB		1101ab101AB10000	1	1
ab is the source				
mov AB1, dvm		1101ab101000011	1	1
mov AB1, x		1101AB101000011	1	1
mov register, bX		0101111011brrrrr	1	1
mov register, mixp		01011111010rrrrr	1	1
mov register, (rN)		000110rrrrrmmnnn	1	1
mov mixp, register		01000111110rrrrr	1	1
mov repc, AB		110101001AB10000	1	1
mov dvm, AB		110101001AB10001	1	1
mov icr, AB		110101001AB10010	1	1
mov x, AB		110101001AB10011	1	1
mov (rN), register		000111rrrrrmmnnn	1	1
mov (rN), bX		1001100B110mmnnn	1	1
mov (sp), register		01000111111rrrrr	1	1
mov rN*, direct		0010nnn0dddddddd	1	1
msu (rJ), (rI)		1101000A1jjiiwqq	1	1
mov ABLH, direct		0011ABL0dddddddd	1	1
mov direct, rN*		011nnn00dddddddd	1	1
mov direct, AB		011AB001dddddddd	1	1
mov direct, ABLH		011ABL10dddddddd	1	1
mov direct, aXHeu		011A0101dddddddd	1	1
mov direct, sv		01101101dddddddd	1	1
mov sv, direct		011111101dddddddd	1	1
mov [##direct add.], aX		1101010A101110	2	2
mov aX1, [##direct add]		1101010A101111..	2	2
mov ##long immediate, register		0101111.000rrrrr	2	2
mov ##long imm., bX		0101111B001	2	2
mov #short, aX1		001A0001vvvvvvvv	1	1
mov #short, aXh		001A0101vvvvvvvv	1	12
mov #shotr, rN*		001nnn11vvvvvvvv	1	1
mov #short, ext 0-3		001X1X01vvvvvvvv	1	1

Table 7. Instruction Coding Table (Continued)

Opcode		Code	Cycles	Words
XX =	00 01 10 11	ext0 ext1 ext2 ext3		
mov #short, sv		00000101vvvvvvvv	1	1
mov register, icr		010011111.rrrrr	1	1
mov #immediate		01001111110.vvvvv	1	1
vvvvv = 5-bit immediate				
mov (rb + #offset7), aX		110110A1ooooooo	1	1
move aX1, (rb + #offset7)		1101110A1ooooooo	1	1
mov (rb + ##offset), aX		1101010A100110..	2	2
movp (aX), register		0000000001Arrrrr	3	1
movp (rN), (rI)		0000011iiqqmmnnn	3	1
movs register, AB		000000010abrrrrr	1	1
movs (rN), AB		000000011abmmnnn	1	1
movs direct, AB		011AB011ddddddddd	1	1
movsi rN*, AB		0100nnn01ABvvvvv	1	1
vvvvv = 5-bit immediate				
movr register, aX		1001110A110rrrrr	1	1
movr (rN), aX		1001110A11mmnnn	1	1
push register		01011110010rrrrr10	1	1
push ##long immediate		0101111101	2	2
pop register		01011110011rrrrr	1	1
swap swap-options		0100100110..swap	1	1
Swap 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101		Operation a0 < –> b0 a0 <–> b1 a1 <–> b0 a1 <–> b1 a0 <–> b0 and a1 –> b1 a0 <–> b1 and a1 –> b0 a0 –> b0 and –> a1 a0 –> b1 –> a1 a1 –> b0 –> a0 a1 –> b0 –>a0 b0 –> a1 –> b1 b0 –> a1 –> b1 b1 –> a0 –> b0 b1 –> a1 –> b0		
banke		010010111...bank	1	1

Table 7. Instruction Coding Table (Continued)

Opcode		Code	Cycles	Words
bank = r0, r1, r4, cfgi Ex.: bank = 0101 means bank switch of r1 and cfgi				
movd (rI), (rJ)		01011111jjiiwqq	4	1
rep register		00001101...rrrrr	1	1
rep #short		00001100vvvvvvvv	1	1
bkrep #short		01011100vvvvvvvv	2	2
bkrep register		01011101000rrrrr	2	2
break		1101001111	1	1
br		010000011000cccc	2/3	2
brr		01010ooooooocccc	2	2
call		010000011100cccc	2/3	2
callr		00010ooooooocccc	2	1
calla		1101010A10000000	3	1
ret		010001011000cccc	2/3	1
retd		1101011110	1	1
reti		0100010111fcccc	2/3	1
f =	0 1	Do not do context switching Do context switching		
retid		1101011111000000	1	1
cntx		11010011100f0000	1	1
f =	0 (s) 1 (r)	Store shadows of context switching Restore shadows of context switching		
rets		00001001vvvvvvvv	3	1
nop		0000000000000000	1	1
modr		000000001fmmnnn	1	1
f =	0 1	Don’t disable modulo Disable modulo		
eint		0100001110000000	1	1
dint		0100001111000000	1	1
trap		0000000000100000	1	1
load page		00000100vvvvvvvv	1	
load modX		0000x01vvvvvvvv	1	
x = vvvvvvvv = 9-bit immediate	0 1	load modi load modj		
load stepX		11011x111vvvvvv	1	

Table 7. Instruction Coding Table (Continued)

Opcode		Code	Cycles	Words
x = vvvvvvv = 7-bit immediate	0 1	load stepi load stepj		
load ps		01001101100000vv	1	1
vv = 2-bit immediate				

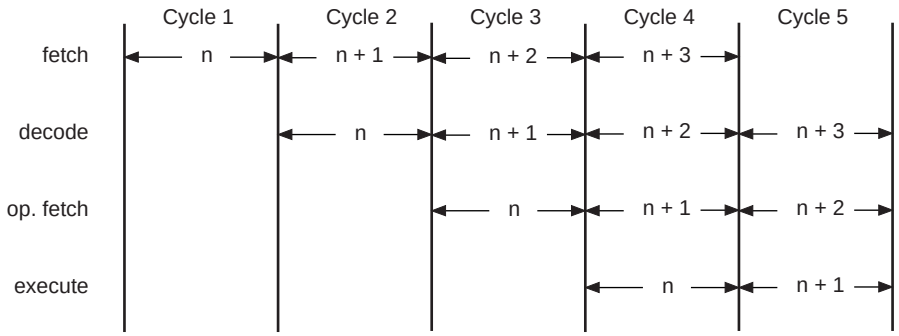
Pipeline Method

The program controller implements a four-level pipeline architecture. In the operation of the pipeline, concurrent fetch, decode, operand fetch and execution occur. Thus, during a given cycle up to four different instructions are being processed, each at a different stage of pipeline. The pipeline is an “interlocking” type.

Figure 10 shows the pipeline operation for sequential single-cycle instructions: The instruction n is executed in cycle four. During this cycle instruction n + 3 is pre-fetched, instruction n + 2 is decoded and the operand needed at instruction n + 1 is fetched.

For multiple-cycle instructions, the pipeline structure may differ but these details are beyond the scope of this document.

Figure 10. Pipeline Operation



Reset Operation

The reset line of the DSP subsystem is fully controlled by the ARM7TDMI core through the bits RA/RB in the SIAP_MD register.

In order to put the DSP subsystem in a correct reset state, some constraints must be respected when asserting/de-asserting the reset signal:

- The OakDSPCore must be held in a reset condition for a minimum of 12 OakDSPCore clock cycles. On removal of the reset, the OakDSPCore immediately starts the execution from location 0x0000 in program memory. In a typical case, the program memory will have been pre-loaded before the removal of the reset.
- The ARM7TDMI and OakDSPCore processors each have their own PLL, so at power-on each processor has its own indeterminate lock period. To guarantee message synchronization between the ARM7TDMI and the OakDSPCore, the ARM7TDMI must assert/de-assert the DSP subsystem reset signal three times.

Program Memory Upload

When the reset is active, the program memory becomes visible from the ARM7TDMI point of view. During reset, the ARM7TDMI has the ability to write the full program memory in order to upload a DSP application. The DSP application is executed as soon as the reset condition is exited.

Interrupts

The OakDSPCore provides four hardware interrupts: three maskable interrupts (INT0, INT1, INT2), and one non-maskable interrupt (NMI). One software interrupt (TRAP) also exists and can be used to mimic hardware interrupt operation by software.

When one of the interrupts NMI, INT0, INT1, INT2 is accepted, the core uses one stack level to store the program counter. Additionally, the core can also perform automatic context switching to save/restore critical parameters. For details, refer to “Interrupt Context Switching” on page 27. Then, the PC is automatically loaded with a predefined address corresponding to the interrupt service routine location.

Table 8 summarizes the predefined interrupt service routine addresses and the interrupt priorities. Since the NMI is not used in the AT75C, its operation is not discussed here.

Table 8. Interrupt Service Routine Addresses and Interrupt Priorities

Interrupt Service Routine Location	Interrupt Name and Function	Priority
0x0000	RESET	1 – highest
0x0002	TRAP/BI - Software interrupt	2
0x0004	NMI – Not used in the AT75C	3
0x0006	INT0 – Indicates that the dual-port mailbox requires service	4
0x000E	INT1 – Indicates that the codec interface requires service	5
0x0016	INT2 – Not used in the AT75C	6 – lowest

INT0, INT1, INT2 Operation

INT0, INT1 and INT2 are individually maskable interrupts. A maskable interrupt is accepted when:

- the OakDSPCore is in an interruptable state (see “Interrupt Latency”)
- the global Interrupt Enable bit within ST0 is set
- the corresponding Interrupt Mask bit (IM0, IM1 or IM2) is set

When a maskable interrupt is accepted, the OakDSPCore performs the following:

- SP - 1 –> SP
  The stack pointer is pre-decremented.
- PC –> (SP)
  The program counter is saved into the stack.
- ##ISR address –> PC
  Control is given to the interrupt service routine.

It should be noted that the IMx mask remains unaffected.

When the interrupt request is acknowledged by the OakDSPCore, the IE bit within ST0 is reset, disabling other maskable interrupts from being serviced. Additional pending interrupts are serviced once the program re-enables (sets) the IE bit.

Return from the interrupt service routine is down through the instructions ret, retd, reti, or retid. The instructions reti and retid set the IE flag, allowing pending interrupts to be serviced. When using the ret or retd instructions, the IE bit must be set explicitly to enable interrupts again.

Interrupt priority is used to arbitrate simultaneous interrupt requests. INT0 has the highest priority, and INT2 the lowest. Nesting is supported if IE is enabled by the current interrupt service routine. The priority between INT0, INT1 and INT2 is significant only if more than one interrupt is received at the same time. The priority scheme is also applied when the IE bit is cleared for some time and more than one maskable interrupt request is received.

When a maskable interrupt INTx is accepted, the corresponding IPx bit in ST2 is set. This can be used in applications that use interrupt polling while disabling (via the IE bit) the automatic response to interrupt requests.

TRAP/BI Operation

TRAP is a software interrupt used to mimic hardware interrupts, while BI is a hardware breakpoint interrupt dedicated to the on-chip emulation module (OCEM) operation. Both TRAP and BI share the same interrupt vector.

During the execution of the TRAP/BI service routine, all other interrupts are disabled.

A TRAP/BI is accepted while the OakDSPCore is in an interruptible state, as stated in “Interrupt Latency”. When the OakDSPCore accepts the TRAP/BI, the following actions are taken:

- SP - 1 –> SP
  The stack pointer is decremented.
- PC –> (SP)
  The program counter is saved into the stack.
- PC –> DVM
  The program counter is saved into the Data Value Match register.
- 0x0002 –> PC
  Control is given to the TRAP service routine.

The TRAP instruction should not be used within another TRAP/BI interrupt service routine.

The TRAP/BI service routine must end with a reti or retid instruction.

The on-chip emulation module (OCEM) uses the BI to provide emulation capability within the OakDSPCore. When BI is used as hardware interrupt, note the following differences between BI and other interrupts:

- The latency is two instruction cycles (compared to one in other interrupts)
- BI has the highest priority
- BI has no masking option
- BI has no option for automatic context switching

Interrupt Latency

The INT0, INT1, INT2 interrupts latency is one machine cycle, assuming that the OakDSPCore is in an interruptible state. The BI latency is two machine cycles, assuming that the OakDSPCore is in an interruptible state. During non-interruptible states, the OakDSPCore will not service the interrupt, but will continue to execute instructions. The interrupt will be accepted once the OakDSPCore exits this state.

Non-interruptible states are:

- during reset
- during the first three instruction fetches after de-activation of the reset. BI is an exception.
- during the execution of multi-cycle instructions
- when no clock is provided to the DSP subsystem
- during a nested repeat loop execution

For specific instructions where interrupts are delayed after their execution, refer to the section “More on Interrupt Latency” on page 167.

On-chip Emulation Module (OCEM)

The OCEM is a standalone module, which is used to provide software debug facilities such as hardware emulation and program flow trace buffering. The OCEM functions include:

- Program address breakpoints with separate match condition counters
- Data address breakpoint
- Data value breakpoint
- Combined data address and data value breakpoint
- Single-step
- Break-on-branch
- Break-on-interrupt
- Break-on-block repeat loop
- Program flow trace buffer

OCEM Operation

The OCEM uses the OakDSPCore breakpoint interrupt (BI) mechanism to implement its emulation functions. The emulation functions include two main tasks:

- Breakpoint generation
- Program flow tracing

The breakpoint generation is done due to a predefined condition that is programmed into the OCEM registers set. Once a condition is met, the OCEM activates the BI mechanism causing the OakDSPCore to suspend any action and jump into the BI service routine. This routine can be a debug monitor or any function allowing code behavior analysis.

Program flow buffering includes dynamic recording of the instructions' addresses, which cause a discontinuity in the code sequence. These addresses are kept in a FIFO within the OCEM block and used afterwards to reconstruct the complete program flow graph.

Program Address Breakpoint Operation

The program address breakpoint condition can be programmed through three program address breakpoint registers (PABP1, PABP2, PABP3) and three corresponding program address breakpoint counters (PABC1, PABC2, PABC3).

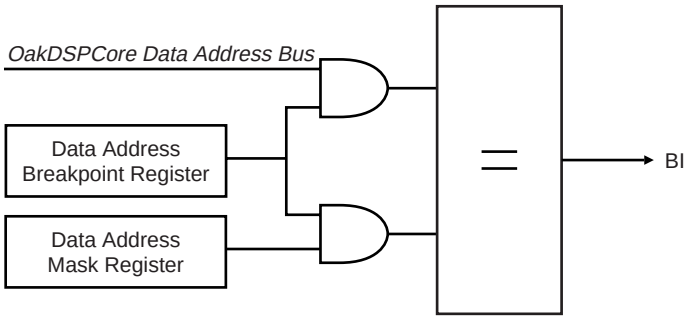
When the OCEM senses a match between the program address from the OakDSPCore and one of the addresses written in the PABPx register, the counter PABCx is decremented. If the counter value is zero, a BI request is issued. If the user wishes to have a breakpoint every time the OakDSPCore fetches a given program address, the corresponding PABC counter should be set to 1 (0 also gives the same behavior). If the user wishes to have a breakpoint only at the nth occurrence, the counter has to be set to n.

The program address breakpoint can be disabled by writing 0 to the PIE/P3E bit within the OCEM STATUS0 register. The BI service routine can check the P1, P2 and P3 bits to discriminate which address match has caused the BI to be triggered.

Data Address Breakpoint Operation

The data address breakpoint is initiated upon a match between the OCEM data address breakpoint (DABP) register and the OakDSPCore data address bus. The data address mask (DAM) register allows expansion of the data address breakpoint to an address space rather than one single address. Figure 11 describes this behavior.

Figure 11. Data Address Breakpoint Logic



The data address breakpoint can be enabled for read or write transactions. It is done by setting the bit DARE (for breaking on read) or DAWE (for breaking on write) in the OCEM MODE register. Detection of a data address match condition is indicated in the DABP bit within the OCEM STATUS0 register.

For more information on data address breakpoint latency, refer to “Data Address/Value Breakpoint Latency”.

Data Value Breakpoint Operation

Data value breakpoint is initiated upon detection of a data match between the OakDSPCore internal data bus and the content of the data value match (DVM) register. It should be noted that the data value breakpoint procedure is initiated only due to a matched data during memory transactions. There is an option to detect a combined data address and value condition. It is enabled by the bit CDVAE within the OCEM MODE register. An indication for this break point type is in the CDVA bit within the OCEM STATUS0 register. Note that if one wants to break on the combined condition, it has to set the CDVAE bit together with DARE and DVRE bits for read transaction and DAWE and DVWE for write transactions.

Note that the data address as well as data value breakpoint is being initiated after the transaction is completed. This is opposed to program address breakpoint, where the breakpoint occurs during the fetch period, i.e., prior to the instruction execution.

For more information on data value breakpoint latency, refer to “Data Address/Value Breakpoint Latency”.

Data Address/Value Breakpoint Latency

Since the data is detected at the operand fetch of the instruction, two more instructions are already in the pipe and issued for decode, thus they are executed before the interrupt is served. This means that for data value breakpoints there is a latency of two instructions and the user will see the status of the OakDSPCore only two instructions after the event has happened.

For data address breakpoint, the latency could have been reduced to one instruction. However, in order to enable combined DABP and DVBP, the trigger of BI request is delayed. Thus, the latency is also two instructions.

Single-step Operation

The single-step operation is controlled by the SSE bit in the MODE0 register. If this bit is set when the program returns from the BI service routine, the BI procedure is re-triggerred once the OakDSPcore has executed a single instruction.

Break-on-branch Operation

The OCEM allows the user to trigger the BI mechanism upon pipe-break events such as branch, call, interrupt or looping. The user can choose which event of the following list would cause a breakpoint:

- BR, BRR, CALL, CALLR, CALLA, RET, RETI, RETD, RETID, RETS, MOV to PC
- Interrupt service start (excluding TRAP)
- Transition from the last back to the first address within a block-repeat loop

Program Flow Trace Buffer

The OCEM contains a 16-stage program flow trace buffer. The buffer dynamically keeps non-linear program addresses. This means only those program addresses that contain an instruction that causes a discontinuity in the program flow. The whole program flow graph can be reconstructed by taking those non-linear addresses and filling in the gaps using the object code.

The advantage of using the “condensed” trace buffer resides in the number of addresses that can effectively be kept in the buffer.

The method behind the condensed trace buffer is composed of four principles:

1. Only branches that have effectively been taken are recorded. For example, a conditional branch not causing code discontinuity due to a false condition will not be recorded.
2. If the non-sequential instruction has a destination address explicitly coded in the program (e.g., BR, BRR, CALL, CALLR), only the instruction address is recorded.
3. If the non-sequential instruction has a destination address not explicitly coded in the program (e.g., CALLA, RET or MOV to PC), the instruction address as well as the target address is recorded.
4. Interrupts are treated as non-sequential instructions. However, they can arrive anywhere and, although the target destination is known, the source (the last instruction address being executed prior to the branch to the interrupt vector) cannot be derived easily. Therefore, for the interrupt case, the source address (i.e., the last instruction address before servicing the interrupt) as well as the destination address (the vector address) is recorded in the trace buffer.

There is an option for trace buffer full breakpoint by setting the bit TBF in the MODE register. The bit TBF indicates the cause of the breakpoint. If the TBF bit is disabled, the trace buffer would contain the last 16 non-sequential recordings. Once the whole buffer is read, it is filled with all 1s.

The trace memory mechanism contains one extra bit per trace stage that is used to tag two-word instructions.

Note that:

- “Non-sequential” addresses within the breakpoint handler are not recorded.
- The PROM address in MOVP instruction and the PRAM address in MOVD instruction are not recorded.
- Non-sequential fetches that are due to TRAP instruction (as well as TRAP/BI breakpoint) are not recorded.

Trace Reading and Decoding Algorithm

Trace reading and decoding requires several operations that must be done sequentially.

In order to decode the addresses written in the trace buffer correctly, reading should take place in the sequence below:

1. Read the tag (bit 0, TREI, in STATUS1 register).
2. Read the PFT register, which holds the most recent entry in the trace buffer.

The read operation updates the PFT register and the tag so as to reflect the next entry of the trace buffer.

This operation (reading TREI and trace one by one) should be repeated 16 times in order to read the whole buffer.

Decoding of the program flow should start from the last address that was read from the buffer (e.g., last entry to the buffer) and proceed backwards.

The addresses that are saved in the trace buffer can be single entry with the tag bit equal to 1, or double entry with the tag bits equal to 0 for both entries.

By using this tag bit, the user can differentiate between single and double entries of the buffer.

Completion of the program flow is done by filling the sequential instructions from the program list between the branches.

Notice that sometimes one or more lines at the beginning of the buffer (addresses first read from the buffer) may contain only 1s, including the tag bit. It means that those lines were not used since the previous buffer's reading and do not contain any information.

OCEM Programming Mode

STATUS0 (0xF7FF)

Reset Value: 0x0000

15	14	13	12	11	10	9	8
SFT	IL	TBF	INT	BR	–	–	–

7	6	5	4	3	2	1	0
PA3	PA2	PA1	ABORT	EREG	CDVA	DA	DV

- **SFT**

When set, this bit indicates that the current breakpoint has been caused by a TRAP instruction.

- **IL**

Indicates the detection of an illegal breakpoint. This breakpoint is activated if the user program tries to access the mailbox space or the OCEM registers, not the breakpoint handler.

- **TBF**

When set, this indicates that the current breakpoint has been caused by the program flow trace buffer being full.

- **INT**

When set, this bit indicates that the current breakpoint has been caused by an accepted interrupt.

- **BR**

Indicates the detection of a branch or block repeat breakpoint.

- **PA3**

Indicates the detection of a program address breakpoint due to a match with PABP3.

- **PA2**

Indicates the detection of a program address breakpoint due to a match with PABP2.

- **PA1**

Indicates the detection of a program address breakpoint due to a match with PABP1.

- **ABORT**

Indicates the detection of a breakpoint due to an external event.

- **EREG**

Indicates the detection of a breakpoint due to an external register transaction.

- **CDVA**

Indicates the detection of a breakpoint due to a combine data value and data address match.

- **DA**

Indicates the detection of a breakpoint due to a data address match.

- **DV**

Indicates the detection of a breakpoint due to a data value match.

STATUS1 (0xF7FE)

Reset Value: Only MVD reset to 0.

15	14	13	12	11	10	9	8
DBG	BOOT	ERR	MVD	–	–	–	–

7	6	5	4	3	2	1	0
–	–	–	–	–	–	–	TREI

- **DBG**

Indicates debug mode. Activated as a result of an indication on the CA1 pin at res rising edge.

- **BOOT**

Indicates boot mode. Activated as a result of an indication on the CA0 pin at res rising edge.

- **ERR**

Indicates the detection of a user reset during the execution of a breakpoint service routine. In this case, the Oak debugger may not follow the user commands.

- **MVD**

Indicates the detection of a MOVD instruction.

- **TREI**

Acts as a tag bit for the program flow trace. When cleared, it indicates that the current trace entry (the one that is at the trace top) has to be combined with the next trace entry for the proper expansion of the user program.

MODE0 (0xF7FD)

Reset Value: 0x0000

15	14	13	12	11	10	9	8
SSE	ILLE	BKRE	TBFE	INTE	BRE	P3E	P2E

7	6	5	4	3	2	1	0
P1E	EXTRE	EXTWE	CDVAE	DARE	DAWE	DVRE	DVWE

- **SSE**

When set, this bit enables single-step operation.

- **ILLE**

Enables the breakpoint on an illegal condition (trying to access the mailbox space not through the TRAP handler).

- **BKRE**

When set, this bit enables the breakpoint when returning to the beginning of a repeat loop.

- **TBFE**

Enables the breakpoint as a result of program flow trace buffer full.

- **INTE**

Enables the breakpoint upon the detection of an interrupt service execution.

- **BRE**

Enables the breakpoint every time the program jumps instead of executing the next sequential instruction.

- **P3E**

Enables the program breakpoint 3. The breakpoint is activated upon a match on the address specified at PABP3.

- **P2E**

Enables the program breakpoint 2. The breakpoint is activated upon a match on the address specified at PABP2.

- **P1E**

Enables the program breakpoint 1. The breakpoint is activated upon a match on the address specified at PABP1.

- **EXTRE**

Enables the breakpoint as a result of an external register read transaction.

- **EXTWE**

Enables the breakpoint as a result of an external register write transaction.

- **CDVAE**

Enables the breakpoint as a result of simultaneous data address and data value match.

- **DARE**

Enables the breakpoint as a result of a read transaction where the address matches with the value in DABP.

- **DAWE**

Enables the breakpoint as a result of a write transaction where the address matches with the value in DABP.

- **DVRE**

Enables the breakpoint as a result of a read transaction where the data value matches with the value in the OakDSPCore's DVM register.

- **DVWE**

Enables the breakpoint as a result of a write transaction where the data value matches with the value in the OakDSPCore's DVM register.

DABP (0xF7FB)

15	14	13	12	11	10	9	8
Data Address Breakpoint Value							

7	6	5	4	3	2	1	0
Data Address Breakpoint Value							

DAM (0xF7FA)

15	14	13	12	11	10	9	8
Data Address Mask Value							

7	6	5	4	3	2	1	0
Data Address Mask Value							

PABC3 (0xF7F9)

15	14	13	12	11	10	9	8
–	–	–	–	–	–	–	–

7	6	5	4	3	2	1	0
–	–	Program Address Breakpoint Counter 3					

PABC2 (0xF7F8)

15	14	13	12	11	10	9	8
–	–	–	–	–	–	–	–

7	6	5	4	3	2	1	0
–	–	Program Address Breakpoint Counter 2					

PABC1 (0xF7F7)

15	14	13	12	11	10	9	8
–	–	–	–	–	–	–	–

7	6	5	4	3	2	1	0
–	–	Program Address Breakpoint Counter 1					

PABP3 (0xF7F3)

15	14	13	12	11	10	9	8
Program Address Breakpoint 3							

7	6	5	4	3	2	1	0
Program Address Breakpoint 3							

PABP2 (0xF7F2)

15	14	13	12	11	10	9	8
Program Address Breakpoint 2							

7	6	5	4	3	2	1	0
Program Address Breakpoint 2							

PABP1 (0xF7F1)

15	14	13	12	11	10	9	8
Program Address Breakpoint 1							

7	6	5	4	3	2	1	0
Program Address Breakpoint 1							

PFT (0xF7F0)

15	14	13	12	11	10	9	8
Program Flow Trace							

7	6	5	4	3	2	1	0
Program Flow Trace							

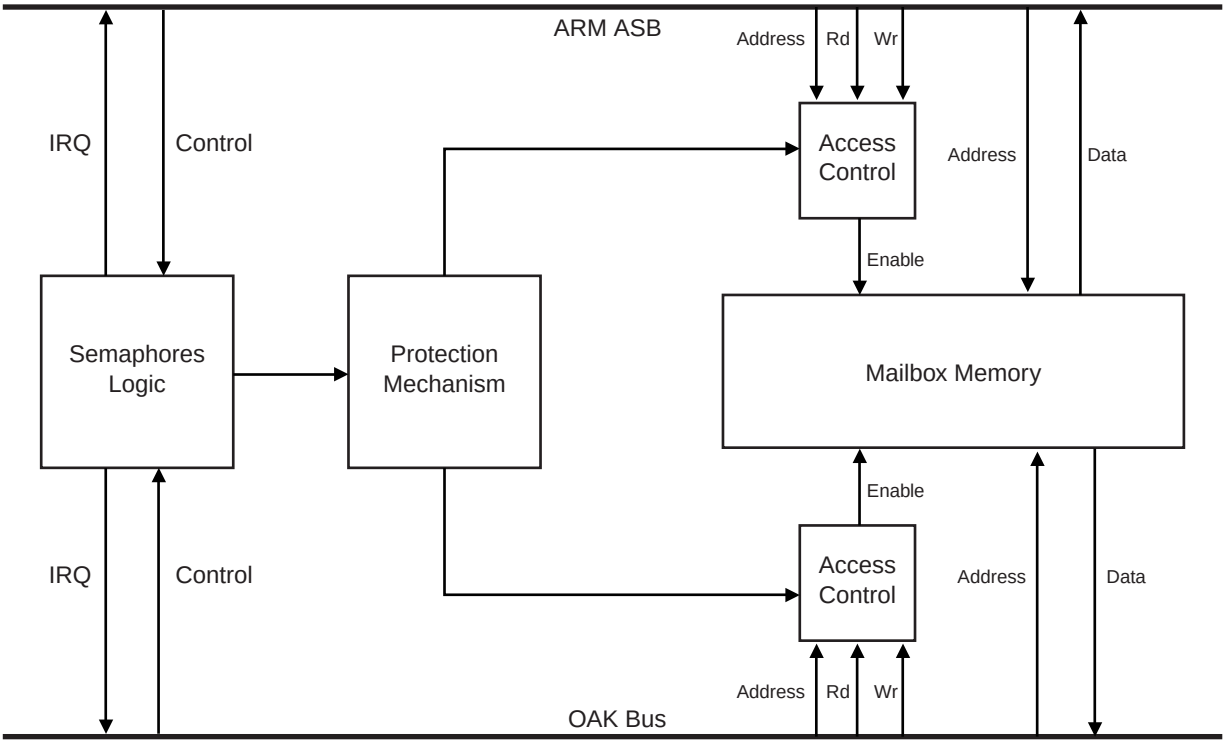
Dual-port Mailbox

Communications between the asynchronous ARM7TDMI and OakDSPCore are via the dual-port mailbox (DPMB). It is assumed that each processor is running asynchronously and that the coherency of communications is maintained by a robust software.

The DPMB consists of 512 bytes of DPRAM and some memory-mapped registers that configure the DPMB and act as semaphores. The DPMB sits on the ARM7TDMI ASB bus and can be accessed by memory-mapped operations. Similarly, the DPMB also sits on the OakDSPCore data bus and can be accessed by memory-mapped operations.

Messages can be sent between the two processors by the sending processor writing data to the DPRAM within the DPMB and signalling via the semaphores. This data is subsequently read by the recipient processor. Individual mailboxes are bi-directional with access permission controlled by the semaphores. The message-passing protocol can be configured for each mailbox as interrupt-driven or polled, in each direction.

Figure 12. Dual-port Mailbox Block Diagram



Dual-port RAM

The major component of the DPMB is a dual-port RAM (DPRAM). It consists of 256 by 16 bits of dual-port static RAM. The DPRAM is divided into eight equal mailboxes, each with its own semaphore register. The hardware implements locks such that both processors never have write access to the same mailbox region. This protection mechanism is based upon the values of the semaphores, which must be maintained by the software.

If a processor attempts to access a mailbox to which it does not have semaphore permission, then this access will be ignored.

The DPMB supports word access from the ARM side, provided the address is word-aligned, and half-word access from both the ARM and Oak sides, provided the address is half-word-aligned. All other accesses will result in a data abort being issued. The DPMB does not decode protection information carried in AMBA™ output BPROT[1:0] and, as a result, does not support Thumb® compiled code.

In order to reduce hardware, the minimum number of address bits are used in decoding register and mailbox addresses. This results in address aliasing. For the ARM side, only address bits ba[5:2] and ba[9] are used in the register decodes and bits ba[8:5] in the mailbox decodes. On the Oak side, only address bits dxap[2:0] are used in the register decode and bits dxap[7:4] are used in the mailbox decode.

Semaphore Operation

The DPMB supports semaphore registers to facilitate asynchronous communication between two processors. Each of the eight mailboxes has its own memory-mapped semaphore register. This avoids any need for complex read-modify-write operations. Each semaphore is configured to control message passing in a single direction, i.e., from the ARM to the Oak when the DPMB ARM-to-Oak flag is set high. When this flag is low, then the direction of transfer is Oak to ARM.

A semaphore can be configured to support either interrupts or polling in either direction. The ARM Interrupt Enable (AIE) flag determines if interrupts shall be raised to the ARM when an associated semaphore operation occurs. Similarly, the Oak Interrupt Enable (OIE) flag determines if an interrupt is raised to the Oak when an associated semaphore operation occurs.

There are no hardware-read restrictions to any semaphore for either processor. This allows a semaphore to be polled freely. However, the software is expected to maintain message-level synchronization and not attempt to write to the same semaphore at the same time. Because the two processors may run asynchronously, the semaphores are re-synchronized to the clock domains of both processors and hence, cycle accuracy is not maintained.

At reset, all semaphores are reset low.

Each semaphore can be set or cleared by either processor by performing a write operation to the semaphore register. A set operation (write high) sets the semaphore register to high and a clear operation (write low) sets the semaphore low. However, a semaphore operation can also raise or clear interrupts, depending on which processor performs the operation and which processor has been enabled as the sender processor. The semantics of semaphore operations are presented in Table 9.

Table 9. Semaphore Operations Semantics

Operation	ARM to Oak High	ARM to Oak Low
ARM Set	Set Semaphore Raise Oak Interrupt	Set Semaphore Clear ARM Interrupt
ARM Clear	Clear Semaphore Clear ARM Interrupt	Clear Semaphore Clear ARM Interrupt Raise Oak Interrupt
Oak Set	Set Semaphore Clear Oak Interrupt	Set Semaphore Raise ARM Interrupt
Oak Clear	Clear Semaphore Clear Oak Interrupt Raise ARM Interrupt	Clear Semaphore Clear Oak Interrupt

DPMB Register Map

Table 10. ARM Registers

Register Address (Offset from Base) [1]	Register	Description	Mailbox Base Address (Offset from Base) [1]				Access
			Config 0	Config 1	Config 2	Config 3	
0x200	DPMBS0	Mailbox Semaphore 0	0x000	0x000	0x000	0x000	Read/write
0x204	DPMBS1	Mailbox Semaphore 1	0x040	0x0E0	0x080	–	Read/write
0x208	DPMBS2	Mailbox Semaphore 2	0x080	0x100	0x100	–	Read/write
0x20C	DPMBS3	Mailbox Semaphore 3	0x0C0	0x120	0x140	–	Read/write
0x210	DPMBS4	Mailbox Semaphore 4	0x100	0x160	0x180	–	Read/write
0x214	DPMBS5	Mailbox Semaphore 5	0x140	0x1A0	0x1A0	–	Read/write
0x218	DPMBS6	Mailbox Semaphore 6	0x180	0x1C0	0x1C0	–	Read/write
0x21C	DPMBS7	Mailbox Semaphore 7	0x1C0	0x1E0	0x1E0	–	Read/write
0x220	DPMBCC	Mailbox Configuration	(0x200)				Read/write

Note: 1. Base address is 0xFA000000 for OakA and 0xFB000000 for OakB.

Table 11. Oak Registers

Register Address	Register	Description	Mailbox Base Address				Access
			Config 0	Config 1	Config 2	Config 3	
0xE800	DPMBS0	Mailbox Sempahore 0	0xE000	0xE000	0xE000	0xE000[1]	Read/write
0xE801	DPMBS1	Mailbox Semaphore 1	0xE020	0xE070	0xE040	–	Read/write
0xE802	DPMBS2	Mailbox Semaphore 2	0xE040	0xE080	0xE080	–	Read/write
0xE803	DPMBS3	Mailbox Semaphore 3	0xE060	0xE090	0xE0A0	–	Read/write
0xE804	DPMBS4	Mailbox Semaphore 4	0xE080	0xE0B0	0xE0C0	–	Read/write
0xE805	DPMBS5	Mailbox Semaphore 5	0xE0A0	0xE0D0	0xE0D0	–	Read/write
0xE806	DPMBS6	Mailbox Semaphore 6	0xE0C0	0xE0E0	0xE0E0	–	Read/write
0xE807	DPMBS7	Mailbox Semaphore 7	0xE0E0	0xE0F0	0xE0F0	–	Read/write

Note: 1. Mailbox 0 for Config 3 allows access to the entire DPRAM.

DPMB Semaphore Registers

The semaphore registers look the same from the ARM and the Oak sides.

31	30	29	28	27	26	25	24
–	–	–	–	–	–	–	–
23	22	21	20	19	18	17	16
–	–	–	–	–	–	–	–
15	14	13	12	11	10	9	8
–	–	–	–	–	–	–	–
7	6	5	4	3	2	1	0
–	–	–	–	–	OIS	AIS	Sem

- **Sem: Semaphore**

When low, the sender has read and write permission to the mailbox. The recipient has no permission to read or write the associated mailbox. When high, the recipient has read and write permission to the mailbox and the sender has no permission.

- **AIS: ARM Interrupt Status**

This flag indicates the value of the ARM interrupt flag. This is a read-only bit; any attempt to write to this bit will be ignored.

- **OIS: Oak Interrupt Status**

This flag indicates the value of the Oak interrupt flag. This is a read-only bit; any attempt to write to this bit will be ignored.

DPMB Configuration Register (DPMBCC)

The DPMB is configured by means of a memory-mapped register that sits on the ARM ASB bus. This register is not accessible by the Oak.

31	30	29	28	27	26	25	24
RESET	MB_CONFIG		–	–	–	–	–
23	22	21	20	19	18	17	16
OIE							
15	14	13	12	11	10	9	8
AIE							
7	6	5	4	3	2	1	0
ATO							

- **ATO[7:0]: ARM To Oak**

The value of this flag conditions the semantics of semaphore operation for the associated mailbox. When high, the ARM is the sender and the Oak is the recipient. When low, the Oak is the sender and the ARM is the recipient.

- **AIE[15:8]: ARM Interrupt Enable**
- When high, appropriate semaphore operations can raise an interrupt to the ARM. When low, interrupts are never raised by any semaphore operation.
- **OIE[23:16]: Oak Interrupt Enable**

When high, appropriate semaphore operations can raise an interrupt to the ARM. When low, interrupts are never raised by any semaphore operation.

- **MB_CONFIG[30:29]: Mailbox Configuration**

Selects one of four possible mailbox configurations. Refer to Table 10.

- **RESET**

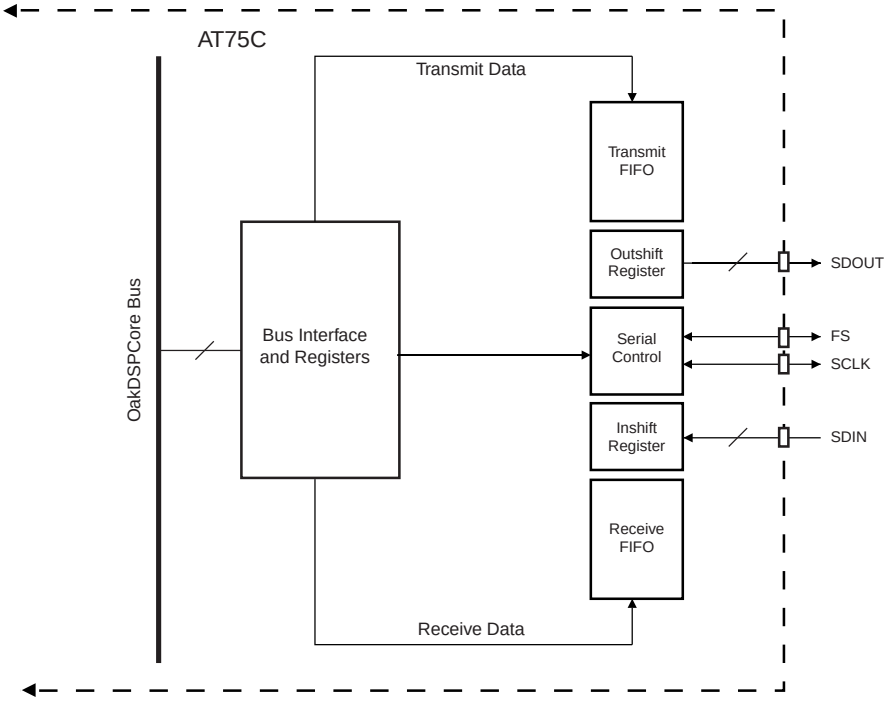
When a high is written to this bit, the DPMB is reset to its initial state, ready for the configuration to be set.

Codec Interface

In the AT75C product family, the DSP subsystem implements a flexible serial port interface that provides a full duplex bi-directional communication with external serial devices such as codecs. The codec interface signals are directly compatible with many industry-standard codecs and other serial devices.

See Figure 13 for a block diagram of the codec interface.

Figure 13. Codec Interface Block Diagram



Codec Interface Registers

The codec interface operates through a set of memory-mapped registers. These registers are listed below. The first three are control registers, which impact the behavior of the interface. They are described in more detail in “Codec Control Register”. These are followed by a status register, which informs the software about the current status of the interface. The status register is described in more detail in “Codec Interface Status Information” on page 161. The two final registers hold the data sent/received to/from the codec device.

Table 12. Codec Interface Registers

Address	Register Name	Description	Access	Reset Value
0xEC00	CODCNT	Codec control register	R/W	0x0800
0xEC02	CODFRM	Codec frame control register	R/W	0x0096
0xEC04	CODSCK	Codec serial clock control register	R/W	0x401F
0xEC06	CODSTS	Codec status register	R	0x0000
0xEC08	CODTX	Codec transmit data register	W	0x0000
0xEC0A	CODRX	Codec receive data register	R	0x0000

During normal codec interface operation, the CODTX is typically loaded with data to be transmitted on the codec interface by the executing program, and its contents read automatically by the codec interface logic to be sent out when a transmission is initiated. The CODRX is loaded automatically by the codec interface logic with data received on the codec interface and read by the executing program to retrieve the received data.

Codec Interface Signals

Only four wires are needed to achieve synchronous bi-directional communications between the codec interface and an external codec. Those wires are described in Table 13.

Table 13. Codec Interface Wires Description

Signal Name	Direction	Description
SCLK	I/O	Serial clock
FS	I/O	Frame synchronization
SDIN	I	Serial Data In
SDOUT	O	Serial Data Out

The SCLK signal is the serial clock. All the other codec interface signals are synchronous to this clock. In particular, the SCLK is used as the shift clock for the codec interface transmit and receive shift registers. The SCLK signal can be delivered either by the external codec device (master mode) or by the AT75C itself (slave mode).

The FS signal is used to indicate the beginning of a transmission between the AT75C and the external codec device. Typically, the state of FS changes when the first bit of transmitted word is delivered on the SDIN/SDOUT lines. When the AT75C is connected to an external stereo codec, the FS signal can also be used to discriminate the right channel against the left channel. The FS signal is synchronous to SCLK. In master mode, it is sampled by the AT75C. In slave mode, it is generated by the AT75C.

SDIN is used to carry the serial bits from the external codec device into the AT75C codec interface. The value on SDIN is sampled by the AT75C each SCLK period.

SDOUT is used to carry the serial bits from the AT75C to the external codec device. A new bit is transmitted each SCLK period.

Codec Control Register

The codec interface contains a control register which configures the operation of the port. This register and the corresponding bit fields are described below.

CODCNT (0xEC00)

15	14	13	12	11	10	9	8
–	–	LOOP	FSC	SDOZ	FSIO	SCIO	IRO

7	6	5	4	3	2	1	0
IRS	IRF	IRE	ITO	ITF	ITE	REN	TEN

- **TEN**

Transmitter enable. When high, the transmitter is enabled.

- **REN**

Receiver enable. When high, the receiver is enabled.

- **ITE**

Transmitter empty interrupt enable. When high, this bit allows an interrupt to be triggered as soon as the transmit FIFO is empty.

- **ITF**

Transmitter full interrupt enable. When high, this bit allows an interrupt to be triggered as soon as the transmit FIFO is full.

- **ITO**

Transmitter overrun interrupt enable. When high, this bit allows an interrupt to be triggered when a transmit overrun condition occurs. Define transmit overrun condition.

- **IRE**

Receiver empty interrupt enable. When high, this bit allows an interrupt to be triggered as soon as the receive FIFO is empty.

- **IRF**

Receiver full interrupt enable. When high, this bit allows an interrupt to be triggered as soon as the receive FIFO is full.

- **IRS**

Receiver sample ready interrupt enable. When high, this bit allows an interrupt to be triggered interrupt when a sample has been received and is ready to be processed. This interrupt is independent of the receive FIFO operation.

- **IRO**

Receiver overrun interrupt enable. When high, this bit allows an interrupt to be triggered when a receive overrun condition occurs. Define receive overrun condition.

- **SCIO**

Serial clock direction. When high, the SCLK pin is an output, and the SCLK signal is derived according to the value in the CODSCLK register. When low, the SCLK pin is an input and is used as the serial bit clock.

- **FSIO**

Frame sync direction. When high, the FS pin is an output and the FS signal is derived according to the value of the CODFRM register. When low, the FS pin is an input and is used as the frame synchronization signal.

- **SDOZ**

SDOUT tri-state mode. When high, the SDO pin is put in high impedance when it does not carry significant data. When low, the SDO pin is always in low impedance, even when it does not carry significant data.

- **FSC**

Frame sync clock select. When high, the FS signal is derived from a division of the system clock. When low, the FS signal is derived from a division of the SCLK, independent of the source of SCLK.

- **LOOP**

Loopback mode. When high, the bitstream on SDO is internally copied on the incoming bitstream, overriding the values on SDI.

CODFRM (0xEC02)

15	14	13	12	11	10	9	8
DD[7:0]							

7	6	5	4	3	2	1	0
FD[7:0]							

- **FD[7:0]**

Frame divisor. This 8-bit number is the value by which to divide the serial clock SCLK to derive the frame sync signal FS.

- **DD[7:0]**

Serial data delay. This is the number of SCLK cycles by which to delay the start of the first data bit from FS.

CODSCK (0xEC04)

15	14	13	12	11	10	9	8
FPS[1:0]		WT[1:0]		DWS	FRL	RES	TES

7	6	5	4	3	2	1	0
SD[7:0]							

- **SD[7:0]**

Serial clock divisor. Value by which to divide the system clock to derive the SCLK output in slave mode:

	DSP System Clock
AT75C310	(Quartz Clock) x 2.5
AT75C220	(Quartz Clock) x 3.75

In master mode, this value is not significant.

- **TES**

Transmitter active edge of SCLK. When high, the data on SDO is shifted out on the falling edge of SCLK. When low, the data on SDO is shifted out on the rising edge of SCLK.

- **RES**

Receiver active edge of SCLK. When high, the data on SDI is sampled on the rising edge of SCLK. When low, the data on SDI is sampled on the falling edge of SCLK.

- **FRL**

Frame sync length. When high, the generated frame sync lasts one SCLK cycle. When low, the generated frame sync has a 50% duty cycle.

- **DWS**

Serial data word width select. When high, the transmit and receive shift registers are 32 bits wide. When low, the shift registers are 16 bits wide.

- **WT[1:0]**

Word type.

00: Single data mode. Each sample represents a sample value.
01: 1 control, 1 data
10: 1 control, 2 data
11: Stereo mode. Two data are transmitted in one sample period, the left data and the right data. The left and right are discriminated by the polarity of FS.

- **FPS[1:0]**

Frames per sample. This value indicates the expected number of serial data words to be transmitted and received during one sample period. This value defines the level reached with receive FIFO before the sample ready interrupt is signalled.

Codec Interface Status Information

A dedicated status register can inform the software about the current state of the codec interface. When the codec interface interrupt is unmasked, the status flags reflect the state of the codec interface interrupt sources. They can be used by the interrupt service routine to recognize the condition that triggered the interrupt request. When the codec interface interrupt is masked, the status bits can be polled.

The status bits are set on the active edge of the interrupt source, and are cleared upon register read.

The codec status register is detailed below.

CODSTS (0xEC06)

15	14	13	12	11	10	9	8
–	–	–	–	–	–	–	–

7	6	5	4	3	2	1	0
–	ROV	TOV	RXW	RXF	RXE	TXF	TXE

- **TXE**

Transmitter empty flag.

- **TXF**

Transmitter full flag.

- **RXE**

Receiver empty flag.

- **RXF**

Receiver full flag.

- **RXW**

Receiver sample ready flag.

- **TOV**

Transmit FIFO overrun.

- **ROV**

Receive FIFO overrun.

Codec Interface Transmit/Receive Registers

CODTX (0xEC08)

15	14	13	12	11	10	9	8
TxD[15:0]							

7	6	5	4	3	2	1	0
TxD[15:0]							

- **TxD[15:0]**

16-bit data word to be added to the location being pointed to within the transmit FIFO. Depending on the configuration of the codec interface, this FIFO location may require a 32-bit word. In this case, the data will be written in two cycles and controlled by the transmit pointer logic.

CODRX (0xEC0A)

15	14	13	12	11	10	9	8
RD[15:0]							

7	6	5	4	3	2	1	0
RD[15:0]							

- **RD[15:0]**

16-bit data word received from the external codec device. This value is latched from the location being pointed to within the receive FIFO. Depending on the configuration of the codec interface, this FIFO location may deliver a 32-bit word. In this case, the data will be fetched in two cycles and controlled by the receive pointer logic.

Codec Interface Operation

This section describes the standard operation of the codec interface, reviewing the different modes available.

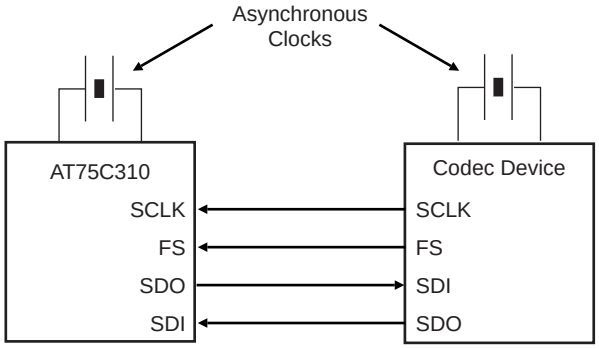
Master Mode

The external codec device and the codec interface are said to be in the master mode when the external codec device delivers the serial clock (SCLK) and the frame sync (FS) information. This mode allows the AT75C and the codec device to be fully asynchronous (i.e., with separate clock sources) because the synchronization is maintained by the SCLK and FS signals. This insures that the codec device will deliver a new sample when it is available and will accept a new sample when it is able to, according to its own conversion timing.

The master mode is recommended because it leads to robust synchronization between the codec device and the application software and does not require accurate timing calculations.

Figure 14 illustrates an AT75C and a codec device connection in master mode.

Figure 14. Connection of a Codec Device in Master Mode



Note that programming FS and SCLK to be in opposite directions is not recommended.

Slave Mode

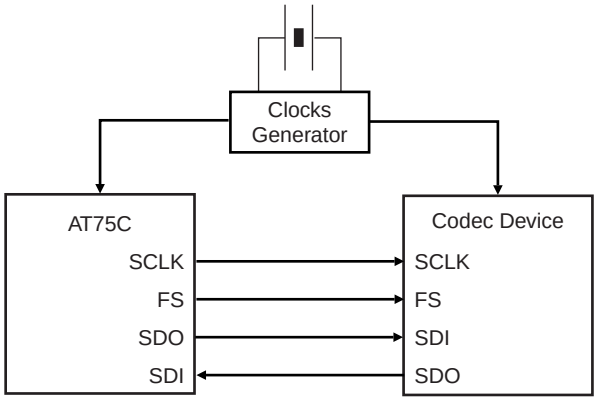
In slave mode the serial clock (SCLK) and the frame sync (FS) signals are delivered by the AT75C. To achieve data synchronization between the AT75C and the codec device, the following rules must be respected:

- The AT75C system clock and the codec device clock must be kept in sync.
- The SCLK generator in the AT75C must be programmed to generate a serial clock with a frequency compatible with the codec device.
- Because the AT75C initiates the transfers, the FS generator must be accurately programmed so that data are exchanged when the codec device is ready to accept/deliver them.

The slave mode operation is not recommended in a typical design because it has strong requirements on the overall system clock strategy and requires fine tuning of the AT75C codec interface. However, it allows the accommodation of some industry-standard codec devices provided that the rules described above are followed.

Figure 15 illustrates an AT75C and a codec device connection in slave mode.

Figure 15. Connection of a Codec Device in Slave Mode



Note that programming SCLK and FS in opposite directions is not recommended.

Serial Clock Configuration

The direction of the serial clock SCLK is controlled by the SCIO bit in CODCNT. When SCIO is low, the codec interface is said to be in master mode; when SCIO is high, the codec interface is said to be in slave mode.

Serial Clock in Master Mode

When SCIO is low, the SCLK pin is an input of the AT75C.

The SCLK signal is internally clocked and the resynchronized signal is used as the shift clock for the transmit and receive shift registers. When SCIO is low, the internal SCLK generator is disabled and the field SD[7:0] in CODSCK is not used.

The active edge of the serial clock can be programmed independently for the transmitter and the receiver. When the TES bit in CODSCK is low, the transmitted bits are shifted out on the rising edge of XSCLK. When TES is high, the transmitted bits are shifted out on the falling edge of XSCLK.

When the RES bit in CODSCK is low, the received bits are latched on the falling edge of XSCLK.

Serial Clock in Slave Mode

When SCIO is high, the SCLK pin is an output of the AT75C. An internal clock generator is used to produce the SCLK signal.

The active edge of SCLK can be programmed independently for the transmitter and the receiver. When the TES bit in CODSCK is low, the transmitted bits are shifted out on the rising edge of SCLK. When TES is high, the transmitted bits are shifted out on the falling edge of SCLK.

When the RES bit in CODSCK is low, the received bits are latched on the falling edge of SCLK.

The generated SCLK signal is a 50% duty cycle clock whose frequency is controlled by the SD[7:0] field in CODSCK. The SCLK is derived from the DSP subsystem clock ((Quartz Clock) x 2.5 for AT75C310, (Quartz Clock) x 3.75 for AT75C220) by a programmable counter. The count value is stored in SD[7:0].

Frame Synchronization Configuration

The direction of the frame synchronization FS is controlled by the FSIO bit in CODCNT.

Frame Synchronization in Master Mode

In master mode, the FS pin is an input. It is internally clocked for resynchronization. When a transition is detected on the synchronized frame sync, the behavior of the codec interface depends on the value of the WT0 and WT1 bits.

When WT1 - WT0 = 00, the codec interface is single data mode. Each communication (i.e., each FS cycle) is used to transfer a sample value.

When WT1 - WT0 = 01, the codec interface transfers two interlaced control/data words. The first word is for codec control, the second is a conversion data.

When WT1 - WT0 = 10, the codec interface transfers three interlaced control/data/data. The first word is for codec control, the second and third words are conversion data.

When WT1 - WT0 = 11, the codec interface is in stereo mode. Each communication (i.e., each FS cycle) is used to transfer two sample values, left and rigth.

Frame Synchronization in Single Data Mode

In this mode, the start of a transfer is detected when a low-to-high transition on the synchronized frame sync occurs. The delay between the low-to-high transition and the first data bit is adjustable through the DD[7:0] field. This value represents the number of SCLK cycles between the low-to-high transition of FS and the occurrence of the first data bit exchange.

When this frame delay has elapsed, the first received bit is sampled on SDIN and the first transmitted bit is shifted out on SDOUT. Then, one bit is transferred per SCLK cycle. When 16 or 32 bits (depending on the DWS field) have been transferred, the codec interface enters an idle mode, waiting for a new frame synchronization event.

Frame Synchronization in Multiple Data Mode

These modes are for WT1 - WT0 = 01 or 10.

When WT1 - WT0 = 01, two identical frames are automatically generated. When WT1 - WT0 = 10, three identical frames are automatically generated. Each individual frame has the same structure as in single data mode.

Frame Synchronization in Stereo Mode

In this mode, two samples are exchanged per FS cycle. The left channel is transferred when FS is high, while the right channel is transferred when FS is low. Both FS edges are used for synchronization.

The start of a transfer is detected when a low-to-high tran-sition on the synchronized frame sync occurs. The delay between the low-to-high transition and the first data bit is adjustable through the DD[7:0] field. This value represents the number of SCLK cycles between the low-to-high transi-tion of FS and the occurrence of the first left channel data bit exchange.

When this frame delay has elapsed, the first received bit is sampled on SDIN and the first transmitted bit is shifted out on SDOUT. Then, one bit is transferred per SCLK cycle. When 16 or 32 bits (depending on the DWS field) have been transferred, the codec interface enters an idle mode, waiting for the high-to-low transition on FS. The start of the right data transfer is detected when a low-to-high transition on the synchronized frame sync occurs. The delay between the low-to-high transition and the first data bit is adjustable through the DD[7:0] field, as for the left data. When this frame delay has elapsed, the first received bit is sampled on SDIN and the first transmitted bit is shifted out on SDOUT. Then, one bit is transferred per SCLK cycle.

When 16 or 32 bits (depending on the DWS field) have been transferred, the codec interface enters an idle mode, waiting for a new frame sync event.

Frame Synchronization in Slave Mode

In slave mode, the FS signal is generated by the AT75C according to the CODCNT, CODFRM and CODSCK registers.

The FS signal is derived from a clock source by a simple divider. If the FSC bit in CODCNT is low, the FS signal is derived from a division of the system clock. If FSC is low, the FS signal is derived from a division of the SCLK, independent of the source of SCLK.

The division ratio between the FS clock source and the FS signal is defined by FD[7:0] in the CODFRM register. This 8-bit number is the value by which to divide the serial clock SCLK to derive the FS signal. If the bit FRL in CODSCK is low, the FS signal is the direct output of the divider and has a 50% duty cycle (sometimes called long FS). If FRL is high, the active FS level will last only one SCLK cycle (also called short FS).

The position of the first data bit, with respect to the FS signal, can also be programmed through the DD field in CODFRM. DD is the number by which to delay the first data bit from FS.

The state machines that control the data exchange are the same in slave mode as in master mode. They rely on the SCLK and FS signals, independent of how these are generated.

Appendix

More on Interrupt Latency

This section contains details on interrupt latency in addition to the details described in “Interrupt Latency” on page 143.

If the processor begins to handle one of the maskable interrupts (INT0, INT1 or INT2), an NMI will be accepted only after the execution of the instruction at the maskable interrupt vector.

Table 14 includes a description of all cases where interrupts are delayed due to execution of a specific instruction. The second column describes the interrupt delayed measured by machine cycles(1). The third column contains the interrupts that are delayed (0, 1, 2 stands for INT0, INT1, INT2, respectively).

Note: 1. A machine cycle is one clock cycle that may be stretched to more than one in case of wait states. It is stretched until the end of the wait interval.

Table 14. Interrupt Latency after Specific Instructions

Current Instruction	Interrupt Delay after Instruction	Interrupt
br, call, ret or reti when the condition is not met; mov soperand(1), sp; movp (aX1). sp; set/rst/chng/addv/subv ##long immediate, sp; Last repetition of instruction during a repeat loop; First repetition of instruction during a repeat loop after returning from an interrupt; First instruction executed after returning from a TRAP/BI routine.	One Cycle	0, 1, 2, NMI
mov soperand, st0; movp (aX1), st0; set/rst/chng/addv/subv ##long immediate, st0; pop st0	Two Cycles	0, 1, 2
	One Cycle	NMI
mov soperand, st2; movp (aX1), st2; set/rst/chng/addv/subv ##long immediate, st2; pop st2	Two Cycles	2
	One Cycle	0, 1, NMI
##long immediate, st0	One Cycle	0, 1, 2
##long immediate, st2	One Cycle	2
retd, retid, mov soperand2, pc; movp (aX1), pc	Two Cycles	0, 1, 2, NMI, BI
mov ##long immediate, pc	One Cycle	0, 1, 2, NMI, BI
rep	Two Cycles	0, 1, 2, NMI, BI

Note: 1. soperand represents every source operand except for a ##long immediate.

Table of Contents

Atmel Headquarters

Corporate Headquarters
2325 Orchard Parkway
San Jose, CA 95131
TEL (408) 441-0311
FAX (408) 487-2600

Europe
Atmel SarL
Route des Arsenaux 41
Casa Postale 80
CH-1705 Fribourg
Switzerland
TEL (41) 26-426-5555
FAX (41) 26-426-5500

Asia
Atmel Asia, Ltd.
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimhatsui
East Kowloon
Hong Kong
TEL (852) 2721-9778
FAX (852) 2722-1369

Japan
Atmel Japan K.K.
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Operations

Memory
Atmel Corporate
2325 Orchard Parkway
San Jose, CA 95131
TEL (408) 436-4270
FAX (408) 436-4314

Microcontrollers
Atmel Corporate
2325 Orchard Parkway
San Jose, CA 95131
TEL (408) 436-4270
FAX (408) 436-4314

Atmel Nantes
La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
TEL (33) 2-40-18-18-50
FAX (33) 2-40-28-19-60

ASIC/ASSP/Smart Cards
Atmel Rousset
Zone Industrielle
13106 Rousset Cedex, France
TEL (33) 4-42-53-64-21
FAX (33) 4-42-53-62-88

Atmel Colorado Springs
1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL (719) 576-3300
FAX (719) 540-1759

Atmel Smart Card ICs
Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
TEL (44) 1355-803-015
FAX (44) 1355-242-743

RF/Automotive
Atmel Heilbronn
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
TEL (49) 71-31-67-0
FAX (49) 71-31-67-2340

Atmel Colorado Springs
1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL (719) 576-3300
FAX (719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Atmel Grenoble
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
TEL (33) 4-76-58-30-00
FAX (33) 4-76-58-34-80

---

e-mail
literature@atmel.com

Web Site
http://www.atmel.com

1368C–08/02/0M