MOTOROLA
MC68030

ENHANCED 32-BIT MICROPROCESSOR USER'S MANUAL

Third Edition

PREFACE

The *MC68030 User's Manual* describes the capabilities, operation, and programming of the MC68030 32-bit second-generation enhanced microprocessor. The manual consists of the following sections and appendix. For detailed information on the MC68030 instruction set refer to M68000PM/AD, *M68000 Family Programmer's Reference Manual.*

Section 1. Introduction

Section 2. Data Organization and Addressing Capabilities

Section 3. Instruction Set Summary

Section 4. Processing States

Section 5. Signal Description

Section 6. On-Chip Cache Memories

Section 7. Bus Operation

Section 8. Exception Processing

Section 9. Memory Management Unit

Section 10. Coprocessor Interface Description

Section 11. Instruction Execution Timing

Section 12. Applications Information

Section 13. Electrical Characteristics

Section 14. Ordering Information and Mechanical Data

Appendix A. M68000 Family Summary

Index

NOTE

In this manual, assertion and negation are used to specify forcing a signal to a particular state. In particular, assertion and assert refer to a signal that is active or true; negation and negate indicate a signal that is inactive or false. These terms are used independently of the voltage level (high or low) that they represent.

The audience of this manual includes systems designers, systems programmers, and applications programmers. Systems designers need some knowledge of all sections, with particular emphasis on Sections 1, 5, 6, 7, 13, 14, and Appendix A. Designers who implement a coprocessor for their system also need a thorough knowledge of Section 10.

Systems programmers should become familiar with Sections 1, 2, 3, 4, 6, 8, 9, 11, and Appendix A. Applications programmers can find most of the information they need in Sections 1, 2, 3, 4, 9, 11, 12, and Appendix A.

From a different viewpoint, the audience for this book consists of users of other M68000 Family members and those who are not familiar with these microprocessors. Users of the other family members can find references to similarities to and differences from the other Motorola microprocessors throughout the manual. However, Section 1 and Appendix A specifically identify the MC68030 within the rest of the family and contrast its differences.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Section 8
Exception Processing

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Concluded)

LIST OF TABLES

LIST OF TABLES (Continued)

SECTION 1
INTRODUCTION

The MC68030 is a second-generation full 32-bit enhanced microprocessor from Motorola. The MC68030 is a member of the M68000 Family of devices that combines a central processing unit (CPU) core, a data cache, an instruction cache, an enhanced bus controller, and a memory management unit (MMU) in a single VLSI device. The processor is designed to operate at clock speeds beyond 20 MHz. The MC68030 is implemented with 32-bit registers and data paths, 32-bit addresses, a rich instruction set, and versatile addressing modes.

The MC68030 is upward object code compatible with the earlier members of the M68000 Family and has the added features of an on-chip MMU, a data cache, and an improved bus interface. It retains the flexible coprocessor interface pioneered in the MC68020 and provides full IEEE floating-point support through this interface with the MC68881 or MC68882 floating-point coprocessor. Also, the internal functional blocks of this microprocessor are designed to operate in parallel, allowing instruction execution to be overlapped. In addition to instruction execution, the internal caches, the on-chip MMU, and the external bus controller all operate in parallel.

The MC68030 fully supports the nonmultiplexed bus structure of the MC68020, with 32 bits of address and 32 bits of data. The MC68030 bus has an enhanced controller that supports both asynchronous and synchronous bus cycles and burst data transfers. It also supports the MC68020 dynamic bus sizing mechanism that automatically determines device port sizes on a cycle-by-cycle basis as the processor transfers operands to or from external devices.

A block diagram of the MC68030 is shown in Figure 1-1. The instructions and data required by the processor are supplied from the internal caches whenever possible. The MMU translates the logical address generated by the processor into a physical address utilizing its address translation cache (ATC). The bus controller manages the transfer of data between the CPU and memory or devices at the physical address.

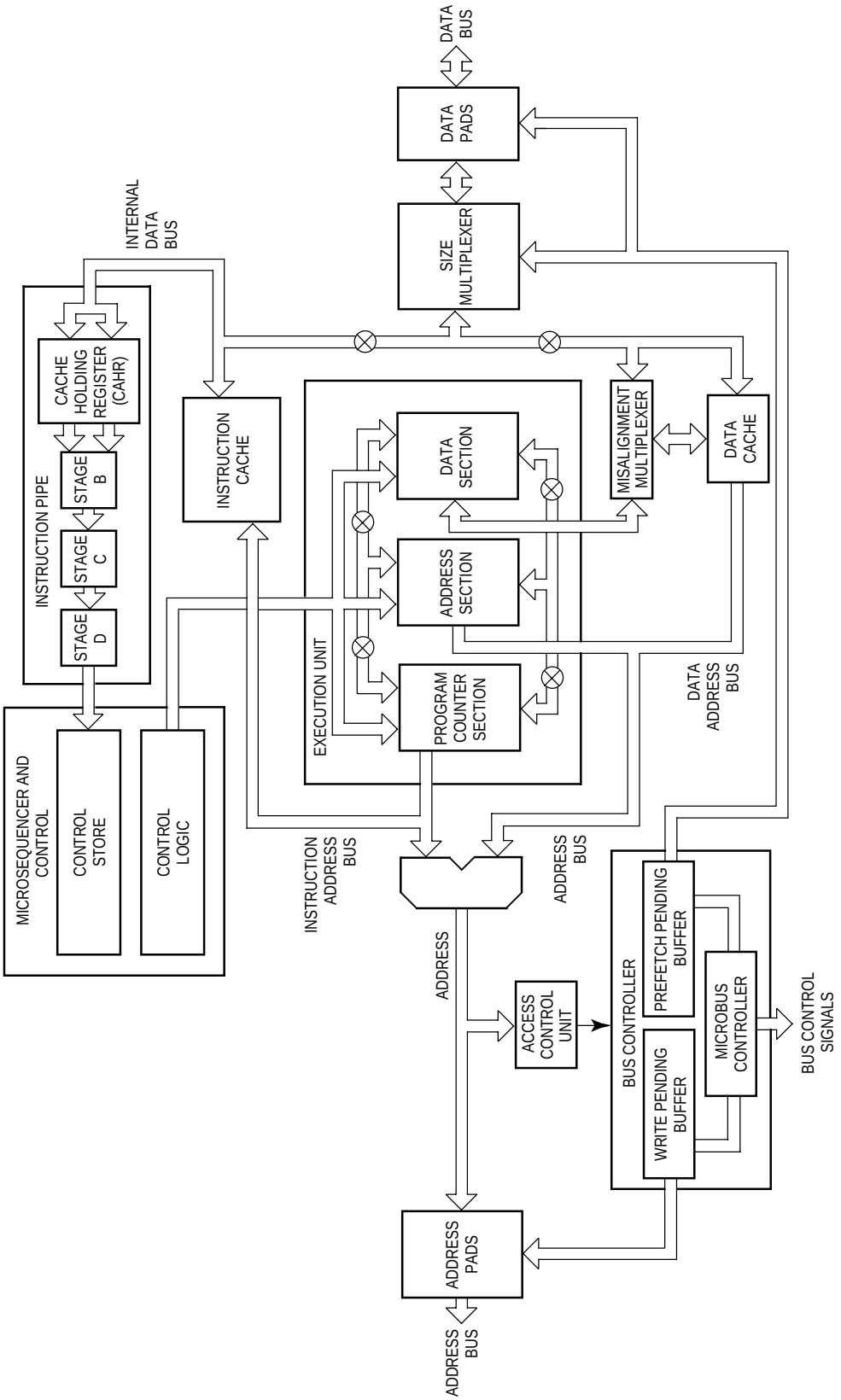


Figure 1-1. Block Diagram

1.1 FEATURES

The features of the MC68030 microprocessor are:

- Object Code Compatible with the MC68020 and Earlier M68000 Microprocessors
- Complete 32-Bit Nonmultiplexed Address and Data Buses
- 16 32-Bit General-Purpose Data and Address Registers
- Two 32-Bit Supervisor Stack Pointers and 10 Special-Purpose Control Registers
- 256-Byte Instruction Cache and 256-Byte Data Cache Can Be Accessed Simultaneously
- Paged MMU that Translates Addresses in Parallel with Instruction Execution and Internal Cache Accesses
- Two Transparent Segments Allow Untranslated Access to Physical Memory To Be D fined for Systems That Transfer Large Blocks of Data between Predefined Physical Addresses — e.g., Graphics Applications
- Pipelined Architecture with Increased Parallelism Allows Accesses to Internal Caches To Occur in Parallel with Bus Transfers and Instruction Execution To Be Overlapped
- Enhanced Bus Controller Supports Asynchronous Bus Cycles (three clocks minimum), Synchronous Bus Cycles (two clocks minimum), and Burst Data Transfers (one clock minimum) all to the Physical Address Space
- Dynamic Bus Sizing Supports 8-, 16-, 32-Bit Memories and Peripherals
- Support for Coprocessors with the M68000 Coprocessor Interface — e.g., Full IEEE Floating-Point Support Provided by the MC68881/MC68882 Floating-Point Coprocessors
- 4-Gbyte Logical and Physical Addressing Range
- Implemented in Motorola's HCMOS Technology That Allows CMOS and HMOS (High-Density NMOS) Gates to be Combined for Maximum Speed, Low Power, and Optimum Die Size
- Processor Speeds Beyond 20 MHz

Both improved performance and increased functionality result from the on-chip implementation of the MMU and the data and instruction caches. The enhanced bus controller and the internal parallelism also provide increased system performance. Finally, the improved bus interface, the reduction in physical size, and the lower power consumption combine to reduce system costs and satisfy cost/performance goals of the system designer.

1.2 MC68030 EXTENSIONS TO THE M68000 FAMILY

In addition to the on-chip instruction cache present in the MC68020, the MC68030 has an internal data cache. Data that is accessed during read cycles may be stored in the on-chip cache, where it is available for subsequent accesses. The data cache reduces the number of external bus cycles when the data operand required by an instruction is already in the data cache.

Performance is enhanced further because the on-chip caches can be internally accessed in a single clock cycle. In addition, the bus controller provides a two-clock cycle synchronous mode and burst mode accesses that can transfer data in as little as one clock per long word.

The MC68030 enhanced microprocessor contains an on-chip MMU that allows address translation to operate in parallel with the CPU core, the internal caches, and the bus controller.

Additional signals support emulation and system analysis. External debug equipment can disable the on-chip caches and the MMU to freeze the MC68030 internal state during breakpoint processing. In addition, the MC68030 indicates:

1. The start of a refill of the instruction pipe
2. Instruction boundaries
3. Pending trace or interrupt processing
4. Exception processing
5. Halt conditions

This status and control information allows external debugging equipment to trace the MC68030 activity and interact nonintrusively with the MC68030 to effectively reduce system debug effort.

1.3 PROGRAMMING MODEL

The programming model of the MC68030 consists of two groups of registers: the user model and the supervisor model. This corresponds to the user and supervisor privilege levels. User programs executing at the user privilege level use the registers of the user model. System software executing at the supervisor level uses the control registers of the supervisor level to perform supervisor functions.

Figure 1-2 shows the user programming model, consisting of 16 32-bit general-purpose registers and two control registers:

- General-Purpose 32-Bit Registers (D0–D7, A0–A7)
- 32-Bit Program Counter (PC)
- 8-Bit Condition Code Register (CCR)

The supervisor programming model consists of the registers available to the user plus 14 control registers:

- Two 32-Bit Supervisor Stack Pointers (ISP and MSP)
- 16-Bit Status Register (SR)
- 32-Bit Vector Base Register (VBR)
- 32-Bit Alternate Function Code Registers (SFC and DFC)
- 32-Bit Cache Control Register (CACR)
- 32-Bit Cache Address Register (CAAR)
- 64-Bit CPU Root Pointer (CRP)
- 64-Bit Supervisor Root Pointer (SRP)
- 32-Bit Translation Control Register (TC)
- 32-Bit Transparent Translation Registers (TT0 and TT1)
- 16-Bit MMU Status Register (MMUSR)

The user programming model remains unchanged from previous M68000 Family microprocessors. The supervisor programming model supplements the user programming model and is used exclusively by the MC68030 system programmers who utilize the supervisor privilege level to implement sensitive operating system functions, I/O control, and memory management subsystems. The supervisor programming model contains all the controls to access and enable the special features of the MC68030. This segregation was carefully planned so that all application software is written to run at the nonprivileged user level and migrates to the MC68030 from any M68000 platform without modification. Since system software is usually modified by system programmers when ported to a new design, the control features are properly placed in the supervisor programming model. For example, the transparent translation feature of the MC68030 is new to the family supervisor programming model for the MC68030 and the two translation registers are new additions to the family supervisor programming model for the MC68030. Only supervisor code uses this feature, and user application programs remain unaffected.

Registers D0–D7 are used as data registers for bit and bit field (1 to 32 bits), byte (8 bit), word (16 bit), long-word (32 bit), and quad-word (64 bit) operations. Registers A0–A6 and the user, interrupt, and master stack pointers are address registers that may be used as software stack pointers or base address registers. Register A7 (shown as A7' and A7'' in Figure 1-3) is a register designation that applies to the user stack pointer in the user privilege level and to either the interrupt or master stack pointer in the supervisor privilege level. In the supervisor privilege level, the active stack pointer (interrupt or master) is called the supervisor stack pointer (SSP). In addition, the address registers may be used for word and long-word operations. All of the 16 general-purpose registers (D0–D7, A0–A7) may be used as index registers.

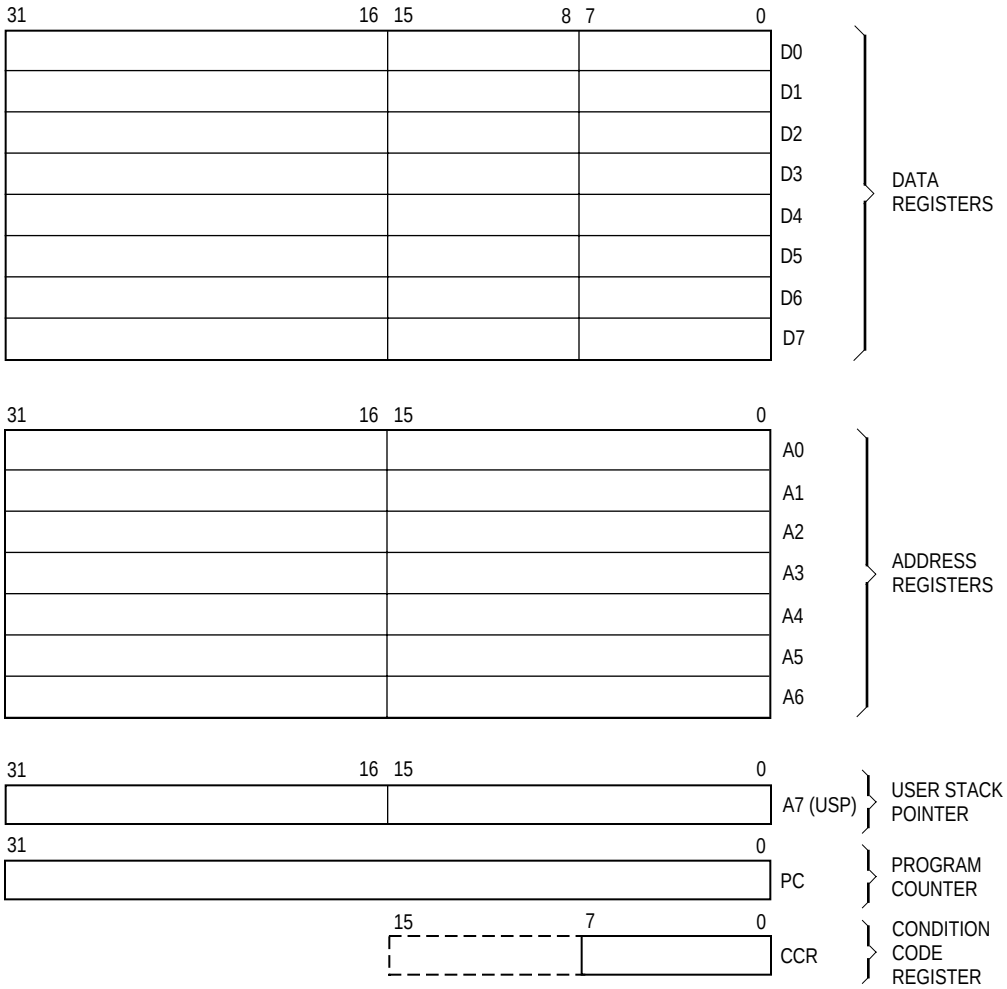


Figure 1-2. User Programming Model

The program counter (PC) contains the address of the next instruction to be executed by the MC68030. During instruction execution and exception processing, the processor automatically increments the contents of the PC or places a new value in the PC, as appropriate.

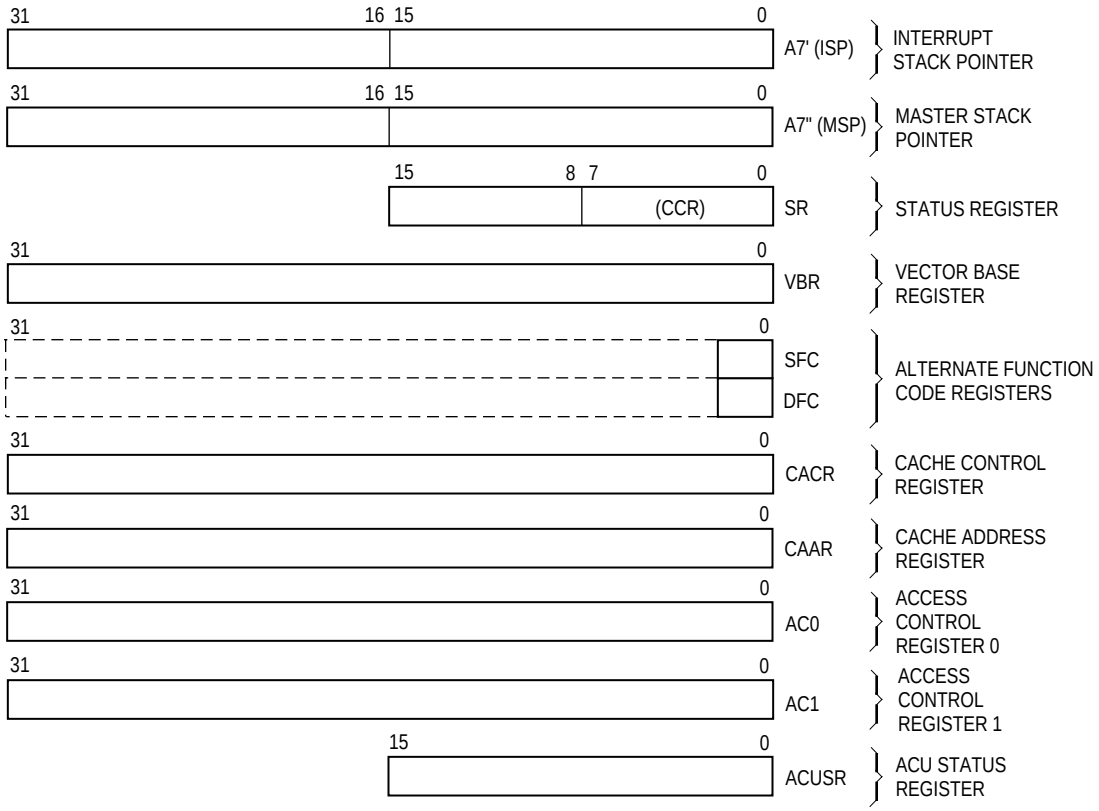


Figure 1-3. Supervisor Programming Model Supplement

The status register, SR, (see Figure 1-4) stores the processor status. It contains the condition codes that reflect the results of a previous operation and can be used for conditional instruction execution in a program. The condition codes are extend (X), negative (N), zero (Z), overflow (V), and carry (C). The user byte containing the condition codes is the only portion of the status register information available in the user privilege level, and it is referenced as the CCR in user programs. In the supervisor privilege level, software can access the full status register, including the interrupt priority mask (three bits) as well as additional control bits. These bits indicate whether the processor is in:

1. One of two trace modes (T1, T0)
2. Supervisor or user privilege level (S)
3. Master or interrupt mode (M)

The vector base register (VBR) contains the base address of the exception vector table in memory. The displacement of an exception vector is added to the value in this register to access the vector table.

Alternate function code registers, SFC and DFC, contain 3-bit function codes. Function codes can be considered extensions of the 32-bit linear address that optionally provide as many as eight 4-Gbyte address spaces. Function codes are automatically generated by the processor to select address spaces for data and program at the user and supervisor privilege levels and a CPU address space for processor functions (e.g., coprocessor communications). Registers SFC and DFC are used by certain instructions to explicitly specify the function codes for operations.

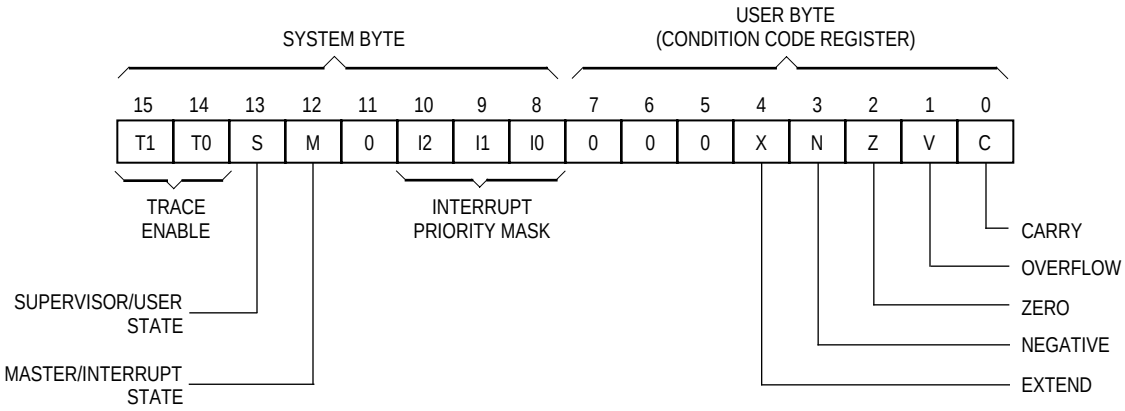


Figure 1-4. Status Register

The cache control register (CACR) controls the on-chip instruction and data caches of the MC68030. The cache address register (CAAR) stores an address for cache control functions.

The CPU root pointer (CRP) contains a pointer to the root of the translation tree for the currently executing task of the MC68030. This tree contains the mapping information for the task's address space. When the MC68030 is configured to provide a separate address space for supervisor routines, the supervisor root pointer (SRP) contains a pointer to the root of the translation tree describing the supervisor's address space.

The translation control register (TC) consists of several fields that control address translation. These fields enable and disable address translation, enable and disable the use of SRP for the supervisor address space, and select or ignore the function codes in translating addresses. Other fields define the size of memory pages, the number of address bits used in translation, and the translation table structure.

The transparent translation registers, TT0 and TT1, can each specify separate blocks of memory as directly accessible without address translation. Logical addresses in these areas become the physical addresses for memory access. Function codes and the eight most significant bits of the address can be used to define the area of memory and type of access; either read, write, or both types of memory access can be directly mapped. The transparent translation feature allows rapid movement of large blocks of data in memory or I/O space without disturbing the context of the on-chip address translation cache or incurring delays associated with translation table lookups. This feature is useful to graphics, controller, and real-time applications.

The MMU status register (MMUSR) contains memory management status information resulting from a search of the address translation cache or the translation tree for a particular logical address.

1.4 DATA TYPES AND ADDRESSING MODES

Seven basic data types are supported:

1. Bits
2. Bit Fields (Fields of consecutive bits, 1–32 bits long)
3. BCD Digits (Packed: 2 digits/byte, Unpacked: 1 digit/byte)
4. Byte Integers (8 bits)
5. Word Integers (16 bits)
6. Long-Word Integers (32 bits)
7. Quad-Word Integers (64 bits)

In addition, the instruction set supports operations on other data types such as memory addresses. The coprocessor mechanism allows direct support of floating-point operations with the MC68881 and MC68882 floating-point coprocessors as well as specialized user-defined data types and functions.

The 18 addressing modes, shown in Table 1-1, include nine basic types:

1. Register Direct
2. Register Indirect
3. Register Indirect with Index
4. Memory Indirect
5. Program Counter Indirect with Displacement
6. Program Counter Indirect with Index
7. Program Counter Memory Indirect
8. Absolute
9. Immediate

The register indirect addressing modes can also postincrement, predecrement, offset, and index addresses. The program counter relative mode also has index and offset capabilities. As in the MC68020, both modes are extended to provide indirect reference through memory. In addition to these addressing modes, many instructions implicitly specify the use of the condition code register, stack pointer, and/or program counter.

1.5 INSTRUCTION SET OVERVIEW

The instructions in the MC68030 instruction set are listed in Table 1-2. The instruction set has been tailored to support structured high-level languages and sophisticated operating systems. Many instructions operate on bytes, words, or long words, and most instructions can use any of the 18 addressing modes.

Table 1-1. Addressing Modes

Addressing Modes	Syntax
Register Direct	
Data Register Direct	Dn
Address Register Direct	An
Register Indirect	
Address Register Indirect	(An)
Address Register Indirect with Postincrement	(An)
Address Register Indirect with Predecrement	–(An)
Address Register Indirect with Displacement	(d_{16},An)
Register Indirect with Index	
Address Register Indirect with Index (8-BitDisplacement)	(d_8,An,Xn)
Address Register Indirect with Index (Base Displacement)	(bd,An,Xn)
Memory Indirect	
Memory Indirect Postindexed	([bd,An],Xn,od)
Memory Indirect Preindexed	([bd,An,Xn],od)
Program Counter Indirect with Displacement	(d_{16},PC)
Program Cou nter Indirect with IndexPC Indirect with Index (8-Bit Displacement)	(d_8,PC,Xn)
PC Indirect with Index (Base Displacement)	(bd,PC,Xn)
Program Counter Memory Indirect	
PC Memory Indirect Postindexed	([bd,PC],Xn,od)
PC Memory Indirect Preindexed	([bd,PC,Xn],od)
Absolute	
Absolute Short	(xxx).W
Absolute Long	(xxx).L
Immediate	#(data)

NOTES:

Dn	=	Data Register, D0–D7
An	=	Address Register, A0–A7
8, $^{d}16$	=	A twos-complement or sign-extended displacement; added as part of the effective address calculation; size is 8 (d_8) or 16 (d_{16}) bits; when omitted, assemblers use a value of zero.
Xn	=	Address or data register used as an index register; form is Xn.SIZE*SCALE, where SIZE is .W or .L indicates index register size) and SCALE is 1, 2, 4, or 8 (index register is multiplied by SCALE); use of SIZE and/or SCALE is optional.
bd	=	A twos-complement base displacement;when present, size can be 16 or 32 bits.
od	=	Outer displacement, added as part of effective address calculation after any memory indirection; use is optional with asize of 16 or 32 bits.
PC	=	Program Counter
(data)	=	Immediate value of 8, 16, or 32 bits
( )	=	Effective Address
[ ]	=	Use as indirect access to long-word address.

1.6 VIRTUAL MEMORY AND VIRTUAL MACHINE CONCEPTS

The full addressing range of the MC68030 is 4 Gbytes (4,294,967,296 bytes) in each of eight address spaces. Even though most systems implement a smaller physical memory, the system can be made to appear to have a full 4 Gbytes of memory available to each user program by using virtual memory techniques.

In a virtual memory system, a user program can be written as if it has a large amount of memory available, when the physical memory actually present is much smaller. Similarly, a system can be designed to allow user programs to access devices that are not physically present in the system, such as tape drives, disk drives, printers, terminals, and so forth. With proper software emulation, a physical system can appear to be any other M68000 computer system to a user program, and the program can be given full access to all of the resources of that emulated system. Such an emulated system is called a virtual machine.

1.6.1 Virtual Memory

A system that supports virtual memory has a limited amount of high-speed physical memory that can be accessed directly by the processor and maintains an image of a much larger virtual memory on a secondary storage device such as a large-capacity disk drive. When the processor attempts to access a location in the virtual memory map that is not resident in physical memory, a page fault occurs. The access to that location is temporarily suspended while the necessary data is fetched from secondary storage and placed in physical memory. The suspended access is then either restarted or continued.

The MC68030 uses instruction continuation to support virtual memory. When a bus cycle is terminated with a bus error, the microprocessor suspends the current instruction and executes the virtual memory bus error handler. When the bus error handler has completed execution, it returns control to the program that was executing when the error was detected, reruns the faulted bus cycle (when required), and continues the suspended instruction.

Table 1-2. Instruction Set

Mnemonic	Description
ABCD	Add Decimal with Extend
ADD	Add
ADDA	Add Address
ADDI	Add Immediate
ADDQ	Add Quick
ADDX	Add with Extend
AND	Logical AND
ANDI	Logical AND Immediate
ASL, ASR	Arithmatic Shift Left and Right
Bcc	Branch Conditionally
BCHG	Test Bit and Change
BCLR	Test Bit and Clear
BFCHG	Test Bit Feild and Change
BFCLR	Test Bit Feild and Clear
BFEXTS	Signed Bit Feild Extract
BFEXTU	Unsigned Bit Feild Extract
BFFO	Bit Feild Find First One
BFINS	Bit Feild Insert
BFSET	Test Bit Feild and Set
BFTST	Test Bit Feild
BKPT	Breakpoint
BRA	Branch
BSET	Test Bit and Set
BSR	Branch to Subroutine
BTST	Test Bit
CAS	Compare and Swap Operands
CAS 2	Compare and Swap Dual Operands
CHK	Check Register Against Bound
CHK2	Check Register Against Upper and Lower Bounds
CLR	Clear
CMP	Compare
CMPA	Compare Address
CMPI	Compare Immediate
CMPM	Compare Memory to Memory
CMP2	Compare Registre Against Upper and Lower Bounds
DBcc	Test Condition, Decrement and Branch
DIVS, DIVSL	Signed Divide
DIVU, DIVUL	Unsigned Divide
EOR	Logical Exclusive OR
EORI	Logical Exclusive OR Immediate
EXG	Exchange Registers
EXT, EXTB	Sign Extend
ILLEGAL	Take Illegal Instruction Trap
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link and Allocate
LSL, LSR	Logical Shift Left and Right
MOVE	Move
MOVEA	Move Address
MOVE CCR	Move Condition Code Register
MOVE SR	Move Status Register

Mnemonic	Description
MOVE USP	Move User Stack Pointer
MOVEC	Move Control Register
MOVEM	Move Multiple Registers
MOVEP	Move Periphral
MOVEQ	Move Quick
MOVES	Move Alternate Address Space
MULS	Signed Multiply
MULU	Unsigned Multiply
NBCD	Negate Decimal with Extend
NEG	Negate
NEGX	Negate with Extend
NOP	No Operation
NOT	Logical Compliment
OR	Logical Inclusive OR
ORI	Logical Inclusive OR Immediate
ORI CCR	Logical Inclusive OR Immediate to Condition Codes
ORI SR	Logical Inclusive OR Immediate to Status Register
PACK	Pack BCD
PEA	Push Effective Address
PFLUSH	Flush Entry(ies) in the ATC
PFLUSHA	Flush All Entries in the ATC
PLOADR, PLOADW	Load Entry into the ATC
PMOVE	Move to-from MMU Registers
PMOVEFD	Move to-from MMU Registers with Flush Disable
PTESTR PTESTW	Test a Logical Address
RESET	Reset External Devices
ROL, ROR	Rotate Left and Right
ROXL, ROXR	Rotate With Extend Left and Right
RTD	Return and Deallocate
RTE	Return from Exception
RTR	Return and Restore Codes
RTS	Return from Subroutine
SBCD	Subtract Decimal With Extend
Scc	Set Conditionally
STOP	Stop
SUB	Subtract
SUBA	Subtract Immediate
SUBI	Subtract Quick
SUBQ	Subtract with Extend
SUBX	Swap Register Words
SWAP	Test Operand and Set
TAS	Trap
TRAP	Trap Conditionally
TRAPcc	Trap on Overflow
TRAPV	Test on Overflow
TST	Test Operand
UNLK	Unlink
UNPK	Unpack BCD

Mnemonic	Description
cpBcc	Branch Conditionally
cpDBcc	Test Coprocessor Condition, Decrement and Branch
cpGEN	Coprocessor General Instruction

Mnemonic	Description
cpRESTORE	Restore Internal State of Coprocessor
cpSAVE	Save Internal State of Coprocessor
cpScc	Set Conditionally
cpTRAPcc	Trap Conditionally

1.6.2 Virtual Machine

A typical use for a virtual machine system is the development of software, such as an operating system, for a new machine also under development and not yet available for programming use. In a virtual machine system, a governing operating system emulates the hardware of the new machine and allows the new software to be executed and debugged as though it were running on the new hardware. Since the new software is controlled by the governing operating system, it is executed at a lower privilege level than the governing operating system. Thus, any attempts by the new software to use virtual resources that are not physically present (and should be emulated) are trapped to the governing operating system and performed by its software.

In the MC68030 implementation of a virtual machine, the virtual application runs at the user privilege level. The governing operating system executes at the supervisor privilege level and any attempt by the new operating system to access supervisor resources or execute privileged instructions causes a trap to the governing operating system.

Instruction continuation is used to support virtual I/O devices in memory-mapped input/output systems. Control and data registers for the virtual device are simulated in the memory map. An access to a virtual register causes a fault and the function of the register is emulated by software.

1.7 THE MEMORY MANAGEMENT UNIT

The MMU supports virtual memory systems by translating logical addresses to physical addresses using translation tables stored in memory. The MMU stores address mappings in an address translation cache (ATC) that contains the most recently used translations. When the ATC contains the address for a bus cycle requested by the CPU, a translation table search is not performed. Features of the MMU include:

- Multiple Level Translation Tables with Short- and Long-Format Descriptors for Efficient Table Space Usage
- Table Searches Automatically Performed in Microcode
- 22-Entry Fully Associative ATC
- Address Translations and Internal Instruction and Data Cache Accesses Performed in Parallel
- Eight Page Sizes Available Ranging from 256 to 32K Bytes
- Two Optional Transparent Blocks
- User and Supervisor Root Pointer Registers
- Write Protection and Supervisor Protection Attributes
- Translations Enabled/Disabled by Software
- Translations Can Be Disabled with External $\overline{\text{MMUDIS}}$ Signal
- Used and Modified Bits Automatically Maintained in Tables and ATC
- Cache Inhibit Output ($\overline{\text{CIOUT}}$) Signal Can Be Asserted on a Page-by-Page Basis
- 32-Bit Internal Logical Address with Capability To Ignore as many as 15 Upper Address Bits
- 3-Bit Function Code Supports Separate Address Spaces
- 32-Bit Physical Address

The memory management function performed by the MMU is called demand paged memory management. Since a task specifies the areas of memory it requires as it executes, memory allocation is supported on a demand basis. If a requested access to memory is not currently mapped by the system, then the access causes a demand for the operating system to load or allocate the required memory image. The technique used by the MC68030 is paged memory management because physical memory is managed in blocks of a specified number of bytes, called page frames. The logical address space is divided into fixed-size pages that contain the same number of bytes as the page frames. Memory management assigns a physical base address to a logical page. The system software then transfers data between secondary storage and memory one or more pages at a time.

1.8 PIPELINED ARCHITECTURE

The MC68030 uses a three-stage pipelined internal architecture to provide for optimum instruction throughput. The pipeline allows as many as three words of a single instruction or three consecutive instructions to be decoded concurrently.

1.9 THE CACHE MEMORIES

Due to locality of reference, instructions and data that are used in a program have a high probability of being reused within a short time. Additionally, instructions and data operands that reside in proximity to the instructions and data currently in use also have a high probability of being utilized within a short period. To exploit these locality characteristics, the MC68030 contains two on-chip logical caches, a data cache, and an instruction cache.

Each of the caches stores 256 bytes of information, organized as 16 entries, each containing a block of four long words (16 bytes). The processor fills the cache entries either one long word at a time or, during burst mode accesses, four long words consecutively. The burst mode of operation not only fills the cache efficiently but also captures adjacent instruction or data items that are likely to be required in the near future due to locality characteristics of the executing task.

The caches improve the overall performance of the system by reducing the number of bus cycles required by the processor to fetch information from memory and by increasing the bus bandwidth available for other bus masters in the system. Addition of the data cache in the MC68030 extends the benefits of cache techniques to all memory accesses. During a write cycle, the data cache circuitry writes data to a cached data item as well as to the item in memory, maintaining consistency between data in the cache and that in memory. However, writing data that is not in the cache may or may not cause the data item to be stored in the cache, depending on the write allocation policy selected in the cache control register (CACR).

SECTION 2
DATA ORGANIZATION AND ADDRESSING CAPABILITIES

Most external references to memory by a microprocessor are either program references or data references; they either access instruction words or operands (data items) for an instruction. Program references are references to the program space, the section of memory that contains the program instructions and any immediate data operands that reside in the instruction stream. Refer to M68000PM/AD, *M68000 Programmer's Reference Manual*, for descriptions of the instructions in the program space. Data references refer to the data space, the section of memory that contains the program data. Data items in the instruction stream can be accessed with the program counter relative addressing modes, and these accesses are classified as program references. A third type of external reference used for coprocessor communications, interrupt acknowledge cycles, and breakpoint acknowledge cycles is classified as a CPU space reference. The MC68030 automatically sets the function codes to access the program space, the data space, or the CPU space for special functions as required. The function codes can be used by the memory management unit to organize separate program (read only) and data (read-write) memory areas.

This section describes the data organization and addressing capabilities of the MC68030. It lists the types of operands used by instructions and describes the registers and their use as operands. Next, the section describes the organization of data in memory and the addressing modes available to access data in memory. Last, the section describes the system stack and user program stacks and queues.

2.1 INSTRUCTION OPERANDS

The MC68030 supports a general-purpose set of operands to serve the requirements of a large range of applications. Operands of MC68030 instructions may reside in registers, in memory, or within the instructions themselves. An instruction operand might also reside in a coprocessor. An operand may be a single bit, a bit field of from 1 to 32 bits in length, a byte (8 bits), a word (16 bits), a long word (32 bits), or a quad word (64 bits). The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation. Coprocessors are designed to support special computation models that require very specific but widely varying data operand types and sizes. Hence, coprocessor instructions can specify operands of any size.

2.2 ORGANIZATION OF DATA IN REGISTERS

The eight data registers can store data operands of 1, 8, 16, 32, and 64 bits, addresses of 16 or 32 bits, or bit fields of 1 to 32 bits. The seven address registers and the three stack pointers are used for address operands of 16 or 32 bits. The control registers (SR, VBR, SFC, DFC, CACR, CAAR, CRP, SRP, TC, TT0, TT1, and MMUSR) vary in size according to function. Coprocessors may define unique operand sizes and support them with on-chip registers accordingly.

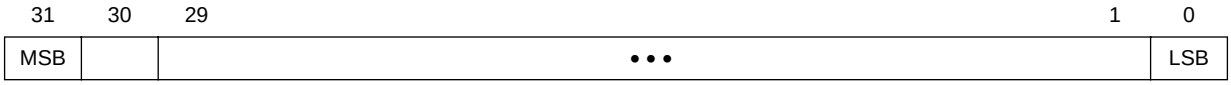
2.2.1 Data Registers

Each data register is 32 bits wide. Byte operands occupy the low-order 8 bits, word operands the low-order 16 bits, and long-word operands the entire 32 bits. When a data register is used as either a source or destination operand, only the appropriate low-order byte or word (in byte or word operations, respectively) is used or changed; the remaining high-order portion is neither used nor changed. The least significant bit of a long-word integer is addressed as bit zero, and the most significant bit is addressed as bit 31. For bit fields, the most significant bit is addressed as bit zero, and the least significant bit is addressed as the width of the field minus one. If the width of the field plus the offset is greater than 32, the bit field wraps around within the register. The following illustration shows the organization of various types of data in the data registers.

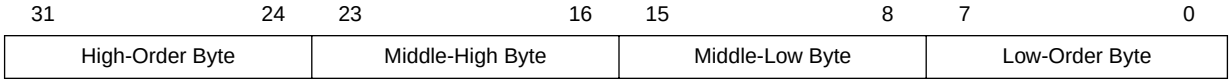
Quad-word data consists of two long words; for example, the product of 32-bit multiply or the quotient of 32-bit divide operations (signed and unsigned). Quad words may be organized in any two data registers without restrictions on order or pairing. There are no explicit instructions for the managment of this data type, although the MOVEM instruction can be used to move a quad word into or out of the registers.

Binary-coded decimal (BCD) data represents decimal numbers in binary form. Although many BCD codes have been devised, the BCD instructions of the M68000 Family support formats which the four least significant bits consist of a binary number having the numeric value of the corresponding decimal number. Two BCD formats are used. In the unpacked BCD format, a byte contains one digit; the four least significant bits contain the binary value and the four most significant bits are undefined. Each byte of the packed BCD format contains two digits; the least significant four bits contain the least significant digit.

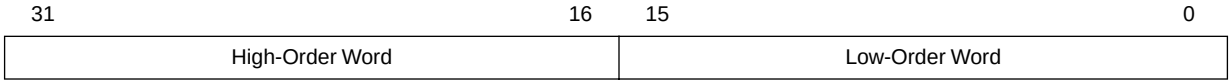
Bit ≤ (0 Modulo (Offset)<31, Offset of 0 = MSB)



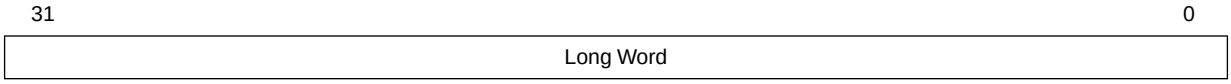
Byte



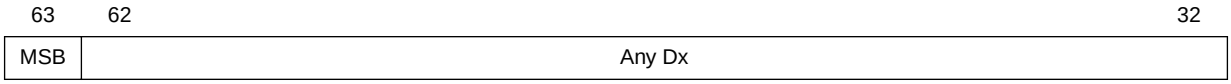
16-Bit Word

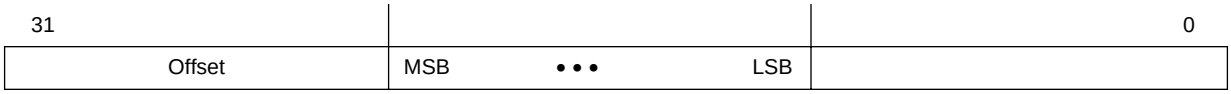


Long Word

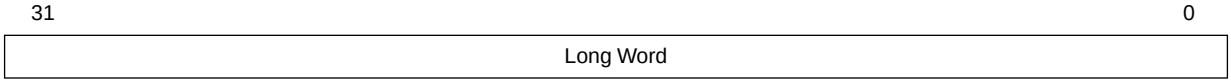


Quad Word



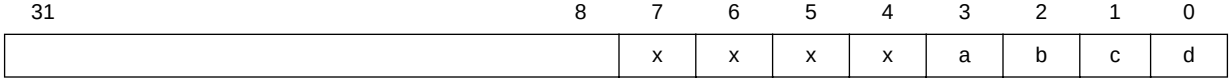


Bit Field (0 ≤ Offset<32, 0<Width ≤ 32)

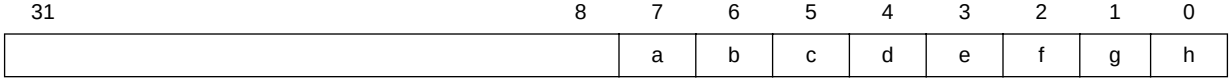


Note: If width + offset < 32, bit filed wraps around within the register.

Unpacked BCD (a = MSB)



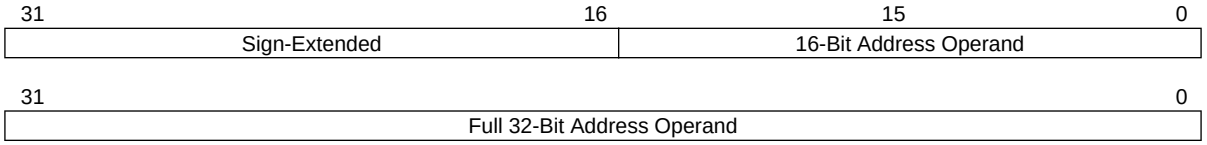
Packed BCD (a = MSB First Digit, e = MSB Second Digit)



Data Organization in Data Registers

2.2.2 Address Registers

Each address register and stack pointer is 32 bits wide and holds a 32-bit address. Address registers cannot be used for byte-sized operands. Therefore, when an address register is used as a source operand, either the low-order word or the entire long-word operand is used, depending upon the operation size. When an address register is used as the destination operand, the entire register is affected, regardless of the operation size. If the source operand is a word size, it is first sign-extended to 32 bits and then used in the operation to an address register destination. Address registers are used primarily for addresses and to support address computation. The instruction set includes instructions that add to, subtract from, compare, and move the contents of address registers. The following example shows the organization of addresses in address registers.



Address Organization in Address Registers

2.2.3 Control Registers

The control registers described in this section contain control information for supervisor functions and vary in size. With the exception of the user portion of the status register (CCR), they are accessed only by instructions at the supervisor privilege level.

The status register (SR), shown in Figure 1–4, is 16 bits wide. Only 12 bits of the status register are defined; all undefined values are reserved by Motorola for future definition. The undefined bits are read as zeros and should be written as zeros for future compatibility. The lower byte of the status register is the CCR. Operations to the CCR can be performed at the supervisor or user privilege level. All operations to the status register and CCR are word-sized operations, but for all CCR operations, the upper byte is read as all zeros and is ignored when written, regardless of privilege level.

The supervisor programming model (see Figure 1–3) shows the control registers. The cache control register (CACR) provides control and status information for the on-chip instruction and data caches. The cache address register (CAAR) contains the address for cache control functions. The vector base register (VBR) provides the base address of the exception vector table. All operations involving the CACR, CAAR, and VBR are long-word operations, whether these registers are used as the source or the destination operand.

The alternate function code registers (SFC and DFC)

are 32-bit registers with only bits 2:0 implemented that contain the address space values (FC0-FC2) for the read or write operands of MOVES, PLOAD, PFLUSH, and PTEST instructions. The MOVEC instruction is used to transfer values to and from the alternate function code registers. These are long-word transfers; the upper 29 bits are read as zeros and are ignored when written.

The remaining control registers in the supervisor programming model are used by the memory management unit (MMU). The CPU root pointer (CRP) and supervisor root pointer (SRP) contain pointers to the user and supervisor address translation trees. Transfers of data to and from these 64-bit registers are quad-word transfers. The translation control register (TC) contains control information for the MMU. The MC68030 always uses long-word transfers to access this 32-bit register. The transparent translation registers (TT0 and TT1) also contain 32 bits each; they identify memory areas for direct addressing without address translation. Data transfers to and from these registers are long-word transfers. The MMU status register (MMUSR) stores the status of the MMU after execution of a PTEST instruction. It is a 16-bit register, and transfers to and from the MMUSR are word transfers. Refer to **Section 9 Memory Management Unit** for more detail.

2.3 ORGANIZATION OF DATA IN MEMORY

Memory is organized on a byte-addressable basis where lower addresses correspond to higher order bytes. The address, N, of a long-word data item corresponds to the address of the most significant byte of the highest order word. The lower order word is located at address N + 2, leaving the least significant byte at address N + 3 (refer to Figure 2–1). Notice that the MC68030 does not require data to be aligned on word boundaries (refer to Figure 2–2), but the most efficient data transfers occur when data is aligned on the same byte boundary as its operand size. However, instruction words must be aligned on word boundaries.

The data types supported in memory by the MC68030 are bit and bit field data; integer data of 8, 16, or 32 bits; 32-bit addresses; and BCD data (packed and unpacked). These data types are organized in memory as shown in Figure 2–2. Note that all of these data types can be accessed at any byte address.

Coprocessors can implement any data types and lengths up to 255 bytes. For example, the MC68881/MC68882 floating-point coprocessors support memory accesses for quad-word-sized items (double-precision floating-point values).

Figure 2A bit operand is specified by a base address that selects one byte in memory (the base byte) and a bit number that selects the one bit in this byte. The most significant bit of the byte is bit 7.

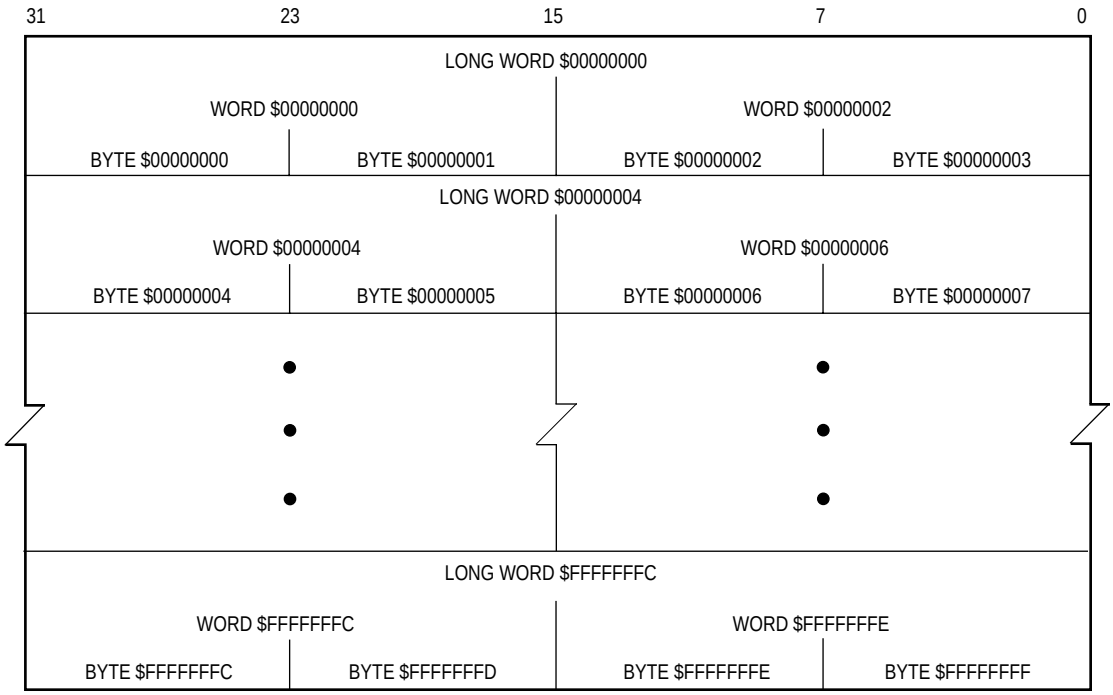
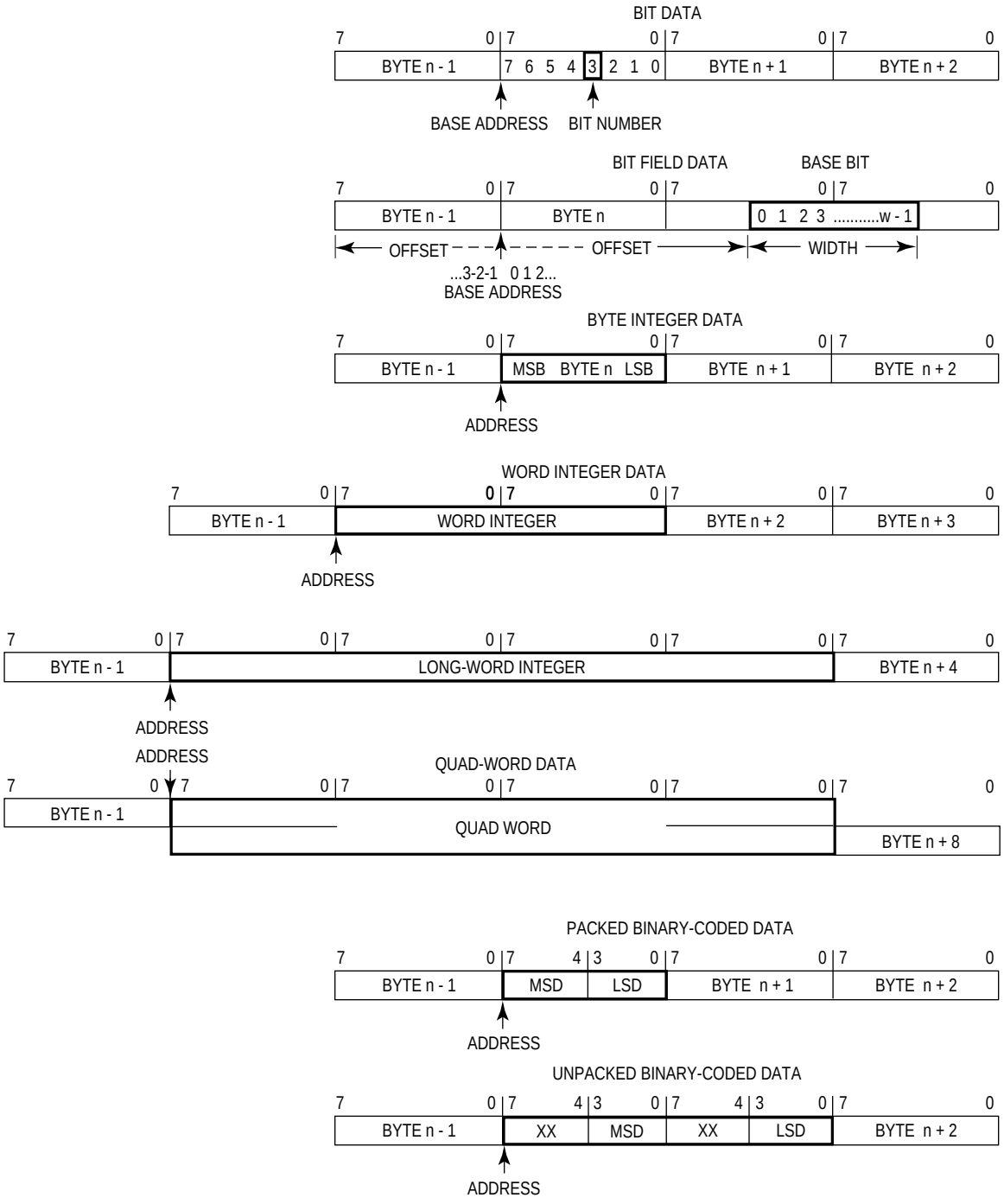


Figure 2-1. Memory Operand Address



XX = USER DEFINED VALUE

Figure 2-2. Memory Data Organization

A bit field operand is specified by:

1. A base address that selects one byte in memory,
2. A bit field offset that indicates the leftmost (base) bit of the bit field in relation to the most significant bit of the base byte, and
3. A bit field width that determines how many bits to the right of the base bit are in the bit field.

The most significant bit of the base byte is bit field offset 0, the least significant bit of the base byte is bit field offset 7, and the least significant bit of the previous byte in memory is bit offset –1. Bit field offsets may have values in the range of -2^{31} to $2^{31}-1$, and bit field widths may range between 1 and 32 bits.

2.4 ADDRESSING MODES

The addressing mode of an instruction can specify the value of an operand (with an immediate operand), a register that contains the operand (with the register direct addressing mode), or how the effective address of an operand in memory is derived. An assembler syntax has been defined for each addressing mode.

Figure 2–3 shows the general format of the single effective address instruction operation word. The effective address field specifies the addressing mode for an operand that can use one of the numerous defined modes. The (eaL designation is composed of two 3-bit fields: the mode field and the register field. The value in the mode field selects one or a set of addressing modes. The register field specifies a register for the mode or a submode for modes that do not use registers.

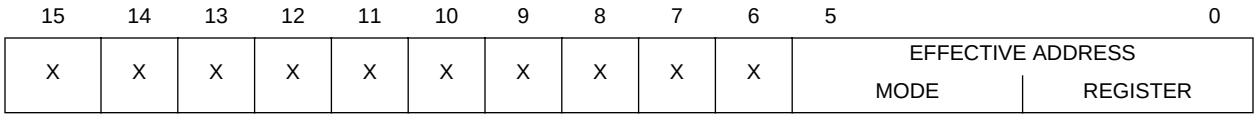


Figure 2-3. Single Effective Address

Many instructions imply the addressing mode for one of the operands. The formats of these instructions include appropriate fields for operands that use only one addressing mode.

The effective address field may require additional information to fully specify the operand address. This additional information, called the effective address extension, is contained in an additional word or words and is considered part of the instruction. Refer to **2.5 Effective Address Encoding Summary** for a description of the extension word formats.

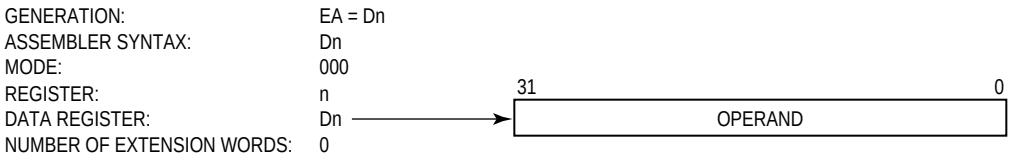
The notational conventions used in the addressing mode descriptions in this section are:

EA	—	Effective address
An	—	Address register n Example: A3 is address register 3
Dn	—	Data register n Example: D5 is data register 5
Xn.SIZE*SCALE	—	Denotes index register n (data or address), the index size (W for word, L for long word), and a scale factor (1, 2, 4, or 8 for no, word, long-word, or quad-word scaling, respectively).
PC	—	The program counter
d_n	—	Displacement value, n bits wide
bd	—	Base displacement
od	—	Outer displacement
L	—	Long-word size
W	—	Word size
( )	—	Identify an indirect address in a register
[ ]	—	Identify an indirect address in memory

When the addressing mode uses a register, the register field of the operation word specifies the register to be used. Other fields within the instruction specify whether the register selected is an address or data register and how the register is to be used.

2.4.1 Data Register Direct Mode

In the data register direct mode, the operand is in the data register specified by the effective address register field.



2.4.2 Address Register Direct Mode

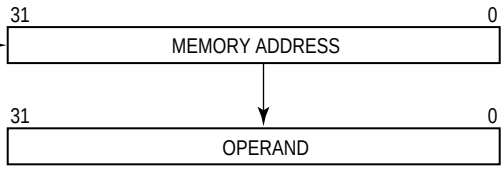
In the address register direct mode, the operand is in the address register specified by the effective address register field.

GENERATION: EA = An
ASSEMBLER SYNTAX: An
MODE: 001
REGISTER: n
ADDRESS REGISTER: An
NUMBER OF EXTENSION WORDS: 0

2.4.3 Address Register Indirect Mode

In the address register indirect mode, the operand is in memory, and the address of the operand is in the address register specified by the register field.

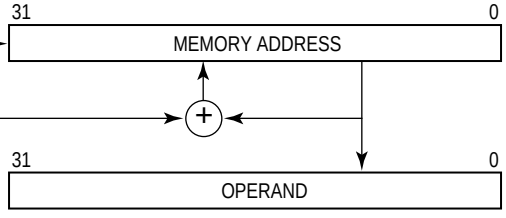
GENERATION: EA = (An)
ASSEMBLER SYNTAX: (An)
MODE: 010
REGISTER: n
ADDRESS REGISTER: An
MEMORY ADDRESS:
NUMBER OF EXTENSION WORDS: 0



2.4.4 Address Register Indirect with Postincrement Mode

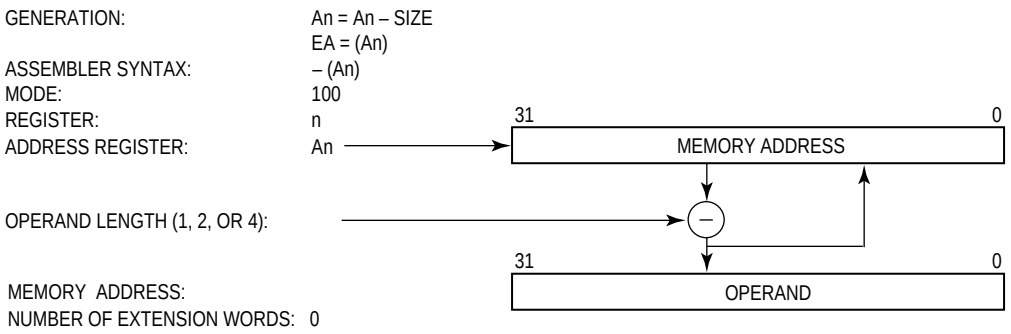
In the address register indirect with postincrement mode, the operand is in memory, and the address of the operand is in the address register specified by the register field. After the operand address is used, it is incremented by one, two, or four depending on the size of the operand: byte, word, or long word. Coprocessors may support incrementing for any size of operand up to 255 bytes. If the address register is the stack pointer and the operand size is byte, the address is incremented by two rather than one to keep the stack pointer aligned to a word boundary.

GENERATION: EA = (An)
An = An + SIZE
ASSEMBLER SYNTAX: (An) +
MODE: 011
REGISTER: n
ADDRESS REGISTER: An
OPERAND LENGTH (1, 2, OR 4):
MEMORY ADDRESS:
NUMBER OF EXTENSION WORDS: 0



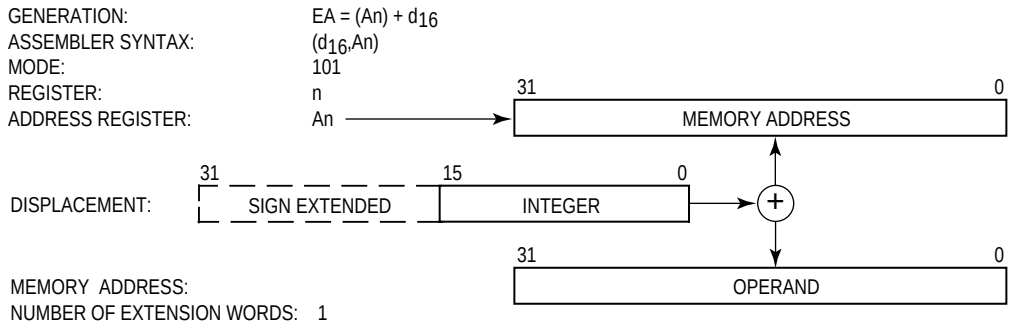
2.4.5 Address Register Indirect with Predecrement Mode

In the address register indirect with predecrement mode, the operand is in memory, and the address of the operand is in the address register specified by the register field. Before the operand address is used, it is decremented by one, two, or four depending on the operand size: byte, word, or long word. Coprocessors may support decrementing for any operand size up to 255 bytes. If the address register is the stack pointer and the operand size is byte, the address is decremented by two rather than one to keep the stack pointer aligned to a word boundary.



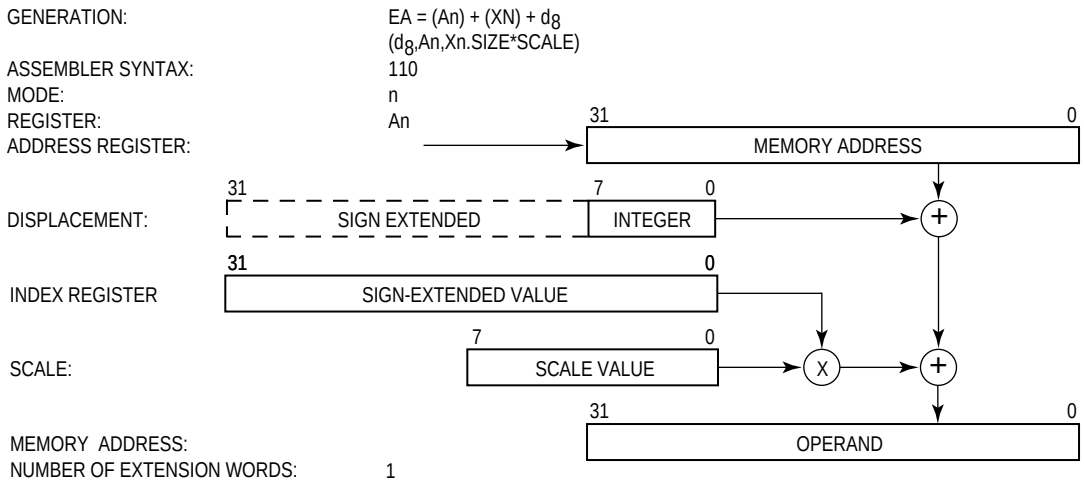
2.4.6 Address Register Indirect with Displacement Mode

In the address register indirect with displacement mode, the operand is in memory. The address of the operand is the sum of the address in the address register plus the sign-extended 16-bit displacement integer in the extension word. Displacements are always sign-extended to 32 bits prior to being used in effective address calculations.



2.4.7 Address Register Indirect with Index (8-Bit Displacement) Mode

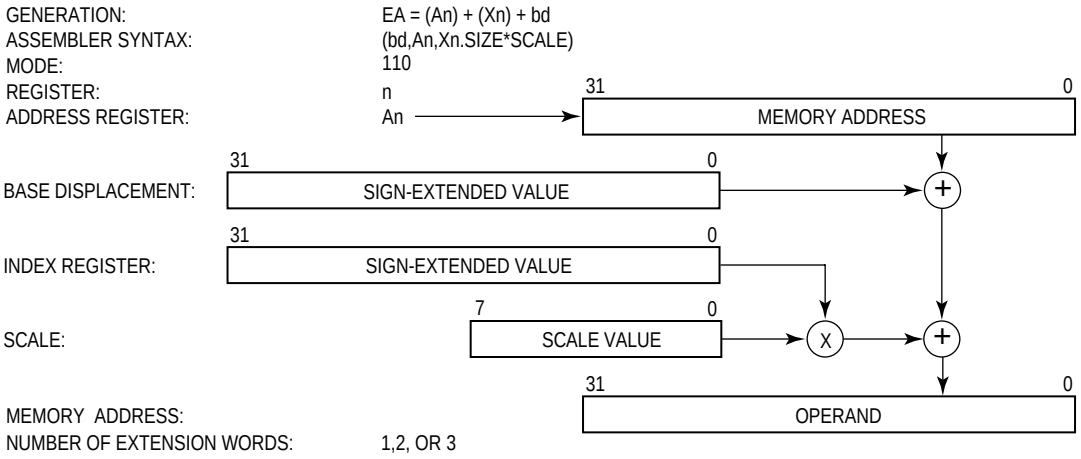
This addressing mode requires one extension word that contains the index register indicator and an 8-bit displacement. The index register indicator includes size and scale information. In this mode, the operand is in memory. The address of the operand is the sum of the contents of the address register, the sign-extended displacement value in the low-order eight bits of the extension word, and the sign-extended contents of the index register (possibly scaled). The user must specify the displacement, the address register, and the index register in this mode.



2.4.8 Address Register Indirect with Index (Base Displacement) Mode

This addressing mode requires an index register indicator and an optional 16- or 32-bit sign-extended base displacement. The index register indicator includes size and scaling information. The operand is in memory. The address of the operand is the sum of the contents of the address register, the scaled contents of the sign-extended index register, and the base displacement.

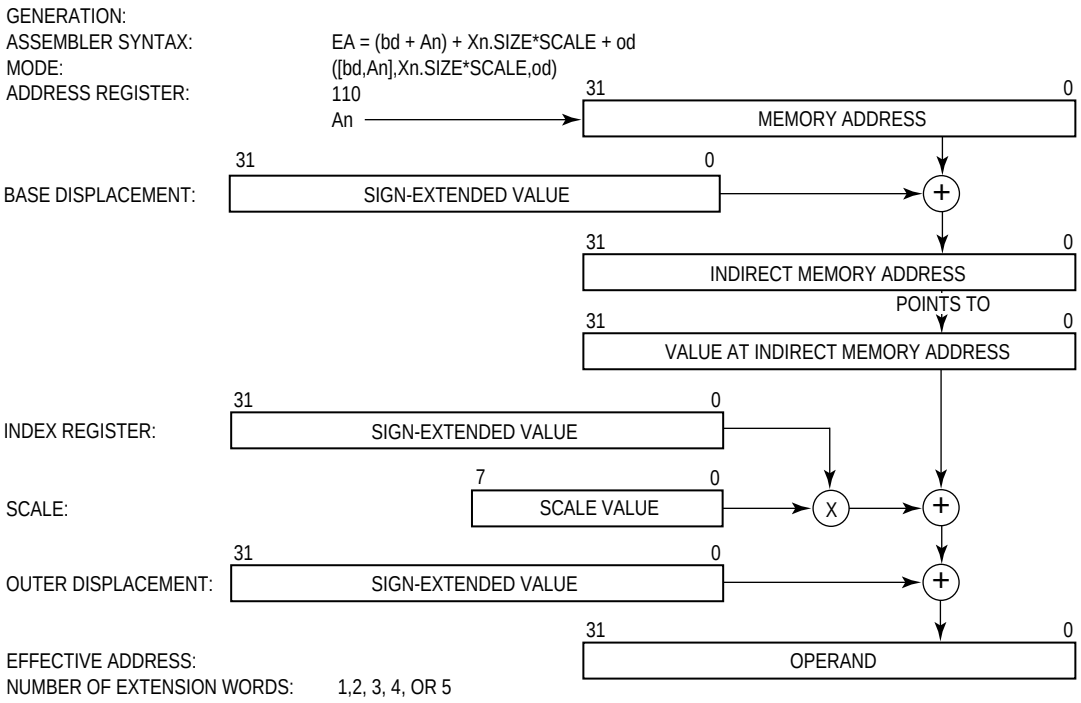
In this mode, the address register, the index register, and the displacement are all optional. If none is specified, the effective address is zero. This mode provides a data register indirect address when no address register is specified and the index register is a data register (Dn).



2.4.9 Memory Indirect Postindexed Mode

In this mode, the operand and its address are in memory. The processor calculates an intermediate indirect memory address using the base register (An) and base displacement (bd). The processor accesses a long word at this address and adds the index operand (Xn.SIZE*SCALE) and the outer displacement to yield the effective address. Both displacements and the index register contents are sign-extended to 32 bits.

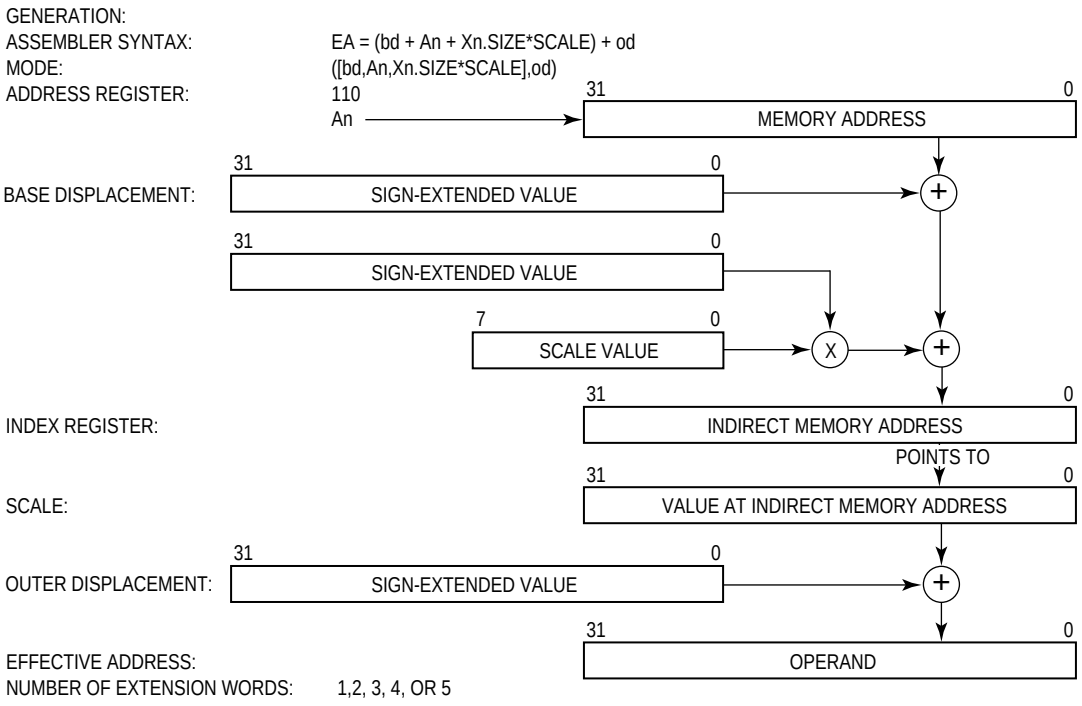
In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. Both the base and outer displacements may be null, word, or long word. When a displacement is omitted or an element is suppressed, its value is taken as zero in the effective address calculation.



2.4.10 Memory Indirect Preindexed Mode

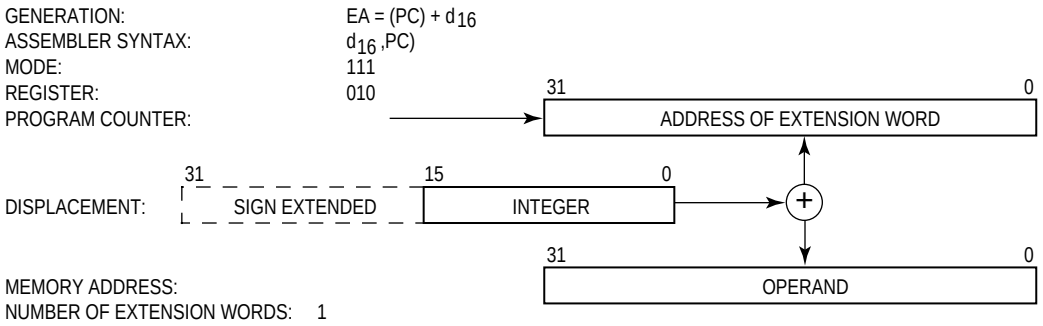
In this mode, the operand and its address are in memory. The processor calculates an intermediate indirect memory address using the base register (An), a base displacement (bd), and the index operand (Xn.SIZE * SCALE). The processor accesses a long word at this address and adds the outer displacement to yield the effective address. Both displacements and the index register contents are sign-extended to 32 bits.

In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. Both the base and outer displacements may be null, word, or long word. When a displacement is omitted or an element is suppressed, its value is taken as zero in the effective address calculation.



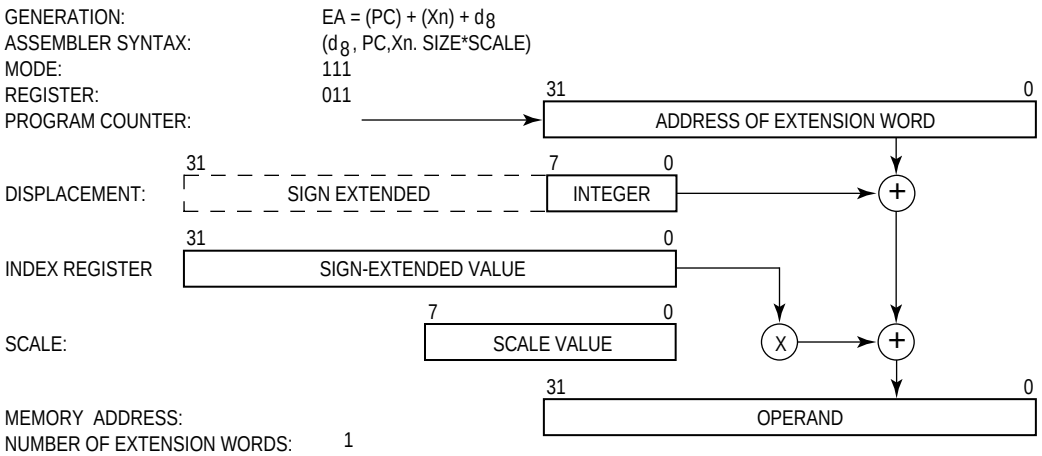
2.4.11 Program Counter Indirect with Displacement Mode

In this mode, the operand is in memory. The address of the operand is the sum of the address in the PC and the sign-extended 16-bit displacement integer in the extension word. The value in the PC is the address of the extension word. The reference is a program space reference and is only allowed for reads (refer to **4.2 Address Space Types**).



2.4.12 Program Counter Indirect with Index (8-Bit Displacement) Mode

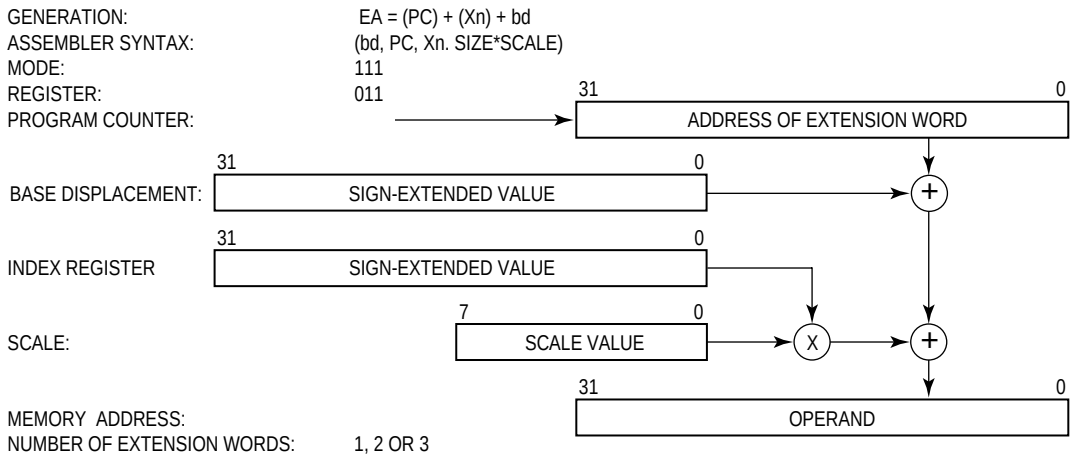
This mode is similar to the address register indirect with index (8-bit displacement) mode described in **2.4.7 Address Register Indirect with Index (8-Bit Displacement) Mode**, but the PC is used as the base register. The operand is in memory. The address of the operand is the sum of the address in the PC, the sign-extended displacement integer in the lower eight bits of the extension word, and the sized, scaled, and sign-extended index operand. The value in the PC is the address of the extension word. This reference is a program space reference and is only allowed for reads. The user must include the displacement, the PC, and the index register when specifying this addressing mode.



2.4.13 Program Counter Indirect with Index (Base Displacement) Mode

This mode is similar to the address register indirect with index (base displacement) mode described in **2.4.8 Address Register Indirect with Index (Base Displacement) Mode**, but the PC is used as the base register. It requires an index register indicator and an optional 16- or 32-bit sign-extended base displacement. The operand is in memory. The address of the operand is the sum of the contents of the PC, the scaled contents of the sign-extended index register, and the base displacement. The value of the PC is the address of the first extension word. The reference is a program space reference and is only allowed for reads (refer to **4.2 Address Space Types**).

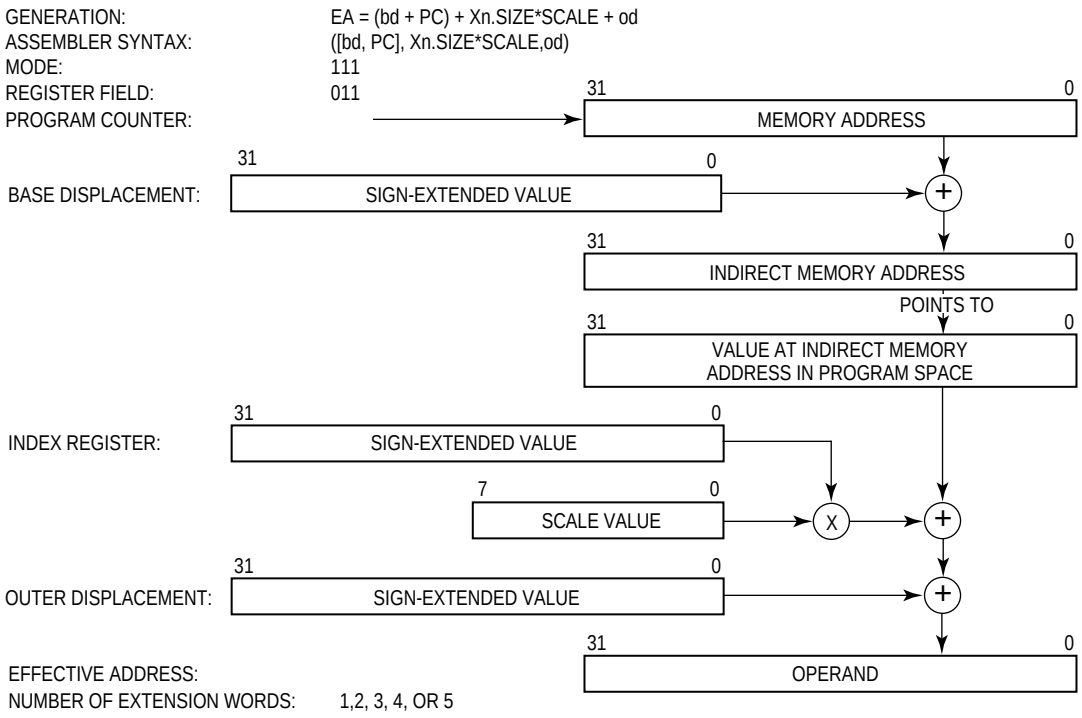
In this mode, the PC, the index register, and the displacement are all optional. However, the user must supply the assembler notation "ZPC" (zero value is taken for the PC) to indicate that the PC is not used. This allows the user to access the program space without using the PC in calculating the effective address. The user can access the program space with a data register indirect access by placing ZPC in the instruction and specifying a data register (Dn) as the index register.



2.4.14 Program Counter Memory Indirect Postindexed Mode

This mode is similar to the memory indirect postindexed mode described in **2.4.9 Memory Indirect Postindexed Mode**, but the PC is used as the base register. Both the operand and operand address are in memory. The processor calculates an intermediate indirect memory address by adding a base displacement (bd) to the PC contents. The processor accesses a long word at that address and adds the scaled contents of the index register and the optional outer displacement (od) to yield the effective address. The value of the PC used in the calculation is the address of the first extension word. The reference is a program space reference and is only allowed for reads (refer to **4.2 Address Space Types**).

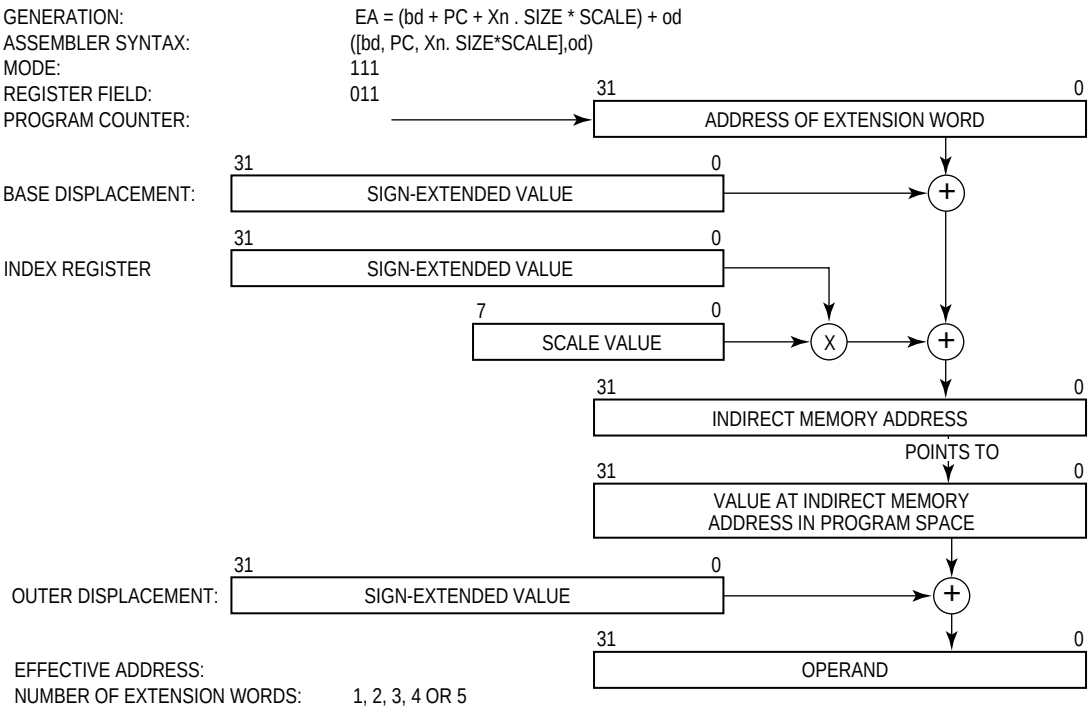
In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. However, the user must supply the assembler notation ZPC (zero value is taken for the PC) to indicate that the PC is not used. This allows the user to access the program space without using the PC in calculating the effective address. Both the base and outer displacements may be null, word, or long word. When a displacement is omitted or an element is suppressed, its value is taken as zero in the effective address calculation.



2.4.15 Program Counter Memory Indirect Preindexed Mode

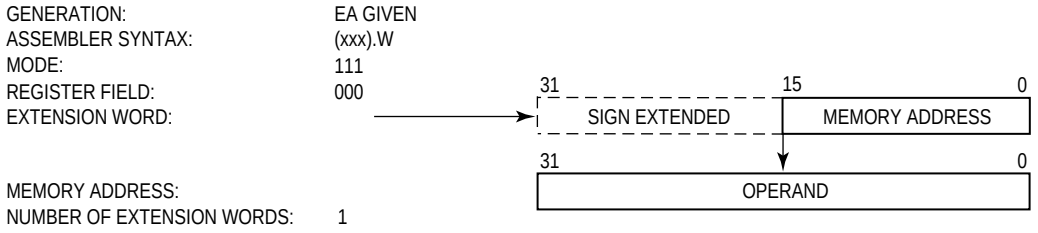
This mode is similar to the memory indirect preindexed mode described in **2.4.10 Memory Indirect Preindexed Mode**, but the PC is used as the base register. Both the operand and operand address are in memory. The processor calculates an intermediate indirect memory address by adding the PC contents, a base displacement (bd), and the scaled contents of an index register. The processor accesses a long word at that address and adds the optional outer displacement (od) to yield the effective address. The value of the PC is the address of the first extension word. The reference is a program space reference and is only allowed for reads (refer to **4.2 Address Space Types**).

In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. However, the user must supply the assembler notation ZPC (zero value is taken for the PC) to indicate that the PC is not used. This allows the user to access the program space without using the PC in calculating the effective address. Both the base and outer displacements may be null, word, or long word. When a displacement is omitted or an element is suppressed, its value is taken as zero in the effective address calculation.



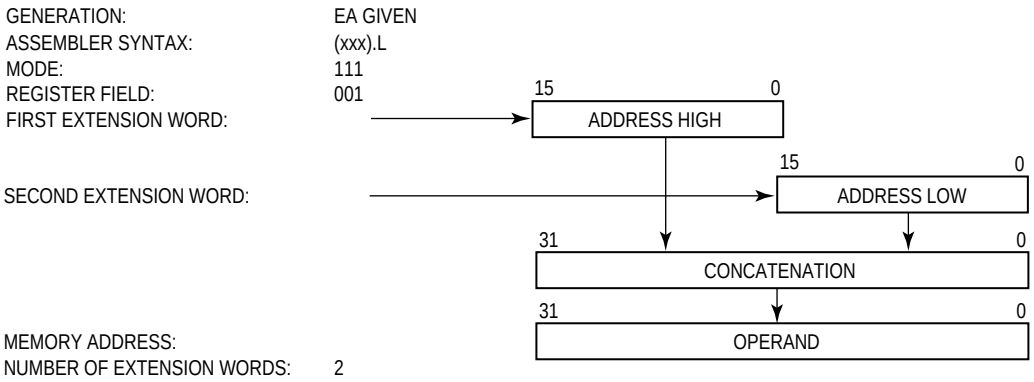
2.4.16 Absolute Short Addressing Mode

In this addressing mode, the operand is in memory, and the address of the operand is in the extension word. The 16-bit address is sign-extended to 32 bits before it is used.



2.4.17 Absolute Long Addressing Mode

In this mode, the operand is in memory, and the address of the operand occupies the two extension words following the instruction word in memory. The first extension word contains the high-order part of the address; the low-order part of the address is the second extension word.



2.4.18 Immediate Data

In this addressing mode, the operand is in one or two extension words:

Byte Operation

Operand is in the low-order byte of the extension word

Word Operation

Operand is in the extension word

Long-Word Operation

The high-order 16 bits of the operand are in the first extension word; the low-order 16 bits are in the second extension word.

Coprocessor instructions can support immediate data of any size. The instruction word is followed by as many extension words as are required.

```
Generation:                    Operand given
Assembler Syntax:              #xxx
Mode Field:                    111
Register Field:                100
Number of Extension Words:     1 or 2, except for coprocessor instructions
```

2.5 EFFECTIVE ADDRESS ENCODING SUMMARY

Most of the addressing modes use one of the three formats shown in Figure 2–4. The single effective address instruction is in the format of the instruction word. The encoding of the mode field of this word selects the addressing mode. The register field contains the general register number or a value that selects the addressing mode when the mode field contains "111." Table 2–2 shows the encoding of these fields. Some indexed or indirect modes use the instruction word followed by the brief format extension word. Other indexed or indirect modes consist of the instruction word and the full format of extension words. The longest instruction for the MC68030 contains 10 extension words. It is a MOVE instruction with full format extension words for both source and destination effective addresses and with 32-bit base displacements and 32-bit outer displacements for both addresses. However, coprocessor instructions can have any number of extension words. Refer to the coprocessor instruction formats in **Section 10 Coprocessor Interface Description**.

For effective addresses that use the full format, the index suppress (IS) bit and the index/ indirect selection (I/IS) field determine the type of indexing and indirection. Table 2–1 lists the indexing and indirection operations corresponding to all combinations of IS and I/IS values.

Table 2-1. IS–I/IS Memory Indirection Encodings

IS	Index/Indirect	Operation
0	000	No Memory Indirection
0	001	Indirect Preindexed with Null Outer Displacement
0	010	Indirect Preindexed with Word Outer Displacement
0	011	Indirect Preindexed with Long Outer Displacement
0	100	Reserved
0	101	Indirect Postindexed with Mull Outer Displacement
0	110	Indirect Postindexed with Word Outer Displacement
0	111	Indirect Postindexed with Long Outer Displacement
1	000	No Memory Indirection
1	001	Memory Indirect with Mull Outer Displacement
1	010	Memory Indirect with Word Outer Displacement
1	011	Memory Indirect with Long Outer Displacement
1	100–111	Reserved

Single Effective Address Instruction Format

15	14	13	12	11	10	9	8	7	6	5	0
X	X	X	X	X	X	X	X	X	X	EFFECTIVE ADDRESS MODE	EFFECTIVE ADDRESS REGISTER

Brief Format Extension Word

15	14 12	11	10 9	8	7 0
D/A	REGISTER	W/L	SCALE	0	DISPLACEMENT

Full Format Extension Word(s)

15	14 12	11	10 9	8	7	6	5 4	3	2 0
D/A	REGISTER	W/L	SCALE	1	BS	IS	BD SIZE	0	I/IS
BASE DISPLACEMENT (0, 1, OR 2 WORDS)									
OUTER DISPLACEMENT (0, 1, OR 2 WORDS)									

Field	Definition
Instruction:	
Register	General Register Number
Extensions:	
Register	Index Register Number
D/A	Index Register Type 0 = Dn 1 = An
W/L	Word/Long-Word Index Size 0 = Sign-Extended Word 1 = Long Word
Scale	Scale Factor 00 =1 01 =2 10 = 4 11 = 8
BS	Base Register Suppress: 0 = Base Register Added 1 = Base Register Suppressed
IS	Index Suppress: 0 = Evaluate and Add Index Operand 1 = Suppress Index Operand
BD SIZE	Base Displacement Size: 00 = Reserved 01 = Null Displacement 10 = Word Displacement 11 Long Displacement
I/IS	Index/Indirect Selection Indirect and Indexing Operand Determined in Conjunction with Bit 6, Index Suppress

Figure 2-4. Effective Address Specification Formats

Effective address modes are grouped according to the use of the mode. They can be classified as follows:

Data	A data addressing effective address mode is one that refers to data operands.
Memory	A memory addressing effective address mode is one that refers to memory operands.
Alterable	An alterable addressing effective address mode is one that refers to alterable (writable) operands.
Control	A control addressing effective address mode is one that refers to memory operands without an associated size.

Table 2–2 shows the categories to which each of the effective addressing modes belong.

Addressing Modes	Mode	Register	Data	Memory	Control	Alterable	Assembler Syntax
Data Register Direct	000	reg. no.	X	—	—	X	Dn
Address Register Direct	001	reg. no.	—	—	—	X	An
Address Register Indirect	010	reg. no	X	X	X	X	(An)
Address Register Indirect with Postincrement	011	reg. no.	X	X	—	X	(An)+
Address Register Indirect with Predecrement	100	reg. no.	X	X	—	X	-(An)
Address Register Indirect with Displacement	101	reg. no.	X	X	X	X	(d_{16},An)
Address Register Indirect with Index (8-Bit Displacement)	110	reg. no.	X	X	X	X	(d_8,An,Xn)
Address Register Indirect with Index (Base Displacement)	110	reg. no.	X	X	X	X	(bd,An,Xn)
Memory Indirect Postindexed	110	reg. no	X	X	X	X	([bd,An],Xn,od)
Memory Indirect Preindexed	110	reg. no.	X	X	X	X	([bd,An,Xn],od)
Absolute Short	111	000	X	X	X	X	(xxx).W
Absolute Long	111	001	X	X	X	X	(xxx).L
Program Counter Indirect with Displacement	111	010	X	X	X	—	(d_{16},PC)
Program Counter Indirect with Index (8-Bit) Displacement	111	011	X	X	X	—	(d_8,PC,Xn)
Program Counter Indirect with Index (Base Displacement)	111	011	X	X	X	—	(bd,PC,Xn)
PC Memory Indirect Postindexed	111	011	X	X	X	—	([bd,PC],Xn,od)
PC Memory Indirect Preindexed	111	011	X	X	X	—	([bd,PC,Xn],od)
Immediate	111	100	X	X	—	—	#⟨data⟩

These categories are sometimes combined, forming new categories that are more restrictive. Two combined classifications are alterable memory or data alterable. The former refers to those addressing modes that are both alterable and memory addresses, and the latter refers to addressing modes that are both data and alterable.

2.6 PROGRAMMER`S VIEW OF ADDRESSING MODES

Extensions to the indexed addressing modes, indirection, and full 32-bit displacements provide additional programming capabilities for both the MC68020 and the MC68030. This section describes addressing techniques that exploit these capabilities and summarizes the addressing modes from a programming point of view.

Several of the addressing techniques described in this section use data registers and address registers interchangeably. While the MC68030 provides this capability, its performance has been optimized for addressing with address registers. The performance of a program that uses address registers in address calculations is superior to that of a program that similarly uses data registers.The performance has been optimized for addressing registers in address calculations is superior to that of a program that similarly uses data registers. The specification of addresses with data registers should be used sparingly (if at all), particularly in programs that require maximum performance.

2.6.1 Addressing Capabilities

In both the MC68020 and the MC68030, setting the base register suppress (BS) bit in the full format extension word (see Figure 2–4) suppresses use of the base address register in calculating the effective address. This allows any index register to be used in place of the base register. Since any of the data registers can be index registers, this provides a data register indirect form (Dn). The mode could be called register indirect (Rn) since either a data register or an address register can be used. This addressing mode is an extension to the M68000 Family because the MC68030 and MC68020 can use both the data registers and the address registers to address memory. The capabilities of specifying the size and scale of an index register (Xn.SIZE*SCALE) in these modes provides additional addressing flexibility. Using the SIZE parameter, either the entire contents of the index register can be used, or the least significant word can be sign-extended to provide a 32-bit index value (refer to Figure 2–5).

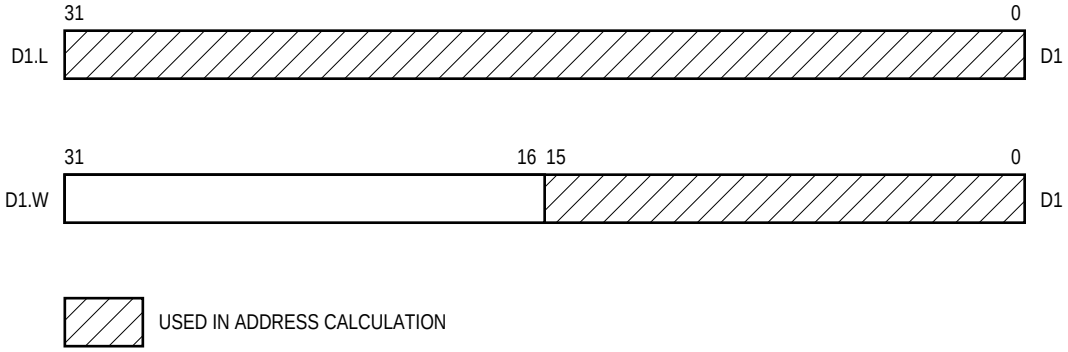


Figure 2-5. Using SIZE in the Index Selection

For both the MC68020 and the MC68030, the register indirect modes can be extended further. Since displacements can be 32 bits wide, they can represent absolute addresses or the results of expressions that contain absolute addresses. This allows the general register indirect form to be (bd,Rn) or (bd,An,Rn) when the base register is not suppressed. Thus, an absolute address can be directly indexed by one or two registers (refer to Figure 2–6).

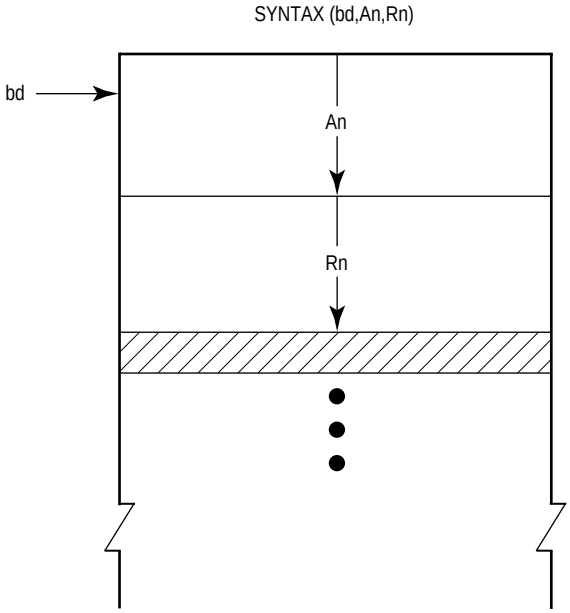
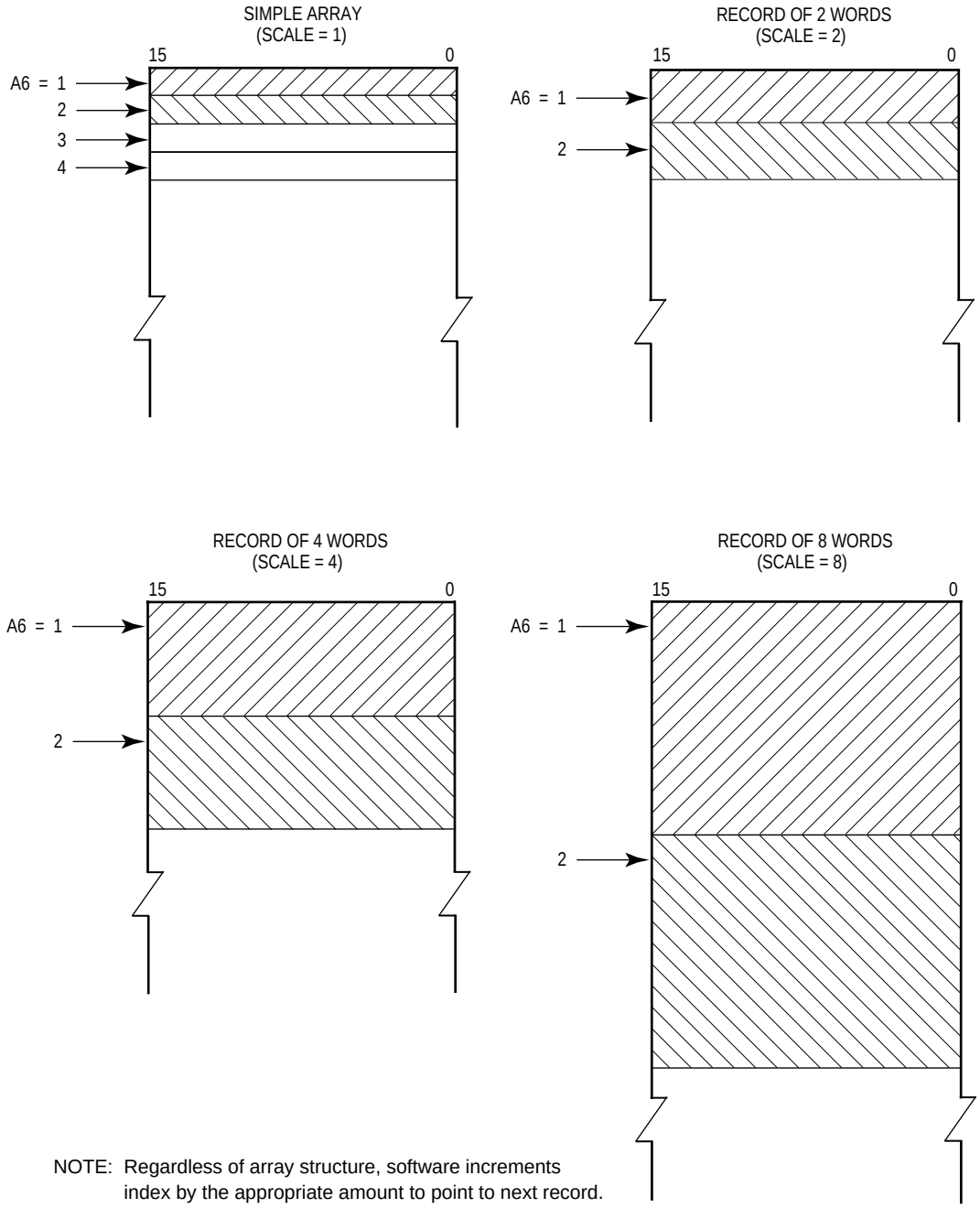


Figure 2-6. Using Absolute Address with Indexes

Scaling provides an optional shifting of the value in an index register to the left by zero, one, two, or three bits before using it in the effective address allocation (the actual value in the index register remains unchanged). This is equivalent to multiplying the register by one, two, four, or eight or direct subscripting into an array of elements of corresponding size using an arithmetic value residing in any of the 16 general registers. Scaling does not add to the effective address calculation time. However, when combined with the appropriate derived modes, it produces additional capabilities. Arrayed structures can be addressed absolutely and then subscripted, (bd,Rn*scale). Another variation that can be derived is (An,Rn*scale). In the first case, the array address is the sum of the contents of a register and a displacement, as shown in Figure 2–7. In the second example. An contains the address of an array and Rn contains a subscript.

SYNTAX: MOVE.W (A5, A6.L*SCALE),(A7)
WHERE:
A5 = ADDRESS OF ARRAY STRUCTURE
A6 = INDEX NUMBER OF ARRAY ITEM
A7 = STACK POINTER



NOTE: Regardless of array structure, software increments index by the appropriate amount to point to next record.

Figure 2-7. Addressing Array Items

The memory indirect addressing modes use a long-word pointer in memory to access an operand. Any of the modes previously described can be used to address the memory pointer. Because the base and index registers can both be suppressed, the displacement acts as an absolute address, providing indirect absolute memory addressing (refer to Figure 2–8).

The outer displacement (od) available in the memory indirect modes is added to the pointer in memory. The syntax for these modes is ([bd,An],Xn,od) and ([bd,An,Xn],od). When the pointer is the address of a structure in memory and the outer displacement is the offset of an item in the structure, the memory indirect modes can access the item efficiently (refer to Figure 2–9).

Memory indirect addressing modes are used with a base displacement in five basic forms:

1. [bd,An] — Indirect, suppressed index register
2. ([bd,An,Xn]) — Preindexed indirect
3. ([bd,An],Xn) — Postindexed indirect
4. ([bd,An,Xn],od) — Preindexed indirect with outer displacement
5. ([bd,An],Xn,od) — Postindexed indirect with outer displacement

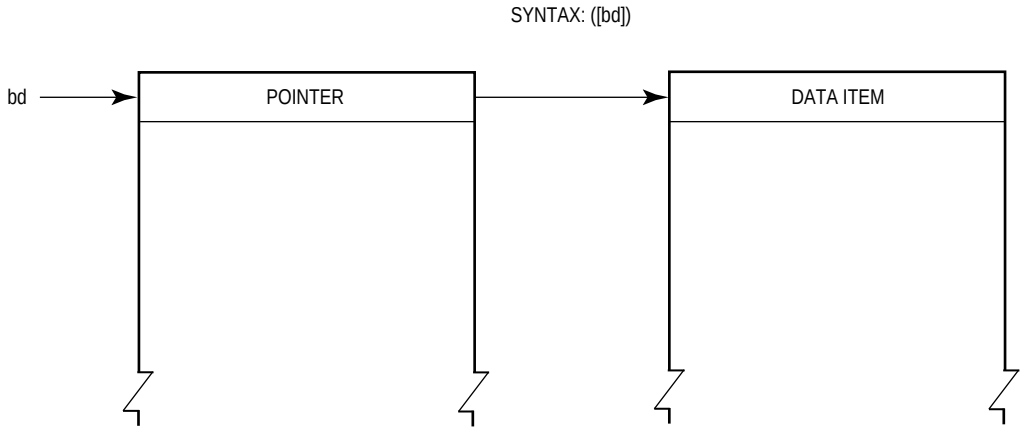


Figure 2-8. Using Indirect Absolute Memory Addressing

The indirect, suppressed index register mode (see Figure 2–10) uses the contents of register An as an index to the pointer located at the address specified by the displacement. The actual data item is at the address in the selected pointer.

The preindexed indirect mode (see Figure 2–11) uses the contents of An as an index to the pointer list structure at the displacement. Register Xn is the index to the pointer, which contains the address of the data item.

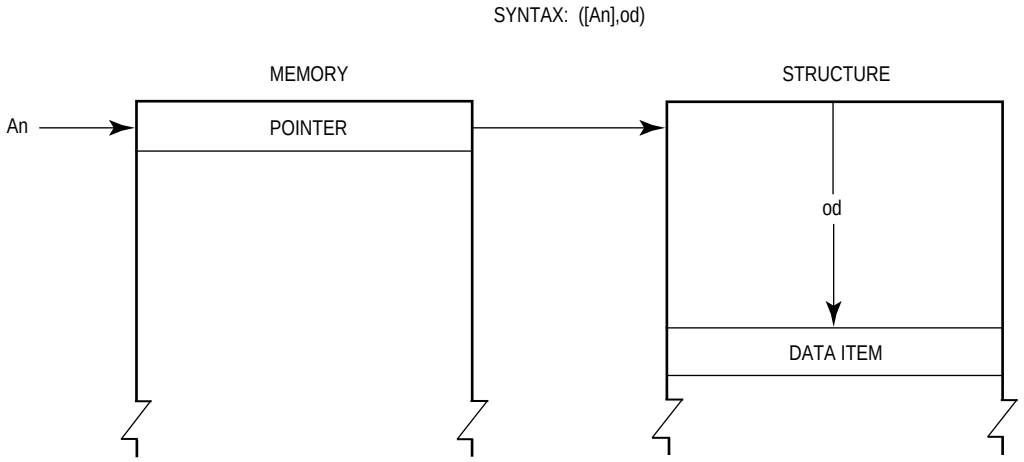


Figure 2-9. Accessing an Item in a Structure Using a Pointer

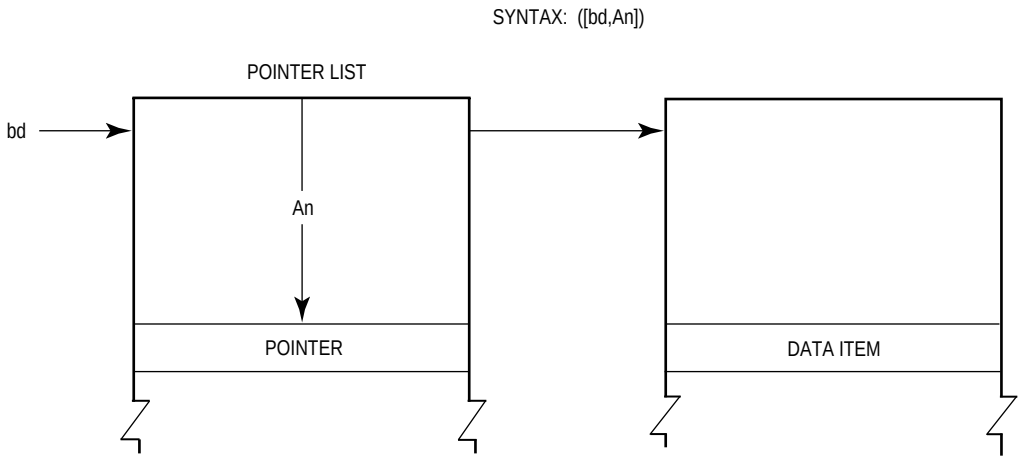


Figure 2-10. Indirect Addressing, Suppressed Index Register

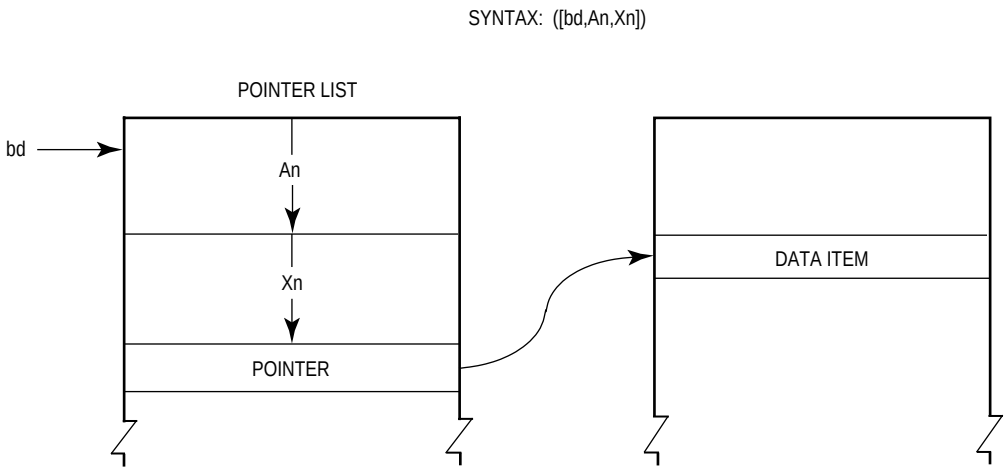


Figure 2-11. Preindexed Indirect Addressing

The postindexed indirect mode (see Figure 2–12) uses the contents of An as an index to the pointer list at the displacement. Register Xn is used as an index to the structure of data items located at the address specified by the pointer. Figure 2–13 shows the preindexed indirect addressing with outer displacement mode.

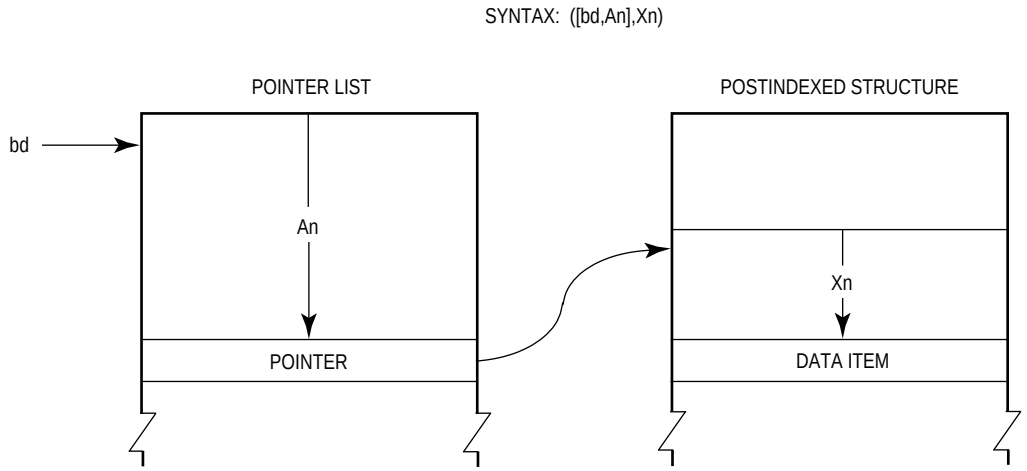


Figure 2-12. Postindexed Indirect Addressing

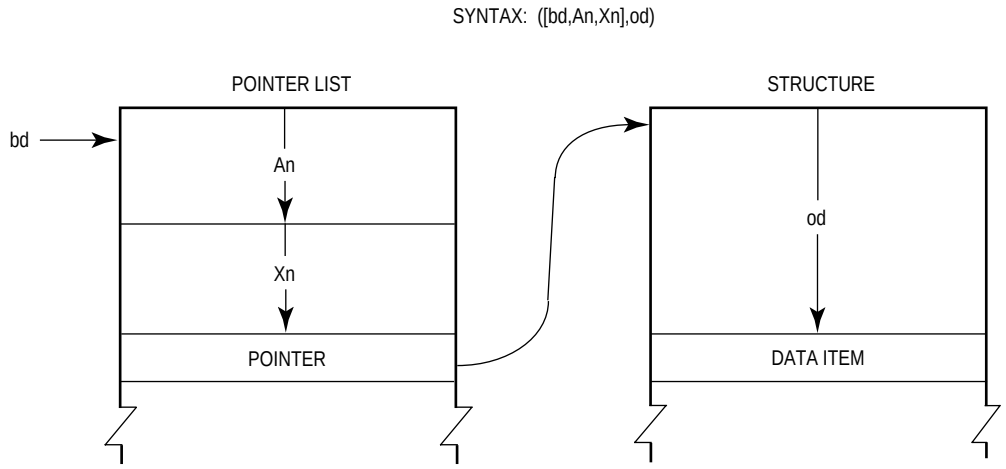


Figure 2-13. Preindexed Indirect Addressing with Outer Displacement

The postindexed indirect mode with outer displacement (see Figure 2–14) uses the contents of An as an index to the pointer list at the displacement. Register Xn is used as an index to the structure of data structures at the address in the pointer. The outer displacement (od) is the displacement of the data item within the selected data structure.

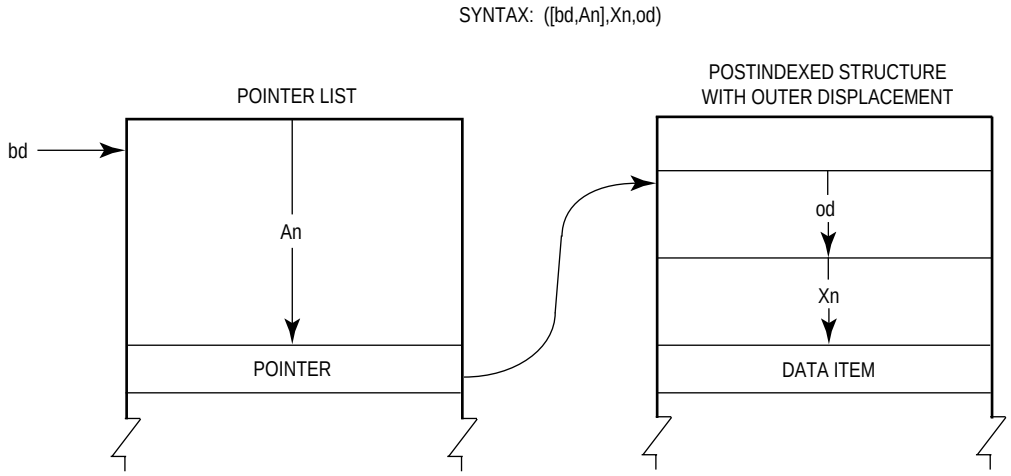


Figure 2-14. Postindexed Indirect Addressing with Outer Displacement

2.6.2 General Addressing Mode Summary

The addressing modes described in the previous section are derived from specific combinations of options in the indexing mode or a selection of two alternate addressing modes. For example, the addressing mode called register indirect (Rn) assembles as the address register indirect if the register is an address register. If Rn is a data register, the assembler uses the address register indirect with index mode using the data register as the indirect register and suppresses the address register by setting the base suppress bit in the effective address specification. Assigning an address register as Rn provides higher performance than using a data register as Rn. Another case is (bd,An), which selects an addressing mode depending on the size of the displacement. If the displacement is 16 bits or less, the address register indirect with displacement mode (d_{16},An) is used. When a 32-bit displacement is required, the address register indirect with index (bd,An,Xn) is used with the index register suppressed.

It is useful to examine the derived addressing modes available to a programmer (without regard to the MC68030 effective addressing mode actually encoded) because the programmer need not be concerned about these decisions. The assembler can choose the more efficient addressing mode to encode.

In the list of derived addressing modes that follows, common programming terms are used. The following definitions apply:

- pointer — long-Word value in a register or in memory which represents an address.
- base — A pointer combined with a displacement to represent an address.
- index — A constant or variable value added into an effective address calculation. A constant index is a displacement. A variable index is always represented by a register containing the value.
- disp — Displacement, a constant index.
- subscript — The use of any of the data or address registers as a variable index subscript into arrays of items 1, 2, 4 or 8 bytes in size.
- relative — An address calculated from the program counter contents. The address is position independent and is in program space. All other addresses but psaddr are in data space.
- addr — An absolute address.
- psaddr — An absolute address in program space. All other addresses but PC relative are in data space.
- preindexed — All modes from absolute address through program counter relative.
- postindexed — Any of the following modes:
  - addr — Absolute address in data space
  - psaddr,ZPC — Absolute address in program space
  - An — Register pointer with constant displacement
  - disp.An — Register pointer with constant displacement
  - addr,An — Absolute address with single variable name
  - disp,Pc — Simple PC relative

The addressing modes defined in programming terms, which are derivations of the addressing modes provided by the MC68030 architecture, are as follows:

Immediate Data — #data:
The data is a constant located in the instruction stream.

Register Direct — Rn:
The contents of a register contain the operand.

Scanning Modes:
(An)+

Address register pointer automatically incremented after use.

– (An)
Address register pointer automatically decremented before use.

Absolute Address:

(addr)
Absolute address in data space.

(psaddr,ZPC)
Absolute address in program space. Symbol ZPC suppresses the PC, but retains PC relative mode to directly access the program space.

Register Pointer:

(Rn)
Register as a pointer.

(disp,Rn)
Register as a pointer with constant index (or base address).

Indexing

(An,Rn)

Register pointer An with variable index Rn.

(disp,An,Rn)

Register pointer with constant and variable index (or a base address with a variable index).

(addr,Rn)

Absolute address with two variable indexes.

Subscripting:

(An,Rn*scale)

Address register pointer subscript.

(disp,An,Rn*scale)

Address register pointer subscript with constant displacement (or base address with subscript).

(addr,Rn*scale)

Absolute address with subscript.

(addr,An,Rn*scale)

Absolute address subscript with variable index.

Program Relative:

(disp,PC)

Simple PC relative.

(disp,PC,Rn)

PC relative with variable index.

(disp,PC,Rn*scale)

PC relative with subscript.

Memory Pointer:

([preindexed])

Memory pointer directly to data operand.

([preindexed],disp)

Memory pointer as base with displacement to data operand.

([postindexed],Rn)

Memory pointer with variable index.

([postindexed],disp,Rn)

Memory pointer with constant and variable index.

([postindexed],Rn*scale)

Memory pointer subscripted.

([postindexed],disp,Rn*scale)

Memory pointer subscripted with constant index.

2.7 M68000 FAMILY ADDRESSING COMPATIBILITY

Programs can be easily transported from one member of the M68000 Family to another in an upward compatible fashion. The user object code of each early member of the family is upward compatible with newer members and can be executed on the newer microprocessor without change. The address extension word(s) are encoded with the information that allows the MC68020/MC68030 to distinguish the new address extension words for the early MC68000/MC68008/MC68010 microprocessors and for the newer 32-bit MC68020/MC68030 microprocessors are shown in Figure 2–15. Notice the encoding for SCALE used by the MC68020/MC68030 is a compatible extension of the M68000 architecture. A value of zero for SCALE is the same encoding for both extension words; hence, software that uses this encoding is both upward and downward compatible across all processors in the product line. However, the other values of SCALE are not found in both extension formats; thus, while software can be easily migrated in an upward compatible direction, only nonscaled addressing is supported in a downward fashion. If the MC68000 were to execute an instruction that encoded a scaling factor, the scaling factor would be ignored and not access the desired memory address. The earlier microprocessors have no knowledge of the extension word formats implemented by newer processors; while they do detect illegal instructions, they do not decode invalid encodings of the extension words as exceptions.

2.8 OTHER DATA STRUCTURES

Stacks and queues are widely used data structures. The MC68030 implements a system stack and also provides instructions that support the use of user stacks and queues.

2.8.1 System Stack

Address register seven (A7) is used as the system stack pointer (SP). Any of the three system stack registers is active at any one time. The M and S bits of the status register determine which stack pointer is used. When S = 0 indicating user mode (user privilege level), the user stack pointer (USP) is the active system stack pointer, and the master and interrupt stack pointers cannot be referenced. When S = 1 indicating supervisor mode (at supervisor privilege level) and M = 1, the master stack pointer (MSP) is the active system stack pointer. When S = 1 and M = 0, the interrupt stack pointer (ISP) is the active system stack pointer. This mode is the MC68030 default mode after reset and corresponds to the MC68000, MC68008, and MC68010 supervisor mode. The term supervisor stack pointer (SSP) refers to the master or interrupt stack pointers, depending on the state of the M bit. When M = 1, the term SSP (or A7) refers to the MSP address register. When M = 0, the term is implicitly referenced by all instructions that use the system stack. Each system stack fills from high to low memory.

(UNABLE TO LOCATE ART. MUST BE RECREATED.)

Figure 2-15. M68000 Family Address Extension Words

A subroutine call saves the program counter on the active system stack, and the return restores it from the active system stack. During the processing of traps and interrupts, both the program counter and the status register are saved on the supervisor stack (either master or interrupt). Thus, the execution of supervisor code is independent of user code and the condition of the user stack; conversely, user programs use the user stack pointer independently of supervisor stack requirements.

To keep data on the system stack aligned for maximum efficiency, the active stack pointer is automatically decremented or incremented by two for all byte-sized operands moved to or from the stack. In long-word-organized memory, aligning the stack pointer on a long-word address signed significantly increases the efficiency of stacking exception frames, subroutine calls and returns, and other stacking operations.

2.8.2 User Program Stacks

The user can implement stacks with the address register indirect with postincrement and predecrement addressing modes. With address register An (n = 0–6), the user can implement a stack that is filled wither from high to low memory or from low to high memory. Important considerations are:

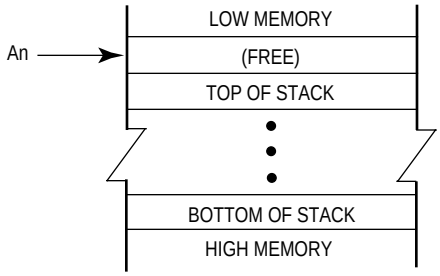
- Use the predecrement mode to decrement the register before its contents are used as the pointer to the stack.
- Use the postincrement mode to increment the register after its contents are used as the pointer to the stack.
- Maintain the stack pointer correctly when byte, word, and long-word items are mixed in these stacks.

To implement stack growth from high to low memory, use:

–(An) to push data on the stack,

(An)+ to pull data from the stack.

For this type of stack, after either a push or a pull operation, register An points to the top item on the stack. This is illustrated as:

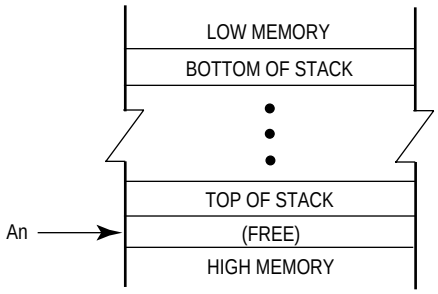


To implement stack growth from low to high memory, use:

(An)+ to push data on the stack,

–An to pull data from the stack.

In this case, after either a push or pull operation, register An points to the next available space on the stack. This is illustrated as:



2.8.3 Queues

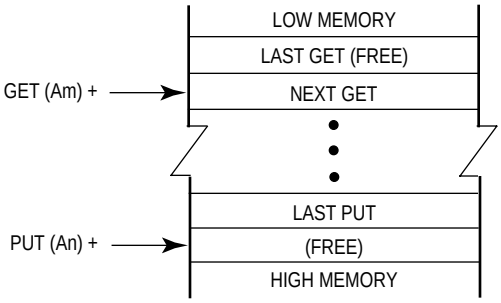
The user can implement queues with the address register indirect with postincrement or predecrement addressing modes. Using a pair of address registers (who of A0–A6), the user can implement a queue which is filled either from high to low memory or from low to high memory. Two registers are used because queues are pushed from one end and pulled from the other. One register, An, contains the "put" pointer; the other, Am, the "get" pointer.

To implement growth of the queue from low to high memory, use:

(An)+ to put data into the queue,

(Am)+ to get data from the queue.

After a "put" operation, the "put" address register points to the next available space in the queue, and the unchanged "get" address register points to the next item to be removed from the queue. After a "get" operation, the "get" address register points to the next item to be removed from the queue, and the unchanged "put" address register points to the next available space in the queue. This is illustrated as:



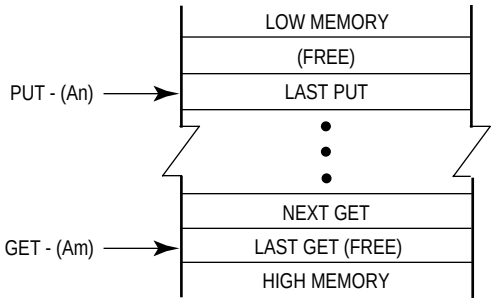
To implement the queue as a circular buffer, the relevant address register should be checked and adjusted, if necessary, before performing the "put" or "get" operation. The address register is adjusted by subtracting the buffer length (in bytes) from the register.

To implement growth of the queue from high to low memory, use:

–(An) to put data into the queue,

–(Am) to get data from the queue.

After a "put" operation, the "put" address register points to the last item place din the queue, and the unchanged "get" address register points to the last item removed from the queue. After a "get" operation, the "get" address register points to the last item removed from the queue, and the unchanged "put" address register points to the last item placed in the queue. This is illustrated as:



To implement the queue as a circular buffer, the "get" or "put" operation should be performed first, and then the relevant address register should be checkout and adjusted, if necessary. The address register is adjusted by adding the buffer length (in bytes) to the register contents.

SECTION 3
INSTRUCTION SET SUMMARY

This section briefly describes the MC68030 instruction set. Refer to the MC68000PM/AD, *MC68000 Programmer's Reference Manual*, for complete details on the MC68030 instruction set.

The following paragraphs include descriptions of the instruction format and the operands used by instructions, followed by a summary of the instruction set. The integer condition codes and floating-point details are discussed. Programming examples for selected instructions are also presented.

3.1 INSTRUCTION FORMAT

All MC68030 instructions consist of at least one word; some have as many as 11 words (see Figure 3–1). The first word of the instruction, called the operation word, specifies the length of the instruction and the operation to be performed. The remaining words, called extension words, further specify the instruction and operands. These words may be floating-point command words, conditional predicates, immediate operands, extensions to the effective address mode specified in the operation word, branch displacements, bit number or bit field specifications, special register specifications, trap operands, pack/unpack constants, or argument counts.

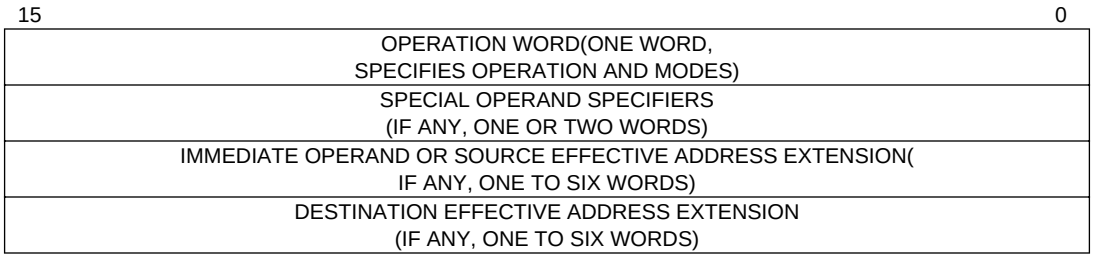


Figure 3-1. Instruction Word General Format

Besides the operation code, which specifies the function to be performed, an instruction defines the location of every operand for the function. Instructions specify an operand location in one of three ways:

1. Register Specification — A register field of the instruction contains the number of the register.
2. Effective Address — An effective address field of the instruction contains address mode information.
3. Implicit Reference — The definition of an instruction implies the use of specific registers.

The register field within an instruction specifies the register to be used. Other fields within the instruction specify whether the register selected is an address or data register and how the register is to be used. **Section 1 Introduction** contains register information.

Effective address information includes the registers, displacements, and absolute addresses for the effective address mode. **Section 2 Data Organization and Addressing Capabilities** describes the effective address modes in detail.

Certain instructions operate on specific registers. These instructions imply the required registers.

3.2 INSTRUCTION SUMMARY

The instructions form a set of tools to perform the following operations:

Data Movement	Bit Field Manipulation
Integer Arithmetic	Binary-Coded Decimal Arithmetic
Logical	Program Control
Shift and Rotate	System Control
Bit Manipulation	Multiprocessor Communications

Each instruction type is described in detail in the following paragraphs

The following notations are used in this section. In the operand syntax statements of the instruction definitions, the operand on the right is the destination operand.

An	=	any address register, A7–A0
Dn	=	any data register, D7–D0
Rn	=	any address or data register
CCR	=	condition code register (lower byte of status register)
cc	=	condition codes from CCR
SR	=	status register
SP	=	active stack pointer
USP	=	user stack pointer
ISP	=	supervisor/interrupt stack pointer
MSP	=	supervisor/master stack pointer
SSP	=	supervisor (master or interrupt) stack pointer
DFC	=	destination function code register
SFC	=	source function code register
Rc	=	control register (VBR, SFC, DFC, CACR)
MRc	=	MMU control register (SRP, URP, TC, DTT0, DTT1, ITT0, ITT1, MMUSR)
MMUSR	=	MMU status register
B, W, L	=	specifies a signed integer data type (twos complement) of byte, word, or long word
S	=	single-precision real data format (32 bits)
D	=	double-precision real data format (64 bits)
X	=	extended-precision real data format (96 bits, 16 bits unused)
P	=	packed BCD real data format (96 bits, 12 bytes)
FPm, FPn	=	any floating-point data register, FP7-FP0
PFcr	=	floating-point system control register (FPCR, FPSR, or FPIAR)
k	=	a twos-complement signed integer (–64 to +17) that specifies the format of a number to be stored in the packed BCD format
d	=	displacement; d_{16} is a 16-bit displacement
⟨ea⟩	=	effective address
list	=	list of registers, for example D3 — D0
#⟨data⟩	=	immediate data; a literal integer
{offset:width}	=	bit field selection
label	=	assemble program label
[m]	=	bit m of an operand
[m:n]	=	bits m through n of operand

X = extend (X) bit in CCR
N = negative (N) bit in CCR
Z = Zero (Z) bit in CCR
V = overflow (V) bit in CCR
C = carry (C) bit in CCR
+ = arithmetic addition or postincrement indicator
– = arithmetic subtraction or predecrement indicator
x = arithmetic multiplication
÷ = arithmetic division or conjunction symbol
~ = invert; operand is logically complemented
Λ = logical AND
V = logical OR
⊕ = logical exclusive OR
Dc = data register, D7-D0 used during compare
Du = data register, D7-D0 used during update
Dr, Dq = data registers, remainder or quotient of divide
Dh, Dl = data registers, high or lo•order 32 bits of product
MSW = most significant word
LSW = least significant word
MSB = most significant bit
FC = function code
{R/W} = read or write indicator
[An] = address extensions

3.2.1 Data Movement Instructions

The MOVE instructions with their associated addressing modes are the basic means of transferring and storing addresses and data. MOVE instructions transfer byte, word, and long-word operands from memory to memory, memory to register, register to memory, and register to register. Address movement instructions (MOVE or MOVEA) transfer word and long-word operands and ensure that only valid address manipulations are executed. In addition to the general MOVE instructions, there are several special data movement instructions: move multiple registers (MOVEM), move peripheral data (MOVEP), move quick (MOVEQ), exchange registers (EXG), load effective address (LEA), push effective address (PEA), link stack (LINK), and unlink stack (UNLK).

Table 3–1 is a summary of the integer and floating-point data movement operations.

Table 3-1. Data Movement Operations

Instruction	Operand Syntax	Operand Size	Operation
EXG	Rn,Rn	32	Rn ↔ Rn
LEA	⟨ea⟩,An	32	⟨ea⟩ → An
LINK	An,#⟨d⟩	16, 32	Sp - 4 → SP; An → (SP); SP → An, SP + D → SP
MOVE MOVEA	⟨ea⟩,⟨ea⟩,An	8, 16, 32 16, 32 → 32	source → destination
MOVEM	list,⟨ea⟩,list	16, 32 16, 32 → 32	listed registers → destination source → listed registers
MOVEP	Dn,(d_{16},An) (d_{16},An),Dn	16, 32	Dn[31:24] → (An + d); Dn[23:16] → An + d + 2); Dn[15:8] → (An + d + 4); Dn[7:0] → (An + d + 6) (An + d) → Dn[31:24]; (An + d + 2) → Dn[23:16]; (An + d + 4) → Dn[15:8]; (An + d + 6) → Dn[7:0]
MOVEQ	#⟨data⟩,Dn	8 → 32	immediate data → destination
PEA	⟨ea⟩	32	SP — 4 → SP; ⟨ea⟩ → (SP)
UNLK	An	32	An → SP; (SP) → An; SP + 4 → SP

3.2.2 Integer Arithmetic Instructions

The integer arithmetic operations include the four basic operations of add (ADD), subtract (SUB), multiply (MUL), and divide (DIV) as well as arithmetic compare (CMP, CMPM, CMP2), clear (CLR), and negate (NEG). The instruction set includes ADD, CMP, and SUB instructions for both address and data operations with all operand sizes valid for data operations. Address operands consist of 16 or 32 bits. The clear and negate instructions apply to all sizes of data operands.

Signed and unsigned MUL and DIV instructions include:

- Word multiply to produce a long-word product
- Long-word multiply to produce and long-word or quad-word product
- Division of a long word divided by a word divisor (word quotient and word remainder)
- Division of a long word or quad word dividend by a long-word divisor (long-word quotient and long-word remainder)

A set of extended instructions provides multiprecision and mixed-size arithmetic. These instructions are add extended (ADDX), subtract extended (SUBX), sign extended (EXT), and negate binary with extend (NEGX). Refer to Table 3–2 for a summary of the integer arithmetic operations.

Table 3-2. Integer Arithmetic Operations

Instruction	Operand Syntax	Operand Size	Operation
ADD ADDA	Dn,⟨ea⟩ ⟨ea⟩,Dn ⟨ea⟩,An	8, 16, 32 8, 16, 32 16, 32	source + destination → destination
ADDI ADDQ	#⟨data⟩,⟨ea⟩#⟨data⟩, ⟨ea⟩	8, 16, 32 8, 16, 32	immediate data + destination → destination
ADDX	Dn,Dn –(An),–(An)	8, 16, 32 8, 16, 32	source + destination + X → destination
CLR	⟨ea⟩	8, 16, 32	0 → destination
CMP CMPA	⟨ea⟩,Dn ⟨ea⟩,An	8, 16, 32 16, 32	destination - source
CMPI	#⟨data⟩,⟨ea⟩	8, 16, 32	destination - immediate data
CMPM	(An) +,(An) +	8, 16, 32	destination - source
CMP2	⟨ea⟩,Rn	8, 16, 32	lower bound < = Rn < = upper bound
DIVS/DIVU DIVSL/DIVUL	⟨ea⟩,Dn ⟨ea⟩,Dr:Dq ⟨ea⟩,Dq ⟨ea⟩,Dr:Dq	32/16 → 16:16 64/32 → 32:32 32/32 → 32 32/32 → 32:32	destination/source → destination (signed or unsigned)
EXT EXTB	Dn Dn Dn	8 → 16 16 → 32 8 → 32	sign-extended destination → destination
MULS/MULU	⟨ea⟩,Dn ⟨ea⟩,Dl (ea),Dh:Dl	16x16 → 32 32x32 → 32 32x32 → 64	source y destination → destination (signed or unsigned)
NEG	⟨ea⟩	8, 16, 32	0 - destination → destination
NEGX	⟨ea⟩	8, 16, 32	0 - destination - X → destination
SUB SUBA	⟨ea⟩,Dn Dn,⟨ea⟩ ⟨ea⟩,An	8, 16, 32 8, 16, 32 16, 32	destination = source → destination
SUBI SUBQ	#⟨data⟩,⟨ea⟩ #⟨data⟩,⟨ea⟩	8, 16, 32 8, 16, 32	destination - immediate data → destination
SUBX	Dn,Dn –(An),–(An)	8, 16, 32 8, 16, 32	destination - source — X → destination

3.2.3 Logical Instructions

The logical operation instructions (AND, OR, EOR, and NOT) perform logical operations with all sizes of integer data operands. A similar set of immediate instructions (ANDI, ORI, and EORI) provide these logical operations with all sizes of immediate data. The TST instruction compares the operand with zero arithmetically, placing the result in the condition code register. Table 3–3 summarizes the logical operations.

Table 3-3. Logical Operations

Instruction	Operand Syntax	Operand Size	Operation
AND	⟨ea⟩,Dn Dn,⟨ea⟩	8, 16, 32 8, 16, 32	source Λ destination → destination
ANDI	#⟨data⟩,⟨ea⟩	8, 16, 32	immediate data Λ destination → destination
EOR	Dn,⟨data⟩,⟨ea⟩	8, 16, 32	source ⊕ destination → destination
EORI	#⟨data⟩,⟨ea⟩	8, 16, 32	immediate data x destination → destination
NOT	⟨ea⟩	8, 16, 32	~ destination → destination
OR	⟨ea⟩,Dn Dn,⟨ea⟩	8, 16, 32 8, 16, 32	source V destination → destination
ORI	#⟨data⟩,⟨ea⟩	8, 16, 32	immediate data V destination → destination
TST	#⟨ea⟩	8, 16, 32	source — 0 to set condition codes

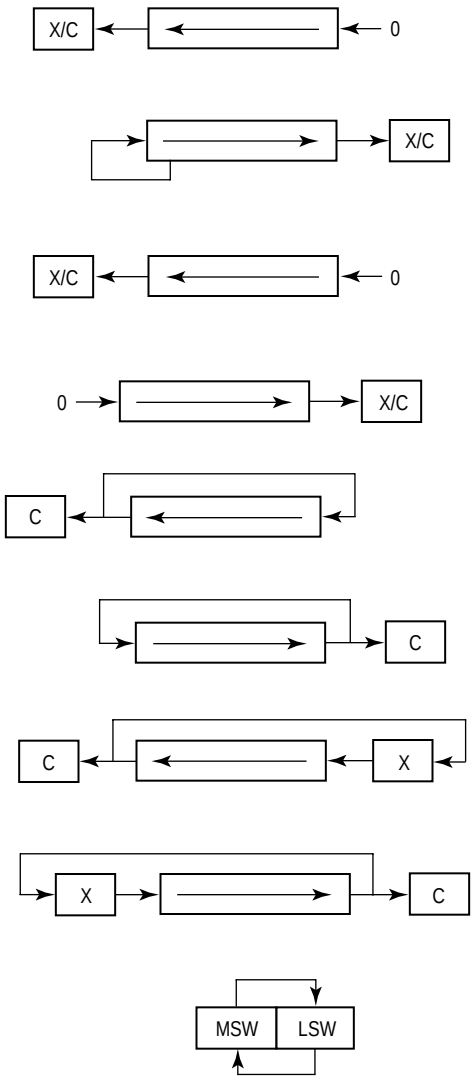
3.2.4 Shift and Rotate Instructions

The arithmetic shift instructions (ASR and ASL) and logical shift instructions (LSR and LSL) provide shift operations in both directions. The ROR, ROL, ROXR, and ROXL instructions perform rotate (circular shift) operations, with and without the extend bit. All shift and rotate operations can be performed on either registers or memory.

Register shift and rotate operations shift all operand sizes. The shift count may be specified in the instruction operation word (to shift from 1–8 places) or in a register (modulo 64 shift count).

Memory shift and rotate operations shift word-length operands one bit position only. The SWAP instruction exchanges the 16-bit halves of a register. Performance of shift/rotate instructions is enhanced so that use of the ROR and ROL instructions with a shift count of eight allows fast byte swapping. Table 3–4 is a summary of the shift and rotate operations.

Table 3-4. Shift and Rotate Operations.



3.2.5 Bit Manipulation Instructions

Bit manipulation operations are accomplished using the following instructions: bit test (BTST), bit test and set (BSET), bit test and clear (BCLR), and bit test and change (BCHG). All bit manipulation operations can be performed on either registers or memory. The bit number is specified as immediate data or in a data register. Register operands are 32 bits long, and memory operands are 8 bits long. In Table 3–5, the summary of the bit manipulation operations, Z refers to bit 2, the zero bit of the status register.

Table 3-5. Bit Manipulation Operations

Instruction	Operand Syntax	Operand Size	Operation
BCHG	Dn,⟨ea⟩ #⟨data⟩,ea	8, 32 8, 32	~ (⟨bit number⟩ of destination) → Z → bit of destination
BCLR	Dn,⟨ea⟩ #⟨data⟩,ea	8, 32 8, 32	~ (⟨bit number⟩ of destination) → Z; — 0 → bit of destination
BSET	Dn,⟨ea⟩ #⟨data⟩,⟨ea⟩	8, 32 8, 32	~ (⟨bit number⟩ of destination) → Z; — 1 → bit of destination
BTST	Dn,⟨ea⟩ #⟨data⟩,ea	8, 32 8, 32	~ (⟨bit number⟩ of destination) → Z

3.2.6 Bit Field Operations

The MC68030 supports variable-length bit field operations on fields of up to 32 bits. The bit field insert (BFINS) instruction inserts a value into a bit field. Bit field extract unsigned (BFEXTU) and bit field extract signed (BFEXTS) extract a value from the field. Bit field find first one (BFFFO) finds the first bit that is set in a bit field. Also included are instructions that are analogous to the bit manipulation operations; bit field test (BFTST), bit field test and set (BFSET), bit field test and clear (BFCLR), and bit field test and change (BFCHG). Table 3–6 is a summary of the bit field operations.

Table 3-6. Bit Field Operations

Instruction	Operand Syntax	Operand Size	Operation
BFCHG	⟨ea⟩ {offset:width}	1 — 32	~ Field → Field
BFCLR	⟨ea⟩ {offset:width}	1 — 32	0's → Field
BFEXTS	⟨ea⟩ {offset:width},Dn	1—32	Field → Dn; Sign Extended
BFEXTU	⟨ea⟩ {offset:width},Dn	1 — 32	Field → Dn; Zero Extended
BFFFO	⟨ea⟩ {offset:width},Dn	1 — 32	Scan for first bit set in field; offset → Dn
BFINS	Dn,⟨ea⟩ {offset:width}	1 — 32	Dn → Field
BFSET	⟨ea⟩ {offset:width}	1 — 32	1's → Field
BFTST	⟨ea⟩ {offset:width}	1 — 32	Field MSB → N; ~ (OR of all bits in field) → Z

NOTE: All bit field instructions set the N and Z bits as shown for BFTST before performing the specified operation.

3.2.7 Binary–coded Decimal Instructions

Five instructions support operations on binary-coded decimal (BCD) numbers. The arithmetic operations on packed BCD numbers are add decimal with extend (ABCD), subtract decimal with extend (SBCD), and negate decimal with extend (NBCD). PACK and UNPACK instructions aid in the conversion of byte encoded numeric data, such as ASCII or EBCDIC strings, to BCD data and vice versa. Table 3–7 is a summary of the BCD operations.

Table 3-7. BCD Operations

Instruction	Operand Syntax	Operand Size	Operation
ABCD	Dn,Dn –(An)	8 8	$source_{10}$ + $destination_{10}$ + X → destination
NBCD	⟨ea⟩	8	0 - $destination_{10}$ –X → destination
PACK	–(An),–(An) #⟨data⟩ Dn,Dn,# ⟨data⟩	16→8 16→8	unpackaged source + immediate data → packed destination
SBCD	Dn,Dn –(An),–(An)	8 8	$destination_{10}$ - $source_{10}$ – X → destination
UNPK	–(An) #⟨data⟩ Dn,Dn,#⟨data⟩	8→16 8→16	packed source → unpacked source unpacked source + immediate data → unpacked destination

3.2.8 Program Control Instructions

A set of subroutine call and return instructions and conditional and unconditional branch instructions perform program control operations. The no operation instruction (NOP) may be used to force synchronization of the internal pipelines. Table 3–8 summarizes these instructions.

Table 3-8. Program Control Operations

Instruction	Operand Syntax	Operand Size	Operation
Integer and Floating-Point Conditional			
Bcc	⟨label⟩	8, 16, 32	if condition true, then PC + d → PC
DBcc	Dn,⟨label⟩	16	if condition false, then Dn — 1 → Dn if Dn ≠ -1, then PC + d → PC
Scc	⟨ea⟩	8	if condition true, then 1's → destination; else 0's → destination
Unconditional			
BRA	⟨label⟩	8, 16, 32	PC + d → PC
BSR	⟨label⟩	8, 16, 32	SP — 4 → SP; PC→(SP); PC + d → PC
JMP	⟨ea⟩	none	destination → PC
JSR	⟨ea⟩	none	SP — 4 → SP; PC→ (SP); destination → PC
NOP	none	none	PC + 2 → PC
Returns			
RTD	#⟨d⟩	16	(SP) → PC; SP + 4 + d → SP
RTR	none	none	(SP) → CCR; SP + 2 → SP; (SP) → PC; SP + 4 → SP
RTS	none	none	(SP) → PC; SP + 4→ SP

Letters cc in the integer instruction mnemonics Bcc, DBcc, and Scc specify testing one of the following conditions:

CC	—	Carry clear	GE	—	Greater or equal
LS	—	Lower or same	PL	—	Plus
CS	—	Carry set	GT	—	Greater than
LT	—	Less than	T	—	Always true*
EQ	—	Equal	HI	—	Higher
MI	—	Minus	VC	—	Overflow clear
F	—	Never true*	LE	—	-Less or equal
NE	—	Not equal	VS	—	Overflow set

*Not applicable to the Bcc instructions.

3.2.9 System Control Instructions

Privileged instructions, trapping instructions, and instructions that use or modify the condition code register (CCR) provide system control operations. Table 3–9 summarizes these instructions. The TRAPcc instruction uses the same conditional tests as the corresponding program control instructions. All of these instructions cause the processor to flush the instruction pipe.

Table 3-9. System Control Operations

Instruction	Operand Syntax	Operand Size	Operation
Privileged			
ANDI	#⟨data⟩,SR	16	immediate data Λ SR → SR
EORI	#⟨data⟩,SR	16	immediate data x SR → SR
MOVE	⟨ea⟩,SR SR,⟨ea⟩	16 16	source → SR SR → destination
MOVE	USP,An An,USP	32 32	USP → An An → USP
MOVEC	Rc,Rn Rn,Rc	32 32	Rc → Rn Rn → Rc
MOVES	Rn, ⟨ea⟩ ⟨ea⟩,Rn	8, 16, 32	Rn → destination using DFC source using SFC → Rn
ORI	#⟨data⟩,SR	16	immediate data V SR → SR
RESET	none	none	assert $\overline{\text{RESET}}$ line
RTE	none	none	(SP) → SR; SP + 2 → SP; (SP) → PC; SP + 4 → SP; Restore stack according to format
STOP	#⟨data⟩	16	immediate data → SR; STOP
Trap Generating			
BKPT	#⟨data⟩	none	run breakpoint cycle, then trap as illegal instruction
CHK	⟨ea⟩,Dn	16, 32	if Dn < 0 or Dn > (ea), then CHK exception
CHK2	⟨ea⟩,Rn	8, 16, 32	if Rn < -lower bound or Rn > -upper bound, then CHK exception
ILLEGAL	none	none	SSP — 2 → SSP; Vector Offset→ (SSP); SSP — 4 → SSP; PC→ (SSP); SSP — 2 → SSP; SR→ (SSP); Illegal Instruction Vector Address → PC
TRAP	#⟨data⟩	none	SSP — 2 → SSP; Format and Vector Offset→(SSP) SSP — 4 → SSP; PC→(SSP); SSP — 2 → SSP; SR→(SSP); Vector Address → PC
TRAPcc	none #⟨data⟩	none 16, 32	if cc true, then TRAP exception
TRAPV	none	none	if V, then take overflow TRAP exception
Condition Code Register			
ANDI	#⟨data⟩,CCR	8	immediate data Λ CCR → CCR
EORI	#⟨data⟩,CCR	8	immediate data ⊕ CCR → CCR
MOVE	⟨ea⟩,CCR CCR,⟨ea⟩	16 16	source → CCR CCR → destination
ORI	#⟨data⟩,CCR	8	immediate data V CCR → CCR

3.2.10 Memory Management Unit Instructions

The PFLUSH instructions flush the address translation caches (ATCs) and can optionally select only nonglobal entries for flushing. PTEST performs a search of the address translation tables, storing results in the MMU status register and loading the entry into the ATC. Table 3–10 summarizes these instructions.

Table 3-10. MMU Instructions

Instruction	Operand Syntax	Operand Size	Operation
PFLUSHA	none	none	Invalidate all ATC entries
PFLUSHA.N	none	none	Invalidate all nonglobal ATC entries
PFLUSH	(An)	none	Invalidate ATC entries at effective address
PFLUSH.N	(An)	none	Invalidate nonglobal ATC entries at effective address
PTEST	(An)	none	Information about logical address → MMU status register

3.2.11 Multiprocessor Instructions

The TAS, CAS, and CAS2 instructions coordinate the operations of processors in multiprocessing systems. These instructions use read-modify-write bus cycles to ensure uninterrupted updating of memory. Coprocessor instructions control the coprocessor operations. Table 3–11 lists these instructions.

Table 3-11. Multiprocessor Operations (Read-Modify-Write)

Instruction	Operand Syntax	Operand Size	Operation
Read-Modify-Write			
CAS	Dc,Du,⟨ea⟩	8, 16, 32	destination — Dc → CC; if Z then Du → destination else destination→Dc
CAS2	Dc1:Dc2,Du1:Du2,(Rn):(Rn)	8, 16, 32	dual operand CAS
TAS	⟨ea⟩	8	destination — 0; set condition codes; 1 → destination [7]
Coprocessor			
cpBcc	⟨label⟩	16, 32	if cpcc true, then PC + d → PC
cpDBcc	label,Dn	16	if cpcc false then Dn –1 → Dn if Dn ≠ –1, then PC + d → PC
cpGEN	User Defined	User Defined	operand → coprocessor
cpRESTORE	⟨ea⟩	none	restore coprocessor state from ⟨ea⟩
cpSAVE	⟨ea⟩	none	save coprocessor state at ⟨ea⟩
cpScc	⟨ea⟩	8	if cpcc true, then 1's → destination; else 0's → destination
cpTRAPcc	none #⟨data⟩	none 16, 32	if cpc true, then TRAPcc exception

3.3 INTEGER CONDITION CODES

The CCR portion of the SR contains five bits which indicate the results of many integer instructions. Program and system control instructions use certain combinations of these bits to control program and system flow.

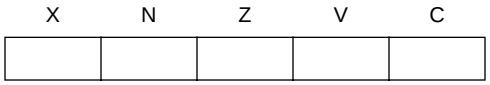
The first four bits represent a condition resulting from a processor operation. The X bit is an operand for multiprecision computations; when it is used, it is set to the value of the C bit. The carry bit and the multiprecision extend bit are separate in the M68000 Family to simplify programming techniques that use them (refer to Table 3–8 as an example).

The condition codes were developed to meet two criteria:

- Consistency across instructions, uses, and instances
- Meaningful Results no change unless it provides useful information

Consistency across instructions means that all instructions that are special cases of more general instructions affect the condition codes in the same way. Consistency across instances means that all instances of an instruction affect the condition codes in the same way. Consistency across uses means that conditional instructions test the condition codes similarly and provide the same results, regardless of whether the condition codes are set by a compare, test, or move instruction.

In the instruction set definitions, the CCR is shown as follows:



where:

X (extend)

Set to the value of the C bit for arithmetic operations. Otherwise not affected or set to a specified result.

N (negative)

Set if the most significant bit of the result is set. Cleared otherwise.

Z (zero)

Set if the result equals zero. Cleared otherwise.

V (overflow)

Set if arithmetic overflow occurs. This implies that the result cannot be represented in the operand size. Cleared otherwise.

C (carry)

Set if a carry out of the most significant bit of the operand occurs for an addition. Also set if a borrow occurs in a subtraction. Cleared otherwise.

3.3.1 Condition Code Computation

Most operations take a source operand and a destination operand, compute, and store the result in the destination location. Single-operand operations take a destination operand, compute, and store the result in the destination location. Table 3–12 lists each instruction and how it affects the condition code bits.

Table 3-12. Condition Code Computations (Sheet 1 of 2)

Operations	X	N	Z	V	C	Special Definition
ABCD	*	U	?	U	?	C =-Decimal Carry Z =-Z Λ $\overline{Rm}$ Λ . . . Λ $\overline{R0}$
ADD, ADDI, ADDQ	*	*	*	?	?	V = Sm Λ $\overline{Dm}$ Λ $\overline{Rm}$ V Sm Λ $\overline{Dm}$ Λ $\overline{Rm}$ C = Sm Λ $\overline{Dm}$ V $\overline{Rm}$ Λ $\overline{Dm}$ V Sm Λ $\overline{Rm}$
ADDX	*	*	?	?	?	V = Sm Λ $\overline{Dm}$ Λ $\overline{Rm}$ V Sm Λ $\overline{Dm}$ Λ $\overline{Rm}$ C = Sm Λ $\overline{Dm}$ V $\overline{Rm}$ Λ $\overline{Dm}$ V Sm Λ $\overline{Rm}$ Z = Z Λ $\overline{Rm}$ Λ . . . Λ $\overline{R0}$
AND, ANDI, EOR, EORI, MOVEQ, MOVE, OR, ORI CLR, EXT, NOT, TAS, TST	—	*	*	0	0	
CHK	—	*	U	U	U	
CHK2, CMP2	—	U	?	U	?	Z = (R = LB) V (R = UB) C = (LB < = UB) Λ (IR < LB) V (R > UB)) V = (UB <LB) Λ (R >UB) Λ (R <LB)
SUB, SUBI, SUBQ	*	*	*	?	?	V = Sm Λ $\overline{Dm}$ Λ $\overline{Rm}$ V Sm Λ $\overline{Dm}$ Λ $\overline{Rm}$ C = Sm Λ $\overline{Dm}$ V $\overline{Rm}$ Λ $\overline{Dm}$ V Sm Λ $\overline{Rm}$
SUBX	*	*	?	?	?	V = Sm Λ $\overline{Dm}$ Λ $\overline{Rm}$ V Sm Λ $\overline{Dm}$ Λ $\overline{Rm}$ C = Sm Λ $\overline{Dm}$ V $\overline{Rm}$ Λ $\overline{Dm}$ V Sm Λ $\overline{Rm}$ Z = Z Λ $\overline{Rm}$ Λ . . . Λ $\overline{R0}$
CAS, CAS2, CMP, CMPI, CMPM	—	*	*	?	?	V = Sm Λ $\overline{Dm}$ Λ $\overline{Rm}$ V Sm Λ $\overline{Dm}$ Λ $\overline{Rm}$ C = Sm Λ $\overline{Dm}$ V $\overline{Rm}$ Λ $\overline{Dm}$ V Sm Λ $\overline{Rm}$
DIVS, DUVI	—	*	*	?	0	V = Division Overflow
MULS, MULU	—	*	*	?	0	V = Multiplication Overflow
SBCD, NBCD	*	U		U	?	C = Decimal Borrow Z = Z Λ $\overline{Rm}$ Λ . . . Λ $\overline{R0}$
NEG	*	*	*	?	?	V = $\overline{Dm}$ Λ $\overline{Rm}$ C = $\overline{Dm}$ V $\overline{Rm}$
NEGX	*	*	?	?	?	V = $\overline{Dm}$ Λ $\overline{Rm}$ C = $\overline{Dm}$ V $\overline{Rm}$ Z = Z Λ $\overline{Rm}$ Λ . . . Λ $\overline{R0}$

Table 3-12. Condition Code Computations (Continued)

Operations	X	N	Z	V	C	Special Definition
BTST, BCHG, BSET, BCLR	—	—	?	—	—	Z = $\overline{Dn}$
BFTST, BFCHG, BFSET, BFCLR	—	?	?	0	0	N = Dm Z = $\overline{Dm} \Lambda \overline{DM-1} \Lambda \ldots \Lambda \overline{D0}$
BFEXTS, BFEXTU, BFFFO	—	?	?	0	0	N = Sm Z = $\overline{Sm} \Lambda \overline{Sm-1} \Lambda \ldots \Lambda \overline{S0}$
BFINS	—	?	?	0	0	N = Dm Z = $\overline{Dm} \Lambda \overline{DM-1} \Lambda \ldots \Lambda \overline{D0}$
ASL	*	*	*			V = $\overline{Dm} \Lambda (\overline{Dm-1} V \ldots V \overline{Dm-r}) V \overline{Dm} \Lambda$ ($\overline{Dm}$ –1 V . . . + Dm –r) C = $\overline{Dm-r+1}$
ASL (R = 0)		*	*	0	0	
LSL, ROXL	*	*	*	0	?	C = Dm –r + 1
LSR (r = 0)	—	*	*	0	0	
ROXL (r = 0)	—	*	*	0	?	C = X
ROL	—	*	*	0	?	C = Dm –r + 1
ROL (r = 0)	—	*	*	0	0	
ASR, LSR, ROXR	*	*	*	0	?	C = Dr –1
ASR, LSR (r = 0)	—	*	*	0	0	
ROXR (r = 0)	—	*	*	0	?	C = X
ROR	—	*	*	0	?	C = Dr –1
ROR (r = 0)	—	*	*	0	0	

— = Not Affected
U = Undefined, Result Meaningless
? = Other — See Special Definition
* = General Case
X = C
N = Rm
Z = $\overline{Rm} \Lambda \ldots \Lambda \overline{R0}$
Sm = Destination Operand — Most Significant Bit
Dm = Destination Operand — Most Significant Bit
Rm = Result Operand — Most Significant Bit
R = Register Tested
n = Bit Number
r = Shift Count
LB = Lower Bound
UB = Upper Bound
Λ = Boolean AND
V = Boolean OR
$\overline{Rm}$ = NOT Rm

3.3.2 Conditional Tests

Table 3–13 lists the condition names, encodings, and tests for the conditional branch and set instructions. The test associated with each condition is a logical formula using the current states of the condition codes. If this formula evaluates to one, the condition is true. If the formula evaluates to zero, the condition is false. For example, the T condition is always true, and the EQ condition is true only if the Z bit condition code is currently true.

Table 3-13. Conditional Tests

Mnemonic	Condition	Encoding	Test
T*	True	0000	1
F*	False	0001	0
HI	High	0010	$\overline{C} \bullet \overline{Z}$
LS	Low or Same	0011	C + Z
CC(HS)	Carry Clear	0100	C
CS(LO)	Carry Set	0101	C
NE	Not Equal	0110	Z
EQ	Equal	0111	Z
VC	Overflow Clear	1000	V
VS	Overflow Set	1001	V
PL	Plus	1010	N
MI	Minus	1011	N
GE	Greater or Equal	1100	$N \bullet V + \overline{N} \bullet \overline{V}$
LT	Less Than	1101	$N \bullet V + N \bullet V$
GT	Greater Than	1110	$N \bullet V \bullet \overline{Z} + \overline{N} \bullet \overline{V} \bullet \overline{Z}$
LE	Less or Equal	1111	$Z + N \bullet \overline{V} + \overline{N} \bullet V$

• = Boolean AND
+ = Boolean OR
$\overline{N}$ = Boolean NOT N
*Not available for the Bcc instruction.

3.4 INSTRUCTION SET SUMMARY

Table 3–14 provides a alphabetized listing of the MC68030 instruction set listed by opcode, operation, and syntax.

Table 3–14 use notational conventions for the operands, the subfields and qualifiers, and the operations performed by the instructions. In the syntax descriptions, the left operand is the source operand, and the right operand is the destination operand. The following list contains the notations used in Table 3–14.

Notation for operands:

PC	—	Program counter
SR	—	Status register
V	—	Overflow condition code
Immediate Data	—	Immediate data from the instruction
Source	—	Source contents
Destination	—	Destination contents
Vector	—	Location of exception vector
+ inf	—	Positive infinity
–inf	—	Negative infinity
⟨fmt⟩	—	Operand data format: byte (B) word (W), long (L), single (S), double (D), extended (X), or packed (P)
FPm	—	One of eight floating-point data registers (always specifies the source register)
FPn	—	One of eight floating-point data registers (always specifies the destination register)

Notation for subfields and qualifiers:

⟨bit⟩ of (operand⟩	—	Selects a single bit of the operand
⟨ea⟩ {offset:width}	—	Selects a bit field
(⟨operand⟩)	—	The contents of the referenced location
⟨operand⟩ $_{10}$	—	The operand is binary-coded decimal; operations are performed in decimal
(⟨address register⟩)	—	The register indirect operation
–(⟨address register⟩)	—	Indicates that the operand register points to the memory
(⟨address register⟩) +	—	Location of the instruction operand — the optional mode qualifiers are -, +, (d), and (d,ix)
#xxx or #⟨data⟩	—	Immediate data that follows the instruction word(s)

Notations for operations that have two operands, written ⟨operand⟩ ⟨op⟩ ⟨operand⟩, where ⟨op⟩ is one of the following:

→	—	The source operand is moved to the destination operand
↔	—	The two operands are exchanged
+	—	The operands are added
–	—	The destination operand is subtracted from the source operand
x	—	The operands are multiplied
÷	—	The source operand is divided by the destination operand
<	—	Relational test, true if source operand is less than destination operand
>	—	Relational test, true if source operand is greater than destination operand
V	—	Logical OR
⊕	—	Logical exclusive OR
Λ	—	Logical AND
shifted by, rotated by	—	The source operand is shifted or rotated by the number of positions specified by the second operand

Notation for single-operand operations:

~⟨operand⟩	—	The operand is logically complemented
⟨operand⟩ sign-extended	—	The operand is sign extended; all bits of the upper portion are made equal to the high-order bit of the lower portion
⟨operand⟩ tested	—	The operand is compared to zero and the condition codes are set appropriately

Notation for other operations:

TRAP	—	Equivalent to Format/Offset Word→(SSP); SSP –2 → SSP; PC→(SSP); SSP – 4 → SSP; SR→(SSP); SSP–2 → SSP; (vector) → PC
STOP	—	Enter the stopped state, waiting for the interrupts
If ⟨condition⟩ then	—	The condition is tested. If true, the operations
⟨operations⟩ else	—	after "then" are performed. If the condition is
⟨operations⟩	—	false and the optional "else" clause is present, the operations after "else" are performed. If the condition is false and else is omitted, the instruction performs no operation. Refer to the Bcc instruction description as an example.

Table 3-14. Instruction Set Summary (Sheet 1 of 5)

Opcode	Operation	Syntax
ABCD	$Source_{10}$ + $Destination_{10}$ + X → Destination	ABCD Dy,Dx ABCD –(Ay),–(Ax)
ADD	Source + Destination → -Destination	ADD ⟨ea⟩,Dn ADD Dn,⟨ea⟩
ADDA	Source + Destination → Destination	ADDA ⟨ea⟩,An
ADDI	Immediate Data + Destination → Destination	ADDI #⟨data⟩,⟨ea⟩
ADDQ	Immediate Data + Destination → Destination	ADDQ #⟨data⟩,⟨ea⟩
ADDX	Source + Destination + X → Destination	ADDX Dy,Dx ADDX –(Ay),–(Ax)
AND	Source Λ Destination → Destination	AND ⟨ea⟩,Dn AND Dn,⟨ea⟩
ANDI	Immediate Data Λ Destination → Destination	ANDI #⟨data⟩,⟨ea⟩
ANDI to CCR	Source Λ CCR → CCR	ANDI #⟨data⟩,CCR
ANDI to SR	If supervisor state then Source Λ SR → SR else TRAP	ANDI #⟨data⟩,SR
ASL,ASR	Destination Shifted by ⟨count⟩ → Destination	ASd Dx,Dy ASd #⟨data⟩,Dy ASd ⟨ea⟩
Bcc	If (condition true) then PC + d → PC	Bcc (label⟩
BCHG	~ (⟨number⟩ of Destination) → Z; ~ (⟨number⟩ of Destination) → ⟨bit number⟩ of Destination	BCHG Dn,⟨ea⟩BCHG #⟨data⟩,⟨ea⟩
BCLR	~ (⟨bit number⟩ of Destination) → Z; 0 → ⟨bit number⟩ of Destination	BCLR Dn,⟨ea⟩BCLR #⟨data⟩,⟨ea⟩
BFCHG	~ (⟨bit field⟩ of Destination) → ⟨bit field⟩ of Destination	BFCHG ⟨ea⟩{offset:width}
BFCLR	0 → ⟨bit field⟩ of Destination	BFCLR ⟨ea⟩{offset:width}
BFEXTS	⟨bit field⟩ of Source → Dn	BFEXTS ⟨ea⟩{offset:width},Dn
BFEXTU	(bit offset⟩ of Source → Dn	BFEXTU ⟨ea⟩{offset:width},Dn
BFFFO	(bit offset⟩ of Source Bit Scan → Dn	BFFFO ⟨ea⟩{offset:width},Dn
BFINS	Dn → ⟨bit field⟩ of Destination	BFINS Dn,⟨ea⟩{offset:width}
BFSET	1s → ⟨bit field⟩ of Destination	BFSET ⟨ea⟩{offset:width}
BFTST	⟨bit field⟩ of Destination	BFTST ⟨ea⟩{offset:width}
BKPT	Run breakpoint acknowledge cycle; TRAP as illegal instruction	BKPT # ⟨data⟩
BRA	PC + d → PC	BRA (label⟩
BSET	~ (⟨bit number⟩ of Destination) → Z; 1 → ⟨bit number⟩ of Destination	BSET Dn,⟨ea⟩BSET #⟨data⟩,⟨ea⟩
BSR	SP – 4 → SP; PC → (SP); PC + d → PC	BSR (label⟩
BTST	–(⟨bit number⟩ of Destination) → Z;	BTST Dn,⟨ea⟩BTST #⟨data⟩,⟨ea⟩

Table 3-14. Instruction Set Summary (Sheet 2 of 5)

Opcode	Operation	Syntax
CAS CAS2	CAS Destination Compare Operand → cc; if Z, Update Operand → Destination else Destination → Compare Operand CAS2 Destination 1 Compare 1 → cc; if Z, Destination 2 Compare → cc; if Z, Update 1 → Destination 1; Update 2 → Destination 2 else Destination 1 → Compare 1; Destination 2 →Compare 2	CAS Dc,Du,⟨ea⟩CAS2 Dc1:Dc2,Du1:Du2,(Rn1):(Rn2)
CHK	If Dn < 0 or >-Source then TRAP	CHK ⟨ea⟩,Dn
CHK2	If Rn < lower bound or Rn > upper bound then TRAP	CHK2 ⟨ea⟩,Rn
CLR	0 → Destination	CLR ⟨ea⟩
CMP	Destination — Source → cc	CMP ⟨ea⟩,Dn
CMPA	Destination — Source	CMPA ⟨ea⟩,An
CMPI	Destination — Immediate Data	CMPI #⟨data⟩,⟨ea⟩
CMPM	Destination — Source → cc	CMPM (Ay) +,(Ax) +
CMP2	Compare Rn < lower-bound or Rn > upper-bound and Set Condition Codes	CMP2 ⟨ea⟩,Rn
cpBcc	If cpcc true then scanPC + d → PC	cpBcc (label⟩
cpDBcc	If cpcc false then (Dn –1 → Dn; if Dn ≠ –1 then scanPC + d → PC	cpDBcc Dn,(label⟩
cpGEN	Pass Command Word to Coprocessor	cpGEN (parameters as defined by coprocessorL
cpRESTORE	If supervisor state then Restore Internal State of Coprocessor else TRAP	cpRESTORE ⟨ea⟩
cpSAVE	If supervisor state the Save Internal State of Coprocessor else TRAP	cpSAVE ⟨save⟩
cpScc	If cpcc true then 1s → Destination else 0s → Destination	
cpTRAPcc	If cpcc true then TRAP	cpTRAPcc cpTRAPcc #⟨data⟩
DBcc	If condition false then (Dn–1 → Dn; If Dn ≠ –1 then PC + d → PC)	DBcc Dn,(label⟩
DIVS DIVSL	Destination/Source → Destination	DIVS.W ⟨ea⟩,Dn32/16 → 16r:16q DIVS.L ⟨ea⟩,Dq 32/32 → 32q DIVS.L ⟨ea⟩,Dr:Dq 64/32 → 32r:32q DIVSL.L ⟨ea⟩,Dr:Dq32/32 → 32r:32q
DIVU DIVUL	Destination/Source → Destination	DIVU.W ⟨ea⟩,Dn32/16 → 16r:16q DIVU.L ⟨ea⟩,Dq 32/32 → 32q DIVU.L ⟨ea⟩,Dr:Dq 64/32 → 32r:32q DIVUL.L ⟨ea⟩,Dr:Dq32/32 → 32r:32q
EOR	Source ⊕ Destination → Destination	EOR Dn,⟨ea⟩
EORI	Immediate Data ⊕ Destination → Destination	EORI #⟨data⟩,⟨ea⟩

Table 3-14. Instruction Set Summary (Sheet 3 of 5)

Opcode	Operation	Syntax
EORI to CCR	Source ⊕ CCR → CCR	EORI #⟨data⟩,CCR
EORI to SR	If supervisor state then Source ⊕ SR → SR else TRAP	EORI #⟨data⟩,SR
EXG	Rx ↔ Ry	EXG Dx,Dy EXG Ax,Ay EXG Dx,Ay EXG Ay,Dx
EXT EXTB	Destination Sign-Extended → Destination	EXT.W Dn extend byte to word EXT.L L Dn extend word to long word EXTB.L Dn extend byte to long word
ILLEGAL	SSP–2 → SSP; Vector Offset → (SSP); SSP–4 → SSP; PC → (SSP); SSP–2 → SSP; SR → (SSP); Illegal Instruction Vector Address → PC	ILLEGAL
JMP	Destination Address → PC	JMP ⟨ea⟩
JSR	SP–4 → SP; PC → (SP) Destination Address → PC	JSR ⟨ea⟩
LEA	⟨ea⟩ → An	LEA ⟨ea⟩,An
LINK	SP — 4 → SP; An → (SP) SP → An, SP + d → SP	LINK An, #(displacement⟩
LSL,LSR	Destination Shifted by ⟨count⟩ → Destination	LSd[5] Dx,Dy LSd[5] #⟨data⟩,Dy LSd[5] ⟨ea⟩
MOVE	Source → Destination	MOVE ⟨ea⟩,⟨ea⟩
MOVEA	Source → Destination	MOVEA ⟨ea⟩,An
MOVE from CCR	CCR → Destination	MOVE CCR,⟨ea⟩
MOVE to CCR	Source → CCR	MOVE ⟨ea⟩,CCR
MOVE from SR	If supervisor state then SR → Destination else TRAP	MOVE SR,⟨ea⟩
MOVE to SR	If supervisor state then Source → SR else TRAP	MOVE ⟨ea⟩,SR
MOVE USP	If supervisor state then USP → An or An → USP else TRAP	MOVE USP,An MOVE An,USP
MOVEC	If supervisor state then Rc → Rn or Rn → Rc else TRAP	MOVEC Rc,Rn MOVEC Rn,Rc
MOVEM	Registers → Destination Source → Registers	MOVEM register list,⟨ea⟩MOVEM ⟨ea⟩,register list
MOVEP	Source → Destination	MOVEP Dx,(d,Ay) MOVEP (d,Ay),Dx
MOVEQ	Immediate Data → Destination	MOVEQ #⟨data⟩,Dn

Table 3-14. Instruction Set Summary (Sheet 4 of 5)

Opcode	Operation	Syntax
MOVES	If supervisor state then Rn → Destination [DFC] or Source [SFC] → Rn else TRAP	MOVES Rn,⟨ea⟩MOVES ⟨ea⟩,Rn
MULS	Source y-Destination → Destination	MULS.W ⟨ea⟩,Dn 16 x 16 → 32 MULS.L ⟨ea⟩,Dl 32 x 32 → 32 MULS.L ⟨ea⟩,Dh:Dl 32 x 32 → 64
MULU	Source y-Destination → Destination	MULU.W ⟨ea⟩,Dn 16 x 16 → 32 MULU.L ⟨ea⟩,Dl 32 x 32 → 32 MULU.L ⟨ea⟩,Dh:Dl 32 x 32 → 64
NBCD	0 — (Destination10) — X → Destination	NBCD ⟨ea⟩
NEG	0 — (Destination) → Destination	NEG ⟨ea⟩
NEGX	0 — (Destination) — X → Destination	NEGX ⟨ea⟩
NOP	None	NOP
NOT	~ Destination → Destination	NOT ⟨ea⟩
OR	Source V Destination → Destination	OR ⟨ea⟩,Dn OR Dn,⟨ea⟩
ORI	Immediate Data V Destination → Destination	ORI #⟨data⟩,⟨ea⟩
ORI to CCR	Source V CCR → CCR	ORI #⟨data⟩,CCR
ORI to SR	If supervisor state then Source V SR → SR else TRAP	ORI #⟨data⟩,SR
PACK	Source (Unpacked BCD) + adjustment → Destintion (Packed BCD)	PACK –(Ax),–(Ay),#⟨adjustment⟩ PACK Dx,Dy,#⟨adjustment⟩
PEA	Sp –4 → SP; ⟨ea⟩ → (SP)	PEA ⟨ea⟩
PFLUSH	If supervisor state then invalidate instruction and data ATC entries for destination address else TRAP	
PLOAD	If supervisor state then entry → ATC else TRAP	
PMOVE	If supervisor state then (Source) → MRn or MRn → (Destination)	
PTEST	If supervisor state then logical address status → MMUSR; entry → ATC else TRAP	
RESET	If supervisor state then Assert $\overline{\text{RSTO}}$ Line else TRAP	RESET
ROL,ROR	Destination Rotated by ⟨count⟩ → Destination	ROd[5] Rx,Dy ROd[5] #⟨data⟩,Dy ROd[5] ⟨ea⟩
ROXL, ROXR	Destination Rotated with X by ⟨count⟩ → Destination	ROXd[5] Dx,Dy ROXd[5] #⟨data⟩,Dy ROXd[5] ⟨ea⟩

Table 3-14. Instruction Set Summary (Concluded)

Opcode	Operation	Syntax
RTD	(SP) → PC; SP + 4 + d → SP	RTD #⟨displacement⟩
RTE	If-supervisor-state then (SP) → SR; SP+2 → SP; (SP) → PC; SP + 4 → SP; restore state and deallocate stack according to (SP) else TRAP	RTE
RTM	Reload Saved Module State from Stack	RTM Rn
RTR	(SP) → CCR; SP + 2 → SP; (SP) → PC; SP + 4 → SP	RTR
RTS	(SP) → PC; SP + 4 → SP	RTS
SBCD	$Destination_{10}$ --$Source_{10}$ –X → Destination	SBCD Dx,Dy SBCD –(Ax),–(Ay)
Scc	If condition true then 1s → Destination else 0s → Destination	Scc ⟨ea⟩
STOP	If supervisor state then Immediate Data → SR; STOP else TRAP	STOP #⟨data⟩
SUB	Destination — Source → Destination	SUB ⟨ea⟩,Dn SUB Dn,⟨ea⟩
SUBA	Destination — Source → Destination	SUBA ⟨ea⟩,An
SUBI	Destination — Immediate Data → Destination	SUBI #⟨data⟩,⟨ea⟩
SUBQ	Destination — Immediate Data → Destination	SUBQ #⟨data⟩,⟨ea⟩
SUBX	Destination — Source – X → Destination	SUBX Dx,Dy SUBX –(Ax),–(Ay)
SWAP	Register [31:16] ↔ Register [15:0]	SWAP Dn
TAS	Destination Tested → Condition Codes; 1 → bit 7 of Destination	TAS ⟨ea⟩
TRAP	SSP –2 → SSP; Format/Offset → (SSP); SSP – 4 → SSP; PC → (SSP); SSP – 2 → SSP; SR → (SSP); Vector Address → PC	TRAP # (vector⟩
TRAPcc	If cc then TRAP	TRAPcc TRAPcc.W # ⟨data⟩TRAPcc.L # ⟨data⟩
TRAPV	If V then TRAP	TRAPV
TST	Destination Tested → Condition Codes	TST ⟨ea⟩
UNLK	An → SP; (SP) → An; SP + 4 → SP	UNLK An
UNPK	Source (Packed BCD) + adjustment → Destination (Unpacked BCD)	UNPACK –(Ax),–(Ay),#(adjustment⟩ UNPACK Dx,Dy,#(adjustment⟩

NOTES:
1. Specifies either the instruction (IC), data (DC), or IC/DC caches.
2. Where r is rounding precision, S or D.
3. A list of any combination of the eight floating-point data registers, with individual register names separated by a slash (/) and/or contiguous blocks of registers specified by the first and last register names separated by a dash (–).
4. A list of any combination of the three floating-point system control registers (FPCR, FPSR, and FPIAR) with indvidual register names separated by a slash (/).
5. Where d is direction, L or R.

3.5 INSTRUCTION EXAMPLES

The following paragraphs provide examples of how to use selected instructions.

3.5.1 Using the CAS and CAS2 Instructions

The CAS instruction compares the value in a memory location with the value in a data register, and copies a second data register into the memory location if the compared values are equal. This provides a means of updating system counters, history information, and globally shared pointers. The instruction uses an indivisible read-modify-write cycle; after CAS reads the memory location, no other instruction can change that location before CAS has written the new value. This provides security in single-processor systems, in multitasking environments, and in multiprocessor environments. In a single-processor system, the operation is protected from instructions of an interrupt routine. In a multitasking environment, no other task can interfere with writing the new value of a system variable. In a multiprocessor environment, the other processors must wait until the CAS instruction completes before accessing a global pointer.

The following code fragment shows a routine to maintain a count, in location SYS_CNTR, of the executions of an operation that may be performed by any process or processor in a system. The routine obtains the current value of the count in register D0 and stores the new count value in register D1. The CAS instruction copies the new count into SYS_CNTR if it is valid. However, if another user has incremented the counter between the time the count was stored and the read-modify-write cycle of the CAS instruction, the write portion of the cycle copies the new count in SYS_CNTR into D0, and the routine branches to repeat the test. The following code sequence guarantees that SYS_CNTR is correctly incremented.

```
          MOVE.W    SYS_CNTR,D0      get the old value of the counter
INC_LOOP  MOVE.W    D0,D1            make a copy of it
          ADDQ.W    #1,D1            and increment it
          CAS.W     D0,D1,SYS_CNTR   if countr value is still the same, update it
          BNE       INC_LOOP         if not, try again
```

The CAS and CAS2 instructions together allow safe operations in the manipulation of system linked lists. Controlling a single location, HEAD in the example, manages a last-in-first-out linked list (see Figure 3–2). If the list is empty, HEAD contains the NULL pointer (0); otherwise, HEAD contains the address of the element most recently added to the list. The code fragment shown in Figure 3–2 illustrates the code for inserting an element. The MOVE instructions load the address in location HEAD into D0 and into the NEXT pointer in the element being inserted, and the address of the new element into D1. The CAS instruction stores the address of the inserted element into location HEAD if the address in HEAD remains unaltered. If HEAD contains a new address, the instruction loads the new address into D0 and branches to the second MOVE instruction to try again.

The CAS2 instruction is similar to the CAS instruction except that it performs two comparisons and updates two variables when the results of the comparisons are equal. If the results of both comparisons are equal, CAS2 copies new values into the destination addresses. If the result of either comparison is not equal, the instruction copies the values in the destination addresses into the compare operands.

```
SINSERT                                  ALLOCATE NEW ENTRY, ADDRESS IN A1
        MOVE.L    HEAD.D0                MOVE HEAD POINTER VALUE TO D0
SILOOP  MOVE.L    D0, (NEXT, A1)         ESTABLISH FORWARD LINK IN NEW ENTRY
        MOVE.L    A1, D1                 MOVE NEW ENTRY POINTER VALUE TO D1
        CAS.L     D0, D1, HEAD           IF WE STILL POINT TO TOP OF STACK, UPDATE THE HEAD POINTER
        BNE       SILOOP                 IF NOT, TRY AGAIN
```

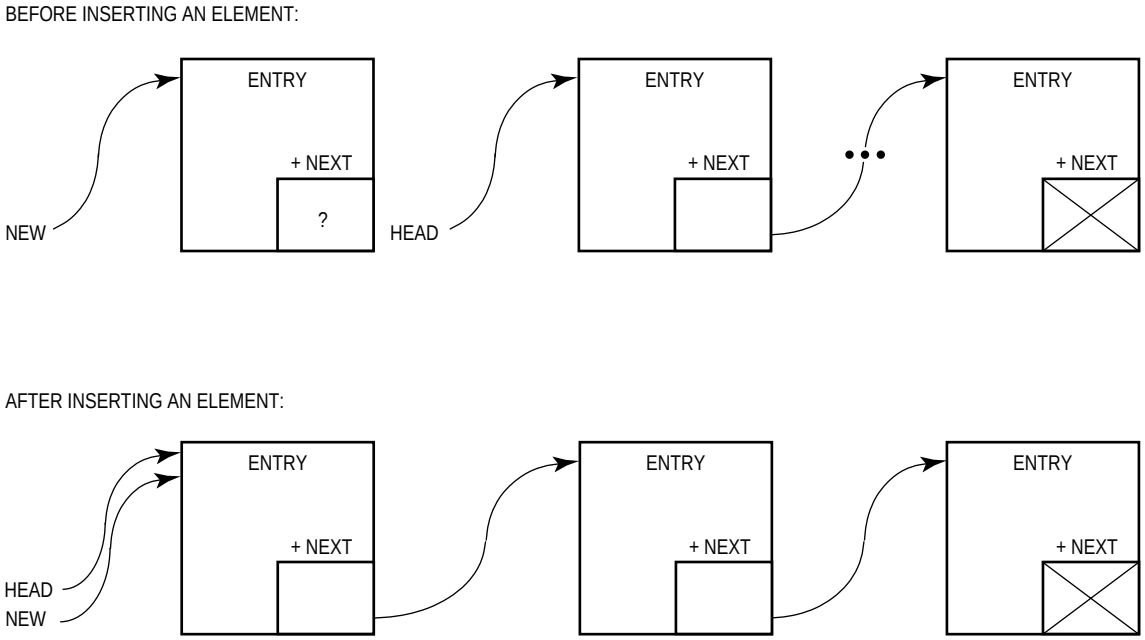


Figure 3-2. Linked List Insertion

The next code (see Figure 3–3) fragment shows the use of a CAS2 instruction to delete an element from a linked list. The first LEA instruction loads the effective address of HEAD into A0. The MOVE instruction loads the address in pointer HEAD into D0. The TST instruction checks for an empty list, and the BEQ instruction branches to a routine at label SDEMPTY if the list is empty. Otherwise, a second LEA instruction loads the address of the NEXT pointer in the newest element on the list into A1, and the following MOVE instruction loads the pointer contents into D1. The CAS2 instruction compares the address of the newest structure to the value in HEAD and the address in D1 to the pointer in the address in A1. If no element has been inserted or deleted by another routine while this routine has been executing, the results of these comparisons are equal, and the CAS2 instruction stores the new value into location HEAD. If an element has been inserted or deleted, the CAS2 instruction loads the new address in location HEAD into D0, and the BNE instruction branches to the TST instruction to try again.

```
SDELETE
                LEA     HEAD, A0            LOAD ADDRESS OF HEAD POINTER INTO A0
                MOVE.L  (A0), D0            MOVE VALUE OF HEAD POINTER INTO D0
SDLOOP          TST.L   D0                  CHECK FOR NULL HEAD POINTER
                BEQ     SDEMPTY             IF EMPTY, NOTHING TO DELETE
                LEA     (NEXT, D0), A1      LOAD ADDRESS OF FORWARD LINK INTO A1
                MOVE.L  (A1), D1            PUT FORWARD LINK VALUE IN D1
                CAS2.L  D0:D1, D1:D1, (A0):(A1)
                                            IF STILL POINT TO ENTRY TO BE DELETED, THEN UPDATE HEAD
                                            AND FORWARD POINTERS
                BNE     SDLOOP              IF NOT, TRY AGAIN
SDEMPTY                                     SUCCESSFUL DELETION, ADDRESS OF DELETED ENTRY IN D0
                                            (MAY BE NULL)
```

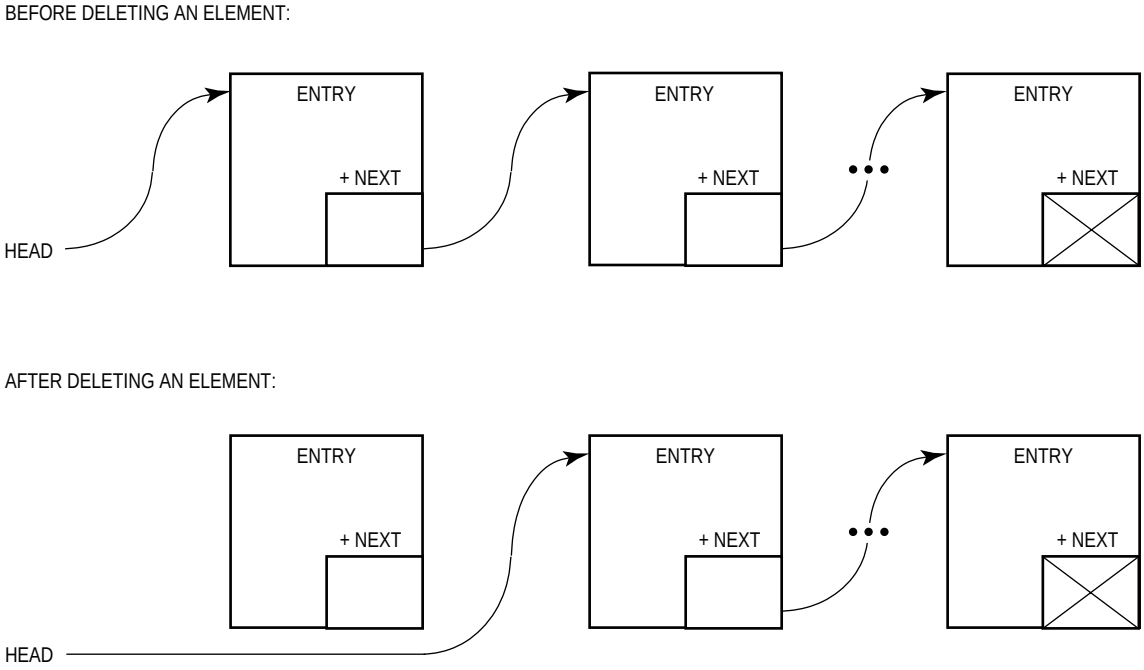


Figure 3-3. Linked List Deletion

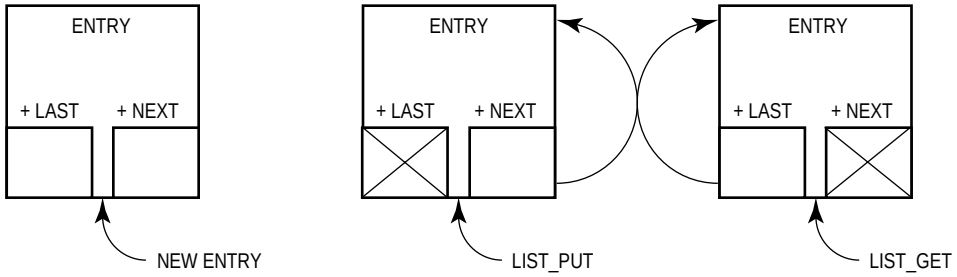
The CAS2 instruction can also be used to correctly maintain a first-in-first-out doubly linked list. A doubly linked list needs two controlled locations, LIST_PUT and LIST_GET, which contain pointers to the last element inserted in the list and the next to be removed, respectively. If the list is empty, both pointers are NULL (0).

The code fragment shown in Figure 3–4 illustrates the insertion of an element in a doubly linked list. The first two instructions load the effective addresses of LIST_PUT and LIST_GET into registers A0 and A1, respectively. The next instruction moves the address of the new element into register D2. Another MOVE instruction moves the address in LIST_PUT into register D0. At label DILOOP, a TST instruction tests the value in D0, and the BEQ instruction branches to the MOVE instruction when D0 is equal to zero. Assuming the list is empty, this MOVE instruction is executed next; it moves the zero in D0 into the NEXT and LAST pointers of the new element. Then the CAS2 instruction moves the address of the new element into both LIST_PUT and LIST_GET, assuming that both of these pointers still contain zero. If not, the BNE instruction branches to the TST instruction at label DILOOP to try again. This time, the BEQ instruction does not branch, and the following MOVE instruction moves the address in D0 to the NEXT pointer of the new element. The CLR instruction clears register D1 to zero, and the MOVE instruction moves the zero into the LAST pointer of the new element. The LEA instruction loads the address of the LAST pointer of the most recently inserted element into register A1. Assuming the LIST_PUT pointer and the pointer in A1 have not been changed, the CAS2 instruction stores the address of the new element into these pointers.

The code fragment to delete an element from a doubly linked list is similar (see Figure 3–5). The first two instructions load the effective addresses of pointers LIST_PUT and LIST_GET into registers A0 and A1, respectively. The MOVE instruction at label DDLOOP moves the LIST_GET pointer into register D1. The BEQ instruction that follows branches out of the routine when the pointer is zero. The MOVE instruction moves the LAST pointer of the element to be deleted into register D2. Assuming this is not the last element in the list, the Z condition code is not set, and the branch to label DDEMPTY does not occur. The LEA instruction loads the address of the NEXT pointer of the element at the address in D2 into register A2. The next instruction, a CLR instruction, clears register D0 to zero. The CAS2 instruction compares the address in D1 to the LIST-GET pointer and to the address in register A2. If the pointers have not been updated, the CAS2 instruction loads the address in D2 into the LIST_GET pointer and zero into the address in register A2.

```
DINSERT                                     (ALLOCATE NEW LIST ENTRY, LOAD ADDRESS INTO A2)
            LEA     LIST_PUT, A0            LOAD ADDRESS OF HEAD POINTER INTO A0
            LEA     LIST_GET, A1            LOAD ADDRESS OF TAIL POINTER INTO A1
            MOVE.L  A2, D2                  LOAD NEW ENTRY POINTER INTO D2
            MOVE.L  (A0), D0                LOAD POINTER TO HEAD ENTRY INTO D0
            TST.L   D0                      IS HEAD POINTER NULL, (0 ENTRIES IN LIST)?
DILOOP      BEQ     DIEMPTY                 IF SO, WE NEED ONLY TO ESTABLISH POINTERS
            MOVE.L  D0, (NEXT, A2)          PUT HEAD POINTER INTO FORWARD POINTER OF NEW ENTRY
            CLR.L   D1                      PUT NULL POINTER VALUE INTO D1
            MOVE.L  D1, (LAST, A2)          PUT NULL POINTER IN BACKWARD POINTER OF NEW ENTRY
            LEA     (LAST, D0), A1          LOAD BACKWARD POINTER OF OLD HEAD ENTRY INTO A1
            CAS2.L  D0:D1,D2:D2,(A0):(A1)   IF WE STILL POINT TO OLD HEAD ENTRY, UPDATE POINTERS
            BNE     DILOOP                  IF NOT, TRY AGAIN
            BRA     DIDONE
DIEMPTY     MOVE.L  D0, (NEXT, A2)          PUT NULL POINTER IN FORWARD POINTER OF NEW ENTRY
            MOVE.L  D0, (LAST, A2)          PUT NULL POINTER IN BACKWARD POINTER OF NEW ENTRY
            CAS2.L  D0:D0,D2:D2,(A0):(A1)   IF WE STILL HAVE NO ENTRIES, SET BOTH POINTERS TO THIS ENTRY
            BNE     DILOOP                  IF NOT, TRY AGAIN
DIDONE                                      SUCCESSFUL LIST ENTRY INSERTION
```

BEFORE INSERTING NEW ENTRY:



AFTER INSERTING NEW ENTRY:

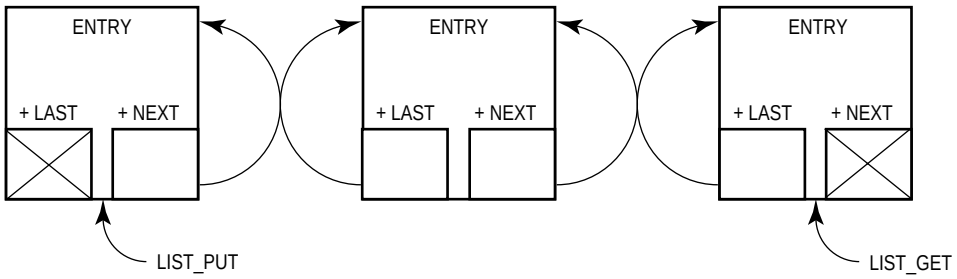
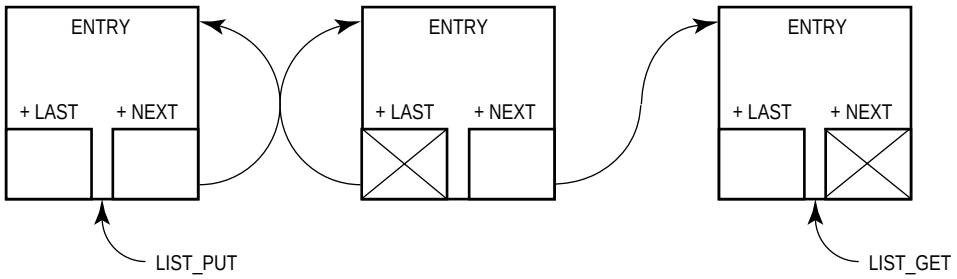


Figure 3-4. Doubly Linked List Insertion

When the list contains only one element, the routine branches to the CAS2 instruction at label DDEMPTY after moving a zero pointer value into D2. This instruction checks the addresses in LIST_PUT and LIST_GET to verify that no other routine has inserted another element or deleted the last element. Then the instruction moves zero into both pointers, and the list is empty.

```
DDELETE
                LEA     LIST_PUT, A0        GET ADDRESS OF HEAD POINTER IN A0
                LEA     LIST_GET, A1        GET ADDRESS OF TAIL POINTER IN A1
DDLOOP          MOVE.L  (A1),D1             MOVE TAIL POINTER INTO D1
                BEQ     DDDONE              IF NO LIST, QUIT
                MOVE.L  (LAST,D1),D2        PUT BACKWARD POINTER IN D2
                BEQ     DDEMPTY             IF ONLY ONE ELEMENT, UPDATE POINTERS
                LEA     (NEXT,D2),A2        PUT ADDRESS OF FORWARD POINTER IN A2
                CLR.L   D0                  PUT NULL POINTER VALUE IN D0
                CAS2.L  D1:D1,D2:D0,(A1):(A2)  IF BOTH POINTERS STILL POINT TO THIS ENTRY , UPDATE THEM
                BNE     DDLOOP              IF NOT, TRY AGAIN
                BRA     DDDONE
DDEMPTY         CAS2.L  D1:D1,D2:D2,(A1):(A0)  IF STILL FIRST ENTRY, SET HEAD AND TAIL POINTERS TO NULL
                BNE     DDLOOP              IF NOT, TRY AGAIN
DDDONE                                      SUCCESSFUL ENTRY DELETION, ADDRESS OF DELETED ENTRY IN D1
                                            (MAY BE NULL)
```

BEFORE DELETING ENTRY:



AFTER DELETING ENTRY:

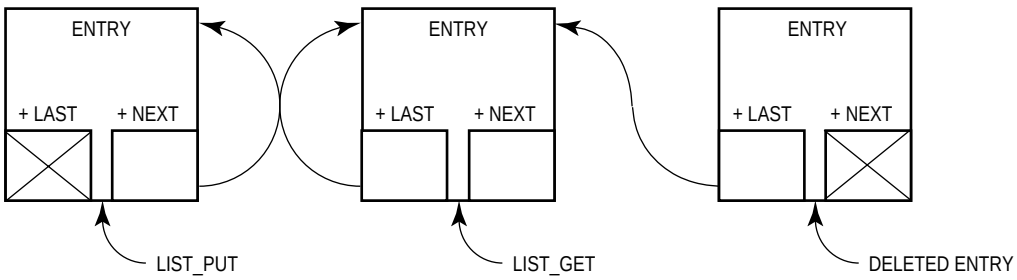


Figure 3-5. Doubly Linked List Deletion

3.5.2 Nested Subroutine Calls

The LINK instruction pushes an address onto the stack, saves the stack address at which the address is stored, and reserves an area of the stack. Using this instruction in a series of subroutine calls results in a linked list of stack frames.

The UNLK instruction removes a stack frame from the end of the list by loading an address into the stack pointer and pulling the value at that address from the stack. When the operand of the instruction is the address of the link address at the bottom of a stack frame, the effect is to remove the stack frame from the stack and from the linked list.

3.5.3 Bit Field Operations

One data type provided by the MC68030 is the bit field, consisting of as many as 32 consecutive bits. A bit field is defined by an offset from an effective address and a width value. The offset is a value in the range of 2^{31} through 2^{31} 1 from the most significant bit (bit 7) at the effective address. The width is a positive number, 1–32. The most significant bit of a bit field is bit 0; the bits number in a direction opposite to the bits of an integer.

The instruction set includes eight instructions that have bit field operands. The insert bit field (BFINS) instruction inserts a bit field stored in a register into a bit field. The extract bit field signed (BFEXTS) instruction loads a bit field into the least significant bits of a register and extends the sign to the left, filling the register. The extract bit field unsigned (BFEXTU) also loads a bit field, but zero fills the unused portion of the destination register.

The set bit field (BFSET) instruction sets all the bits of a field to ones. The clear bit field (BFCLR) instruction clears a field. The change bit field (BFCHG) instruction complements all the bits in a bit field. These three instructions all test the previous value of the bit field, setting the condition codes accordingly. The test bit field (BFTST) instruction tests the value in the field, setting the condition codes appropriately without altering the bit field. The find first one in bit field (BFFFO) instruction scans a bit field from bit 0 to the right until it finds a bit set to one and loads the bit offset of the first set bit into the specified data register. If no bits in the field are set, the field offset and the field width is loaded into the register.

An important application of bit field instructions is the manipulation of the exponent field in a floating-point number. In the IEEE standard format, the most significant bit is the sign bit of the mantissa. The exponent value begins at the next most significant bit position; the exponent field does not begin on a byte boundary. The extract bit field (BFEXTU) instruction and the BFTST instruction are the most useful for this application, but other bit field instructions can also be used.

Programming of input and output operations to peripherals requires testing, setting, and inserting of bit fields in the control registers of the peripherals, which is another application for bit field instructions. However, control register locations are not memory locations; therefore, it is not always possible to insert or extract bit fields of a register without affecting other fields within the register.

Another widely used application for bit field instructions is bit-mapped graphics. Because byte boundaries are ignored in these areas of memory, the field definitions used with bit field instructions are very helpful.

3.5.4 Pipeline Synchronization with the Nop Instruction

Although the no operation (NOP) instruction performs no visible operation, it serves an important purpose. It forces synchronization of the integer unit pipeline by waiting for all pending bus cycles to complete. All previous integer instructions and floating-point external operand accesses complete execution before the NOP begins. The NOP instruction does not synchronize the FPU pipeline; floating-point instructions with floating-point register operand destinations can be executing when the NOP begins.

SECTION 4
PROCESSING STATES

This section describes the processing states of the MC68030. It describes the functions of the bits in the supervisor portion of the status register and the actions taken by the processor in response to exception conditions.

Unless the processor has halted, it is always in either the normal or the exception processing state. Whenever the processor is executing instructions or fetching instructions or operands, it is in the normal processing state. The processor is also in the normal processing state while it is storing instruction results or communicating with a coprocessor.

NOTE

Exception processing refers specifically to the transition from normal processing of a program to normal processing of system routines, interrupt routines, and other exception handlers. Exception processing includes all stacking operations, the fetch of the exception vector, and filling of the instruction pipe caused by an exception. It has completed when execution of the first instruction of the exception handler routine begins.

The processor enters the exception processing state when an interrupt is acknowledged, when an instruction is traced or results in a trap, or when some other exceptional condition arises. Execution of certain instructions or unusual conditions occurring during the execution of any instructions can cause exceptions. External conditions, such as interrupts, bus errors, and some coprocessor responses, also cause exceptions. Exception processing provides an efficient transfer of control to handlers and routines that process the exceptions.

A catastrophic system failure occurs whenever the processor receives a bus error or generates an address error while in the exception processing state. This type of failure halts the processor. For example, if during the exception processing of one bus error another bus error occurs, the MC68030 has not completed the transition to normal processing and has not completed saving the internal state of the machine, so the processor assumes that the system is not operational and halts. Only an external reset can restart a halted processor. (When the processor executes a STOP instruction, it is in a special type of normal processing state, one without bus cycles. It is stopped, not halted.)

4.1 PRIVILEGE LEVELS

The processor operates at one of two levels of privilege: the user level or the supervisor level. The supervisor level has higher privileges than the user level. Not all processor or coprocessor instructions are permitted to execute in the lower privileged user level, but all are available at the supervisor level. This allows a separation of supervisor and user so the supervisor can protect system resources from uncontrolled access. The processor uses the privilege level indicated by the S bit in the status register to select either the user or supervisor privilege level and either the user stack pointer or a supervisor stack pointer for stack operations. The processor identifies a bus access (supervisor or user mode) via the function codes so that differentiation between supervisor and user can be maintained. The memory management unit uses the indication of privilege level to control and translate memory accesses to protect supervisor code, data, and resources from access by user programs.

In many systems, the majority of programs execute at the user level. User programs can access only their own code and data areas and can be restricted from accessing other information. The operating system typically executes at the supervisor privilege level. It has access to all resources, performs the overhead tasks for the user level programs, and coordinates their activities.

4.1.1 Supervisor Privilege Level

The supervisor level is the higher privilege level. The privilege level is determined by the S bit of the status register; if the S bit is set, the supervisor privilege level applies, and all instructions are executable. The bus cycles for instructions executed at the supervisor level are normally classified as supervisor references, and the values of the function codes on FC0–FC2 refer to supervisor address spaces.

In a multitasking operating system, it is more efficient to have a supervisor stack space associated with each user task and a separate stack space for interrupt associated tasks. The MC68030 provides two supervisor stacks, master and interrupt; the M bit of the status register selects which of the two is active. When the M bit is set to one, supervisor stack pointer references (either implicit or by specifying address register A7) access the master stack pointer (MSP). The operating system sets the MSP for each task to point to a task-related area of supervisor data space. This separates task-related supervisor activity from asynchronous, I/O-related supervisor tasks that may be only coincidental to the currently executing task. The master stack (MSP) can separately maintain task control information for each currently executing user task, and the software updates the MSP when a task switch is performed, providing an efficient means for transferring task-related stack items. The other supervisor stack (ISP) can be used for interrupt control information and workspace area as interrupt handling routines require.

When the M bit is clear, the MC68030 is in the interrupt mode of the supervisor privilege level, and operation is the same as in the MC68000, MC68008, and MC68010 supervisor mode. (The processor is in this mode after a reset operation.) All supervisor stack pointer references access the interrupt stack pointer (ISP) in this mode.

The value of the M bit in the status register does not affect execution of privileged instructions; both master and interrupt modes are at the supervisor privilege level. Instructions that affect the M bit are MOVE to SR, ANDI to SR, EORI to SR, ORI to SR, and RTE. Also, the processor automatically saves the M-bit value and clears it in the SR as part of the exception processing for interrupts.

All exception processing is performed at the supervisor privilege level. All bus cycles generated during exception processing are supervisor references, and all stack accesses use the active supervisor stack pointer.

4.1.2 User Privilege Level

The user level is the lower privilege level. The privilege level is determined by the S bit of the status register; if the S bit is clear, the processor executes instructions at the user privilege level.

Most instructions execute at either privilege level, but some instructions that have important system effects are privileged and can only be executed at the supervisor level. For instance, user programs are not allowed to execute the STOP instruction or the RESET instruction. To prevent a user program from entering the supervisor privilege level, except in a controlled manner, instructions that can alter the S bit in the status register are privileged. The TRAP #n instruction provides controlled access to operating system services for user programs.

The bus cycles for an instruction executed at the user privilege level are classified as user references, and the values of the function codes on FC0-FC2 specify user address spaces. The memory management unit of the processor, when it is enabled, uses the value of the function codes to distinguish between user and supervisor activity and to control access to protected portions of the address space. While the processor is at the user level, references to the system stack pointer implicitly, or to address register seven (A7) explicitly, refer to the user stack pointer (USP).

4.1.3 Changing Privilege Level

To change from the user to the supervisor privilege level, one of the conditions that causes the processor to perform exception processing must occur. This causes a change from the user level to the supervisor level and can cause a change from the master mode to the interrupt mode. Exception processing saves the current values of the S and M bits of the status register (along with the rest of the status register) on the active supervisor stack, and then sets the S bit, forcing the processor into the supervisor privilege level. When the exception being processed is an interrupt and the M bit is set, the M bit is cleared, putting the processor into the interrupt mode. Execution of instructions continues at the supervisor level to process the exception condition.

To return to the user privilege level, a system routine must execute one of the following instructions: MOVE to SR, ANDI to SR, EORI to SR, ORI to SR, or RTE. The MOVE, ANDI, EORI, and ORI to SR and RTE instructions execute at the supervisor privilege level and can modify the S bit of the status register. After these instructions execute, the instruction pipeline is flushed and is refilled from the appropriate address space. This is indicated externally by the assertion of the $\overline{\text{REFILL}}$ signal.

The RTE instruction returns to the program that was executing when the exception occurred. It restores the exception stack frame saved on the supervisor stack. If the frame on top of the stack was generated by an interrupt, trap, or instruction exception, the RTE instruction restores the status register and program counter to the values saved on the supervisor stack. The processor then continues execution at the restored program counter address and at the privilege level determined by the S bit of the restored status register. If the frame on top of the stack was generated by a bus fault (bus error or address error exception), the RTE instruction restores the entire saved processor state from the stack.

4.2 ADDRESS SPACE TYPES

The processor specifies a target address space for every bus cycle with the function code signals according to the type of access required. In addition to distinguishing between supervisor/user and program/data, the processor can identify special processor cycles, such as the interrupt acknowledge cycle, and the memory management unit can control accesses and translate addresses appropriately. Table 4-1 lists the types of accesses defined for the MC68030 and the corresponding values of function codes FC0–FC2.

Table 4-1. Address Space Encodings

FC2	FC1	FC0	Address Space
0	0	0	(Undefined, Reserved)*
0	0	1	User Data Space
0	1	0	User Program Space
0	1	1	(Undefined, Reserved)*
1	0	0	(Undefined, Reserved)*
1	0	1	Supervisor Data Space
1	1	0	Supervisor Program Space
1	1	1	CPU Space

*Address space 3 is reserved for user definition; whereas, 0 and 4 are reserved for future use by Motorola.

The memory locations of user program and data accesses are not predefined. Neither are the locations of supervisor data space. During reset, the first two long words beginning at memory location zero in the supervisor program space are used for processor initialization. No other memory locations are explicitly defined by the MC68030.

A function code of $7 ([FC2:FC0] = 111) selects the CPU address space. This is a special address space that does not contain instructions or operands but is reserved for special processor functions. The processor uses accesses in this space to communicate with external devices for special purposes. For example, all M68000 processors use the CPU space for interrupt acknowledge cycles. The MC68020 and MC68030 also generate CPU space accesses for breakpoint acknowledge and coprocessor operations.

Supervisor programs can use the MOVES instruction to access all address spaces, including the user spaces and the CPU address space. Although the MOVES instruction can be used to generate CPU space cycles, this may interfere with proper system operation. Thus, the use of MOVES to access the CPU space should be done with caution.

4.3 EXCEPTION PROCESSING

An exception is defined as a special condition that pre-empts normal processing. Both internal and external conditions cause exceptions. External conditions that cause exceptions are interrupts from external devices, bus errors, coprocessor detected errors, and reset. Instructions, address errors, tracing, and breakpoints are internal conditions that cause exceptions. The TRAP, TRAPcc, TRAPV, cpTRAPcc, CHK, CHK2, RTE, and DIV instructions can all generate exceptions as part of their normal execution. In addition, illegal instructions, privilege violations, and coprocessor protocol violations cause exceptions.

Exception processing, which is the transition from the normal processing of a program to the processing required for the exception condition, involves the exception vector table and an exception stack frame. The following paragraphs describe the vector table and a generalized exception stack frame. Exception processing is discussed in detail in **Section 8 Exception Processing**. Coprocessor detected exceptions are discussed in detail in **Section 10 Coprocessor Interface Description**.

4.3.1 Exception Vectors

The vector base register (VBR) contains the base address of the 1024-byte exception vector table, which consists of 256 exception vectors. Exception vectors contain the memory addresses of routines that begin execution at the completion of exception processing. These routines perform a series of operations appropriate for the corresponding exceptions. Because the exception vectors contain memory addresses, each consists of one long word, except for the reset vector. The reset vector consists of two long words: the address used to initialize the interrupt stack pointer and the address used to initialize the program counter.

The address of an exception vector is derived from an 8-bit vector number and the VBR. The vector numbers for some exceptions are obtained from an external device; others are supplied automatically by the processor. The processor multiplies the vector number by four to calculate the vector offset, which it adds to the VBR. The sum is the memory address of the vector. All exception vectors are located in supervisor data space, except the reset vector, which is located in supervisor program space. Only the initial reset vector is fixed in the processor's memory map; once initialization is complete, there are no fixed assignments. Since the VBR provides the base address of the vector table, the vector table can be located anywhere in memory; it can even be dynamically relocated for each task that is executed by an operating system. Details of exception processing are provided in **Section 8 Exception Processing**, and Table 8-1 lists the exception vector assignments.

4.3.2 Exception Stack Frame

Exception processing saves the most volatile portion of the current processor context on the top of the supervisor stack. This context is organized in a format called the exception stack frame. This information always includes a copy of the status register, the program counter, the vector offset of the vector, and the frame format field. The frame format field identifies the type of stack frame. The RTE instruction uses the value in the format field to properly restore the information stored in the stack frame and to deallocate the stack space. The general form of the exception stack frame is illustrated in Figure 4-1. Refer to **Section 8 Exception Processing** for a complete list of exception stack frames.

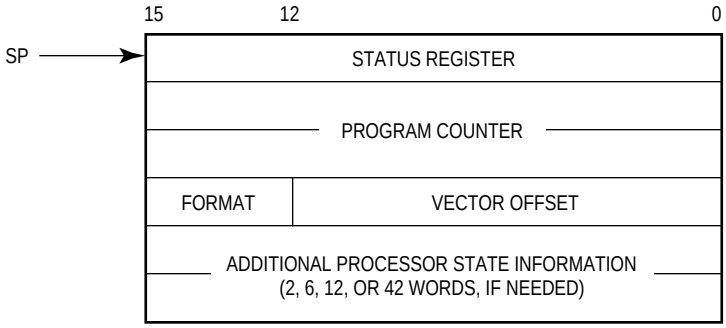


Figure 4-1. General Exception Stack Frame

SECTION 5
SIGNAL DESCRIPTION

This section contains brief descriptions of the input and output signals in their functional groups, as shown in Figure 5-1. Each signal is explained in a brief paragraph with reference to other sections that contain more detail about the signal and the related operations.

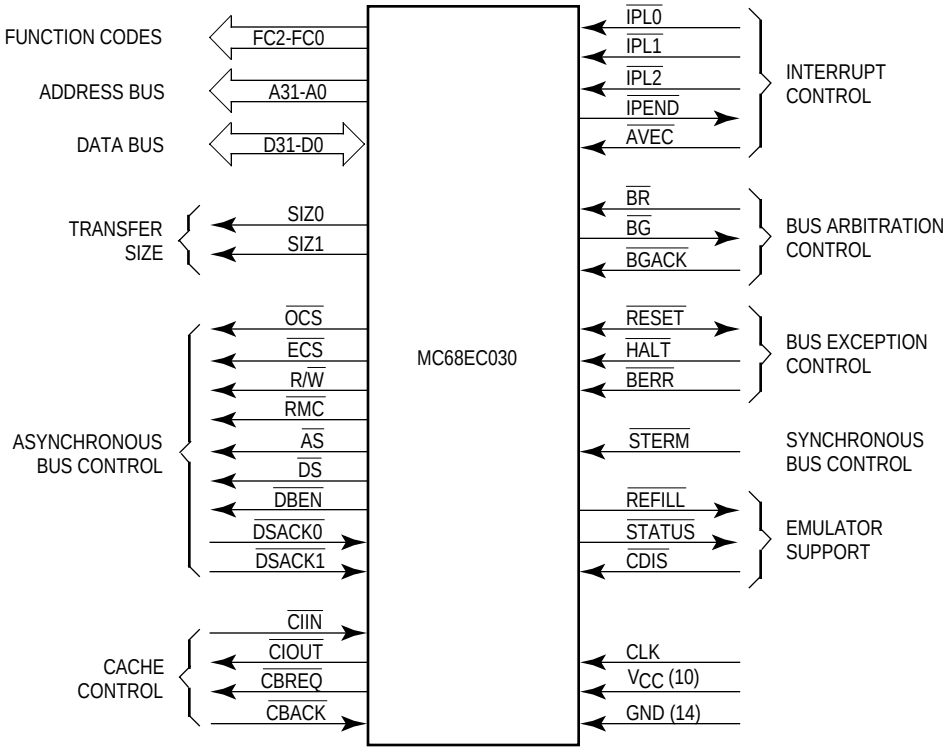


Figure 5-1. Functional Signal Groups

NOTE

In this section and in the remainder of the manual, **assertion** and **negation** are used to specify forcing a signal to a particular state. In particular, assertion and assert refer to a signal that is active or true; negation and negate indicate a signal that is inactive or false. These terms are used independently of the voltage level (high or low) that they represent.

5.1 SIGNAL INDEX

The input and output signals for the MC68030 are listed in Table 5-1. Both the names and mnemonics are shown along with brief signal descriptions. For more detail on each signal, refer to the paragraph in this section named for the signal and the reference in that paragraph to a description of the related operations.

Guaranteed timing specifications for the signals listed in Table 5-1 can be found in M68030EC/D, *MC68030 Electrical Specifications*.

Table 5-1. Signal Index (Sheet 1 of 2)

Signal Name	Mnemonic	Function
Function Codes	FC0–FC2	3-bit function code used to identify the address space of each bus cycle.
Address Bus	A0–A31	32-bit address bus.
Data Bus	D0–D31	32-bit data bus used to transfer 8, 16, 24, or 32 bits of data per bus cycle.
Size	SIZ0/SIZ1	Indicates the number of bytes remaining to be transferred for this cycle. These signals, together with A0 and A1, define the active sections of the data bus.
Operand Cycle Start	$\overline{\text{OCS}}$	Identical operation to that of ECS except that OCS is asserted only during the first bus cycle of an operand transfer.
External Cycle Start	$\overline{\text{ECS}}$	Provides an indication that a bus cycle is beginning.
Read/Write	R/$\overline{\text{W}}$	Defines the bus transfer as a processor read or write.
Read-Modify-Write Cycle	$\overline{\text{RMC}}$	Provides an indicator that the current bus cycle is part of an indivisible read-modify-write operation.
Address Strobe	$\overline{\text{AS}}$	Indicates that a valid address is on the bus.
Data Strobe	$\overline{\text{DS}}$	Indicates that valid data is to be placed on the data bus by an external device or has been placed on the data bus by the MC68030.
Data Buffer Enable	$\overline{\text{DBEN}}$	Provides an enable signal for external data buffers.

Table 5-1. Signal Index (Sheet 2 of 2)

Signal Name	Mnemonic	Function
Data Transfer and Size Acknowledge	$\overline{\text{DSACK0}}$/ $\overline{\text{DSACK1}}$	Bus response signals that indicate the requested data transfer operation is completed. In addition, these two lines indicate the size of the external bus port on a cycle-by-cycle basis and are used for asynchronous transfers.
Synchronous Termination	$\overline{\text{STERM}}$	Bus response signal that indicates a port size of 32 bits and that data may be latched on the next falling clock edge.
Cache Inhibit In	$\overline{\text{CIIN}}$	Prevents data from being loaded into the MC68030 instruction and data caches.
Cache Inhibit Out	$\overline{\text{CIOUT}}$	Reflects the CI bit in ATC entries or TTx register; indicates that external caches should ignore these accesses
Cache Burst Request	$\overline{\text{CBREQ}}$	Indicates a burst request for the instruction or data cache.
Cache Burst Acknowledge	$\overline{\text{CBACK}}$	Indicates that the accessed device can operate in burst mode.
Interrupt Priority Level	$\overline{\text{IPL0}}$–$\overline{\text{IPL2}}$	Provides an encoded interrupt level to the processor.
Interrupt Pending	$\overline{\text{IPEND}}$	Indicates that an interrupt is pending.
Autovector	$\overline{\text{AVEC}}$	Requests an autovector during an interrupt acknowledge cycle.
Bus Request	$\overline{\text{BR}}$	Indicates that an external device requires bus mastership.
Bus Grant	$\overline{\text{BG}}$	Indicates that an external device may assume bus mastership.
Bus Grant Acknowledge	$\overline{\text{BGACK}}$	Indicates that an external device has assumed bus mastership.
Reset	$\overline{\text{RESET}}$	System reset.
Halt	$\overline{\text{HALT}}$	Indicates that the processor should suspend bus activity.
Bus Error	$\overline{\text{BERR}}$	Indicates that an erroneous bus operation is being attempted.
Cache Disable	$\overline{\text{CDIS}}$	Dynamically disables the on-chip cache to assist emulator support.
MMU Disable	$\overline{\text{MMUDIS}}$	Dynamically disables the translation mechanism of the MMU.
Pipe Refill	$\overline{\text{REFILL}}$	Indicates when the MC68030 is beginning to fill pipeline.
Microsequencer Status	$\overline{\text{STATUS}}$	Indicates the state of the microsequencer
Clock	CLK	Clock input to the processor.
Power Supply	V_{CC}	Power supply.
Ground	GND	Ground connection

5.2 FUNCTION CODE SIGNALS (FC0–FC2)

These three-state outputs identify the address space of the current bus cycle. Table 4-1 shows the relationship of the function code signals to the privilege levels and the address spaces. Refer to **4.2 Address Space Types** for more information.

5.3 ADDRESS BUS (A0–A31)

These three-state outputs provide the address for the current bus cycle, except in the CPU address space. Refer to **4.2 Address Space Types** for more information on the CPU address space. A31 is the most significant address signal. Refer to **7.1.2 Address Bus** for information on the address bus and its relationship to bus operation.

5.4 DATA BUS (D0–D31)

These three-state bidirectional signals provide the general-purpose data path between the MC68030 and all other devices. The data bus can transfer 8, 16, 24, or 32 bits of data per bus cycle. D31 is the most significant bit of the data bus. Refer to **7.1.4 Data Bus** for more information on the data bus and its relationship to bus operation.

5.5 TRANSFER SIZE SIGNALS (SIZ0, SIZ1)

These three-state outputs indicate the number of bytes remaining to be transferred for the current bus cycle. With A0, A1, $\overline{\text{DSACK0}}$, $\overline{\text{DSACK1}}$, and $\overline{\text{STERM}}$, SIZ0 and SIZ1 define the number of bits transferred on the data bus. Refer to **7.2.1 Dynamic Bus Sizing** for more information on the size signals and their use in dynamic bus sizing.

5.6 BUS CONTROL SIGNALS

The following signals control synchronous bus transfer operations for the MC68030.

5.6.1 Operand Cycle Start ($\overline{OCS}$)

This output signal indicates the beginning of the first external bus cycle for an instruction prefetch or a data operand transfer. $\overline{OCS}$ is not asserted for subsequent cycles that are performed due to dynamic bus sizing or operand misalignment. **7.1.1 Bus Control Signals** for information about the relationship of $\overline{OCS}$ to bus operation.

5.6.2 External Cycle Start ($\overline{ECS}$)

This output signal indicates the beginning of a bus cycle of any type. **7.1.1 Bus Control Signals** for information about the relationship of $\overline{ECS}$ to bus operation.

5.6.3 Read/Write (R/$\overline{W}$)

This three-state output signal defines the type of bus cycle. A high level indicates a read cycle; a low level indicates a write cycle. Refer to **7.1.1 Bus Control Signals** for information about the relationship of R/$\overline{W}$ to bus operation.

5.6.4 Read-Modify-Write Cycle ($\overline{RMC}$)

This three-state output signal identifies the current bus cycle as part of an indivisible read-modify-write operation; it remains asserted during all bus cycles of the read-modify-write operation. Refer to **7.1.1 Bus Control Signals** for information about the relationship of $\overline{RMC}$ to bus operation.

5.6.5 Address Strobe ($\overline{AS}$)

This three-state output indicates that a valid address is on the address bus. The function code, size, and read/write signals are also valid when $\overline{AS}$ is asserted. Refer to **7.1.3 Address Strobe** for information about the relationship of $\overline{AS}$ to bus operation.

5.6.6 Data Strobe ($\overline{DS}$)

During a read cycle, this three-state output indicates that an external device should place valid data on the data bus. During a write cycle, the data strobe indicates that the MC68030 has placed valid data on the bus. During two-clock synchronous write cycles, the MC68030 does not assert $\overline{DS}$. Refer to **7.1.5 Data Strobe** for more information about the relationship of $\overline{DS}$ to bus operation.

5.6.7 Data Buffer Enable ($\overline{DBEN}$)

This output is an enable signal for external data buffers. This signal may not be required in all systems. The timing of this signal may preclude its use in a system that supports two-clock synchronous bus cycles. Refer to **7.1.6 Data Buffer Enable** for more information about the relationship of $\overline{DBEN}$ to bus operation.

5.6.8 Data Transfer and Size Acknowledge ($\overline{DSACK0}$, $\overline{DSACK1}$)

These inputs indicate the completion of a requested data transfer operation. In addition, they indicate the size of the external bus port at the completion of each cycle. These signals apply only to asynchronous bus cycles. Refer to **7.1.7 Bus Cycle Termination Signals** for more information on these signals and their relationship to dynamic bus sizing.

5.6.9 Synchronous Termination ($\overline{STERM}$)

This input is a bus handshake signal indicating that the addressed port size is 32 bits and that data is to be latched on the next falling clock edge for a read cycle. This signal applies only to synchronous operation. Refer to **7.1.7 Bus Cycle Termination Signals** for more information about the relationship of $\overline{STERM}$ to bus operation.

5.7 CACHE CONTROL SIGNALS

The following signals relate to the on-chip caches.

5.7.1 Cache Inhibit Input ($\overline{\text{CIIN}}$)

This input signal prevents data from being loaded into the MC68030 instruction and data caches. It is a synchronous input signal and is interpreted on a bus-cycle-by-bus-cycle basis. $\overline{\text{CIIN}}$ is ignored during all write cycles. Refer to **6.1 On-Chip Cache Organization and Operation** for information on the relationship of CIIN to the on-chip caches.

5.7.2 Cache Inhibit Output ($\overline{\text{CIOUT}}$)

This three-state output signal reflects the state of the CI bit in the address translation cache entry for the referenced logical address, indicating that an external cache should ignore the bus transfer. When the referenced logical address is within an area specified for transparent translation, the CI bit of the appropriate transparent translation register controls the state of $\overline{\text{CIOUT}}$. Refer to **Section 9 Memory Management Unit** for more information about the address translation cache and transparent translation. Also, refer to **Section 6 On-Chip Cache Memories** for the effect of $\overline{\text{CIOUT}}$ on the internal caches.

5.7.3 Cache Burst Request ($\overline{\text{CBREQ}}$)

This three-state output signal requests a burst mode operation to fill a line in the instruction or data cache. Refer to **6.1.3 Cache Filling** for filling information and **7.3.7 Burst Operation Cycles** for bus cycle information pertaining to burst mode operations.

5.7.4 Cache Burst Acknowledge ($\overline{\text{CBACK}}$)

This input signal indicates that the accessed device can operate in the burst mode and can supply at least one more long word for the instruction or data cache. Refer to **7.3.7 Burst Operation Cycles** for information about burst mode operation.

5.8 INTERRUPT CONTROL SIGNALS

The following signals are the interrupt control signals for the MC68030.

5.8.1 Interrupt Priority Level Signals

These input signals provide an indication of an interrupt condition and the encoding of the interrupt level from a peripheral or external prioritizing circuitry. $\overline{IPL2}$ is the most significant bit of the level number. For example, since the $\overline{IPL}$n signals are active low, $\overline{IPL0}$–$\overline{IPL2}$ equal to $5 corresponds to an interrupt request at interrupt level 2. Refer to **8.1.9 Interrupt Exceptions** for information on MC68030 interrupts.

5.8.2 Interrupt Pending ($\overline{IPEND}$)

This output signal indicates that an interrupt request has been recognized internally and exceeds the current interrupt priority mask in the status register (SR). This output is for use by external devices (coprocessors and other bus masters, for example) to predict processor operation on the following instruction boundaries. Refer to **8.1.9 Interrupt Exceptions** for interrupt information. Also, refer to **7.4.1 Interrupt Acknowledge Bus Cycles** for bus information related to interrupts.

5.8.3 Autovector ($\overline{AVEC}$)

This input signal indicates that the MC68030 should generate an automatic vector during an interrupt acknowledge cycle. Refer to **7.4.1.2 Autovector Interrupt Acknowledge Cycle** for more information about automatic vectors.

5.9 BUS ARBITRATION CONTROL SIGNALS

The following signals are the three bus arbitration control signals used to determine which device in a system is the bus master.

5.9.1 Bus Request ($\overline{BR}$)

This input signal indicates that an external device needs to become the bus master. This is typically a "wire-ORed” input (but does not need to be constructed from open-collector devices). Refer to **7.7 Bus Arbitration** for more information.

5.9.2 Bus Grant ($\overline{BG}$)

This output indicates that the MC68030 will release ownership of the bus master when the current processor bus cycle completes. Refer to **7.7.2 Bus Grant** for more information.

5.9.3 Bus Grant Acknowledge ($\overline{BGACK}$)

This input indicates that an external device has become the bus master. Refer to **7.7.3 Bus Grant Acknowledge** for more information.

5.10 BUS EXCEPTION CONTROL SIGNALS

The following signals are the bus exception control signals for the MC68030.

5.10.1 Reset ($\overline{RESET}$)

This bidirectional open-drain signal is used to initiate a system reset. An external reset signal resets the MC68030 as well as all external devices. A reset signal from the processor (asserted as part of the RESET instruction) resets external devices only; the internal state of the processor is not altered. Refer to **7.8 Reset Operation** for a description of reset bus operation and **8.1.1 Reset Exception** for information about the reset exception.

5.10.2 Halt ($\overline{HALT}$)

The halt signal indicates that the processor should suspend bus activity or, when used with $\overline{BERR}$, that the processor should retry the current cycle. Refer to **7.5 Bus Exception Control Cycles** for a description of the effects of $\overline{HALT}$ on bus operations.

5.10.3 Bus Error ($\overline{BERR}$)

The bus error signal indicates that an invalid bus operation is being attempted or, when used with $\overline{HALT}$, that the processor should retry the current cycle. Refer to **7.5 Bus Exception Control Cycles** for a description of the effects of $\overline{BERR}$ on bus operations.

5.11 EMULATOR SUPPORT SIGNALS

The following signals support emulation by providing a means for an emulator to disable the on-chip caches and memory management unit and by supplying internal status information to an emulator. Refer to **Section 12 Applications Information** for more detailed information on emulation support.

5.11.1 Cache Disable ($\overline{\text{CDIS}}$)

The cache disable signal dynamically disables the on-chip caches to assist emulator support. Refer to **6.1 On-Chip Cache Organization and Operation** for information about the caches; refer to **Section 12 Applications Information** for a description of the use of this signal by an emulator. $\overline{\text{CDIS}}$ does not flush the data and instruction caches; entries remain unaltered and become available again when $\overline{\text{CDIS}}$ is negated.

5.11.2 MMU Disable ($\overline{\text{MMUDIS}}$)

The MMU disable signal dynamically disables the translation of addresses by the MMU. Refer to **9.4 Address Translation Cache** for a description of address translation; refer to **Section 12 Applications Information** for a description of the use of this signal by an emulator. The assertion of $\overline{\text{MMUDIS}}$ does not flush the address translation cache (ATC); ATC entries become available again when $\overline{\text{MMUDIS}}$ is negated.

5.11.3 Pipeline Refill ($\overline{\text{REFILL}}$)

The pipeline refill signal indicates that the MC68030 is beginning to refill the internal instruction pipeline. Refer to **Section 12 Applications Information** for a description of the use of this signal by an emulator.

5.11.4 Internal Microsequencer Status ($\overline{\text{STATUS}}$)

The microsequencer status signal indicates the state of the internal microsequencer. The varying number of clocks for which this signal is asserted indicates instruction boundaries, pending exceptions, and the halted condition. Refer to **Section 12 Applications Information** for a description of the use of this signal by an emulator.

5.12 CLOCK (CLK)

The clock signal is the clock input to the MC68030. It is a TTL-compatible signal. Refer to **Section 12 Applications Information** for suggestions on clock generation.

5.13 POWER SUPPLY CONNECTIONS

The MC68030 requires connection to a V_{CC} power supply, positive with respect to ground. The V_{CC} connections are grouped to supply adequate current for the various sections of the processor. The ground connections are similarly grouped. **Section 14 Ordering Information and Mechanical Data** describes the groupings of V_{CC} and ground connections, and **Section 12 Applications Information** describes a typical power supply interface.

5.14 SIGNAL SUMMARY

Table 5-2 provides a summary of the electrical characteristics of the signals discussed in this section.

Table 5-2. Signal Summary

Signal Function	Signal Name	Input/Output	Active State	Three-State
Function Codes	FC0–FC2	Output	High	Yes
Address Bus	A0–A31	Output	High	Yes
Data Bus	D0–D31	Input/Output	High	Yes
Transfer Size	SIZ0/SIZ1	Output	High	Yes
Operand Cycle Start	$\overline{\text{OCS}}$	Output	Low	No
External Cycle Start	$\overline{\text{ECS}}$	Output	Low	No
Read/Write	R/$\overline{\text{W}}$	Output	High/Low	Yes
Read-Modify-Write Cycle	$\overline{\text{RMC}}$	Output	Low	Yes
Address Strobe	$\overline{\text{AS}}$	Output	Low	Yes
Data Strobe	$\overline{\text{DS}}$	Output	Low	Yes
Data Buffer Enable	$\overline{\text{DBEN}}$	Output	Low	Yes
Data Transfer and Size Acknowledge	$\overline{\text{DSACK0}}$/ $\overline{\text{DSACK1}}$	Input	Low	—
Synchronous Termination	$\overline{\text{STERM}}$	Input	Low	—
Cache Inhibit In	$\overline{\text{CIIN}}$	Input	Low	—
Cache Inhibit Out	$\overline{\text{CIOUT}}$	Output	Low	Yes
Cache Burst Request	$\overline{\text{CBREQ}}$	Output	Low	Yes
Cache Burst Acknowledge	$\overline{\text{CBACK}}$	Input	Low	—
Interrupt Priority Level	$\overline{\text{IPL0}}$–$\overline{\text{IPL2}}$	Input	Low	—
Interrupt Pending	$\overline{\text{IPEND}}$	Output	Low	No
Autovector	$\overline{\text{AVEC}}$	Input	Low	—
Bus Request	$\overline{\text{BR}}$	Input	Low	—
Bus Grant	$\overline{\text{BG}}$	Output	Low	No
Bus Grant Acknowledge	$\overline{\text{BGACK}}$	Input	Low	—
Reset	$\overline{\text{RESET}}$	Input/Output	Low	No
Halt	$\overline{\text{HALT}}$	Input	Low	—
Bus Error	$\overline{\text{BERR}}$	Input	Low	—
Cache Disable	$\overline{\text{CDIS}}$	Input	Low	—
MMU Disable	$\overline{\text{MMUDIS}}$	Input	Low	—
Pipeline Refill	$\overline{\text{REFILL}}$	Output	Low	No
Microsequencer Status	$\overline{\text{STATUS}}$	Output	Low	No
Clock	CLK	Input	—	—
Power Supply	VCC	Input	—	—
Ground	GND	Input	—	—

SECTION 6
ON-CHIP CACHE MEMORIES

The MC68030 microprocessor includes a 256-byte on-chip instruction cache and a 256-byte on-chip data cache that are accessed by logical (virtual) addresses. These caches improve performance by reducing external bus activity and increasing instruction throughput.

Reduced external bus activity increases overall performance by increasing the availability of the bus for use by external devices (in systems with more than one bus master, such as a processor and a DMA device) without degrading the performance of the MC68030. An increase in instruction throughput results when instruction words and data required by a program are available in the on-chip caches and the time required to access them on the external bus is eliminated. Additionally, instruction throughput increases when instruction words and data can be accessed simultaneously.

As shown in Figure 6-1, the instruction cache and the data cache are connected to separate on-chip address and data buses. The address buses are combined to provide the logical address to the memory management unit (MMU). The MC68030 initiates an access to the appropriate cache for the requested instruction or data operand at the same time that it initiates an access for the translation of the logical address in the address translation cache of the MMU. When a hit occurs in the instruction or data cache and the MMU validates the access on a write, the information is transferred from the cache (on a read) or to the cache and the bus controller (on a write). When a hit does not occur, the MMU translation of the address is used for an external bus cycle to obtain the instruction or operand. Regardless of whether or not the required operand is located in one of the on-chip caches, the address translation cache of the MMU performs logical-to-physical address translation in parallel with the cache lookup in case an external cycle is required.

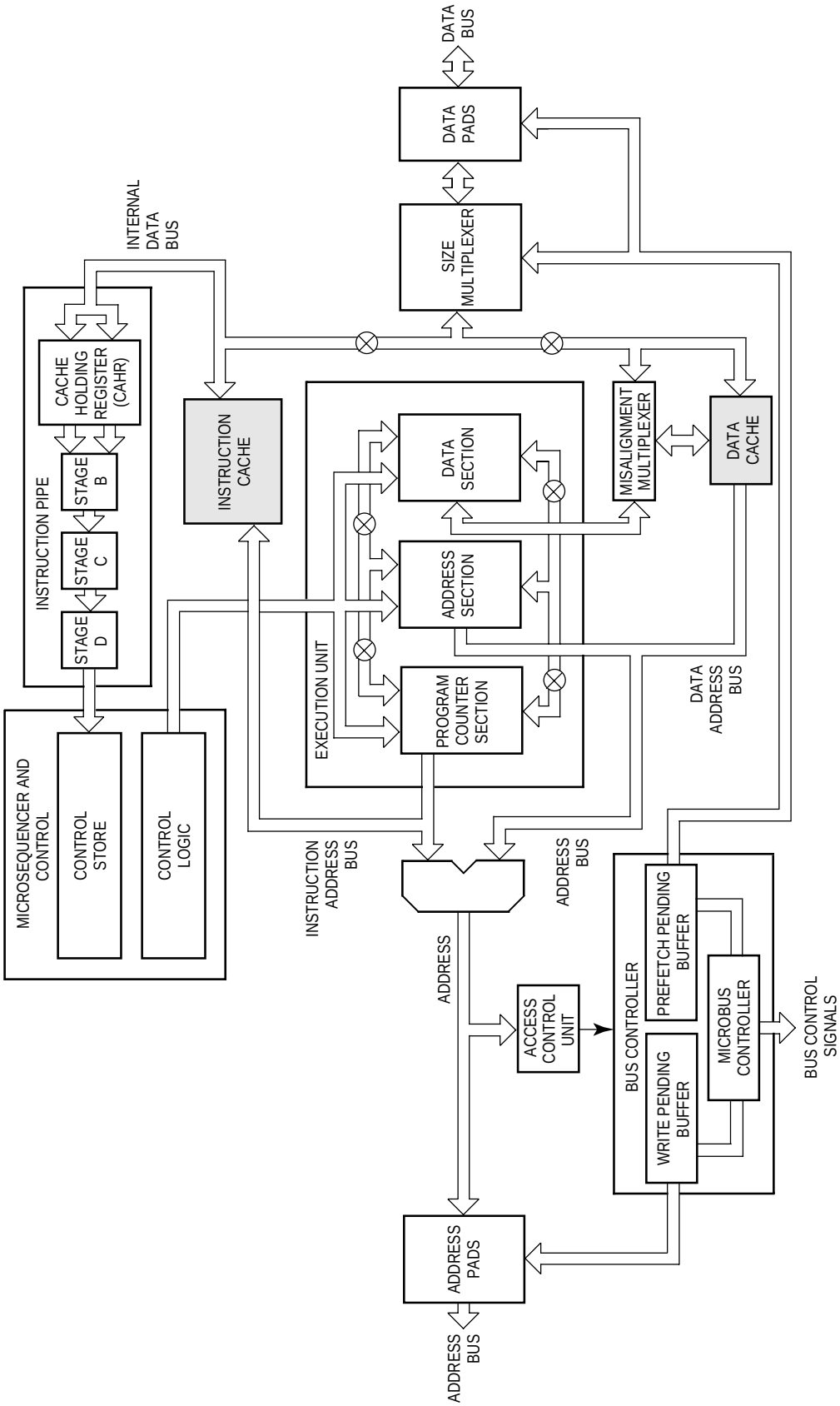


Figure 6-1. Internal Caches and the MC68030

6.1 ON-CHIP CACHE ORGANIZATION AND OPERATION

Both on-chip caches are 256-byte direct-mapped caches, each organized as 16 lines. Each line consists of four entries, and each entry contains four bytes. The tag field for each line contains a valid bit for each entry in the line; each entry is independently replaceable. When appropriate, the bus controller requests a burst mode operation to replace an entire cache line. The cache control register (CACR) is accessible by supervisor programs to control the operation of both caches.

System hardware can assert the cache disable ($\overline{\text{CDIS}}$) signal to disable both caches. The assertion of $\overline{\text{CDIS}}$ disables the caches, regardless of the state of the enable bits in CACR. $\overline{\text{CDIS}}$ is primarily intended for use by in-circuit emulators.

Another input signal, cache inhibit in ($\overline{\text{CIIN}}$), inhibits caching of data reads or instruction prefetches on a bus-cycle by bus-cycle basis. Examples of data that should not be cached are data for I/O devices and data from memory devices that cannot supply a full port width of data, regardless of the size of the required operand.

Subsequent paragraphs describe how $\overline{\text{CIIN}}$ is used during the filling of the caches.

An output signal, cache inhibit out ($\overline{\text{CIOUT}}$), reflects the state of the cache inhibit (CI) bit from the MMU of either the address translation cache entry that corresponds to a specified logical address or the transparent translation register that corresponds to that address. Whenever the appropriate CI bit is set for either a read or a write access and an external bus cycle is required, $\overline{\text{CIOUT}}$ is asserted and the instruction and data caches are ignored for the access. This signal can also be used by external hardware to inhibit caching in external caches.

Whenever a read access occurs and the required instruction word or data operand is resident in the appropriate on-chip cache (no external bus cycle is required), the MMU is completely ignored, unless an invalid translation resides in the MMU at that time (see next two paragraphs). Therefore, the state of the corresponding CI bits in the MMU are also ignored. The MMU is used to validate all accesses that require external bus cycles; an address translation must be available and valid, protections are checked, and the $\overline{\text{CIOUT}}$ signal is asserted appropriately.

An external access is defined as “cachable” for either the instruction or data cache when all the following conditions apply:

- The cache is enabled with the appropriate bit in the CACR set.
- The $\overline{\text{CDIS}}$ signal is negated.
- The $\overline{\text{CIIN}}$ signal is negated for the access.
- The $\overline{\text{CIOUT}}$ signal is negated for the access.
- The MMU validates the access.

Because both the data and instruction caches are referenced by logical addresses, they should be flushed during a task switch or at any time the logical-to-physical address mapping changes, including when the MMU is first enabled. In addition, if a page descriptor is currently marked as valid and is later changed to the invalid type (due to a context switch or a page replacement operation) *entries in the on-chip instruction or data cache corresponding to the physical page must be first cleared (invalidated)*. Otherwise, if on-chip cache entries are valid for pages with descriptors in memory marked invalid, processor operation is unpredictable.

Data read and write accesses to the same address should also have consistent cachability status to ensure that the data in the cache remains consistent with external memory. For example, if $\overline{\text{CIOUT}}$ is negated for read accesses within a page and the MMU configuration is changed so that $\overline{\text{CIOUT}}$ is subsequently asserted for write accesses within the same page, those write accesses do not update data in the cache, and stale data may result. Similarly, when the MMU maps multiple logical addresses to the same physical address, all accesses to those logical addresses should have the same cachability status.

6.1.1 Instruction Cache

The instruction cache is organized with a line size of four long words, as shown in Figure 6-2. Each of these long words is considered a separate cache entry as each has a separate valid bit. All four entries in a line have the same tag address. Burst filling all four long words can be advantageous when the time spent in filling the line is not long relative to the equivalent bus-cycle time for four nonburst long-word accesses, because of the probability that the contents of memory adjacent to or close to a referenced operand or instruction is also required by subsequent accesses. Dynamic RAMs supporting fast access modes (page, nibble, or static column) are easily employed to support the MC68030 burst mode.

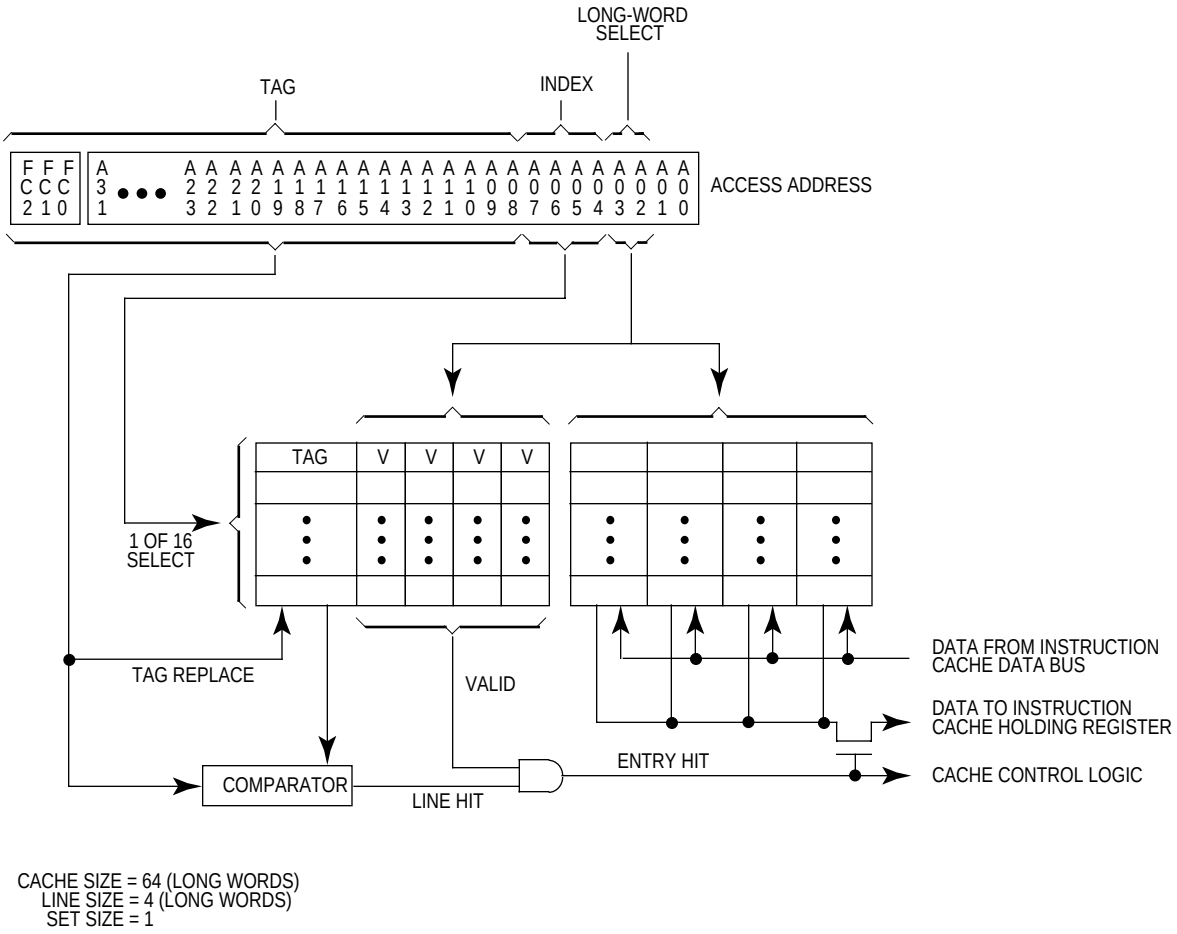


Figure 6-2. On-Chip Instruction Cache Organization

When enabled, the instruction cache is used to store instruction prefetches (instruction words and extension words) as they are requested by the CPU. Instruction prefetches are normally requested from sequential memory addresses except when a change of program flow occurs (e.g., a branch taken) or when an instruction is executed that can modify the status register, in which cases the instruction pipe is automatically flushed and refilled. The output signal $\overline{\text{REFILL}}$ indicates this condition. For more information on the operation of this signal, refer to **Section 12 Applications Information**.

In the instruction cache, each of the 16 lines has a tag consisting of the 24 most significant logical address bits, the FC2 function code bit (used to distinguish between user and supervisor accesses), and the four valid bits (one corresponding to each long word). Refer to Figure 6-2 for the instruction cache organization. Address bits A7–A4 select one of 16 lines and its associated tag. The comparator compares the address and function code bits in the selected tag with address bits A31–A8 and FC2 from the internal prefetch request to determine if the requested word is in the cache. A cache hit occurs when there is a tag match and the corresponding valid bit (selected by A3–A2) is set. On a cache hit, the word selected by address bit A1 is supplied to the instruction pipe.

When the address and function code bits do not match or the requested entry is not valid, a miss occurs. The bus controller initiates a long-word prefetch operation for the required

instruction word and loads the cache entry, provided the entry is cachable. A burst mode operation may be requested to fill an entire cache line. If the function code and address bits match and the corresponding long word is not valid (but one or more of the other three valid bits for that line are set) a single entry fill operation replaces the required long word only, using a normal prefetch bus cycle or cycles (no burst).

6.1.2 Data Cache

The data cache stores data references to any address space except CPU space (FC=$7), including those references made with PC relative addressing modes and accesses made with the MOVES instruction. Operation of the data cache is similar to that of the instruction cache, except for the address comparison and cache filling operations. The tag of each line in the data cache contains function code bits FC0, FC1, and FC2 in addition to address bits A31–A8. The cache control circuitry selects the tag using bits A7–A4 and compares it to the corresponding bits of the access address to determine if a tag match has occurred. Address bits A3–A2 select the valid bit for the appropriate long word in the cache to determine if an entry hit has occurred. Misaligned data transfers may span two data cache entries. In this case, the processor checks for a hit one entry at a time. Therefore, it is possible that a portion of the access results in a hit and a portion results in a miss. The hit and miss are treated independently. Figure 6-3 illustrates the organization of the data cache.

The operation of the data cache differs for read and write cycles. A data read cycle operates exactly like an instruction cache read cycle; when a miss occurs, an external cycle is initiated to obtain the operand from memory, and the data is loaded into the cache if the access is cachable. In the case of a misaligned operand that spans two cache entries, two long words are required from memory. Burst mode operation may also be initiated to fill an entire line of the data cache. Read accesses from the CPU address space and address translation table search accesses are not stored in the data cache.

The data cache on the MC68030 is a writethrough cache. When a hit occurs on a write cycle, the data is written both to the cache and to external memory (provided the MMU validates the access), regardless of the operand size and even if the cache is frozen. If the MMU determines that the access is invalid, the write is aborted, the corresponding entry is invalidated, and a bus error exception is taken. Since the write to the cache completes before the write to external memory, the cache contains the new value even if the external write terminates in a bus error. The value in the data cache might be used by another instruction before the external write cycle has completed, although this should not have any adverse consequences. Refer to **7.6 Bus Synchronization** for the details of bus synchronization.

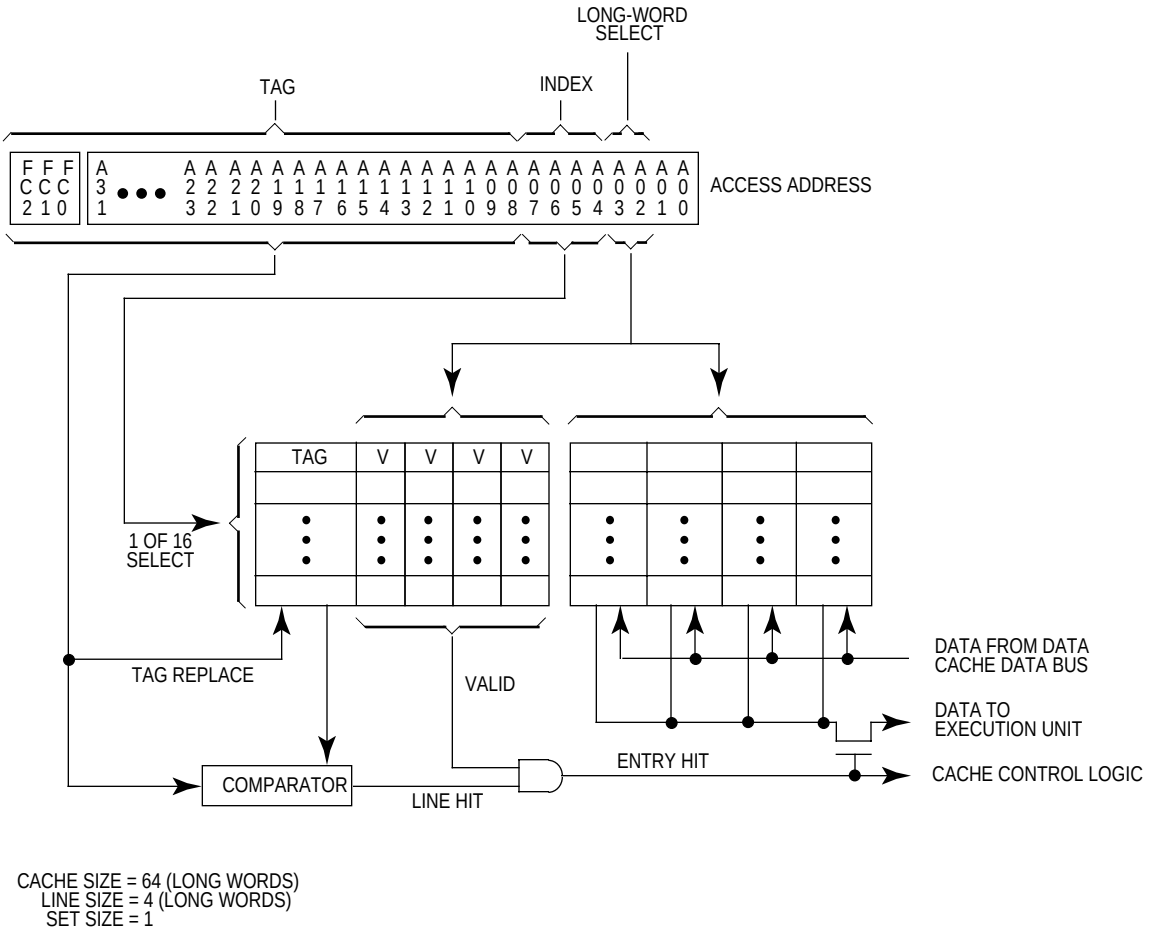


Figure 6-3. On-Chip Data Cache Organization

6.1.2.1 WRITE ALLOCATION. The supervisor program can configure the data cache for either of two types of allocation for data cache entries that miss on write cycles. The state of the write allocation (WA) bit in the cache control register specifies either no write allocation or write allocation with partial validation of the data entries in the cache on writes.

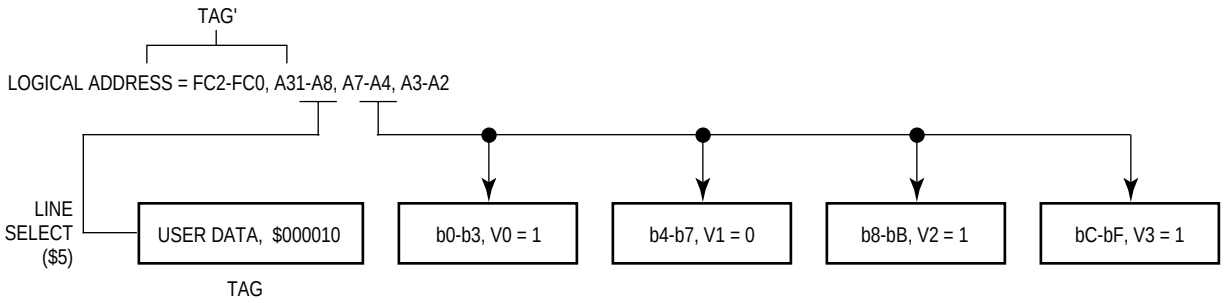
When no write allocation is selected (WA=0), write cycles that miss do not alter the data cache contents. In this mode, the processor does not replace entries in the cache during write operations. The cache is updated only during a write hit.

When write allocation is selected (WA=1), the processor always updates the data cache on cachable write cycles, but only validates an updated entry that hits or an entry that is updated with long-word data that is long-word aligned. When a tag miss occurs on a write of long-word data that is long-word aligned, the corresponding tag is replaced, and only the long word being written is marked as valid. The other three entries in the cache line are invalidated when a tag miss occurs on a misaligned long-word write or on a byte or word write, the data is not written in the cache, the tag is unaltered, and the valid bit(s) are cleared. Thus, an aligned long-word data write may replace a previously valid entry; whereas, a misaligned data write or a write of data that is not long word may invalidate a previously valid entry or entries.

Write allocation eliminates stale data that may reside in the cache because of either of two unique situations: multiple mapping of two or more logical addresses to one physical address within the same task or allowing the same physical location to be accessed by both supervisor and user mode cycles. Stale data conditions can arise when operating in the no-write-allocation mode and all the following conditions are satisfied:

- Multiple mapping (object aliasing) is allowed by the operating system.
- A read cycle loads a value for an “aliased” physical address into the data cache.
- A write cycle occurs, referencing the same aliased physical object as above but using a different logical address, causing a cache miss and no update to the cache (has the same page offset).
- The physical object is then read using the first alias, which provides stale data from the cache.

In this case, the data in the cache no longer matches that in physical memory and is stale. Since the write-allocation mode updates the cache during write cycles, the data in the cache remains consistent with physical memory. Note that when $\overline{\text{CIOUT}}$ is asserted, the data cache is completely ignored, even on write cycles operating in the write-allocation mode. Also note that since the $\overline{\text{CIIN}}$ signal is ignored on write cycles, cache entries may be created for noncachable data (when $\overline{\text{CIIN}}$ is asserted on a write) when operating in the write-allocation mode. Figure 6-4 shows the manner in which each mode operates in five different situations.



	NO WRITE ALLOCATE	WRITE ALLOCATE
EXAMPLE 1: USER WORD WRITE OF b2'-b3' to $00001052 (CACHE HIT, ALWAYS UPDATE CACHE AND MEMORY)	A) START EXTERNAL CYCLE B) b2-b3 ← b2'-b3'	A) START EXTERNAL CYCLE B) b2-b3 ← b2'-b3'
EXAMPLE 2: USER LONG-WORD WRITE OF b6'-b9' to $00001056 (TAG MATCH, LONG-WORD DATA, MISALIGNED, b6-b7 RESULT IN A CACHE MISS, b8-b9 RESULT IN A CACHE HIT)	A) START EXTERNAL CYCLE B) b8-b9 ← b8'-b9'	A) START EXTERNAL CYCLE B) b8-b9 ← b8'-b9'

Figure 6-4. No-Write-Allocation and Write-Allocation Mode Examples

6.1.2.2 READ-MODIFY-WRITE ACCESSES. The read portion of a read-modify-write cycle is always forced to miss in the data cache. However, if the system allows internal caching of read-modify-write cycle operands ($\overline{\text{CIOUT}}$ and $\overline{\text{CIIN}}$ both negated), the processor either uses the data read from memory to update a matching entry in the data cache or creates a new entry with the read data in the case of no matching entry. The write portion of a read-modify-write operation also updates a matching entry in the data cache. In the case of a cache miss on the write, the allocation of a new cache entry for the data being written is controlled by the WA bit. Table search accesses, however, are completely ignored by the data cache; it is never updated for a table search access.

6.1.3 Cache Filling

The bus controller can load either cache in either of two ways:

- Single entry mode
- Burst fill mode

In the single entry mode, the bus controller loads a single long-word entry of a cache line. In the burst fill mode, an entire line (four long words) can be filled. Refer to **Section 7 Bus Operation** for detailed information about the bus cycles required for both modes.

6.1.3.1 SINGLE ENTRY MODE. When a cachable access is initiated and a burst mode operation is not requested by the MC68030 or is not supported by external hardware, the bus controller transfers a single long word for the corresponding cache entry. An entire long word is required. If the port size of the responding device is smaller than 32 bits, the MC68030 executes all bus cycles necessary to fill the long word.

When a device cannot supply its entire port width of data, regardless of the size of the transfer, the responding device must consistently assert the cache inhibit input ($\overline{\text{CIIN}}$) signal. For example, a 32-bit port must always supply 32 bits, even for 8- and 16-bit transfers; a 16-bit port must supply 16 bits, even for 8-bit transfers. The MC68030 assumes that a 32-bit termination signal for the bus cycle indicates availability of 32 valid data bits, even if only 16 or 8 bits are requested. Similarly, the processor assumes that a 16-bit termination signal indicates that all 16 bits are valid. If the device cannot supply its full port width of data, it must assert $\overline{\text{CIIN}}$ for all bus cycles corresponding to a cache entry.

When a cachable read cycle provides data with both $\overline{\text{CIIN}}$ and $\overline{\text{BERR}}$ negated, the MC68030 attempts to fill the cache entry. Figure 6-5 shows the organization of a line of data in the caches. The notation b0, b1, b2, and so forth identifies the bytes within the line. For each entry in the line, a valid bit in the associated tag corresponds to a long-word entry to be loaded. Since a single valid bit applies to an entire long word, a single entry mode operation must provide a full 32 bits of data. Ports less than 32 bits wide require several read cycles for each entry.

Figure 6-5 shows an example of a byte data operand read cycle starting at byte address $03 from an 8-bit port. Provided the data item is cachable, this operation results in four bus cycles. The first cycle requested by the MC68030 reads a byte from address $03. The 8-bit $\overline{\text{DSACKx}}$ response causes the MC68030 to fetch the remainder of the long word starting at address $00. The bytes are latched in the following order: b3, b0, b1, and b2. Note that during cache loading operations, devices must indicate the same port size consistently throughout all cycles for that long-word entry in the cache.

Figure 6-6 shows the access of a byte data operand from a 16-bit port. This operation requires two read cycles. The first cycle requests the byte at address $03. If the device responds with a 16-bit $\overline{\text{DSACKx}}$ encoding, the word at address $02 (including the requested byte) is accepted by the MC68030. The second cycle requests the word at address $00. Since the device again responds with a 16-bit $\overline{\text{DSACKx}}$ encoding, the remaining two bytes of the long word are latched, and the cache entry is filled.

(UNABLE TO LOCATE ART)

Figure 6-5. Single Entry Mode Operation — 8-Bit Port

(UNABLE TO LOCATE ART)

Figure 6-6. Single Entry Mode Operation — 16-Bit Port

With a 32-bit port, the same operation is shown in Figure 6-7. Only one read cycle is required. All four bytes (including the requested byte) are latched during the cycle.

(UNABLE TO LOCATE ART)

Figure 6-7. Single Entry Mode Operation — 32-Bit Port

If a requested access is misaligned and spans two cache entries, the bus controller attempts to fill both associated long-word cache entries. An example of this is an operand request for a long word on an odd-word boundary. The MC68030 first fetches the initial byte(s) of the operand (residing in the first long word) and then requests the remaining bytes to fill that cache entry (if the port size is less than 32 bits) before it requests the remainder of the operand and corresponding long word to fill the second cache entry. If the port size is 32 bits, the processor performs two accesses, one for each cache entry.

Figure 6-8 shows a misaligned access of a long word at address $06 from an 8-bit port requiring eight bus cycles to complete. Reading this long-word operand requires eight read cycles, since accesses to all eight addresses return 8-bit port-size encodings. These cycles fetch the two cache entries that the requested long-word spans. The first cycle requests a long word at address $06 and accepts the first requested byte (b6). The subsequent transfers of the first long word are performed in the following order: b7, b4, b5. The remaining four read cycles transfer the four bytes of the second cache entry. The sequence of access for the entire operation is b6, b7, b4, b5, b8, b9, bA, and bB.

(UNABLE TO LOCATE ART)

Figure 6-8. Single Entry Mode Operation — Misaligned Long Word and 8-Bit Port

The next example, shown in Figure 6-9, is a read of a misaligned long-word operand from devices that return 16-bit $\overline{\text{DSACK}}$x encodings. The processor accepts the first portion of the operand, the word from address $06, and requests a word from address $04 to fill the cache entry. Next, the processor reads the word at address $08, the second portion of the operand, and stores it in the cache also. Finally, the processor accesses the word at $0A to fill the second long-word cache entry.

(UNABLE TO LOCATE ART)

Figure 6-9. Single Entry Mode Operation — Misaligned Long Word and 16-Bit Port

Two read cycles are required for a misaligned long-word operand transfer from devices that return 32-bit $\overline{\text{DSACK}}$x encodings. As shown in Figure 6-10, the first read cycle requests the long word at address $06 and latches the long word at address $04. The second read cycle requests and latches the long word corresponding to the second cache entry at address $08. Two read cycles are also required if $\overline{\text{STERM}}$ is used to indicate a 32-bit port instead of the 32-bit $\overline{\text{DSACK}}$x encoding.

(UNABLE TO LOCATE ART)

Figure 6-10. Single Entry Mode Operation — Misaligned Long Word and 32-Bit DSACKx Port

If all bytes of a long word are cachable, $\overline{\text{CIIN}}$ must be negated for all bus cycles required to fill the entry. If any byte is not cachable, $\overline{\text{CIIN}}$ must be asserted for all corresponding bus cycles. The assertion of the $\overline{\text{CIIN}}$ signal prevents the caches from being updated during read cycles. Write cycles (including the write portion of a read-modify-write cycle) ignore the assertion of the $\overline{\text{CIIN}}$ signal and may cause the data cache to be altered, depending on the state of the cache (whether or not the write cycle hits), the state of the WA bit in the $\overline{\text{CACR}}$, and the conditions indicated by the MMU.

The occurrence of a bus error while attempting to load a cache entry aborts the entry fill operation but does not necessarily cause a bus error exception. If the bus error occurs on a read cycle for a portion of the required operand (not the remaining bytes of the cache entry) to be loaded into the data cache, the processor immediately takes a bus error exception. If

the read cycle in error is made only to fill the data cache (the data is not part of the target operand), no exception occurs, but the corresponding entry is marked invalid. For the instruction cache, the processor marks the entry as invalid, but only takes an exception if the execution unit attempts to use the instruction word(s).

6.1.3.2 BURST MODE FILLING. Burst mode filling is enabled by bits in the cache control register. The data burst enable bit must be set to enable burst filling of the data cache. Similarly, the instruction burst enable bit must be set to enable burst filling of the instruction cache. When burst filling is enabled and the corresponding cache is enabled, the bus controller requests a burst mode fill operation in either of these cases:

- A read cycle for either the instruction or data cache misses due to the indexed tag not matching.
- A read cycle tag matches, but all long words in the line are invalid.

The bus controller requests a burst mode fill operation by asserting the cache burst request signal ($\overline{\text{CBREQ}}$). The responding device may sequentially supply one to four long words of cachable data, or it may assert the cache inhibit input signal ($\overline{\text{CIIN}}$) when the data in a long word is not cachable. If the responding device does not support the burst mode and it terminates cycles with $\overline{\text{STERM}}$, it should not acknowledge the request with the assertion of the cache burst acknowledge ($\overline{\text{CBACK}}$) signal. The MC68030 ignores the assertion of $\overline{\text{CBACK}}$ during cycles terminated with $\overline{\text{DSACK}}$x.

The cache burst request signal ($\overline{\text{CBREQ}}$) requests burst mode operation from the referenced external device. To operate in the burst mode, the device or external hardware must be able to increment the low-order address bits if required, and the current cycle must be a 32-bit synchronous transfer ($\overline{\text{STERM}}$ must be asserted) as described in **Section 7 Bus Operation**. The device must also assert $\overline{\text{CBACK}}$ (at the same time as $\overline{\text{STERM}}$) at the end of the cycle in which the MC68030 asserts $\overline{\text{CBREQ}}$. $\overline{\text{CBACK}}$ causes the processor to continue driving the address and bus control signals and to latch a new data value for the next cache entry at the completion of each subsequent cycle (as defined by $\overline{\text{STERM}}$), for a total of up to four cycles (until four long words have been read).

When a cache burst is initiated, the first cycle attempts to load the cache entry corresponding to the instruction word or data item explicitly requested by the execution unit. The subsequent cycles are for the subsequent entries in the cache line. In the case of a misaligned transfer when the operand spans two cache entries within a cache line, the first cycle corresponds to the cache entry containing the portion of the operand at the lower address.

Figure 6-11 illustrates the four cycles of a burst operation and shows that the second, third, and fourth cycles are run in burst mode. A distinction is made between the first cycle of a burst operation and the subsequent cycles because the first cycle is requested by the microsequencer and the burst fill cycles are requested by the bus controller. Therefore, when data from the first cycle is returned, it is immediately available for the execution unit (EU). However, data from the burst fill cycles is not available to the EU until the burst operation is complete. Since the microsequencer makes two separate requests for misaligned data operands, only the first portion of the misaligned operand returned during a

burst operation is available to the EU after the first cycle is complete. The microsequencer must wait for the burst operation to complete before requesting the second portion of the operand. Normally, the request for the second portion results in a data cache hit unless the second cycle of the burst operation terminates abnormally.

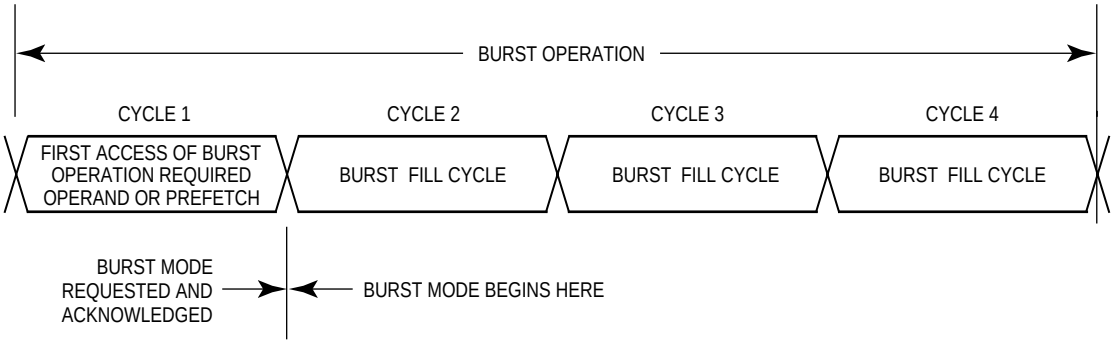


Figure 6-11. Burst Operation Cycles and Burst Mode

The bursting mechanism allows addresses to wrap around so that the entire four long words in the cache line can be filled in a single burst operation, regardless of the initial address and operand alignment. Depending on the structure of the external memory system, address bits A2 and A3 may have to be incremented externally to select the long words in the proper order for loading into the cache. The MC68030 holds the entire address bus constant for the duration of the burst cycle. Figure 6-12 shows an example of this address wraparound. The initial cycle is a long-word access from address $6. Because the responding device returns $\overline{\text{CBACK}}$ and $\overline{\text{STERM}}$ (signaling a 32-bit port), the entire long word at base address $04 is transferred. Since the initial address is $06 when CBREQ is asserted, the next entry to be burst filled into the cache should correspond to address $08, then $0C, and last, $00. This addressing is compatible with existing nibble-mode dynamic RAMs, and can be supported by page and static column modes with an external modulo 4 counter for A2 and A3.

(UNABLE TO LOCATE ART)

Figure 6-12. Burst Filling Wraparound Example

The MC68030 does not assert $\overline{\text{CBREQ}}$ during the first portion of a misaligned access if the remainder of the access does not correspond to the same cache line. Figure 6-13 shows an example in which the first portion of a misaligned access is at address $0F. With a 32-bit port, the first access corresponds to the cache entry at address $0C, which is filled using a single-entry load operation. The second access, at address $10 corresponding to the second cache line, requests a burst fill and the processor asserts $\overline{\text{CBREQ}}$. During this burst operation, long words $10, $14, $18, and $1C are all filled in that order.

(UNABLE TO LOCATE ART)

Figure 6-13. Deferred Burst Filling Example

The processor does not assert $\overline{\text{CBREQ}}$ if any of the following conditions exist:

- The appropriate cache is not enabled
- Burst filling for the cache is not enabled
- The cache freeze bit for the appropriate cache is set
- The current operation is the read portion of a read-modify-write operation
- The MMU has inhibited caching for the current page
- The cycle is for the first access of an operand that spans two cache lines (crosses a modulo 16 boundary)

Additionally, the assertion of $\overline{\text{CIIN}}$ and $\overline{\text{BERR}}$ and the premature negation of $\overline{\text{CBACK}}$ affect burst operation as described in the following paragraphs.

The assertion of $\overline{\text{CIIN}}$ during the first cycle of a burst operation causes the data to be latched by the processor, and if the requested operand is aligned (the entire operand is latched in the first cycle), the data is passed on to the instruction pipe or execution unit. However, the data is not loaded into its corresponding cache. In addition, the MC68030 negates $\overline{\text{CBREQ}}$, and the burst operation is aborted. If a portion of the requested operand remains to be read (due to misalignment), a second read cycle is initiated at the appropriate address with $\overline{\text{CBREQ}}$ negated.

The assertion of $\overline{\text{CIIN}}$ during the second, third, or fourth cycle of a burst operation prevents the data during that cycle from being loaded into the appropriate cache and causes $\overline{\text{CBREQ}}$ to negate, aborting the burst operation. However, if the data for the cycle contains part of the requested operand, the execution unit uses that data.

The premature negation of the $\overline{\text{CBACK}}$ signal during the burst operation causes the current cycle to complete normally, loading the data successfully transferred into the appropriate cache. However, the burst operation aborts and $\overline{\text{CBREQ}}$ negates.

A bus error occurring during a burst operation also causes the burst operation to abort. If the bus error occurs during the first cycle of a burst (i.e., before burst mode is entered), the data read from the bus is ignored, and the entire associated cache line is marked “invalid”. If the access is a data cycle, exception processing proceeds immediately. If the cycle is for an instruction fetch, a bus error exception is made pending. This bus error is processed only if the execution unit attempts to use either instruction word. Refer to **11.2.2 Instruction Pipe** for more information about pipeline operation.

For either cache, when a bus error occurs after the burst mode has been entered (that is, on the second cycle or later), the cache entry corresponding to that cycle is marked invalid, but the processor does not take an exception (the microsequencer has not yet requested the data). In the case of an instruction cache burst, the data from the aborted cycle is completely ignored. Pending instruction prefetches are still pending and are subsequently run by the processor. If the second cycle is for a portion of a misaligned data operand fetch and a bus error occurs, the processor terminates the burst operation and negates $\overline{\text{CBREQ}}$. Once the burst terminates, the microsequencer requests a read cycle for the second portion. Since the burst terminated abnormally for the second cycle of the burst, the data cache

results in a miss, and a second external cycle is required. If $\overline{\text{BERR}}$ is again asserted, the MC68030 then takes an exception.

On the initial access of a burst operation, a “retry“(indicated by the assertion of $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$) causes the processor to retry the bus cycle and assert $\overline{\text{CBREQ}}$ again. However, signaling a retry with simultaneous $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ during the second, third, or fourth cycle of a burst operation does not cause a retry operation, even if the requested operand is misaligned. Assertion of $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ during burst fill cycles of a burst operation causes independent bus error and halt operations. The processor remains halted until HALT is negated, and then handles the bus error as described in the previous paragraphs.

6.2 CACHE RESET

When a hardware reset of the processor occurs, all valid bits of both caches are cleared. The cache enable bits, burst enable bits, and the freeze bits in the cache control register (CACR) for both caches (refer to Figure 6-14) are also cleared, effectively disabling both caches. The WA bit in the CACR is also cleared.

6.3 CACHE CONTROL

Only the MC68030 cache control circuitry can directly access the cache arrays, but the supervisor program can set bits in the CACR to exercise control over cache operations. The supervisor also has access to the cache address register (CAAR), which contains the address for a cache entry to be cleared.

6.3.1 Cache Control Register

The CACR, shown in Figure 6-14, is a 32-bit register that can be written or read by the MOVEC instruction or indirectly modified by a reset. Five of the bits (4-0) control the instruction cache; six other bits (13-8) control the data cache. Each cache is controlled independently of the other, although a similar operation can be performed for both caches by a single MOVEC instruction. For example, loading a long word in which bits 3 and 11 are set into the CACR clears both caches. Bits 31-14 and 7-5 are reserved for Motorola definition. They are currently read as zeros and are ignored when written. For future compatibility, writes should not set these bits.

31 … 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00000000000000000	WA	DBE	CD	CED	FD	ED	0	0	0	IBE	CI	CEI	FI	EI

WA = Write Allocate
DBE = Data Burst Enable
CD = Clear Data Cache
CED = Clear Entry in Data Cache
FD = Freeze Data Cache
ED = Freeze Data Cache
IBE = Instruction Burst Enable
CI = Clear Instruction Cache
CEI = Clear Entry in Instruction Cache
FI = Freeze Instruction Cache
EI = Enable Instruction Cache

Figure 6-14. Cache Control Register

6.3.1.1 WRITE ALLOCATE. Bit 13, the WA bit, is set to select the write-allocation mode (refer to **6.1.2.1 Write Allocation**) for write cycles. Clearing this bit selects the no-write-allocation mode. A reset operation clears this bit. The supervisor should set this bit when it shares data with the user task or when any task maps multiple logical addresses to one physical address. If the data cache is disabled or frozen, the WA bit is ignored.

6.3.1.2 DATA BURST ENABLE. Bit 12, the DBE bit, is set to enable burst filling of the data cache. Operating systems and other software set this bit when burst filling of the data cache is desired. A reset operation clears the DBE bit.

6.3.1.3 CLEAR DATA CACHE. Bit 11, the CD bit, is set to clear all entries in the data cache. Operating systems and other software set this bit to clear data from the cache prior to a context switch. The processor clears all valid bits in the data cache at the time a MOVEC instruction loads a one into the CD bit of the CACR. The CD bit is always read as a zero.

6.3.1.4 CLEAR ENTRY IN DATA CACHE. Bit 10, the CED bit, is set to clear an entry in the data cache. The index field of the CAAR (see Figure 6-15) corresponding to the index and long-word select portion of an address specifies the entry to be cleared. The processor clears only the specified long word by clearing the valid bit for the entry at the time a MOVEC instruction loads a one into the CED bit of the CACR, regardless of the states of the ED and FD bits. The CED bit is always read as a zero.

6.3.1.5 FREEZE DATA CACHE. Bit 9, the FD bit, is set to freeze the data cache. When the FD bit is set and a miss occurs during a read or write of the data cache, the indexed entry is not replaced. However, write cycles that hit in the data cache cause the entry to be updated even when the cache is frozen. When the FD bit is clear, a miss in the data cache during a read cycle causes the entry (or line) to be filled, and the filling of entries on writes that miss are then controlled by the WA bit. A reset operation clears the FD bit.

6.3.1.6 ENABLE DATA CACHE. Bit 8, the ED bit, is set to enable the data cache. When it is cleared, the data cache is disabled. A reset operation clears the ED bit. The supervisor normally enables the data cache, but it can clear ED for system debugging or emulation, as required. Disabling the data cache does not flush the entries. If it is enabled again, the previously valid entries remain valid and can be used.

6.3.1.7 INSTRUCTION BURST ENABLE. Bit 4, the IBE bit, is set to enable burst filling of the instruction cache. Operating systems and other software set this bit when burst filling of the instruction cache is desired. A reset operation clears the IBE bit.

6.3.1.8 CLEAR INSTRUCTION CACHE. Bit 3, the CI bit, is set to clear all entries in the instruction cache. Operating systems and other software set this bit to clear instructions from the cache prior to a context switch. The processor clears all valid bits in the instruction cache at the time a MOVEC instruction loads a one into the CI bit of the CACR. The CI bit is always read as a zero.

6.3.1.9 CLEAR ENTRY IN INSTRUCTION CACHE. Bit 2, the CEI bit, is set to clear an entry in the instruction cache. The index field of the CAAR (see Figure 6-15) corresponding to the index and long-word select portion of an address specifies the entry to be cleared. The processor clears only the specified long word by clearing the valid bit for the entry at the time a MOVEC instruction loads a one into the CEI bit of the CACR, regardless of the states of the EI and FI bits. The CEI bit is always read as a zero.

6.3.1.10 FREEZE INSTRUCTION CACHE. Bit 1, the FI bit, is set to freeze the instruction cache. When the FI bit is set and a miss occurs in the instruction cache, the entry (or line) is not replaced. When the FI bit is cleared to zero, a miss in the instruction cache causes the entry (or line) to be filled. A reset operation clears the FI bit.

6.3.1.11 ENABLE INSTRUCTION CACHE. Bit 0, the EI bit, is set to enable the instruction cache. When it is cleared, the instruction cache is disabled. A reset operation clears the EI bit. The supervisor normally enables the instruction cache, but it can clear EI for system debugging or emulation, as required. Disabling the instruction cache does not flush the entries. If it is enabled again, the previously valid entries remain valid and may be used.

6.3.2 Cache Address Register

The CAAR is a 32-bit register shown in Figure 6-15. The index field (bits 7-2) contains the address for the “clear cache entry” operations. The bits of this field correspond to bits 7-2 of addresses; they specify the index and a long word of a cache line. Although only the index field is used currently, all 32 bits of the register are implemented and are reserved for use by Motorola.

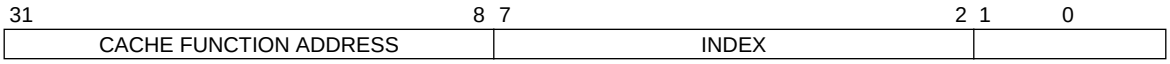


Figure 6-15. Cache Address Register

SECTION 7
BUS OPERATION

This section provides a functional description of the bus, the signals that control it, and the bus cycles provided for data transfer operations. It also describes the error and halt conditions, bus arbitration, and the reset operation. Operation of the bus is the same whether the processor or an external device is the bus master; the names and descriptions of bus cycles are from the point of view of the bus master. For exact timing specifications, refer to **Section 13 Electrical Characteristics**.

The MC68030 architecture supports byte, word, and long-word operands, allowing access to 8-, 16-, and 32-bit data ports through the use of asynchronous cycles controlled by the data transfer and size acknowledge inputs ($\overline{\text{DSACK0}}$ and $\overline{\text{DSACK1}}$).

Synchronous bus cycles controlled by the synchronous termination signal ($\overline{\text{STERM}}$) can only be used to transfer data to and from 32-bit ports.

The MC68030 allows byte, word, and long-word operands to be located in memory on any byte boundary. For a misaligned transfer, more than one bus cycle may be required to complete the transfer, regardless of port size. For a port less than 32 bits wide, multiple bus cycles may be required for an operand transfer due to either misalignment or a port width smaller than the operand size. Instruction words and their associated extension words must be aligned on word boundaries. The user should be aware that misalignment of word or long-word operands can cause the MC68030 to perform multiple bus cycles for the operand transfer; therefore, processor performance is optimized if word and long-word memory operands are aligned on word or long-word boundaries, respectively.

7.1 BUS TRANSFER SIGNALS

The bus transfers information between the MC68030 and an external memory, coprocessor, or peripheral device. External devices can accept or provide 8 bits, 16 bits, or 32 bits in parallel and must follow the handshake protocol described in this section. The maximum number of bits accepted or provided during a bus transfer is defined as the port width. The MC68030 contains an address bus that specifies the address for the transfer and a data bus that transfers the data. Control signals indicate the beginning of the cycle, the address space and the size of the transfer, and the type of cycle. The selected device then controls the length of the cycle with the signal(s) used to terminate the cycle. Strobe signals, one for the address bus and another for the data bus, indicate the validity of the address and provide timing information for the data.

The bus can operate in an asynchronous mode identical to the MC68020 bus for any port width. The bus and control input signals used for asynchronous operation are internally synchronized to the MC68030 clock, introducing a delay. This delay is the time period required for the MC68030 to sample an asynchronous input signal, synchronize the input to the internal clocks of the processor, and determine whether it is high or low. Figure 7-1 shows the relationship between the clock signal and the associated internal signal of a typical asynchronous input.

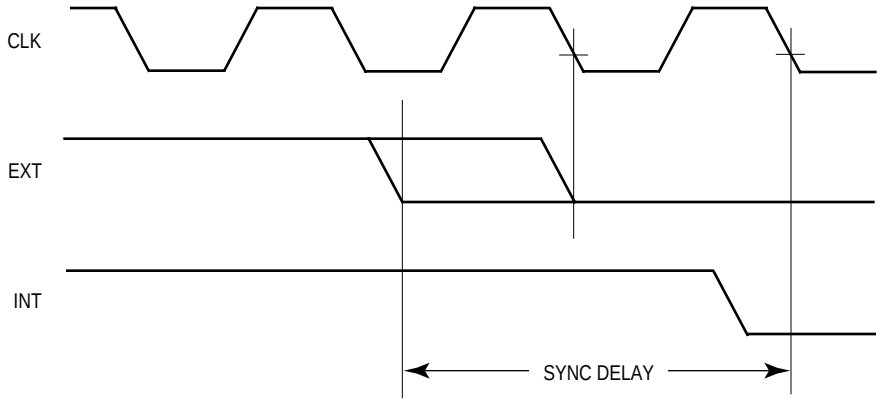


Figure 7-1. Relationship between External and Internal Signals

Furthermore, for all asynchronous inputs, the processor latches the level of the input during a sample window around the falling edge of the clock signal. This window is illustrated in Figure 7-2. To ensure that an input signal is recognized on a specific falling edge of the clock, that input must be stable during the sample window. If an input makes a transition during the window time period, the level recognized by the processor is not predictable; however, the processor always resolves the latched level to either a logic high or low before using it. In addition to meeting input setup and hold times for deterministic operation, all input signals must obey the protocols described in this section.

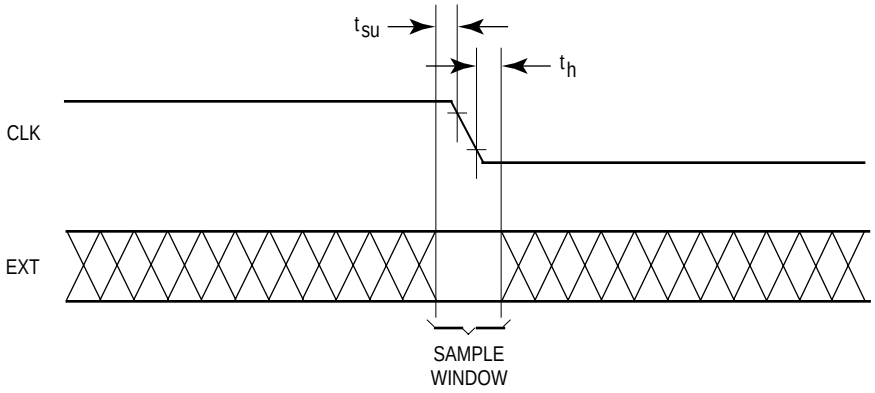


Figure 7-2. Asynchronous Input Sample Window

A device with a 32-bit port size can also provide a synchronous mode transfer. In synchronous operation, input signals are externally synchronized to the processor clock, and the synchronizing delay is not incurred.

Synchronous inputs ($\overline{\text{STERM}}$, $\overline{\text{CBACK}}$, and $\overline{\text{CIIN}}$) must remain stable during a sample window for all rising edges of the clock during a bus cycle (i.e., while address strobe ($\overline{\text{AS}}$) is asserted), regardless of when the signals are asserted or negated, to ensure proper operation. This sample window is defined by the synchronous input setup and hold times (see MC68030EC/D, *MC68030 Electrical Specifications*).

7.1.1 Bus Control Signals

The external cycle start ($\overline{\text{ECS}}$) signal is the earliest indication that the processor is initiating a bus cycle. The MC68030 initiates a bus cycle by driving the address, size, function code, read/write, and cache inhibit-out outputs and by asserting $\overline{\text{ECS}}$. However, if the processor finds the required program or data item in an on-chip cache, if a miss occurs in the address translation cache (ATC) of the memory management unit (MMU), or if the MMU finds a fault with the access, the processor aborts the cycle before asserting $\overline{\text{AS}}$. $\overline{\text{ECS}}$ can be used to initiate various timing sequences that are eventually qualified with $\overline{\text{AS}}$. Qualification with AS may be required since, in the case of an internal cache hit, an ATC miss, or an MMU fault, a bus cycle may be aborted after $\overline{\text{ECS}}$ has been asserted. The assertion of $\overline{\text{AS}}$ ensures that the cycle has not been aborted by these internal conditions.

During the first external bus cycle of an operand transfer, the operand cycle start ($\overline{\text{OCS}}$) signal is asserted with $\overline{\text{ECS}}$. When several bus cycles are required to transfer the entire operand, $\overline{\text{OCS}}$ is asserted only at the beginning of the first external bus cycle. With respect to $\overline{\text{OCS}}$, an "operand" is any entity required by the execution unit, whether a program or data item.

The function code signals (FC0–FC2) are also driven at the beginning of a bus cycle. These three signals select one of eight address spaces (refer to Table 4-1) to which the address applies. Five address spaces are presently defined. Of the remaining three, one is reserved

for user definition and two are reserved by Motorola for future use. The function code signals are valid while AS is asserted.

At the beginning of a bus cycle, the size signals (SIZ0 and SIZ1) are driven along with $\overline{\text{ECS}}$ and the FC0–FC2. SIZ0 and SIZ1 indicate the number of bytes remaining to be transferred during an operand cycle (consisting of one or more bus cycles) or during a cache fill operation from a device with a port size that is less than 32 bits. Table 7-2 shows the encoding of SIZ0 and SIZ1. These signals are valid while $\overline{\text{AS}}$ is asserted.

The read/write (R/$\overline{\text{W}}$) signal determines the direction of the transfer during a bus cycle. This signal changes state, when required, at the beginning of a bus cycle and is valid while $\overline{\text{AS}}$ is asserted. R/$\overline{\text{W}}$ only transitions when a write cycle is preceded by a read cycle or vice versa. The signal may remain low for two consecutive write cycles.

The read-modify-write cycle signal ($\overline{\text{RMC}}$) is asserted at the beginning of the first bus cycle of a read-modify-write operation and remains asserted until completion of the final bus cycle of the operation. The $\overline{\text{RMC}}$ signal is guaranteed to be negated before the end of state 0 for a bus cycle following a read-modify-write operation.

7.1.2 Address Bus

The address bus signals (A0–A31) define the address of the byte (or the most significant byte) to be transferred during a bus cycle. The processor places the address on the bus at the beginning of a bus cycle. The address is valid while $\overline{\text{AS}}$ is asserted.

7.1.3 Address Strobe

$\overline{\text{AS}}$ is a timing signal that indicates the validity of an address on the address bus and of many control signals. It is asserted one-half clock after the beginning of a bus cycle.

7.1.4 Data Bus

The data bus signals (D0–D31) comprise a bidirectional, nonmultiplexed parallel bus that contains the data being transferred to or from the processor. A read or write operation may transfer 8, 16, 24, or 32 bits of data (one, two, three, or four bytes) in one bus cycle. During a read cycle, the data is latched by the processor on the last falling edge of the clock for that bus cycle. For a write cycle, all 32 bits of the data bus are driven, regardless of the port width or operand size. The processor places the data on the data bus one-half clock cycle after $\overline{AS}$ is asserted in a write cycle.

7.1.5 Data Strobe

The data strobe ($\overline{DS}$) is a timing signal that applies to the data bus. For a read cycle, the processor asserts $\overline{DS}$ to signal the external device to place data on the bus. It is asserted at the same time as $\overline{AS}$ during a read cycle. For a write cycle, $\overline{DS}$ signals to the external device that the data to be written is valid on the bus. The processor asserts $\overline{DS}$ one full clock cycle after the assertion of $\overline{AS}$ during a write cycle.

7.1.6 Data Buffer Enable

The data buffer enable signal ($\overline{DBEN}$) can be used to enable external data buffers while data is present on the data bus. During a read operation, $\overline{DBEN}$ is asserted one clock cycle after the beginning of the bus cycle and is negated as $\overline{DS}$ is negated. In a write operation, $\overline{DBEN}$ is asserted at the time $\overline{AS}$ is asserted and is held active for the duration of the cycle. In a synchronous system supporting two-clock bus cycles, $\overline{DBEN}$ timing may prevent its use.

7.1.7 Bus Cycle Termination Signals

During asynchronous bus cycles, external devices assert the data transfer and size acknowledge signals ($\overline{DSACK0}$ and/or $\overline{DSACK1}$) as part of the bus protocol. During a read cycle, the assertion of $\overline{DSACKx}$ signals the processor to terminate the bus cycle and to latch the data. During a write cycle, the assertion of $\overline{DSACKx}$ indicates that the external device has successfully stored the data and that the cycle may terminate. These signals also indicate to the processor the size of the port for the bus cycle just completed, as shown in Table 7-1. Refer to **7.3.1 Asynchronous Read Cycle** for timing relationships of $\overline{DSACK0}$ and $\overline{DSACK1}$.

For synchronous bus cycles, external devices assert the synchronous termination signal ($\overline{\text{STERM}}$) as part of the bus protocol. During a read cycle, the assertion of $\overline{\text{STERM}}$ causes the processor to latch the data. During a write cycle, it indicates that the external device has successfully stored the data. In either case, it terminates the cycle and indicates that the transfer was made to a 32-bit port. Refer to **7.3.2 Asynchronous Write Cycle** for timing relationships of $\overline{\text{STERM}}$.

The bus error ($\overline{\text{BERR}}$) signal is also a bus cycle termination indicator and can be used in the absence of $\overline{\text{DSACKx}}$ or $\overline{\text{STERM}}$ to indicate a bus error condition. It can also be asserted in conjunction with $\overline{\text{DSACKx}}$ or $\overline{\text{STERM}}$ to indicate a bus error condition, provided it meets the appropriate timing described in this section and in MC68030EC/D, *MC68030 Electrical Specifications*. Additionally, the $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ signals can be asserted together to indicate a retry termination. Again, the $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ signals can be asserted simultaneously in lieu of or in conjunction with the $\overline{\text{DSACKx}}$ or $\overline{\text{STERM}}$ signals.

Finally, the autovector ($\overline{\text{AVEC}}$) signal can be used to terminate interrupt acknowledge cycles, indicating that the MC68030 should internally generate a vector number to locate an interrupt handler routine. $\overline{\text{AVEC}}$ is ignored during all other bus cycles.

7.2 DATA TRANSFER MECHANISM

The MC68030 architecture supports byte, word, and long-word operands allowing access to 8-, 16-, and 32-bit data ports through the use of asynchronous cycles controlled by $\overline{\text{DSACK0}}$ and $\overline{\text{DSACK1}}$. It also supports synchronous bus cycles to and from 32-bit ports, terminated by $\overline{\text{STERM}}$. Byte, word, and long-word operands can be located on any byte boundary, but misaligned transfers may require additional bus cycles, regardless of port size.

When the processor requests a burst mode fill operation, it asserts the cache burst request ($\overline{\text{CBREQ}}$) signal to attempt to fill four entries within a line in one of the on-chip caches. This mode is compatible with nibble, static column, or page mode dynamic RAMs. The burst fill operation uses synchronous bus cycles, each terminated by $\overline{\text{STERM}}$, to fetch as many as four long words.

7.2.1 Dynamic Bus Sizing

The MC68030 dynamically interprets the port size of the addressed device during each bus cycle, allowing operand transfers to or from 8-, 16-, and 32-bit ports. During an asynchronous operand transfer cycle, the slave device signals its port size (byte, word, or long word) and indicates completion of the bus cycle to the processor through the use of the $\overline{\text{DSACKx}}$ inputs. Refer to Table 7-1 for $\overline{\text{DSACKx}}$ encodings and assertion results.

Table 7-1. $\overline{\text{DSACK}}$ Codes and Results

$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$	Result
H	H	Insert Wait States in Current Bus Cycle
H	L	Complete Cycle — Data Bus Port Size is 8 Bits
L	H	Complete Cycle — Data Bus Port Size is 16 Bits
L	L	Complete Cycle — Data Bus Port Size is 32 Bits

For example, if the processor is executing an instruction that reads a long-word operand from a long-word aligned address, it attempts to read 32 bits during the first bus cycle. (Refer to **7.2.2 Misaligned Operands** for the case of a word or byte address.) If the port responds that it is 32 bits wide, the MC68030 latches all 32 bits of data and continues with the next operation. If the port responds that it is 16 bits wide, the MC68030 latches the 16 bits of valid data and runs another bus cycle to obtain the other 16 bits. The operation for an 8-bit port is similar, but requires four read cycles. The addressed device uses the $\overline{\text{DSACKx}}$ signals to indicate the port width. For instance, a 32-bit device *always* returns $\overline{\text{DSACKx}}$ for a 32-bit port (regardless of whether the bus cycle is a byte, word, or long-word operation).

Dynamic bus sizing requires that the portion of the data bus used for a transfer to or from a particular port size be fixed. A 32-bit port must reside on data bus bits 0–31, a 16-bit port must reside on data bus bits 16-32, and an 8-bit port must reside on data bus bits 24-31. This requirement minimizes the number of bus cycles needed to transfer data to 8- and 16-bit ports and ensures that the MC68030 correctly transfers valid data. The MC68030 always attempts to transfer the maximum amount of data on all bus cycles; for a long-word operation, it always assumes that the port is 32 bit wide when beginning the bus cycle.

The bytes of operands are designated as shown in Figure 7-3. The most significant byte of a long-word operand is OP0, and OP3 is the least significant byte. The two bytes of a word-length operand are OP2 (most significant) and OP3. The single byte of a byte-length operand is OP3. These designations are used in the figures and descriptions that follow.

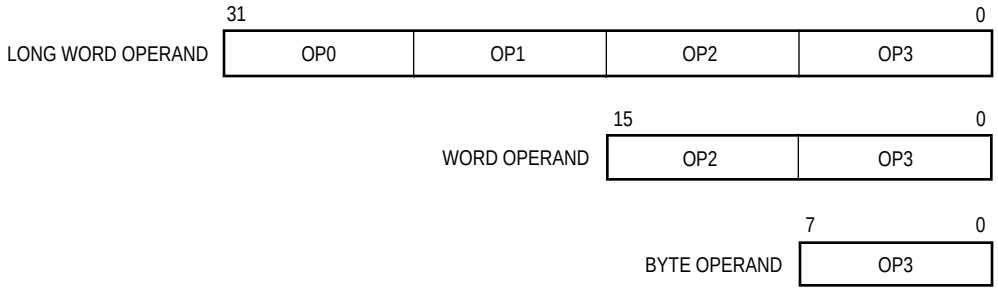


Figure 7-3. Internal Operand Representation

Figure 7-4 shows the required organization of data ports on the MC68030 bus for 8, 16, and 32-bit devices. The four bytes shown in Figure 7-4 are connected through the internal data bus and data multiplexer to the external data bus. This path is the means through which the MC68030 supports dynamic bus sizing and operand misalignment. Refer to **7.2.2 Misaligned Operands** for the definition of misaligned operand. The data multiplexer establishes the necessary connections for different combinations of address and data sizes.

The multiplexer takes the four bytes of the 32-bit bus and routes them to their required positions. For example, OP0 can be routed to D24–D31, as would be the normal case, or it can be routed to any other byte position to support a misaligned transfer. The same is true for any of the operand bytes. The positioning of bytes is determined by the size (SIZ0 and SIZ1) and address (A0 and A1) outputs.

The SIZ0 and SIZ1 outputs indicate the remaining number of bytes to be transferred during the current bus cycle, as shown in Table 7-2.

The number of bytes transferred during a write or noncachable read bus cycle is equal to or less than the size indicated by the SIZ0 and SIZ1 outputs, depending on port width and operand alignment. For example, during the first bus cycle of a long-word transfer to a word port, the size outputs indicate that four bytes are to be transferred, although only two bytes are moved on that bus cycle. Cachable read cycles must always transfer the number of bytes indicated by the port size.

A0 and A1 also affect operation of the data multiplexer. During an operand transfer, A2–A31 indicate the long-word base address of that portion of the operand to be accessed; A0 and A1 indicate the byte offset from the base. Table 7-3 shows the encodings of A0 and A1 and the corresponding byte offsets from the long-word base.

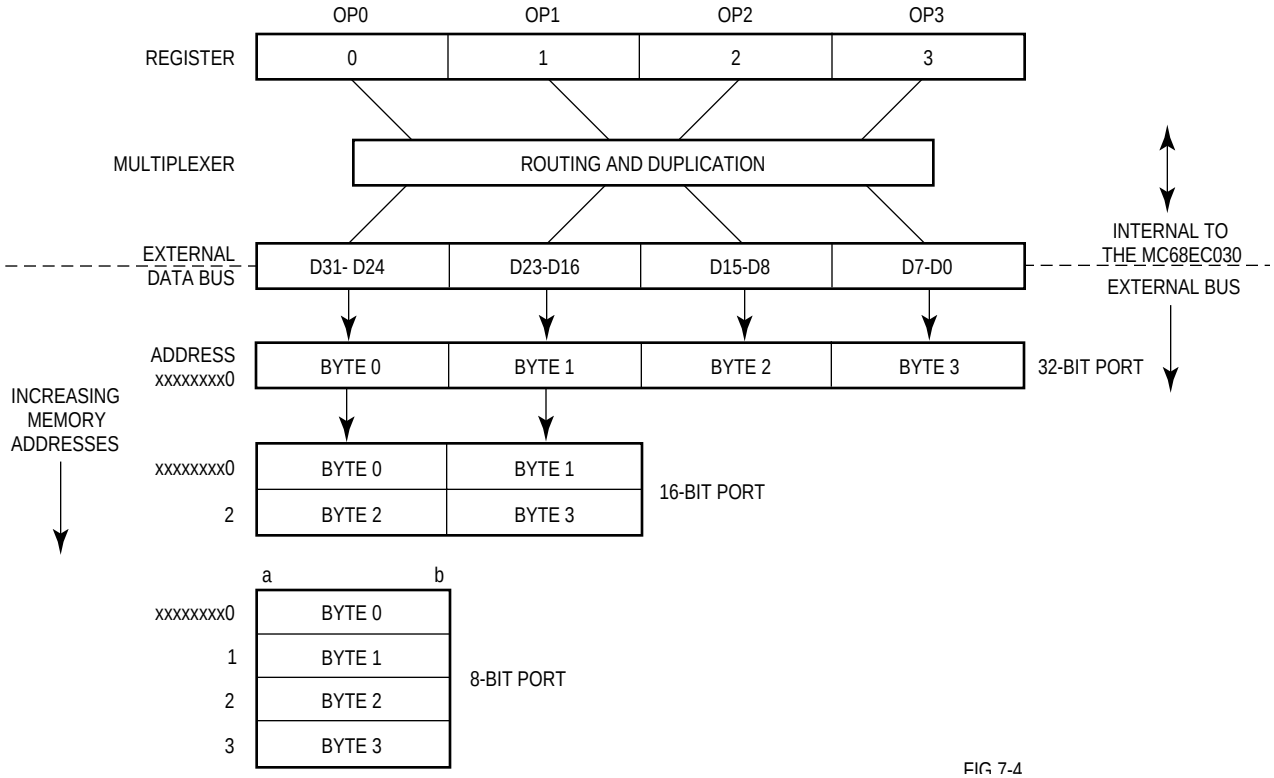


Figure 7-4. MC68030 Interface to Various Port Sizes

Table 7-4 lists the bytes required on the data bus for read cycles that are cachable. The entries shown as OPn are portions of the requested operand that are read or written during that bus cycle and are defined by SIZ0, SIZ1, A0, and A1 for the bus cycle. The PRn and the Nn bytes correspond to the previous and next bytes in memory, respectively, that must be valid on the data bus for the specified port size (long word or word) so that the internal caches operate correctly. (For cachable accesses, the MC68030 assumes that all portions of the data bus for a given port size are valid.) This same table applies to noncachable read cycles except that the bytes labeled PRn and Nn are not required and can be replaced by “don't cares”.

Table 7-2. Size Signal Encoding

SIZ1	SIZ0	Size
0	1	Byte
1	0	Word
1	1	3 Bytes
0	0	Long Word

Table 7-3. Address Offset Encodings

A1	A0	Offset
0	0	+0 Bytes
0	1	+1 Byte
1	0	+2 Bytes
1	1	+3 Bytes

Table 7-4. Data Bus Requirements for Read Cycles.

(Table did not make it over in the conversion from Word)

Table 7-5 lists the combinations of SIZ0, SIZ1, A0, and A1 and the corresponding pattern of the data transfer for write cycles from the internal multiplexer of the MC68030 to the external data bus.

Figure 7-5 shows the transfer of a long-word operand to a word port. In the first bus cycle, the MC68030 places the four operand bytes on the external bus. Since the address is long-word aligned in this example, the multiplexer follows the pattern in the entry of Table 7-5 corresponding to SIZ0_SIZ1_A0_A1=0000. The port latches the data on bits D16–D31 of the data bus, asserts $\overline{\text{DSACK1}}$ ($\overline{\text{DSACK0}}$ remains negated), and the processor terminates the bus cycle. It then starts a new bus cycle with SIZ0_SIZ1_A0_A1=1010 to transfer the remaining 16 bits. SIZ0 and SIZ1 indicate that a word remains to be transferred; A0 and A1 indicate that the word corresponds to an offset of two from the base address. The multiplexer follows the pattern corresponding to this configuration of the size and address signals and places the two least significant bytes of the long word on the word portion of the bus (D16–D31). The bus cycle transfers the remaining bytes to the word-size port. Figure 7-6 shows the timing of the bus transfer signals for this operation.

Table 7-5. MC68030 Internal to External Data Bus.

(Table did not make it over in the conversion from Word)

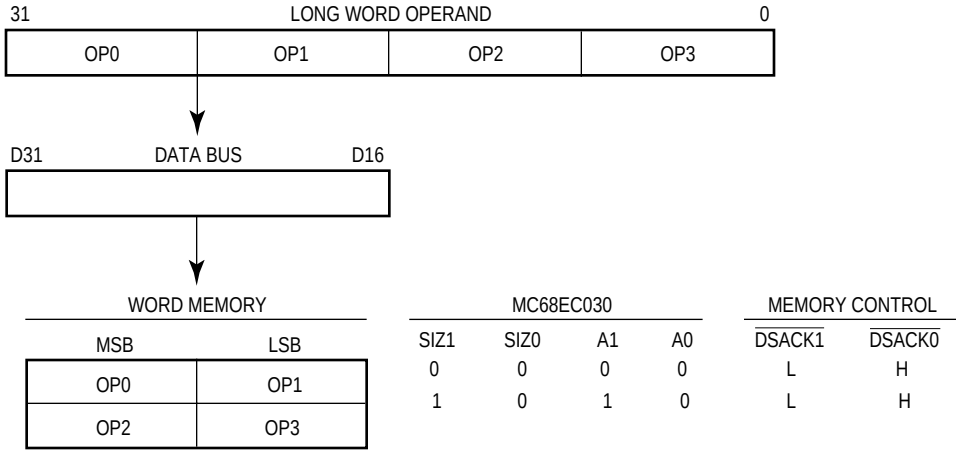


Figure 7-5. Example of Long-Word Transfer to Word Port

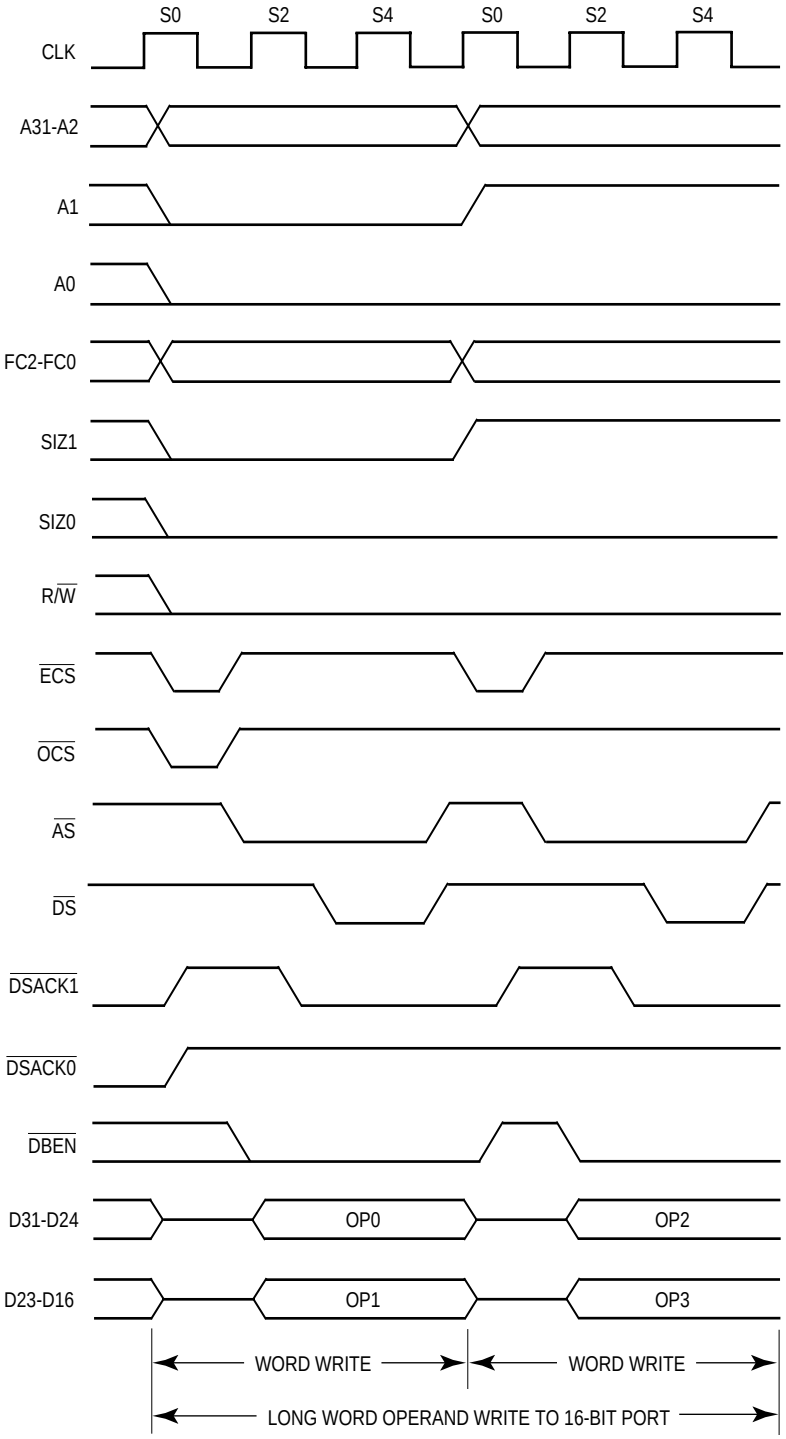


Figure 7-6. Long-Word Operand Write Timing (16-Bit Data Port)

Figure 7-7 shows a word transfer to an 8-bit bus port. Like the preceding example, this example requires two bus cycles. Each bus cycle transfers a single byte. The size signals for the first cycle specify two bytes; for the second cycle, one byte. Figure 7-8 shows the associated bus transfer signal timing.

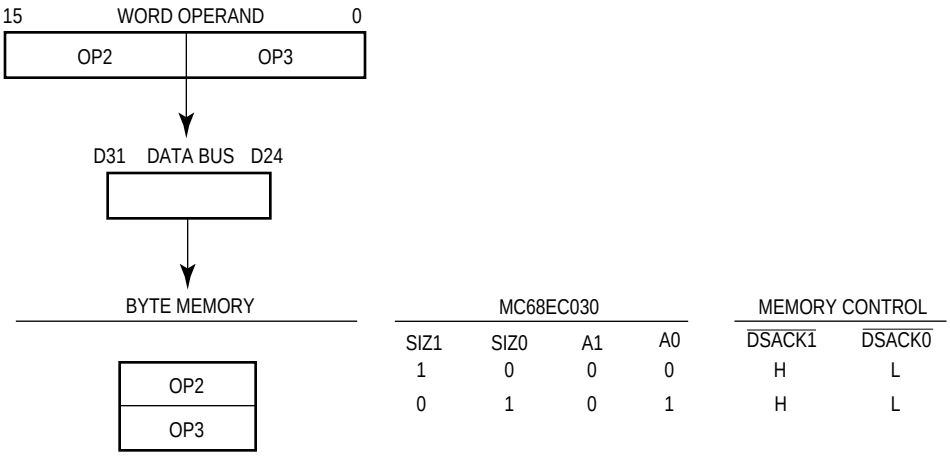


Figure 7-7. Example of Word Transfer to Byte Port

7.2.2 Misaligned Operands

Since operands may reside at any byte boundaries, they may be misaligned. A byte operand is properly aligned at any address; a word operand is misaligned at an odd address; a long word is misaligned at an address that is not evenly divisible by four. The MC68000, MC68008, and MC68010 implementations allow long-word transfers on odd-word boundaries but force exceptions if word or long-word operand transfers are attempted at odd-byte addresses. Although the MC68030 does not enforce any alignment restrictions for data operands (including PC relative data addresses), some performance degradation occurs when additional bus cycles are required for long-word or word operands that are misaligned. For maximum performance, data items should be aligned on their natural boundaries. All instruction words and extension words must reside on word boundaries. Attempting to prefetch an instruction word at an odd address causes an address error exception.

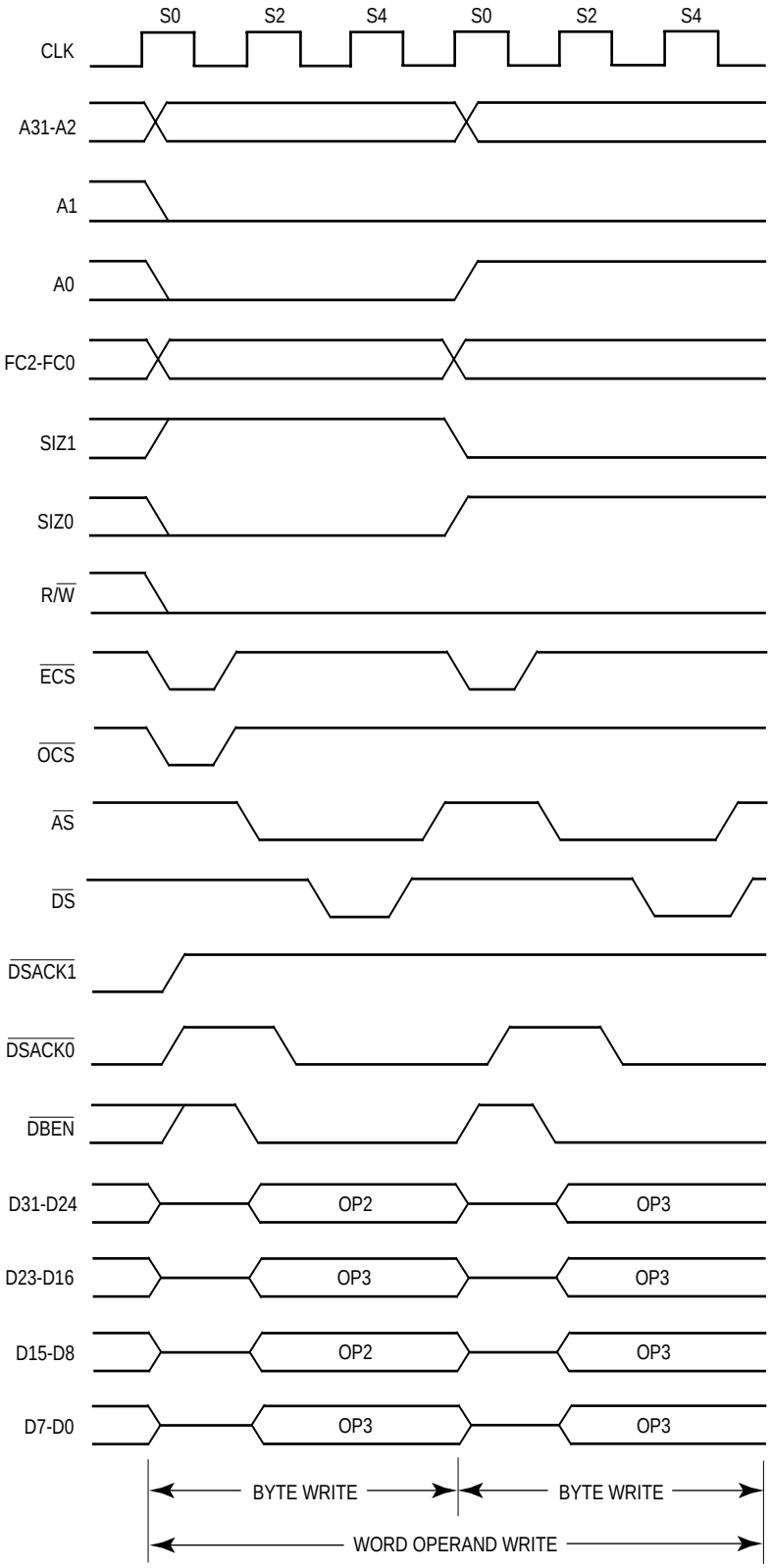


Figure 7-8. Word Operand Write Timing (8-Bit Data Port)

Figure 7-9 shows the transfer of a long-word operand to an odd address in word-organized memory, which requires three bus cycles. For the first cycle, the size signals specify a long-word transfer, and the address offset (A2:A0) is 001. Since the port width is 16 bits, only the first byte of the long word is transferred. The slave device latches the byte and acknowledges the data transfer, indicating that the port is 16 bits wide. When the processor starts the second cycle, the size signals specify that three bytes remain to be transferred with an address offset (A2:A0) of 010. The next two bytes are transferred during this cycle. The processor then initiates the third cycle, with the size signals indicating one byte remaining to be transferred. The address offset (A2:A0) is now 100; the port latches the final byte; and the operation is complete. Figure 7-10 shows the associated bus transfer signal timing.

Figure 7-11 shows the equivalent operation for a cachable data read cycle.

Figures 7-12 and 7-13 show a word transfer to an odd address in word-organized memory. This example is similar to the one shown in Figures 7-9 and 7-10 except that the operand is word sized and the transfer requires only two bus cycles.

Figure 7-14 shows the equivalent operation for a cachable data read cycle.

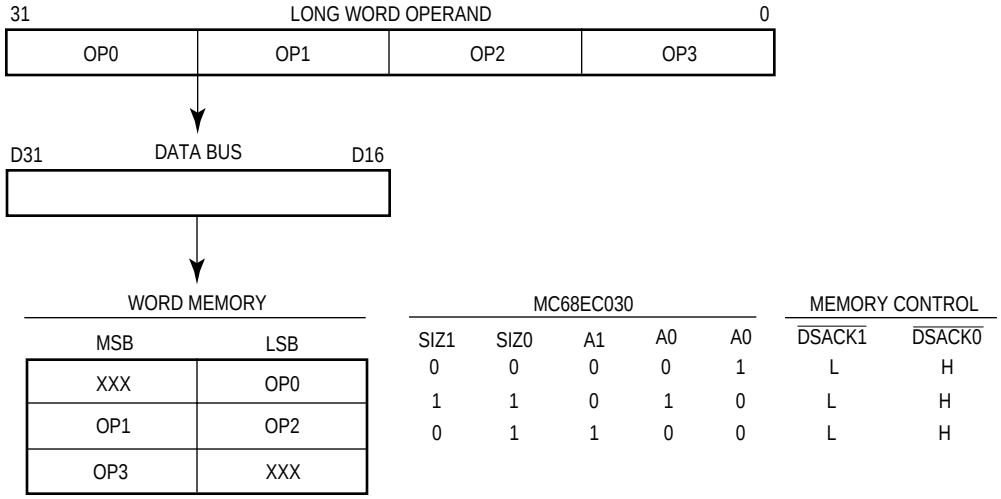


Figure 7-9. Misaligned Long-Word Transfer to Word Port Example

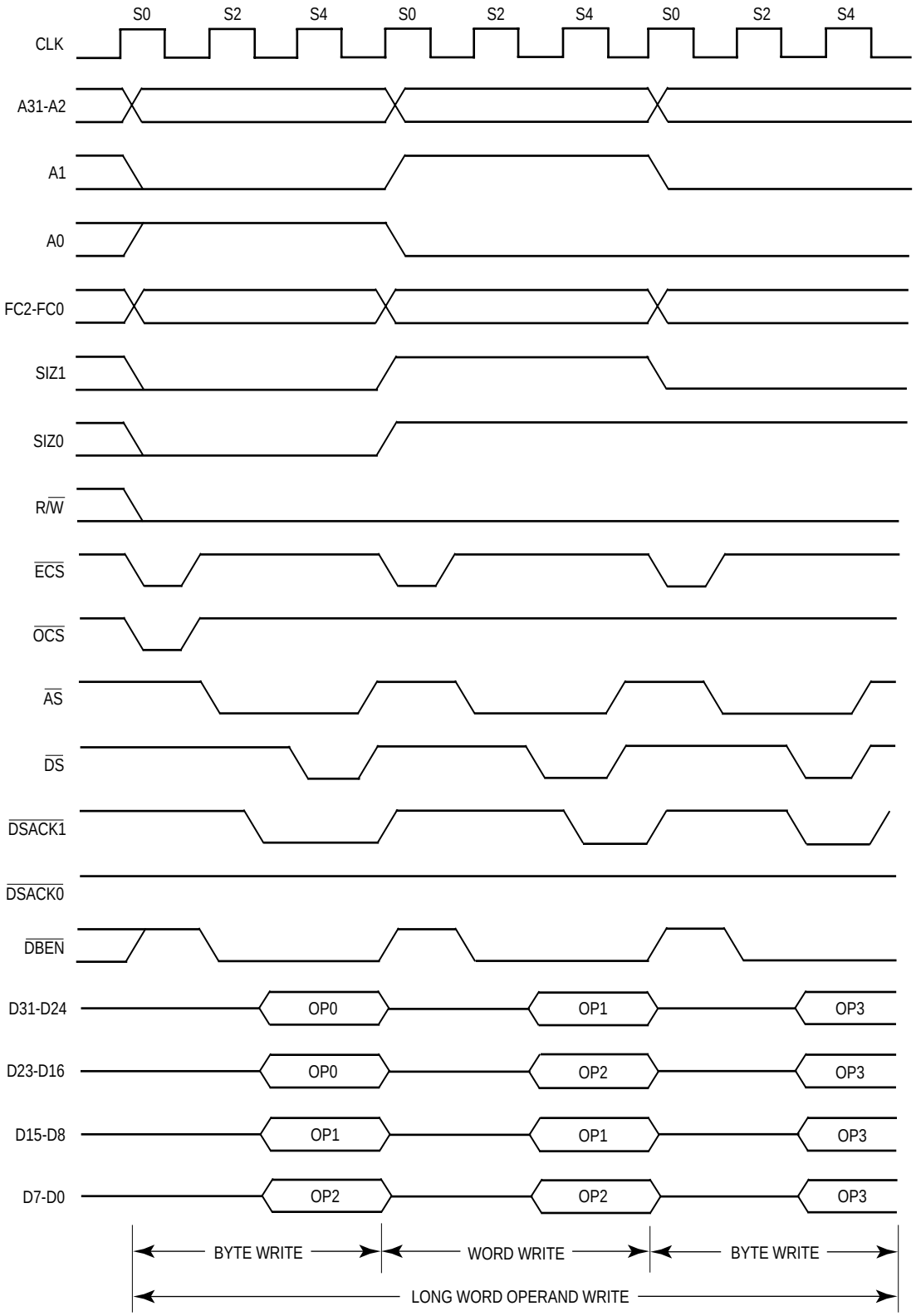


Figure 7-10. Misaligned Long-Word Transfer to Word Port

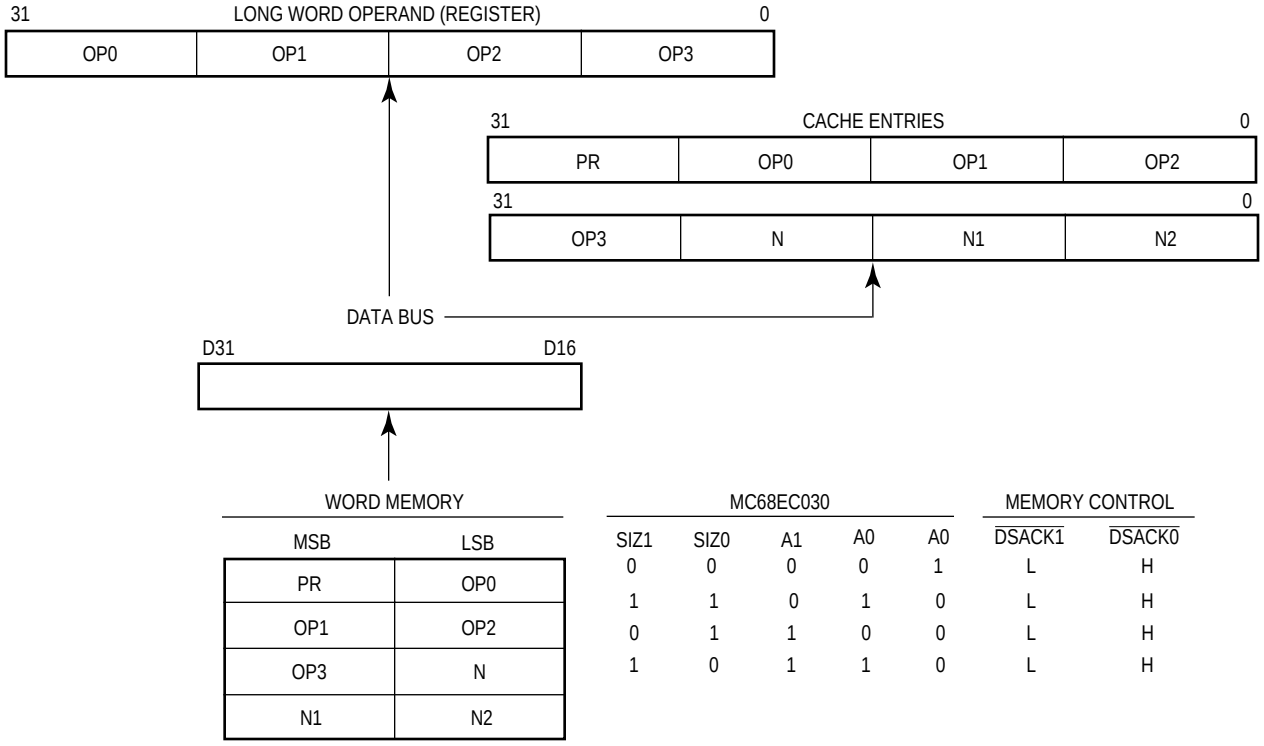


Figure 7-11. Misaligned Cachable Long-Word Transfer from Word Port Example

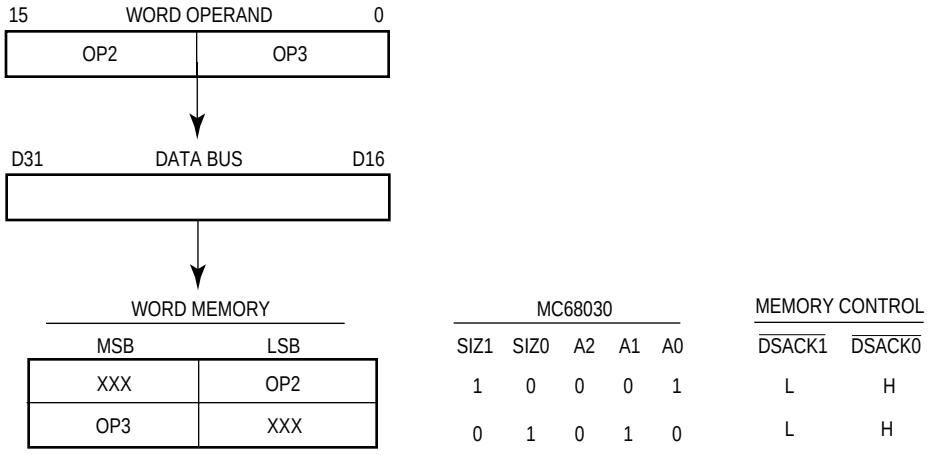


Figure 7-12. Misaligned Word Transfer to Word Port Example

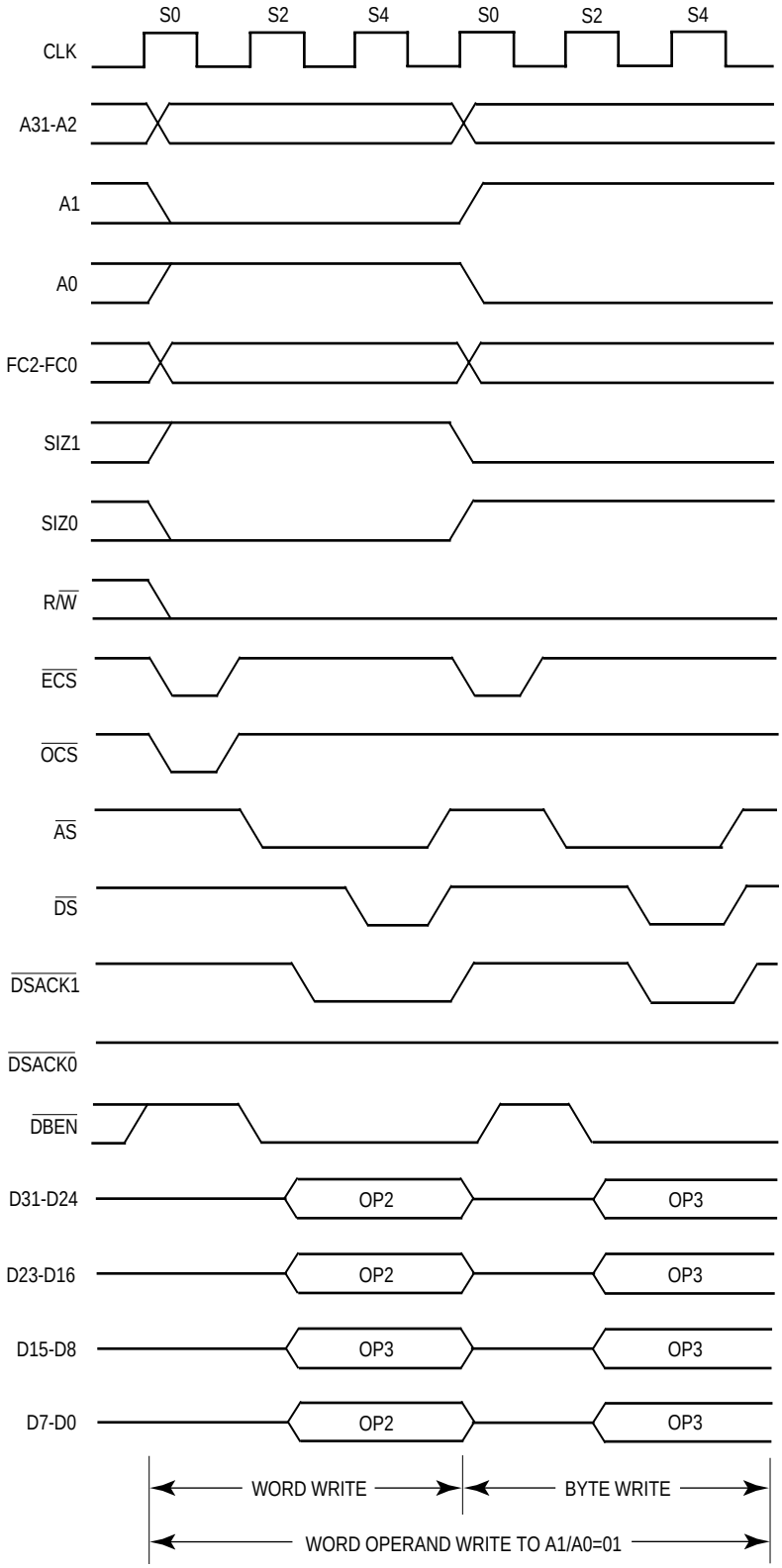


Figure 7-13. Misaligned Word Transfer to Word Port

Figures 7-15 and 7-16 show an example of a long-word transfer to an odd address in long-word-organized memory. In this example, a long-word access is attempted beginning at the least significant byte of a long-word-organized memory. Only one byte can be transferred in the first bus cycle. The second bus cycle then consists of a three-byte access to a long-word boundary. Since the memory is long-word organized, no further bus cycles are necessary.

Figure 7-17 shows the equivalent operation for a cachable data read cycle.

7.2.3 Effects of Dynamic Bus Sizing and Operand Misalignment

The combination of operand size, operand alignment, and port size determines the number of bus cycles required to perform a particular memory access. Table 7-6 shows the number of bus cycles required for different operand sizes to different port sizes with all possible alignment conditions for write cycles and noncachable read cycles.

Table 7-6. Memory Alignment and Port Size Influence on Write Bus Cycles

A1/A0	Number of Bus Cycles			
	00	01	10	11
Instruction*	1:2:4	N/A	N/A	N/A
Byte Operand	1:1:1	1:1:1	1:1:1	1:1:1
Word Operand	1:1:2	1:2:2	1:1:2	2:2:2
Long-Word Operand	1:2:4	2:3:4	2:2:4	2:3:4

Data Port Size — 32 Bits:16 Bits:8 Bits
*Instruction prefetches are always two words from a long-word boundary.

This table shows that bus cycle throughput is significantly affected by port size and alignment. The MC68030 system designer and programmer should be aware of and account for these effects, particularly in time-critical applications.

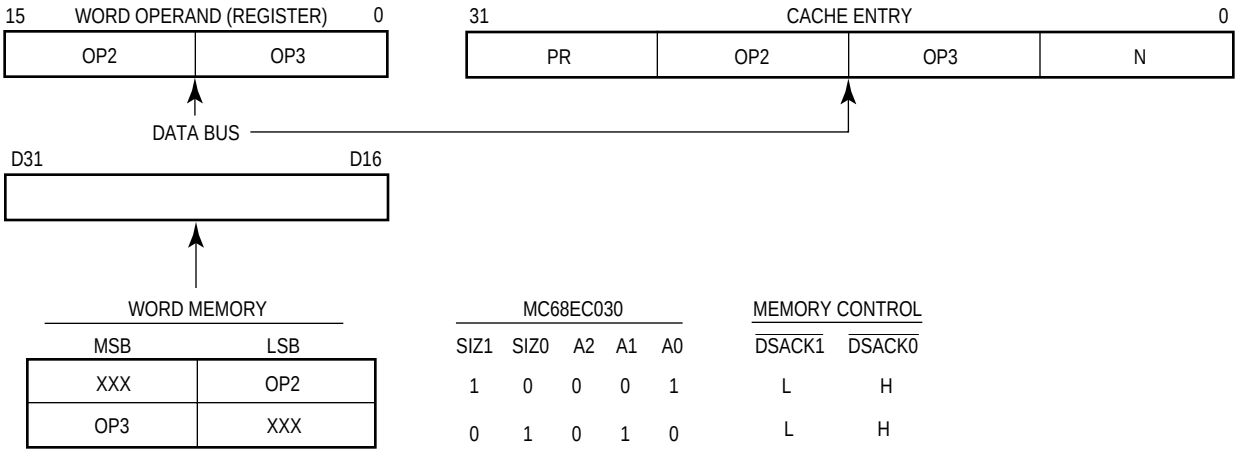


Figure 7-14. Example of Misaligned Cachable Word Transfer from Word Bus

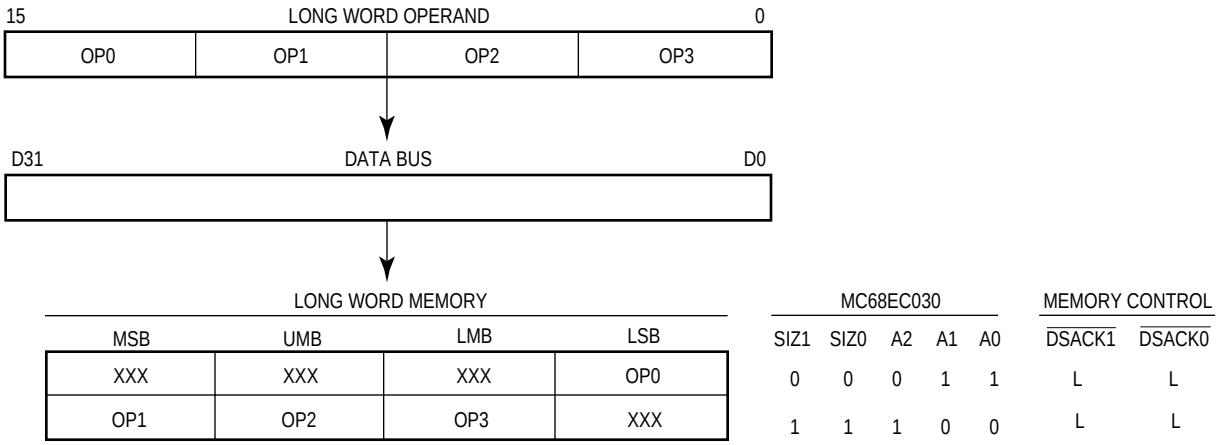


Figure 7-15. Misaligned Long-Word Transfer to Long-Word Port

Table 7-6 shows that the processor always prefetches instructions by reading a long word from a long-word address (A1:A0=00), regardless of port size or alignment. When the required instruction begins at an odd-word boundary, the processor attempts to fetch the entire 32 bits and loads both words into the instruction cache, if possible, although the second one is the required word. Even if the instruction access is not cached, the entire 32 bits are latched into an internal cache holding register from which the two instructions words can subsequently be referenced. Refer to **Section 11 Instruction Execution Timing** for a complete description of the cache holding register and pipeline operation.

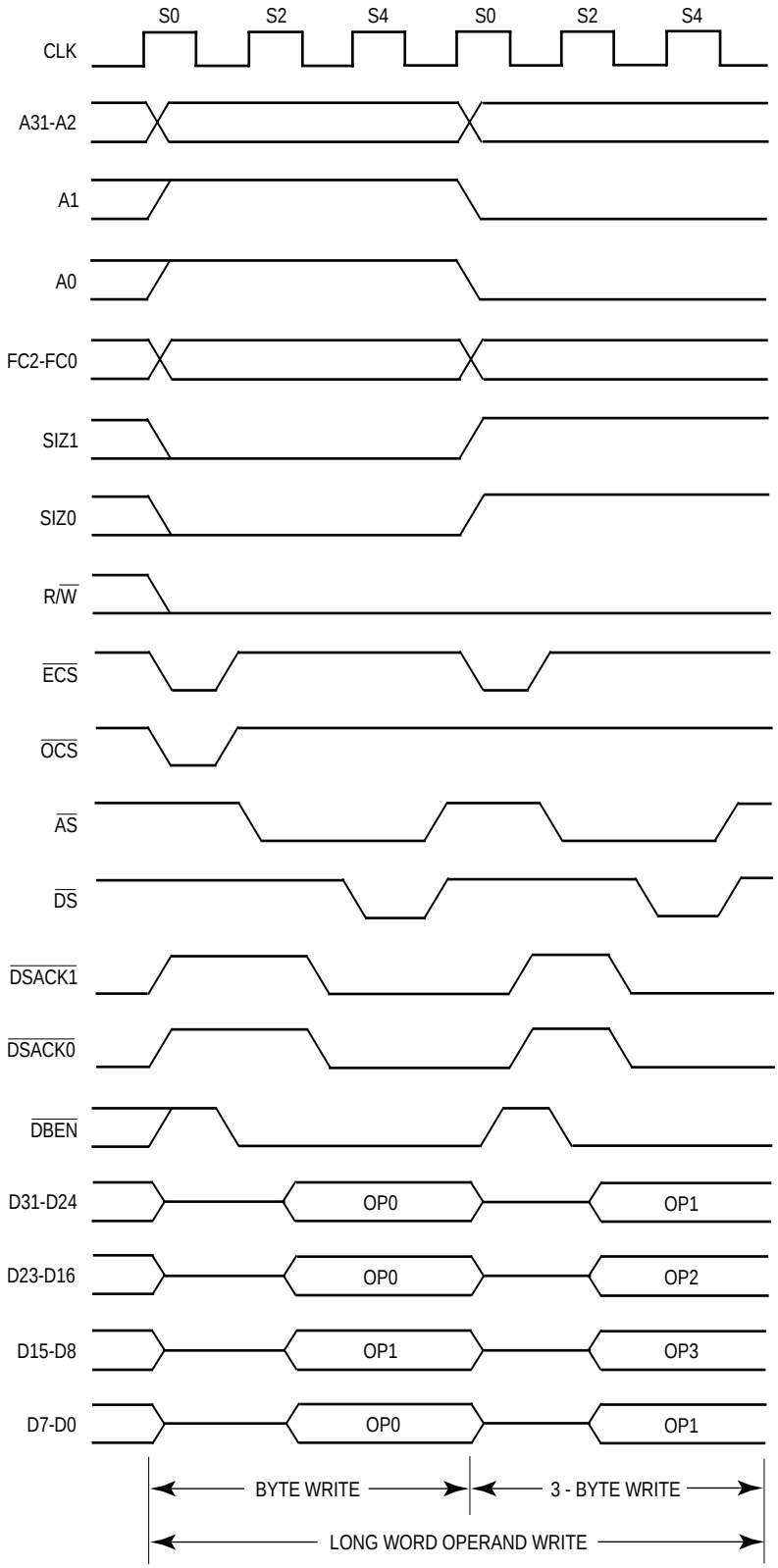


Figure 7-16. Misaligned Write Cycles to Long-Word Port

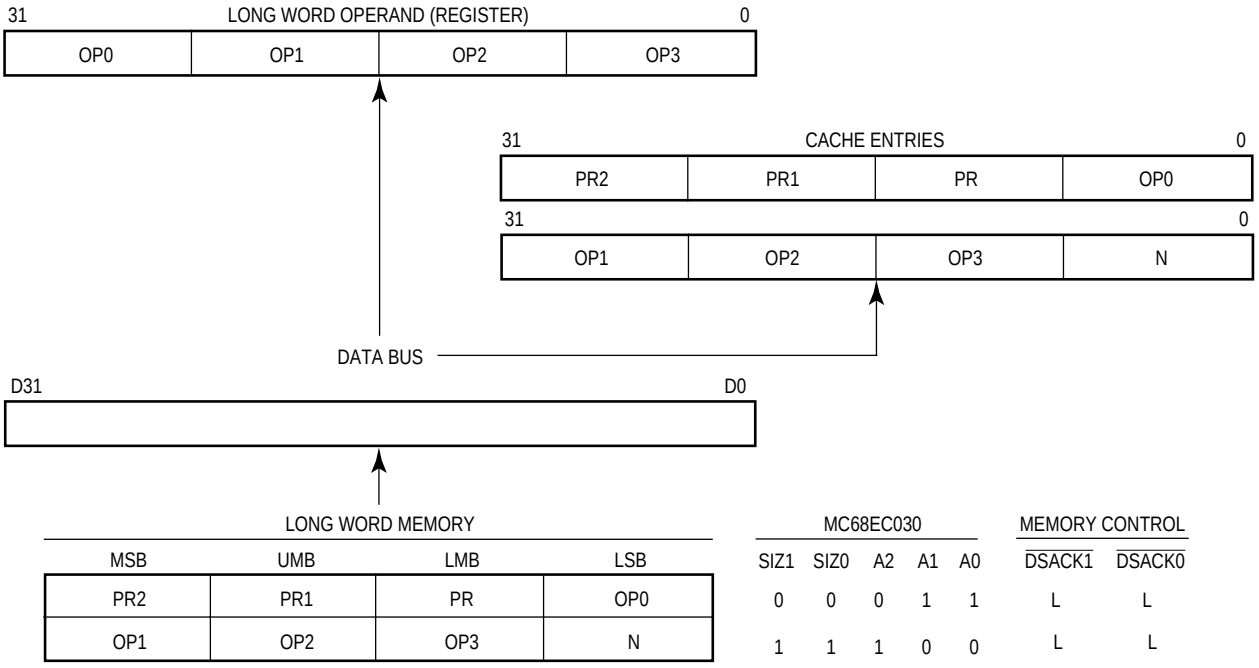


Figure 7-17. Misaligned Cachable Long-Word Transfer from Long-Word Bus

7.2.4 Address, Size, and Data Bus Relationships

The data transfer examples show how the MC68030 drives data onto or receives data from the correct byte sections of the data bus. Table 7-7 shows the combinations of the size signals and address signals that are used to generate byte enable signals for each of the four sections of the data bus for noncachable read cycles and all write cycles if the addressed device requires them. The port size also affects the generation of these enable signals as shown in the table. The four columns on the right correspond to the four byte enable signals. Letters B, W, and L refer to port sizes: B for 8-bit ports, W for 16-bit ports, and L for 32-bit ports. The letters B, W, and L imply that the byte enable signal should be true for that port size. A dash (—) implies that the byte enable signal does not apply.

The MC68030 always drives all sections of the data bus because, at the start of a write cycle, the bus controller does not know the port size. The byte enable signals in the table apply only to read operations that are not to be internally cached and to write operations. For cachable read cycles, during which the data is cached, the addressed port must drive all sections of the bus on which it resides.

Table 7-7. Data Bus Write Enable Signals for Byte, Word, and Long-Word Ports

Transfer Size	SIZ1	SIZ0	A1	A0	Data Bus Active Sections Byte (B) - Word (W) - Long-Word (L) Ports			
					D31:D24	D23:D16	D15:D8	D7:D0
Byte	0	1	0	0	BWL	—	—	—
	0	1	0	1	B	WL	—	—
	0	1	1	0	BW	—	L	—
	0	1	1	1	B	W	—	L
Word	1	0	0	0	BWL	WL	—	—
	1	0	0	1	B	WL	L	—
	1	0	1	0	BW	W	L	L
	1	0	1	1	B	W	—	L
3 Byte	1	1	0	0	BWL	WL	L	—
	1	1	0	1	B	WL	L	L
	1	1	1	0	BW	W	L	L
	1	1	1	1	B	W	—	L
Long Word	0	0	0	0	BWL	WL	L	L
	0	0	0	1	B	WL	L	L
	0	0	1	0	BW	W	L	L
	0	0	1	1	B	W	—	L

The table shows that the MC68030 transfers the number of bytes specified by the size signals to or from the specified address unless the operand is misaligned or the number of bytes is greater than the port width. In these cases, the device transfers the greatest number of bytes possible for the port. For example, if the size is four bytes and the address offset (A1:A0) is 01, a 32-bit slave can only receive three bytes in the current bus cycle. A 16- or 8-bit[lz slave can only receive one byte. The table defines the byte enables for all port sizes. Byte data strobes can be obtained by combining the enable signals with the data strobe signal. Devices residing on 8-bit ports can use the data strobe by itself since there is only one valid byte for every transfer. These enable or strobe signals select only the bytes required for write cycles or for noncachable read cycles. The other bytes are not selected, which prevents incorrect accesses in sensitive areas such as I/O.

Figure 7-18 shows a logic diagram for one method for generating byte data enable signals for 16- and 32-bit ports from the size and address encodings and the read/write signal.

7.2.5 MC68030 versus MC68020 Dynamic Bus Sizing

The MC68030 supports the dynamic bus sizing mechanism of the MC68020 for asynchronous bus cycles (terminated with DSACKx) with two restrictions. First, for a cachable access within the boundaries of an aligned long word, the port size must be consistent throughout the transfer of each long word. For example, when a byte port resides at address $00, addresses $01, $02, and $03 must also correspond to byte ports. Second, the port must supply as much data as it signals as port size, regardless of the transfer size indicated with the size signals and the address offset indicated by A0 and A1 for cachable accesses. Otherwise, dynamic bus sizing is identical in the two processors.

7.2.6 Cache Filling

The on-chip data and instruction caches, described in **Section 6 On-Chip Cache Memories**, are each organized as 16 lines of four long-word entries each. For each line, a tag contains the most significant bits of the logical address, FC2 (instruction cache) or FC0–FC2 (data cache), and a valid bit for each entry in the line. An entry fill operation loads an entire long word accessed from memory into a cache entry. This type of fill operation is performed when one entry of a line is not valid and an access is cachable. A burst fill operation is requested when a tag miss occurs for the current cycle or when all four entires in the cache line are invalid (provided the cache is enabled and burst filling for the cache is enabled). The burst fill operation attempts to fill all four entries in the line. To support burst filling, the slave device must have a 32-bit port and must have a burst mode capability; that is, it must acknowledge a burst request with the cache burst acknowledge (CBACK) signal. It must also terminate the burst accesses with STERM and place a long word on the data bus for each transfer. The device may continue to supply successive long words, asserting STERM with each one, until the cache line is full. For further information about filling the cache, both entry fills and burst mode fills, refer to **6.1.3 Cache Filling**, **7.3.4 Synchronous Read Cycle**, **7.3.5 Synchronous Write Cycle**, and **7.3.7 Burst Operation Cycles**, which discuss in detail the required bus cycles.

7.2.7 Cache Interactions

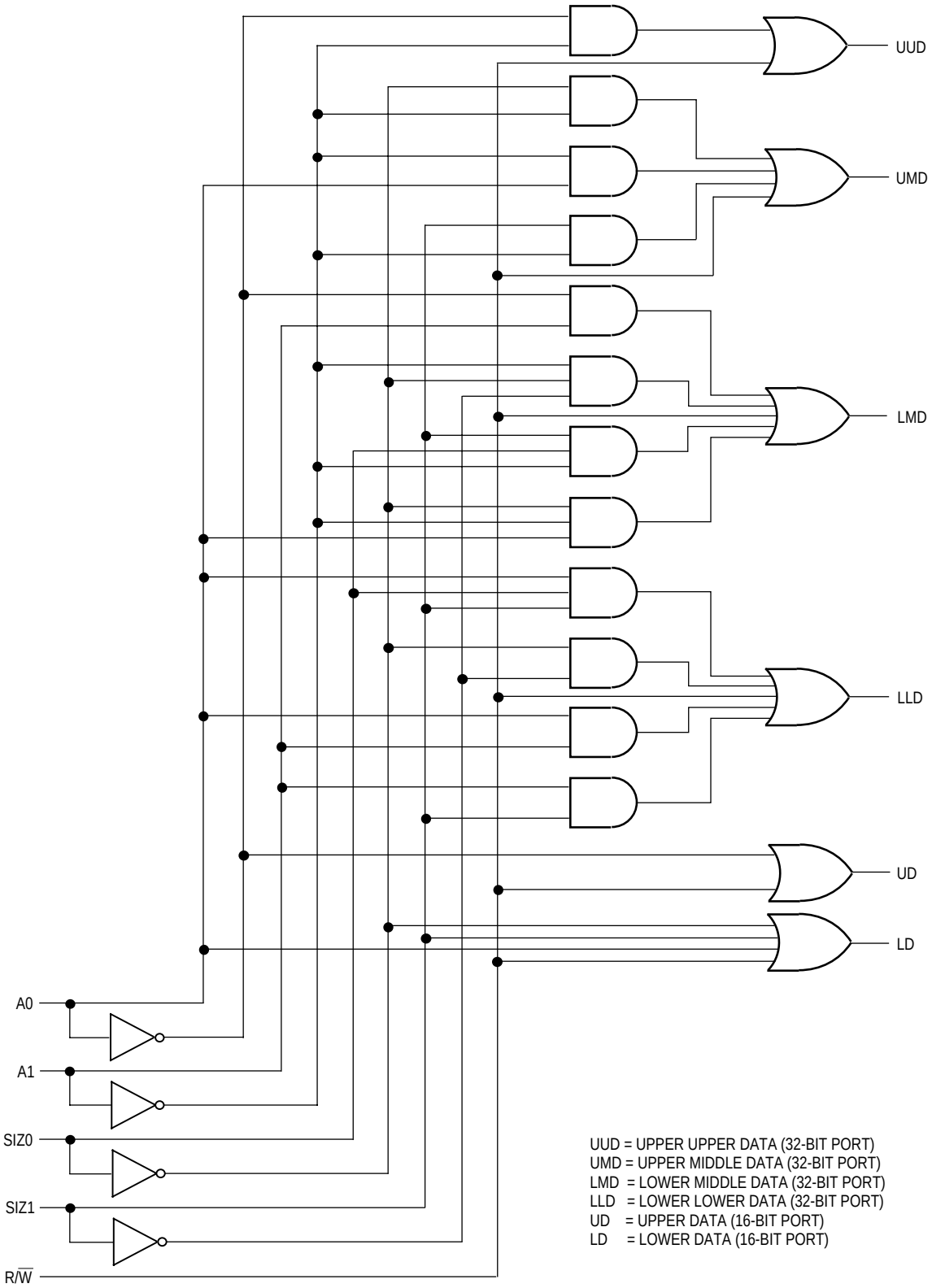
The organization and requirements of the on-chip instruction and data caches affect the interpretation of the $\overline{\text{DSACK}}$x and $\overline{\text{STERM}}$ signals. Since the MC68030 attempts to load all data operands and instructions that are cachable into the on-chip caches, the bus may operate differently when caching is enabled. Specifically, on cachable read cycles that terminate normally, the low-order address signals (A0 and A1) and the size signals do not apply.

The slave device must supply as much aligned data on the data bus as its port size allows, regardless of the requested operand size. This means that an 8-bit port must supply a byte, a 16-bit port must supply a word, and a 32-bit port must supply an entire long word. This data is loaded into the cache. For a 32-bit port, the slave device ignores A0 and A1 and supplies the long word beginning at the long-word boundary on the data bus. For a 16-bit[lz port, the device ignores A0 and supplies the entire word beginning at the lower word boundary on D16–D31 of the data bus. For a byte port, the device supplies the addressed byte on D24–D31.

If the addressed device cannot supply port-sized data or if the data should not be cached, the device must assert cache inhibit in ($\overline{\text{CIIN}}$) as it terminates the read cycle. If the bus cycle terminates abnormally, the MC68030 does not cache the data. For details of interactions of port sizes, misalignments, and cache filling, refer to **6.1.3 Cache Filling**.

The caches can also affect the assertion of $\overline{\text{AS}}$ and the operation of a read cycle. The search of the appropriate cache by the processor begins when the microsequencer requires an instruction or a data item. At this time, the bus controller may also initiate an external bus cycle in case the requested item is not resident in the instruction or data cache. If the bus is not occupied with another read or write cycle, the bus controller asserts the $\overline{\text{ECS}}$ signal (and the $\overline{\text{OCS}}$ signal, if appropriate). If an internal cache hit occurs, the external cycle aborts, and AS is not asserted. This makes it possible to have $\overline{\text{ECS}}$ asserted on multiple consecutive clock cycles. Notice that there is a minimum time specified from the negation of $\overline{\text{ECS}}$ to the next assertion of $\overline{\text{ECS}}$ (refer to MC68030EC/D, *MC68030 Electrical Specifications*.

Instruction prefetches can occur every other clock so that if, after an aborted cycle due to an instruction cache hit, the bus controller asserts $\overline{\text{ECS}}$ on the next clock, this second cycle is for a data fetch. However, data accesses that hit in the data cache can also cause the assertion of $\overline{\text{ECS}}$ and an aborted cycle. Therefore, since instruction and data accesses are mixed, it is possible to see multiple successive $\overline{\text{ECS}}$ assertions on the external bus if the processor is hitting in both caches and if the bus controller is free. Note that, if the bus controller is executing other cycles, these aborted cycles due to cache hits may not be seen externally. Also, $\overline{\text{OCS}}$ is asserted for the first external cycle of an operand transfer. Therefore, in the case of a misaligned data transfer where the first portion of the operand results in a cache hit (but the bus controller did not begin an external cycle and then abort it) and the second portion in a cache miss, $\overline{\text{OCS}}$ is asserted for the second portion of the operand.



NOTE: These select lines can be combined with the address decode circuitry or all can be generated within the same programmed array logic unit.

Figure 7-18. Byte Data Select Generation for 16- and 32-Bit Ports

7.2.8 Asynchronous Operation

The MC68030 bus may be used in an asynchonous manner. In that case, the external devices connected to the bus can operate at clock frequencies different from the clock for the MC68030. Asynchronous operation requires using only the handshake line ($\overline{AS}$, $\overline{DS}$, $\overline{DSACK1}$, $\overline{DSACK0}$, $\overline{BERR}$, and $\overline{HALT}$) to control data transfers. Using this method, AS signals the start of a bus cycle, and $\overline{DS}$ is used as a condition for valid data on a write cycle. Decoding the size outputs and lower address lines (A0 and A1) provides strobes that select the active portion of the data bus. The slave device (memory or peripheral) then responds by placing the requested data on the correct portion of the data bus for a read cycle or latching the data on a write cycle, and asserting the $\overline{DSACK1}$/$\overline{DSACK0}$ combination that corresponds to the port size to terminate the cycle. If no slave responds or the access is invalid, external control logic asserts the $\overline{BERR}$ or $\overline{BERR}$ and $\overline{HALT}$ signal(s) to abort or retry the bus cycle, respectively.

The $\overline{DSACKx}$ signals can be asserted before the data from a slave device is valid on a read cycle. The length of time that $\overline{DSACKx}$ may precede data is given by parameter #31, and it must be met in any asynchronous system to insure that valid data is latched into the processor. (Refer to MC68030EC/D, *MC68030 Electrical Specifications* for timing parameters.) Notice that no maximum time is specified from the assertion of $\overline{AS}$ to the assertion of $\overline{DSACKx}$. Although the processor can transfer data in a minimum of three clock cycles when the cycle is terminated with $\overline{DSACKx}$, the processor inserts wait cycles in clock period increments until $\overline{DSACKx}$ is recognized.

The $\overline{BERR}$ and/or $\overline{HALT}$ signals can be asserted after the $\overline{DSACKx}$ signal(s) is asserted. $\overline{BERR}$ and/or $\overline{HALT}$ must be asserted within the time given as parameter #48, after $\overline{DSACKx}$ is asserted in any asynchronous system. If this maximum delay time is violated, the processor may exhibit erratic behavior.

For asynchronous read cycles, the value of $\overline{CIIN}$ is internally latched on the rising edge of bus cycle state 4. Refer to **7.3.1 Asynchronous Read Cycle** for more details on the states for asynchonous read cycles.

During any bus cycle terminated by $\overline{DSACKx}$ or $\overline{BERR}$, the assertion of $\overline{CBACK}$ is completely ignored.

7.2.9 Synchronous Operation with $\overline{DSACKx}$

Although cycles terminated with the $\overline{DSACKx}$ signals are classified as asynchronous and cycles terminated with $\overline{STERM}$ are classified as synchronous, cycles terminated with $\overline{DSACKx}$ can also operate synchronously in that signals are interpreted relative to clock edges.

The devices that use these cycles must synchronize the responses to the MC68030 clock to be synchronous. Since they terminate bus cycles with the $\overline{DSACKx}$ signals, the dynamic bus sizing capabilities of the MC68030 are available. In addition, the minimum cycle time for these cycles is also three clocks.

To support those systems that use the system clock to generate $\overline{DSACKx}$ and other asynchronous inputs, the asynchronous input setup time (parameter #47A) and the asynchronous input hold time (parameter #47B) are given. If the setup and hold times are met for the assertion or negation of a signal, such as $\overline{DSACKx}$, the processor can be guaranteed to recognize that signal level on that specific falling edge of the system clock. If the assertion of $\overline{DSACKx}$ is recognized on a particular falling edge of the clock, valid data is latched into the processor (for a read cycle) on the next falling clock edge provided the data meets the data setup time (parameter #27). In this case, parameter #31 for asynchronous operation can be ignored. The timing parameters referred to are described in MC68030EC/D, *MC68030 Electrical Specifications*. If a system asserts $\overline{DSACKx}$ for the required window around the falling edge of S2 and obeys the proper bus protocol by maintaining $\overline{DSACKx}$ (and/or $\overline{BERR}$/$\overline{HALT}$) until and throughout the clock edge that negates AS (with the appropriate asynchronous input hold time specified by parameter #47B), no wait states are inserted. The bus cycle runs at its maximum speed (three clocks per cycle) for bus cycles terminated with $\overline{DSACKx}$.

To assure proper operation in a synchronous system when $\overline{BERR}$ or $\overline{BERR}$ and $\overline{HALT}$ is asserted after $\overline{DSACKx}$, $\overline{BERR}$ (and $\overline{HALT}$) must meet the appropriate setup time (parameter #27A) prior to the falling clock edge one clock cycle after $\overline{DSACKx}$ is recognized. This setup time is critical, and the MC68030 may exhibit erratic behavior if it is violated.

When operating synchronously, the data-in setup and hold times for synchronous cycles may be used instead of the timing requirements for data relative to the $\overline{DS}$ signal.

The value of $\overline{CIIN}$ is latched on the rising edge of bus cycle state 4 for all cycles terminated with $\overline{DSACKx}$.

7.2.10 Synchronous Operation with $\overline{STERM}$

The MC68030 supports synchronous bus cycles terminated with $\overline{STERM}$. These cycles, for 32-bit ports only, are similar to cycles terminated with $\overline{DSACKx}$. The main difference is that $\overline{STERM}$ can be asserted (and data can be transferred) earlier than for a cycle terminated with $\overline{DSACKx}$, causing the processor to perform a minimum access time transfer in two clock periods. However, wait cycles can be inserted by delaying the assertion of $\overline{STERM}$ appropriately.

Using $\overline{STERM}$ instead of $\overline{DSACKx}$ in any bus cycle makes the cycle synchronous. Any bus cycle is synchronous if:

1. Neither $\overline{DSACKx}$ nor $\overline{AVEC}$ is recognized during the cycle.
2. The port size is 32 bits.
3. Synchronous input setup and hold time requirements (specifications #60 and #61) for $\overline{STERM}$ are met.

Burst mode operation requires the use of $\overline{STERM}$ to terminate each of its cycles. The first cycle of any burst transfer must be a synchronous cycle as described in the preceding paragraph. The exact timing of this cycle is controlled by the assertion of $\overline{STERM}$, and wait cycles can be inserted as necessary. However, the minimum cycle time is two clocks. If a burst operation is initiated and allowed to terminate normally, the second, third, and fourth cycles latch data on successive falling edges of the clock at a minimum. Again, the exact timing for these subsequent cycles is controlled by the timing of $\overline{STERM}$ for each of these cycles, and wait cycles can be inserted as necessary.

Although the synchronous input signals ($\overline{\text{STERM}}$, $\overline{\text{CIIN}}$, and $\overline{\text{CBACK}}$) must be stable for the appropriate setup and hold times relative to every rising edge of the clock during which $\overline{\text{AS}}$ is asserted, the assertion or negation of $\overline{\text{CBACK}}$ and $\overline{\text{CIIN}}$ is internally latched on the rising edge of the clock for which $\overline{\text{STERM}}$ is asserted in a synchronous cycle.

The $\overline{\text{STERM}}$ signal can be generated from the address bus and function code value and does not need to be qualified with the $\overline{\text{AS}}$ signal. If $\overline{\text{STERM}}$ is asserted and no cycle is in progress (even if the cycle has begun, $\overline{\text{ECS}}$ is asserted and then the cycle is aborted), $\overline{\text{STERM}}$ is ignored by the MC68030.

Similarly, $\overline{\text{CBACK}}$ can be asserted independently of the assertion of $\overline{\text{CBREQ}}$. If a cache burst is not requested, the assertion of $\overline{\text{CBACK}}$ is ignored.

The assertion of $\overline{\text{CIIN}}$ is ignored when the appropriate cache is not enabled or when cache inhibit out ($\overline{\text{CIOUT}}$) is asserted. It is also ignored during write cycles or translation table searches.

NOTE

$\overline{\text{STERM}}$ and $\overline{\text{DSACKx}}$ should never be asserted during the same bus cycle.

7.3 DATA TRANSFER CYCLES

The transfer of data between the processor and other devices involves the following signals:

- Address Bus A0–A31
- Data Bus D0–D31
- Control Signals

The address and data buses are both parallel nonmultiplexed buses. The bus master moves data on the bus by issuing control signals, and the asynchronous/synchronous bus uses a handshake protocol to insure correct movement of the data. In all bus cycles, the bus master is responsible for de-skewing all signals it issues at both the start and the end of the cycle. In addition, the bus master is responsible for de-skewing the acknowledge and data signals from the slave devices. The following paragraphs define read, write, and read-modify-write cycle operations. An additional paragraph describes burst mode transfers.

Each of the bus cycles is defined as a succession of states. These states apply to the bus operation and are different from the processor states described in **Section 4 Processing States**. The clock cycles used in the descriptions and timing diagrams of data transfer cycles are independent of the clock frequency. Bus operations are described in terms of external bus states.

7.3.1 Asynchronous Read Cycle

During a read cycle, the processor receives data from a memory, coprocessor, or peripheral device. If the instruction specifies a long-word operation, the MC68030 attempts to read four bytes at once. For a word operation, it attempts to read two bytes at once, and for a byte operation, one byte. For some operations, the processor requests a three-byte transfer. The processor properly positions each byte internally. The section of the data bus from which each byte is read depends on the operand size, address signals (A0–A1), $\overline{\text{CIIN}}$ and $\overline{\text{CIOUT}}$, whether the internal caches are enabled, and the port size. Refer to **7.2.1 Dynamic Bus Sizing**, **7.2.2 Misaligned Operands**, and **7.2.6 Cache Filling** for more information on dynamic bus sizing, misaligned operands, and cache interactions.

Figure 7-19 is a flowchart of an asynchronous long-word read cycle. Figure 7-20 is a flowchart of a byte read cycle. The following figures show functional read cycle timing diagrams specified in terms of clock periods. Figure 7-21 corresponds to byte and word read cycles from a 32-bit port. Figure 7-22 corresponds to a long-word read cycle from an 8-bit port. Figure 7-23 also applies to a long-word read cycle, but from a 16-bit port.

State 0

The read cycle starts in state 0 (S0). The processor drives $\overline{\text{ECS}}$ low, indicating the beginning of an external cycle. When the cycle is the first external cycle of a read operand operation, operand cycle start ($\overline{\text{OCS}}$) is driven low at the same time. During S0, the processor places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the cycle. The processor drives R/$\overline{\text{W}}$ high for a read cycle and drives $\overline{\text{DBEN}}$ inactive to disable the data buffers. SIZ0–SIZ1 become valid, indicating the number of bytes requested to be transferred. $\overline{\text{CIOUT}}$ also becomes valid, indicating the state of the MMU CI bit in the address translation descriptor or in the appropriate TTx register.

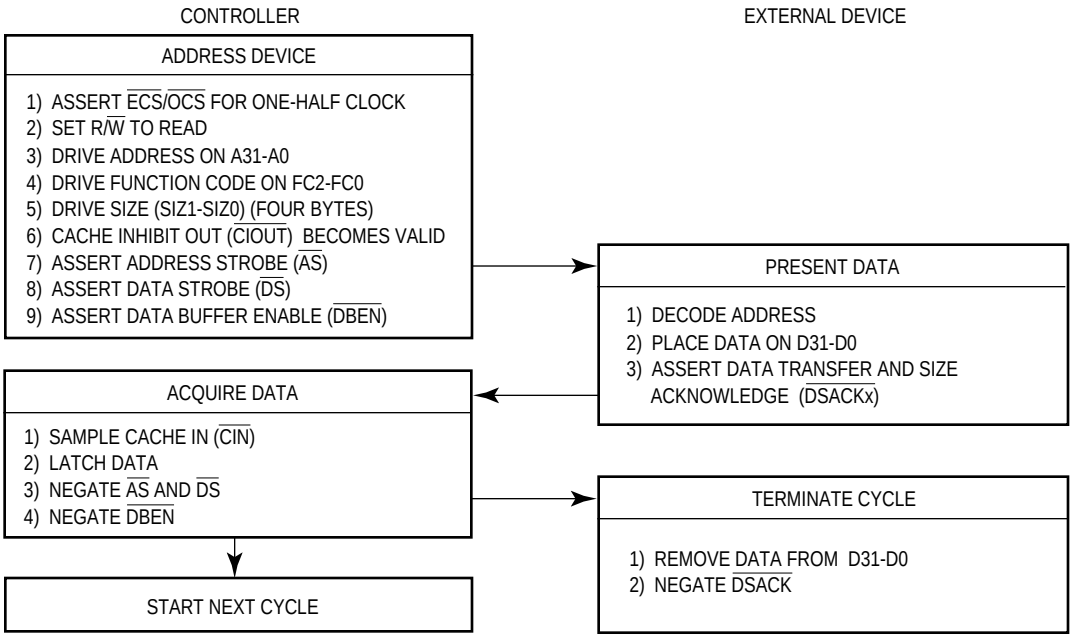


Figure 7-19. Asynchronous Long-Word Read Cycle Flowchart

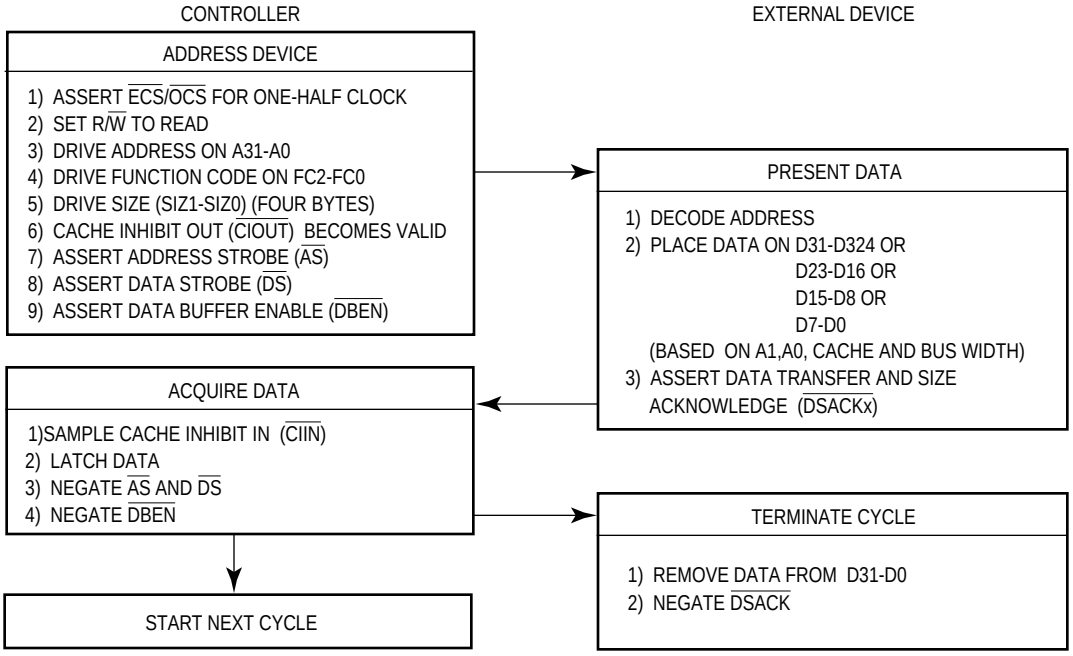


Figure 7-20. Asynchronous Byte Read Cycle Flowchart

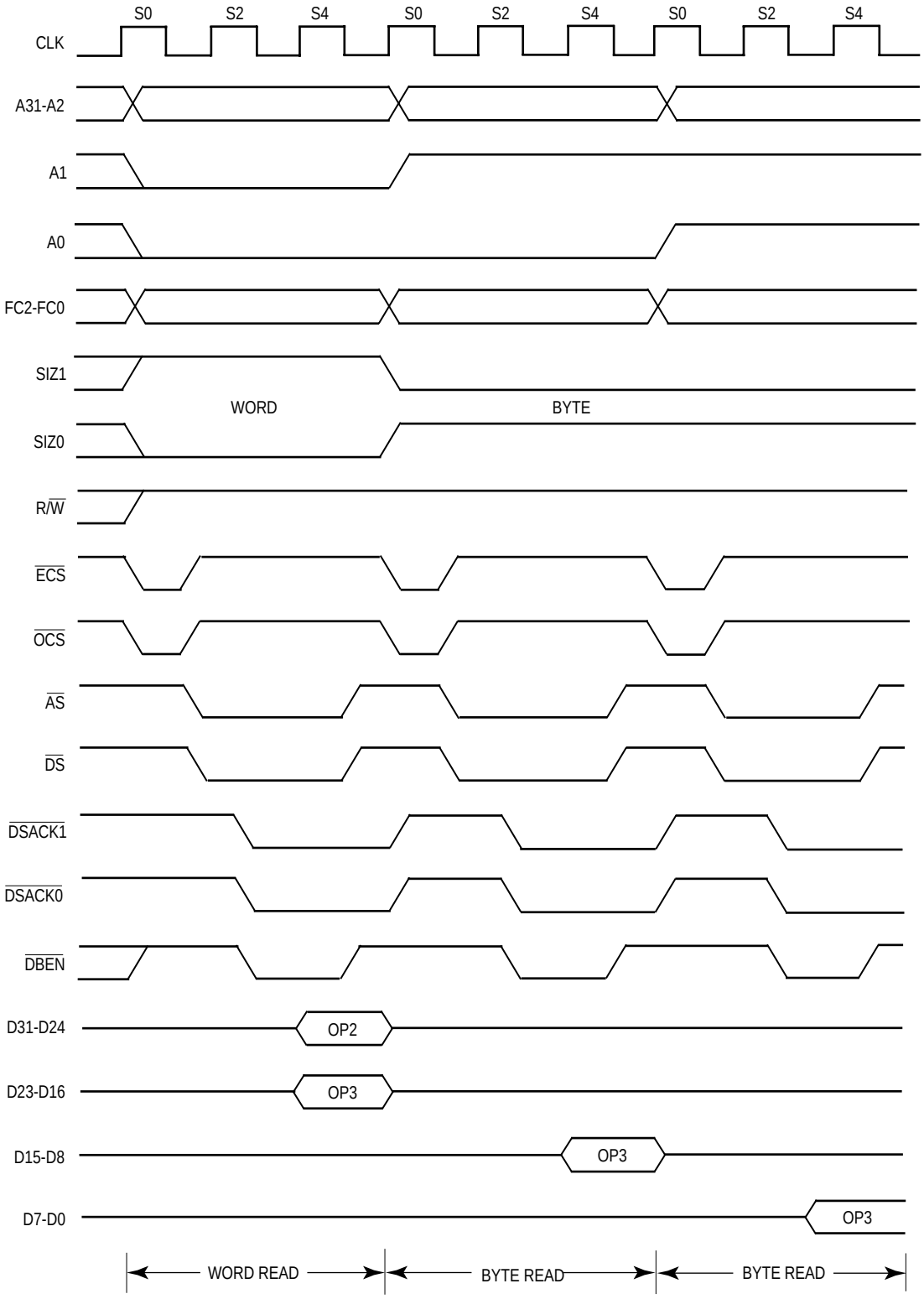


Figure 7-21. Asynchronous Byte and Word Read Cycles — 32-Bit Port

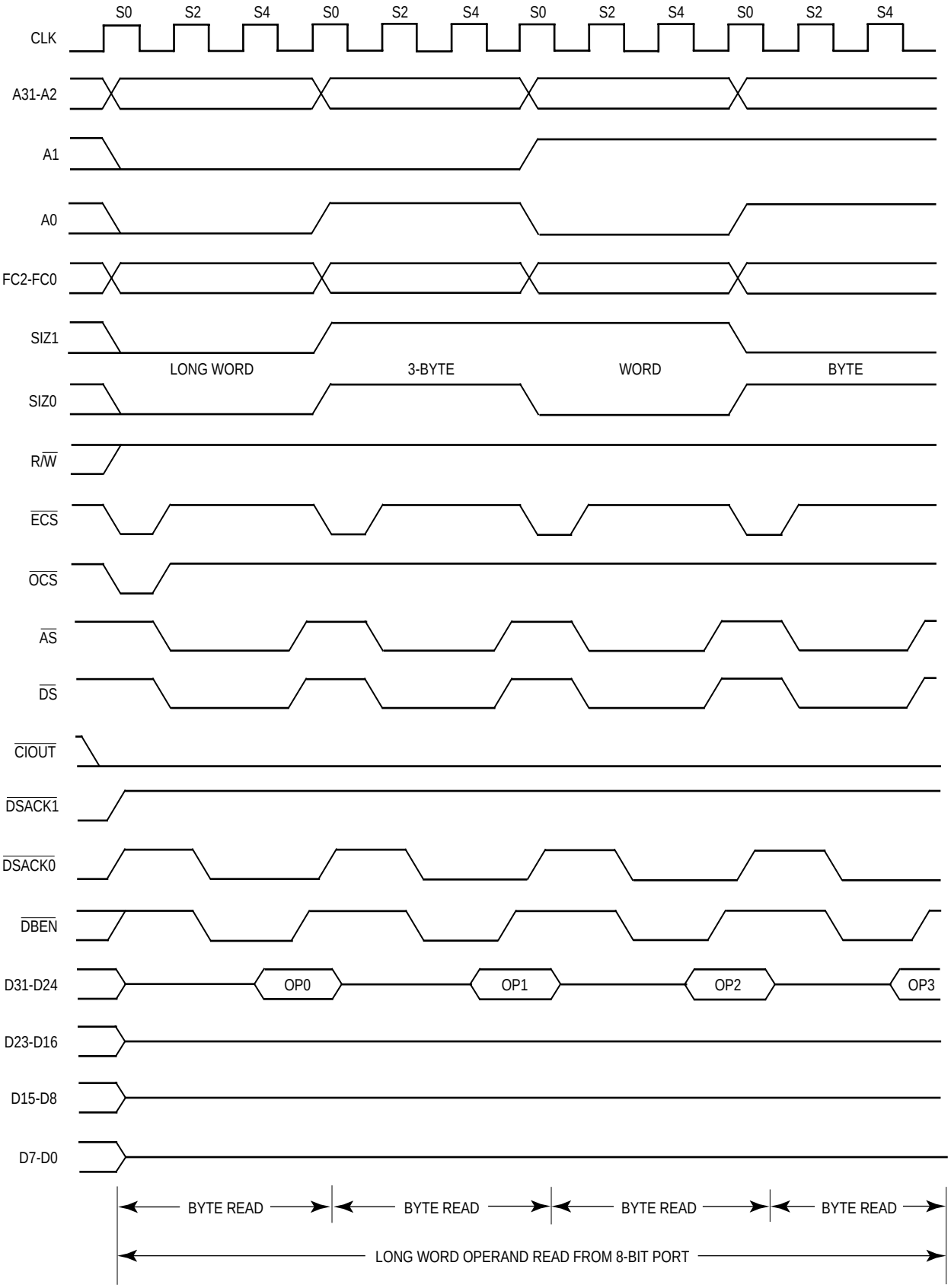


Figure 7-22. Long-Word Read — 8-Bit Port with $\overline{\text{CIOUT}}$ Asserted

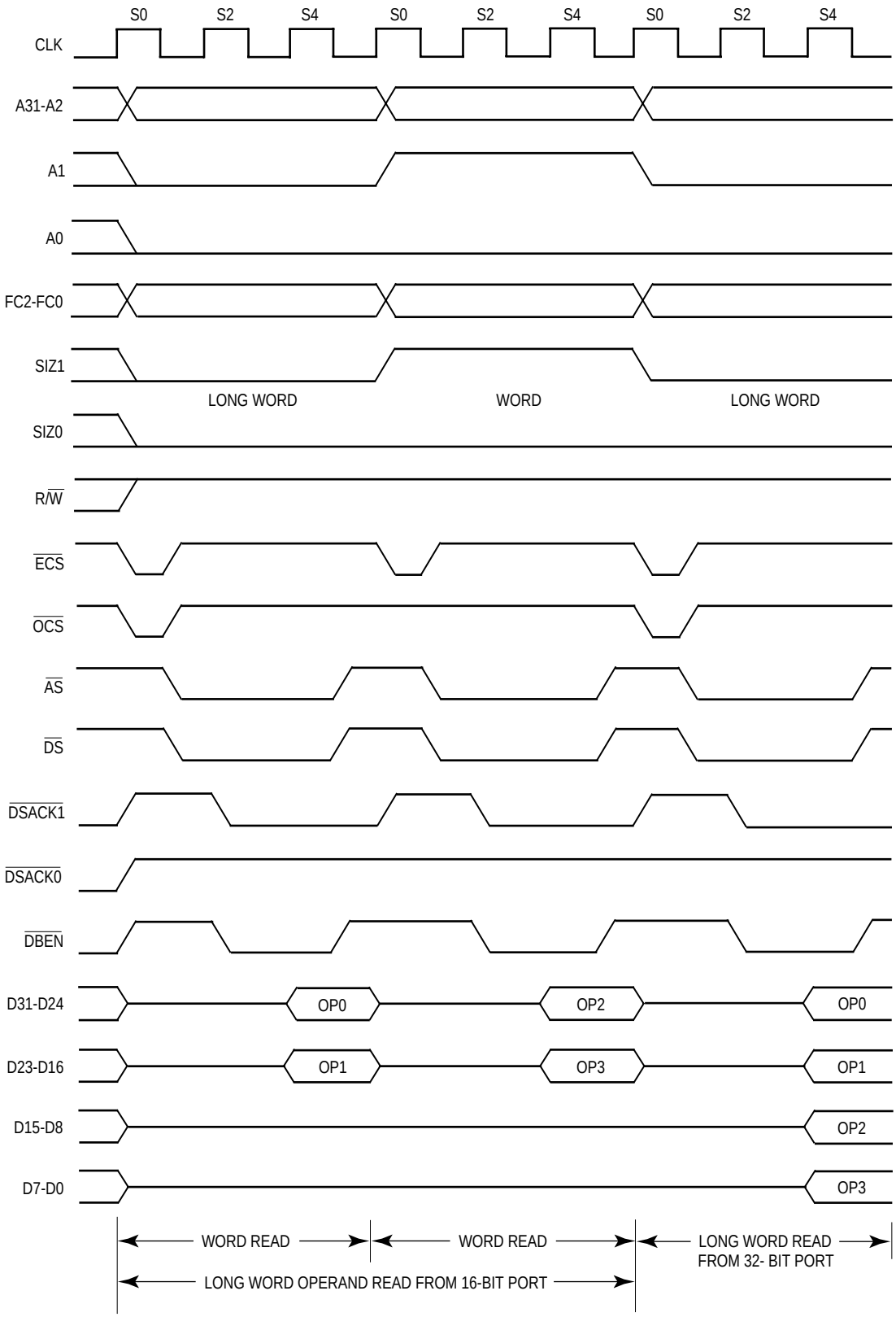


Figure 7-23. Long-Word Read — 16-Bit and 32-Bit Port

State 1

One-half clock later in state 1 (S1), the processor asserts $\overline{AS}$ indicating that the address on the address bus is valid. The processor also asserts $\overline{DS}$ also during S1. In addition, the $\overline{ECS}$ (and $\overline{OCS}$, if asserted) signal is negated during S1.

State 2

During state 2 (S2), the processor asserts DBEN to enable external data buffers. The selected device uses R/$\overline{W}$, SIZ0–SIZ1, A0–A1, $\overline{CIOUT}$, and $\overline{DS}$ to place its information on the data bus, and drives CIIN if appropriate. Any or all of the bytes (D24–D31, D16–D23, D8–D15, and D0–D7) are selected by SIZ0–SIZ1 and A0–A1. Concurrently, the selected device asserts $\overline{DSACKx}$.

State 3

As long as at least one of the $\overline{DSACKx}$ signals is recognized by the end of S2 (meeting the asynchronous input setup time requirement), data is latched on the next falling edge of the clock, and the cycle terminates. If $\overline{DSACKx}$ is not recognized by the start of state 3 (S3), the processor inserts wait states instead of proceeding to states 4 and 5. To ensure that wait states are inserted, both $\overline{DSACK0}$ and $\overline{DSACK1}$ must remain negated throughout the asynchronous input setup and hold times around the end of S2. If wait states are added, the processor continues to sample the $\overline{DSACKx}$ signals on the falling edges of the clock until one is recognized.

State 4

The processor samples $\overline{CIIN}$ at the beginning of state 4 (S4). Since $\overline{CIIN}$ is defined as a synchronous input, whether asserted or negated, it must meet the appropriate synchronous input setup and hold times on every rising edge of the clock while $\overline{AS}$ is asserted. At the end of S4, the processor latches the incoming data.

State 5

The processor negates $\overline{AS}$, $\overline{DS}$, and $\overline{DBEN}$ during state 5 (S5). It holds the address valid during S5 to provide address hold time for memory systems. R/$\overline{W}$, SIZ0–SIZ1, and FC0–FC2 also remain valid throughout S5.

The external device keeps its data and $\overline{DSACKx}$ signals asserted until it detects the negation of $\overline{AS}$ or $\overline{DS}$ (whichever it detects first). The device must remove its data and negate $\overline{DSACKx}$ within approximately one clock period after sensing the negation of $\overline{AS}$ or $\overline{DS}$. $\overline{DSACKx}$ signals that remain asserted beyond this limit may be prematurely detected for the next bus cycle.

7.3.2 Asynchronous Write Cycle

During a write cycle, the processor transfers data to memory or a peripheral device.

Figure 7-24 is a flowchart of a write cycle operation for a long-word transfer. The following figures show the functional write cycle timing diagrams specified in terms of clock periods. Figure 7-25 shows two write cycles (between two read cycles with no idle time) for a 32-bit port. Figure 7-26 shows byte and word write cycles to a 32-bit port. Figure 7-27 shows a long-word write cycle to an 8-bit port. Figure 7-28 shows a long-word write cycle to a 16-bit port.

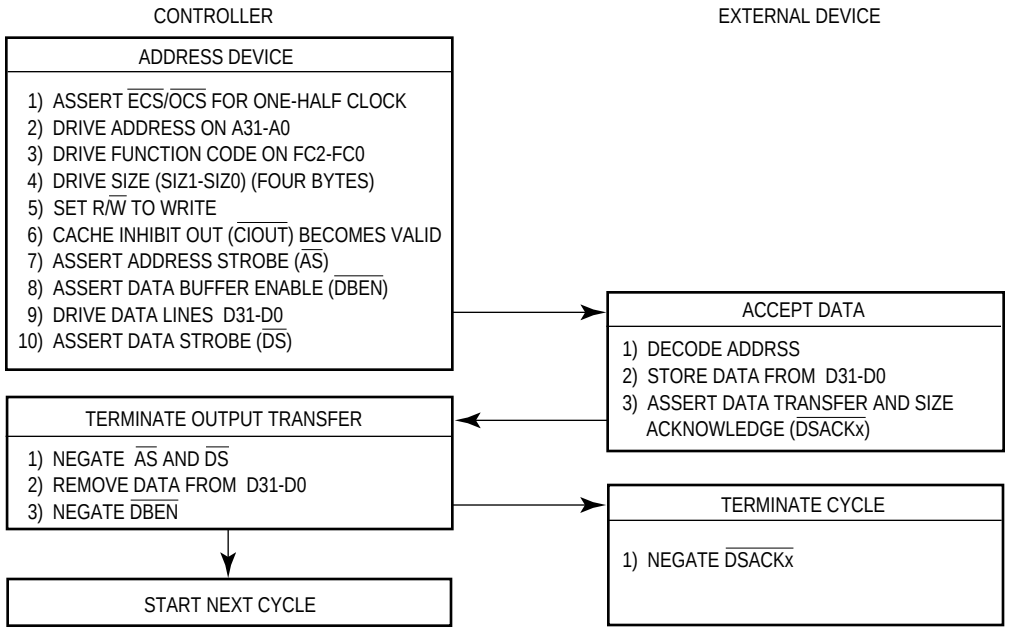


Figure 7-24. Asynchronous Write Cycle Flowchart

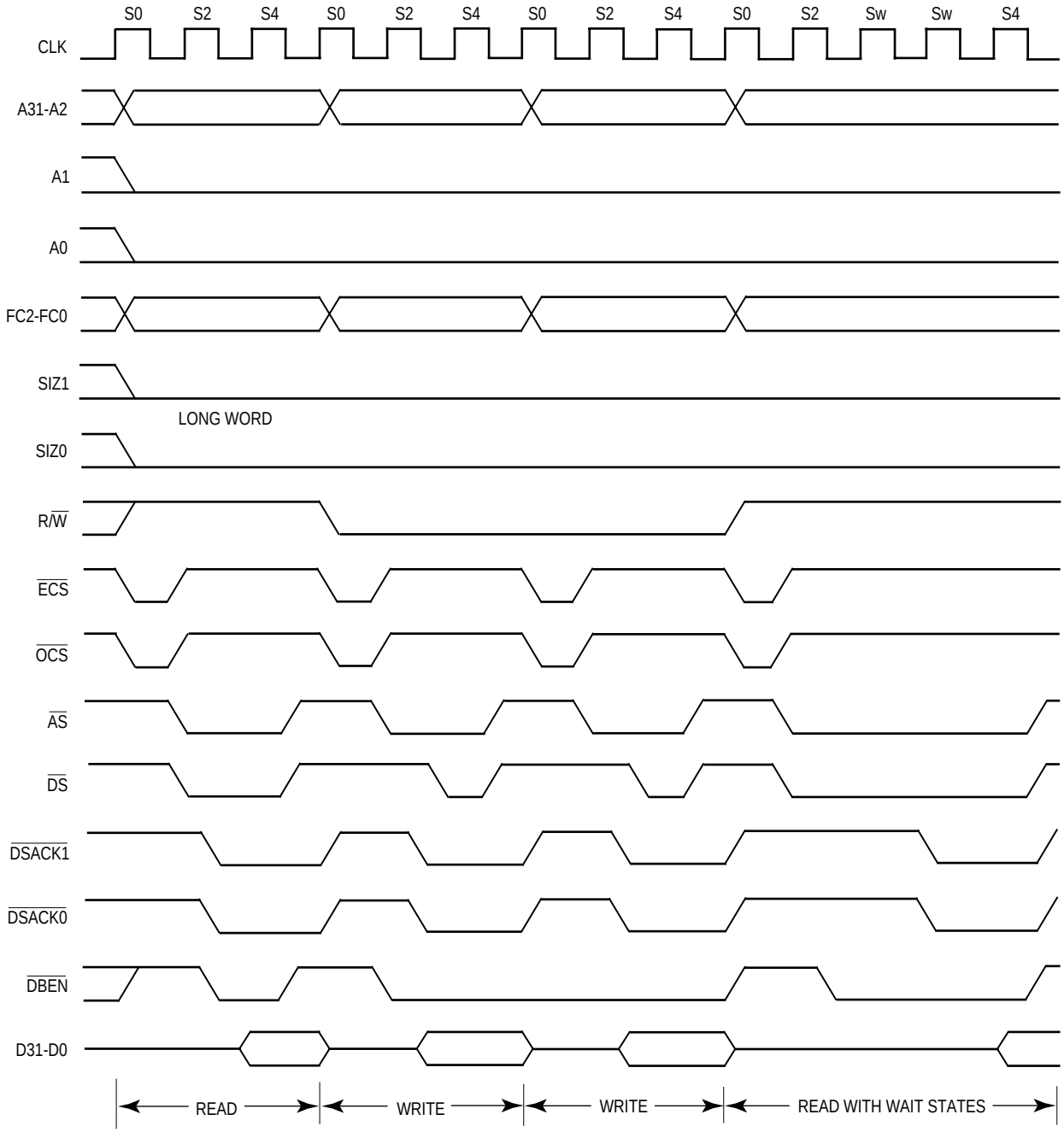


Figure 7-25. Asynchronous Read-Write-Read Cycles — 32-Bit Port

State 0

The write cycle starts in S0. The processor drives $\overline{\text{ECS}}$ low, indicating the beginning of an external cycle. When the cycle is the first external cycle of a write operation, $\overline{\text{OCS}}$ is driven low at the same time. During S0, the processor places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the cycle. The processor drives R/W low for a write cycle. SIZ0–SIZ1 become valid, indicating the number of bytes to be transferred. $\overline{\text{CIOUT}}$ also becomes valid, indicating

the state of the MMU CI bit in the address translation descriptor or in the appropriateTTx register.

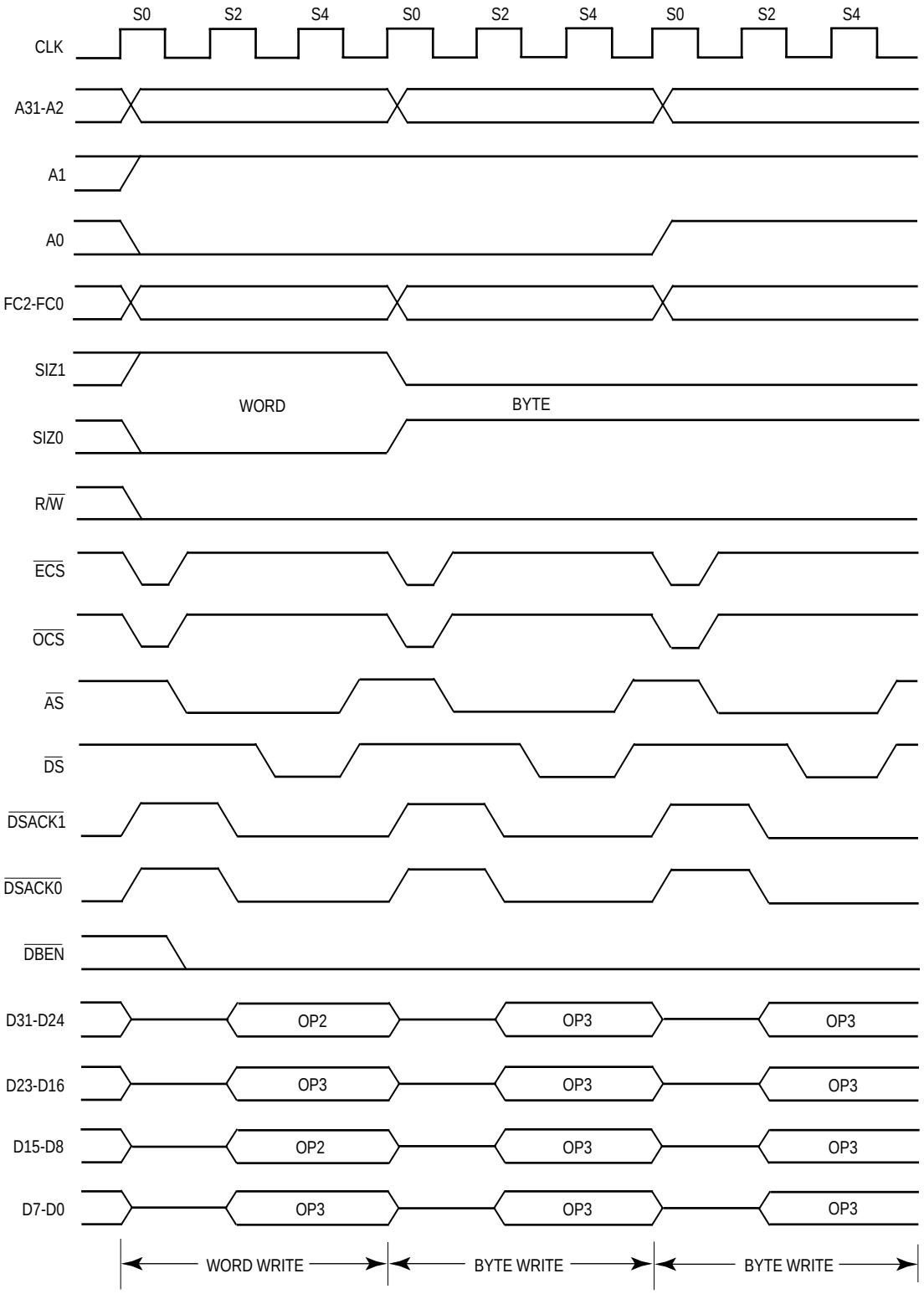


Figure 7-26. Asynchronous Byte and Word Write Cycles — 32-Bit Port

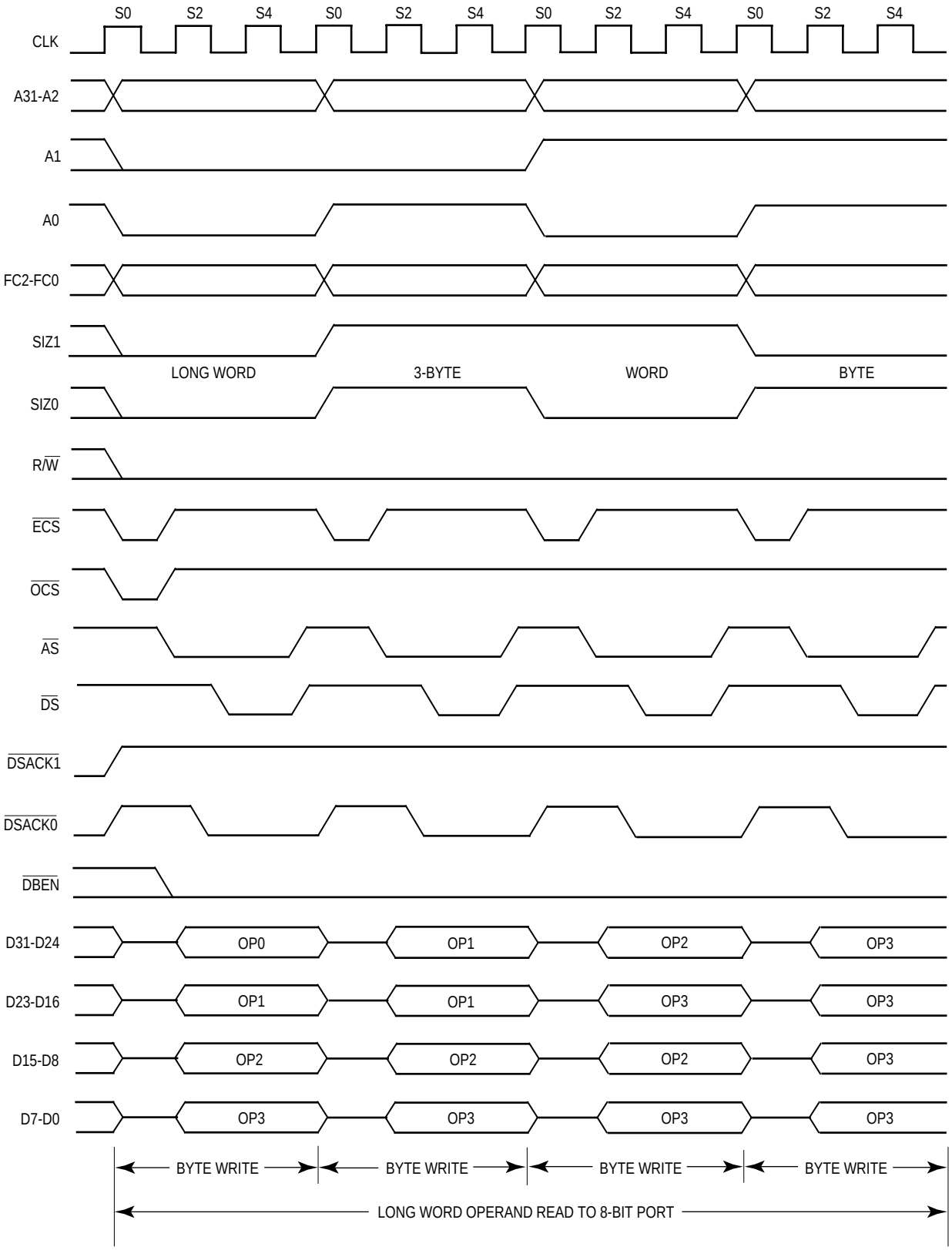


Figure 7-27. Long-Word Operand Write — 8-Bit Port

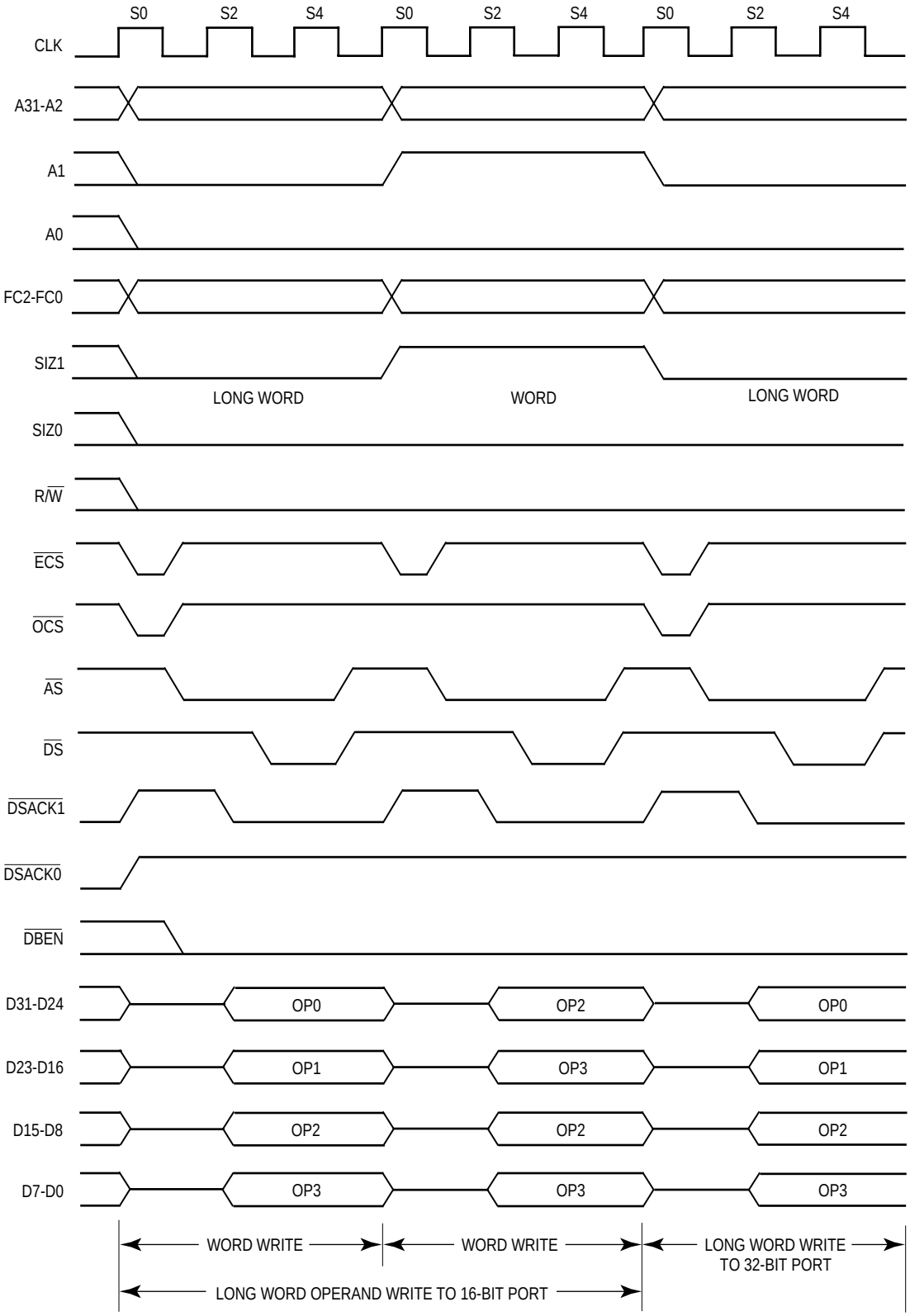


Figure 7-28. Long-Word Operand Write — 16-Bit Port

State 1

One-half clock later in S1, the processor asserts $\overline{AS}$, indicating that the address on the address bus is valid. The processor also asserts $\overline{DBEN}$ during S1, which can enable external data buffers. In addition, the $\overline{ECS}$ (and $\overline{OCS}$, if asserted) signal is negated during S1.

State 2

During S2, the processor places the data to be written onto the D0–D31, and samples $\overline{DSACKx}$ at the end of S2.

State 3

The processor asserts $\overline{DS}$ during S3, indicating that the data is stable on the data bus. As long as at least one of the $\overline{DSACKx}$ signals is recognized by the end of S2 meeting the asynchronous input setup time requirement), the cycle terminates one clock later. If $\overline{DSACKx}$ is not recognized by the start of S3, the processor inserts wait states instead of proceeding to S4 and S5. To ensure that wait states are inserted, both $\overline{DSACK0}$ and $\overline{DSACK1}$ must remain negated throughout the asynchronous input setup and hold times around the end of S2. If wait states are added, the processor continues to sample the $\overline{DSACKx}$ signals on the falling edges of the clock until one is recognized. The selected device uses R/$\overline{W}$, $\overline{DS}$, SIZ0–SIZ1, and A0–A1 to latch data from the appropriate byte(s) of the data bus (D24–D31, D16–D23, D8–D15, and D0–D7). SIZ0–SIZ1 and A0–A1 select the bytes of the data bus. If it has not already done so, the device asserts $\overline{DSACKx}$ to signal that it has successfully stored the data.

State 4

The processor issues no new control signals during S4.

State 5

The processor negates $\overline{AS}$ and $\overline{DS}$ during S5. It holds the address and data valid during S5 to provide address hold time for memory systems. R/$\overline{W}$, SIZ0–SIZ1, FC0–FC2, and $\overline{DBEN}$ also remain valid throughout S5.

The external device must keep $\overline{DSACKx}$ asserted until it detects the negation of $\overline{AS}$ or $\overline{DS}$ (whichever it detects first). The device must negate $\overline{DSACKx}$ within approximately one clock period after sensing the negation of $\overline{AS}$ or $\overline{DS}$. $\overline{DSACKx}$ signals that remain asserted beyond this limit may be prematurely detected for the next bus cycle.

7.3.3 Asynchronous Read-Modify-Write Cycle

The read-modify-write cycle performs a read, conditionally modifies the data in the arithmetic logic unit, and may write the data out to memory. In the MC68030 processor, this operation is indivisible, providing semaphore capabilities for multiprocessor systems. During the entire read-modify-write sequence, the MC68030 asserts the $\overline{\text{RMC}}$ signal to indicate that an indivisible operation is occurring. The MC68030 does not issue a bus grant ($\overline{\text{BG}}$) signal in response to a bus request ($\overline{\text{BR}}$) signal during this operation. The read portion of a read-modify-write operation is forced to miss in the data cache because the data in the cache would not be valid if another processor had altered the value being read. However, read-modify-write cycles may alter the contents of the data cache as described in **6.1.2 Data Cache**.

No burst filling of the data cache occurs during a read-modify-write operation.

The test and set (TAS) and compare and swap (CAS and CAS2) instructions are the only MC68030 instructions that utilize read-modify-write operations. Depending on the compare results of the CAS and CAS2 instructions, the write cycle(s) may not occur. Table search accesses required for the MMU are always read-modify-write cycles to the supervisor data space. During these cycles, a write does not occur unless a descriptor is updated. No data is internally cached for table search accesses since the MMU uses physical addresses to access the tables. Refer to **Section 9 Memory Management Unit** for information about the MMU.

Figure 7-29 is a flowchart of the asynchronous read-modify-write cycle operation. Figure 7-30 is an example of a functional timing diagram of a TAS instruction specified in terms of clock periods.

State 0

The processor asserts $\overline{\text{ECS}}$ and $\overline{\text{OCS}}$ in S0 to indicate the beginning of an external operand cycle. The processor also asserts $\overline{\text{RMC}}$ in S0 to identify a read-modify-write cycle. The processor places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the operation. SIZ0–SIZ1 become valid in S0 to indicate the operand size. The processor drives R/$\overline{\text{W}}$ high for the read cycle and sets 4 according to the value of the MMU CI bit in the address translation descriptor or in the appropriate TTx register.

State 1

One-half clock later in S1, the processor asserts $\overline{\text{AS}}$, indicating that the address on the address bus is valid. The processor asserts $\overline{\text{DS}}$ during S1. In addition, the $\overline{\text{ECS}}$ (and $\overline{\text{OCS}}$, if asserted) signal is negated during S1.

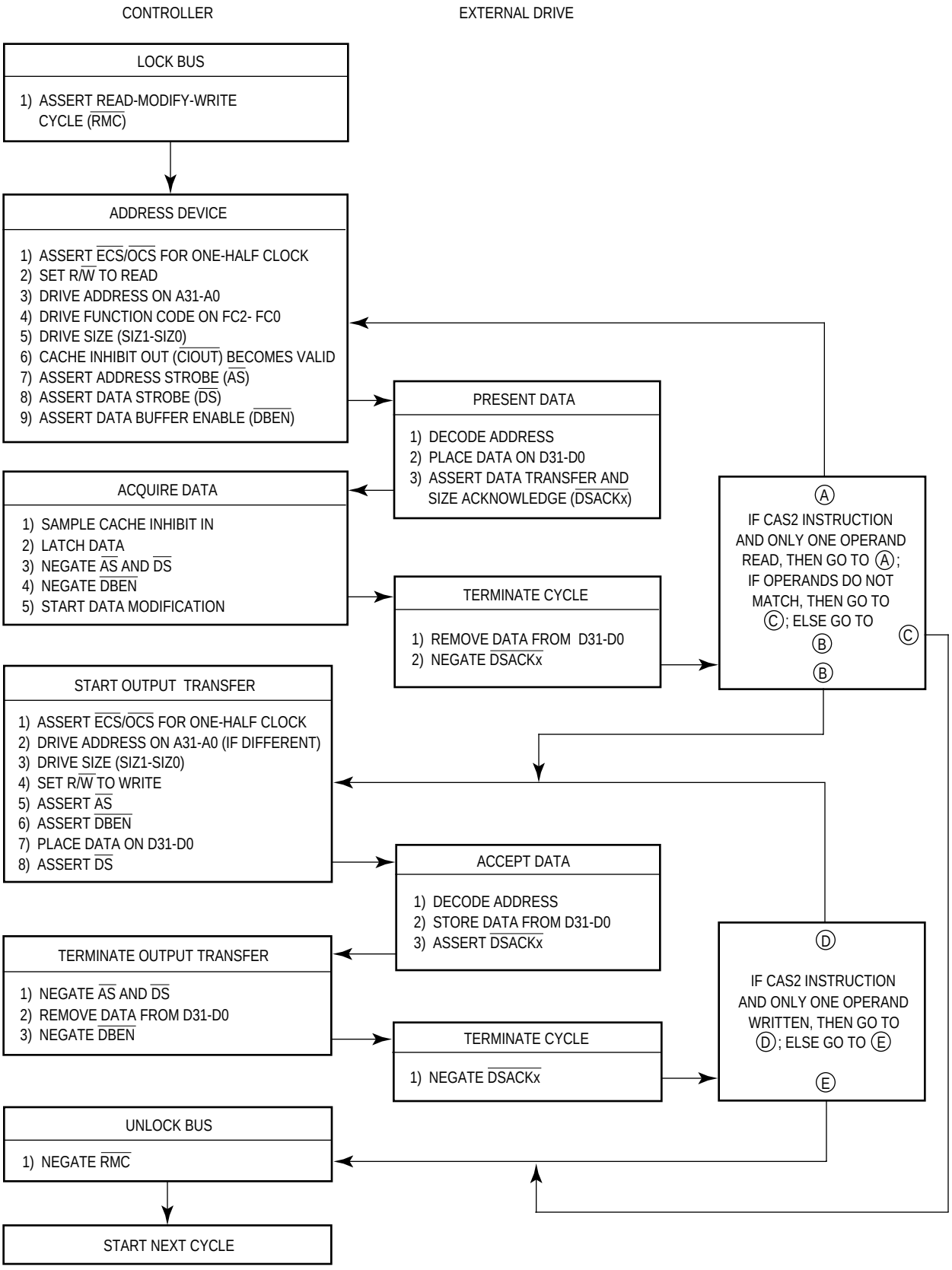


Figure 7-29. Asynchronous Read-Modify-Write Cycle Flowchart

State 2

During state 2 (S2), the processor drives $\overline{\text{DBEN}}$ active to enable external data buffers. The selected device uses R/$\overline{\text{W}}$, SIZ0–SIZ1, A0–A1, and $\overline{\text{DS}}$ to place information on the data bus. Any or all of the bytes (D24–D31, D16–D23, D8–D15, and D0–D7) are selected by SIZ0–SIZ1 and A0–A1. Concurrently, the selected device may assert the $\overline{\text{DSACKx}}$ signals.

State 3

As long as at least one of the $\overline{\text{DSACKx}}$ signals is recognized by the end of S2 (meeting the asynchronous input setup time requirement), data is latched on the next falling edge of the clock, and the cycle terminates. If $\overline{\text{DSACKx}}$ is not recognized by the start of S3, the processor inserts wait states instead of proceeding to S4 and S5. To ensure that wait states are inserted, both $\overline{\text{DSACK0}}$ and $\overline{\text{DSACK1}}$ must remain negated throughout the asynchronous input setup and hold times around the end of S2. If wait states are added, the processor continues to sample the $\overline{\text{DSACKx}}$ signals on the falling edges of the clock until one is recognized.

State 4

The processor samples the level of $\overline{\text{CIIN}}$ at the beginning of S4. At the end of S4, the processor latches the incoming data.

State 5

The processor negates $\overline{\text{AS}}$, $\overline{\text{DS}}$, $\overline{\text{and}}$ $\overline{\text{DBEN}}$ during S5. If more than one read cycle is required to read in the operand(s), S0–S5 are repeated for each read cycle. When finished reading, the processor holds the address, R/$\overline{\text{W}}$, and FC0–FC2 valid in preparation for the write portion of the cycle.

The external device keeps its data and $\overline{\text{DSACKx}}$ signals asserted until it detects the negation of $\overline{\text{AS}}$ or $\overline{\text{DS}}$ (whichever it detects first). The device must remove the data and negate $\overline{\text{DSACKx}}$ within approximately one clock period after sensing the negation of $\overline{\text{AS}}$ or $\overline{\text{DS}}$. $\overline{\text{DSACKx}}$ signals that remain asserted beyond this limit may be prematurely detected for the next portion of the operation.

Idle States

The processor does not assert any new control signals during the idle states, but it may internally begin the modify portion of the cycle at this time. S6-S11 are omitted if no write cycle is required. If a write cycle is required, the R/W signal remains in the read mode until S6 to prevent bus conflicts with the preceding read portion of the cycle; the data bus is not driven until S8.

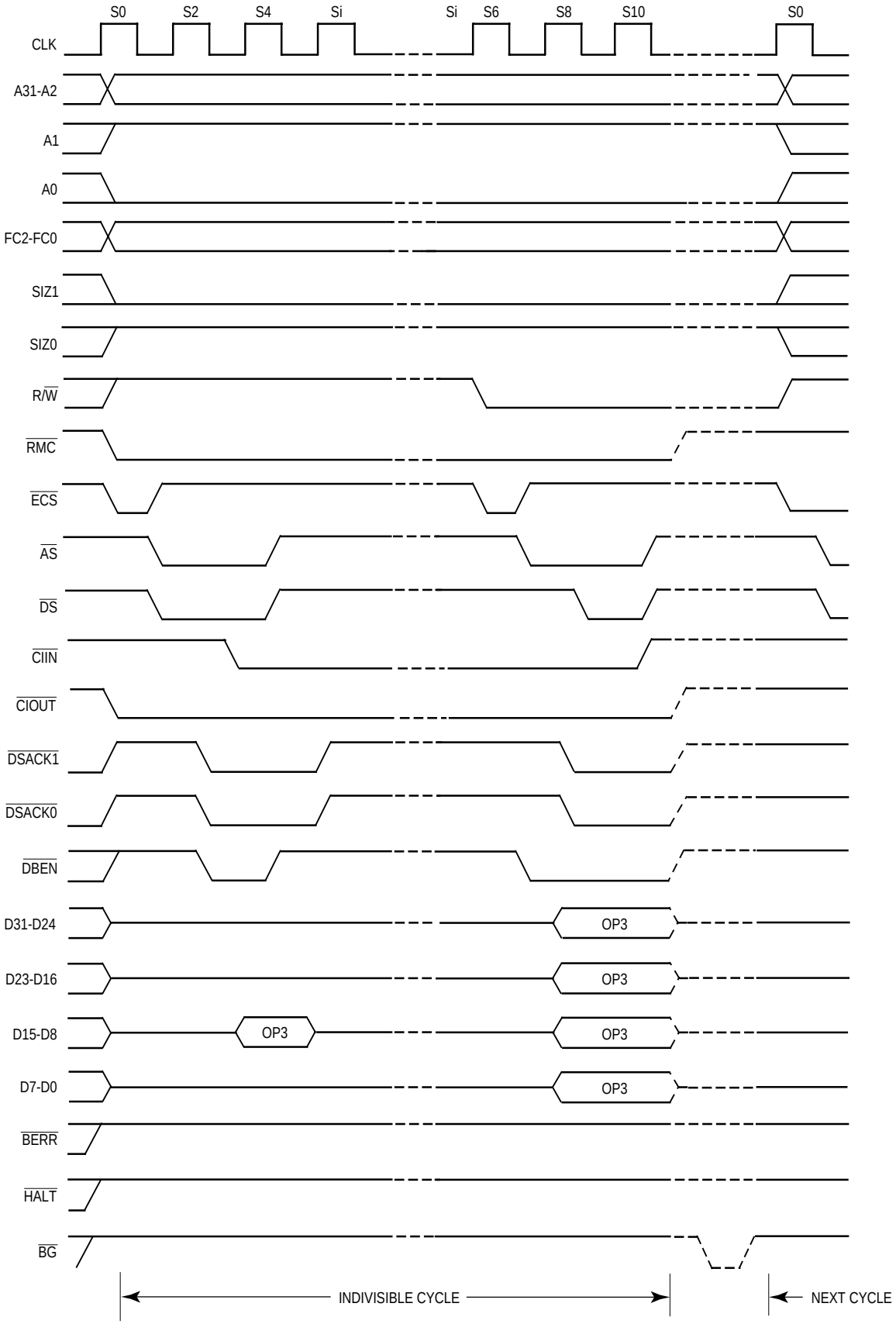


Figure 7-30. Asynchronous Byte Read-Modify-Write Cycle — 32-Bit Port (TAS Instruction with $\overline{\text{CIOUT}}$ or $\overline{\text{CIIN}}$ Asserted)

State 6

The processor asserts $\overline{ECS}$ and $\overline{OCS}$ in S6 to indicate that another external cycle is beginning. The processor drives R/$\overline{W}$ low for a write cycle. $\overline{CIOUT}$ also becomes valid, indicating the state of the MMU CI bit in the address translation descriptor or in a relevant TTx register. Depending on the write operation to be performed, the address lines may change during S6.

State 7

In S7, the processor asserts $\overline{AS}$, indicating that the address on the address bus is valid. The processor also asserts $\overline{DBEN}$, which can be used to enable data buffers during S7. In addition, the $\overline{ECS}$ (and $\overline{OCS}$, if asserted) signal is negated during S7.

State 8

During S8, the processor places the data to be written onto D0–D31.

State 9

The processor asserts $\overline{DS}$ during S9 indicating that the data is stable on the data bus. As long as at least one of the $\overline{DSACKx}$ signals is recognized by the end of S8 (meeting the asynchronous input setup time requirement), the cycle terminates one clock later. If $\overline{DSACKx}$ is not recognized by the start of S9, the processor inserts wait states instead of proceeding to S10 and S11. To ensure that wait states are inserted, both $\overline{DSACK0}$ and $\overline{DSACK1}$ must remain negated throughout the asynchronous input setup and hold times around the end of S8. If wait states are added, the processor continues to sample $\overline{DSACKx}$ signals on the falling edges of the clock until one is recognized.

The selected device uses R/$\overline{W}$, $\overline{DS}$, SIZ0–SIZ1, and A0–A1 to latch data from the appropriate section(s) of the data bus (D24–D31, D16–D23, D8–D15, and D0–D7). SIZ0–SIZ1 and A0–A1 select the data bus sections. If it has not already done so, the device asserts $\overline{DSACKx}$ when it has successfully stored the data.

State 10

The processor issues no new control signals during S10.

State 11

The processor negates $\overline{AS}$ and $\overline{DS}$ during S11. It holds the address and data valid during S11 to provide address hold time for memory systems. R/$\overline{W}$ and FC0–FC2 also remain valid throughout S11.

If more than one write cycle is required, S6-S11 are repeated for each write cycle.

The external device keeps $\overline{DSACKx}$ asserted until it detects the negation of AS or DS (whichever it detects first). The device must remove its data and negate $\overline{DSACKx}$ within approximately one clock period after sensing the negation of $\overline{AS}$ or $\overline{DS}$.

7.3.4 Synchronous Read Cycle

A synchronous read cycle is terminated differently from an asynchronous read cycle; otherwise, the cycles assert and respond to the same signals, in the same sequence. $\overline{STERM}$ rather than $\overline{DSACKx}$ is asserted by the addressed external device to terminate a synchronous read cycle. Since $\overline{STERM}$ must meet the synchronous setup and hold times with respect to all rising edges of the clock while AS is asserted, it does not need to be synchronized by the processor. Only devices with 32-bit ports may assert $\overline{STERM}$. $\overline{STERM}$ is also used with the $\overline{CBREQ}$ and $\overline{CBACK}$ signals during burst mode operation. It provides a two-clock (minimum) bus cycle for 32-bit ports and single-clock (minimum) burst accesses, although wait states can be inserted for these cycles as well. Therefore, a synchronous cycle terminated with $\overline{STERM}$ with one wait cycle is a three-clock bus cycle. However, note that $\overline{STERM}$ is asserted one-half clock later than $\overline{DSACKx}$ would be for a similar asynchronous cycle with zero wait cycles (also three clocks). Thus, if dynamic bus sizing is not needed, $\overline{STERM}$ can be used to provide more decision time in an external cache design than is available with $\overline{DSACKx}$ for three-clock accesses.

Figure 7-31 is a flowchart of a synchronous long-word read cycle. Byte and word operations are similar. Figure 7-32 is a functional timing diagram of a synchronous long-word read cycle.

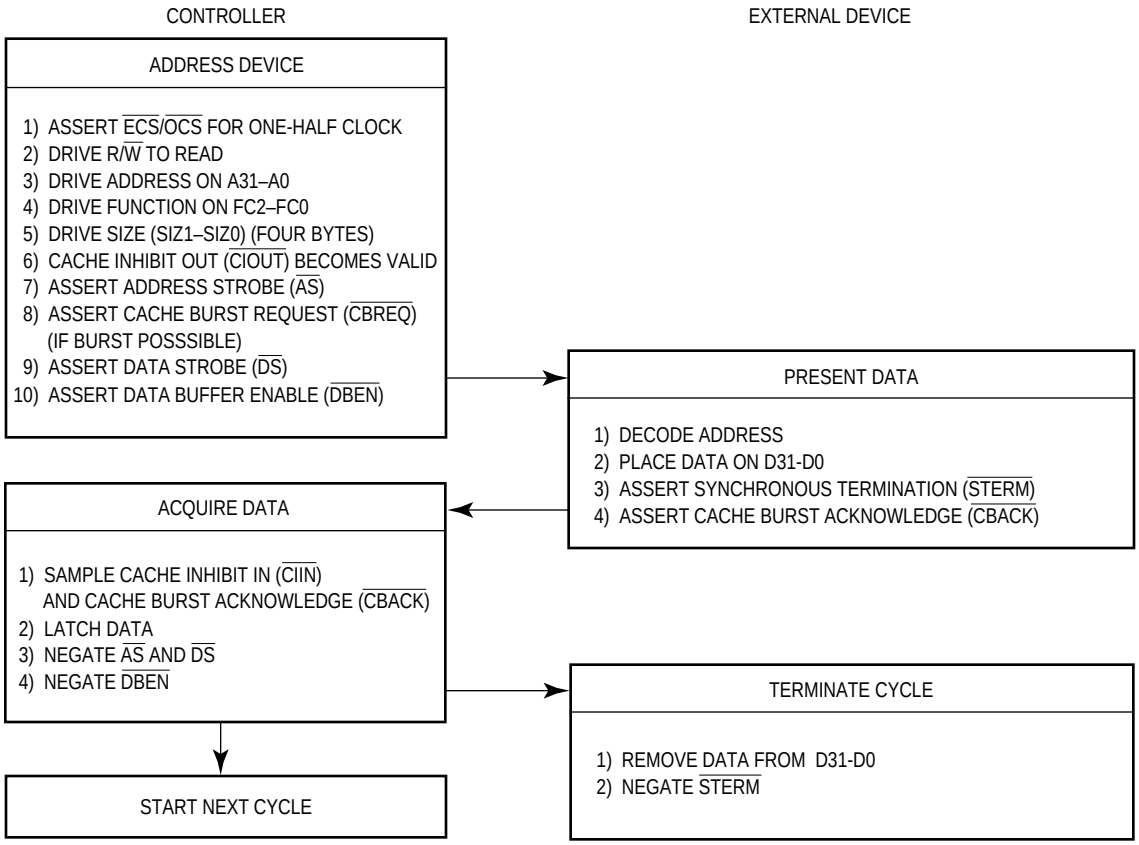


Figure 7-31. Synchronous Long-Word Read Cycle Flowchart — No Burst Allowed

State 0

The read cycle starts with S0. The processor drives $\overline{ECS}$ low, indicating the beginning of an external cycle. When the cycle is the first cycle of a read operand operation, $\overline{OCS}$ is driven low at the same time. During S0, the processor places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the cycle. The processor drives R/$\overline{W}$ high for a read cycle and drives $\overline{DBEN}$ inactive to disable the data buffers. SIZ1-SIZ0 become valid, indicating the number of bytes to be transferred. $\overline{CIOUT}$ also becomes valid, indicating the state of the MMU CI bit in the address translation descriptor or in the appropriate TTx register.

State 1

One-half clock later in S1, the processor asserts $\overline{AS}$, indicating that the address on the address bus is valid. The processor also asserts $\overline{DS}$ during S1. If the burst mode is enabled for the appropriate on-chip cache and all four long words of the cache entry are invalid, (i.e., four long words can be read in), $\overline{CBREQ}$ is asserted. In addition, the $\overline{ECS}$ (and $\overline{OCS}$, if asserted) signal is negated during S1.

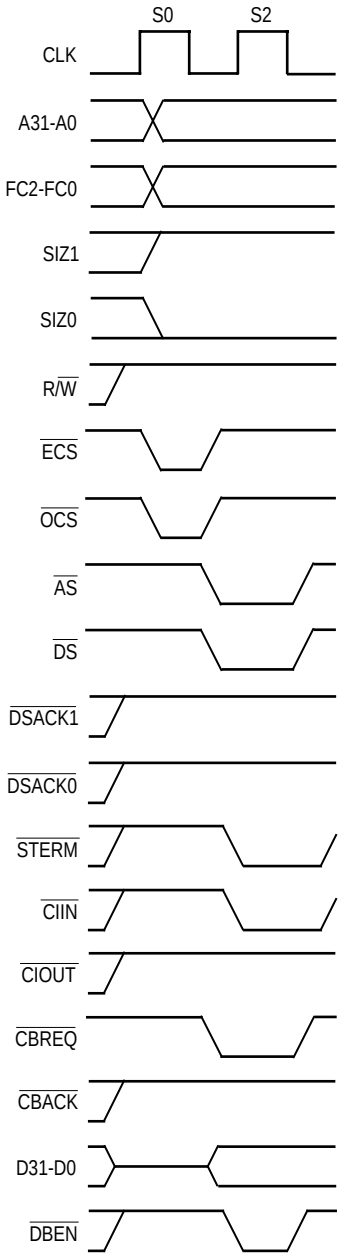


Figure 7-32. Synchronous Read with CIIN Asserted and CBACK Negated

State 2

The selected device uses R/W, SIZ0–SIZ1, A0–A1, and CIOUT to place its information on the data bus. Any or all of the byte sections of the data bus (D24–D31, D16–D23, D8–D15, and D0–D7) are selected by SIZ0–SIZ1 and A0–A1. During S2, the processor drives DBEN active to enable external data buffers. In systems that use two-clock synchronous bus cycles, the timing of DBEN may prevent its use. At the beginning of S2, the processor samples the level of STERM. If STERM is recognized, the processor latches the incoming data at the end of S2. If the selected data is not to be cached for the

current cycle or if the device cannot supply 32 bits, $\overline{\text{CIIN}}$ must be asserted at the same time as $\overline{\text{STERM}}$. In addition, the state of $\overline{\text{CBACK}}$ is latched when $\overline{\text{STERM}}$ is recognized.

Since $\overline{\text{CIIN}}$, $\overline{\text{CBACK}}$, and $\overline{\text{STERM}}$ are synchronous signals, they must meet the synchronous input setup and hold times for all rising edges of the clock while AS is asserted. If $\overline{\text{STERM}}$ is negated at the beginning of S2, wait states are inserted after S2, and $\overline{\text{STERM}}$ is sampled on every rising edge thereafter until it is recognized. Once $\overline{\text{STERM}}$ is recognized, data is latched on the next falling edge of the clock (corresponding to the beginning of S3).

State 3

The processor negates $\overline{\text{AS}}$, $\overline{\text{DS}}$, and $\overline{\text{DBEN}}$ during S3. It holds the address valid during S3 to simplify memory interfaces. R/$\overline{\text{W}}$, SIZ0–SIZ1, and FC0–FC2 also remain valid throughout S3.

The external device must keep its data asserted throughout the synchronous hold time for data from the beginning of S3. The device must remove its data within one clock after asserting $\overline{\text{STERM}}$ and negate $\overline{\text{STERM}}$ within two clocks after asserting $\overline{\text{STERM}}$; otherwise, the processor may inadvertently use $\overline{\text{STERM}}$ for the next bus cycle.

7.3.5 Synchronous Write Cycle

A synchronous write cycle is terminated differently from an asynchronous write cycle and the data strobe may not be useful. Otherwise, the cycles assert and respond to the same signal, in the same sequence. $\overline{\text{STERM}}$ is asserted by the external device to terminate a synchronous write cycle. The discussion of $\overline{\text{STERM}}$ in the preceding section applies to write cycles as well as to read cycles.

$\overline{\text{DS}}$ is not asserted for two-clock synchronous write cycles; therefore, the clock ($\overline{\text{CLK}}$) may be used as the timing signal for latching the data. In addition, there is no time from the latest assertion of $\overline{\text{AS}}$ and the required assertion of $\overline{\text{STERM}}$ for any two-clock synchronous bus cycle. The system must qualify a memory write with the assertion of $\overline{\text{AS}}$ to ensure that the write is not aborted by internal conditions within the MC68030.

Figure 7-33 is a flowchart of a synchronous write cycle. Figure 7-34 is a functional timing diagram of this operation with wait states.

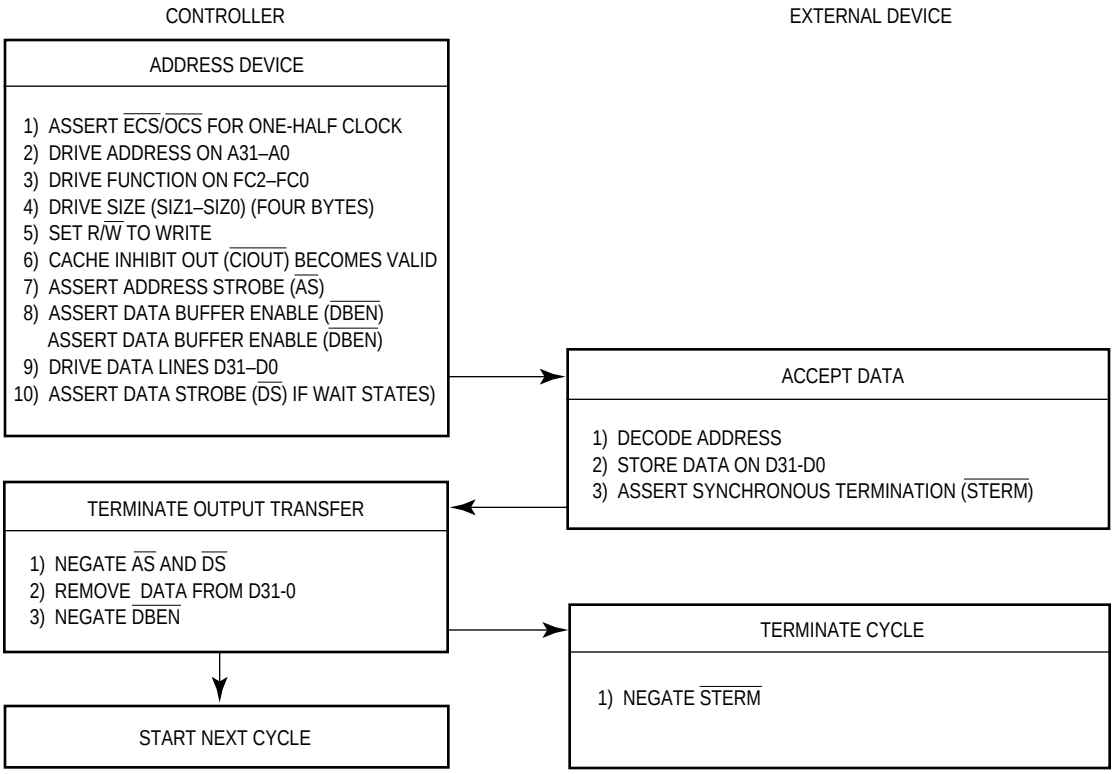


Figure 7-33. Synchronous Write Cycle Flowchart

State 0

The write cycle starts with S0. The processor drives ECS low, indicating the beginning of an external cycle. When the cycle is the first cycle of a write operation, OCS is driven low at the same time. During S0, the processor places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the cycle. The processor drives R/W low for a write cycle. SIZ0–SIZ1 become valid, indicating the number of bytes to be transferred. CIOUT also becomes valid, indicating the state of the MMU CI bit in the address translation descriptor or in the appropriate TTx register.

State 1

One-half clock later in S1, the processor asserts AS, indicating that the address on the address bus is valid. The processor also asserts DBEN during S1, which may be used to enable the external data buffers. In addition, the ECS (and OCS, if asserted) signal is negated during S1.

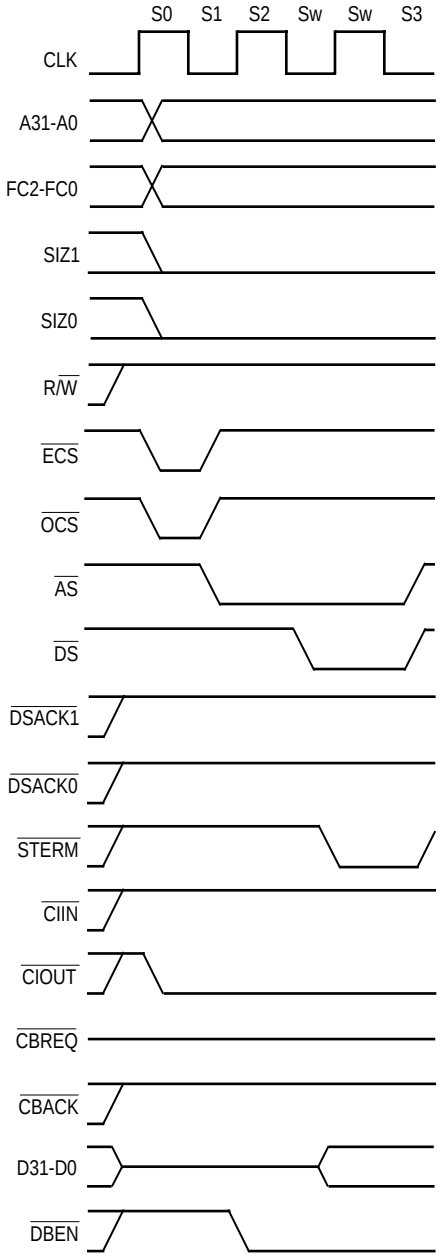


Figure 7-34. Synchronous Write Cycle with Wait States — $\overline{\text{CIOUT}}$ Asserted

State 2

During S2, the processor places the data to be written onto D0–D31. The selected device uses R/$\overline{W}$, CLK, SIZ0–SIZ1, and A0–A1 to latch data from the appropriate section(s) of the data bus (D24–D31, D16–D23, D8–D15, and D0–D7). SIZ0–SIZ1 and A0–A1 select the data bus sections. The device asserts $\overline{\text{STERM}}$ when it has successfully stored the data. If the device does not assert $\overline{\text{STERM}}$ by the rising edge of S2, the processor inserts wait states until it is recognized. The processor asserts $\overline{\text{DS}}$ at the end of S2 if wait states are inserted. For zero-wait-state synchronous write cycles, $\overline{\text{DS}}$ is not asserted.

State 3

The processor negates $\overline{AS}$ (and $\overline{DS}$, if necessary) during S3. It holds the address and data valid during S3 to simplify memory interfaces. R/$\overline{W}$, SIZ0–SIZ1, FC0–FC2, and $\overline{DBEN}$ also remain valid throughout S3.

The addressed device must negate $\overline{STERM}$ within two clock periods after asserting it, or the processor may use $\overline{STERM}$ for the next bus cycle.

7.3.6 Synchronous Read-Modify-Write Cycle

A synchronous read-modify-write operation differs from an asynchronous read-modify-write operation only in the terminating signal of the read and write cycles and in the use of $\overline{CLK}$ instead of $\overline{DS}$ latching data in the write cycle. Like the asynchronous operation, the synchronous read-modify-write operation is indivisible. Although the operation is synchronous, the burst mode is never used during read-modify-write cycles.

Figure 7-35 is a flowchart of the synchronous read-modify-write operation. Timing for the cycle is shown in Figure 7-36.

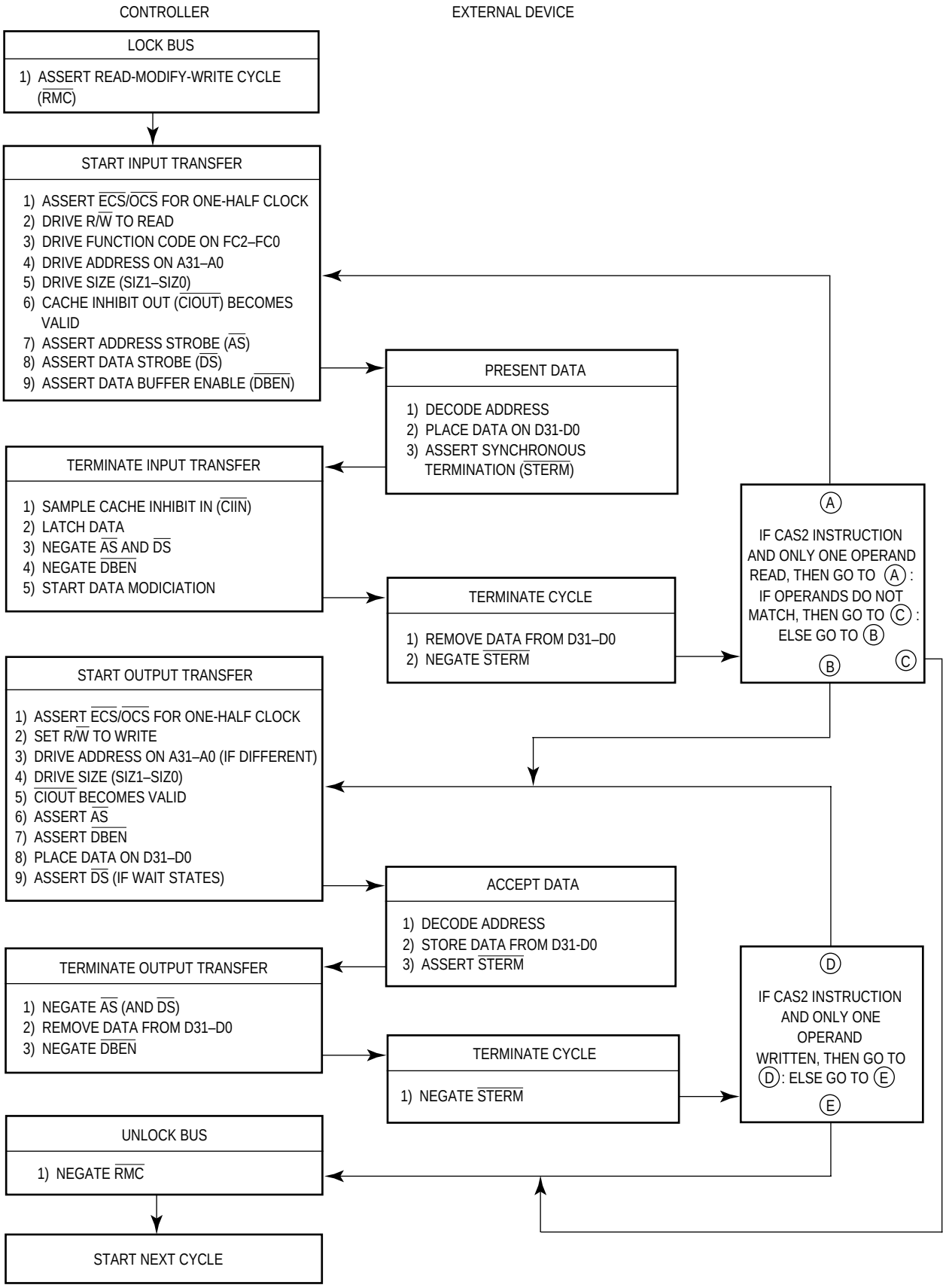


Figure 7-35. Synchronous Read-Modify-Write Cycle Flowchart

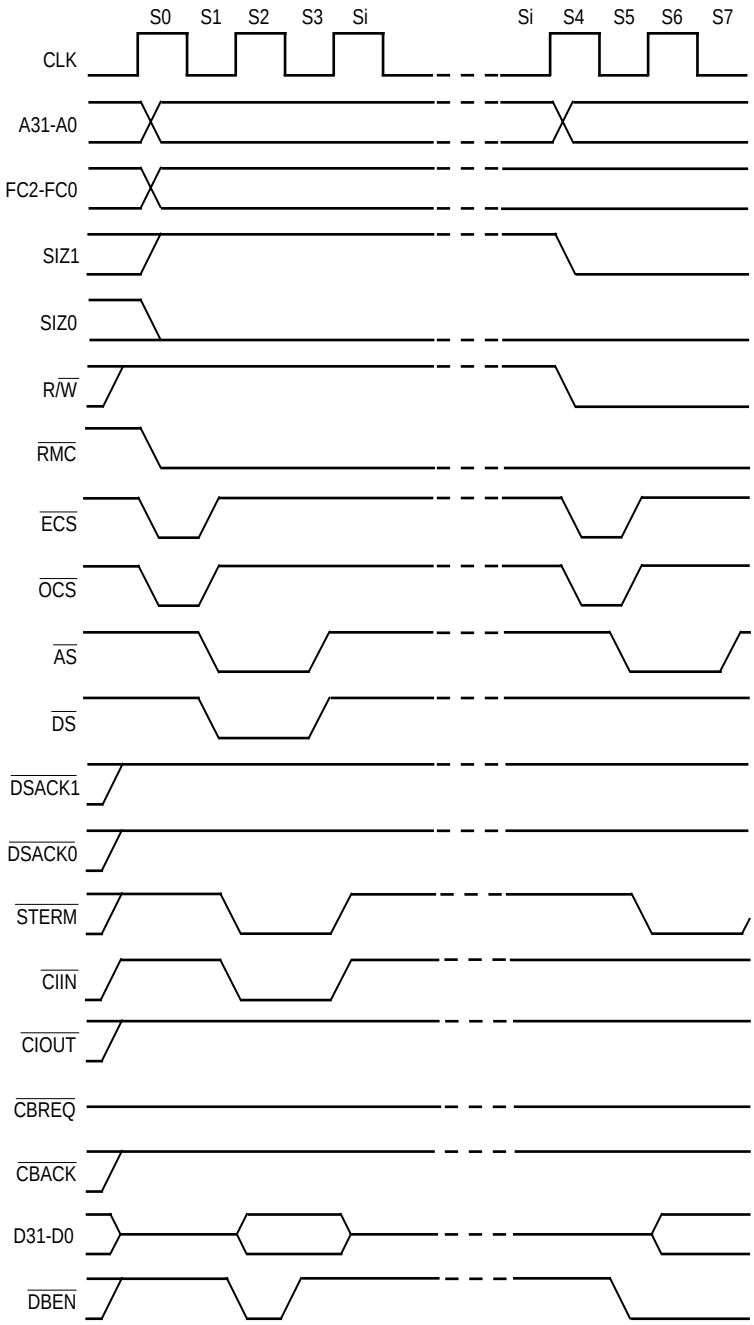


Figure 7-36. Synchronous Read-Modify-Write Cycle Timing — $\overline{\text{CIIN}}$ Asserted

State 0

The processor asserts $\overline{\text{ECS}}$ and $\overline{\text{OCS}}$ in S0 to indicate the beginning of an external operand cycle. The processor also asserts $\overline{\text{RMC}}$ in S0 to identify a read-modify-write cycle. The processor places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the operation. SIZ0–SIZ1 become valid in S0 to indicate the operand size. The processor drives R/$\overline{\text{W}}$ high for a read cycle

and sets CIOUT to the value of the MMU CI bit in the address translation descriptor or in the appropriate TTx register. The processor drives DBEN inactive to disable the data buffers.

State 1

One-half clock later in S1, the processor asserts AS, indicating that the address on the address bus is valid. The processor also asserts DS during S1. In addition, the ECS (and OCS, if asserted) signal is negated during S1.

State 2

The selected device uses R/W, SIZ0–SIZ1, A0–A1, and CIOUT to place its information on the data bus. Any or all of the byte sections (D24–D31, D16–D23, D8–D15, and D0–D7) are selected by SIZ0–SIZ1 and A0–A1. During S2, the processor drives DBEN active to enable external data buffers. In systems that use two-clock synchronous bus cycles, the timing of DBEN may prevent its use. At the beginning of S2, the processor samples the level of STERM. If STERM is recognized, the processor latches the incoming data. If the selected data is not to be cached for the current cycle or if the device cannot supply 32 bits, CIIN must be asserted at the same time as STERM.

Since CIIN and STERM are synchronous signals, they must meet the synchronous nput setup and hold times for all rising edges of the clock while AS is asserted. If STERM is negated at the beginning of S2, wait states are inserted after S2, and STERM is sampled on every rising edge thereafter until it is recognized. Once STERM is recognized, data is latched on the next falling edge of the clock (corresponding to the beginning of S3).

State 3

The processor negates $\overline{AS}$, $\overline{DS}$, and $\overline{DBEN}$ during S3. If more than one read cycle is required to read in the operand(s), S0–S3 are repeated accordingly. When finished with the read cycle, the processor holds the address, R/$\overline{W}$, and FC0–FC2 valid in preparation for the write portion of the cycle.

The external device must keep its data asserted throughout the synchronous hold time for data from the beginning of S3. The device must remove the data within one-clock cycle after asserting $\overline{STERM}$ to avoid bus contention. It must also negate $\overline{STERM}$ within two clocks after asserting $\overline{STERM}$; otherwise, the processor may inadvertently use $\overline{STERM}$ for the next bus cycle.

Idle States

The processor does not assert any new control signals during the idle states, but it may begin the modify portion of the cycle at this time. The R/$\overline{W}$ signal remains in the read mode until S4 to prevent bus conflicts with the preceding read portion of the cycle; the data bus is not driven until S6.

State 4

The processor asserts $\overline{ECS}$ and $\overline{OCS}$ in S4 to indicate that an external cycle is beginning. The processor drives R/W low for a write cycle. $\overline{CIOUT}$ also becomes valid, indicating the state of the MMU CI bit in the address translation descriptor or in the appropriate TTx register. Depending on the write operation to be performed, the address lines may change during S4.

State 5

In state 5 (S5), the processor asserts $\overline{AS}$ to indicate that the address on the address bus is valid. The processor also asserts $\overline{DBEN}$ during S5, which can be used to enable external data buffers.

State 6

During S6, the processor places the data to be written onto the D0–D31.

The selected device uses R/$\overline{W}$, $\overline{CLK}$, SIZ0–SIZ1, and A0–A1 to latch data from the appropriate byte(s) of the data bus (D24–D31, D16–D23, D8–D15, and D0–D7). SIZ0–SIZ1 and A0–A1 select the data bus sections. The device asserts $\overline{STERM}$ when it has successfully stored the data. If the device does not assert $\overline{STERM}$ by the rising edge of S6, the processor inserts wait states until it is recognized. The processor asserts $\overline{DS}$ at the end of S6 if wait states are inserted. Note that for zero-wait-state synchronous write cycles, $\overline{DS}$ is not asserted.

State 7

The processor negates AS (and DS, if necessary) during S7. It holds the address and data valid during S7 to simplify memory interfaces. R/W and FC0–FC2 also remain valid throughout S7.

If more than one write cycle is required, S8-S11 are repeated for each write cycle.

The external device must negate STERM within two clock periods after asserting it, or the processor may inadvertently use STERM for the next bus cycle.

7.3.7 Burst Operation Cycles

The MC68030 supports a burst mode for filling the on-chip instruction and data caches.

The MC68030 provides a set of handshake control signals for the burst mode. When a miss occurs in one of the caches, the MC68030 initiates a bus cycle to obtain the required data or instruction stream fetch. If the data or instruction can be cached, the MC68030 attempts to fill a cache entry. Depending on the alignment for a data access, the MC68030 may attempt to fill two cache entries. The processor may also assert CBREQ to request a burst fill operation. That is, the processor can fill additional entries in the line. The MC68030 allows a burst of as many as four long words.

The mechanism that asserts the CBREQ signal for burstable cache entries is enabled by the data burst enable (DBE) and instruction burst enable (IBE) bits of the cache control register (CACR) for the data and instruction caches, respectively. Either of the following conditions cause the MC68030 to initiate a cache burst request (and assert CBREQ) for a cachable read cycle:

- The logical address and function code signals of the current instruction or data fetch do not match the indexed tag field in the respective instruction or data cache.
- All four long words corresponding to the indexed tag in the appropriate cache are marked invalid.

However, the MC68030 does not assert CBREQ during the first portion of a misaligned access if the remainder of the access does not correspond to the same cache line. Refer to **6.1.3.1 Single Entry Mode** for details.

If the appropriate cache is not enabled or if the cache freeze bit for the cache is set, the processor does not assert $\overline{\text{CBREQ}}$. $\overline{\text{CBREQ}}$ is not asserted during the read or write cycles of any read-modify-write operation.

The MC68030 allows burst filling only from 32-bit ports that terminate bus cycles with STERM and respond to $\overline{\text{CBREQ}}$ by asserting $\overline{\text{CBACK}}$. When the MC68030 recognizes $\overline{\text{STERM}}$ and $\overline{\text{CBACK}}$ and it has asserted $\overline{\text{CBREQ}}$, it maintains $\overline{\text{AS}}$, $\overline{\text{DS}}$, R/$\overline{\text{W}}$, A0–A31, FC0–FC2, SIZ0–SIZ1 in their current state throughout the burst operation. The processor continues to accept data on every clock during which $\overline{\text{STERM}}$ is asserted until the burst is complete or an abnormal termination occurs.

$\overline{\text{CBACK}}$ indicates that the addressed device can respond to a cache burst request by supplying one more long word of data in the burst mode. It can be asserted independently of the $\overline{\text{CBREQ}}$ signal, and burst mode is only initiated if both of these signals are asserted for a synchronous cycle. If the MC68030 executes a full burst operation and fetches four long words, $\overline{\text{CBREQ}}$ is negated after $\overline{\text{STERM}}$ is asserted for the third cycle, indicating that the MC68030 only requests one more long word (the fourth cycle). $\overline{\text{CBACK}}$ can then be negated, and the MC68030 latches the data for the fourth cycle and completes the cache line fill.

The following conditions can abort a burst fill:

- $\overline{\text{CIIN}}$ asserted,
- $\overline{\text{BERR}}$ asserted, or
- $\overline{\text{CBACK}}$ negated prematurely.

The processing of a bus error during a burst fill operation is described in **7.5.1 Bus Errors**.

For the purposes of halting the processor or arbitrating the bus away from the processor with $\overline{\text{BR}}$, a burst operation is a single cycle since AS remains asserted during the entire operation. If the $\overline{\text{HALT}}$ signal is asserted during a burst operation, the processor halts at the end of the operation. Refer to **7.5.3 Halt Operation** for more information about the halt operation. An alternate bus master requesting the bus with $\overline{\text{BR}}$ may become bus master at the end of the operation provided $\overline{\text{BR}}$ is asserted early enough to be internally synchronized before another processor cycle begins. Refer to **7.7 Bus Arbitration** for more information about bus arbitration.

The simultaneous assertion of $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ during a bus cycle normally indicates that the cycle should be retried. However, during the second, third, or fourth cycle of a burst operation, this signal combination indicates a bus error condition, which aborts the burst operation. In addition, the processor remains in the halted state until $\overline{\text{HALT}}$ is negated. For information about bus error processing, refer to **7.5.1 Bus Errors.**

Figure 7-37 is a flowchart of the burst operation. The following timing diagrams show various burst operations. Figure 7-38 shows burst operations for long-word requests with two wait states inserted in the first access and one wait cycle inserted in the subsequent accesses. Figure 7-39 shows a burst operation that fails to complete normally due to $\overline{\text{CBACK}}$ negating prematurely. Figure 7-40 shows a burst operation that is deferred because the entire operand does not correspond to the same cache line. Figure 7-41 shows a burst operation aborted by $\overline{\text{CIIN}}$. Because $\overline{\text{CBACK}}$ corresponds to the next cycle, three long words are transferred even though $\overline{\text{CBACK}}$ is only asserted for two clock periods.

The burst operation sequence begins with states S0–S3, which are very similar to those states for a synchronous read cycle except that $\overline{\text{CBREQ}}$ is asserted. S4-S9 perform the final three reads for a complete burst operation.

State 0

The burst operation starts with S0. The processor drives $\overline{\text{ECS}}$ low, indicating the beginning of an external cycle. When the cycle is the first cycle of a read operation, $\overline{\text{OCS}}$ is driven low at the same time. During S0, the processor places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the cycle. The processor drives R/W high, indicating a read cycle, and drives $\overline{\text{DBEN}}$ inactive to disable the data buffers. SIZ0–SIZ1 become valid, indicating the number of operand bytes to be transferred. $\overline{\text{CIOUT}}$ also becomes valid, indicating the state of the MMU CI bit in the address translation descriptor or in the appropriate TTx register.

State 1

One-half clock later in S1, the processor asserts $\overline{\text{AS}}$ to indicate that the address on the address bus is valid. The processor also asserts $\overline{\text{DS}}$ during S1. $\overline{\text{CBREQ}}$ is also asserted, indicating that the MC68030 can perform a burst operation into one of its caches and can read in four long words. In addition, $\overline{\text{ECS}}$ (and $\overline{\text{OCS}}$, if asserted) is negated during S1.

State 2

The selected device uses R/$\overline{\text{W}}$, SIZ0–SIZ1, A0–A1, and $\overline{\text{CIOUT}}$ to place the data on the data bus. (The first cycle must supply the long word at the corresponding long-word boundary.) All of the byte sections (D24–D31, D16–D23, D8–D15, and D0–D7) of the data bus must be driven since the burst operation latches 32 bits on every cycle. During S2, the processor drives $\overline{\text{DBEN}}$ active to enable external data buffers. In systems that use two-clock synchronous bus cycles, the timing of $\overline{\text{DBEN}}$ may prevent its use. At the beginning of S2, the processor tests the level of $\overline{\text{STERM}}$. If $\overline{\text{STERM}}$ is recognized, the processor latches the incoming data at the end of S2. For the burst operation to proceed, $\overline{\text{CBACK}}$ must be asserted when $\overline{\text{STERM}}$ is recognized. If the data for the current cycle is

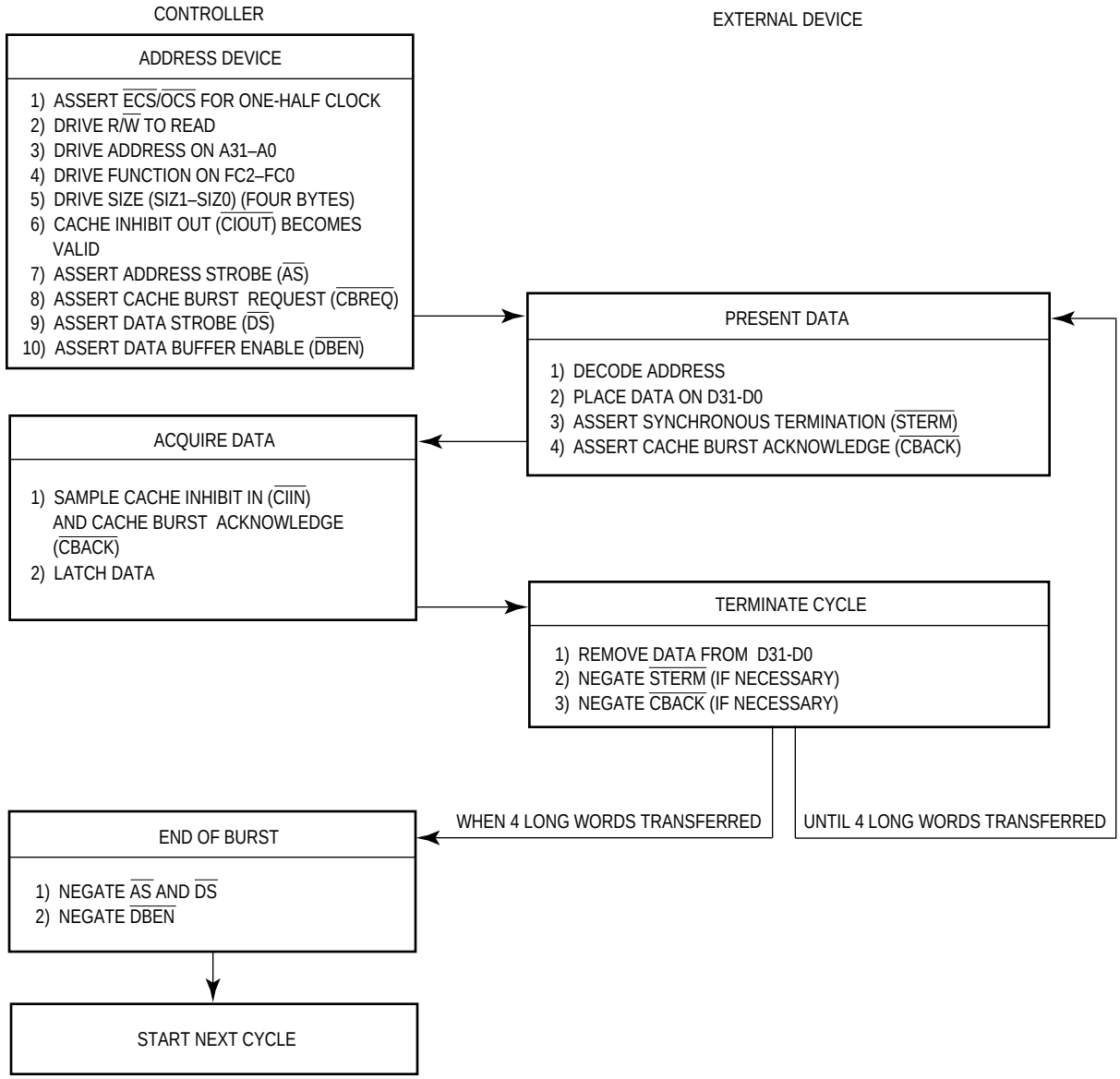


Figure 7-37. Burst Operation Flowchart — Four Long Words Transferred

not to be cached, CIIN must be asserted at the same time as STERM. The assertion of CIIN also has the effect of aborting the burst operation.

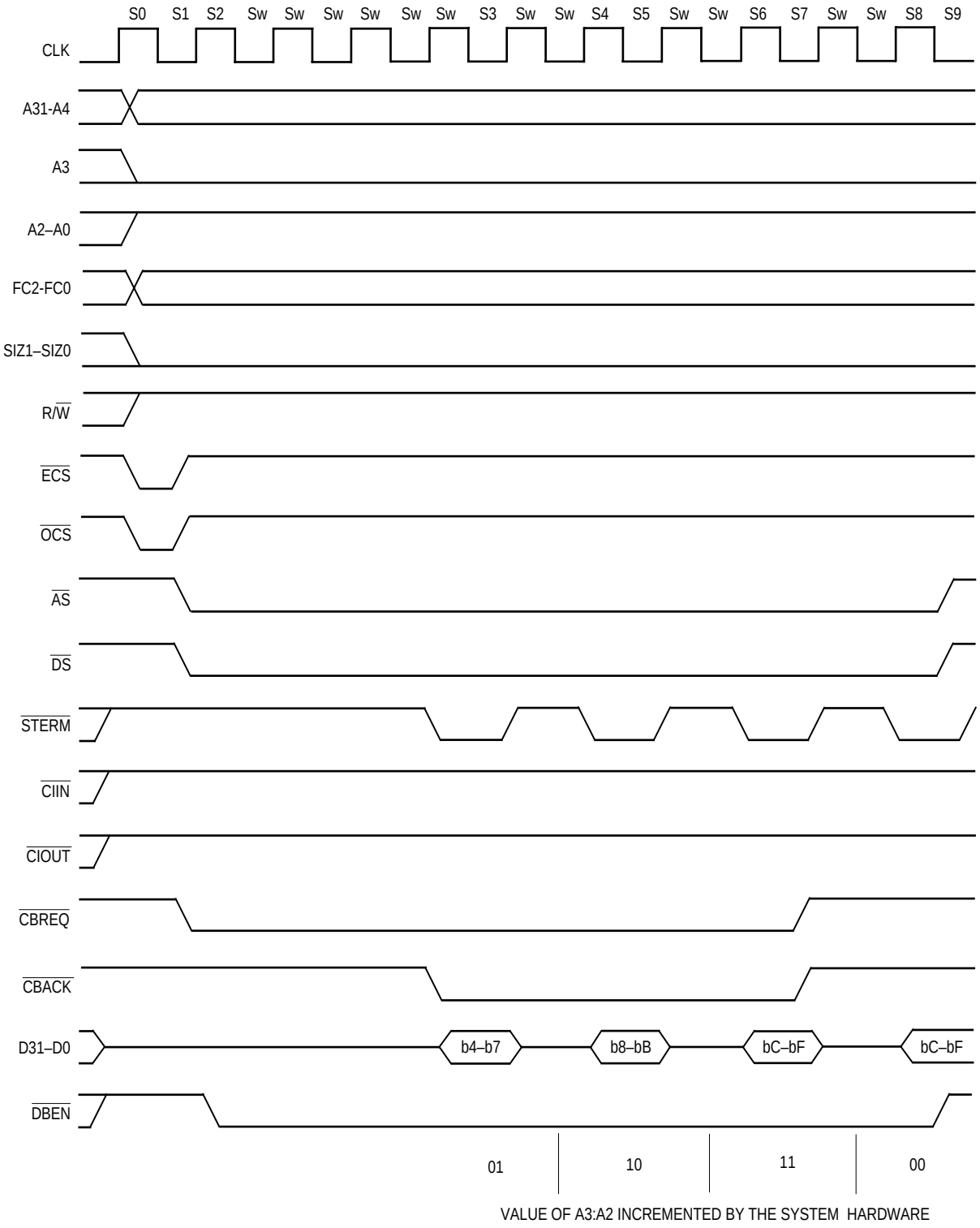
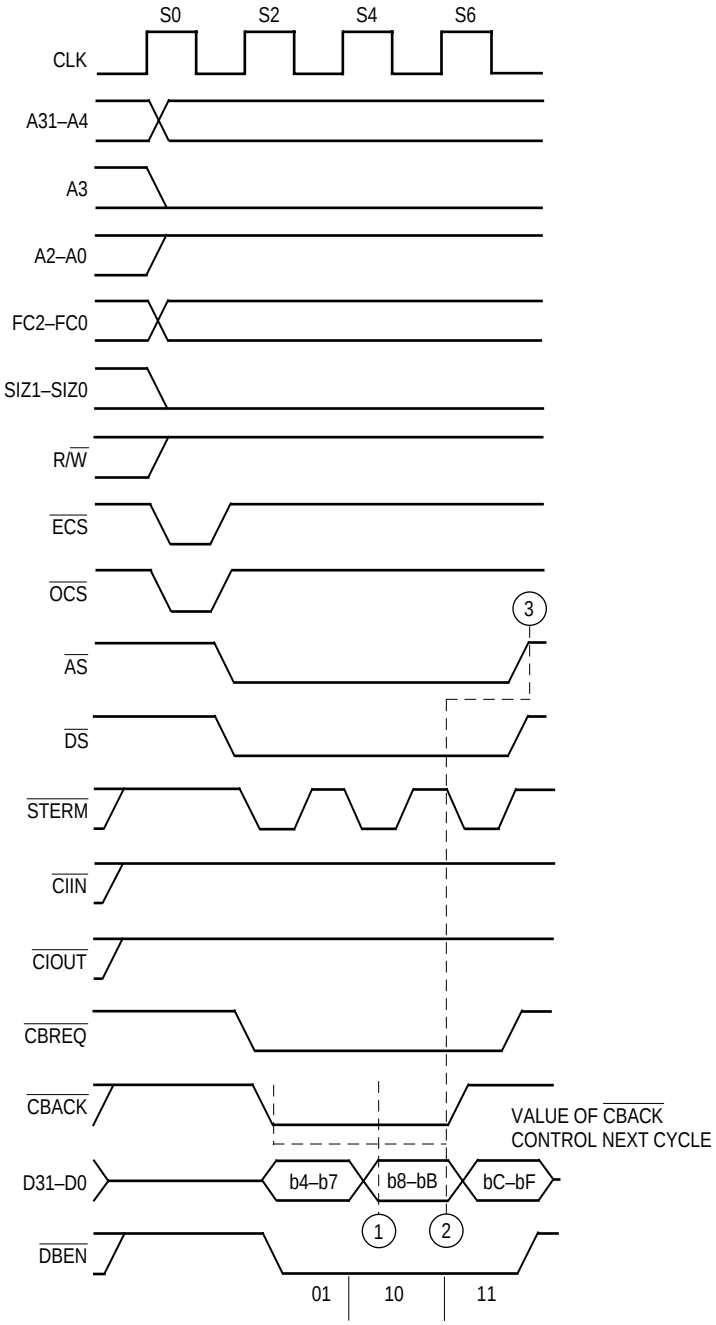


Figure 7-38. Long-Word Operand Request from $07 with Burst Request and Wait Cycle



VALUE OF A3:A2 INCREMENTED BY THE SYSTEM HARDWARE

NOTES:
1. Assertion of CBACK causes data to be placed on D31–D0.
2. Continued assertion of CBACK causes data to be placed on D31–D0.
3. Negation of CBACK causes AS to be negated.

Figure 7-39. Long-Word Operand Request from $07 with Burst Request — CBACK Negated Early

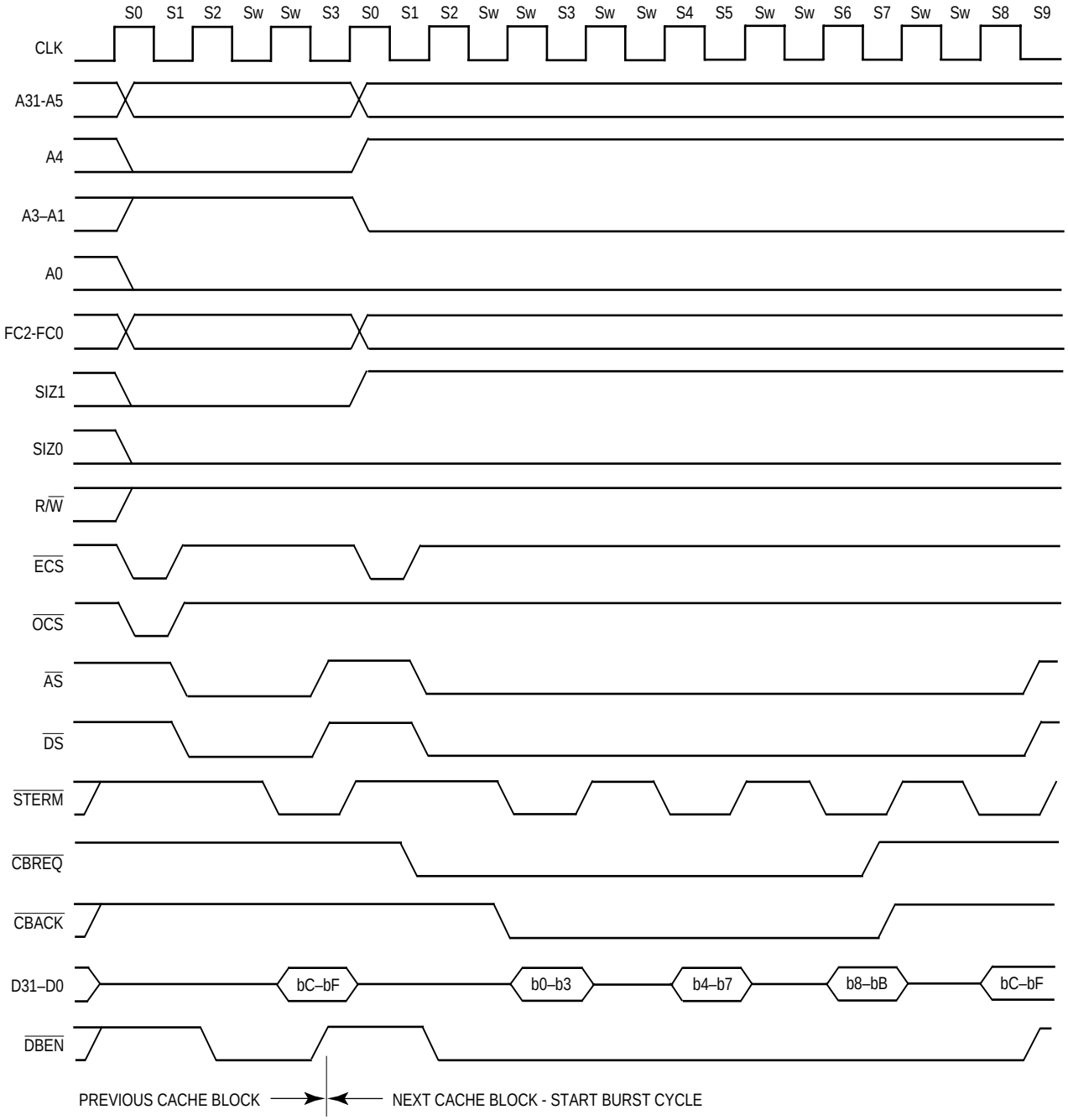


Figure 7-40. Long-Word Operand Request from $0E — Burst Fill Deferred

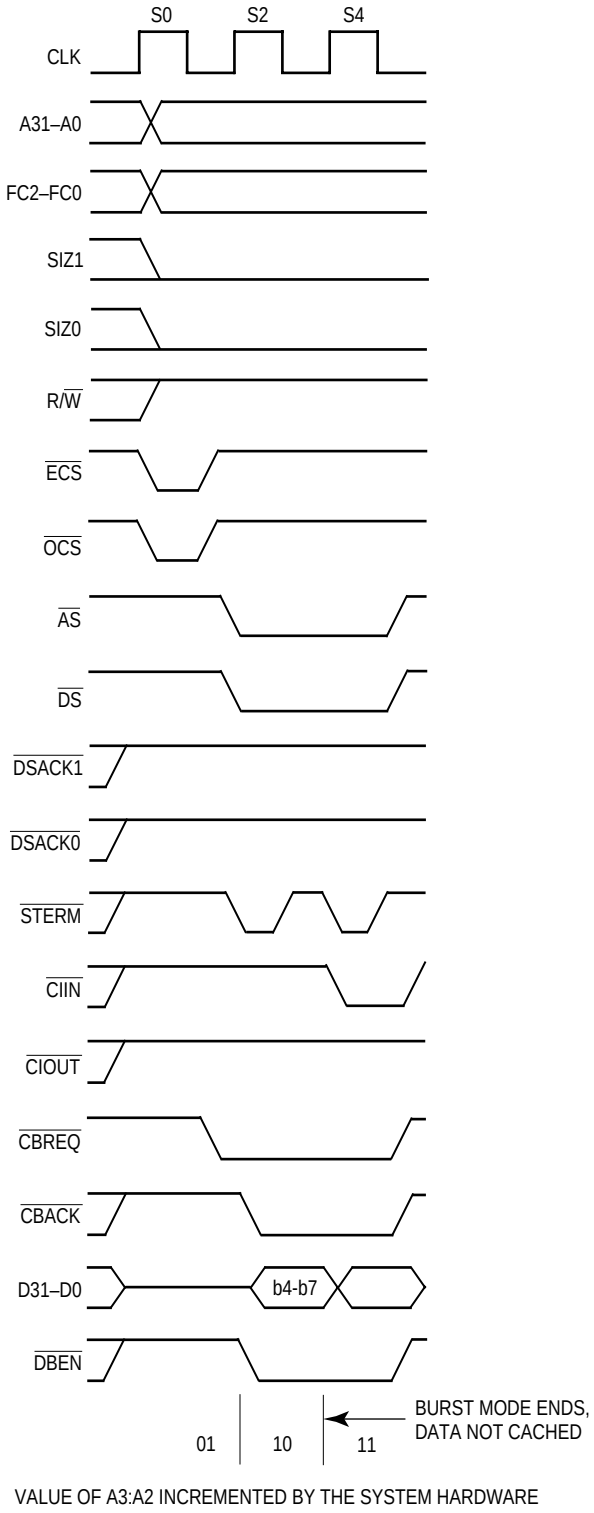


Figure 7-41. Long-Word Operand Request from $07 with Burst Request — CBACK and CIIN Asserted

Since $\overline{\text{CIIN}}$, $\overline{\text{CBACK}}$, and $\overline{\text{STERM}}$ are synchronous signals, they must meet the synchronous input setup and hold times for all rising edges of the clock while AS is asserted. If $\overline{\text{STERM}}$ is negated at the beginning of S2, wait states are inserted after S2, and $\overline{\text{STERM}}$ is sampled on every rising edge of the clock thereafter until it is recognized. Once $\overline{\text{STERM}}$ is recognized, data is latched on the next falling edge of the clock (corresponding to the beginning of S3).

State 3

The processor maintains $\overline{\text{AS}}$, $\overline{\text{DS}}$, and $\overline{\text{DBEN}}$ asserted during S3. It also holds the address valid during S3 for continuation of the burst. R/$\overline{\text{W}}$, SIZ0–SIZ1, and FC0–FC2 also remain valid throughout S3.

The external device must keep the data driven throughout the synchronous hold time for data from the beginning of S3. The device must negate $\overline{\text{STERM}}$ within one clock after asserting $\overline{\text{STERM}}$; otherwise, the processor may inadvertently use $\overline{\text{STERM}}$ prematurely for the next burst access. $\overline{\text{STERM}}$ need not be negated if subsequent accesses do not require wait cycles.

State 4

At the beginning of S4, the processor tests the level of $\overline{\text{STERM}}$. This state signifies the beginning of burst mode, and the remaining states correspond to burst fill cycles. If $\overline{\text{STERM}}$ is recognized, the processor latches the incoming data at the end of S4. This data corresponds to the second long word of the burst. If $\overline{\text{STERM}}$ is negated at the beginning of S4, wait states are inserted instead of S4 and S5, and $\overline{\text{STERM}}$ is sampled on every rising edge of the clock thereafter until it is recognized. As for synchronous cycles, the states of $\overline{\text{CBACK}}$ and $\overline{\text{CIIN}}$ are latched at the time $\overline{\text{STERM}}$ is recognized. The assertion of $\overline{\text{CBACK}}$ at this time indicates that the burst operation should continue, and the assertion of $\overline{\text{CIIN}}$ indicates that the data latched at the end of S4 should not be cached and that the burst should abort.

State 5

The processor maintains all the signals on the bus driven throughout S5 for continuation of the burst. The same hold times for $\overline{\text{STERM}}$ and data described for S3 apply here.

State 6

This state is identical to S4 except that once $\overline{\text{STERM}}$ is recognized, the third long wordof data for the burst is latched at the end of S6.

State 7

During this state, the processor negates $\overline{\text{CBREQ}}$, and the memory device may negate $\overline{\text{CBACK}}$. Aside from this, all other bus signals driven by the processor remain driven. The same hold times for $\overline{\text{STERM}}$ and data described for S3 apply here.

State 8

This state is identical to S4 except that $\overline{\text{CBREQ}}$ is negated, indicating that the processor cannot continue to accept more data after this. The data latched at the end of S8 corresponds to the fourth long word of the burst.

State 9

The processor negates $\overline{\text{AS}}$, $\overline{\text{DS}}$, and $\overline{\text{DBEN}}$ during S9. It holds the address, R/$\overline{\text{W}}$, SIZ0–SIZ1, and FC0–FC2 valid throughout S9. The same hold times for data described for S3 apply here.

Note that the address bus of the MC68030 remains driven to a constant value for the duration of a burst transfer operation (including the first transfer before burst mode is entered). If an external memory system requires incrementing of the long-word base address to supply successive long words of information, this function must be performed by external hardware. Additionally, in the case of burst transfers that cross a 16-byte boundary (i.e., the first long word transferred is not located at A3/A2=00), the external hardware must correctly control the continuation or termination of the burst transfer as desired. The burst may be terminated by negating CBACK during the transfer of the most significant long word of the 16-byte image (A3/A2=11) or may be continued (with $\overline{\text{CBACK}}$ asserted) by providing the long word located at A3/A2=00 (i.e., the count sequence wraps back to zero and continues as necessary). The MC68030 caches assume the higher order address lines (A4-A31) remain unchanged as the long-word accesses wrap back around to A3/A2=00.

7.4 CPU SPACE CYCLES

FC0–FC2 select user and supervisor program and data areas as listed in Table 4-1. The area selected by FC0–FC2=$7 is classified as the CPU space. The interrupt acknowledge, breakpoint acknowledge, and coprocessor communication cycles described in the following sections utilize CPU space.

The CPU space type is encoded on A16-A19 during a CPU space operation and indicates the function that the processor is performing. On the MC68030, three of the encodings are implemented as shown in Figure 7-42. All unused values are reserved by Motorola for future additional CPU space types.

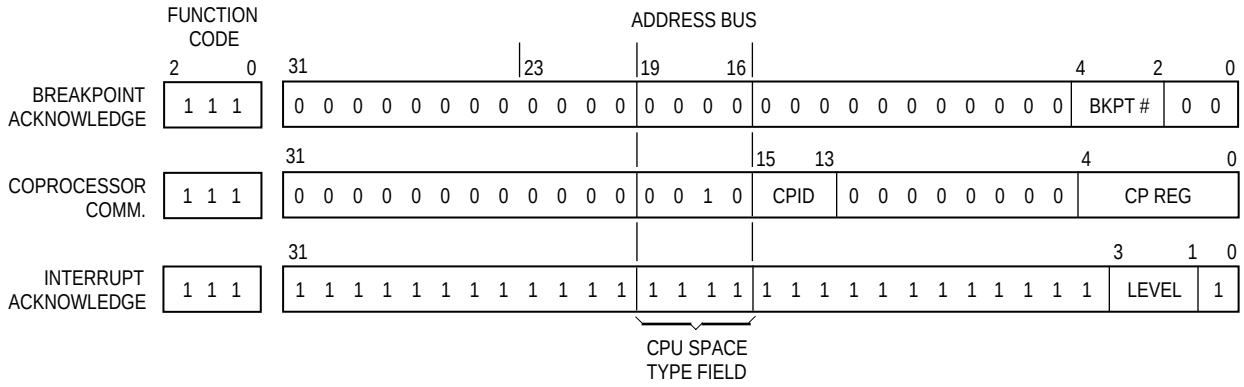


Figure 7-42. MC68030 CPU Space Address Encoding

7.4.1 Interrupt Acknowledge Bus Cycles

When a peripheral device signals the processor (with the IPL0–IPL2 signals) that the device requires service, and the internally synchronized value on these signals indicates a higher priority than the interrupt mask in the status register (or that a transition has occurred in the case of a level 7 interrupt), the processor makes the interrupt a pending interrupt. Refer to **8.1.9 Interrupt Exceptions** for details on the recognition of interrupts.

The MC68030 takes an interrupt exception for a pending interrupt within one instruction boundary (after processing any other pending exception with a higher priority). The following paragraphs describe the various kinds of interrupt acknowledge bus cycles that can be executed as part of interrupt exception processing.

7.4.1.1 INTERRUPT ACKNOWLEDGE CYCLE — TERMINATED NORMALLY. When the MC68030 processes an interrupt exception, it performs an interrupt acknowledge cycle to obtain the number of the vector that contains the starting location of the interrupt service routine.

Some interrupting devices have programmable vector registers that contain the interrupt vectors for the routines they use. The following paragraphs describe the interrupt acknowledge cycle for these devices. Other interrupting conditions or devices cannot supply a vector number and use the autovector cycle described in **7.4.1.2 Autovector Interrupt Acknowledge Cycle**.

The interrupt acknowledge cycle is a read cycle. It differs from the asynchronous read cycle described in **7.3.1 Asynchronous Read Cycle** or the synchronous read cycle described in **7.3.4 Synchronous Read Cycle** in that it accesses the CPU address space. Specifically, the differences are:

1. FC0–FC2 are set to seven (FC0/FC1/FC2=111) for CPU address space.
2. A1, A2, and A3 are set to the interrupt request level (the inverted values of IPL0, iPL1, and IPL2, respectively).
3. The CPU space type field (A16-A19) is set to $F, the interrupt acknowledge code.
4. A20–A31, A4–A15, and A0 are set to one.

The responding device places the vector number on the data bus during the interrupt acknowledge cycle. Beyond this, the cycle is terminated normally with either $\overline{\text{STERM}}$ or $\overline{\text{DSACK}}$x. Figure 7-43 is the flowchart of the interrupt acknowledge cycle.

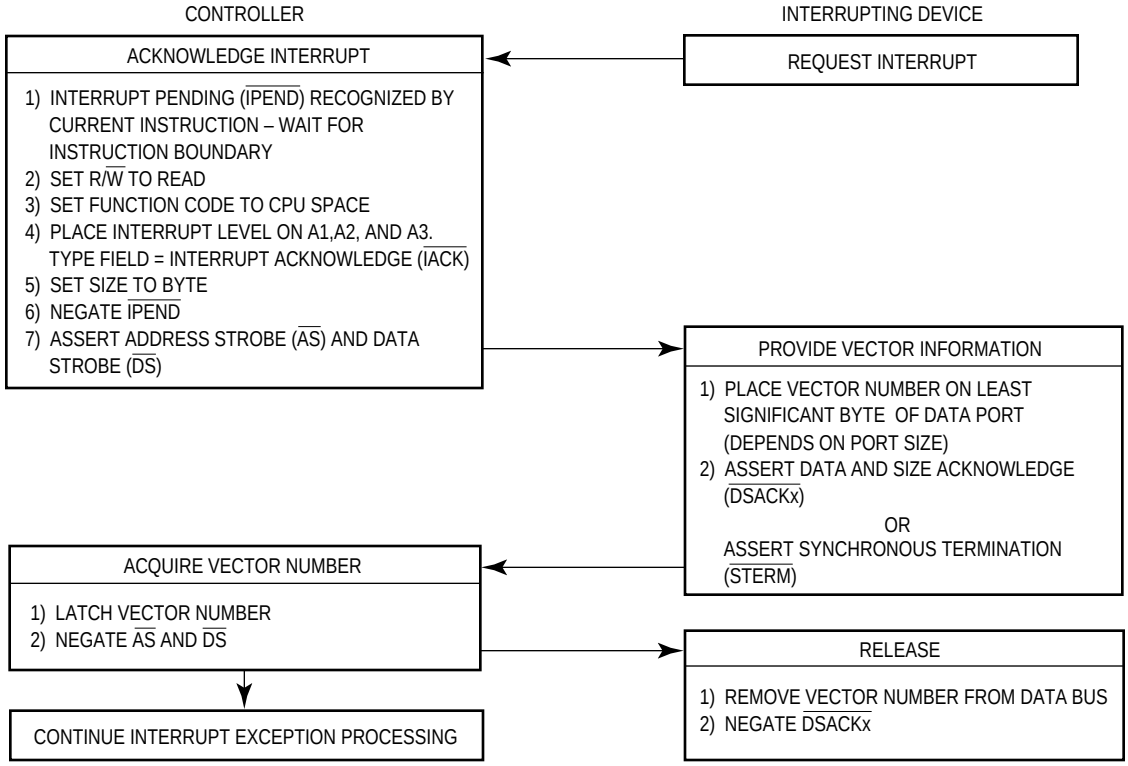


Figure 7-43. Interrupt Acknowledge Cycle Flowchart

Figure 7-44 shows the timing for an interrupt acknowledge cycle terminated with $\overline{\text{DSACK}}$x.

7.4.1.2 AUTOVECTOR INTERRUPT ACKNOWLEDGE CYCLE. When the interrupting device cannot supply a vector number, it requests an automatically generated vector or autovector. Instead of placing a vector number on the data bus and asserting $\overline{\text{DSACK}}$x or $\overline{\text{STERM}}$, the device asserts the autovector signal ($\overline{\text{AVEC}}$) to terminate the cycle. Neither $\overline{\text{STERM}}$ nor $\overline{\text{DSACK}}$x may be asserted during an interrupt acknowledge cycle terminated by $\overline{\text{AVEC}}$.

The vector number supplied in an autovector operation is derived from the interrupt level of the current interrupt. When $\overline{\text{AVEC}}$ is asserted instead of $\overline{\text{DSACK}}$ or $\overline{\text{STERM}}$ during an interrupt acknowledge cycle, the MC68030 ignores the state of the data bus and internally generates the vector number, the sum of the interrupt level plus 24 ($18). There are seven distinct autovectors that can be used, corresponding to the seven levels of interrupt available with signals $\overline{\text{IPL0}}$–$\overline{\text{IPL2}}$. Figure 7-45 shows the timing for an autovector operation.

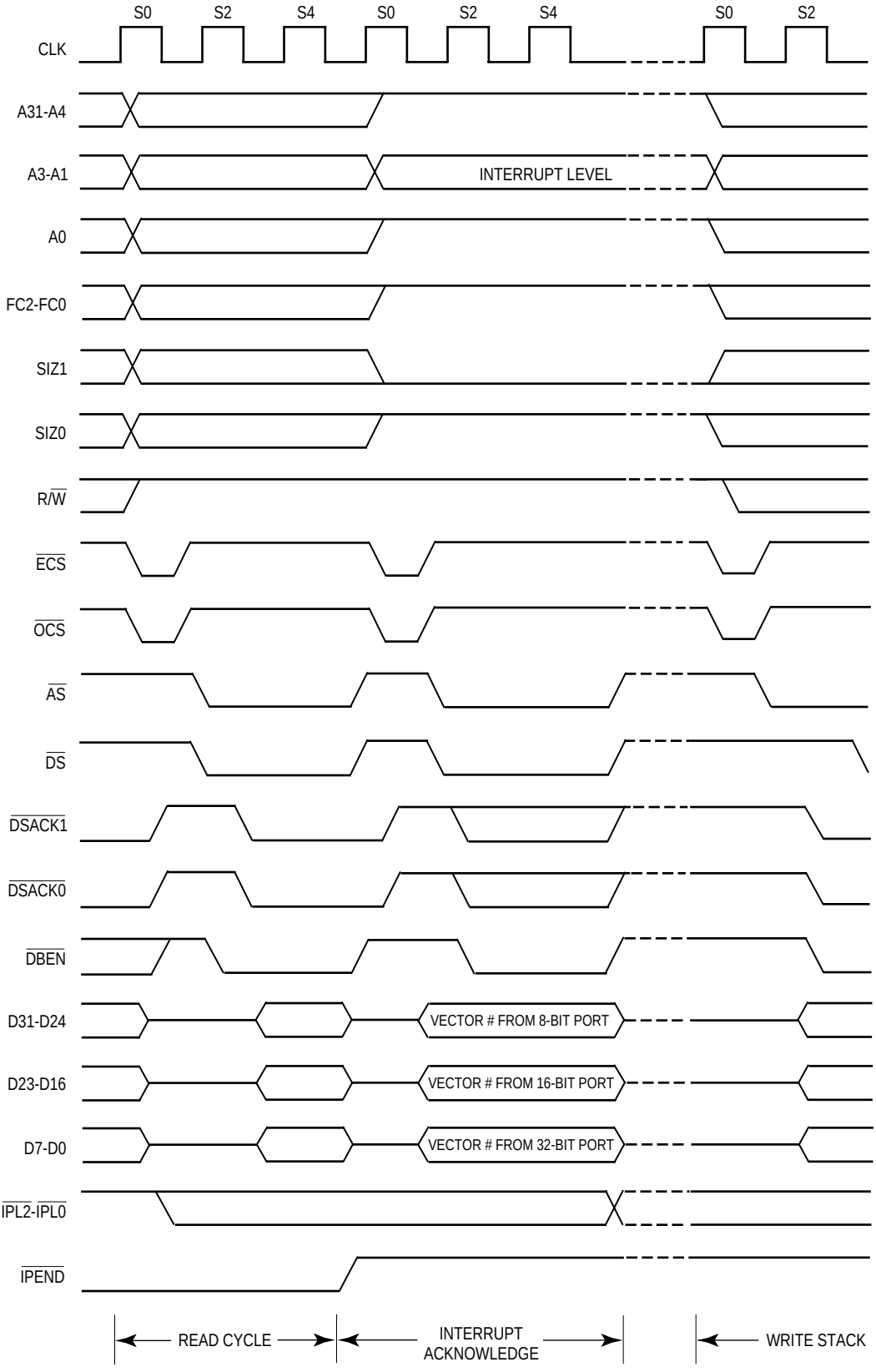


Figure 7-44. Interrupt Acknowledge Cycle Timing

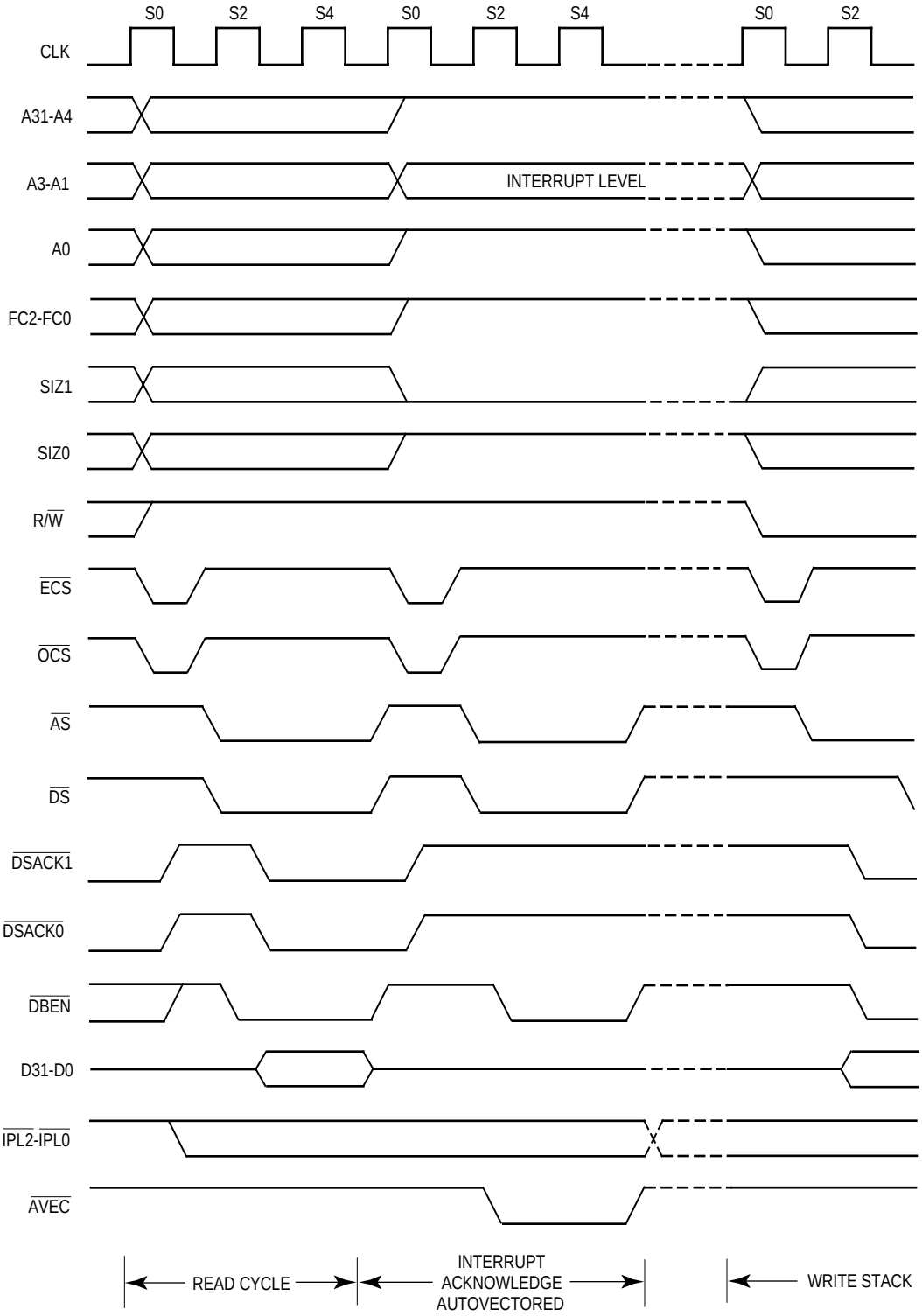


Figure 7-45. Autovector Operation Timing

7.4.1.3 SPURIOUS INTERRUPT CYCLE. When a device does not respond to an interrupt acknowledge cycle with $\overline{\text{AVEC}}$, $\overline{\text{STERM}}$, or $\overline{\text{DSACK}}$x, the external logic typically returns $\overline{\text{BERR}}$. The MC68030 automatically generates the spurious interrupt vector number, 24, instead of the interrupt vector number in this case. If $\overline{\text{HALT}}$ is also asserted, the processor retries the cycle.

7.4.2 Breakpoint Acknowledge Cycle

The breakpoint acknowledge cycle is generated by the execution of a breakpoint instruction ($\overline{\text{BKPT}}$). The breakpoint acknowledge cycle allows the external hardware to provide an instruction word directly into the instruction pipeline as the program executes. This cycle accesses the CPU space with a type field of zero and provides the breakpoint number specified by the instruction on address lines A2–A4. If the external hardware terminates the cycle with $\overline{\text{DSACK}}$x or $\overline{\text{STERM}}$, the data on the bus (an instruction word) is inserted into the instruction pipe, replacing the breakpoint opcode, and is executed after the breakpoint acknowledge cycle completes. The breakpoint instruction requires a word to be transferred so that if the first bus cycle accesses an 8-bit port, a second cycle is required. If the external logic terminates the breakpoint acknowledge cycle with $\overline{\text{BERR}}$ (i.e., no instruction word available), the processor takes an illegal instruction exception. Figure 7-46 is a flowchart of the breakpoint acknowledge cycle. Figure 7-47 shows the timing for a breakpoint acknowledge cycle that returns an instruction word. Figure 7-48 shows the timing for a breakpoint acknowledge cycle that signals an exception.

7.4.3 Coprocessor Communication Cycles

The MC68030 coprocessor interface provides instruction-oriented communication between the processor and as many as seven coprocessors. The bus communication required to support coprocessor operations uses the MC68030 CPU space with a type field of $2.

Coprocessor accesses use the MC68030 bus protocol except that the address bus supplies access information rather than a 32-bit address. The CPU space type field (A16-A19) for a coprocessor operation is $2. A13-A15 contain the coprocessor identification number (CpID), and A0–A4 specify the coprocessor interface register to be accessed. Coprocessor accesses to a CpID of zero correspond to MMU instructions and are not generated by the MC68030 as a result of the coprocessor interface. These cycles can only be generated by the MOVES instruction. Refer to **Section 10 Coprocessor Interface Description** for further information.

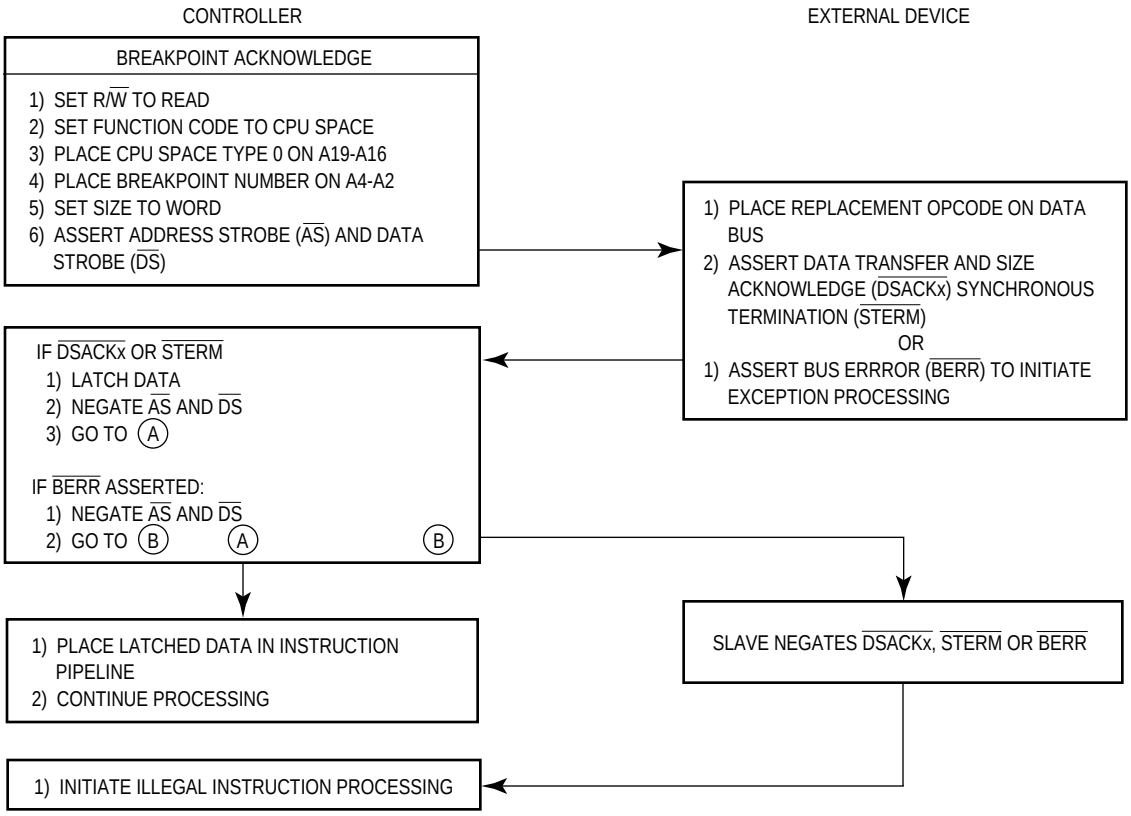


Figure 7-46. Breakpoint Operation Flow

7.5 BUS EXCEPTION CONTROL CYCLES

The MC68030 bus architecture requires assertion of either DSACKx or STERM from an external device to signal that a bus cycle is complete. DSACKx, STERM, or AVEC is not asserted if:

- The external device does not respond.
- No interrupt vector is provided.
- Various other application-dependent errors occur.

External circuitry can provide BERR when no device responds by asserting DSACKx, STERM, or AVEC within an appropriate period of time after the processor asserts AS. This allows the cycle to terminate and the processor to enter exception processing for the error condition.

The MMU can also detect an internal bus error. This occurs when the processor attempts to access an address in a protected area of memory (a user program attempts to access supervisor data, for example) or after the MMU receives a bus error while searching the address table for an address translation description.

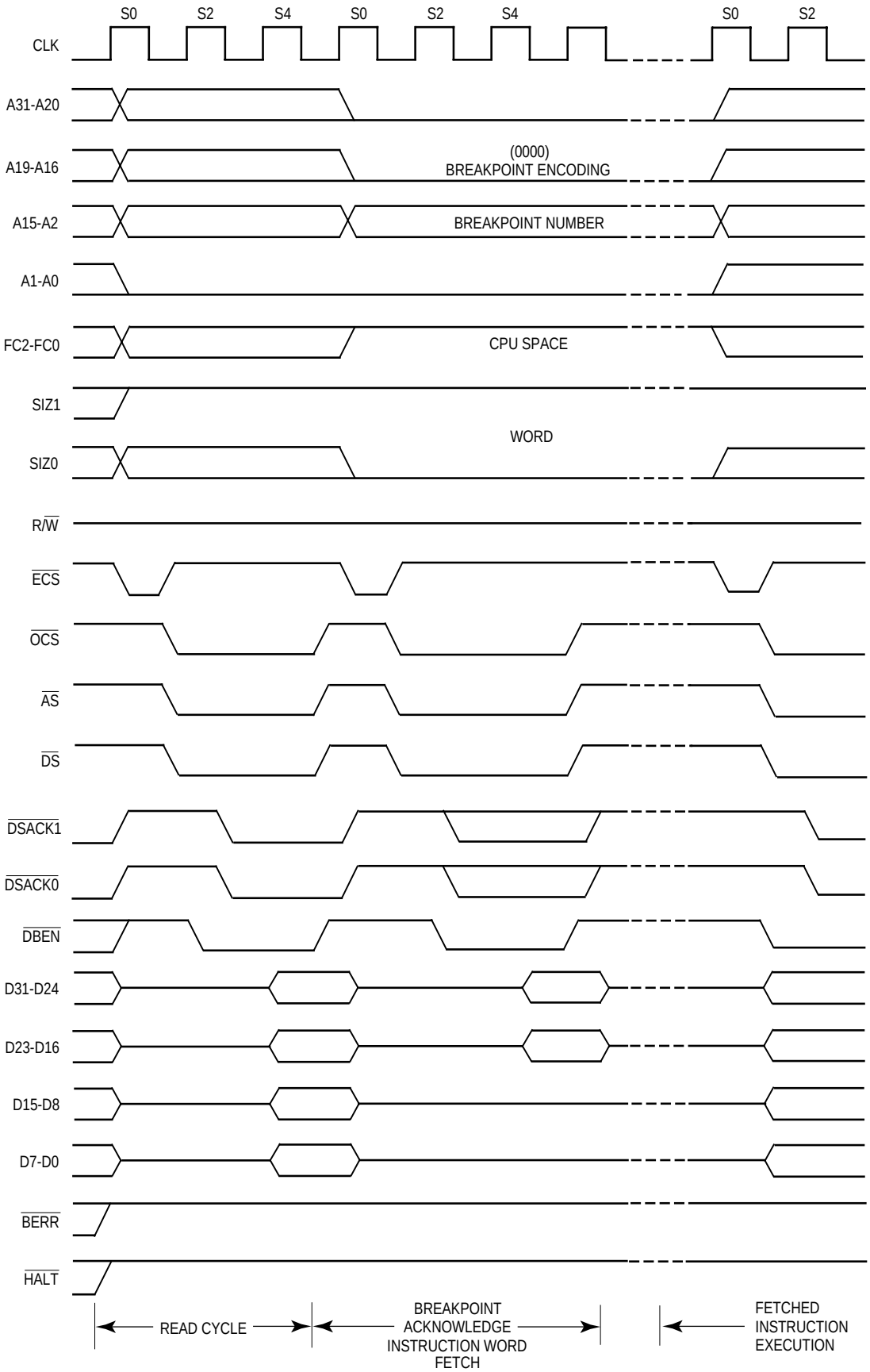


Figure 7-47. Breakpoint Acknowledge Cycle Timing

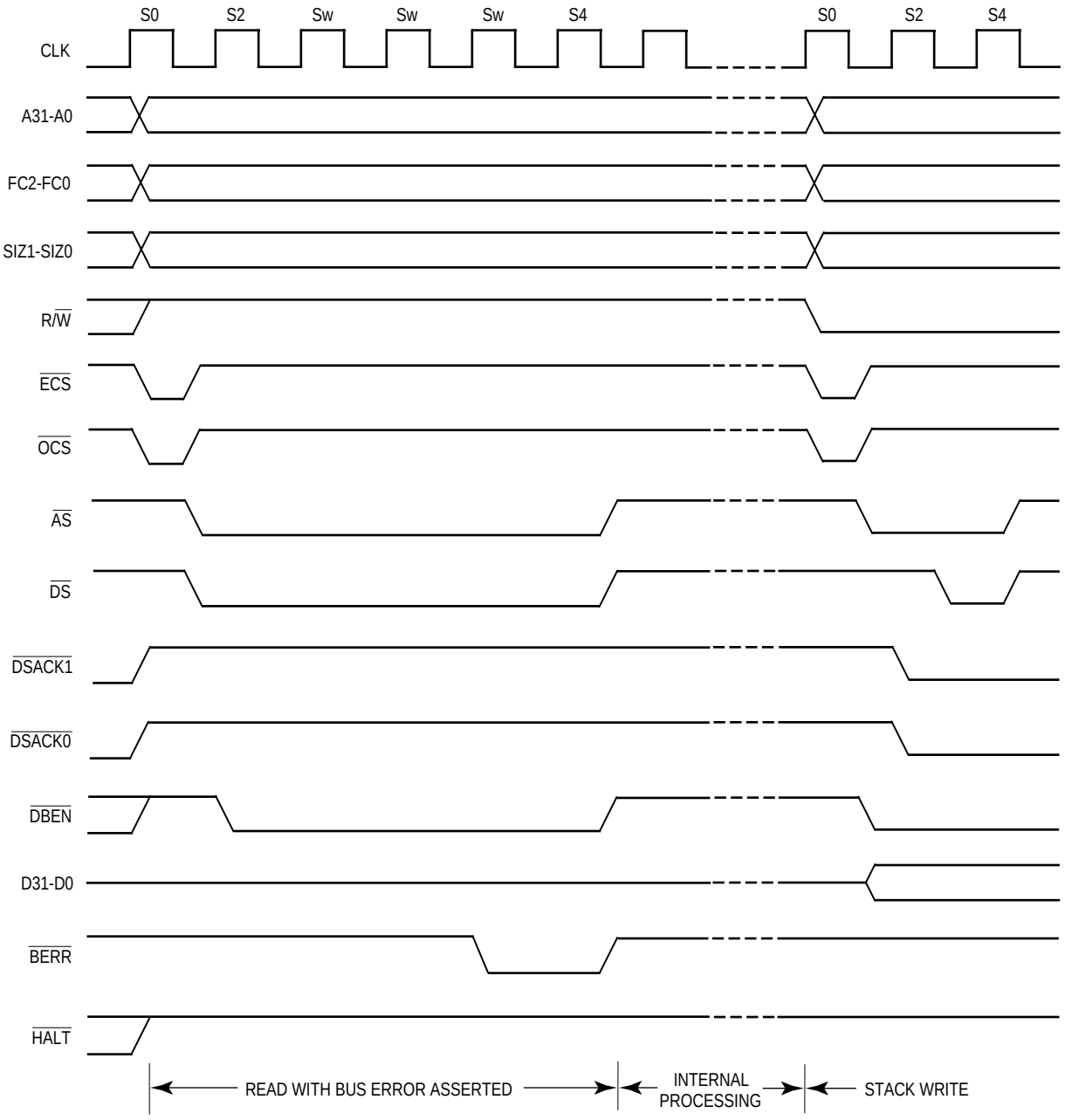


Figure 7-48. Breakpoint Acknowledge Cycle Timing (Exception Signaled)

Another signal that is used for bus exception control is $\overline{\text{HALT}}$. This signal can be asserted by an external device for debugging purposes to cause single bus cycle operation or (in combination with $\overline{\text{BERR}}$) a retry of a bus cycle in error.

To properly control termination of a bus cycle for a retry or a bus error condition, $\overline{\text{DSACKx}}$, $\overline{\text{BERR}}$, and $\overline{\text{HALT}}$ can be asserted and negated with the rising edge of the MC68030 clock. This assures that when two signals are asserted simultaneously, the required setup time (#47A) and hold time (#47B) for both of them is met for the same falling edge of the processor clock. (Refer to MC68030EC/D, MC68030 Electrical Specifications for timing requirements.) This or some equivalent precaution should be designed into the external circuitry that provides these signals.

The acceptable bus cycle terminations for asynchronous cycles are summarized in relation to $\overline{\text{DSACKx}}$ assertion as follows (case numbers refer to Table 7-8):

Normal Termination:

$\overline{\text{DSACKx}}$ is asserted; $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ remain negated (case 1).

Halt Termination:

$\overline{\text{HALT}}$ is asserted at same time or before $\overline{\text{DSACKx}}$, and $\overline{\text{BERR}}$ remains negated (case 2).

Bus Error Termination:

$\overline{\text{BERR}}$ is asserted in lieu of, at the same time, or before $\overline{\text{DSACKx}}$ (case 3) or after $\overline{\text{DSACKx}}$ (case 4), and $\overline{\text{HALT}}$ remains negated; $\overline{\text{BERR}}$ is negated at the same time or after $\overline{\text{DSACKx}}$.

Retry Termination:

$\overline{\text{HALT}}$ and $\overline{\text{BERR}}$ are asserted in lieu of, at the same time, or before $\overline{\text{DSACKx}}$ (case 5) or after $\overline{\text{DSACKx}}$ (case 6); $\overline{\text{BERR}}$ is negated at the same time or after $\overline{\text{DSACKx}}$; $\overline{\text{HALT}}$ may be negated at the same time or after $\overline{\text{BERR}}$.

Table 7-8. DSACK, BERR, and HALT Assertion Results

Case No.	Control Signal	Asserted on Rising Edge of State		Result
		N	N+2	
1	DSACKx BERR HALT	A NA NA	S NA X	Normal cycle terminate and continue.
2	DSACKx BERR HALT	A NA A/S	S NA S	Normal cycle terminate and halt. Continue when HALT negated.
3	DSACKx BERR HALT	NA/A A NA	X S NA	Terminate and take bus error exception, possibly deferred.
4	DSACKx BERR HALT	A NA NA	X A NA	Terminate and take bus error exception, possibly deferred.
5	DSACKx BERR HALT	NA/A A A/S	X S S	Terminate and retry when HALT negated.
6	DSACKx BERR HALT	A NA NA	X A A	Terminate and retry when HALT negated.

LEGEND:
N — The number of current even bus state (e.g., S2, S4, etc.)
A — Signal is asserted in this bus state
NA — Signal is not asserted in this state
X — Don't care
S — Signal was asserted in previous state and remains asserted in this state

Table 7-8 shows various combinations of control signal sequences and the resulting bus cycle terminations. To ensure predictable operation, BERR and HALT should be negated according to the specifications in MC68030EC/D, *MC68030 Electrical Specifications*. DSACKx, BERR, and HALT may be negated after AS. If DSACKx or BERR remain asserted into S2 of the next bus cycle, that cycle may be terminated prematurely.

The termination signal for a synchronous cycle is STERM. An analogous set of bus cycle termination cases exists in relationship to STERM assertion. Note that STERM and DSACKx must never both be asserted in the same cycle. STERM has setup time (#60) and hold time (#61) requirements relative to each rising edge of the processor clock while AS is asserted. Bus error and retry terminations during burst cycles operate as described in **6.1.3.2 Burst Mode Filling**, **7.5.1 Bus Errors**, and **7.5.2 Retry Operation.**

For $\overline{\text{STERM}}$, the bus cycle terminations are summarized as follows (case numbers refer to Table 7-9):

Normal Termination:

$\overline{\text{STERM}}$ is asserted; $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ remain negated (case 1).

Halt Termination:

$\overline{\text{HALT}}$ is asserted before $\overline{\text{STERM}}$, and $\overline{\text{BERR}}$ remains negated (case 2).

Bus Error Termination:

$\overline{\text{BERR}}$ is asserted in lieu of, at the same time, or before $\overline{\text{STERM}}$ (case 3) or after $\overline{\text{STERM}}$ (case 4), and $\overline{\text{HALT}}$ remains negated; $\overline{\text{BERR}}$ is negated at the same time or after $\overline{\text{STERM}}$.

Retry Termination:

$\overline{\text{HALT}}$ and $\overline{\text{BERR}}$ are asserted in lieu of, at the same time, or before $\overline{\text{STERM}}$ (case 5) or after $\overline{\text{STERM}}$ (case 6); $\overline{\text{BERR}}$ is negated at the same time or after $\overline{\text{STERM}}$; $\overline{\text{HALT}}$ may be negated at the same time or after $\overline{\text{BERR}}$.

Table 7-9. $\overline{\text{STERM}}$, $\overline{\text{BERR}}$, and $\overline{\text{HALT}}$ Assertion Results

Case No.	Control Signal	Asserted on Rising Edge of State		Result
		N	N+2	
1	$\overline{\text{STERM}}$ $\overline{\text{BERR}}$ $\overline{\text{HALT}}$	A NA NA	— — —	Normal cycle terminate and continue.
2	$\overline{\text{STERM}}$ $\overline{\text{BERR}}$ $\overline{\text{HALT}}$	NA NA A/S	A NA S	Normal cycle terminate and halt. Continue when $\overline{\text{HALT}}$ negated.
3	$\overline{\text{STERM}}$ $\overline{\text{BERR}}$ $\overline{\text{HALT}}$	NA A/S NA	A S NA	Terminate and take bus error exception, possibly deferred.
4	$\overline{\text{STERM}}$ $\overline{\text{BERR}}$ $\overline{\text{HALT}}$	A A N/A	— — —	Terminate and take bus error exception, possibly deferred.
5	$\overline{\text{STERM}}$ $\overline{\text{BERR}}$ $\overline{\text{HALT}}$	NA A A/S	A S S	Terminate and retry when $\overline{\text{HALT}}$ negated.
6	$\overline{\text{STERM}}$ $\overline{\text{BERR}}$ $\overline{\text{HALT}}$	A A A	— — —	Terminate and retry when $\overline{\text{HALT}}$ negated.

LEGEND:
N —The number of current even bus state (e.g., S2, S4, etc.)
A —Signal is asserted in this bus state
NA —Signal is not asserted in this state
X —Don't care
S —Signal was asserted in previous state and remains asserted in this state
— —State N+2 not part of bus cycle

EXAMPLE A:

A system uses a watchdog timer to terminate accesses to an unpopulated address space. The timer asserts $\overline{\text{BERR}}$ after timeout (case 3).

EXAMPLE B:

A system uses error detection and correction on RAM contents. The designer may:

1. Delay $\overline{\text{DSACKx}}$ until data is verified; assert $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ simultaneously to indicate to the processor to automatically retry the error cycle (case 5) or, if data is valid, assert $\overline{\text{DSACKx}}$ (case 1).
2. Delay $\overline{\text{DSACKx}}$ until data is verified and assert $\overline{\text{BERR}}$ with or without $\overline{\text{DSACKx}}$ if data is in error (case 3). This initiates exception processing for software handling of the condition.
3. Return $\overline{\text{DSACKx}}$ prior to data verification. If data is invalid, $\overline{\text{BERR}}$ is asserted on the next clock cycle (case 4). This initiates exception processing for software handling of the condition.
4. Return $\overline{\text{DSACKx}}$ prior to data verification; if data is invalid, assert $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ on the next clock cycle (case 6). The memory controller can then correct the RAM prior to or during the automatic retry.

7.5.1 Bus Errors

The bus error signal can be used to abort the bus cycle and the instruction being executed. $\overline{\text{BERR}}$ takes precedence over $\overline{\text{DSACKx}}$ or $\overline{\text{STERM}}$ provided it meets the timing constraints described in MC68030EC/D, *MC68030 Electrical Specifications*. If $\overline{\text{BERR}}$ does not meet these constraints, it may cause unpredictable operation of the MC68030. If $\overline{\text{BERR}}$ remains asserted into the next bus cycle, it may cause incorrect operation of that cycle.

When the bus error signal is issued to terminate a bus cycle, the MC68030 may enter exception processing immediately following the bus cycle, or it may defer processing the exception. The instruction prefetch mechanism requests instruction words from the bus controller and the instruction cache before it is ready to execute them. If a bus error occurs on an instruction fetch, the processor does not take the exception until it attempts to use that instruction word. Should an intervening instruction cause a branch or should a task switch occur, the bus error exception does not occur.

The bus error signal is recognized during a bus cycle in any of the following cases:

- $\overline{\text{DSACKx}}$ (or $\overline{\text{STERM}}$) and $\overline{\text{HALT}}$ are negated and $\overline{\text{BERR}}$ is asserted.
- $\overline{\text{HALT}}$ and $\overline{\text{BERR}}$ are negated and $\overline{\text{DSACKx}}$ is asserted. $\overline{\text{BERR}}$ is then asserted within one clock cycle ($\overline{\text{HALT}}$ remains negated).
- $\overline{\text{BERR}}$ is asserted and recognized on the next falling clock edge following the rising clock edge on which $\overline{\text{STERM}}$ is asserted and recognized ($\overline{\text{HALT}}$ remains negated).

When the processor recognizes a bus error condition, it terminates the current bus cycle in the normal way. Figure 7-49 shows the timing of a bus error for the case in which neither $\overline{\text{DSACKx}}$ nor $\overline{\text{STERM}}$ is asserted. Figure 7-50 shows the timing for a bus error that is asserted after $\overline{\text{DSACKx}}$. Exceptions are taken in both cases. (Refer to **8.1.2 Bus Error Exception** for details of bus error exception processing.) When $\overline{\text{BERR}}$ is asserted during a read cycle that supplies data to either on-chip cache, the data in the cache is marked invalid. However, when a write cycle that writes data into the data cache results in an externally generated bus error, the data in the cache is not marked invalid.

In the second case, where $\overline{\text{BERR}}$ is asserted after $\overline{\text{DSACKx}}$ is asserted, $\overline{\text{BERR}}$ must be asserted within specification #48 (refer to MC68030EC/D, *MC68030 Electrical Specifications*) for purely asynchronous operation, or it must be asserted and remain stable during the sample window, defined by specifications #27A and #47B, around the next falling edge of the clock after $\overline{\text{DSACKx}}$ is recognized. If $\overline{\text{BERR}}$ is not stable at this time, the processor may exhibit erratic behavior. $\overline{\text{BERR}}$ has priority over $\overline{\text{DSACKx}}$. In this case, data may be present on the bus, but may not be valid. This sequence may be used by systems that have memory error detection and correction logic and by external cache memories.

The assertion of $\overline{\text{BERR}}$ described in the third case (recognized after $\overline{\text{STERM}}$) has requirements similar to those described in the preceding paragraph. $\overline{\text{BERR}}$ must be stable throughout the sample window for the next falling edge of the clock, as defined by specifications #27A and #28A. Figure 7-51 shows the timing for this case.

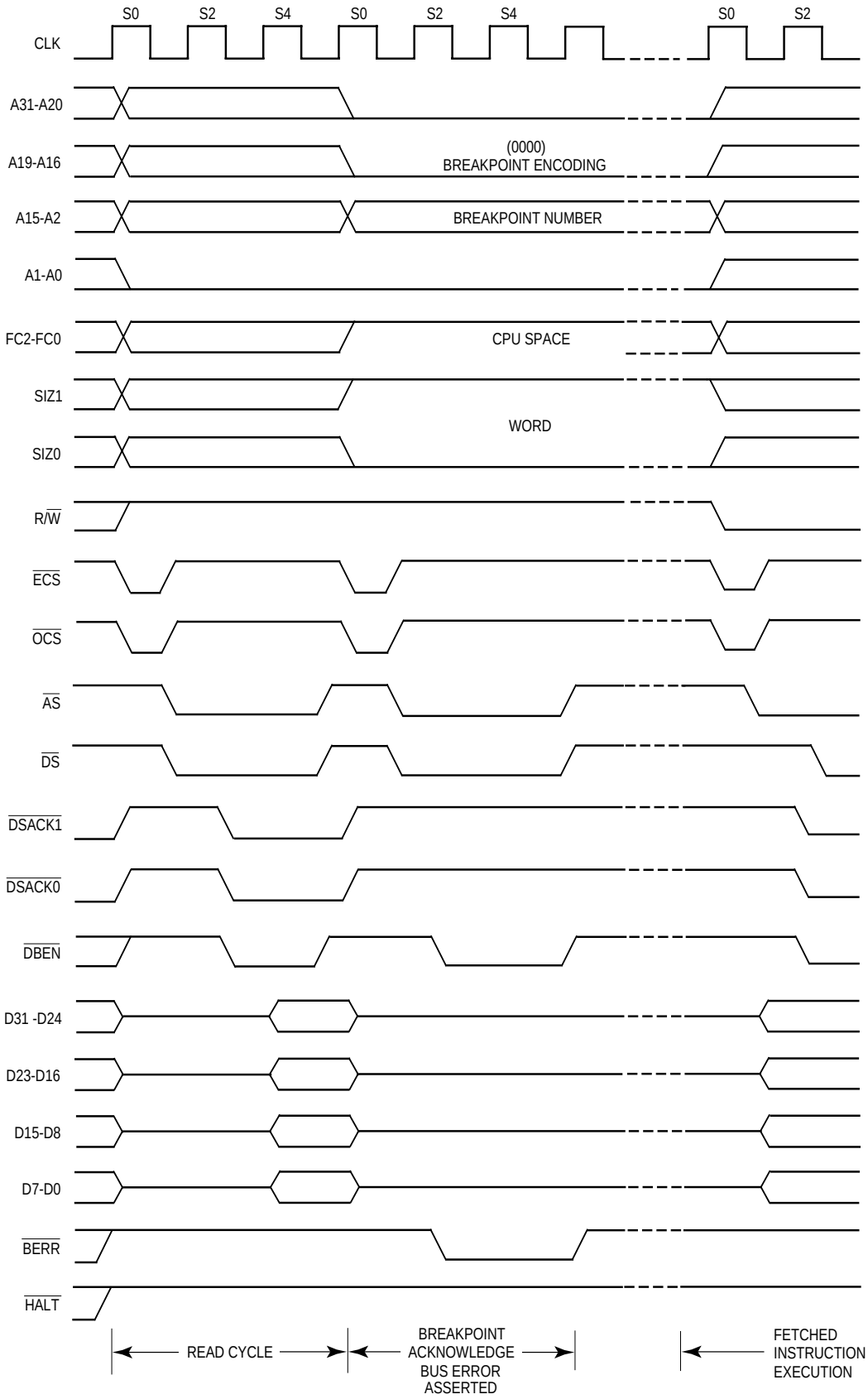


Figure 7-49. Bus Error without DSACKx

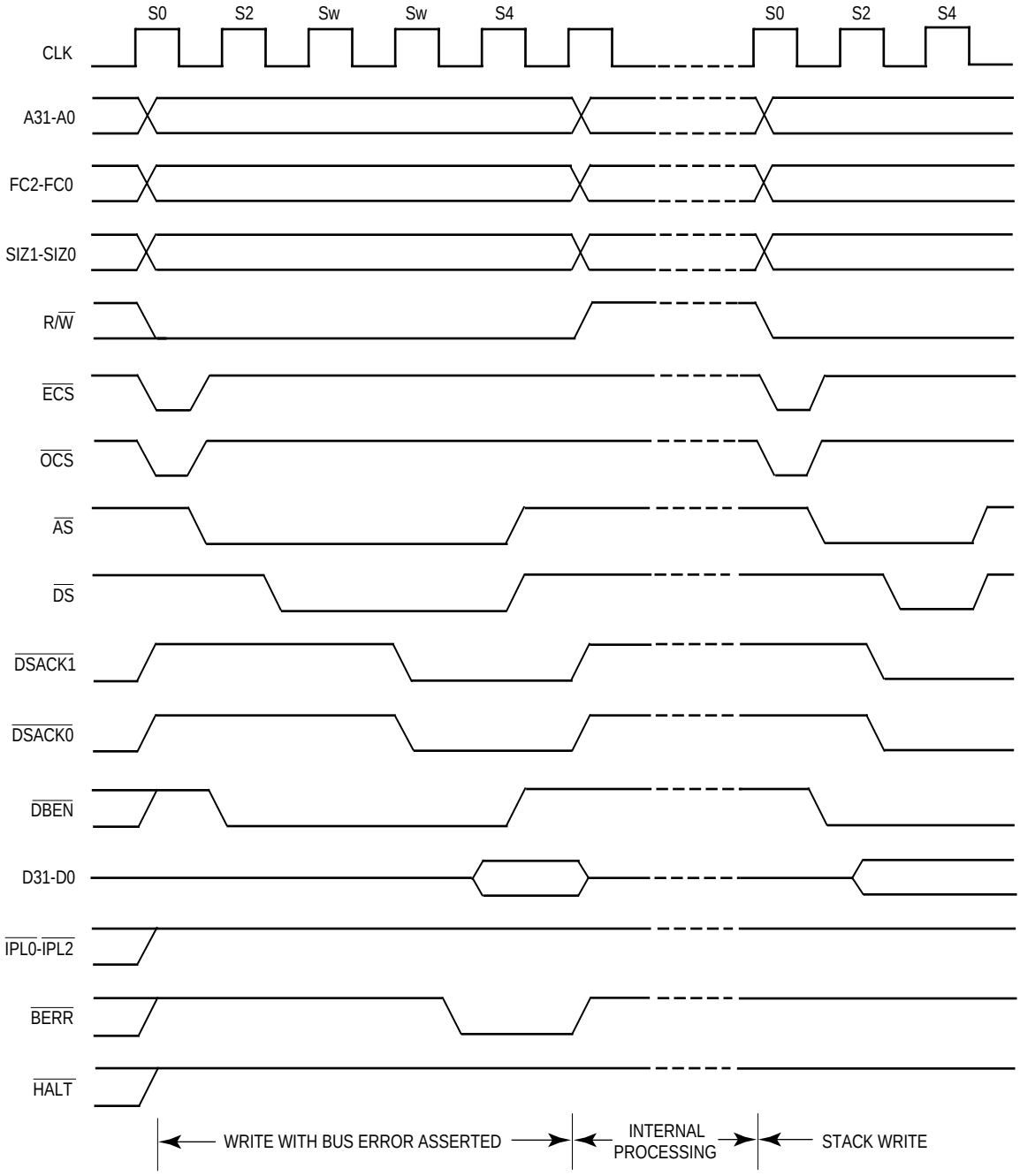


Figure 7-50. Late Bus Error with DSACKx

A bus error occurring during a burst fill operation is a special case. If a bus error occurs during the first cycle of a burst, the data is ignored, the entire cache line is marked invalid, and the burst operation is aborted. If the cycle is for an instruction fetch, a bus error exception is made pending. This bus error is processed only if the execution unit attempts to use either of the two words latched during the bus cycle. If the cycle is for a data fetch, the bus error exception is taken immediately. Refer to **Section 11 Instruction Execution Timing** for more information about pipeline operation.

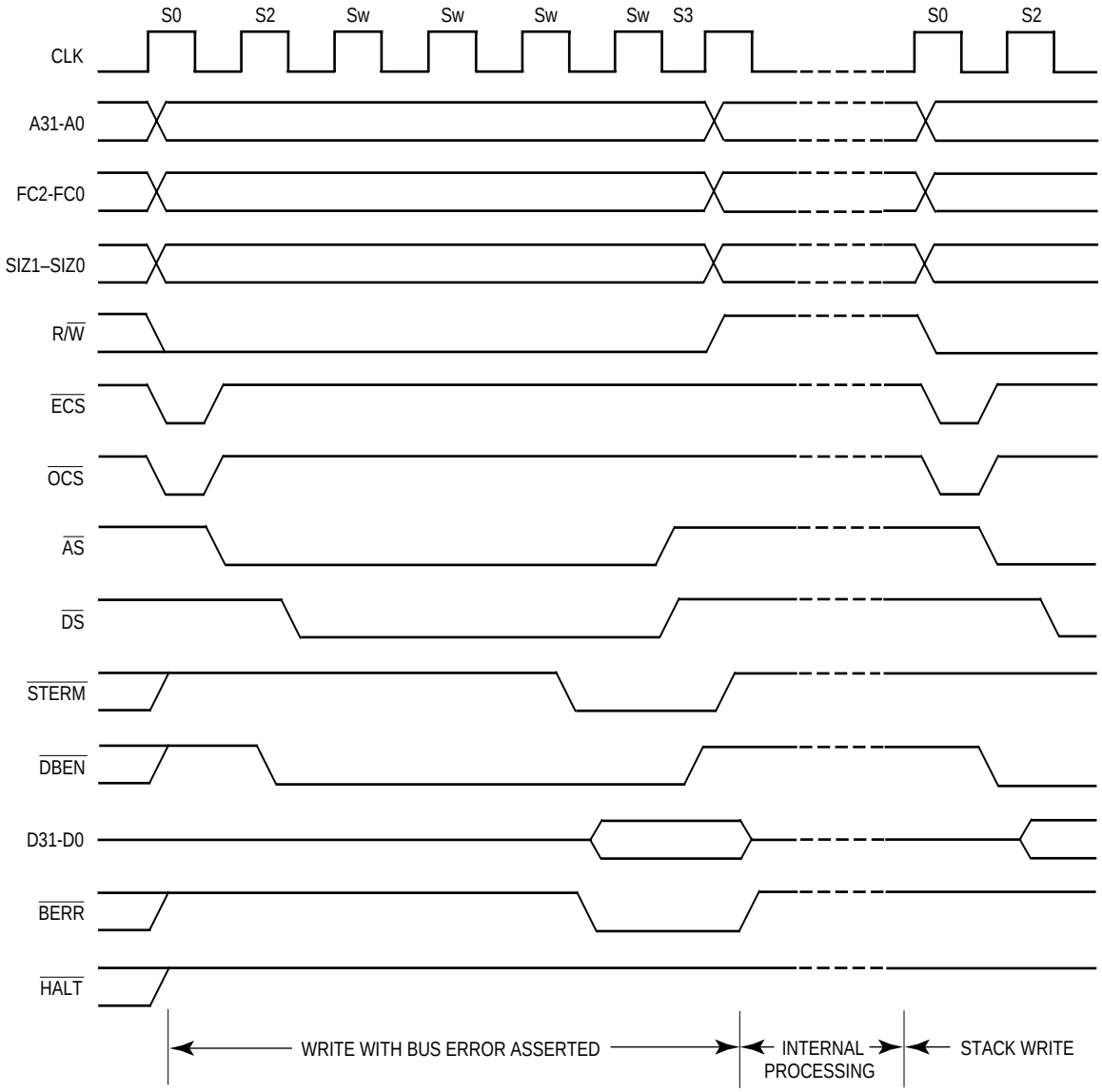


Figure 7-51. Late Bus Error with STERM — Exception Taken

When a bus error occurs after the burst mode has been entered (that is, on the second access or later), the processor terminates the burst operation, and the cache entry corresponding to that cycle is marked invalid, but the processor does not take an exception (see Figure 7-52). If the second cycle is for a portion of a misaligned operand fetch, the processor runs another read cycle for the second portion with CBREQ negated, as shown in Figure 7-53. If BERR is asserted again, the MC68030 then takes an exception. The MC68030 supports late bus errors during a burst fill operation; the timing is the same relative to STERM and the clock as for a late bus error in a normal synchronous cycle.

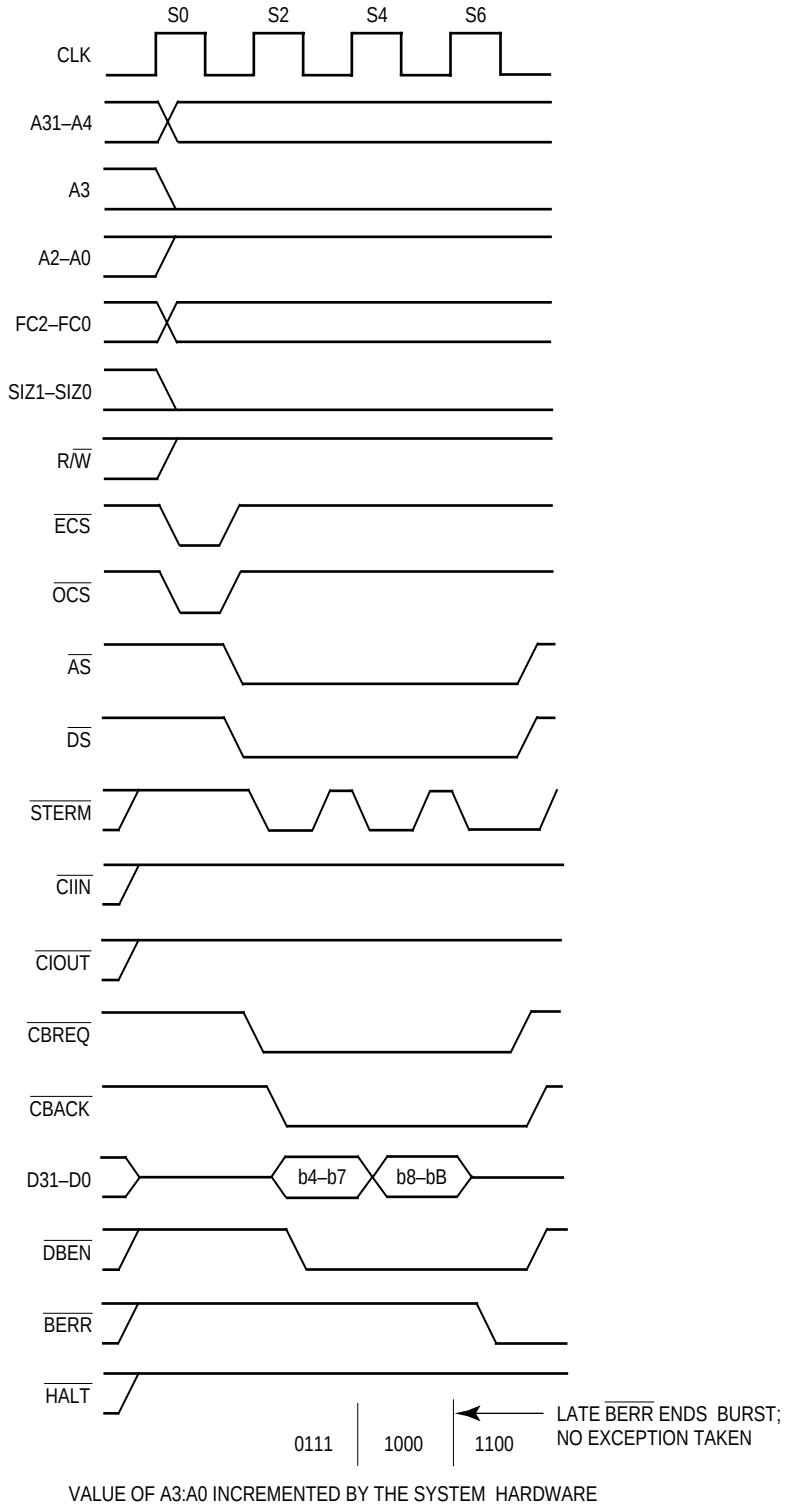


Figure 7-52. Long-Word Operand Request — Late $\overline{\text{BERR}}$ on Third Access

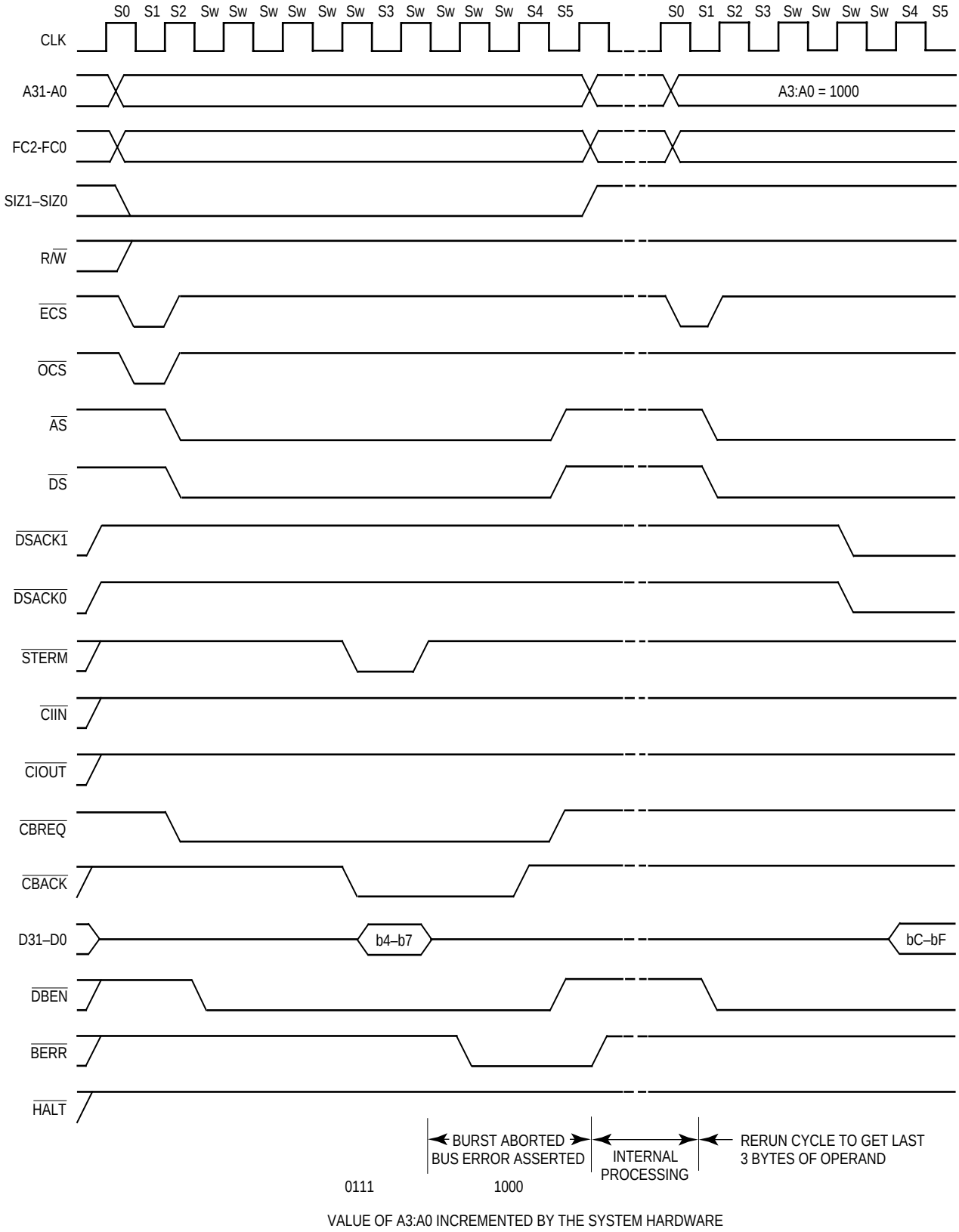


Figure 7-53. Long-Word Operand Request — $\overline{\text{BERR}}$ on Second Access

7.5.2 Retry Operation

When the BERR and HALT signals are both asserted by an external device during a bus cycle, the processor enters the retry sequence. A delayed retry, similar to the delayed bus error signal described previously, can also occur, both for synchronous and asynchronous cycles.

The processor terminates the bus cycle, places the control signals in their inactive state, and does not begin another bus cycle until the HALT signal is negated by external logic. After a synchronization delay, the processor retries the previous cycle using the same access information (address, function code, size, etc.) The BERR signal should be negated before S2 of the read cycle to ensure correct operation of the retried cycle. Figure 7-54 shows a retry operation of an asynchronous cycle, and Figure 7-55 shows a retry operation of a synchronous cycle.

The processor retries any read or write cycle of a read-modify-write operation separately; RMC remains asserted during the entire retry sequence.

On the initial access of a burst operation, a retry (indicated by the assertion of BERR and HALT) causes the processor to retry the bus cycle and assert CBREQ again. Figure 7-56 shows a late retry operation that causes an initial burst operation to be repeated. However, signaling a retry with simultaneous BERR and HALT during the second, third, or fourth cycle of a burst operation does not cause a retry operation, even if the requested operand is misaligned. Assertion of BERR and HALT during a subsequent cycle of a burst operation causes independent BERR and HALT operations. The external bus activity remains halted until HALT is negated and the processor acts as previously described for the bus error during a burst operation.

Asserting BR along with BERR and HALT provides a relinquish and retry operation. The MC68030 does not relinquish the bus during a read-modify-write operation, except during the first read cycle. Any device that requires the processor to give up the bus and retry a bus cycle during a read-modify-write cycle must either assert BERR and BR only (HALT must not be included) or use the single wire arbitration method discussed in **7.7.4 Bus Arbitration Control**. The bus error handler software should examine the read-modify-write bit in the special status word (refer to **8.2.1 Special Status Word (SSW)**) and take the appropriate action to resolve this type of fault when it occurs.

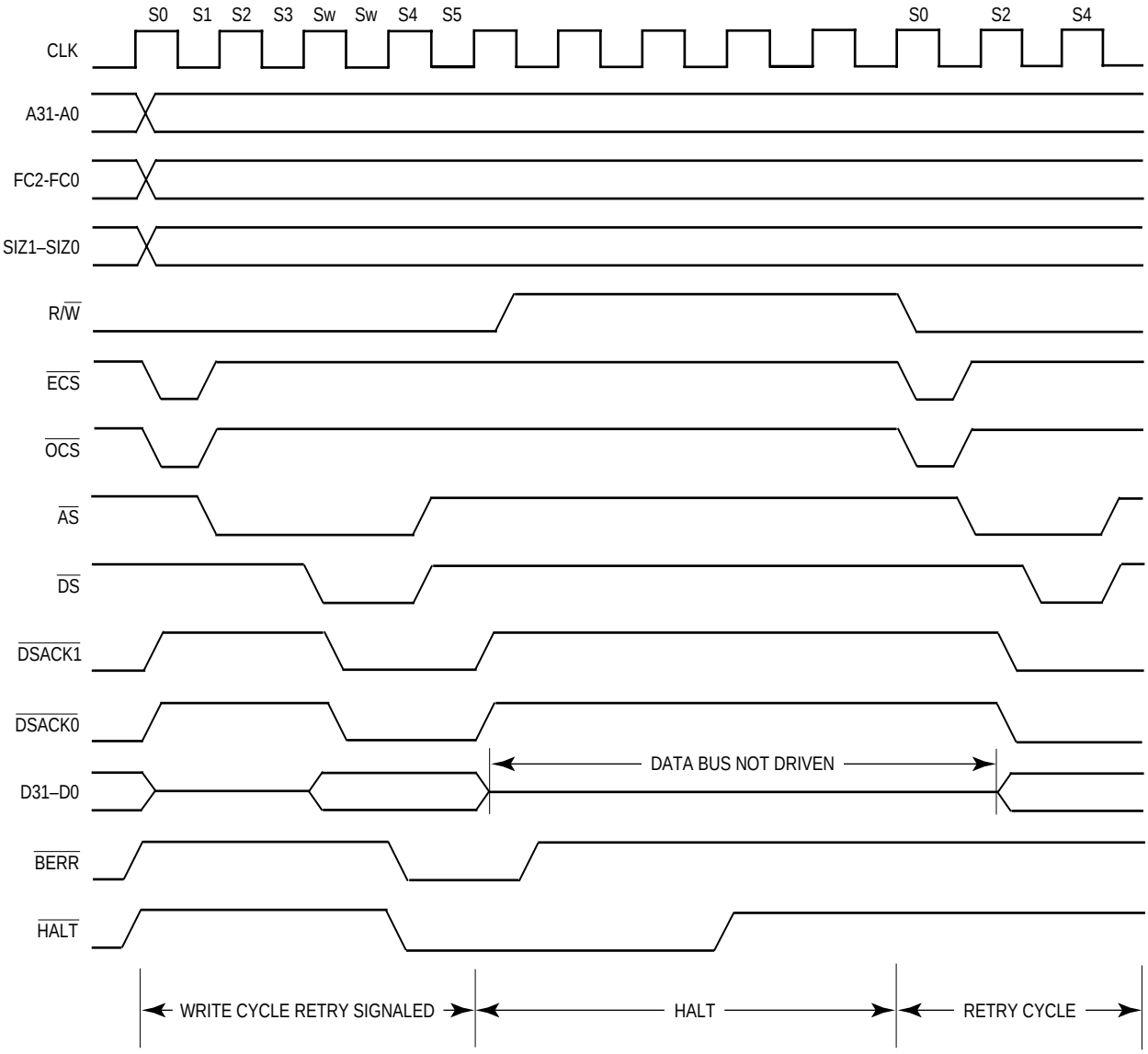


Figure 7-54. Asynchronous Late Retry

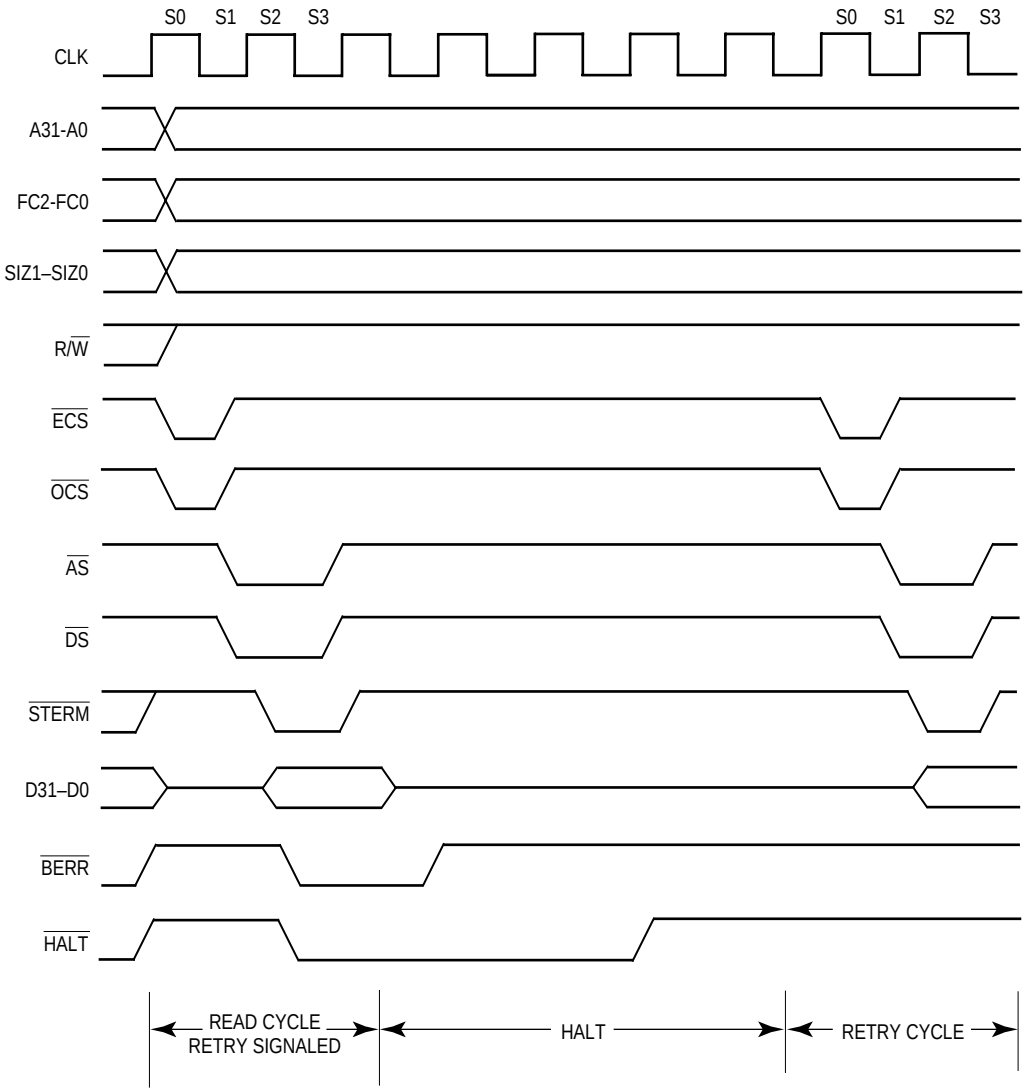


Figure 7-55. Synchronous Late Retry

7.5.3 Halt Operation

When HALT is asserted and BERR is not asserted, the MC68030 halts external bus activity at the next bus cycle boundary. HALT by itself does not terminate a bus cycle. Negating and reasserting HALT in accordance with the correct timing requirements provides a single-step (bus cycle to bus cycle) operation. The HALT signal affects external bus cycles only; thus, a program that resides in the instruction cache and performs no data writes (or reads that miss in the data cache) may continue executing, unaffected by the HALT signal.

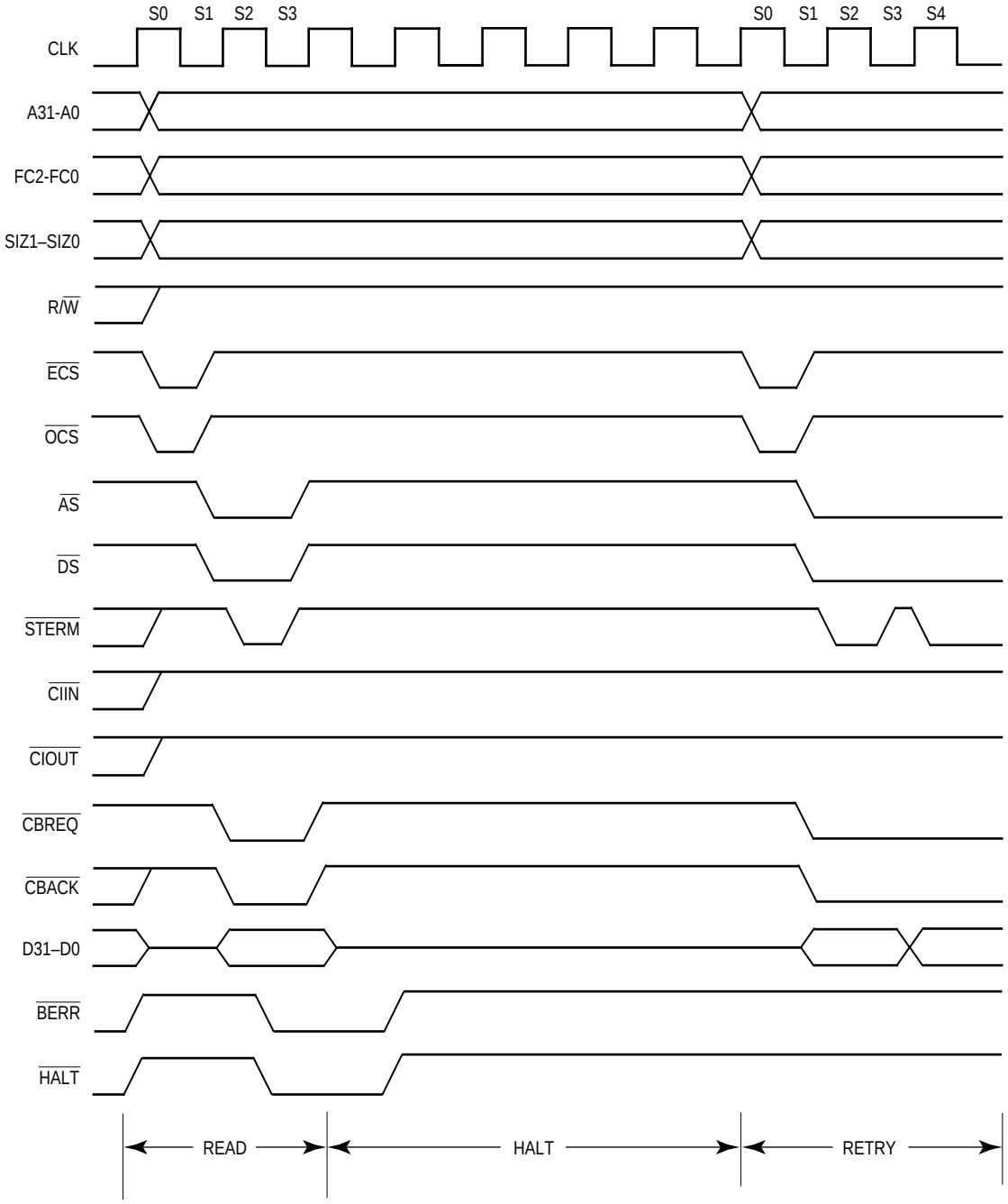


Figure 7-56. Late Retry Operation for a Burst

The single-cycle mode allows the user to proceed through (and debug) external processor operations, one bus cycle at a time. Figure 7-57 shows the timing requirements for a single-cycle operation. Since the occurrence of a bus error while $\overline{\text{HALT}}$ is asserted causes a retry operation, the user must anticipate retry cycles while debugging in the single-cycle mode. The single-step operation and the software trace capability allow the system debugger to trace single bus cycles, single instructions, or changes in program flow. These processor capabilities, along with a software debugging package, give complete debugging flexibility.

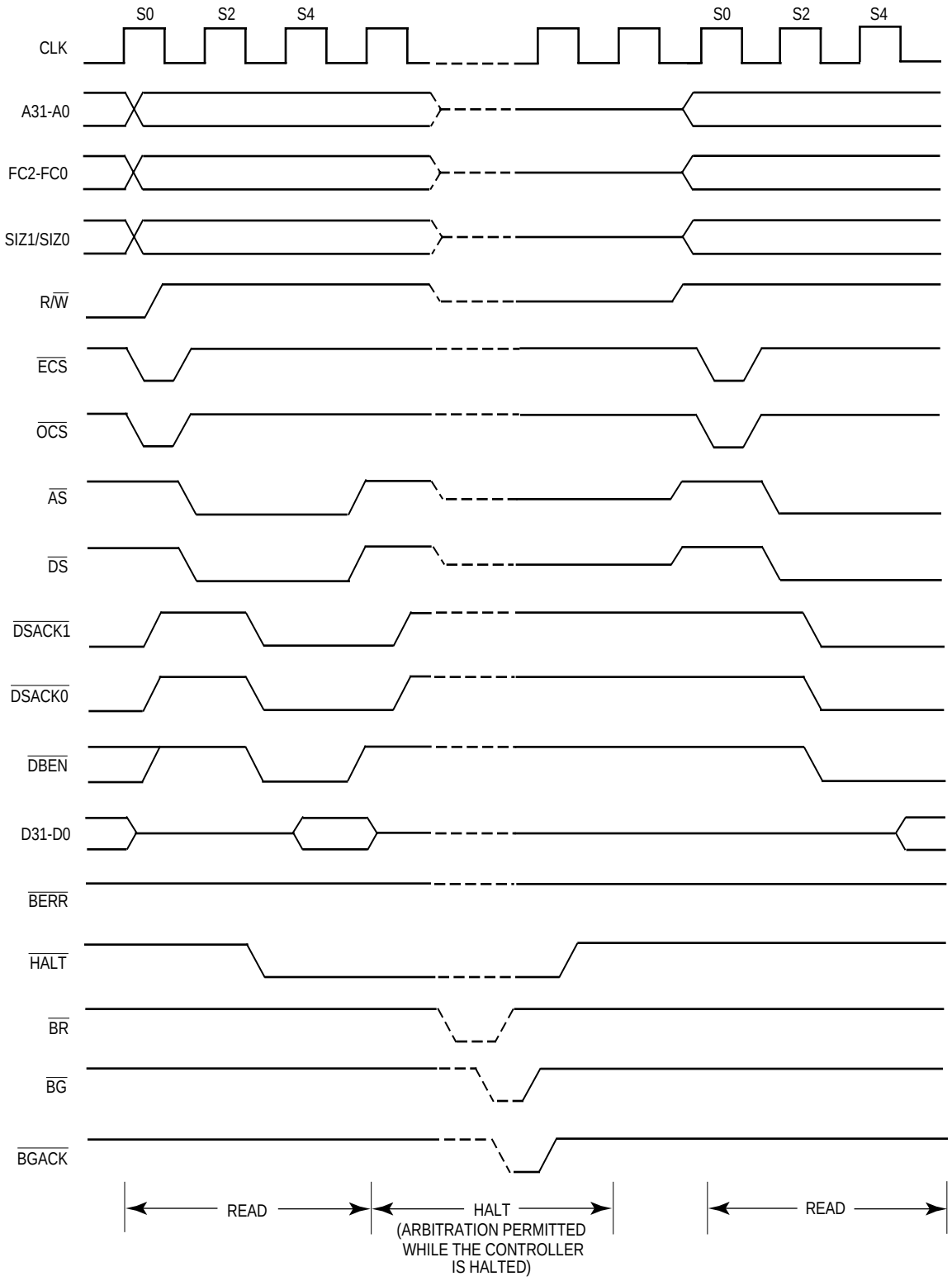


Figure 7-57. Halt Operation Timing

When the processor completes a bus cycle with the $\overline{\text{HALT}}$ signal asserted, the data bus is placed in the high-impedance state, and bus control signals are driven inactive (not high-impedance state); the address, function code, size, and read/write signals remain in the same state. The halt operation has no effect on bus arbitration (refer to **7.7 Bus Arbitration**). When bus arbitration occurs while the MC68030 is halted, the address and control signals are also placed in the high-impedance state. Once bus mastership is returned to the MC68030, if $\overline{\text{HALT}}$ is still asserted, the address, function code, size, and read/write signals are again driven to their previous states. The processor does not service interrupt requests while it is halted, but it may assert the $\overline{\text{IPEND}}$ signal as appropriate.

7.5.4 Double Bus Fault

When a bus error or an address error occurs during the exception processing sequence for a previous bus error, a previous address error, or a reset exception, the bus or address error causes a double bus fault. For example, the processor attempts to stack several words containing information about the state of the machine while processing a bus error exception. If a bus error exception occurs during the stacking operation, the second error is considered a double bus fault. Only an external reset operation can restart a halted processor. However, bus arbitration can still occur (refer to **7.7 Bus Arbitration**).

The MC68030 indicates that a double bus fault condition has occurred by continuously asserting the $\overline{\text{STATUS}}$ signal until the processor is reset. The processor asserts $\overline{\text{STATUS}}$ for one, two, or three clock periods to signal other microsequencer status indications. Refer to **Section 12 Applications Information** for a description of the interpretation of the STATUS signal.

A second bus error or address error that occurs after exception processing has completed (during the execution of the exception handler routine or later) does not cause a double bus fault. A bus cycle that is retried does not constitute a bus error or contribute to a double bus fault. The processor continues to retry the same bus cycle as long as the external hardware requests it.

7.6 BUS SYNCHRONIZATION

The MC68030 overlaps instruction execution; that is, during bus activity for one instruction, instructions that do not use the external bus can be executed. Due to the independent operation of the on-chip caches relative to the operation of the bus controller, many subsequent instructions can be executed, resulting in seemingly nonsequential instruction execution. When this is not desired and the system depends on sequential execution following bus activity, the NOP instruction can be used. The NOP instruction forces instruction and bus synchronization in that it freezes instruction execution until all pending bus cycles have completed.

An example of the use of the NOP instruction for this purpose is the case of a write operation of control information to an external register, where the external hardware attempts to control program execution based on the data that is written with the conditional assertion of $\overline{\text{BERR}}$. If the data cache is enabled and the write cycle results in a hit in the data cache, the cache is updated. That data, in turn, may be used in a subsequent instruction before the external write cycle completes. Since the MC68030 cannot process the bus error until the end of the bus cycle, the external hardware has not successfully interrupted program execution. To prevent a subsequent instruction from executing until the external cycle completes, a NOP instruction can be inserted after the instruction causing the write. In this case, bus error exception processing proceeds immediately after the write before subsequent instructions are executed. This is an irregular situation, and the use of the NOP instruction for this purpose is not required by most systems.

Note that even in a system with error detection/correction circuitry, the NOP is not required for this synchronization. Since the MMU always checks the validity of write cycles before they proceed to the data cache and are executed externally, the MC68030 is guaranteed to write correct data to the cache. Thus, there is no danger in subsequent instructions using erroneous data from the cache before an external bus error signals an error.

A bus synchronization example is given in Figure 7-58.

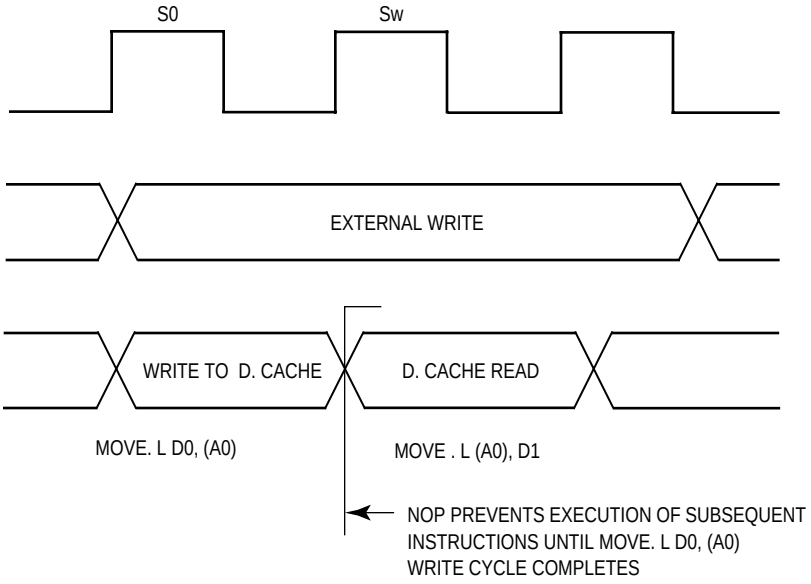


Figure 7-58. Bus Synchronization Example

7.7 BUS ARBITRATION

The bus design of the MC68030 provides for a single bus master at any one time: either the processor or an external device. One or more of the external devices on the bus can have the capability of becoming bus master. Bus arbitration is the protocol by which an external device becomes bus master; the bus controller in the MC68030 manages the bus arbitration signals so that the processor has the lowest priority. External devices that need to obtain the bus must assert the bus arbitration signals in the sequences described in the following paragraphs. Systems having several devices that can become bus master require external circuitry to assign priorities to the device so that, when two or more external devices attempt to become bus master at the same time, the one having the highest priority becomes bus master first. The sequence of the protocol is:

1. An external device asserts the bus request signal.
2. The processor asserts the bus grant signal to indicate that the bus will become available at the end of the current bus cycle.
3. The external device asserts the bus grant acknowledge signal to indicate that it has assumed bus mastership.

$\overline{BR}$ may be issued any time during a bus cycle or between cycles. $\overline{BG}$ is asserted in response to $\overline{BR}$; it is usually asserted as soon as $\overline{BR}$ has been synchronized and recognized, except when the MC68030 has made an internal decision to execute a bus cycle. Then, the assertion of $\overline{BG}$ is deferred until the bus cycle has begun. Additionally, BG is not asserted until the end of a read-modify-write operation (when $\overline{RMC}$ is negated) in response to a $\overline{BR}$ signal. When the requesting device receives BG and more than one external device can be bus master, the requesting device should begin whatever arbitration is required. The external device asserts $\overline{BGACK}$ when it assumes bus mastership and

maintains $\overline{\text{BGACK}}$ during the entire bus cycle (or cycles) for which it is bus master. The following conditions must be met for an external device to assume mastership of the bus through the normal bus arbitration procedure:

- It must have received $\overline{\text{BG}}$ through the arbitration process.
- $\overline{\text{AS}}$ must be negated, indicating that no bus cycle is in progress, and the external device must ensure that all appropriate processor signals have been placed in the high-impedance state (by observing specification #7 in MC68030EC/D, *MC68030 Electrical Specifications*).
- The termination signal ($\overline{\text{DSACKx}}$ or $\overline{\text{STERM}}$) for the most recent cycle must have become inactive, indicating that external devices are off the bus (optional, refer to **7.7.3 Bus Grant Acknowledge**).
- $\overline{\text{BGACK}}$ must be inactive, indicating that no other bus master has claimed ownership of the bus.

Figure 7-59 is a flowchart showing the detail involved in bus arbitration for a single device. Figure 7-60 is a timing diagram for the same operation. This technique allows processing of bus requests during data transfer cycles.

The timing diagram shows that $\overline{\text{BR}}$ is negated at the time that $\overline{\text{BGACK}}$ is asserted. This type of operation applies to a system consisting of the processor and one device capable of bus mastership. In a system having a number of devices capable of bus mastership, the bus request line from each device can be wire-ORed to the processor. In such a system, more than one bus request can be asserted simultaneously.

The timing diagram in Figure 7-60 shows that $\overline{\text{BG}}$ is negated a few clock cycles after the transition of the $\overline{\text{BGACK}}$ signal. However, if bus requests are still pending after the negation of $\overline{\text{BG}}$, the processor asserts another $\overline{\text{BG}}$ within a few clock cycles after it was negated. This additional assertion of $\overline{\text{BG}}$ allows external arbitration circuitry to select the next bus master before the current bus master has finished with the bus. The following paragraphs provide additional information about the three steps in the arbitration process.

Bus arbitration requests are recognized during normal processing, $\overline{\text{RESET}}$ assertion, $\overline{\text{HALT}}$ assertion, and even when the processor has halted due to a double bus fault.

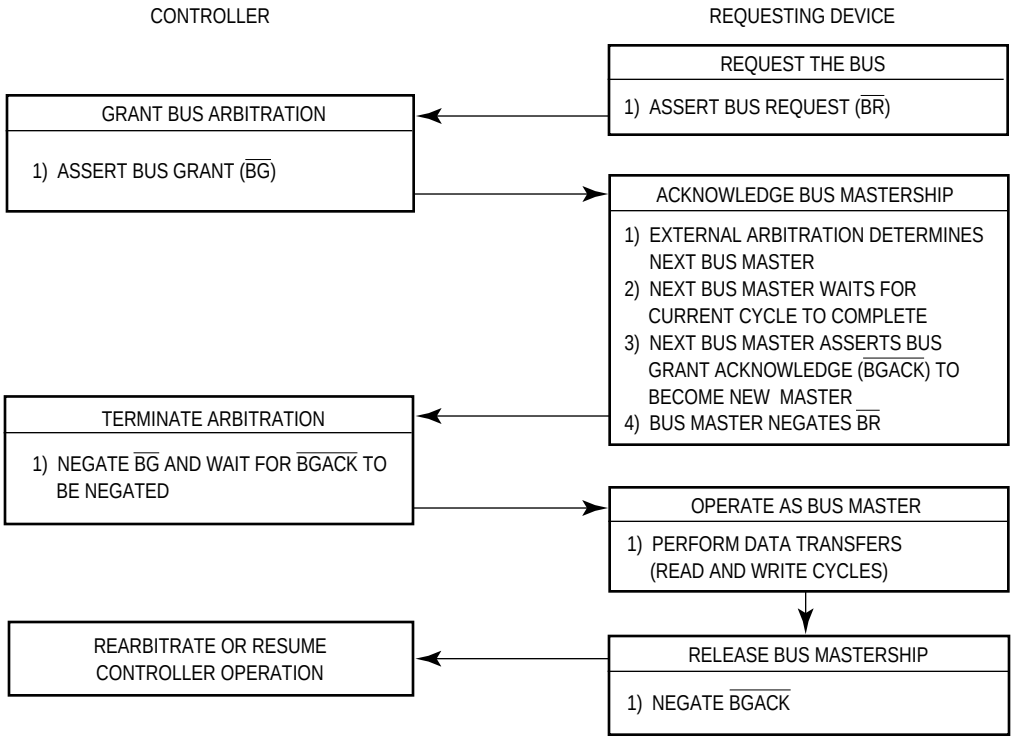


Figure 7-59. Bus Arbitration Flowchart for Single Request

7.7.1 Bus Request

External devices capable of becoming bus masters request the bus by asserting $\overline{BR}$. This can be a wire-ORed signal (although it need not be constructed from open-collector devices) that indicates to the processor that some external device requires control of the bus. The processor is effectively at a lower bus priority level than the external device and relinquishes the bus after it has completed the current bus cycle (if one has started).

If no acknowledge is received while the $\overline{BR}$ is active, the processor remains bus master once $\overline{BR}$ is negated. This prevents unnecessary interference with ordinary processing if the arbitration circuitry inadvertently responds to noise or if an external device determines that it no longer requires use of the bus before it has been granted mastership.

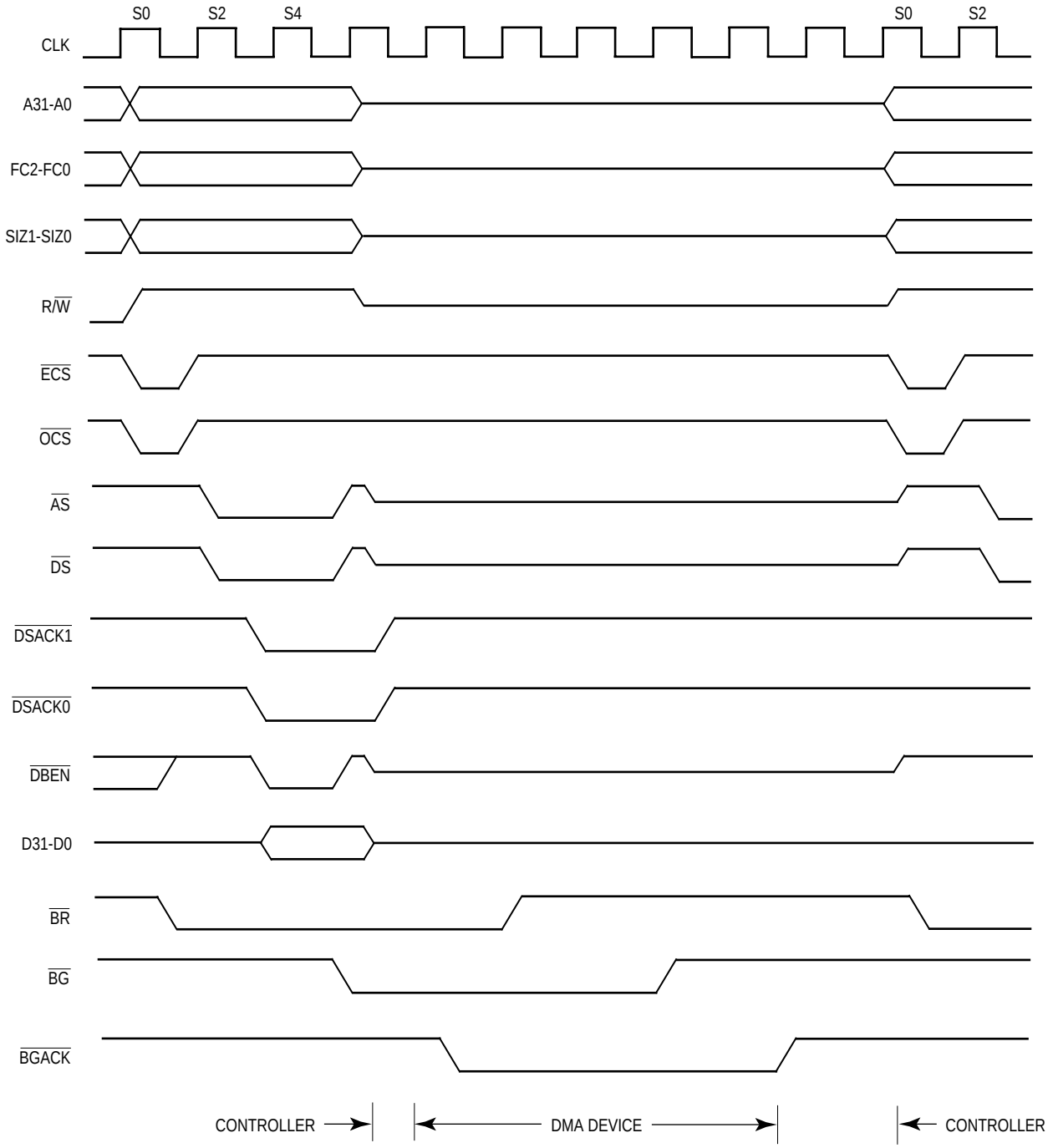


Figure 7-60. Bus Arbitration Operation Timing

7.7.2 Bus Grant

The processor asserts $\overline{BG}$ as soon as possible after receipt of $\overline{BR}$. This is immediately following internal synchronization except during a read-modify-write cycle or following an internal decision to execute a bus cycle. During a read-modify-write cycle, the processor does not assert $\overline{BG}$ until the entire operation has completed. $\overline{RMC}$ is asserted to indicate

that the bus is locked. In the case an internal decision to execute another bus cycle, $\overline{BG}$ is deferred until the bus cycle has begun.

$\overline{BG}$ may be routed through a daisy-chained network or through a specific priority-encoded network. The processor allows any type of external arbitration that follows the protocol.

7.7.3 Bus Grant Acknowledge

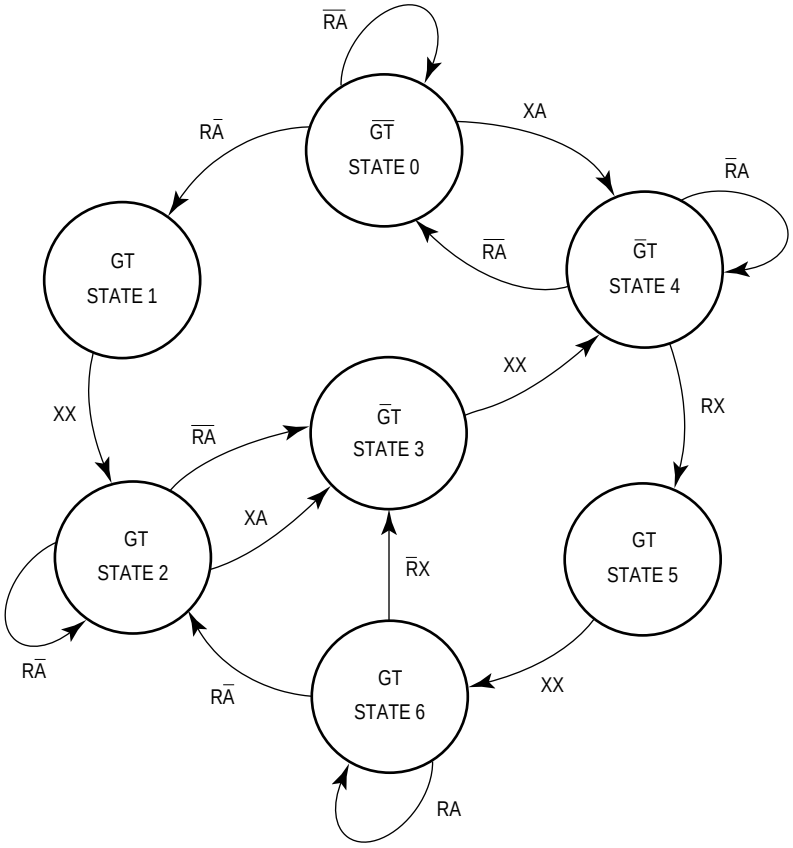
Upon receiving $\overline{BG}$, the requesting device waits until $\overline{AS}$, $\overline{DSACKx}$ (or synchronous termination, $\overline{STERM}$), and $\overline{BGACK}$ are negated before asserting its own $\overline{BGACK}$. The negation of the AS indicates that the previous master releases the bus after specification #7 (refer to MC68030EC/D, *MC68030 Electrical Specifications*). The negation of $\overline{DSACKx}$ or $\overline{STERM}$ indicates that the previous slave has completed its cycle with the previous master. Note that in some applications, $\overline{DSACKx}$ might not be used in this way.

General-purpose devices are then connected to be dependent only on $\overline{AS}$. When $\overline{BGACK}$ is asserted, the device is the bus master until it negates BGACK. BGACK should not be negated until all bus cycles required by the alternate bus master are completed. Bus mastership terminates at the negation of $\overline{BGACK}$. The $\overline{BR}$ from the granted device should be negated after $\overline{BGACK}$ is asserted. If a BR is still pending after the assertion of $\overline{BGACK}$, another $\overline{BG}$ is asserted within a few clocks of the negation of $\overline{BG}$, as described in the **7.7.4 Bus Arbitration Control**. Note that the processor does not perform any external bus cycles before it reasserts $\overline{BG}$ in this case.

7.7.4 Bus Arbitration Control

The bus arbitration control unit in the MC68030 is implemented with a finite state machine. As discussed previously, all asynchronous inputs to the MC68030 are internally synchronized in a maximum of two cycles of the processor clock.

As shown in Figure 7-61, input signals labeled R and A are internally synchronized versions of the $\overline{BR}$ and $\overline{BGACK}$ signals, respectively. The $\overline{BG}$ output is labeled G, and the internal high-impedance control signal is labeled T. If T is true, the address, data, and control buses are placed in the high-impedance state after the next rising edge following the negation of $\overline{AS}$ and $\overline{RMC}$. All signals are shown in positive logic (active high), regardless of their true active voltage level.



R - BUS REQUEST
A - BUS GRANT ACKNOWLEDGE
G - BUS GRANT
T - THREE-STATE CONTROL TO BUS CONTROL LOGIC
X - DON'T CARE

NOTE: The $\overline{BG}$ output will not be asserted while $\overline{RMC}$ is asserted.

Figure 7-61. Bus Arbitration State Diagram

State changes occur on the next rising edge of the clock after the internal signal is valid. The $\overline{BG}$ signal transitions on the falling edge of the clock after a state is reached during which G changes. The bus control signals (controlled by T) are driven by the processor, immediately following a state change, when bus mastership is returned to the MC68030.

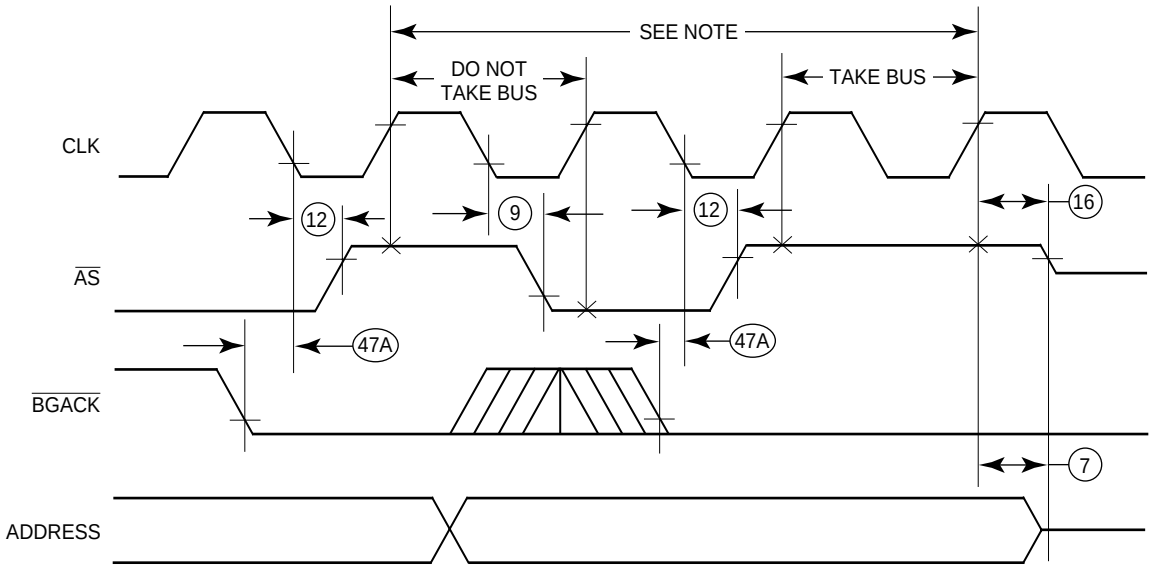
State 0, at the top center of the diagram, in which G and T are both negated, is the state of the bus arbiter while the processor is bus master. Request R and acknowledge A keep the arbiter in state 0 as long as they are both negated. When a request R is received, both grant G and signal T are asserted (in state 1 at the top left). The next clock causes a change to state 2, at the lower left, in which G and T are held. The bus arbiter remains in that state until acknowledge A is asserted or request R is negated. Once either occurs, the arbiter changes to the center state, state 3, and negates grant G. The next clock takes the arbiter to state 4, at the upper right, in which grant G remains negated and signal T remains asserted. With acknowledge A asserted, the arbiter remains in state 4 until A is negated or request R is

again asserted. When A is negated, the arbiter returns to the original state, state 0, and negates signal T. This sequence of states follows the normal sequence of signals for relinquishing the bus to an external bus master. Other states apply to other possible sequences of combinations of R and A. As shown by the path from state 0 to state 4, $\overline{\text{BGACK}}$ alone can be used to place the processor's external bus buffers in the high-impedance state, providing single-wire arbitration capability.

The read-modify-write sequence is normally indivisible to support semaphore operations and multiprocessor synchronization. During this indivisible sequence, the MC68030 asserts the RMC signal and causes the bus arbitration state machine to ignore bus requests (assertions of BR) that occur after the first read cycle of the read-modify-write sequence by not issuing bus grants (asserting BG).

In some cases, however, it may be necessary to force the MC68030 to release the bus during an read-modify-write sequence. One way for an alternate bus master to force the MC68030 to release the bus applies only to the first read cycle of an read-modify-write sequence. The MC68030 allows normal bus arbitration during this read cycle; a normal relinquish and retry operation (asserting $\overline{\text{BERR}}$, $\overline{\text{HALT}}$, and $\overline{\text{BR}}$ at the same time) is used. Note that this method applies only to the first read cycle of the read-modify-write sequence, but this method preserves the integrity of the read-modify-write sequence without imposing any constraint on the alternate bus master.

A second method is single-wire arbitration, the timing of which is shown in Figure 7-62. An alternate master forces the MC68030 to release the bus by asserting $\overline{\text{BGACK}}$ and waits for $\overline{\text{AS}}$ to negate before taking the bus. It applies to all bus cycles of a read-modify-write sequence, but can cause system integrity problems if used improperly. The alternate bus master must guarantee the integrity of the read-modify-write sequence by not altering the contents of memory locations accessed by the read-modify-write sequence. Note that for the method to operate properly, $\overline{\text{AS}}$ must be observed to be negated (high) on two consecutive clock edges before the alternate bus master takes the bus. Waiting for this condition ensures that any current or pending bus activity has completed or has been pre-empted.



NOTE: The alternate bus master must sample $\overline{AS}$ high on two consecutive rising edges of the clock (after $\overline{BGACK}$ is recognized low) before taking the bus.

Figure 7-62. Single-Wire Bus Arbitration Timing Diagram

A timing diagram of the bus arbitration sequence during a processor bus cycle is shown in Figure 7-60. The bus arbitration sequence while the bus is inactive (i.e., executing internal operations such as a multiply instruction) is shown in Figure 7-63.

7.8 RESET OPERATION

$\overline{RESET}$ is a bidirectional signal with which an external device resets the system or the processor resets external devices. When power is applied to the system, external circuitry should assert $\overline{RESET}$ for a minimum of 520 clocks after V_{CC} is within tolerance. Figure 7-64 is a timing diagram of the powerup reset operation, showing the relationships between $\overline{RESET}$, V_{CC}, and bus signals. The clock signal is required to be stable by the time V_{CC} reaches the minimum operating specification. During the reset period, the entire bus three-states (except for non-three-statable signals, which are driven to their inactive state). Once $\overline{RESET}$ negates, all control signals are driven to their inactive state, the data bus is in read mode, and the address bus is driven. After this, the first bus cycle for reset exception processing begins.

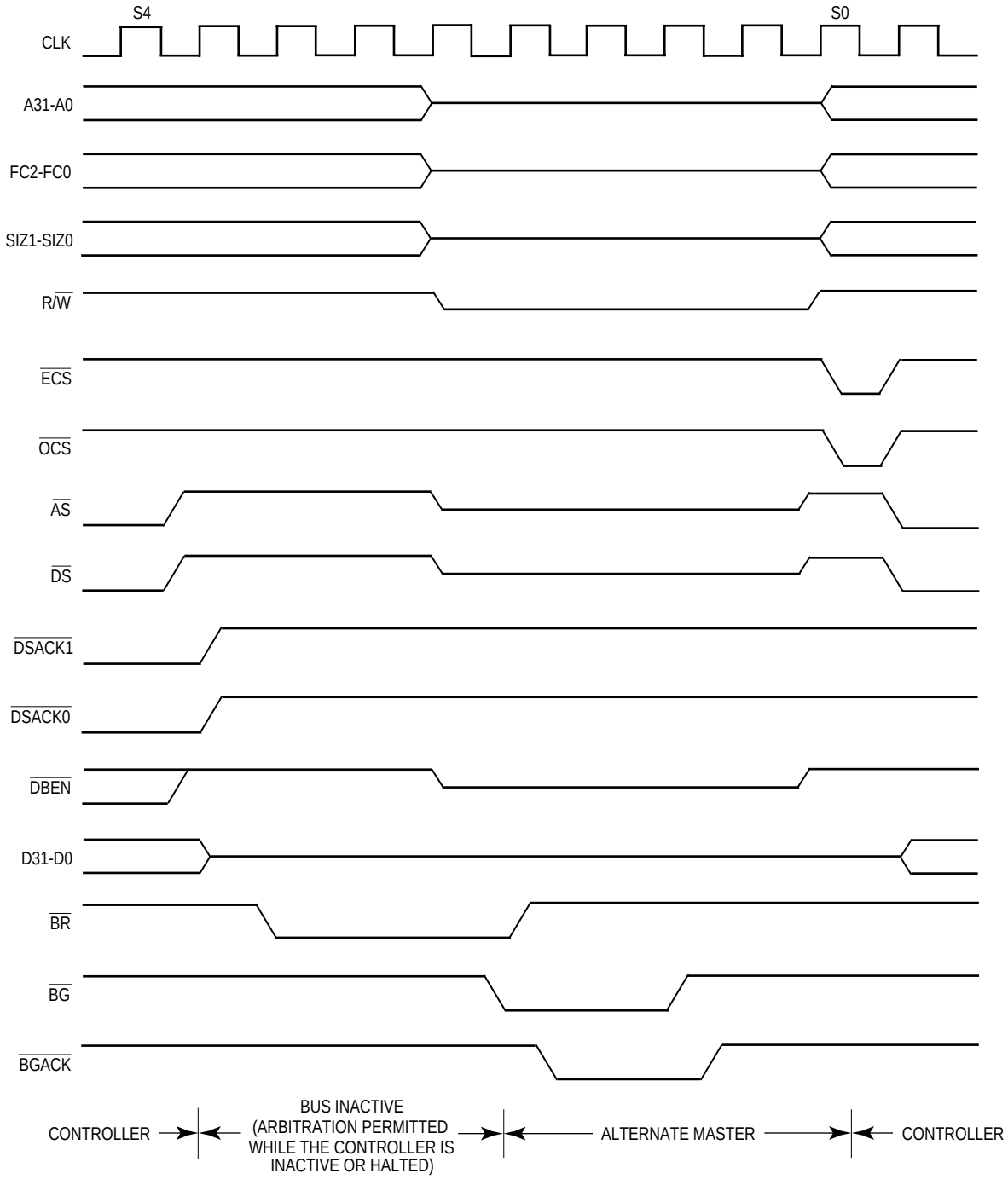


Figure 7-63. Bus Arbitration Operation (Bus Inactive)

The external RESET signal resets the processor and the entire system. Except for the initial reset, RESET should be asserted for at least 520 clock periods to ensure that the processor resets. Asserting RESET for 10 clock periods is sufficient for resetting the processor logic; the additional clock periods prevent a reset instruction from overlapping the external RESET signal.

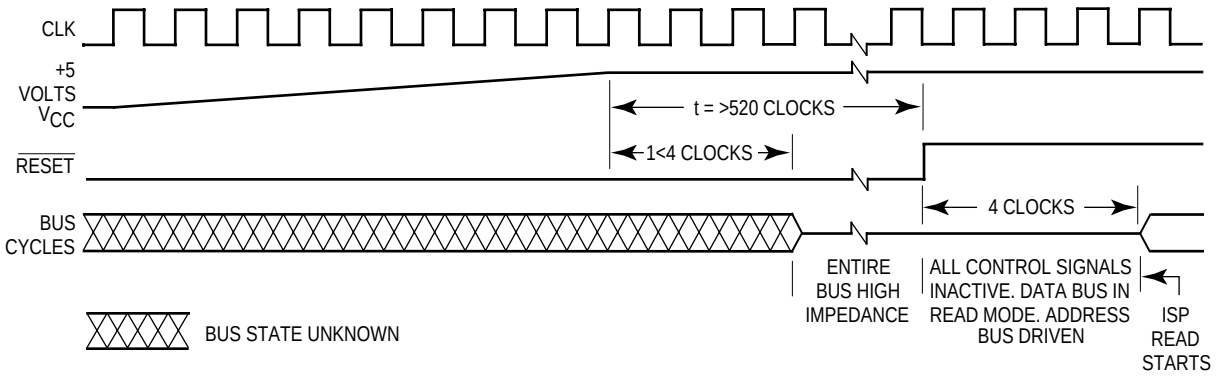


Figure 7-64. Initial Reset Operation Timing

Resetting the processor causes any bus cycle in progress to terminate as if $\overline{\text{DSACKx}}$, $\overline{\text{BERR}}$, or $\overline{\text{STERM}}$ had been asserted. In addition, the processor initializes registers appropriately for a reset exception. Exception processing for a reset operation is described in **8.1.1 Reset Exception**.

When a reset instruction is executed, the processor drives the $\overline{\text{RESET}}$ signal for 512 clock cycles. In this case, the processor resets the external devices of the system, and the internal registers of the processor are unaffected. The external devices connected to the $\overline{\text{RESET}}$ signal are reset at the completion of the reset instruction. An external $\overline{\text{RESET}}$ signal that is asserted to the processor during execution of a reset instruction must extend beyond the reset period of the instruction by at least eight clock cycles to reset the processor. Figure 7-65 shows the timing information for the reset instruction.

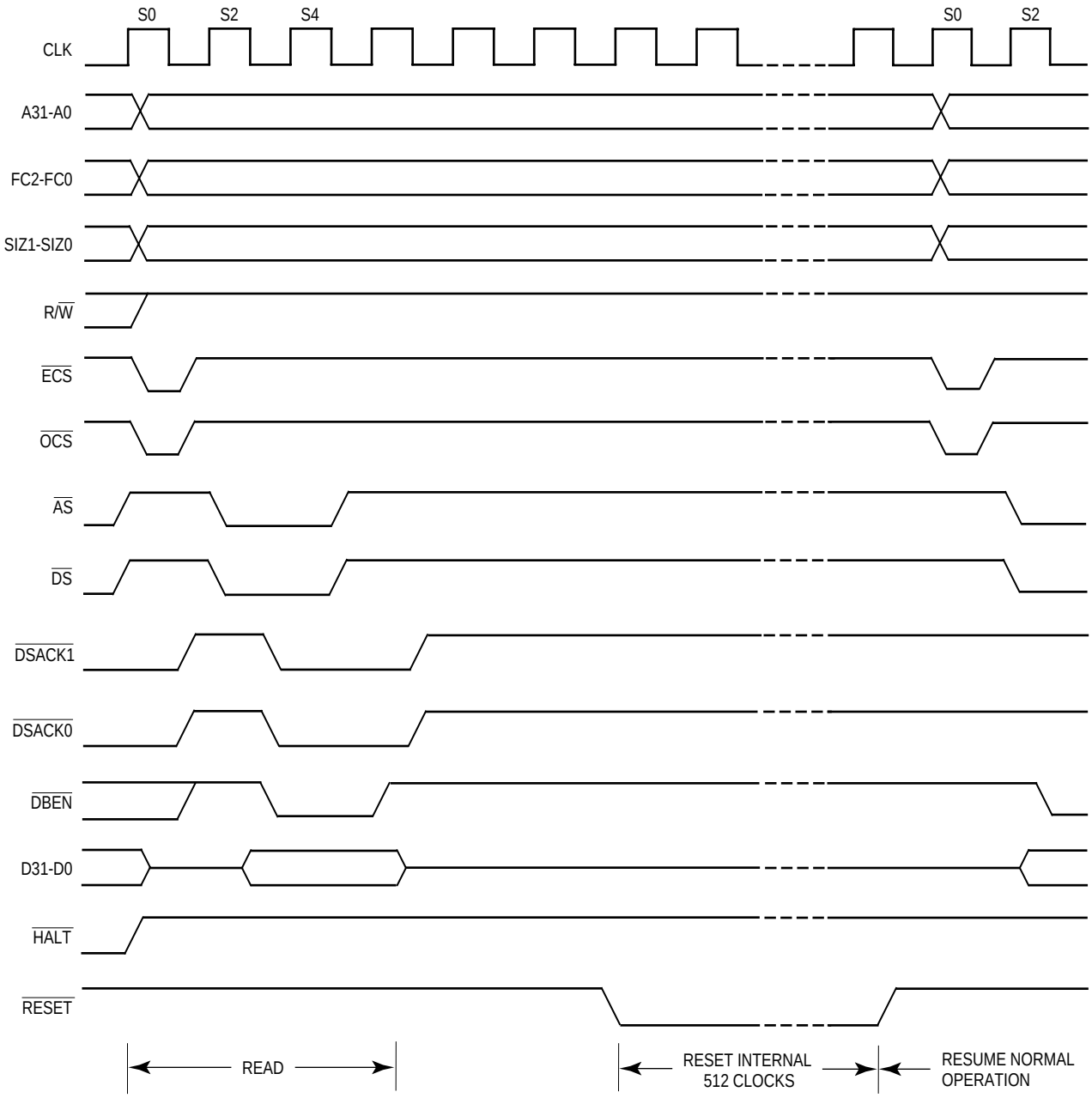


Figure 7-65. Processor-Generated Reset Operation