MC68040UM/AD

MC68040

32-BIT MICROPROCESSOR USER'S MANUAL

MOTOROLA

MC68040

32-BIT THIRD-GENERATON MICROPROCESSOR

PREFACE

The complete documentation package for the MC68040 consists of the MC68040UM/AD, *MC68040 User's Manaul*, the MC68000PM/AD, *M68000 Programmer's Reference Manual*, and the MC68040DH/AD, *MC68040 Designer's Handbook*.

The *MC68040 User' Manual* describes the capabilities, operation, and programming of the MC68040 32-bit third-generation microprocessor. The *M68000 Programmer's Reference Manaul* contains the complete instruction set of all the M68000 Family. The *MC68040 Designer's Handbook* contains detailed timing and electrical specifications and system design guidelines and information.

This user's manual is organized as follows:

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

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 (Concluded)

LIST OF TABLES

LIST OF TABLES (Continued)

LIST OF TABLES (Concluded)

SECTION 1
INTRODUCTION

The MC68040 is Motorola's third generation of M68000-compatible, high-performance, 32-bit microprocessors. The MC68040 is a virtual memory microprocessor employing multiple, concurrent execution units and a highly integrated architecture to provide very high performance in a monolithic HCMOS device. The MC68040 integrates an MC68030-compatible integer unit an MC68881/MC68882-compatible floating-point unit (FPU), dual independent demand-paged memory management units (MMUs) for instruction and data stream accesses, and independent 4K-byte instruction and data caches. A high degree of instruction execution parallelism is achieved through the use of multiple independent execution pipelines, multiple internal buses, and a full internal Harvard architecture, including the separate physical caches for both instruction and data accesses. Cache functionality is enhanced by the inclusion of on-chip bus snooping logic to directly support cache coherency in multimaster applications.

The MC68040 is user object-code compatible with previous members of the M68000 Family and is specifically optimized to reduce the execution time of compiler-generated code. The MC68040 is implemented in Motorola's latest HCMOS technology, providing an ideal balance between speed, power, and physical device size.

Figure 1-1 provides a simplified block diagram of the MC68040. Instruction execution is pipelined in both the integer unit and FPU, which interface to fully independent data and instruction memory units. Each memory unit consists of an MMU, an address translation cache (ATC), a main cache, and a snoop controller. The ATCs decrease logical-to-physical address translation overhead by storing recently-used translations, while the bus snooper circuit ensures cache coherency in multimaster applications. External memory requests from each cache are prioritized by the bus controller, which executes bus transfers on the external bus.

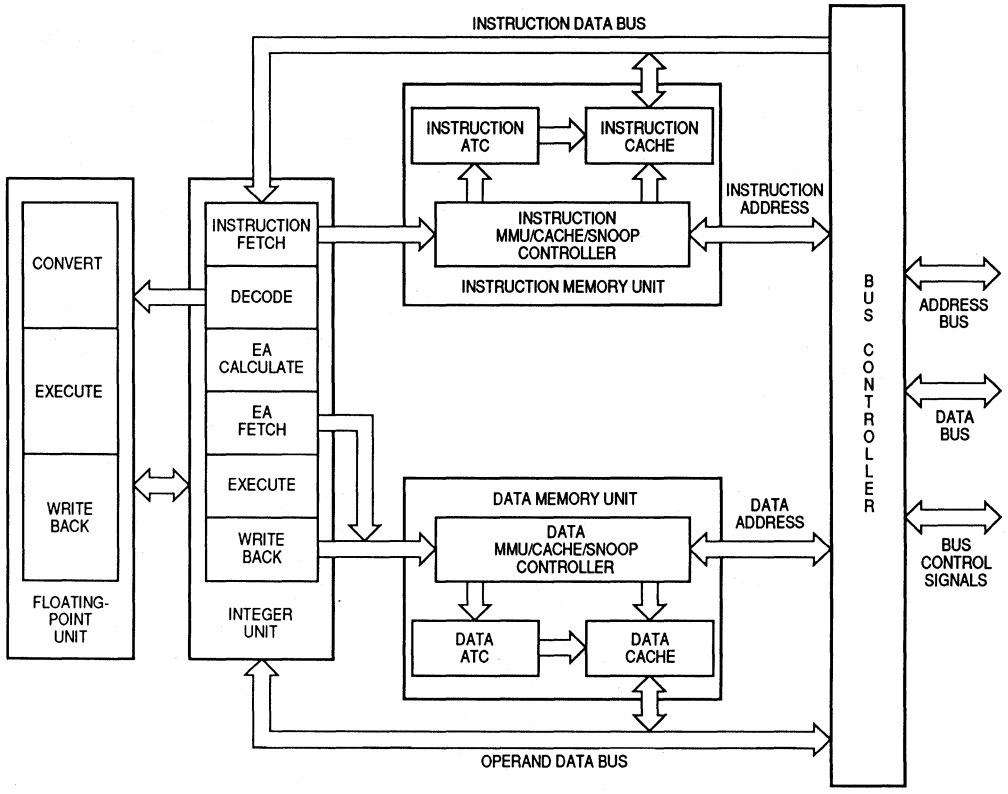


Figure 1-1. MC68040 Block Diagram

1.1 FEATURES

The main features of the MC68040 include:

- MC68030-Compatible Integer Execution Unit
- MC68881/MC68882-Compatible Floating-Point Execution Unit
- Independent Instruction and Data Memory Management Units (MMUs)
- 4K-Byte Physical Instruction Cache and 4K-Byte Physical Data Cache Accessible Simultaneously
- Low Latency Bus Accesses for Reduced Cache-Miss Penalty
- Multimaster/Multiprocessor Support via Bus Snooping
- Concurrent integer unit, FPU, MMU, and Bus Controller Operation Maximizes Throughput

- 32-Bit, Nonmultiplexed External Address and Data Buses with Synchronous Interface
- User Object-Code Compatibility with All Earlier M68000 Microprocessors
- 4-Gigabyte Direct Addressing Range
- Software Support Including Optimizing C Compiler and UNIX™ System V Port

The on-chip FPU and large physical instruction and data caches result in improved system performance and increased functionality. The independent instruction and data MMUs and increased internal parallelism also improve performance.

1.2 MC68040 EXTENSIONS TO THE M68000 FAMILY

The MC68040 contains an on-chip FPU which is user object-code compatible with the MC68882 floating-point coprocessor and is compatible with the ANSI/IEEE Standard 754 for binary floating-point arithmetic. The FPU has been optimized to execute the most commonly used subset of the MC68882 instruction set, and includes additional instruction formats for single- and double-precision rounding of results. Any floating-point instructions not directly supported in hardware are emulated in software. Floating-point instructions in the FPU execute concurrently with integer instructions in the integer unit.

The MC68040 integer unit pipeline has been expanded to include effective address calculation and operand fetch, with commonly used effective addressing modes further optimized. Conditional branches are optimized for the more common case of the branch taken, and both execution paths of the branch are fetched and decoded to minimize refilling of the instruction pipeline. The user instruction MOVE16 has been added to the instruction set to support efficient 16-byte memory-to-memory data transfers.

Memory management in the MC68040 has been improved by including separate, independent paged MMUs for instruction and data accesses. Each MMU stores recently used address mappings in separate 64-entry ATCs. Table searches are performed with hardwired logic instead of microcode in order to minimize search time. Each MMU also has two transparent translation registers available that define a one-to-one mapping for address space segments ranging in size from 16 Mbytes to 4 Gbytes each.

UNIX is a registered trademark of AT&T Bell Laboratories.

Separate 4K-byte on-chip instruction and data caches operate independently, and are accessed in parallel with address translation. Each cache and corresponding MMU resides on a separate internal address bus and data bus, allowing simultaneous access to both. The data cache provides writethrough or copyback write modes that can be configured on a page-by-page basis. The caches are physically mapped, reducing software support for multitasking operating systems, and support external bus snooping to maintain cache coherency in multimaster systems.

The MC68040 bus controller supports a high-speed, nonmultiplexed synchronous external bus interface. Burst accesses are supported for both reads and writes to provide high data transfer rates to and from the caches. Additional bus signals support bus snooping and external cache tag maintenance.

1.3 PROGRAMMING MODEL

The MC68040 integrates the functions of the integer unit, MMU, and FPU. The registers depicted in the programming model (see Figure 1-2) provide operand storage and control for the three units. The registers are partitioned into two levels of privilege: user and supervisor. User programs, executing in the lower-privilege mode, can only use the resources of the user model. System software executing in the supervisor mode has unrestricted access to all processor resources.

The user programming model consists of 16 general-purpose, 32-bit registers and two control registers, and is the same as the user programming model of the MC68030. The MC68040 user programming model also incorporates the MC68882 programming model consisting of eight, 80-bit floating-point data registers, a floating-point control register, a floating-point status register, and a floating-point instruction address register.

The supervisor programming model is used exclusively by MC68040 system programmers to implement operating system functions, I/O control, and memory management subsystems. This supervisor/user distinction in the M68000 architecture allows all application software to be written to execute in the nonprivileged user mode and migrate to the MC68040 from any M68000 platform without modification. Since system software is usually modified by system designers when porting to a new design, the control features are properly placed in the supervisor programming model. For example, the

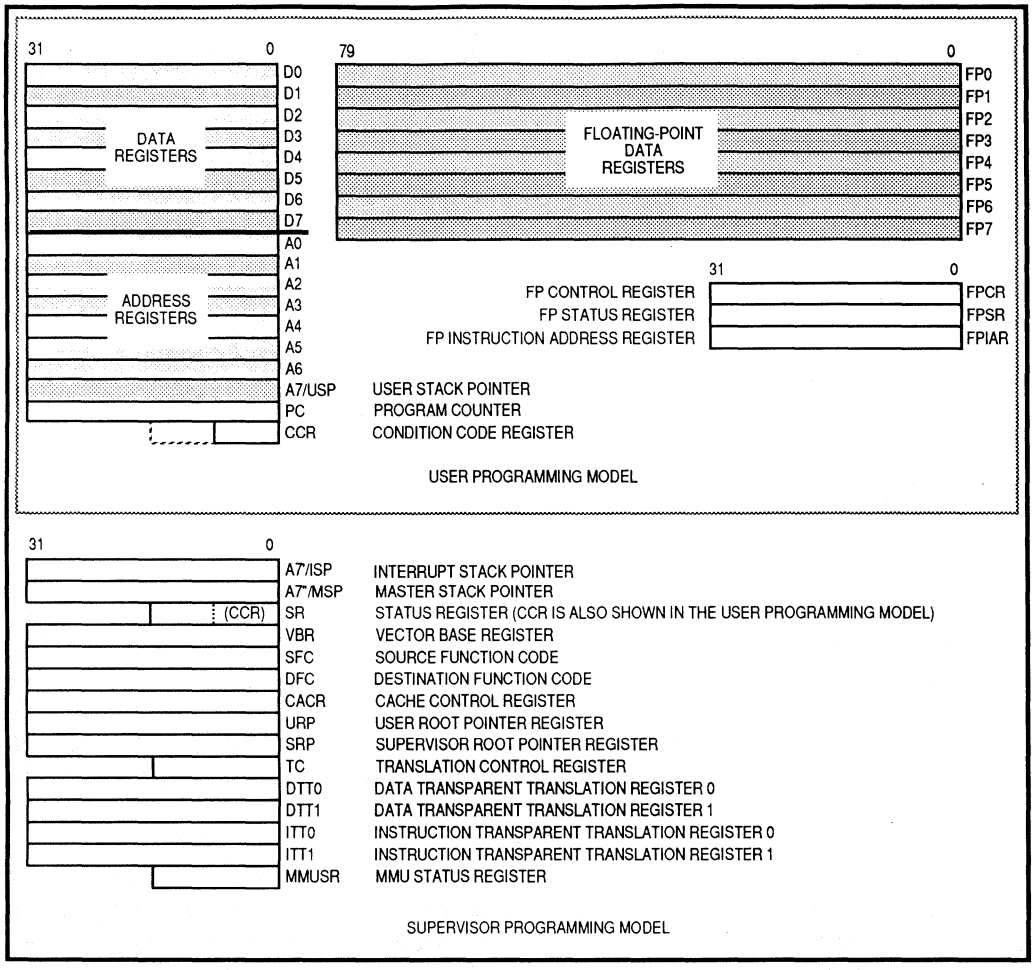


Figure 1-2. Programming Model

transparent translation registers of the MC68040 can only be read or written by the supervisor software. Programming resources of user application programs are unaffected by the existence of the transparent translation registers.

Data registers (D7–D0) contain operands for bit and bit field, byte, word, long-word, and quad-word operations. Address registers (A6–A0) and the stack pointer register (A7) are address registers may be used as software stack pointers or base address registers. Register A7, is also used as an user stack pointer in user mode, and is either the interrupt (A7') or master stack pointer (A7'') in supervisor mode. In supervisor mode, the active stack pointer (interrupt or master) is selected based on the setting of the master (M) bit

in the status register (SR). In addition, A7–A0 may be used for word and long-word operations. Registers D7–D0 and A7–A0 may be used as index registers.

The eight, 80-bit, floating-point data registers (FP7–FP0) are analogous to the integer data registers of all M68000 Family processors. Floating-point data registers always contain extended-precision numbers. All external operands, regardless of the data format, are converted to extended-precision values before being used in any floating-point calculation or stored in a floating-point data register.

The program counter (PC) usually contains the address of the instruction being executed by the MC68040. 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. The status register (SR) contains the condition codes that reflect the results of a previous operation and can be used for conditional instruction execution in a program. The lower byte of the SR is accessible in user mode as the condition code register (CCR). Access to the upper byte of the SR, which contains operation control information, is restricted to the supervisor mode.

As part of exception processing, the vector number of the exception provides an index into the exception vector table. The base address of the exception vector table is stored in the vector base register (VBR).

Alternate source function code (SFC) and destination function code (DFC) registers contain 3-bit function codes, which can be considered extensions of the 32-bit logical address. Function codes are automatically generated by the processor to select address spaces for data and program accesses in the user and supervisor modes. The alternate function code registers are used by certain instructions to explicitly specify the function codes for various operations.

The cache control register (CACR) controls enabling of the on-chip instruction and data caches of the MC68040.

The supervisor root pointer (SRP) and user root pointer (URP) registers point to the root of the address translation table tree to be used for supervisor mode and user mode accesses. The URP is used if function code bit 2 (FC2) of the logical address is zero, and the SRP is used if FC2 is one.

The translation control register (TC) enables logical-to-physical address translation and selects either 4K or 8K page sizes. There are four transparent translation registers: two for instruction accesses (ITT1–ITT0), and two for data accesses (DTT1–DTT0). These registers allow portions of the logical address space to be transparently mapped and accessed without the use of resident descriptors in an ATC. The MMU status register (MMUSR) contains status information derived from the execution of a PTEST instruction. The PTEST instruction searches the translation tables for the logical address as specified by this instruction's effective address field and the DFC, and returns status information corresponding to the translation.

The 32-bit floating-point control register (FPCR) contains an exception enable byte that enables/disables traps for each class of floating-point exceptions and a mode byte that sets the user-selectable modes. The FPCR can be read or written to by the user and is cleared by a hardware reset or a restore operation of the null state. When cleared, the FPCR provides the IEEE 754 standard defaults for floating-point exceptions. The floating-point status register (FPSR) contains a condition code byte, quotient bits, an exception status byte, and an accrued exception byte. All bits in the FPSR can be read or written by the user. Execution of most floating-point instructions modifies this register.

For the subset of the FPU instructions that generate exception traps, the 32-bit floating-point instruction address register (FPIAR) is loaded with the logical address of an instruction before the instruction is executed. This address can then be used by a floating-point exception handler to locate a floating-point instruction that has caused an exception. The floating-point instructions FMOVE (to/from the FPCR, FPSR, or FPIAR) and FMOVEM cannot generate floating-point exceptions; therefore, these instructions do not modify the FPIAR. Thus, the FMOVE and FMOVEM instructions can be used to read the FPIAR in the trap handler without changing the previous value.

1.4 DATA TYPES AND ADDRESSING MODES

The MC68040 supports the basic data types shown in Table 1-1. Some data types apply only to the integer unit, some only to the FPU, and some to both. In addition, the instruction set supports operations on other data types such as memory addresses.

Table 1-1. Data Types

Operand Data Type	Size	Supported By:	Notes
Bit	1 Bit	IU	—
Bit Field	1-32 Bits	IU	Field of Consecutive Bit
BCD	8 Bits	IU	Packed: 2 Digits/Byte Unpacked: 1 Digit/Byte
Byte Integer	8 Bits	IU, FPU	—
Word Integer	16 Bits	IU, FPU	—
Long-Word Integer	32 Bits	IU, FPU	—
Quad-Word Integer	64 Bits	IU	Any Two Data Registers
16-Byte	128 Bits	IU	Memory-Only, Aligned to 16-Byte Boundary
Single-Precision Real	32 Bits	FPU	1-Bit Sign, 8-Bit Exponent, 23-Bit Mantissa
Double-Precision Real	64 Bits	FPU	1-Bit Sign, 11-Bit Exponent, 52-Bit Mantissa
Extended-Precision Real	80 Bits	FPU	1-Bit Sign, 15-Bit Exponent, 64-Bit Mantissa

IU = Integer Unit

The three integer data formats that are common to both the integer unit and the FPU (byte, word, and long word) are the standard twos-complement data formats defined in the M68000 Family architecture. Whenever an integer is used in a floating-point operation, the integer is automatically converted by the FPU to an extended-precision floating-point number before being used.

Single- and double-precision floating-point data formats are implemented in the FPU as defined by the IEEE standard. These data formats are used for most calculations with real numbers.

The extended-precision data format is also in conformance with the IEEE standard, but the standard does not specify this format to the bit level as it does for single- and double-precision. The memory format for the FPU consists of 96 bits (three long words). Only 80 bits are actually used; the other 16 bits are for future expansibility and for long-word alignment of the floating-point data structures in memory. The extended-precision format has a 15-bit exponent, a 64-bit mantissa, and a 1-bit mantissa sign. Extended-precision numbers are intended for use as temporary variables, intermediate values, or where extra precision is needed.

The MC68040 addressing modes are shown in Table 1-2. The register indirect addressing modes support postincrement, predecrement, offset, and indexing, which are particularly useful for handling data structures common to sophisticated applications and high-level languages. The program counter indirect mode also has indexing and offset capabilities; this addressing mode

is typically required to support position-independent software. In addition to these addressing modes, the MC68040 provides index sizing and scaling features that enhance software performance. Data formats are supported orthogonally by all arithmetic operations and by all appropriate addressing modes.

Table 1-2. 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-Bit Displacement)	(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)
Porgram Counter Indirect with Index	
PC 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, D7–D0
An = Address Register, A7–A0
d_8, d_{16} = A twos-complement or sign-extended displacement; added as port 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 effectie address calculation after any memory indirection; use is optional with size of 16 or 32 bits.
PC = Program Counter
<data> = Immediate value of 8, 16, or 32 bits.
( ) = Effective Address
[ ] = Used as indirect access to long-word address.

1.5 INSTRUCTION SET OVERVIEW

The instructions provided by the MC68040 are listed in Table 1-3. The instruction set has been tailored to support high-level languages and is optimized for those instructions most commonly executed (however, all instructions listed are fully supported). Many instructions operate on bytes, words, and long words, and most instructions can use any of the addressing modes of Table 1-2.

The floating-point instructions for the MC68040 are a commonly used subset of the MC68881/MC68882 instruction set with new arithmetic instructions to explicitly select single- or double-precision rounding. The remaining unimplemented instructions are less frequently used and are efficiently emulated in software, maintaining compatibility with the MC68881/MC68882 floating-point coprocessors.

The MC68040 instruction set includes MOVE16, a new user instruction which allows high-speed transfers of 16-byte blocks between external devices such as memory-to-memory, or coprocessor-to-memory.

1.6 MEMORY MANAGEMENT UNITS

The data and instruction MMUs support virtual memory systems by translating logical addresses to physical addresses using translation tables stored in memory. Each MMU stores recently used address mappings in an ATC, reducing average translation time. Features of the MMUs include:

- Instruction and Data MMUs are Fully Independent
- 64-Entry, Four-Way Set-Associative ATCs
- Table Searches Automatically Performed by Hardware
- Address Translation and Cache Indexing Performed in Parallel
- 4K or 8K Page Sizes
- Two Optional Transparent Blocks for each MMU
- Fixed Three-Level Translation Table Structure with Optional Indirection
- User and Supervisor Root Pointer Registers
- Global Attribute for Selective ATC Flushing
- Write Protection and Supervisor Protection Attributes

Table 1-3. Instruction Set Summary

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	Arithmetic Shift Left and Right
Bcc	Branch Conditionally
BCHG	Test Bit and Change
BCLR	Test Bit and Clear
BFCHG	Test Bit Field and Change
BFCLR	Test Bit Field and Clear
BFEXTS	Signed Bit Field Extract
BFEXTU	Unsigned Bit Field Extract
BFFFO	Bit Field Find First One
BFINS	Bit Field Insert
BFSET	Test Bit Field and Set
BFTST	Test Bit Field
BRA	Branch
BSET	Test Bit and Set
BSR	Branch to Subroutine
BTST	Test Bit
CAS	Compare and Swap Operands
CAS2	Compare and Swap Dual Operands
CHK	Check Register Against Bounds
CHK2	Check Register Against Upper and Lower Bounds
*CINV	Invalidate Cache Entries
CLR	Clear
CMP	Compare
CMPA	Compare Address
CMPI	Compare Immediate
CMPM	Compare Memory to Memory
CMP2	Compare Register Against Upper and Lower Bounds
*CPUSH	Push then Invalidate Cache Entries
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
*FABS	Floating-Point Absolute Value
*FADD	Floating-Point Add
FBcc	Floating-Point Branch
FCMP	Floating-Point Compare
FDBcc	Floating-Point Decrement and Branch
*FDIV	Floating-Point Divide
*FMOVE	Move Floating-Point Register
FMOVEM	Move Multiple Floating-Point Registers
*FMUL	Floating-Point Multiply
*FNEG	Floating-Point Negate
FRESTORE	Restore Floating-Point Internal State
FSAVE	Save Floating-Point Internal State
FScc	Floating-Point Set According to Condition
*FSQRT	Floating-Point Square Root
*FSUB	Floating-Point Subtract
FTRAPcc	Floating-Point Trap-On Condition
FTST	Floating-Point Test
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
*MOVE16	16-Byte Block Move
MOVEA	Move Address
MOVE CCR	Move Condition Code Register
MOVE SR	Move Status Register
MOVE USP	Move User Stack Pointer
*MOVEC	Move Control Register
MOVEM	Move Multiple Registers
MOVEP	Move Peripheral
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 Complement
OR	Logical Inclusive OR
ORI	Logical Inclusive OR Immediate
PACK	Pack BCD
PEA	Push Effective Address
*PFLUSH	Flush Entry(ies) in the ATCs
*PTEST	Test a Logical Address
RESET	Reset External Devices
ROL, ROR	Rotate Left and Right
ROXL, RORX	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 Address
SUBI	Subtract Immediate
SUBQ	Subtract Quick
SUBX	Subtract with Extend
SWAP	Swap Register Words
TAS	Test Operand and Set
TRAP	Trap
TRAPcc	Trap Conditionally
TRAPV	Tap on Overflow
TST	Test Operand
UNLK	Unlink
UNPK	Unpack BCD

*MC68040 additions or alterations to the MC68030 and MC68881/M68882 instruction set.

- Pages may be Specified as Writethrough, Copyback, Noncachable, or Noncachable I/O
- Translations Enabled/Disabled by Software
- Used and Modified Status Automatically Maintained in Tables and ATCs
- Translations can be Disabled by External MMUDIS Signal
- Cache Inhibit Out (CIOUT) Signals can be Asserted on a Page-by-Page Basis
- 32-Bit Physical Address with Two User-Defined Attribute Signals

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 MC68040 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. The memory management software 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.

1.7 INSTRUCTION AND DATA CACHES

Because of the phenomenon of 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 residing near the instructions and data currently in use also have a high probability of being utilized within a short period. The MC68040 takes advantage of these locality characteristics with its two on-chip physical caches, one for instructions and one for data. Both caches are organized as four-way set-associated with 64 sets of four lines. Each line contains four long words, for a storage capability of 4K bytes for each cache, or 8K bytes total. The processor fills the cache lines using burst mode accesses which transfer the entire line as four long words. This 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 transfers required by the processor to fetch information from memory and by increasing the bus bandwidth available for other bus masters in the system. To further improve system performance, the data cache in the MC68040 supports both copyback and writethrough caching modes for storing write accesses. For writes that hit in copyback pages, the data is used to update the cache line without writing the data to memory immediately. This "dirty" data is copied to memory only when required, either to allow replacement of the cache line by new data, or as a result of explicit flushing of the cache line, resulting in a lower bus bandwidth requirement for the processor. Cache coherency for both caches is maintained by bus snooping logic which allows the MC68040 to monitor accesses by an alternate bus master. When an alternate master performs bus transfers, the MC68040 can update cache lines which hit during an external write, or source data from dirty data cache lines while inhibiting data from memory during external reads.

SECTION 2
PROGRAMMING MODEL

This section describes the MC68040 programming model, which is separated into the user and supervisor programming models. User programs, executing at the user privilege level, can only use the registers of the user model. System software executing in the supervisor mode has access to all registers and uses the control registers of the supervisor mode to perform supervisor functions. A brief description of the registers accessible at each level is presented in the following paragraphs.

2.1 PROCESSING STATES

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.

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 the filling of the instruction pipe caused by an exception. This processing is 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 any other exceptional condition arises. Execution of certain instructions or unusual internal conditions that occur during the execution of any instructions can cause exceptions. External conditions, such as interrupts and bus errors, 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 exception processing of one bus error another bus error occurs, the MC68040 has not

completed the transition to normal processing and has not completed saving the internal state of the machine; thus, 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. The processor is stopped, not halted.)

2.1.1 Privilege Levels

The processor operates in one of two privilege modes: user or supervisor. The supervisor mode has higher privileges than the user mode. Not all instructions are permitted to execute in the user mode, but all are available in the supervisor mode. This difference allows the supervisor to protect system resources from uncontrolled access. The processor uses the privilege mode indicated by the S bit in the status register (SR) to select either the user or supervisor mode and either the user stack pointer (USP) or a supervisor stack pointer (SP) for stack operations. The integer unit identifies a logical address as accessing either the supervisor or user address space so that differentiation between supervisor and user can be maintained. The memory management units (MMUs) use the indicated privilege mode to control and translate memory accesses to protect supervisor code, data, and resources from access by user programs.

In many systems, most programs execute in the user privilege mode. User programs access only their own code and data areas and are restricted from accessing other information. Executing in the supervisor privilege mode, the operating system has access to all resources, performs management and service tasks for the user-level programs, and coordinates their activities.

2.1.1.1 SUPERVISOR MODE. The supervisor mode is the higher privilege level. The privilege level is determined by the S bit of the SR; if set, the processor executes instructions in the supervisor mode. The bus cycles for instructions executed in the supervisor mode are normally classified as supervisor references, and the values on the transfer modifier pins (TM2–TM0) indicate supervisor accesses.

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

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 MC68040 provides two supervisor stack pointers, master (MSP) and interrupt (ISP); the M bit of the SR selects which of the two is active. When the M bit is set , supervisor stack pointer references (either implicit or by specifying address register A7) access the MSP. The operating system sets the MSP for each task to point to a task-related area of supervisor data space. This procedure separates task-related supervisor activity from asynchronous, I/O-related supervisor tasks that may be only coincidental to the currently executing task. The 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 ISP, can be used for interrupt control information and for workspace area as interrupt handling routines require.

When the M bit is clear, the MC68040 is in the interrupt mode of the supervisor privilege level, and operation is the same as in any other M68000 Family processor when in supervisor mode. This mode is the default condition after reset, and all supervisor stack pointer references access the ISP.

The value of the M bit in the SR does not affect execution of privileged instructions. Instructions that affect the M bit are MOVE to SR, ANDI to SR, EORI to SR, ORI to SR, and RTE. The processor automatically saves the M-bit value and clears it in the SR as part of the exception processing for interrupts.

2.1.1.2 USER MODE. The user mode is the lower privilege level. If the S bit of the SR is clear, the processor executes instructions in the user mode. Most instructions execute at either privilege level, but some instructions that have important system effects are privileged and can only execute in the supervisor mode. For instance, user programs cannot execute the STOP or RESET instructions. To prevent a user program from entering the supervisor mode, 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 in the user mode are classified as user references, and the values on the signals TM2–TM0 indicate user accesses. When enabled, the MMUs use the indicated privilege level to distinguish between user and supervisor activity and to control access to protected

portions of the address space. While the processor is operating in the user mode, explicit references to the system stack pointer or to address register seven (A7) refer to the user stack pointer.

2.1.1.3 CHANGING PRIVILEGE LEVEL. During exception processing the processor changes from user to supervisor mode. Exception processing saves the current value of the SR on the active supervisor stack and then sets the S bit, forcing the processor into the supervisor mode. 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 in the privileged exception handler to process the exception condition.

To return to the user mode, a system routine must execute one of the following instructions: MOVE to SR, ANDI to SR, EORI to SR, ORI to SR, or RTE. These instructions, which execute in the supervisor mode, can modify the S bit of the SR. After these instructions execute, the instruction pipeline is flushed and is refilled from the appropriate address space.

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

2.1.2 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, and reset. Instructions, address errors, and tracing are internal conditions that cause exceptions. For example, the TRAP, TRAPcc, FTRAPcc, CHK, RTE, DIV, and FDIV instructions can generate exceptions as part of their normal execution. In addition, illegal instructions, unimplemented floating-point instructions and data types, and privilege violations cause exceptions.

Exception processing, which is the transition from the normal processing of a program to the processing required for the exception condition, uses the exception vector table and an exception stack frame. The following paragraphs describes the vector table and a generalized exception stack frame.

Exception processing is discussed in detail in **SECTION 9 EXCEPTION PROCESSING**.

2.1.2.1 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 vector consists of one long word, except for the reset vector. The reset vector consists of two long words: the address used to initialize the ISP and the address used to initialize the PC.

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 accessed as supervisor data references, except the reset vector, which is accessed as a supervisor program reference. 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. Details of exception processing are provided in **SECTION 9 EXCEPTION PROCESSING**.

2.1.2.2 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 SR, the PC, 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 frame format field to properly restore the information stored in the stack frame and to deallocate the stack space appropriately. The general form of the exception stack frame is illustrated in Figure 2-1. Refer to **SECTION 9 EXCEPTION PROCESSING** for a complete description of the various exception stack frames.

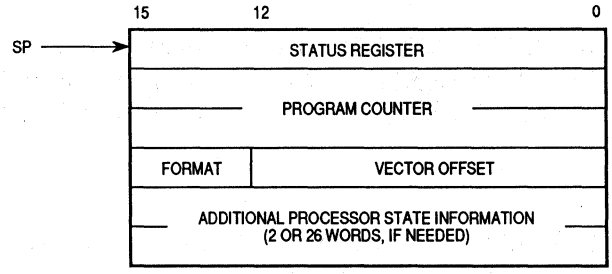


Figure 2-1. General Form of Exception Stack Frame

2.2 REGISTER DESCRIPTION

The programming model of the MC68040 consists of two groups of registers: the user model and the supervisor model, which correspond to the user and supervisor modes. User programs, executing in the user mode, can only use the registers of the user model. System software executing in the supervisor mode has access to all registers and uses the control registers of the supervisor model to perform supervisor functions. The following paragraphs provide a brief description of the registers in the user and supervisor models.

2.2.1 User Programming Model

The user programming model is shown in detail in Figure 2-2. The integer portion of the user programming model, which is the same as previous M68000 Family microprocessors, consists of the following registers:

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

The floating-point portion of the user programming model, which is identical to the programming model for the MC68881/MC68882 floating-point coprocessors, consists of the following registers:

- 8 Floating-Point Data Registers (FP7–FP0)
- 16-Bit Floating-Point Control Register (FPCR)
- 32-Bit Floating-Point Status Register (FPSR)
- 32-Bit Floating-Point Instruction Address Register (FPIAR)

The following paragraphs described each group of registers.

2.2.1.1 DATA REGISTERS (D7–D0). These registers 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. These registers may also be used as index registers.

2.2.1.2 ADDRESS REGISTERS (A7–A0). These registers may be used as software stack pointers, index registers, or base address registers. The address registers may be used for word and long-word operations.

Register A7 is used as a hardware stack pointer during stacking for subroutine calls and exception handling. The register designation A7 refers to three different registers: the USP (A7) in the user programming model and either the ISP or MSP (A7′ and A7″) in the supervisor programming model. In the supervisor programming model, the active stack pointer (ISP or MSP) is called the supervisor stack pointer.

2.2.1.3 PROGRAM COUNTER (PC). The PC contains the address of the currently executing instruction. During instruction execution and exception processing, the processor automatically increments the contents of the program counter or places a new value in the program counter, as appropriate. For some addressing modes the PC may used as a pointer for PC-relative addressing.

2.2.1.4 CONDITION CODE REGISTER (CCR). The CCR is the lower byte of the SR and is the only portion of the SR available in the user mode. See **2.2.2.2 STATUS REGISTER (SR)** for further information.

2.2.1.5 FLOATING-POINT DATA REGISTERS (FP7–FP0). These floating-point data registers are analogous to the integer data registers of all M68000 Family processors. The floating-point data registers always contain extended-precision numbers. All external operands, regardless of the data format, are converted to extended-precision values before being used in any calculation or stored in a floating-point data register. A reset or a null-restore operation sets FP7–FP0 to positive, nonsignaling not-a-numbers (NANs). For a complete description of NANs and floating-point data formats, see **SECTION 3 DATA ORGANIZATION AND ADDRESSING CAPABILITIES**.

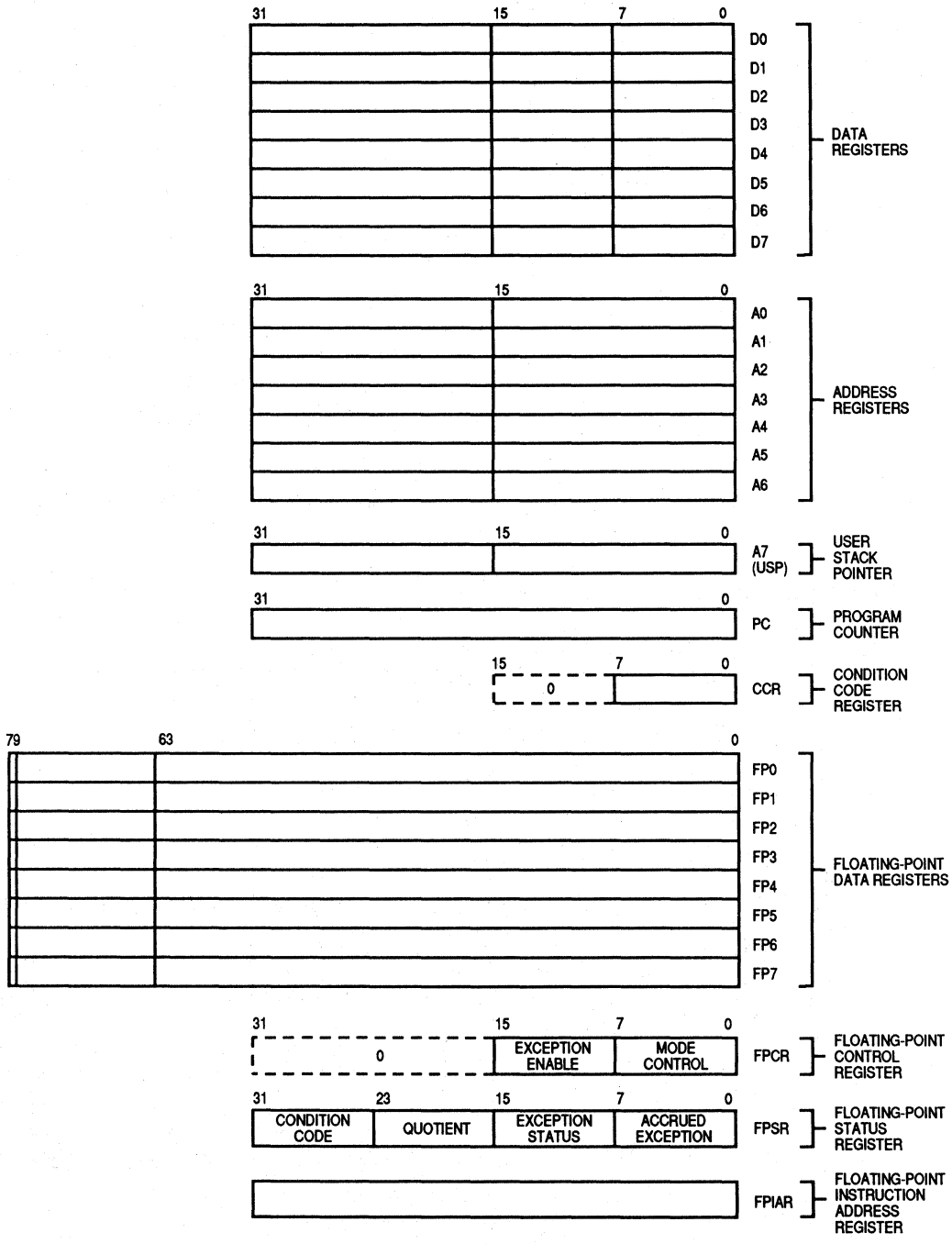


Figure 2-2. User Programming Model

2.2.1.6 FLOATING-POINT CONTROL REGISTER (FPCR). The FPCR (see Figure 2-2) contains an exception enable byte that enables/disables traps for each class of floating-point exceptions and a mode byte that sets the user selectable modes. The FPCR can be read or written to by the user. Bits 16 through 31 are reserved for future definition by Motorola. These bits are always read as zero and are ignored during write operations. The FPCR is cleared by the reset function or a restore operation of the null state. When cleared, this register provides the IEEE standard defaults.

2.2.1.6.1 Exception Enable Byte. One of the bits of the exception enable byte (ENABLE) (see Figure 2-3) corresponds to each floating-point exception class. The user can separately enable traps for each class of floating point-point exceptions.

When the processor set a bit in the FPSR EXC byte and the corresponding bit in the FPCR ENABLE byte is also set, an exception is signaled. The address of the exception handler is derived from the vector address corresponding to the exception. When a user writes to the ENABLE byte that enables a class of floating-point exceptions. a previously generated floating-point exception does not cause a trap to be taken regardless of the value in the FPSR EXC byte.

The bits in the FPSR EXC byte and FPCR enable byte occupy the same positions within each byte. Dual and triple exceptions can be generated by a single instruction execution. When multiple exceptions occur with traps enable for more than one exception class, the highest priority exception is reported; the lower priority exceptions are never reported or taken. The exception handler must check for multiple exceptions. The bits of the ENABLE byte are organized in decreasing priority with bit 15 being the highest and bit 8 the lowest.

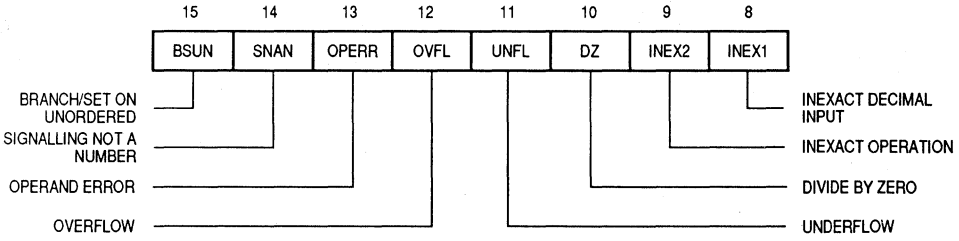


Figure 2-3. FPCR Exception Enable Byte

2.2.1.6.2 MODE CONTROL BYTE. The mode control byte (MODE) (see Figure 2-4) controls the user-selectable rounding modes and rounding precisions. A zero in this byte selects the IEEE defaults.

The rounding mode (RND) specifies how inexact results are rounded. Refer to **SECTION 9 EXCEPTIONS** for a detailed description of the rounding algorithm used.

The rounding precision (PREC) selects where rounding of the mantissa occurs. For extended precision, the results is rounded to a 64-bit boundary; single precision results is rounded to a 24-bit boundary, and double precision is rounded to a 53-bit boundary.

The single and double rounding precisions are provided for emulation of machines that only support those precisions. When the MC68040 performs any operation, the calculation is carried out using extended precision inputs and the intermediate result is calculated as if to produce infinite precision. After the calculation is complete, this intermediate result is rounded to the selected precision and stored in the destination.

If the destination is a floating-point data register (FP0–FP7), the stored value is in the extended precision format rounded to the precision specified by the PREC bits. Thus, all mantissa bits beyond the selected precision are zero after the rounding operation. If the single or double precision mode is selected, the exponent value is in the correct range for the single or double precision format (although it is stored in extended precision format).

If the destination is memory location, the PREC bits are ignored. In this case, a number in the extended precision format is taken from the source floating-point data register, rounded to the destination format precision and then written to memory.

The execution speed of all instructions is degraded significantly when single or double precision rounding modes are used. When operating in these modes, the MC68040 produces the same results as any other machine that conforms to the IEEE standard without supporting extended precision calculations. The results may not be the same as performing the same operation in extended precision and storing the results in the single or double precision format.

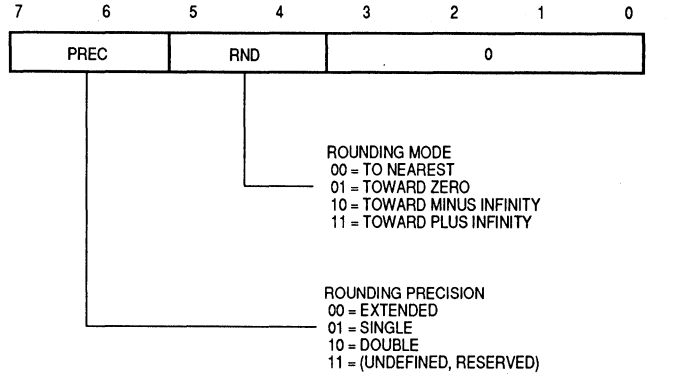


Figure 2-4. FPCR Mode Control Byte

2.2.1.7 FLOATING-POINT STATUS REGISTER (FPSR). The FPSR (see Figure 2-2) contains a floating-point condition code byte (FPCC), a floating-point exception status byte (EXC), quotient bits, and a floating-point accrued exception byte (AEXC). All bits in the FPSR can be read or written by the user. Execution of most floating-point instructions modifies this register. The reset function or a restore operation of the null state clears the FPSR.

2.2.1.7.1 Foating-Point Condition Code Byte. The floating-point condition code (FPCC) byte (see Figure 2-5) contains four condition code bits that are set at the end of all arithmetic instructions involving the floating-point data registers. The FMOVE FPm, ⟨ea⟩, move multiple floating-point data register, and move system control register instructions do not affect the FPCC.

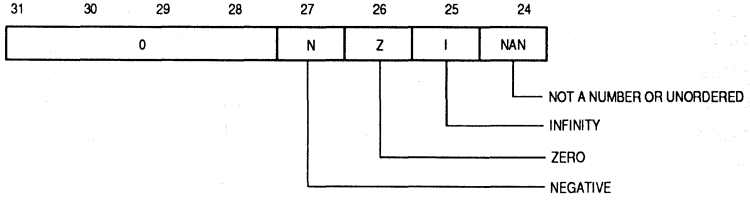


Figure 2-5. FPSR Condition Code Byte

The operation result data type determines how the four condition code bits are set. Table 2-1 lists the condition code bit setting for each result data type.The MC68040 generates only eight of the 16 possible combinations. Loading the FPCC with one of the other combinations and executing a conditional instruction may produce an unexpected branch condition.

Table 2-1. Condition Code versus Results Data Type

N	Z	I	NAN	Results Data Type
0	0	0	0	+ Normalized or Denormalized
1	0	0	0	− Normalized or Denormalized
0	1	0	0	+ 0
1	1	0	0	− 0
0	0	1	0	+ Infinity
1	0	1	0	− Infinity
0	0	0	1	+ NAN
1	0	0	1	− NAN

The IEEE standard defines the following four conditions and only requires the generation of the condition codes as a result of a floating-point compare operation. In addition to this requirement, the FPCP can test these conditions at the end of any operation affecting the condition codes.

EQ Equal To
GT Greater Than
LT Less Than
UN Unordered

An unordered condition occurs when one or both of the operands in a floating-point compare operation is a NAN. For purposes of the floating-point conditional branch, set byte on condition, decrement and branch on condition, and trap on condition instructions, the MC68040 logically combines the four condition codes to form the IEEE conditions according to the following equations:

$$EQ = Z$$
$$GT = \overline{NvNANvZ}$$
$$LT = N\Lambda\overline{NANvZ}$$
$$UN = NAN$$

where:

"Λ" = Logical AND
"v" = Logical OR

Note that the setting of the floating-point condition codes is independent of the operation executed; the condition codes only indicate that data type of the result generated. Unlike other M68000 condition codes, the IEEE defined conditions can always be derived from the data type of the result. The setting of the M68000 integer condition codes is dependent upon the operation executed as well as the result.

To aid programmers of floating-point subroutine libraries, the MC68040 implements the four previously described floating-point condition code bits in hardware instead of the four IEEE defined conditions. The IEEE conditions are derived by an instruction when needed. For example, the programmers of a complex arithmetic multiply subroutine usually prefers to handle "special" data types such as zeros, infinities, or NANs, separately from "normal" data types. The floating-point condition codes allow users to efficiently detect and handle these "special" values.

2.2.1.7.2 Quotient Byte. The quotient byte (see Figure 2-6) is provided for compatibility with MC68881/MC68882 Floating-Point Unit. This byte contains the seven least-significant bits of the quotient (unsigned) and the sign of the entire quotient.

The quotient bits can be used in argument reduction for transcendentals and other functions. For example, seven bits are more than enough to determine the quadrant of a circle in which an operand resides. The quotient bits remain set until they are cleared by the user.

2.2.1.7.3 Exception Status Byte. The exception status byte (EXC) (see Figure 2-7) contains a bit for each floating-point exceptions that may have occurred during the most recent arithmetic instruction or move operation. This byte

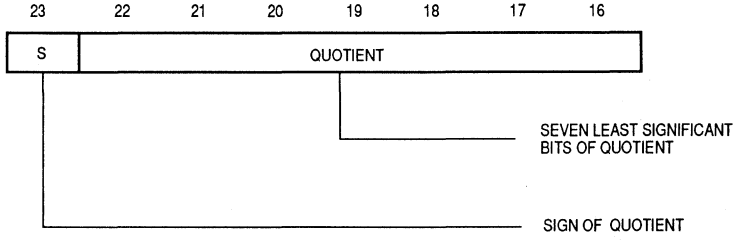


Figure 2-6. FPSR Quotient Byte

is cleared at the start of most operations; operations that cannot generate any floating point exceptions do not clear this byte. This byte can be used by an exception handler to determine which floating- point exception(s) caused a trap.

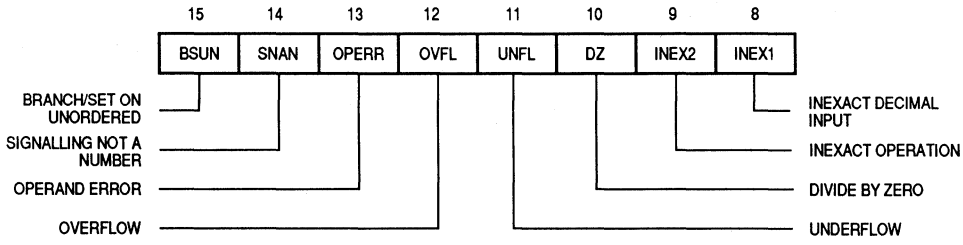


Figure 2-7. FPSR Exception Status Byte

If a bit is set in the EXC byte and the corresponding bit in the ENABLE byte in the floating-point control register is also set, an exception is signaled. When a floating- point exceptions is detected by the MC68040, the corresponding bit in the EXC byte is set, even if the trap for that exception class is disabled. (A user write operation to the FPSR, which sets a bit in the EXC byte, does not cause a trap to be taken regardless of the value in the ENABLE byte).

2.2.1.7.4 Accured Exception Byte. The accured exception byte (AEXC) contains five exception bits (see Figure 2-8) required by the IEEE standard for trap disabled operation. These exceptions are logical combinations of the bits in the EXC byte. The AEXC byte contains the history of all floating-point exceptions that have occurred since the user last cleared the AEXC byte. In normal operations, only the user clears this byte by writing to the FPSR. The AEXC is cleared by a reset or a restore operation of the null state.

Many users elect to disable traps for all or part of the floating-point exception classes. The AEXC byte is provided to make it unnecessary to poll the EXC byte after each floating-point instruction. At the end of most operations (FMOVEM and FMOVE excluded), the bits in the EXC byte are logically combined to form an AEXC value that is logically ORed into the existing AEXC byte. This operation creates "sticky" floating-point exception bits in the AEXC byte that the user need poll only once (for example, at the end of a series of floating-point operations).

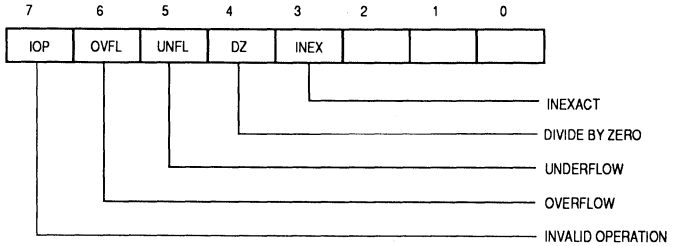


Figure 2-8. FPSR Accured Exception Byte

The setting or clearing of bits in the AEXC byte does not cause an exception nor does it prevent taking an exception. The relationship between the bits in the EXC byte and the bits in the AEXC is shown by the following equations. These equations apply to setting the AEXC bits at the end of each operation that affects the AEXC byte:

AEXC(IOP) = AEXC(IOP)vEXC(SNANvOPERR)
AEXC(OVFL) = AEXC(OVFL)vEXC(OVFL)
AEXC(UNFL) = AEXC(UNFL)vEXC(UNFLLINEX2
AEXC(DZ) = AEXC(DZ)vEXC(DZ
AEXC(INEX) = AEXC(INEX)vEXC(INEX1vINEX2vOVFL)

Where: "v" = Logical OR
"L" = Logical AND

2.2.1.8 FLOATING-POINT INSTRUCTION ADDRESS REGISTER (FPIAR). The floating-point instructions operate concurrently with the integer unit. That is, the integer unit can be executing instructions while the floating-point unit (FPU) is simultaneously executing a floating-point instruction. Additionally, the FPU can concurrently execute two floating-point instructions. As a result of this nonsequential instruction execution, the PC value stacked by the MC68040, in response to a floating-point exception trap, may not point to the offending instruction.

For the subset of the FPU instructions that generate exception traps, the 32-bit FPIAR is loaded with the logical address of the instruction before the instruction is executed. This address can then be used by a floating-point exception handler to locate a floating-point instruction that has caused an exception. Since the FPU FMOVE to/from the FPCR, FPSR, or FPIAR and

FMOVEM instructions cannot generate floating-point exceptions, these instructions do not modify the FPIAR. These instructions can be used to read the FPIAR in the trap handler without changing the previous value. The FPIAR is cleared by a reset or a null-restore operation.

2.2.2 Supervisor Programming Model

The supervisor programming model (see Figure 2-9) is used exclusively by system programmers to implement sensitive operating system functions, I/O control, and MMU subsystems. All the accesses that affect the control features of the MC68040 are in the supervisor programming model. Thus, all application software is written to run in the user mode and migrates to the MC68040 from any M68000 platform without modification.

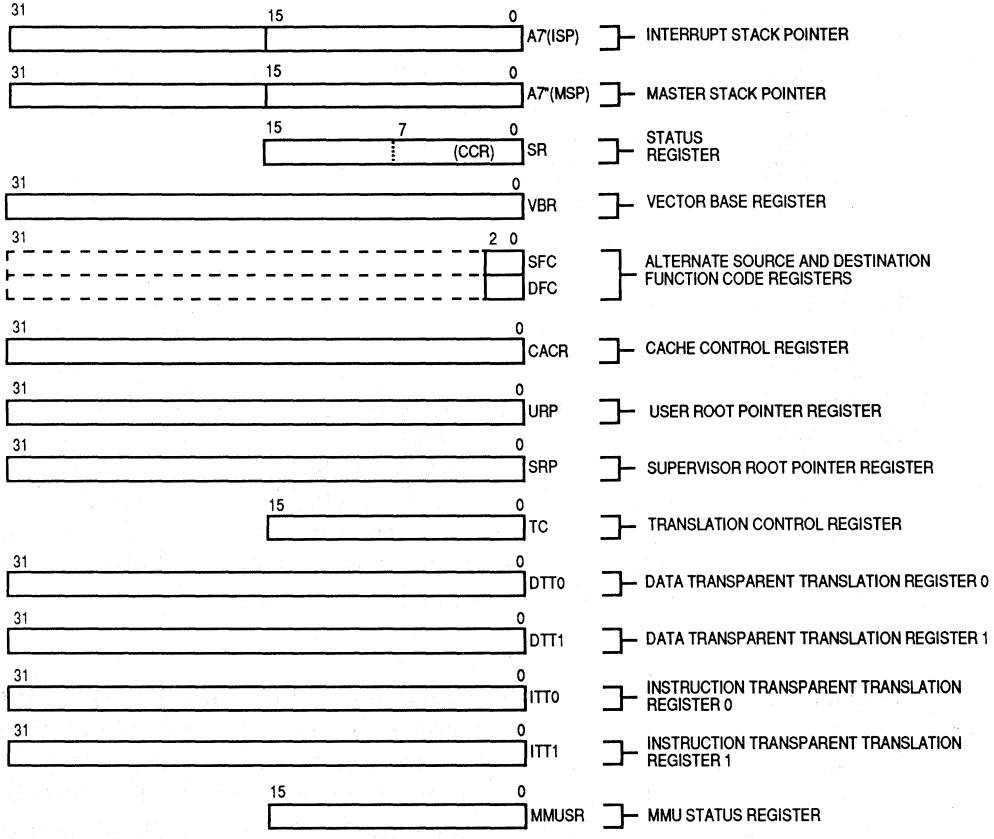


Figure 2-9. Supervisor Programming Model

The supervisor programming model consists of the registers available to the user as well as the following control registers:

- Two, 32-Bit Supervisor Stack Pointers Interrupt Stack Pointer (ISP) and Master Stack Pointer (MSP)
- 16-Bit Status Register (SR)
- 32-Bit Vector Base Register (VBR)
- Two, 32-Bit Alternate Function Code Registers Source Function Code (SFC) and Destination Function Code (DFC)
- 32-Bit Cache Control Register (CACR)
- 32-Bit User Root Pointer (URP)
- 32-Bit Supervisor Root Pointer (SRP)
- 16-Bit Translation Control Register (TC)
- Two, 32-Bit Data Transparent Translation Registers (DTT0 and DTT1)
- Two, 32-Bit Instruction Transparent Translation Registers (ITT0 and ITT1)
- 16-Bit MMU Status Register (MMUSR)

The following paragraphs describe the supervisor programming model registers. Additional information on the ISP, MSP, SR and VBR registers can be found in **SECTION 9 EXCEPTION PROCESSING**. Refer to **SECTION 7 INSTRUCTION AND DATA CACHES** for information on the CACR and to **SECTION 6 MEMORY MANAGEMENT** for information on the URP, SRP, TC, DTTn, ITTn, and MMUSR registers.

2.2.2.1 INTERRUPT AND MASTER STACK POINTERS (A7′ and A7″). The interrupt and master stack pointers are general-purpose address registers for the supervisor mode that may be used as software stack pointers, index registers, or base address registers. The interrupt and master stack pointers may be used for word and long-word operations.

Register A7 refers to three different registers; the USP (A7) in the user programming model and the ISP and MSP (A7′ and A7″) in the supervisor programming model. In the supervisor programming model, the active stack pointer (ISP or MSP) is called the supervisor stack pointer.

2.2.2.2 STATUS REGISTER (SR). The SR (see Figure 2-10), which stores the processor status, contains the condition codes that reflect the results of a previous operation and codes 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 SR information available in the user mode; it is referenced as the CCR in user programs. In the supervisor mode, software

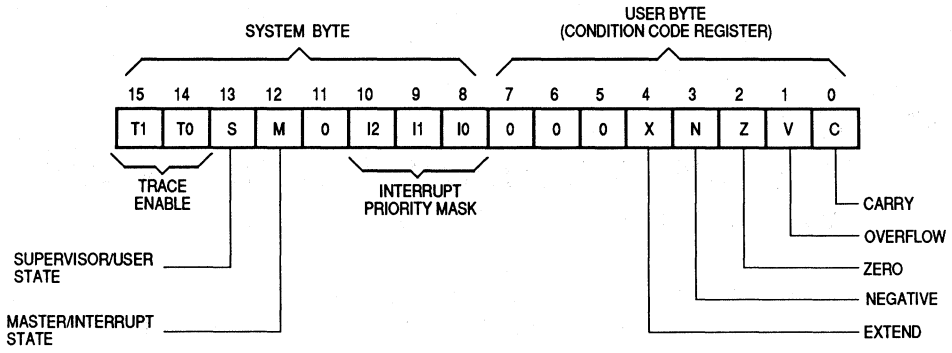


Figure 2-10. Status Register

can access the full SR, including the interrupt priority mask as well as additional control bits. These bits indicate the following states for the processor: one of two trace modes (T1, T0), supervisor or user mode (S), and master or interrupt mode (M).

2.2.2.3 VECTOR BASE REGISTER (VBR). The 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.

2.2.2.4 ALTERNATE FUNCTION CODE REGISTERS (SFC and DFC). The alternate function code registers contain 3-bit function codes. Function codes can be considered extensions of the 32-bit logical address that optionally provides as many as eight, 4 Gbyte address spaces. Function codes are automatically generated by the processor to select address spaces for data and programs at the user and supervisor modes. SFC and DFC registers are used by certain instructions to explicitly specify the function codes for operations.

SECTION 3
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 program space, the section of memory that contains the program instructions and any immediate data operands residing in the instruction stream. 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; however, these accesses are classified as program references. The MC68040 automatically accesses the program space or data space as required.

This section describes the data organization and addressing capabilities of the MC68040. It lists the type 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. Finally, the section describes the system stack and user stacks and queues.

3.1 INTEGER UNIT OPERAND DATA FORMATS

The MC68040, with its integer unit and floating-point unit (FPU), supports the operand data types shown in Table 3-1. The operand types supported by the integer unit include the data types supported by the MC68030 plus a new data type (16-byte block) for the MOVE16 instruction. Integer unit operands can reside in registers, in memory, or within the instructions themselves, and may be a single bit, a bit field, a byte, a word, a long word, a quad word, or a 16-byte block. The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation.

Table 3-1. Data Types

Operand Data Type	Size	Supported by:	Notes
Bit	1 Bit	Integer Unit	—
Bit Field	1-32 Bits	Integer Unit	Field of Consecutive Bits
BCD	8 Bits	Integer Unit	Packed: 2 Digits/Byte Unpacked: 1 Digit/Byte
Byte Integer	8 Bits	Integer Unit, FPU	—
Word Integer	16 Bits	Integer Unit, FPU	—
Long-Word Integer	32 Bits	Integer Unit, FPU	—
Quad-Word Integer	64 Bits	Integer Unit	Any Two Data Registers
16 Byte	128 Bits	Integer Unit	Memory-Only, Aligned to 16-Byte Boundary
Single-Precision Real	32 Bits	FPU	1-Bit Sign, 8-Bit Exponent, 23-Bit Mantissa
Double-Precision Real	64 Bits	FPU	1-Bit Sign, 11-Bit Exponent, 52-Bit Mantissa
Extended-Precision Real	80 Bits	FPU	1-Bit Sign, 15-Bit Exponent, 64-Bit Mantissa

3.2 FLOATING-POINT UNIT OPERAND DATA FORMATS

The following paragraphs describe the FPU unit operand data formats. Six data formats are supported: three signed binary integer formats and three binary floating-point formats. All data formats are supported uniformly by all arithmetic instructions. These formats are as follows:

Byte Integer (B)
Single Precision Real (S)
Word Integer (W)
Double Precision Real (D)
Long-Word Integer (L)
Extended Precision Real (X)

The capital letter in parenthesis is the suffix added to a floating-point instruction in the assembly language syntax to specify the data format of operands external to the MC68040.

A seventh data format, packed decimal real (P), is not directly supported in hardware, but is implicitly supported by trapping as an unimplemented data type (instead of as an illegal instruction) to allow efficient emulation in software. Refer to **SECTION 9 EXCEPTION PROCESSING** for detailed information.

Within the floating-point data formats, there are five types of numbers that can be represented: normalized numbers, denormalized numbers, zeros, infinities, and not-a-numbers (NANs). These data types are represented with special encodings corresponding to each data format.

3.2.1 Integer Data Formats

The three signed (twos complement) integer data formats supported by the FPU (byte, word, and long word) are identical to those supported by the integer unit (see Figure 3-1).

Since all FPU operations are performed in full extended-precision format, signed integer operands are converted to extended precision before the specified operation is performed. Thus, mixed-mode arithmetic is implicitly supported.

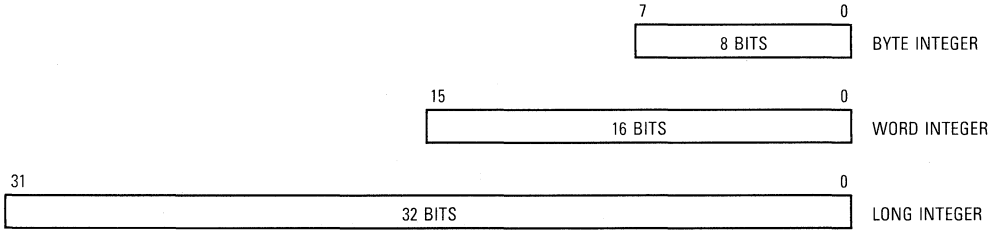


Figure 3-1. Signed Integer Data Formats

3.2.2 Binary Real-Data Formats

Floating-point numbers can be encoded in any of three data formats: single precision (32 bits), double precision (64 bits), and double-extended precision (96 bits, 80 of which are used). All three formats fully comply with the *IEEE Standard for Binary Floating-Point Arithmetic*.

NOTE

The single-extended-precision format defined in the IEEE standard is redundant in a device that supports the double-extended-precision format. Thus, all references in this manual to extended precision imply double-extended precision as defined by the IEEE standard.

Since all floating-point internal operations are performed in extended precision, single- and double-precision operands are converted to extended-precision values before the specified operation is performed. Thus, mixed-mode arithmetic is implicitly supported. Memory formats for the real-data formats are shown in Figure 3-2.

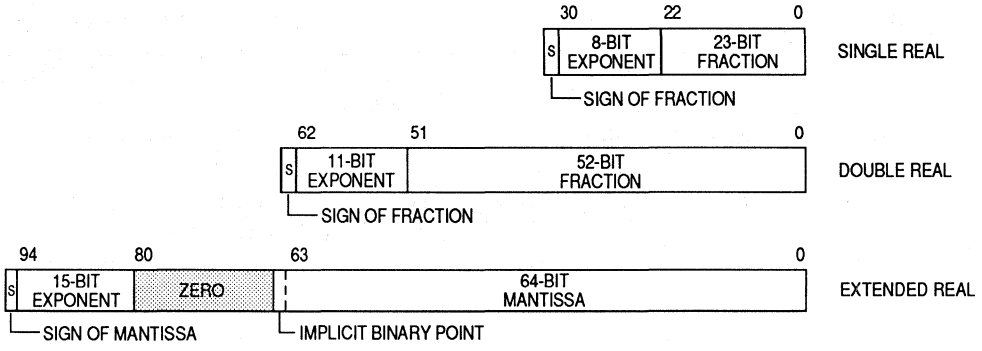


Figure 3-2. Binary Real-Data Formats

The exponent in all three binary formats is an unsigned binary integer with an implied bias added to it. The bias values for single, double, and extended precision are 127, 1023, and 16383, respectively. When the bias is subtracted from the value of the exponent, the result represents a signed twos-complement power of two that yields the magnitude of a normalized floating-point number when multiplied by the mantissa. Since biased exponents are used, a program can execute an integer-compare instruction (CMP) to compare floating-point numbers in memory (regardless of the absolute magnitude of the exponents).

Data formats for single- and double-precision numbers differ slightly from the data formats for extended-precision numbers in the representation of the mantissa. For all three precisions, normalized mantissa is always in the range [1.0 . . . 2.0]. The extended-precision data format explicitly represents the entire mantissa, including the explicit integer part bit. However, for single- and double-precision data formats, only the fractional portion of the mantissa is explicitly represented; the integer part, always one, is implied.

The IEEE standard has created the term "significand" to bridge this difference and to avoid the historical implications of the term mantissa. The IEEE standard defines a significand as the component of a binary floating-point number

that consists of an explicit or implicit leading bit to the left of the implied binary point. This manual interchangeably uses the terms mantissa and significand, defined as follows:

Single-Precision Mantissa	= Single-Precision Significand
	= 1.<23-Bit Fraction Field>
Double-Precision Mantissa	= Double-Precision Significand
	= 1.<52-Bit Fraction Field>
Extended-Precision Mantissa	= Extended-Precision Significand
	= 1.Fraction
	= <64-Bit Mantissa Field>

NOTE

Throughout this manual, ranges are specified using traditional set notation with the format "bound . . . bound" specifying the boundaries of the range. The type brackets enclosing the range defines whether the endpoint is inclusive or exclusive. A square bracket indicates inclusive, and a parenthesis indicates exclusive. For example, the range specification "[1.0 . . . 2.0]" defines the range of numbers greater than or equal to 1.0 and less than or equal to 2.0. The range specification "[0.0 +inf]" defines the range of numbers greater than 0.0 and less than or equal to positive infinity.

Each of the three floating-point data formats can represent five, unique, floating-point data types:

Normalized Numbers
Denormalized Numbers
Zeros
Infinities
Not-a-numbers (NANs)

The normalized data type never uses the maximum or minimum exponent value for a given format (except for the extended-precision format see following note). These exponent values in each precision are reserved for representing the special data types: zeros, infinities, denormalized numbers, and NANs. Details of each type number for each format are shown in **3.2.3 Floating-Point Data Format Details**.

NOTE

There is a subtle difference between the definition of an extended-precision number with an exponent equal to zero and a single- or double-precision number with an exponent equal to zero. If the exponent of a single- or double-precision number is zero, the number is defined to be denormalized, and the implied integer bit is also zero. However, an extended-precision number with an exponent of zero may have an explicit integer bit equal to one, which results in a normalized number (even though the exponent is equal to the minimum value).

For simplicity, the following discussion treats all three real formats in the same manner, where an exponent value of zero identifies a denormalized number. It should be noted that the extended precision format may deviate from this rule.

3.2.2.1 NORMALIZED NUMBERS. Normalized numbers encompass all representable real values between the overflow and underflow thresholds: i.e., those numbers whose exponents lie between the maximum and minimum values. Normalized numbers may be positive or negative. For normalized numbers, the implied integer part bit in single and double precision is one. In extended precision, the integer bit is explicitly a one (see Figure 3-3).

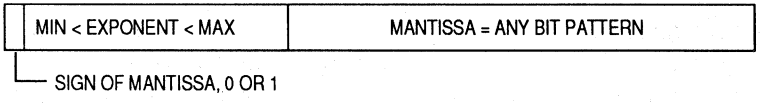


Figure 3-3. Format of Normalized Numbers

3.2.2.2 DENORMALIZED NUMBERS. Denormalized numbers represent real values near the underflow threshold (underflow is detected for a given data format and operation when the result exponent is less than or equal to the minimum exponent value). Denormalized numbers may be positive or negative. For denormalized numbers, the implied integer part bit in single and double precision is a zero (0). In extended precision, the integer bit is explicitly a zero (0), (see Figure 3-4).

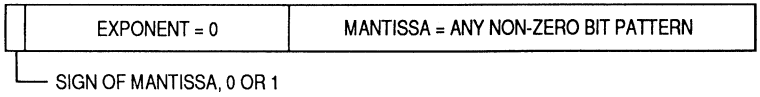


Figure 3-4. Format of Denormalized Numbers

Traditionally, floating-point number systems perform a "flush-to-zero" when underflow is detected. This leaves a large gap in the number line between the smallest magnitude normalized number and zero. The IEEE standard implements gradual underflows: the result mantissa is shifted right (denormalized) while the result exponent is incremented until the result exponent reaches the minimum value. If all the mantissa bits of the result are shifted off to the right during this denormalization, the result becomes zero. In many cases, gradual underflow limits the potential underflow damage to no more than a round-off error. (This underflow and denormalization description ignores the effects of rounding and the user-selectable rounding modes). Thus, the large gap in the number line created by "flush-to-zero" number systems is filled with representable (denormalized) numbers in the IEEE "gradual underflow" floating-point number system.

Since the extended-precision data format has an explicit integer part bit, a number can be formatted with a nonzero exponent (less than the maximum value) and a zero integer bit, which is not defined by the IEEE standard. Such a number is called an unnormalized number.

Denormalized and unnormalized numbers are not directly supported in hardware, but are implicitly supported by trapping as an unimplemented data type to allow efficient conversion in software. Refer to **SECTION 9 EXCEPTION PROCESSING** for more details.

3.2.2.3 ZEROS. Zeros are signed (positive or negative) and represent the real values +0.0 and −0.0 (see Figure 3-5).

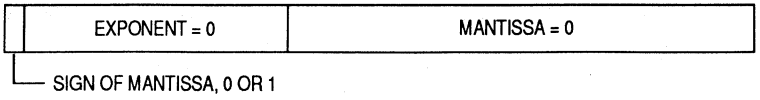
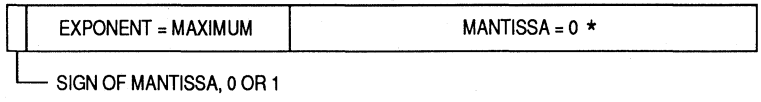


Figure 3-5. Format of Zero

3.2.2.4 INFINITIES. Infinities are signed (positive or negative) and represent real values that exceed the overflow threshold. Overflow is detected for a given data format and operation when the result exponent is greater than or equal to the maximum exponent value. (This overflow description ignores the effects of rounding and the user-selectable rounding models.) For extended-precision infinities, the MSB of the mantissa (the integer bit) can be either one or zero (see Figure 3-6).



* For the extended-precision format, the most significant bit of the mantissa (the integer bit) is a don't care.

Figure 3-6. Format of Infinity

3.2.2.5 NOT-A-NUMBERS. When created by the FPU, NANs represent the results of operations having no mathematical interpretation, such as infinity divided by infinity. All operations involving a NAN operand as an input return a NAN result. When created by the user, NANs can protect against unitialized variables and arrays or represent user-defined special number types. For extended-precision NANs, the MSB of the mantissa (the integer bit) can be either one or zero (see Figure 3-7).

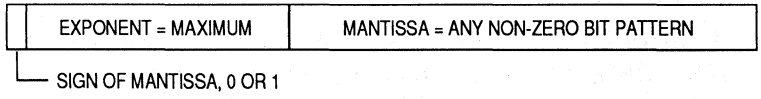


Figure 3-7. Format of NANs

Two different types of NANs are implemented by the FPU. The value of the MSB of the fraction identifies the type. The identifying bit is the MSB of the mantissa for single and double precision, and the MSB of the mantissa minus one for extended precision. NANs with a leading fraction bit equal to one are nonsignaling NANs; NANs with a leading fraction bit equal to zero are signaling NANs (SNANs). A SNAN can be used as an escape mechanism for a user-defined, non-IEEE data type. The FPU never creates a SNAN as a result of an operation.

The IEEE specification defines the manner in which a NAN is processed when used as an input to an operation. Particularly, if a SNAN is used as an input and the SNAN trap is not enabled, a nonsignaling NAN must be returned as a result. The FPU accomplishes this by using the source SNAN, setting the MSB of the fraction, and storing the resultant nonsignaling NAN in the destination. Because of the IEEE formats for NANs, the result of setting the most significant fraction bit of a SNAN is always a nonsignaling NAN.

When NANs are created by the FPU, they always contain the same bit pattern in the mantissa; for any precision, all bits of the mantissa are ones. When a NAN is created by the user, any nonzero bit pattern can be stored in the mantissa.

3.2.3 Floating-Point Data Format Details

Tables 3-2 through 3-4 provide the format specification details for the single (S), double (D), and extended (X) precision binary real data formats.

3.3 ORGANIZATION OF DATA IN REGISTERS

The following paragraphs provide a description of data organization within the data, address, and control registers.

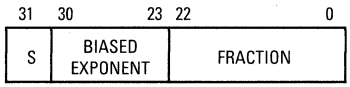
3.3.1 Integer Data Registers

Each integer data register is 32 bits wide. Byte operands occupy the lower order 8 bits, word operands the lower 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 LSB of a long-word integer is addressed as bit zero and the MSB is addressed as bit 31. For bit fields, MSB is addressed as bit zero, and the LSB 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. Figure 3-8 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 integer data registers without

Table 3-2. Single-Precision Binary Real-Data Format

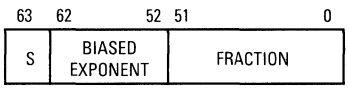
Memory Format:



Field Size (in Bits):	
s = Sign	1
e = Biased Exponent	8
f = Fraction	23
Total	32
Interpretation of Sign:	
Positive Mantissa, s =	0
Negative Mantissa, s =	1
Normalized Numbers:	
Bias of e	+127 ($7F)
Range of e	0 < e < 255 ($FF)
Range of f	Zero or Nonzero
Mantissa = Significand =	1.f
Relation to Representation of Real Numbers	$(-1)^s \times 2^{e-127} \times 1.f$
Denormalized Numbers:	
e = Format Minimum =	0 ($00)
Bias of e	+126 ($7E)
Range of f	Nonzero
Mantissa = Significand =	0.f
Relation to Representation of Real Numbers	$(-1)^s \times 2^{-126} \times 0.f$
Signed Zeros:	
e = Format Minimum =	0 ($00)
f = Mantissa = Significand =	0.f = 0.0
Signed Infinities:	
e = Format Maximum =	255 ($FF)
f = Mantissa = Significand =	0.f = 0.0
NANs (Not-A-Number):	
s =	Don't Care
e = Format Maximum =	255 ($FF)
f =	Non-Zero
Representation of f	0.1xxxx . . . xxxx, Nonsignaling
	0.0xxxx . . . xxxx, Signaling
xxxx . . . xxxx	Nonzero Bit Pattern
f When Created by the FPCP	.11111 . . . 1111
Ranges (Approximate):	
Maximum Positive Normalized	3.4×10^{38}
Minimum Positive Normalized	$1.2 \times 10^{s-38}$
Minimum Positive Denormalized	1.4×10^{-45}

Table 3-3. Double-Precision Binary Real-Data Format

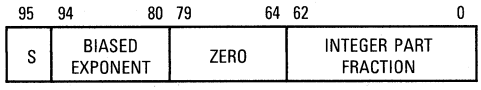
Memory Format:



Field Size (in Bits):	
s = Sign	1
e = Biased Exponent	11
f = Fraction	52
Total	64
Interpretation of Sign:	
Positive Mantissa, s =	0
Negative Mantissa, s =	1
Normalized Numbers:	
Bias of e	+1023
Range of e	$0 < e < 2047$ ($7FF)
Range of f	Zero or Nonzero
Mantissa = Significand =	1.f
Relation to Representation of Real Numbers	$(-1)^s \times 2^{e-1023} \times 1.f$
Denormalized Numbers:	0 ($000)
e = Format Minimum =	
Bias of e	+1022 ($3FE)
Range of f	Nonzero
Mantissa = Significand =	0.f
Relation to Representation of Real Numbers	$(-1)^s \times 2^{-1022} \times 0.f$
Signed Zeros:	
e = Format Minimum =	0 ($00)
f = Mantissa = Significand =	0.f = 0.0
Signed Infinities:	
e = Format Maximum =	2047 ($7FF)
f = Mantissa = Significand =	0.f = 0.0
NANs (Not-A-Number):	
s =	Don't Care
e = Format Maximum =	2047 ($7FF)
f =	Nonzero
Representation of f	0.1xxxx . . . xxxx, Nonsignaling
	0.0xxxx . . . xxxx, Signaling
xxxx . . . xxxx	Nonzero Bit Pattern
f When Created by the FPCP	.11111 . . . 1111
Ranges (Approximate):	
Maximum Positive Normalized	18×10^{307}
Minimum Positive Normalized	2.2×10^{-308}
Minimum Positive Denormalized	4.9×10^{-324}

Table 3-4. Extended-Precision Binary Real-Data Format

Memory Format:



Field Size (in Bits):	
s = Sign	1
e = Biased Exponent	15
u = Zero, Reserved	16
j = Integer Part	1
f = Fraction	63
Total	96
Interpretation of Unused Bits:	
Input	Don't Care
Output	All Zeros
Interpretation of Sign:	
Positive Mantissa, s =	0
Negative Mantissa, s =	1
Normalized Numbers:	
Bias of e	+16383 ($3FFF)
Range of e	0 <= e < 32767 ($7FFF)
j =	1
Range of f	Zero or Nonzero
j.f = Mantissa = Significand =	1.f
Relation to Representation of Real Numbers	$(-1)^s \times 2^{e-16383} \times j.f$
Denormalized Numbers:	
e = Format Minimum	0 ($0000)
Bias of e	+16383 ($3FFF)
j =	0
Range of f	Nonzero
j.f = Mantissa = Significand =	0.f
Relation to Representation of Real Numbers	$(-1)^s \times 2^{-16383} \times 0.f$
Signed Zeros:	
e = Format Minimum =	0 ($0000)
j.f = Mantissa = Significand =	0.0
Signed Infinities:	
e = Format Maximum =	32767 ($7FFF)
j =	Don't Care
j.f = Mantissa = Significand	j.000 . . . 0000
NANs (Not-A-Numbers):	
s =	Don't Care
j =	Don't Care
e = Format Maximum =	32767 ($7FFF)
f =	Nonzero
Representation of f	j.1xxx . . . xxxx, Nonsignaling j.0xxx . . . xxxx, Signaling
xxx . . . xxxx	Nonzero Bit Pattern
f When Created by the FPCP	1.11111 . . . 1111
Ranges (Approximate):	
Maximum Positive Normalized	6×10^{4931}
Minimum Positive Normalized	8×10^{-4933}
Minimum Positive Denormalized	9×10^{4952}

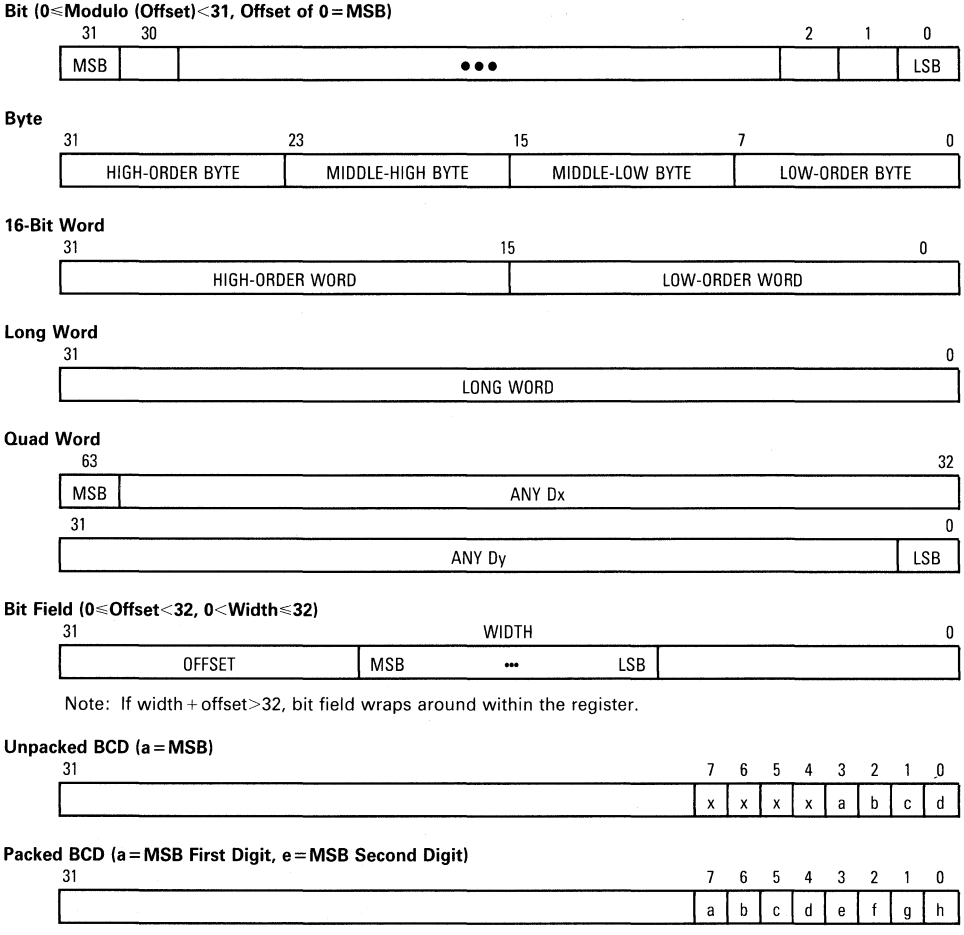


Figure 3-8. Data Organization in Integer Data Registers

restrictions on order or pairing. There are no explicit instructions for the management 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 in which the LSBs 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; four LSBs contain the binary value and the four MSBs are undefined. Each byte of the packed BCD format contains two digits; the least significant four bits contain the least significant digit.

3.3.2 Floating-Point Data Registers

The eight, 80-bit floating-point data registers (FP7–FP0) are analogous to the integer data registers (D7–D0) and are completely general purpose (i.e., any instruction may use any register). The allowable data formats for the floating-point data registers are explained in detail in the following paragraphs.

The FPU supports several data formats and data types with on-chip hardware. Other data formats, such as packed-decimal real-data format, are supported by software emulation (see Table 3-5).

Table 3-5. FPU Data Formats and Data Types

	Data Formats						
Data Types	**Single-Precision Real**	**Double-Precision Real**	**Extended-Precision Real**	**Packed-Decimal Real**	**Byte Integer**	**Word Integer**	**Long-Word Integer**
Normal	*	*	*	«	*	*	*
Zero	*	*	*	«	*	*	*
Infinity	*	*	*	«			
NAN	*	*	*	«			
Denormalized	«	«	«	«			
Unnormalized			«	«			

NOTES:
* = Data Format/Type Supported by On-Chip FPU Hardware
« = Data Format/Type Supported by Software

3.3.2.1 INTERNAL DATA FORMAT. All floating-point internal operations are performed in extended precision. Regardless of data format, all external operands are converted to extended-precision values before the specified operation is performed.

The format of an intermediate result is shown in Figure 3-9. The intermediate-result exponent for some dyadic operations (multiply and divide) can easily overflow or underflow the 15-bit exponent of the designation FP register. In order to simplify the overflow and underflow detection, intermediate results in the FPU maintain a 17-bit, twos-complement integer exponent. When an

overflow or underflow intermediate result is detected, the intermediate 17-bit exponent is always converted into a 15-bit biased exponent before it is stored in a floating-point data register. Additionally, the mantissa is maintained internally as 67 bits for rounding purposes, but is always rounded to 64 bits (or less, depending on the selected rounding precision) before it is stored in a floating-point data register.

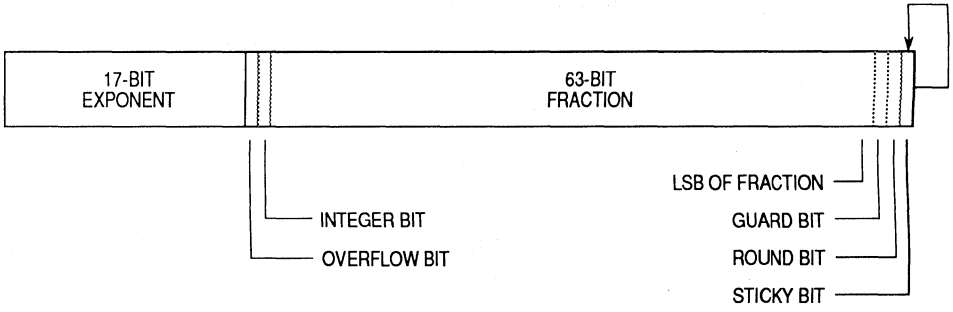


Figure 3-9. Intermediate-Result Format

3.3.2.2 FORMAT CONVERSIONS. Two cases of conversion between two data formats are as follows:

1) Converting an operand in any memory data format to the extended-precision data format and storing it in a floating-point data register or using it as the source operand for an arithmetic operation.
2) Converting the extended-precision value in a floating-point data register to any data format and storing it in a memory destination or integer register.

Since the internal data format used by the FPU is always extended precision, all external operands, regardless of data format, are converted to extended-precision values before the specified operation is performed. If the external operand, regardless of data format, is a denormalized number, the number is normalized before the operation is performed. Conversion and normalization apply not only to loading a floating-point data register but also to external operands involved in arithmetic operations.

Because floating-point data registers always contain extended-precision data format values, an external extended-precision denormalized number moved into a floating-point data register is stored as an extended-precision denormalized number. The number is first normalized and then denormalized before it is stored in the designated floating-point data register. This method simplifies the handling of all other data formats and types.

If an external operand is an extended-precision unnormalized number, the number is normalized before it is used in an arithmetic operation. If the external operand is an extended-precision unnormalized zero (i.e., with a mantissa of all zeros), the number is converted to an extended-precision normalized zero before the specified operation is performed. This normalization and conversion applies to loading a floating-point data register. The regular use of unnormalized inputs not only defeats the purpose of the IEEE standard, but also may produce gross inaccuracy in the results.

Conversion from the extended-precision data format to any of the other five floating-point data formats occurs when the contents of a floating-point data register are stored to memory or to an integer data register. Since no operation performed by the FPU can create an unnormalized result, the result of moving the contents of a floating-point data register to an extended-precision external destination can never be an unnormalized number.

3.3.3 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 address-computation support. The instruction set includes instructions that add to, compare, and move the contents of address registers. Figure 3-10 shows the organization of addresses in address registers.

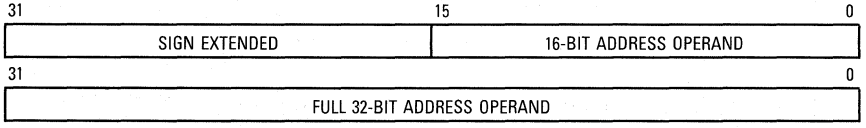


Figure 3-10. Address Organization in Address Registers

3.3.4 Control Registers

The control registers (refer to Figure 2-3) vary in size according to function. The lower byte of the status register (SR), floating-point control register (FPCR), floating-point status register (FPSR), and floating-point instruction address register (FPIAR) are accessible at the user privilege level. All other control registers may be accessed only at the supervisor privilege level.

NOTE

Some control registers have undefined bits reserved for future definition by Motorola. Those particular bits are read as zeros and must be written as zeros for future compatibility.

Although the SR is 16 bits wide, only 12 bits are defined. The undefined bits are reserved by Motorola for future definition. The lower byte of the SR is the condition code register (CCR). Operations to the CCR can be performed in the supervisor or user mode. All operations to the SR 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 32-bit FPCR contains an exception enable byte that enables/disables traps for each class of floating-point exceptions and a mode byte that sets the user-selectable modes. The FPCR can be read or written to by the user. Bits 31–16 are reserved for future definition by Motorola.

The 32-bit FPSR contains a condition code byte, an exception status byte, quotient bits, and an accrued exception byte. Execution of most floating-point instructions modifies this register.

For the subset of the FPU instructions that generate exception traps, the 32-bit FPIAR register is loaded with the logical address of an instruction before the instruction is executed (unless all arithmetic exceptions are disabled). This address can then be used by a floating-point exception handler to locate a floating-point instruction that has caused an exception.

The vector base register (VBR) provides the base address of the exception vector table. The cache control register (CACR) provides control and status information for the on-chip instruction and data caches.

The alternate function code registers (SFC and DFC) are 32-bit registers with only bits 0–2 implemented. These bits contain the address space values for the read or write operands of MOVES, PFLUSH, and PTEST instructions. The

MOVEC instruction is used to transfer values to and from the SFC and DFC. These are long-word transfers; the upper 29 bits are read as zeros and are ignored when written.

The remaining control registers are used by the MMU. The user root pointer (URP) and supervisor root pointer (SRP) contain pointers to the user and supervisor address translation trees. Transfers of data to and from these 32-bit registers are long-word transfers. The translation control (TC) register contains information for the MMU. The MC68040 always uses word transfers to access this 16-bit register. The 32-bit transparent translation registers (DTT0, DTT1, ITT0, ITT1) 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. Transfers to and from the MMUSR are word transfers. Refer to **SECTION 6 MEMORY MANAGEMENT UNIT** for more details.

3.4 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 MSB of the highest-order word. The lower-order word is located at address N+2, leaving the LSB at address N+3 (see Figure 3-11). The MC68040 does not require data to be aligned on word boundaries, 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.

All data formats are organized in memory consistent with the M68000 Family data organization, i.e., the MSB is located at the lowest address (nearest $00000000), with each successive LSB located at the next address (N+1, N+2, etc.). The LSB is located at the highest address (nearest $FFFFFFFF).

3.4.1 Integer Data Formats

The following integer data types are supported in memory by the MC68040: bit and bit-field data; signed and unsigned integer data of 8, 16, or 32 bits; 16-byte block; 32-bit addresses; and BCD (packed and unpacked). These data types are organized in memory as shown in Figure 3-12.

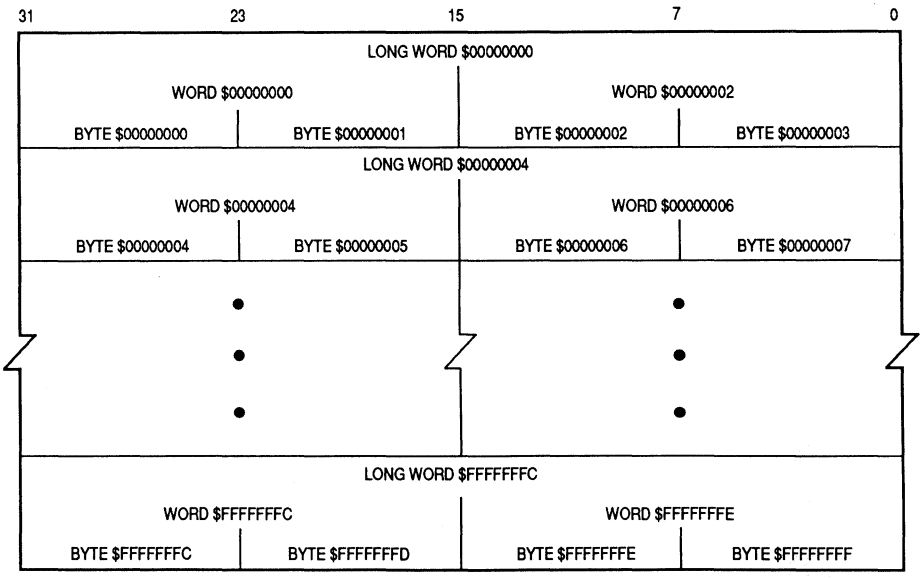


Figure 3-11. Memory Operand Addressing

A bit operand is specified by a base address that selects one byte in memory (the base byte) and a bit number that selects one bit in the base byte. The MSB of the byte is seven.

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 MSB 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 MSB of the base byte is bit-field offset 0, the LSB of the base byte is bit-field offset 7, and the LSB of the previous byte in memory is bit-field offset −1. Bit-field offsets may have values in the range of 2^{-31} to $2^{31}-1$, and bit-field widths may range from 1 to 32 bits.

A 16-byte block operand (supported by the MOVE16 instruction) consists of a block of 16 bytes, aligned to a 16-byte boundary. This operand is specified by an address that can point to any byte in the block.

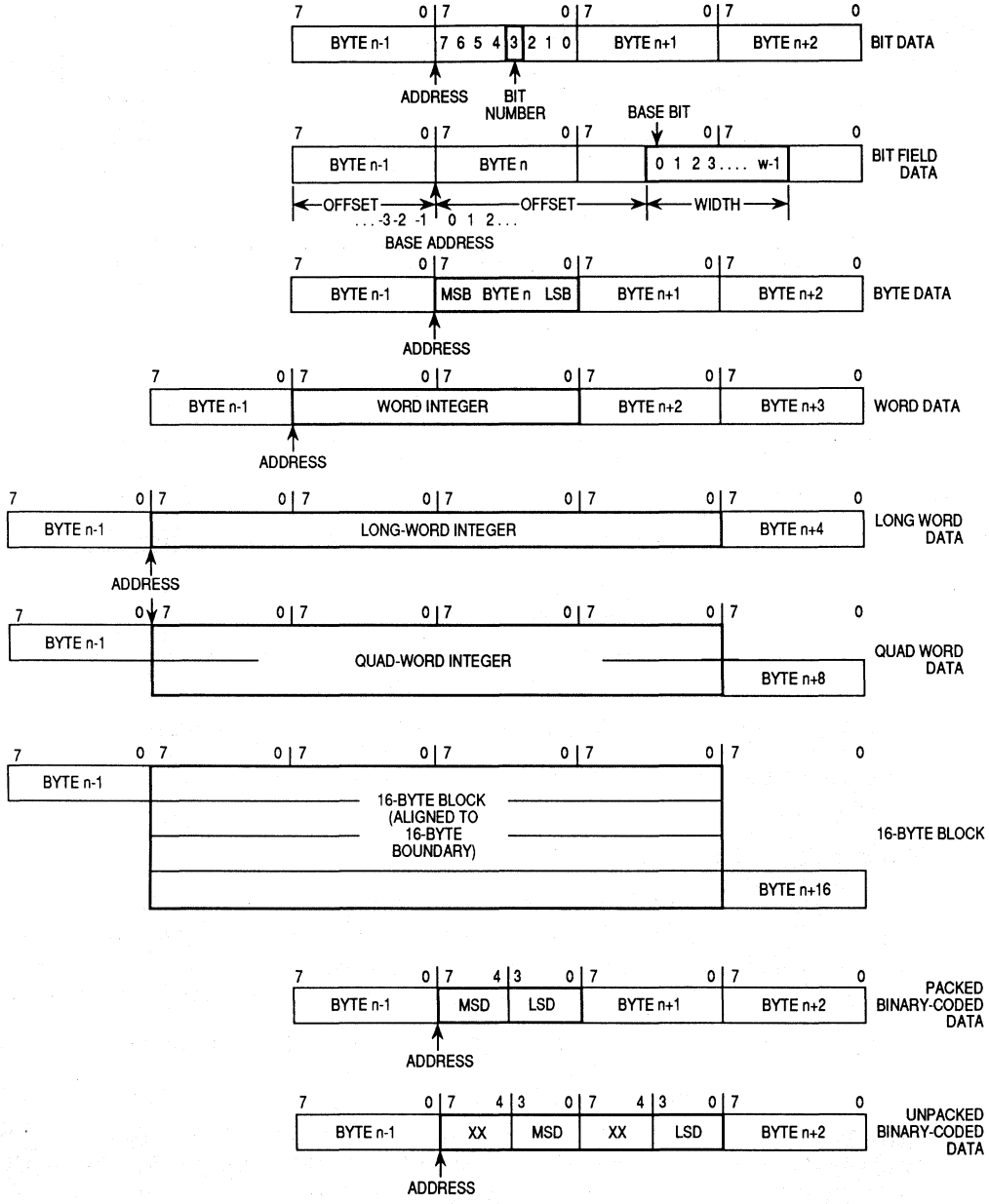


Figure 3-12. Memory Organization for Integer Operands

3.4.2 Floating-Point Data Formats

Figure 3-13 shows the floating-point data format for the single- (S), double- (D), and extended-precision (X) binary real-data floating-point organization in memory. Tables 3-2 through 3-4 provide the format specification details for these formats.

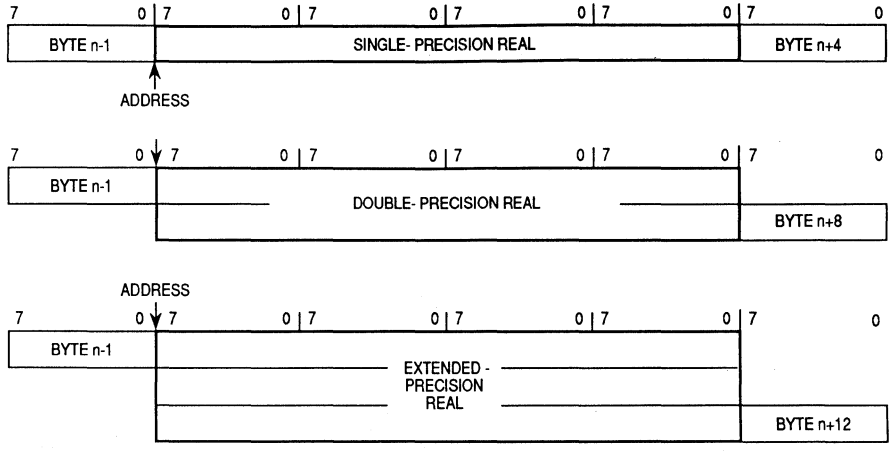


Figure 3-13. Memory Organization for Floating-Point Operands

3.5 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 3-14 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 ⟨ea⟩ 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 sub-mode for modes not using registers.

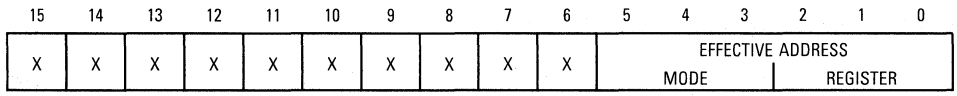


Figure 3-14. Single-Effective-Address Instruction Operation Word

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 **3.6 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:

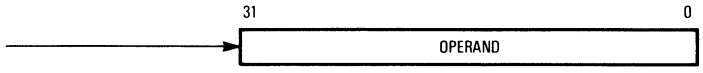
EA—Effective address
An—Address register n
Example: A3 is address register 3
Dn—Data register n
Example: D5 is data register 5
Xn.SIZE*SCALE—Denotes index register n (data or address), the index size (W for word, L for long word), and a scale factor (1, 2, 4, or 8, for no-word, word, long-word, or 8 for 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.

3.5.1 Data Register Direct Mode

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

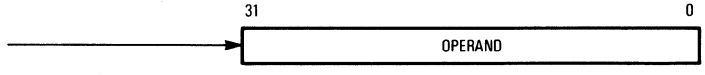
GENERATION:	EA = Dn
ASSEMBLER SYNTAX:	Dn
MODE:	000
REGISTER:	n
DATA REGISTER:	Dn
NUMBER OF EXTENSION WORDS:	0



3.5.2 Address Register Direct Mode

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

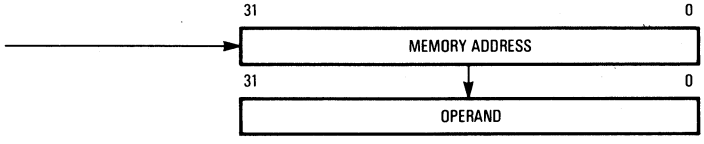
GENERATION:	EA = An
ASSEMBLER SYNTAX:	An
MODE:	001
REGISTER:	n
ADDRESS REGISTER:	An
NUMBER OF EXTENSION WORDS:	0



3.5.3 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.

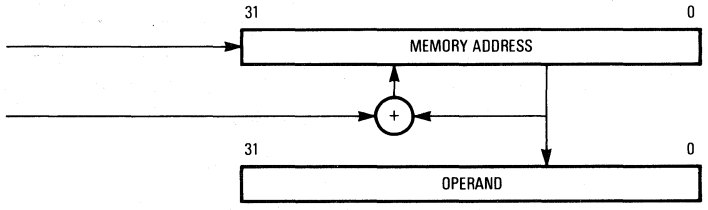
GENERATION:	EA = (An)
ASSEMBLER SYNTAX:	(An)
MODE:	010
REGISTER:	n
ADDRESS REGISTER:	An
MEMORY ADDRESS:	
NUMBER OF EXTENSION WORDS:	0



3.5.4 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.

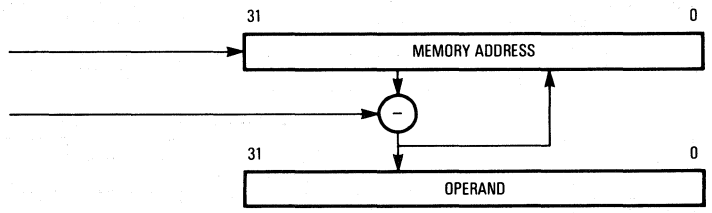
GENERATION: EA = (An)
An = An + SIZE
ASSEMBLER SYNTAX: (An) +
MODE: 011
REGISTER: n
ADDRESS REGISTER: An
OPERAND LENGTH (1, 2, OR 4):
MEMORY ADDRESS:
NUMBER OF EXTENSION WORDS: 0



3.5.5 Address Register Indirect with Predecrement Mode

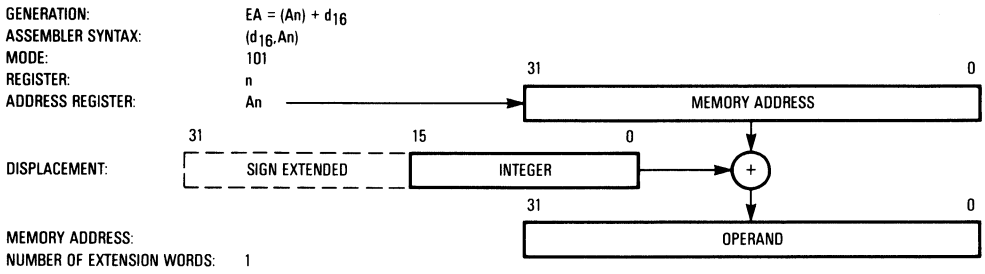
In the address register indirect with predecrement mode, the operand is in memory and the address of the operand is in the address register specified by the register field. Before the operand address is used, it is decremented by one, two, or four depending on the operand size: byte, word, or long word. 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.

GENERATION: An = An – SIZE
EA = (An)
ASSEMBLER SYNTAX: – (An)
MODE: 100
REGISTER: n
ADDRESS REGISTER: An
OPERAND LENGTH (1, 2, OR 4):
MEMORY ADDRESS:
NUMBER OF EXTENSION WORDS: 0



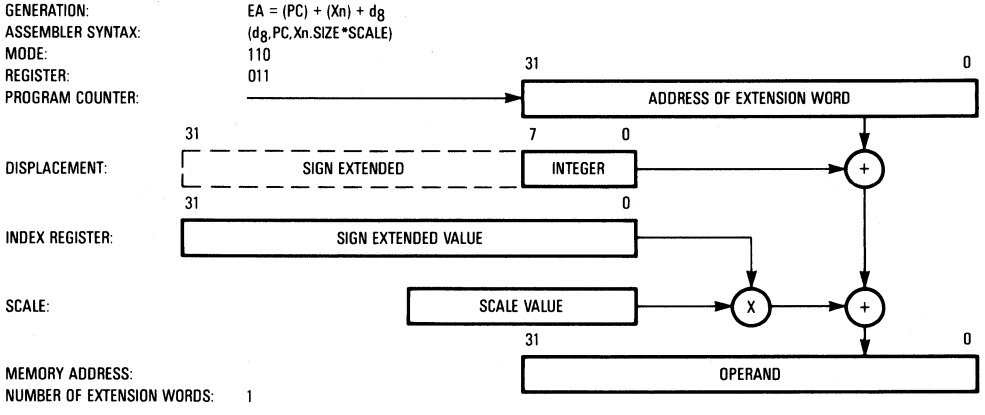
3.5.6 Address Register Indirect with Displacement Mode

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.



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

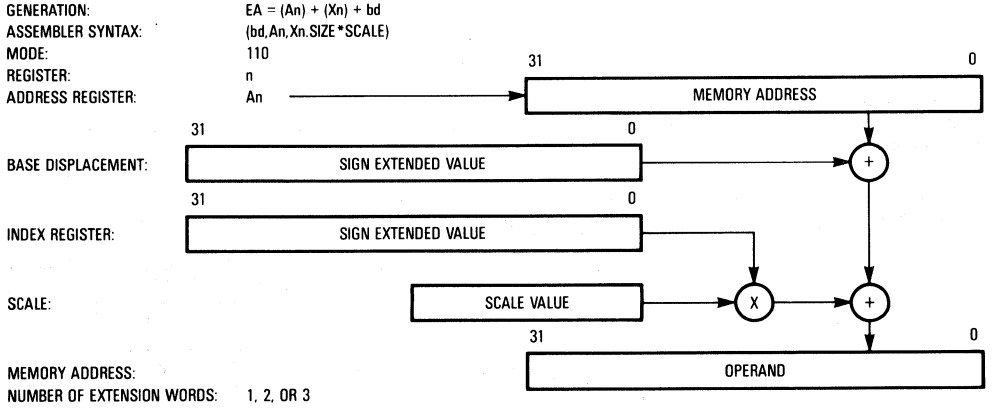
This addressing mode requires one extension word that contains the index register indicator and an 8-bit displacement. The index register indicator includes size and scale information. In this mode, the operand is in memory. The address of the operand is the sum of the contents of the address register, 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.



3.5.8 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).



3.5.9 Memory Indirect Postindexed Mode

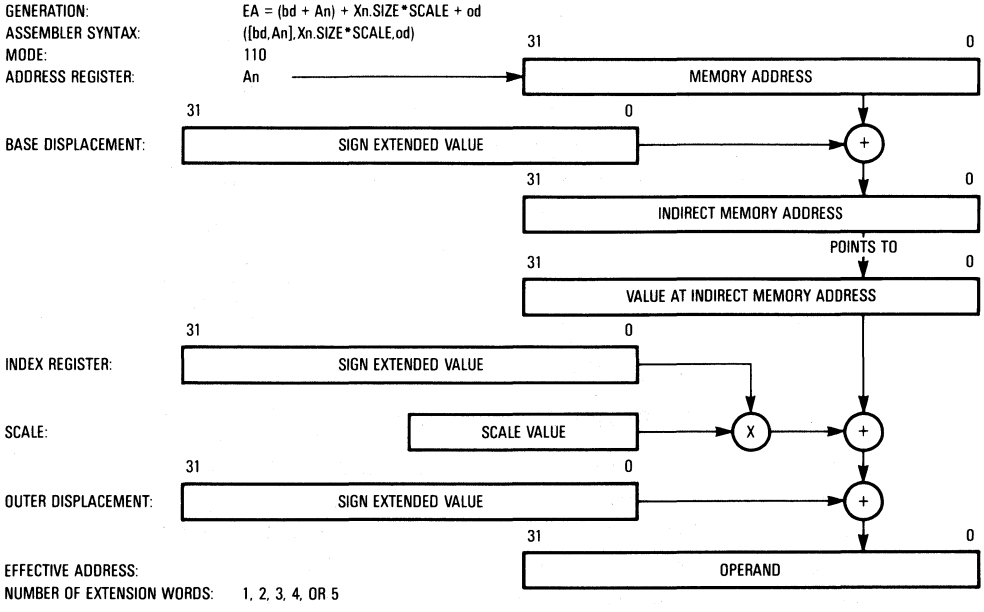
In this mode, the operand and its address are in memory. The processor calculates an intermediate indirect memory address using the base register (An) and base displacement (bd). The processor accesses a long word at this address and adds the index operand (Xn.SIZE*SCALE) and the outer displacement to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

3.5.10 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.

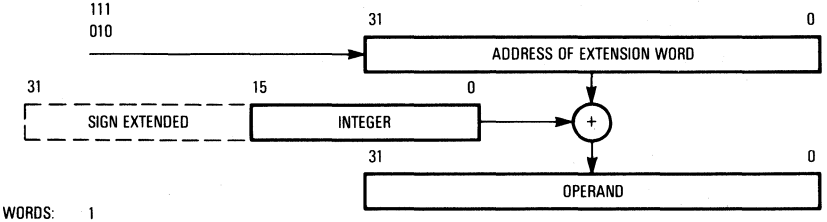
In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. Both the base and outer displacements may be null, word, or long word. When a displacement is omitted or an element is suppressed, its value is taken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



3.5.11 Program Counter Indirect with Displacement Mode

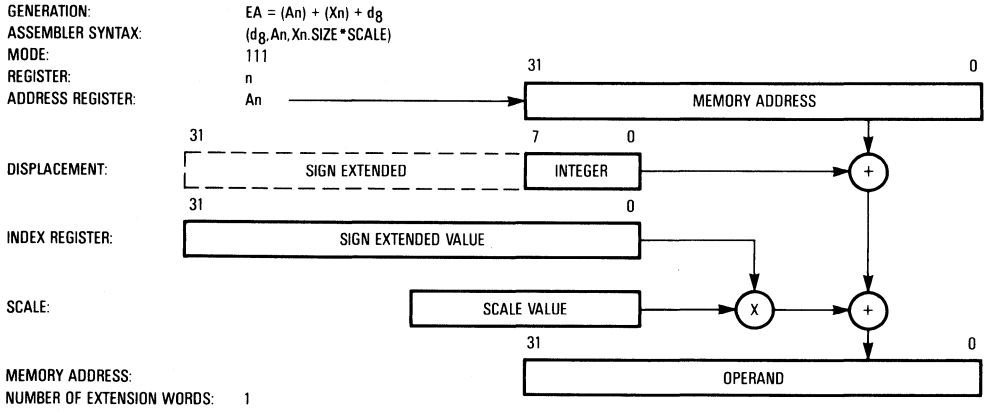
In this mode, the operand is in memory. The address of the operand is the sum of the address in the program counter and the sign-extended 16-bit displacement integer in the extension word. The value in the program counter is the address of the extension word. The reference is a program space reference and is only allowed for reads.

GENERATION: EA = (PC) + d_{16}
ASSEMBLER SYNTAX: (d_{16},PC)
MODE: 111
REGISTER: 010
PROGRAM COUNTER:
DISPLACEMENT:
MEMORY ADDRESS:
NUMBER OF EXTENSION WORDS: 1



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

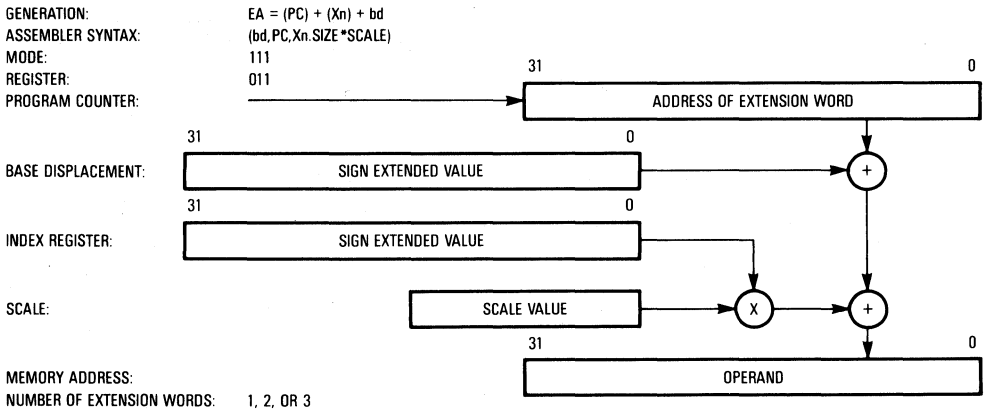
This mode is similar to the address register indirect with index (8-bit displacement) mode described in **3.5.7 Address Register Indirect with Index (8-Bit Displacement) Mode**, except the PC is used as the base register. The operand is in memory. The address of the operand is the sum of the address in the program counter, 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.



3.4.13 Program Counter Indirect with Index (Base Displacement) Mode

This mode is similar to the address register indirect with index (base displacement) mode described in **3.5.8 Address Register Indirect with Index (Base Displacement) Mode**, except 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.

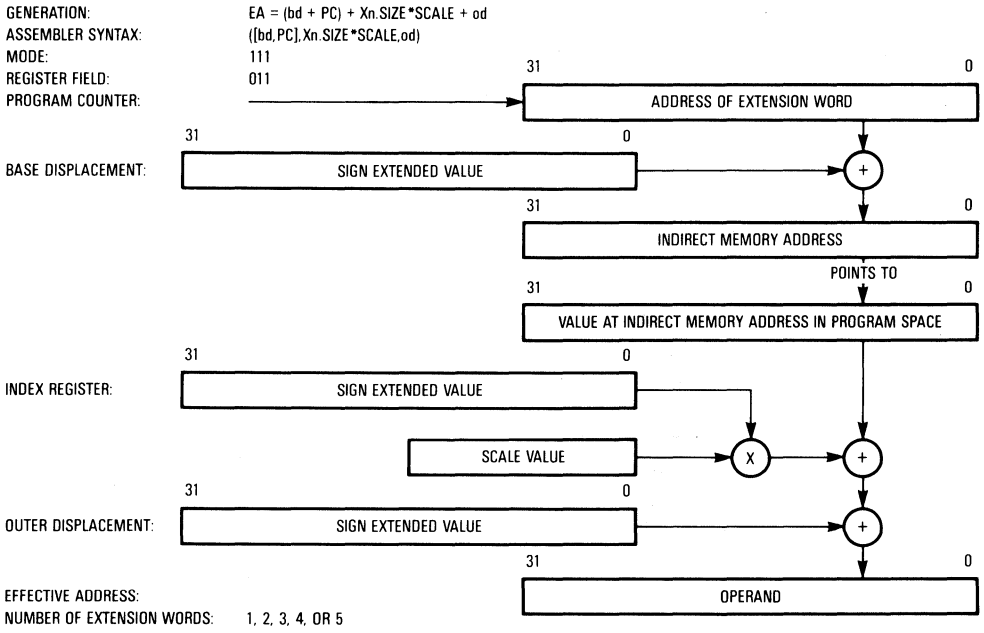
In this mode, the PC, the index register, and the displacement are all optional. However, the user must supply the assembler notation "ZPC" (zero value is taken for the PC) to indicate that the PC is not used. This allows the user to access the program space, without using the PC in calculating the effective 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.



3.5.14 Program Counter Memory Indirect Postindexed Mode

This mode is similar to the memory indirect postindexed mode described in **3.5.9 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.

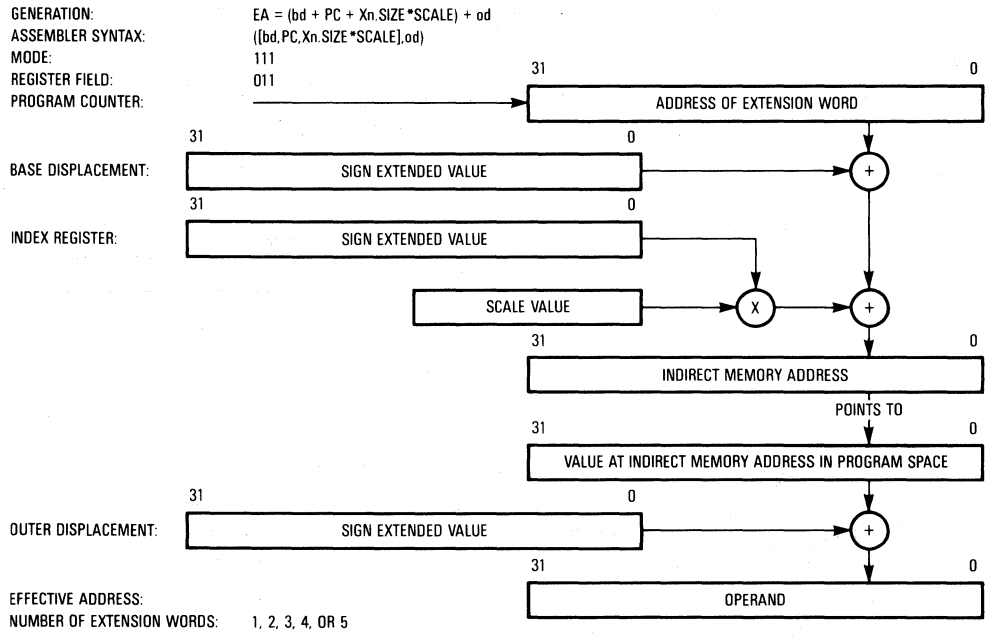
In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. However, the user must supply the assembler notation ZPC to indicate that the PC is not used. This allows the user to access the program space, without 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.



3.5.15 Program Counter Memory Indirect Preindexed Mode

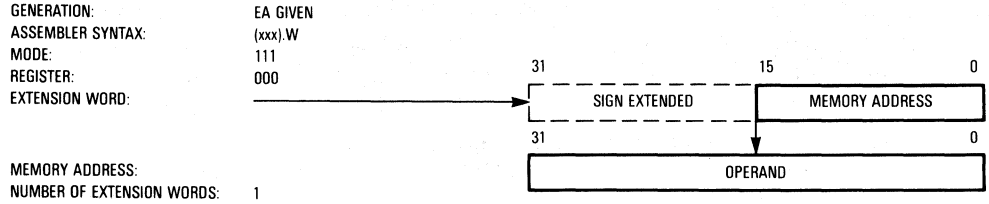
This mode is similar to the memory indirect preindexed mode described in **3.5.10 Memory Indirect Preindexed Mode**, but the PC is used as the base register. Both the operand and operand address are in memory. The processor calculates an intermediate indirect memory address by adding the PC contents, a base displacement, and the scaled contents of an index register. The processor accesses a long word at that address and adds the optional outer displacement 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.

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 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.



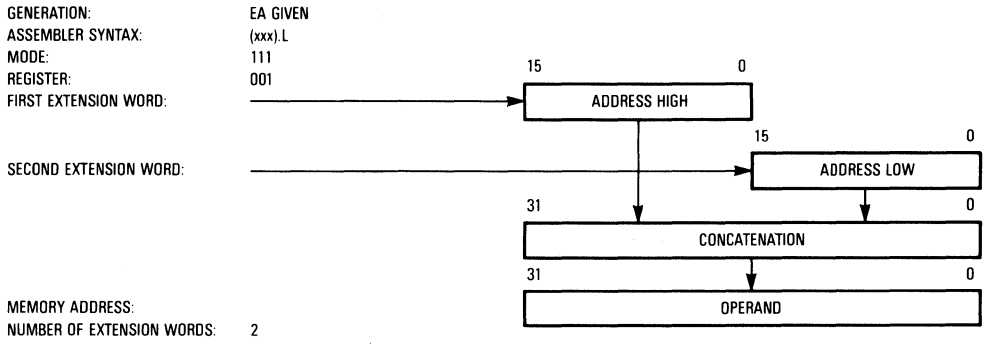
3.5.16 Absolute Short Address Mode

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.



3.5.17 Absolute Long Address 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.



3.5.18 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.

Floating-Point Single-Precision Operation
The single-precision operand is in two extension words.

Floating-Point Double-Precision Operation
The double-precision operand is in four extension words.

Floating-Point Extended-Precision Operation
The extended-precision operand is in six extension words.

Floating-Point Packed-Decimal Real Operation
Packed-decimal real operands are supported by software emulation, and therefore have a length dependent on the implementation.

The immediate data format is as follows:

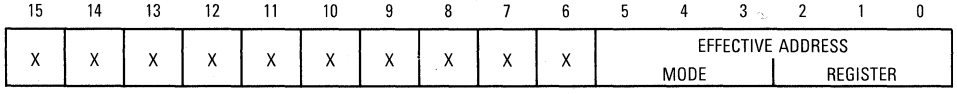
Generation:	Operand given
Assembler Syntax:	#xxx
Mode Field:	111
Register Field:	100
Number of Extension Words:	1, 2, 4 or 6, except for packed decimal real operands

3.6 EFFECTIVE ADDRESS ENCODING SUMMARY

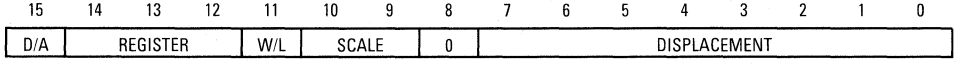
Most of the addressing modes use one of the three formats shown in Table 3-6. The single effective address instruction is in the format of the instruction word. The encoding of the mode field of this word selects the addressing mode. The register field contains the general register number or a value that selects the addressing mode when the mode field contains "111". 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 MC68040 contains ten extension words. It is a MOVE instruction with full format extension words for both the source and destination effective addresses, and with 32-bit base displacements and 32-bit outer displacements for both addresses.

Table 3-6. Effective Address Specification Formats

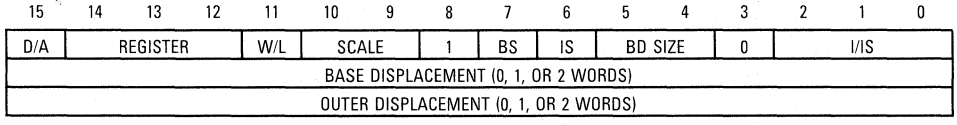
Single Effective Address Instruction Format



Brief Format Extension Word



Full Format Extension Word(s)



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

Field	Definition
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

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 3-7 lists the indexing and indirection operations corresponding to all combinations of IS and I/IS values.

Table 3-7. 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 Null 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 Null Outer Displacement
1	010	Memory Indirect with Word Outer Displacement
1	011	Memory Indirect with Long Outer Displacement
1	100–111	Reserved

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 3-8 shows the categories to which each of the effective addressing modes belong.

Table 3-8. Effective Addressing Mode Categories

Address Modes	Mode Field	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.

3.7 PROGRAMMER'S VIEW OF ADDRESSING MODES

Extensions to the indexed addressing modes, indirection, and full 32-bit displacements provide additional programming capabilities for the MC68020, the MC68030, and the MC68040. The following paragraphs describe addressing techniques that exploit these capabilities and summarize the addressing modes from a programming point of view.

Several of the addressing techniques described use data registers and address registers interchangeably. While the MC68040 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 specification of addresses with data registers should be used sparingly (if at all), particularly in programs that require maximum performance.

3.7.1 Addressing Capabilities

In the MC68020, MC68030, and the MC68040, setting the base register suppress (BS) bit in the full format extension word (Table 3-6) 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 MC68040, MC68030, and MC68020 can use both the data registers and the address registers to address memory. The capability 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 (see Figure 3-15).

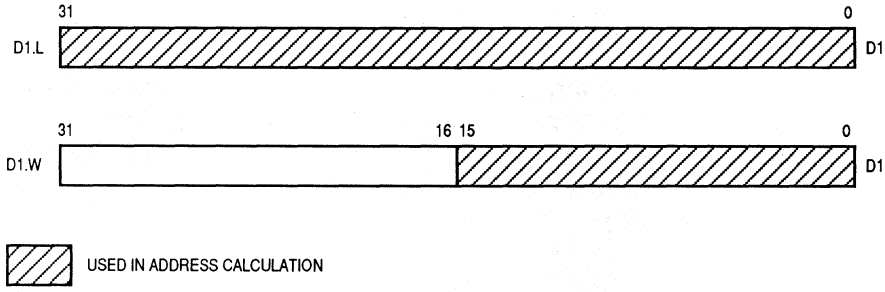


Figure 3-15. Using SIZE in the Index Selection

For the MC68020, MC68030, and the MC68040, 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 (see Figure 3-16).

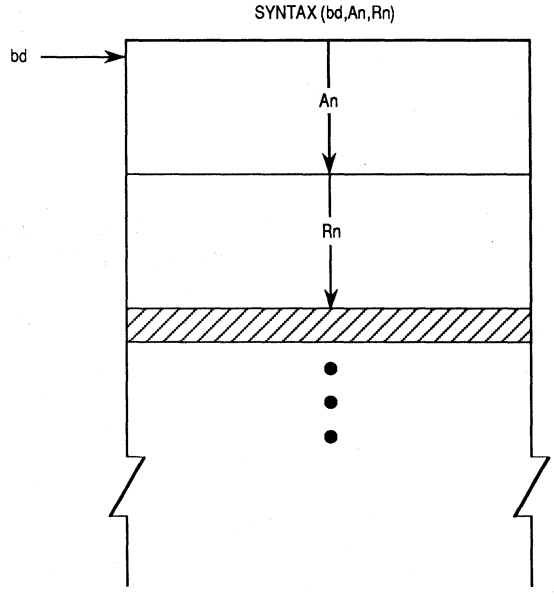


Figure 3-16. Using Absolute Address with Indexes

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 calculation (the actual value in the index register remains unchanged). This is equivalent to multiplying the register by one, two, four, or eight for 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), for example. Optionally, an address register that contains a dynamic displacement can be included in the address calculation (bd,An,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 3-17. In the second example, An contains the address of an array and Rn contains a subscript.

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 (see to Figure 3-18).

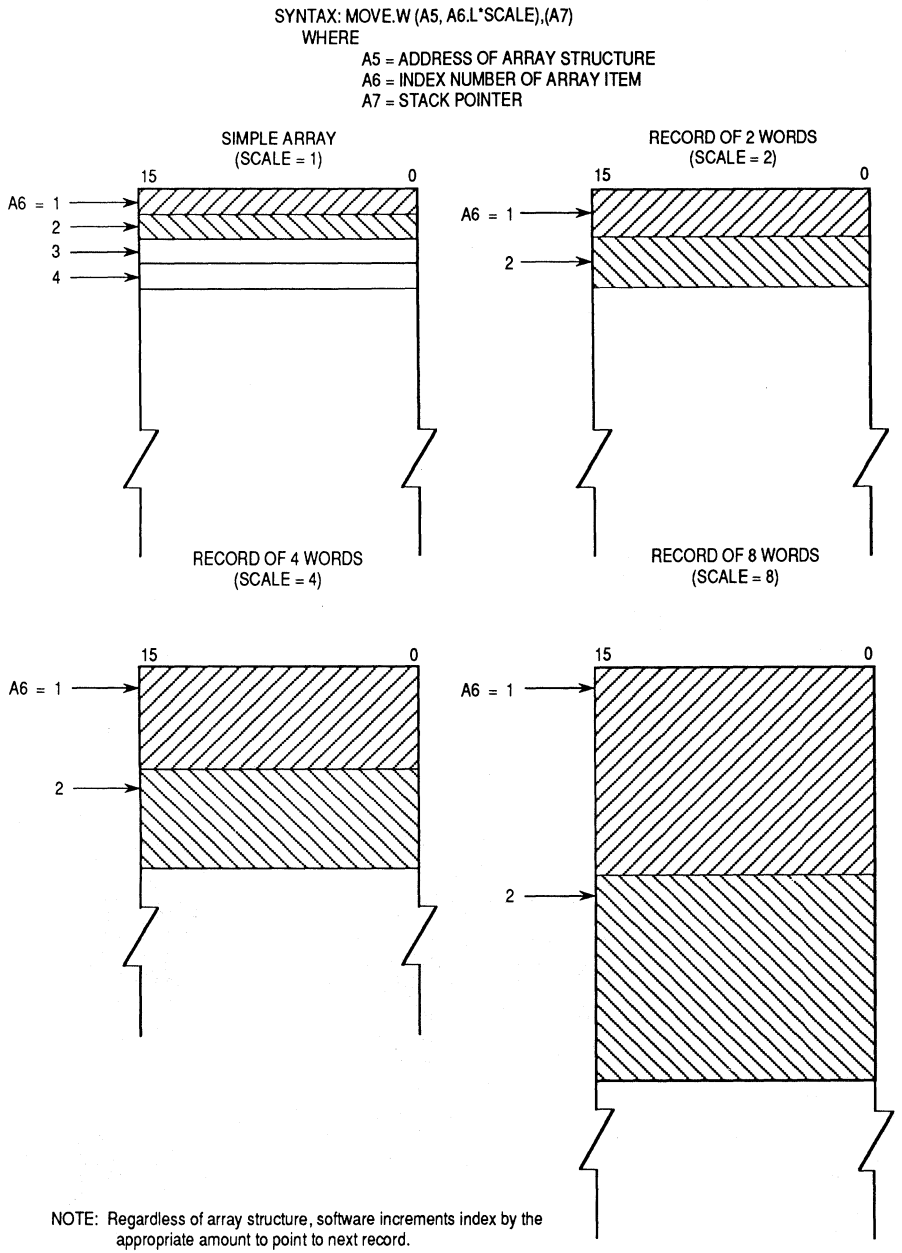


Figure 3-17. Addressing Array Items

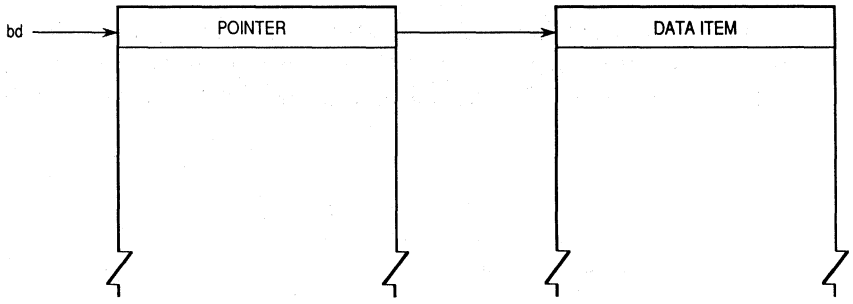


Figure 3-18. Using Indirect Absolute Memory Addressing

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 (see to Figure 3-19).

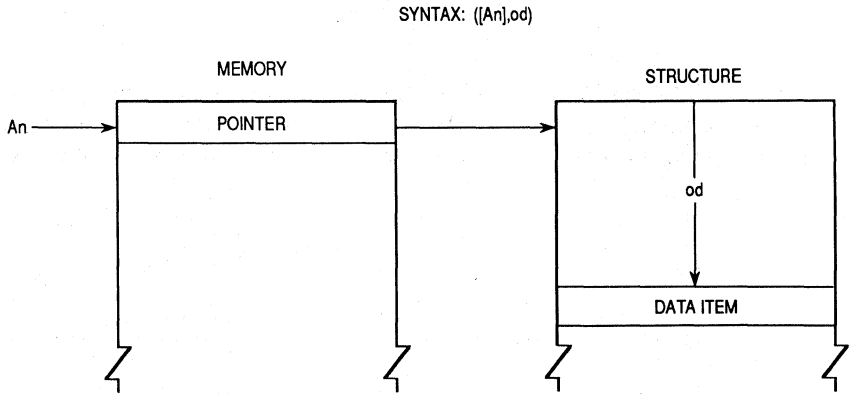


Figure 3-19. Accessing an Item in a Structure Using Pointer

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

The indirect, suppressed index register mode (see Figure 3-20) 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.

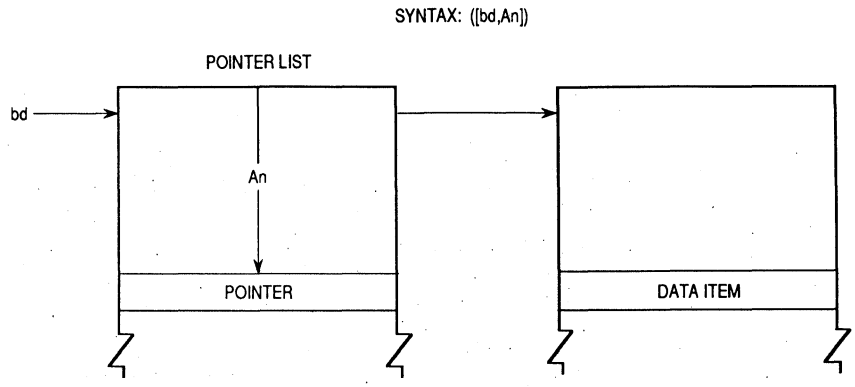


Figure 3-20. Indirect Addressing, Suppressed Index Register

The preindexed indirect mode (see Figure 3-21) 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.

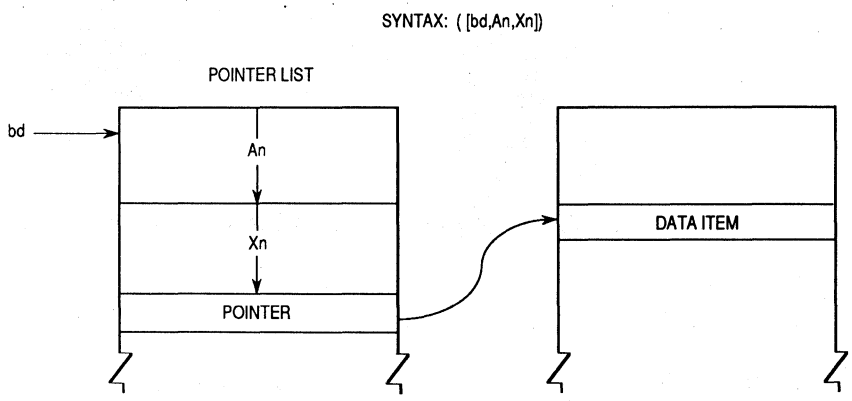


Figure 3-21. Preindexed Indirect Addressing

The postindexed indirect mode (sse Figure 3-22) 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 3-23 shows the preindexed indirect addressing with outer displacement mode.

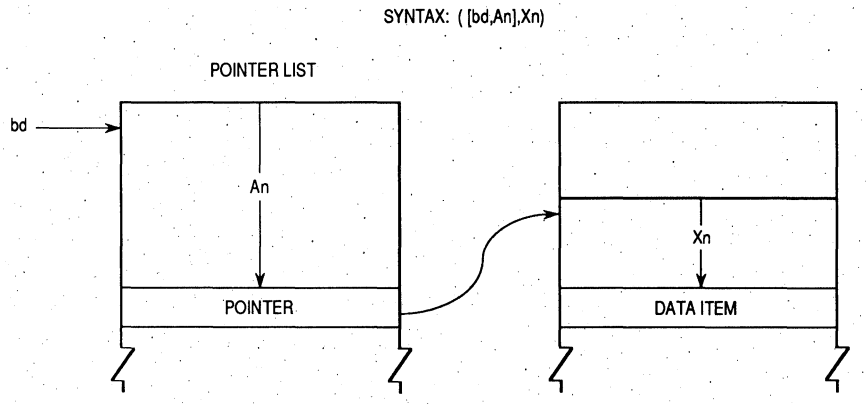


Figure 3-22. Postindexed Indirect Addressing

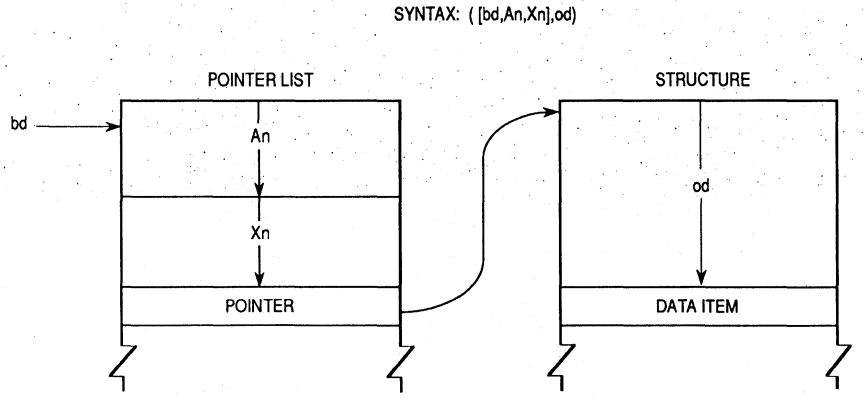


Figure 3-23. Preindexed Indirect with Outer Displacement

The postindexed indirect mode with outer displacement, Figure 3-24, 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.

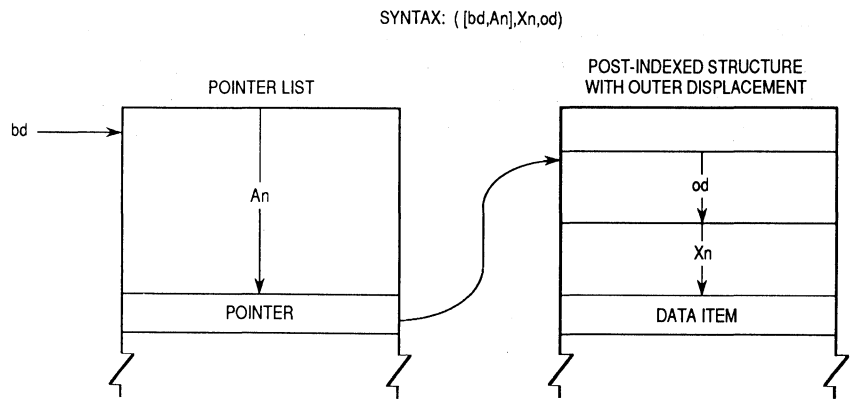


Figure 3-24. Postindexed Indirect Addressing with Outer Displacement

3.7.2 General Addressing Mode Summary

The addressing modes described in **3.7.1 Addressing Capabilities** 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 MC68040 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. These 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 one, two, four, or eight 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.
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 MC68040 architecture are:

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

Register Direct — Rn:
The contents of a register is 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 variable index.

(addr,An,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.

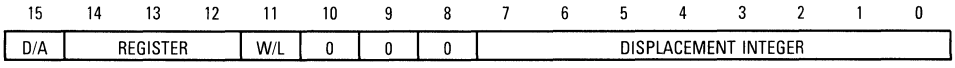
([postindexed],Rn*SCALE) — Memory pointer subscripted.

([postindexed], disp, Rn*SCALE) — Memory pointer subscripted with constant index.

3.8 M68000 FAMILY ADDRESSING COMPATIBILITY

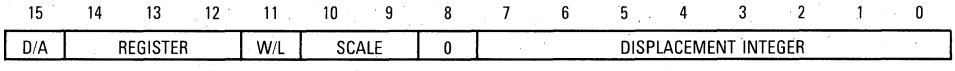
Programs can be easily transported from one member of the M68000 processor family to another in an upward compatible fashion. The user object code of each early member of the family is upward compatible with newer members, and can be executed on the newer microprocessor without change. The address extension word(s) are encoded with the information that allows the MC68020/MC68030/MC68040 to distinguish the new address extensions to the basic M68000 Family architecture. The address extension words for the early MC68000/MC68008/MC68l0 microprocessors and for the newer 32-bit MC68020/MC68030/MC68040 microprocessors are shown in Figure 3-25. Notice the encoding for SCALE used by the MC68020/MC68030/MC68040 is a compatible extension of the M68000 architecture. A value of zero for SCALE is the same encoding for both extension words; therefore, 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; so, 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, and while they do detect illegal instructions, they do not decode invalid encodings of the extension words as exceptions.

MC68000/MC68008/MC68010 Address Extension Word



D/A: 0 = Data Register Select
1 = Address Register Select
W/L: 0 = Word-Sized Operation
1 = Long-Word-Sized Operation

MC68020/MC68030/MC68040 Address Extension Word



D/A: 0 = Data Register Select
1 = Address Register Select
W/L: 0 = Word-Sized Operation
1 = Long-Word-Sized Operation
SCALE: 00 = Scale Factor 1 (Compatible with MC68000)
01 = Scale Factor 2 (Extension to MC68000)
10 = Scale Factor 4 (Extension to MC68000)
11 = Scale Factor 8 (Extension to MC68000)

Figure 3-25. M68000 Family Address Extension Words

3.9 OTHER DATA STRUCTURES

Stacks and queues are widely used data structures. The MC68040 implements a system stack and also provides instructions that support the use of user stacks and queues.

3.9.1 System Stack

Address register seven (A7) is used as the system stack pointer (SP). One of the three system stack registers (MSP, ISP, USP) is active at any one time. The M and S bits of the SR determine which SP is used. When S = 0 indicating user mode 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 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 MC68040 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 SSP (or A7) refers to the ISP address register. The active system stack pointer is implicitly referenced by all instructions that use the system stack. Each system stack fills from high to low memory.

A subroutine call saves the PC on the active system stack, and the return restores it from the active system stack. During the processing of traps and interrupts, both the PC and the SR are saved on the supervisor stack (either master or interrupt). Thus, the execution of supervisor level code is independent of user code and the condition of the user stack; conversely, user programs use the USP 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-size operands moved to or from the stack. In long-word-organized memory, aligning the stack pointer on a long-word address significantly increases the efficiency of stacking exception frames, subroutine calls and returns, and other stacking operations.

3.9.2 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 through 6), the user can implement a stack that is filled either from high memory to low memory or from low memory to high memory. Important considerations are:

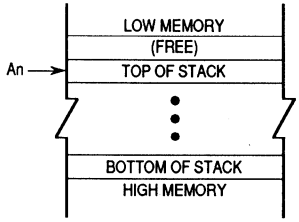
- Use the predecrement mode to decrement the register before its contents are used as the pointer to the stack.
- Use the postincrement mode to increment the register after its contents are used as the pointer to the stack.
- Maintain the stack pointer correctly when byte, word, and long-word items are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

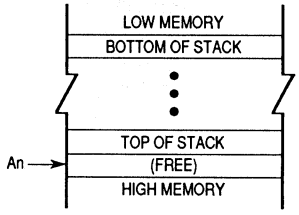


To implement stack growth from low-to-high memory, use:

(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:



3.9.3 Queues

The user can implement queues with the address register indirect with postincrement or predecrement addressing modes. Using a pair of address registers (two of A0 through A6), the user can implement a queue which is filled either from high memory to low memory, or from low memory 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.

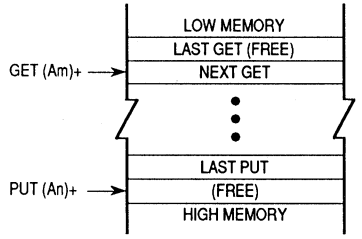
To implement growth of the queue from low-to-high memory, use:

(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:



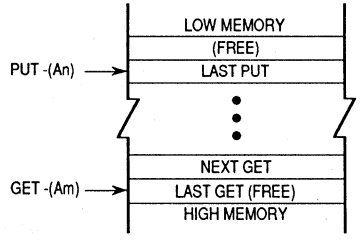
To implement the queue as a circular buffer, the relevant address register should be checked and adjusted, if necessary, before performing the "put" or "get" operation. The address register is adjusted by subtracting the buffer length (in bytes) from the register.

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 placed in 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 checked and adjusted, if necessary. The address register is adjusted by adding the buffer length (in bytes) to the register contents.

SECTION 4
INSTRUCTION SET SUMMARY

This section briefly describes the MC68040 instruction set. Refer to the MC68000PM/AD, *MC68000 Programming Reference Manual* for complete details on the MC68040 instruction set.

The following 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.

4.1 INSTRUCTION FORMAT

All MC68040 instructions consist of at least one word; some have as many as 11 words (see Figure 4-1). 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.

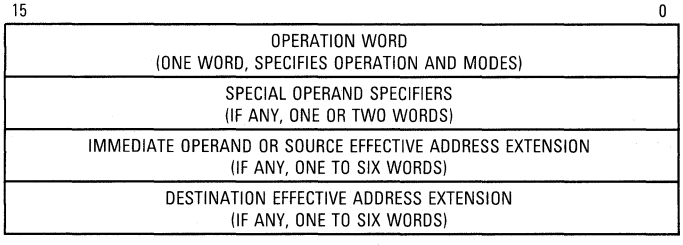


Figure 4-1. Instruction Word General Format

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:

- Register Specification — A register field of the instruction contains the number of the register.
- Effective Address — An effective address field of the instruction contains address mode information.
- 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 2 PROGRAMMING MODEL** contains detailed register information.

Effective address information includes the registers, displacements, and absolute addresses for the effective address mode. **SECTION 3 DATA ORGANIZATION AND ADDRESSING CAPABILITIES** describes the effective address modes in detail.

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

4.2 INSTRUCTION SUMMARY

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

Data Movement
Integer Arithmetic
Floating-Point Arithmetic
Logical
Shift and Rotate
Bit Manipulation
Bit Field Manipulation
Binary Coded Decimal Arithmetic
Program Control
System Control
Memory Management
Cache Maintenance
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
FPcr = 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 decimal 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
× = 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 low 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

4.2.1 Data Movement Instructions

The MOVE and FMOVE 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 16-byte block (MOVE16), 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). The MOVE16 instruction is an MC68040 extension to the M68000 instruction set.

The FMOVE instructions move operands into, between, and from the floating-point data registers. Data format conversion functions for the FPU instructions are implicitly supported since all external data formats movement of operands to and from the floating-point control and status registers (FPCR, FPSR, and FPIAR). For operands moved into a floating-point data register, FSMOVE and FDMOVE explicitly select single and double precision rounding of the result, respectively. FMOVEM moves any combination of either floating-point data registers or floating-point control registers. Table 4-1 is a summary of the integer and floating-point data movement operations.

Table 4-1. Data Movement Operations

Instruction	Operand Syntax	Operand Size	Operation
EXG	Rn, Rn	32	Rn ⇹ Rn
FMOVE	FPm,FPn <ea>,FPn FPm,<ea> <ea>,FPcr FPcr,<ea>	X B,W,L,S,D,X,P B,W,L,S,D,X,P 32 32	source ➧ destination
FSMOVE, FDMOVE	FPm,FPn <ea>,FPn	X B,W,L,S,D,X	source ➧ destination, round destination to single or double precision
FMOVEM	<ea>,<list>[1] <ea>,Dn <list>[1],<ea> Dn,<ea>	32,X X 32,X X	listed registers ➧ destination source ➧ listed registers
LEA	<ea>,An	32	<ea> ➧ An
LINK	An,#<d>	16,32	Sp − 4 ➧ SP; An ➧ (SP); SP ➧ An, SP + D ➧ SP
MOVE MOVE16 MOVEA	<ea>,<ea> <ea>,<ea> <ea>,An	8,16,32 16 bytes 16,32 ➧ 32	source ➧ destination aligned 16-byte block ➧ destination
MOVEM	list,<ea> <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

NOTE 1: The register list may include any combination of the eight floating-point data registers, or it may contain any combination of the three control registers (FPCR, FPSR, and FPIAR). If the register list mask resides in a data register, only floating-point data registers may be specified.

4.2.2 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 extend (EXT), and negate binary with extend (NEGX). Refer to Table 4-2 for a summary of the integer arithmetic operations.

Table 4-2. 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	16 × 16 ➧ 32 32 × 32 ➧ 32 32 × 32 ➧ 64	source × 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

4.2.3 Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions supported by the MC68040 are an enhanced subset of the MC68881/MC68882 floating-point coprocessor instructions. Several instructions in the MC68040 include explicit single and double precision rounding of the result as part of their operation. For example, the FADD instruction uses the default rounding precision selected in the FPU, while FSADD and FDADD force rounding of the result to single and double precision, respectively. The following paragraphs describe the floating-point instructions, which are organized into two categories of operation: dyadic (requiring two operands), and monadic (requiring one operand).

The dyadic floating-point instructions provide several arithmetic functions that require two input operands, such as add and subtract. For these operations, the first operand may be located in memory, in an integer data register, or in a floating-point data register, and the second operand is always contained in a floating-point data register. The results of the operation are stored in the register specified as the second operand. All operations support any data format, and return results rounded to either extended precision, or single or double precision for the instructions which explicitly specify the rounding precision (such as FSADD and FDADD). The general format of the dyadic instructions is given in Table 4-3; the available operations are listed in Table 4-4.

Table 4-3. Dyadic Floating-Point Operation Format

Instruction	Operand Syntax	Operand Format	Operation
F⟨dop⟩	⟨ea⟩,FPn FPm,FPn	B,W,L,S,D,X,P X	FPn ⟨function⟩ source ➧ FPn

where:
<dop> is any one of the dyadic operation specifiers.

Table 4-4. Dyadic Floating-Point Operations

Instruction	Function
FADD, FSADD, FDADD	Add
FCMP	Compare
FDIV, FSDIV, FDDIV	Divide
FMUL, FSMUL, FDMUL	Multiply
FSUB, FSSUB, FDSUB	Subtract

The monadic floating-point instructions provide several arithmetic functions that require only one input operand. Unlike the integer counterparts to these functions (e.g., NEG <ea>), a source and a destination may be specified. The operation is performed on the source operand and the result is stored in the destination, which is always a floating-point data register. When the source is not a floating-point data register, all data formats are supported; the data format is always extended precision for register-to-register operations. The general format of these instuctions is shown in Table 4-5, and the available operations are listed in Table 4-6.

Table 4-5. Monadic Floating-Point Operation Format

Instruction	Operand Syntax	Operand Format	Operation
F⟨mop⟩	⟨ea⟩,FPn	B,W,L,S,D,X,P	source ➧ function ➧ FPn
	FPm,FPn	X	
	FPn	X	FPn ➧ function ➧ FPn

where:
<mop> is any one of the monadic operation specifiers.

Table 4-6. Monadic Floating-Point Operations

Instruction	Function
FABS, FSABS, FDABS	Absolute Value
FNEG, FSNEG, FDNEG	Negate
FSQRT, FSSQRT, FDSQRT	Square Root

4.2.4 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. Table 4-7 summarizes the logical operations.

Table 4-7. 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 ⊕ 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

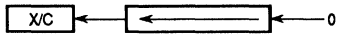
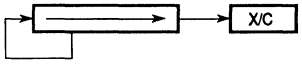
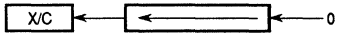
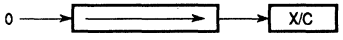
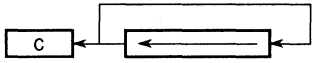
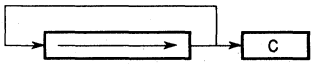
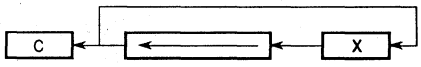
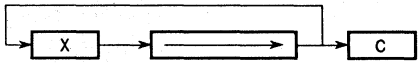
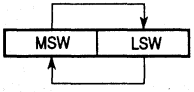
4.2.5 Shift and Rotate Instructions

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 4-8 is a summary of the shift and rotate operations.

Table 4-8. Shift and Rotate Operations

Instruction	Operand Syntax	Operand Size	Operation
ASL	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ASR	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
LSL	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
LSR	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ROL	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ROR	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ROXL	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ROXR	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
SWAP	Dn	32	

4.2.6 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 4-9, the summary of the bit manipulation operations, Z refers to the zero bit of the status register.

Table 4-9. 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

4.2.7 Bit Field Instructions

The MC68040 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 4-10 is a summary of the bit field operations.

Table 4-10. 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.

4.2.8 Binary Coded Decimal Instructions

Five instructions support operations on binary coded decimal (BCD) numbers. The arithmetic operations on packed binary coded decimal 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 4-11 is a summary of the binary coded decimal operations.

Table 4-11. Binary Coded Decimal Operations

Instruction	Operand Syntax	Operand Size	Operation
ABCD	Dn,Dn –(An),–(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),–(An) #⟨data⟩ Dn,Dn,#⟨data⟩	8 ➧ 16 8 ➧ 16	packed source ➧ unpacked source unpacked source + immediate data ➧ unpacked destination

4.2.9 Program Control Instructions

A set of subroutine call and return instructions and conditional and unconditional branch instructions perform program control operations. Also included are test operand instructions (TST and FTST) that set the integer or floating-point condition codes for use by the other program and system control instructions, and a no operation instruction (NOP) that may be used to force synchronization of the internal pipelines. Table 4-12 summarizes these instructions.

Table 4-12. Program Control Operations

Instruction	Operand Syntax	Operand Size	Operation
Integer and Floating-Point Conditional			
Bcc, FBcc	<label>	8,16,32	if condition true, then PC + d ➧ PC
DBcc, FDBcc	Dn,<label>	16	if condition false, then Dn − 1 ➧ Dn if Dn ≠ −1, then PC + d ➧ PC
Scc, FScc	<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
FNOP	none	none	PC + 4 ➧ 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
Test Operand			
TST	<ea>	8, 16, 32	set integer condition codes
FTST	<ea> FPn	B,W,L,S,D,X,P X	set floating-point condition codes

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

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

*Not applicable to the Bcc instructions.

The conditional mnemonics for the floating-point conditional instructions are shown in Table 4-13, along with the conditional test function. The FPU supports 32 conditional tests that are separated into two groups; 16 that cause an exception if an unordered condition is present when the conditional test is attempted, and 16 that do not cause an exception if an unordered condition is present. (An unordered condition occurs when an input to an arithmetic operation is a NAN.) Refer to **4.4.2 Conditional Test Definitions** for a detailed description of the conditional equation used by each test.

Table 4-13. FPU Conditional Test Mnemonics

Exception on Unordered		No Exception on Unordered	
GE	Greater Than or Equal	OGE	Ordered Greater Than or Equal
GL	Greater Than or Less Than	OGL	Ordered Greater Than or Less Than
GLE	Greater Than or Less	OR	Ordered
GT	Greater Than	OGT	Ordered Greater Than
LE	Less Than or Equal	OLE	Ordered Less Than or Equal
LT	Less Than	OLT	Ordered Less Than
NGE	Not (greater than or equal)	UGE	Unordered or Greater Than Equal
NGL	Not (greater than or less than)	UEQ	Unordered or Equal
NGLE	Not (greater than or less than or equal)	UN	Unordered
NGT	Not Greater Than	UGT	Unordered or Greater Than
NLE	Not (less than or equal)	ULE	Unordered or Less Than or Equal
NLT	Not Less Than	ULT	Unordered or Less Than
SEQ	Signaling Equal	EQ	Equal
SNE	Signaling Not Equal	NE	Not Equal
SF	Signaling Always False	F	Always False
ST	Signaling Always True	T	Always True

4.2.10 System Control Instructions

Privileged instructions, trapping instructions, and instructions that use or modify the condition code register (CCR) provide system control operations. Table 4-14 summarizes these instructions. FSAVE and FRESTORE save and restore the nonuser visible portion of the FPU during context switches in a virtual memory or multitasking system. The conditional trap instructions use the same conditional tests as their corresponding program control instructions and allow an optional 16- or 32-bit immediate operand to be included as part of the instruction for passing parameters to the operating system. All of these instructions cause the processor to flush the instruction pipe. See **4.4.5 Operation Post Processing** for more details on condition codes.

Table 4-14. System Control Operations

Instruction	Operand Syntax	Operand Size	Operation
Privileged			
ANDI	#<data>,SR	16	immediate data ∧ SR ➧ SR
EORI	#<$\overline{\text{data}}$>,SR	16	immediate data ⊕ SR ➧ SR
FRESTORE	<ea>	none	state frame ➧ internal floating-point registers
FSAVE	<ea>	none	internal floating-point registers ➧ state frame
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 $\overline{\text{RSTO}}$ 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, the 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
FTRAPcc	none #<data>	none 16,32	if floating-point cc true, then TRAP exception
TRAPV	none	none	if V then take overflow TRAP exception
Condition Code Register			
ANDI	#<data>,CCR	8	immediate data ∧ 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

4.2.11 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 4-15 summarizes these instructions.

Table 4-15. 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

4.2.12 Cache Instructions

The cache instructions provide maintenance functions for managing the instruction and data caches. CINV invalidates cache entries in both caches, and CPUSH pushes dirty data from the data cache to update memory. Both instructions can operate on either or both caches, and can select a single cache line, all lines in a page, or the entire cache. Table 4-16 summarizes these instructions.

Table 4-16. Cache Instructions

Instruction	Operand Syntax	Operand Size	Operation
CINVL	caches,(An)	none	Invalidate cache line
CINVP	caches, (An)	none	Invalidate cache page
CINVA	caches	none	Invalidate entire cache
CPUSHL	caches,(An)	none	Push selected dirty data cache lines, then invalidate selected cache lines
CPUSHP	caches, (An)	none	
CPUSHA	caches	none	

4.2.13 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. Table 4-17 lists these instructions.

Table 4-17. Multiprocessor Operations (Read-Modify-Write)

Instruction	Operand Syntax	Operand Size	Operation
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]

4.3 INTEGER CONDITION CODES

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

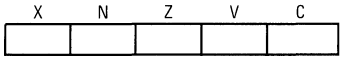
The first four bits represent a condition of the result of a processor operation. The X bit is an operand for multiprecision computations; when it is used, it is set to the value of the carry bit. The carry bit and the multiprecision extend bit are separate in the M68000 Family to simplify programming techniques that use them.

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 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.

4.3.1 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 4-18 lists each instruction and how it affects the condition code bits.

Table 4-18. Condition Code Computations (Sheet 1 of 2)

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

Table 4-18. Condition Code Computations (Sheet 2 of 2)

Operations	X	N	Z	V	C	Special Definition
SBCD, NBCD	*	U	?	U	?	C = Decimal Borrow Z = Z Λ $\overline{Rm}$ Λ . . . Λ $\overline{Ro}$
NEG	*	*	*	?	?	V = Dm Λ Rm C = Dm V Rm
NEGX	*	*	?	?	?	V = Dm Λ Rm C = Dm V Rm Z = Z Λ $\overline{Rm}$ Λ . . . Λ $\overline{R0}$
BTST, BCHG, BSET, BCLR	—	—	?	—	—	Z = $\overline{Dn}$
BFTST, BFCHG, BFSET, BFCLR	—	?	?	0	0	N = Dm Z = $\overline{Dm}$ Λ $\overline{DM-1}$ Λ . . . Λ $\overline{D0}$
BFEXTS, BFEXTU, BFFFO	—	?	?	0	0	N = Sm Z = $\overline{Sm}$ Λ $\overline{Sm-1}$ Λ . . . Λ $\overline{S0}$
BFINS	—	?	?	0	0	N = Dm Z = $\overline{Dm}$ Λ $\overline{DM-1}$ Λ . . . Λ $\overline{D0}$
ASL	*	*	*	?	?	V = Dm Λ ($\overline{Dm-1}$ V . . . V $\overline{Dm-r}$) V $\overline{Dm}$ Λ (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 = $\overline{Rm}$
Z = $\overline{Rm}$ Λ . . . Λ $\overline{R0}$
Sm = Source 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

4.3.2 Conditional Tests

Table 4-19 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 4-19. Conditional Tests

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

• = Boolean AND
+ = Boolean OR
$\overline{N}$ = Boolean NOT N

*Not available for the Bcc instruction.

4.4 FLOATING-POINT DETAILS

The following paragraphs describe accuracy considerations and conditional tests which can be used to change program flow based on the floating-point conditions. The operation tables in the instruction descriptions are also discussed, followed by details on NANs and floating-point condition codes.

4.4.1 Computational Accuracy

Whenever an attempt is made to represent a real number in a binary format of finite precision, there is a possibility that the number cannot be represented exactly; this is commonly referred to as round-off error. Furthermore, when two inexact numbers are used in a calculation, the error present in each number is reflected and possibly aggravated in the result.

One of the major reasons that the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std. 754-1985) was developed was to define the error bounds for calculation of binary floating-point values so that all machines conforming to the standard produce the same results for an operation. The operation must meet the following conditions.

1. same input values,
2. same rounding mode, and
3. same precision.

The IEEE standard specifies not only the format of data items, but also defines:

- the maximum allowable error that may be introduced during a calculation, and
- the manner in which rounding of the result is performed.

The *IEEE Specification for Binary Floating-Point Arithmetic* specifies that the following operations must be supported for each data format: add, subtract, multiply, divide, remainder, square root, integer part, and compare. Conversions between the various data formats are also required. In addition to these arithmetic functions (remainder and integer part are supported in software), the FPU also supports the nontranscendental operations of: absolute value, negate, and test. Since the IEEE specification defines the error bounds to which all calculations are performed, the result obtained by any conforming machine can be predicted exactly for a particular precision and rounding mode. The error bound defined by the IEEE specification is one-half unit in the last place of the destination data format in the round-to-nearest mode, and one unit in the last place in the other rounding modes.

The FPU performs all calculations using a 67-bit mantissa for the intermediate results. The three bits beyond the precision of the extended format allow the FPU to perform all calculations as if to infinite performing calculations in this manner, the final result is always correct for the specified destination data format before rounding is performed (unless an overflow or underflow error occurs). The specified rounding operation then produces a number that is as close as possible to the infinitely-precise-intermediate value and is still representable in the selected precision. An example of how the 67-bit mantissa allows the FPU to meet the error bound of the IEEE specification is as follows:

	Mantissa	l	g	r	s	
Intermediate Result:	x.x.....x00		1	0	0	(Tie Case)
Round-to-Nearest Result:	x.x.....x00					

In this case, the least-significant bit (l) of the rounded result is not incremented, even though the guard bit (g) is set in the intermediate result. The IEEE standard specifies that tie cases should be handled in this manner. Assuming that the destination data format is extended, if the difference between the infinitely precise intermediate result and the round-to-nearest result is calculated, the relative difference is 2^{-64} (the value of the guard bit). This error is equal to one-half of the value of the least significant bit, and is the worst-case error that can be introduced when using the round-to-nearest mode. Thus, the term one-half unit in the last place correctly identifies the error bound for this operation. This error specification is the relative error present in the result; the absolute error bound is equal to $2^{\text{exponent}} \times 2^{-64}$. An example of the error bound for the other rounding modes is as follows:

	Mantissa	l	g	r	s
Intermediate Result:	x.x.....x00		1	1	1
Round-to-Zero Result:	x.x.....x00				

In this case, the difference between the infinitely precise result and the rounded result is $2^{-64}+2^{-65}+2^{-66}$, which is slightly less than 2^{-63} (the value of the least significant bit). Thus, the error bound for this operation is not more than one unit in the last place. For all of the arithmetic operations, these error bounds are met by the FPU, thus providing accurate and repeatable results.

4.4.2 Conditional Test Definitions

The FPU provides a very simple mechanism for performing conditional tests of the result of any arithmetic floating-point operation. First, the condition code bits in the FPSR are set or cleared at the end of any arithmetic operation or move operation to a single floating-point data register. The condition code bits are always set consistently based on the result of the operation. Second, 32 conditional tests are provided that allow floating-point conditional instructions to test floating-point conditions in exactly the same way as the integer conditional instructions test the interger condition codes. The combination of the consistent setting of the condition code bits and the simple programming of conditional instructions gives the MC68040 a very flexible, high-performance method of altering program flow based on floating-point results.

One important programming consideration is that the inclusion of the NAN data type in the IEEE floating-point number system requires each conditional test to include the NAN condition code bit in its Boolean equation. Because a comparison of a NAN with any other data type is unordered (i.e., it is impossible to determine if a NAN is bigger or smaller than an in-range number), the compare instruction sets the NAN condition code bit when an unordered compare is attempted. All arithmetic instructions also set the NAN bit if the result of an operation is a NAN. The conditional instructions interpret the NAN condition code bit equal to one as the unordered condition.

The inclusion of the unordered condition in floating-point branches destroys the familiar trichotomy relationship (greater than, equal, less than) that exists for integers. For example, the opposite of floating-point branch greater than (FBGT) is not floating-point branch less than or equal (FBLE). Rather, the opposite condition is floating-point branch not greater than (FBNGT). If the result of the previous instruction was unordered, FBNGT is true; whereas, both FBGT and FBLE would be false since unordered fails both of these tests (and sets BSUN). Compiler programmers should be particularly careful of the lack of trichotomy in the floating-point branches since it is common for compilers to invert the sense of conditions.

In the following paragraphs, the conditional tests are described in three main categories:

1. IEEE nonaware tests,
2. IEEE aware tests, and
3. Miscellaneous.

The set of IEEE nonaware tests is best used:

1. when porting a program from a system that does not support the IEEE standard to a conforming system, or
2. when generating high-level language code that does not support IEEE floating-point concepts (i.e., the unordered condition).

When using the set of IEEE nonaware tests, the user receives a BSUN exception whenever a branch is attempted and the NAN condition code bit is set, unless the branch is an FBEQ or and FBNE. If the BSUN trap is enabled in the FPCR register, the exception causes a trap. Therefore, the IEEE nonaware program is interrupted if an unexpected condition occurs.

The IEEE aware branch set should be used in programs that contain ordered and unordered conditions by compilers and programmers who are knowledgeable of the IEEE standard. Since the ordered or unordered attribute is explicitly included in the conditional test, the BSUN bit is not set in the status register EXC byte when the unordered condition occurs.

Table 4-20 lists the IEEE nonaware tests. All the conditional tests in Table 4-20, except EQ and NE, set the BSUN bit in the status register exception byte if the NAN condition code bit is set when a conditional instruction is executed.

Table 4-20. IEEE Nonaware Tests

Mnemonic	Definition	Equation	Predicate
EQ	Equal	Z	000001
NE	Not Equal	$\overline{Z}$	001110
GT	Greater Than	$\overline{NAN} v Z v \overline{N}$	010010
NGT	Not Greater Than	$NAN v Z v N$	011101
GE	Greater Than or Equal	$Z v (\overline{NAN} v \overline{N})$	010011
NGE	Not (greater than or equal)	$NAN v (N \wedge \overline{Z})$	011100
LT	Less Than	$N \wedge (\overline{NAN} v \overline{Z})$	010100
NLT	Not Less Than	$NAN v (Z v \overline{N})$	011011
LE	Less Than or Equal	$Z v (N \wedge \overline{NAN})$	010101
NLE	Not (less than or equal)	$NAN v (\overline{N} v \overline{Z})$	011010
GL	Greater or Less Than	$\overline{NAN} v \overline{Z}$	010110
NGL	Not (greater or less than)	$NAN v Z$	011001
GLE	Greater, Less or Equal	$\overline{NAN}$	010111
NGLE	Not (greater, less or equal)	NAN	011000

where:
"v" = Logical OR
"$\wedge$" = Logical AND

Table 4-21 lists the IEEE aware tests. None of the conditional tests in Table 4-21 set the BSUN bit in the status register exception byte under any circumstances.

Table 4-21. IEEE Aware Tests

Mnemonic	Definition	Equation	Predicate
EQ	Equal	Z	000001
NE	Not Equal	$\overline{Z}$	001110
OGT	Ordered Greater Than	$\overline{NAN \vee Z \vee N}$	000010
ULE	Unordered or Less or Equal	NANvZvN	001101
OGE	Ordered Greater Than or Equal	Zv($\overline{NAN \vee N}$)	000011
ULT	Unordered or Less Than	NANv(N$\wedge\overline{Z}$)	001100
OLT	Ordered Less Than	N$\wedge$($\overline{NAN \vee Z}$)	000100
UGE	Unordered or Greater or Equal	NANvZvN	001011
OLE	Ordered Less Than or Equal	Zv(N$\wedge\overline{NAN}$)	000101
UGT	Unordered or Greater Than	NANv($\overline{N \vee Z}$)	001010
OGL	Ordered Greater or Less Than	$\overline{NAN \vee Z}$	000110
UEQ	Unordered or Equal	NANvZ	001001
OR	Ordered	$\overline{NAN}$	000111
UN	Unordered	NAN	001000

where:
"v" = Logical OR
"$\wedge$" = Logical AND

The miscellaneous tests shown in Table 4-22 are not generally used but are implemented for completeness of the set. If the NAN condition code bit is set, T and F do not set the BSUN bit, but SF, ST, SEQ, and SNE do set the BSUN bit.

Table 4-22. Miscellaneous Tests

Mnemonic	Definition	Equation	Predicate
F	False	False	000000
T	True	True	001111
SF	Signaling False	False	010000
ST	Signaling True	True	011111
SEQ	Signaling Equal	Z	010001
SNE	Signaling Not Equal	$\overline{Z}$	011110

4.4.3 Operation Tables

An operation table is included for most floating-point instructions. This table lists the result data types for the instruction based on types of input operand(s). For example, Figure 4-2 illustrates the table for the FADD instruction.

Destination \ Source	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
In Range +	Add		Add		+inf	−inf
In Range −						
Zero +	Add		+0.0	0.0[1]	+inf	−inf
Zero −			0.0[1]	−0.0		
Infinity +	+inf		+inf		+inf	NAN[2]
Infinity −	−inf		−inf		NAN[2]	−inf

NOTES:
1. Returns +0.0 in rounding modes RN, RZ, and RP; returns −0.0 in RM.
2. Sets the OPERR bit in the FPSR exception byte.
3. If either operand is a NAN, refer to **4.4.4 NANs** for more information.

Figure 4-2. Operation Table Example (FADD Instruction)

In the example shown in Figure 4-2, the type of the source operand is shown along the top, and the type of the destination operand is shown along the side. In-range numbers are normalized, denormalized, unnormalized real numbers, or integers that are converted to normalized or denormalized extended precision numbers upon entering the FPU.

From Figure 4-2, it can be seen that if both the source and destination operand are positive zero, the result is also a positive zero. For another example, if the source operand is a positive zero and the destination operand is an in-range number, then the ADD algorithm is executed to obtain the result. If a labal such as ADD appears in the table, it indicates that the FPU performs the indicated operation and returns the correct result.

A third example of using the tables is when a source operand is plus infinity, and the destination operand is minus infinity. Since the result of such an operation is undefined, a not-a-number (NAN) is returned as the result, and the OPERR bit is set in the floating-point status register (FPSR) exception byte.

4.4.4 NANs

In addition to the data types covered in the operation tables for each floating-point instruction, NANs can also be used as inputs to an arithmetic operation. The operation tables do not contain a row and column for NANs because NANs are handled the same way in all operations.

If either operand (but not both operands) of an operation is a nonsignaling NAN, then that NAN is returned as the result. If both operands are nonsignaling NANs, then the destination operand nonsignaling NAN is returned as the result.

If either operand to an operation is a signaling NAN (SNAN), then the SNAN bit is set in the FPSR EXC byte. If the SNAN trap enable bit is set in the floating-point control register (FPCR) ENABLE byte, then the trap is taken and the destination is not modified. If the SNAN trap enable bit is not set, then the SNAN is converted to a nonsignaling NAN (by setting the SNAN bit in the operand to a one), and the operation continues as described in the preceding paragraph for nonsignaling NANs.

4.4.5 Operation Post Processing

Most operations end with a post processing step. While reading the summary for each instruction, it should be assumed that an instruction performs post processing unless the summary specifically states that the instruction does not do so. The following paragraphs describe post processing in detail.

4.4.5.1 SETTING FLOATING-POINT CONDITION CODES. Unlike the integer arithmetic condition codes, the floating-point condition codes are either not changed by an instruction or are always set in the same way by any instruction. Therefore, it is not necessary to include details of condition code settings for each floating-point instruction in the detailed instruction descriptions. The following paragraphs describe how floating-point condition codes are set for all instructions that modify any condition codes.

Refer to **SECTION 2 PROGRAMMING MODEL** for a description of the floating-point condition code byte. The four conditions code bits are:

N—Sign of Mantissa
Z—Zero
I—Infinity
NAN—Not-A-Number

The condition code bits differ slightly from the integer condition codes. The floating-point condition codes are not dependent on the type of operation being performed, but rather, can be set at the end of the operation by examining the result. (The M68000 integer condition codes bits N and Z have this characteristic, but the V and C bits are set differently for different instructions.) At the end of any floating-point operation, the result is inspected, and the condition code bits are set or cleared accordingly. For example, if the result of an operation is a positive normalized number, then all of the condition code bits are set to zero. If the result is a minus infinity, then the N and I bits are set, and the Z and NAN bits are cleared.

4.4.5.2 UNDERFLOW, ROUND, OVERFLOW. During calculation of an arithmetic result, the arithmetic logic unit (ALU) of the FPU has more precision and range than the 80-bit extended precision format. However, the final result of these operations is an extended precision floating-point value. In some cases, an internal result becomes either smaller or larger than can be represented in extended precision. Also, the operation may have generated a larger exponent or more bits of precision than can be represented in the chosen rounding precision. For these reasons, every arithmetic instruction ends by rounding the result and checking for overflow and underflow.

At the completion of an arithmetic operation, the internal result is checked to see if it is too small to be represented as a normalized number in the selected precision. If so, the underflow (UNFL) bit is set in the FPSR EXC byte. It is also denormalized unless denormalization provides a zero value. Denormalizing a number causes a loss of accuracy, but a zero is not returned unless absolutely necessary. If a number is grossly underflowed, the FPU returns a correctly signed zero or the correctly signed smallest denormalized number, depending on the rounding mode in effect.

If no underflow occurs, the internal result is rounded according to the user-selected rounding precision and rounding mode. After rounding, the inexact bit (INEX2) is set appropriately. Lastly, the magnitude of the result is checked to see if it is too large to be represented in the current rounding precision. If so, the overflow (OVFL) bit is set and a correctly signed infinity or correctly signed largest normalized number is returned, depending on the rounding mode in effect.

For details on underflow, rounding, and overflow refer to **SECTION 9 EXCEPTION PROCESSING**.

4.5 INSTRUCTION SET SUMMARY

Table 4-23 provides a alphabetized listing of the MC68040 instruction set listed by opcode, operation and syntax.

Table 4-23 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 lists contain the notations used in Table 4-23.

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 detination 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 operator
−(<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

×—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 interrupts

If <condition> then <operations> else <operations>—The condition is tested. If true, the operations after "then" are performed. If the condition is 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 4-23. Instruction Set Summary (Sheet 1 of 7)

Opcode	Operation	Syntax
ABCD	$Source_{10} + Destination_{10} + X \blacklozenge Destination$	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∧Destination ♦ Destination	AND ⟨ea⟩,Dn AND Dn,⟨ea⟩
ANDI	Immediate Data∧Destination ♦ Destination	ANDI #⟨data⟩,⟨ea⟩
ANDI to CCR	Source∧CCR ♦CCR	ANDI #⟨data⟩,CCR
ANDI to SR	If supervisor state the Source∧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 4-23. Instruction Set Summary (Sheet 2 of 7)

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 Dn > Source then TRAP	CHK ⟨ea⟩,Dn
CHK2	If Rn < lower bound or Rn > upper bound then TRAP	CHK2 ⟨ea⟩,Rn
CINV	If supervisor state then invalidate selected cache lines else TRAP	CINVL <caches>[1],(An) CINVP <caches>[1],(An) CINVA <caches>[1]
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
CPUSH	If supervisor state then if data cache then push selected dirty data cache lines invalidate selected cache lines else TRAP	CPUSHL <caches>[1],(An) CPUSHP <caches>[1],(An) CPUSHA <caches>[1]
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⟩,Dn 32/16 ⇨ 16r:16q DIVS.L ⟨ea⟩,Dq 32/32 ⇨ 32q DIVS.L ⟨ea⟩,Dr:Dq 64/32 ⇨ 32r:32q DIVSL.L ⟨ea⟩,Dr:Dq 32/32 ⇨ 32r:32q
DIVU DIVUL	Destination/Source ⇨ Destination	DIVU.W ⟨ea⟩,Dn 32/16 ⇨ 16r:16q DIVU.L ⟨ea⟩,Dq 32/32 ⇨ 32q DIVU.L ⟨ea⟩,Dr:Dq 64/32 ⇨ 32r:32q DIVUL.L ⟨ea⟩,Dr:Dq 32/32 ⇨ 32r:32q
EOR	Source ⊕ Destination ⇨ Destination	EOR Dn,⟨ea⟩
EORI	Immediate Data ⊕ Destination ⇨ Destination	EORI #⟨data⟩,⟨ea⟩
EORI to CCR	Source ⊕ CCR ⇨ CCR	EORI #⟨data⟩,CCR
EORI to SR	If supervisor state the Source ⊕ SR ⇨ SR else TRAP	EORI #⟨data⟩,SR
EXG	Rx ⇔ Ry	EXG Dx,Dy EXG Ax,Ay EXG Dx,Ay EXG Ay,Dx

Table 4-23. Instruction Set Summary (Sheet 3 of 7)

Opcode	Operation	Syntax
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
FABS	Absolute Value of Source ➧ FPn	FABS.⟨fmt⟩ ⟨ea⟩,FPn FABS.X FPm,FPn FABS.X FPn FrABS.⟨fmt⟩;2 ⟨ea⟩,FPn FrABS.X[2] FPm,FPn FrABS.X[2] FPn
FADD	Source + FPn ➧ FPn	FADD.⟨fmt⟩ ⟨ea⟩,FPn FADD.X FPm,FPn FrADD.⟨fmt⟩[2] ⟨ea⟩,FPn FrADD.X[2] FPm,FPn
FBcc	If conditio true, then PC + d ➧ PC	FBcc.⟨size⟩ ⟨label⟩
FCMP	FPn — Source	FCMP.⟨fmt⟩ ⟨ea⟩,FPn FCMP.X FPm,FPn
FDBcc	If condition true then no operation else Dn − 1 ➧ Dn if Dn ≠ −1 then PC + d ➧ PC else execute next instruction	FDBcc Dn,⟨label⟩
FDIV	FPn (÷) Source ➧ FPn	FDIV.⟨fmt⟩ ⟨ea⟩,FPn FDIV.X FPm,FPn FrDIV.⟨fmt⟩[2] ⟨ea⟩,FPn FrDIV.X[2] FPm,FPn
FMOVE	Source ➧ Destination	FMOVE.⟨fmt⟩ ⟨ea⟩,FPn FMOVE.⟨fmt⟩ FPM,⟨ea⟩ FMOVE.P FPm,⟨ea⟩{Dn} FMOVE.P FPm,⟨ea⟩{#k} FrMOVE.⟨fmt⟩[2] ⟨ea⟩,FPn
FMOVE	Source ➧ Destination	FMOVE.L ⟨ea⟩,FPcr FMOVE.L FPcr,⟨ea⟩
FMOVEM	Register List ➧ Destination Source ➧ Register List	FMOVEM.X ⟨list⟩[3],⟨ea⟩ FMOVEM.X Dn,⟨ea⟩ FMOVEM.X ⟨ea⟩,⟨list⟩[3] FMOVEM.X ⟨ea⟩,Dn
FMOVEM	Register List ➧ Destination Source ➧ Register List	FMOVEM.L ⟨list⟩[4],⟨ea⟩ FMOVEM.L ⟨ea⟩,⟨list⟩[4]
FMUL	Source × FPn ➧ FPn	FMUL.⟨fmt⟩ ⟨ea⟩,FPn FMUL.X FPm,FPn FrMUL⟨fmt⟩[2] ⟨ea⟩,FPn FrMUL.X[2] FPm,FPn
FNEG	−(Source) ➧ FPn	FNEG.⟨fmt⟩ ⟨ea⟩,FPn FNEG.X FPm,FPn FNEG.X FPn FrNEG.⟨fmt⟩[2] ⟨ea⟩,FPn FrNEG.X[2] FPm,FPn FrNEG.X[2] FPn
FNOP	None	FNOP

Table 4-23. Instruction Set Summary (Sheet 4 of 7)

Opcode	Operation	Syntax
FRESTORE	If in supervisor state then FPU State Frame ➧ Internal State else TRAP	FRESTORE ⟨ea⟩
FSAVE	If in supervisor state then FPU Internal State ➧ State Frame else TRAP	FSAVE ⟨ea⟩
FScc	If (condition true) then 1s ➧ Destination else 0s ➧ Destination	FScc.⟨size⟩ ⟨ea⟩
FSQRT	Square Root of Source ➧ FPn	FSQRT.⟨fmt⟩ ⟨ea⟩,FPn FSQRT.X FPm,FPn FSQRT.X FPn FrSQRT.⟨fmt⟩[2] ⟨ea⟩,FPn FrSQRT[2] FPm,FPn FrSQRT[2] FPn
FSUB	FPn – Source ➧ FPn	FSUB.⟨fmt⟩ ⟨ea⟩,FPn FSUB.X FPm,FPn FrSUB.⟨fmt⟩ ⟨ea⟩,FPn FrSUB.X[2] FPm,FPn
FTRAPcc	If condition true, then TRAP	FTRAPcc FTRAPcc.W #⟨data⟩ FTRAPcc.L #⟨data⟩
FTST	Condition Codes for Operand ➧ FPCC	FTST.⟨fmt⟩ ⟨ea⟩ FTST.X FPm
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 to 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⟩

Table 4-23. Instruction Set Summary (Sheet 5 of 7)

Opcode	Operation	Syntax
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
MOVE16	Source block > Destination block	MOVE16 (Ax)+,(Ay)+ MOVE16 xxx.L,(An) MOVE16 xxx.L,(An)+ MOVE16 (An),xxx.L MOVE16 (An)+,xxx.L
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
MOVES	If supervisor state then Rn ➧ Destination [DFC] or Source [SFC] ➧ Rn else TRAP	MOVES Rn,⟨ea⟩ MOVES ⟨ea⟩,Rn
MULS	Source × Destination ➧ Destination	MULS.W ⟨ea⟩,Dn 16 × 16 ➧ 32 MULS.L ⟨ea⟩,Dl 32 × 32 ➧ 32 MULS.L ⟨ea⟩,Dh:Dl 32 × 32 ➧ 64
MULU	Source × Destination ➧ Destination	MULU.W ⟨ea⟩,Dn 16 × 16 ➧ 32 MULU.L ⟨ea⟩,Dl 32 × 32 ➧ 32 MULU.L ⟨ea⟩,Dh:Dl 32 × 32 ➧ 64
NBCD	0 − (Destination$_{10}$) − 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⟩

Table 4-23. Instruction Set Summary (Sheet 6 of 7)

Opcode	Operation	Syntax
PFLUSH	If supervisor state then invalidate instruction and data ATC entries for destination address else TRAP	PFLUSH (An) PFLUSHN (An) PFLUSHA PFLUSHAN
PTEST	If supervisor state then logical address status ⇨ MMUSR; entry ⇨ ATC else TRAP	PTESTR (An) PTESTW (An)
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⟩
RTD	(SP) ⇨ PC; SP+4+d ⇨ SP	RTD #⟨displacement⟩
RTE	If supervisor state the (SP) ⇨ SR; SP+2 ⇨ SP; (SP) ⇨ PC; SP+4 ⇨ SP; restore state and deallocate stack according to (SP) else TRAP	RTE
RTR	(SP) ⇨ CCR; SP+2 ⇨ SP; (SP) ⇨ PC; SP+4 ⇨ SP	RTR
RTS	(SP) ⇨ PC; SP+4 ⇨ SP	RTS
SBCD	$\text{Destination}_{10} - \text{Source}_{10} - X$ ⇨ 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⟩

Table 4-23. Instruction Set Summary (Sheet 7 of 7)

Opcode	Operation	Syntax
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 individual register names separated by a slash (/).
5. where d is direction, L or R.

4.6 INSTRUCTION EXAMPLES

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

4.6.1 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. But 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 counter 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 4-4). 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 4-4, 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.

The next code (see Figure 4-5) 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 lable SDEMPTY if the list is empty. Otherwise, a second LEA instruction loads the address of the NEXT

```
                                    ALLOCATE NEW ENTRY, ADDRESS IN A1
SINSERT     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
```

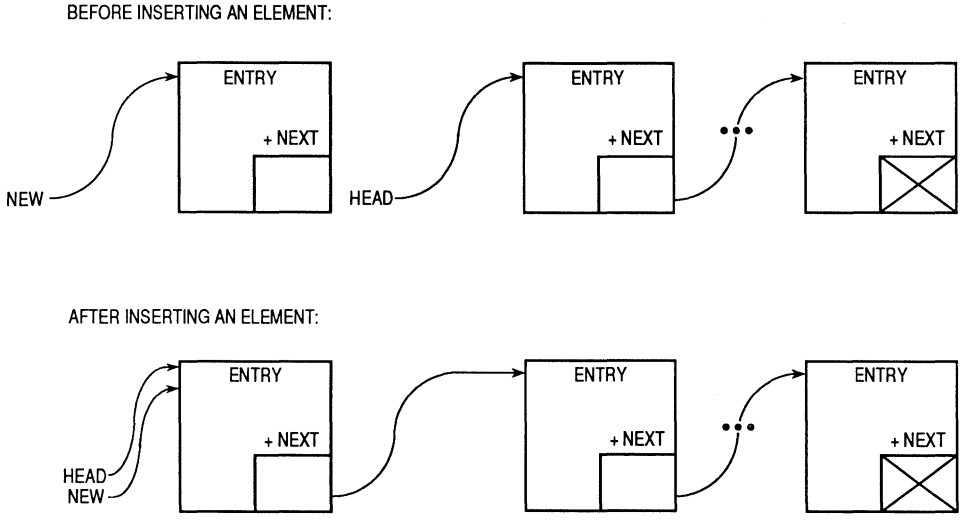


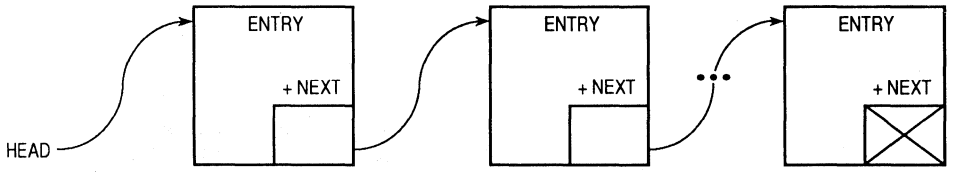
Figure 4-3. Linked List Insertion

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.

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).

```
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
            CAS.2   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)
```


BEFORE DELETING AN ELEMENT:



AFTER DELETING AN ELEMENT:

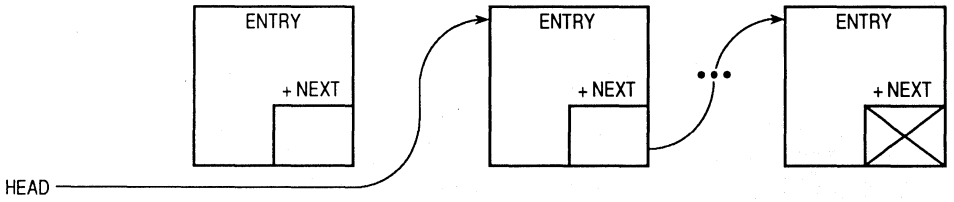


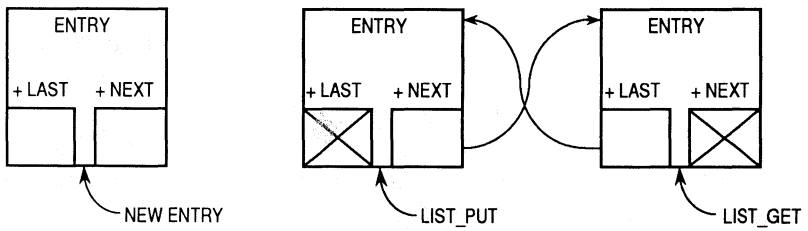
Figure 4-4. Linked List Deletion

The code fragment in Figure 4-6 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.

```
DINSERT
                LEA     LIST_PUT, A0    (ALLOCATE NEW LIST ENTRY, LOAD ADDRESS INTO A2)
                LEA     LIST_GET, A1    LOAD ADDRESS OF HEAD POINTER INTO A0
                MOVE.L  A2, D2          LOAD ADDRESS OF TAIL POINTER INTO A1
                MOVE.L  (A0), D0        LOAD NEW ENTRY POINTER INTO D2
DILOOP          TST.L   D0              LOAD POINTER TO HEAD ENTRY INTO D0
                BEQ     DIEMPTY         IS HEAD POINTER NULL, (0 ENTRIES IN LIST)
                MOVE.L  D0, (NEXT, A2)  IF SO, WE ONLY TO ESTABLIST POINTERS
                CLR.L   D1              PUT HEAD POINTER INTO FORWARD POINTER OF NEW ENTRY
                MOVE.L  D1, (LAST, A2)  PUT NULL POINTER VALUE INTO D1
                LEA     (LAST, D0), A1  PUT NULL POINTER IN BACKWARD POINTER OF NEW ENTRY
                CAS2.L  D0:D1,D2:D2,(A0):(A1)  LOAD BACKWARD POINTER OF OLD HEAD ENTRY INTO A1
                BNE     DILOOP          IF WE STILL POINT TO OLD HEAD ENTRY, UPDATE POINTERS
                BRA     DIDONE          IF NOT, TRY AGAIN
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:

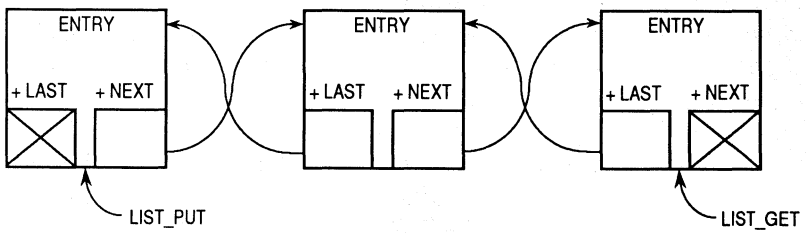


Figure 4-5. Doubly-Linked List Insertion

The code fragment to delete an element from a doubly-linked list is similar (see Figure 4-7). 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.

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.

4.6.2 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.

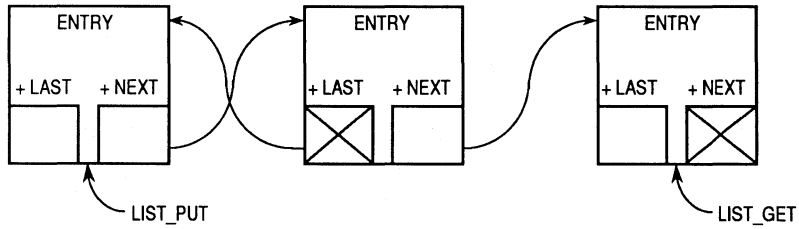
The UNLK instruction removes a stack frame from the end of the list by loading an address into the stack pointer and pulling the value at that address from the stack. When the operand of the instruction is the address of the link address at the bottom of a stack frame, the effect is to remove the stack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

4.6.3 Bit Field Instructions

One of the data types 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 through 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.

```
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     DDONE               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
            CRL.L   D0                  PUT NULL POINTER VALUE IN D0
DDEMPTY     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     DDONE
            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
DDONE                                   SUCCESSFUL ENTRY DELETION, ADDRESS OF DELETED
                                        ENTRYIN D1 (MAY BE NULL)
```


BEFORE DELETING NEW ENTRY:



AFTER DELETING NEW ENTRY:

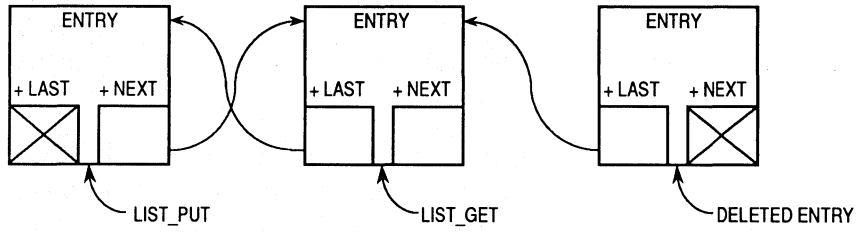


Figure 4-6. Doubly-Linked List Deletion

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. This 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.

4.6.4 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 5
SIGNAL DESCRIPTION

This section contains brief descriptions of the input and output signals in their functional groups (see Figure 5-1). Each signal is explained in a brief paragraph with reference to other sections that contain more detailed information about the signal and the related operations. The names, mnemonics, and signal descriptions of the input and output signals for the MC68040 are listed in Table 5-1. Guaranteed timing specifications for these signals can be found in **SECTION 11 ELECTRICAL CHARACTERISTICS**.

NOTE

Assertion and **negation** are used to specify forcing a signal to a particular state. **Assertion** and **assert** refer to a signal that is active or true. **Negation** and **negate** refer to a signal that is inactive or false. These terms are used independently of the voltage level (high or low) that they represent.

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

Signal Name	Mnemonic	Function
Address Bus	A31–A0	32-bit address bus used to address any of 4 Gbytes.
Data Bus	D31–D0	32-bit data bus used to transfer up to 32 bits of data per bus transfer.
Transfer Type	TT1,TT0	Indicates the general transfer type: normal, MOVE16, alternate logical function code, and acknowledge.
Transfer Modifier	TM2,TM0	Indicates supplemental information about the access.
Transfer Line Number	TLN1,TLN0	Indicates which cache line in a set is being pushed or loaded by the current line transfer.
User Programmable Attributes	UPA1,UPA0	User-defined signals, controlled by the corresponding user attribute bits from the address translation entry.
Read/Write	R/$\overline{W}$	Identifies the transfer as a read or write.
Transfer Size	SIZ1,SIZ0	Indicates the data transfer size. These signals, together with A0 and A1, define the active sections of the data bus.
Bus Lock	$\overline{LOCK}$	Indicates a bus transfer is part of a read-modify-write operation, and that the sequence of transfers should not be interrupted.
Bus Lock End	$\overline{LOCKE}$	Indicates the current transfer is the last in a locked sequence of transfers.

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

Signal Name	Mnemonic	Function
Cache Inhibit Out	$\overline{\text{CIOUT}}$	Indicates the processor will not cache the current bus transfer.
Transfer Start	$\overline{\text{TS}}$	Indicates the beginning of a bus transfer.
Transfer in Progress	$\overline{\text{TIP}}$	Asserted for the duration of a bus transfer.
Transfer Acknowledge	$\overline{\text{TA}}$	Asserted to acknowledge a bus transfer.
Transfer Error Acknowledge	$\overline{\text{TEA}}$	Indicates an error condition exists for a bus transfer.
Transfer Cache Inhibit	$\overline{\text{TCI}}$	Indicates the current bus transfer should not be cached.
Transfer Burst Inhibit	$\overline{\text{TBI}}$	Indicates the slave cannot handle a line burst access.
Data Latch Enable	DLE	Alternate clock input used to latch input data when the processor is operating in DLE mode.
Snoop Control	SC1,SC0	Indicates the snooping operation required during an alternate master access.
Memory Inhibit	$\overline{\text{MI}}$	Inhibits memory devices from responding to an alternate master access during snooping operations.
Bus Request	$\overline{\text{BR}}$	Asserted by the processor to request bus mastership.
Bus Grant	$\overline{\text{BG}}$	Asserted by an arbiter to grant bus mastership to the processor.
Bus Busy	$\overline{\text{BB}}$	Asserted by the current bus master to indicate it has assumed ownership of the bus.
Cache Disable	$\overline{\text{CDIS}}$	Dynamically disables the internal caches to assist emulator support.
MMU Disable	$\overline{\text{MDIS}}$	Disables the translation mechanism of the MMUs.
Reset In	$\overline{\text{RSTI}}$	Processor reset.
Reset Out	$\overline{\text{RSTO}}$	Asserted during execution of a RESET instruction to reset external devices.
Interrupt Priority Level	$\overline{\text{IPL2}}$–$\overline{\text{IPL0}}$	Provides an encoded interrupt level to the processor.
Interrupt Pending	$\overline{\text{IPEND}}$	Indicates an interrupt is pending.
Autovector	$\overline{\text{AVEC}}$	Used during an interrupt acknowledge transfer to request internal generation of the vector number.
Processor Status	PST3–PST0	Indicates internal processor status.
Bus Clock	BCLK	Clock input used to derive all bus signal timing.
Processor Clock	PCLK	Clock input used for internal logic timing. The PCLK frequency is exactly 2X the BCLK frequency.
Test Clock	TCK	Clock signal for the IEEE P1149.1 Test Access Port (TAP).
Test Mode Select	TMS	Selects the principle operations of the test-support circuitry.
Test Data Input	TDI	Serial data input for the TAP.
Test Data Output	TDO	Serial data output for the TAP.
Test Reset	$\overline{\text{TRST}}$	Provides an asynchronous reset of the TAP controller.
Power Supply	V_{CC}	Power supply.
Ground	GND	Ground connection.

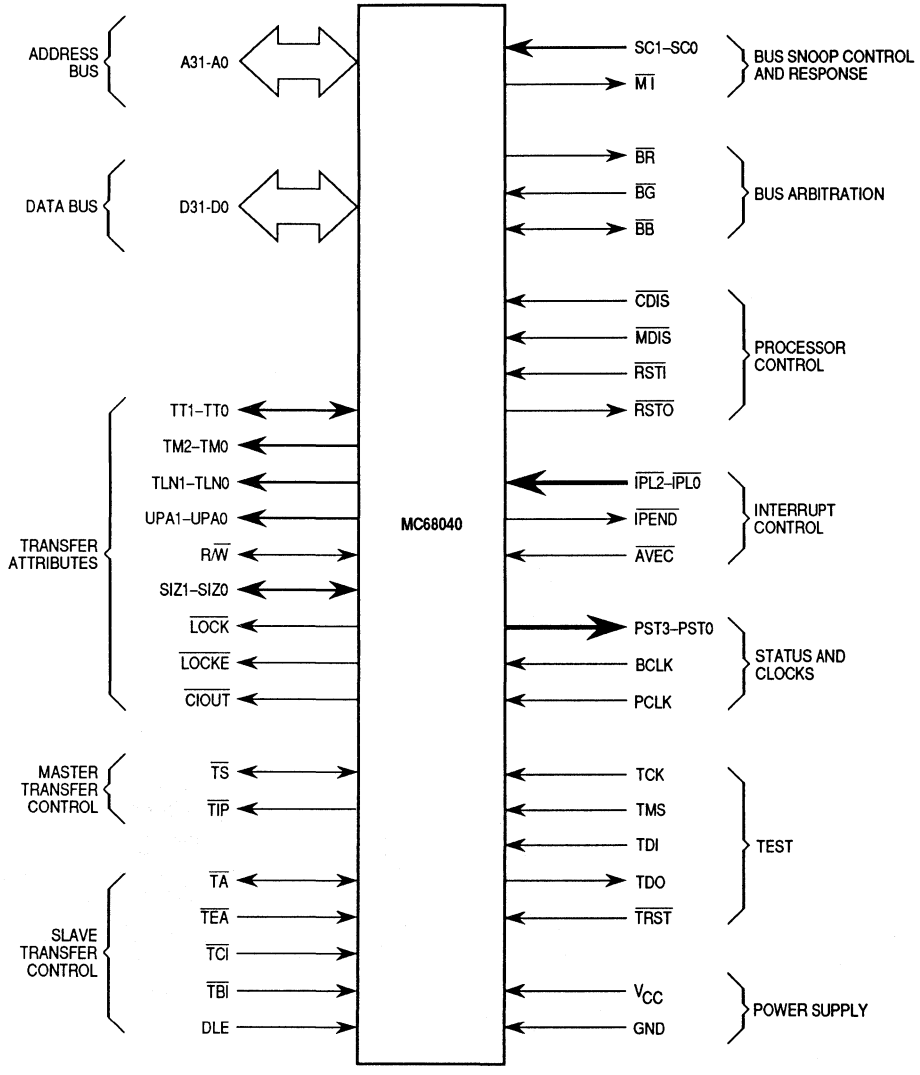


Figure 5-1. Functional Signal Groups

5.1 ADDRESS BUS (A31–A0)

These three-state bidirectional signals provide the address of the first item of a bus transfer (except for acknowledge transfers) when the MC68040 is the bus master. When an alternate master is controlling the bus, these signals are examined by the processor (snooped) to determine whether the processor should intervene in the access to maintain cache coherency.

A multiplexed bus mode is selectable during processor reset (by the level on $\overline{\text{CDIS}}$) which allows the address bus and data bus to be physically tied together for multiplexed bus applications. Refer to **SECTION 8 BUS OPERATION** for detailed information about the relationship of the address bus to bus operation, and the multiplexed bus mode.

5.2 DATA BUS (D31–D0)

These three-state bidirectional signals provide the general-purpose data path between the MC68040 and all other devices. The data bus can transfer 8, 16, or 32 bits of data per bus transfer. During a burst transfer, the data lines are time-multiplexed to carry all 128 bits of the burst request using four 32-bit transfers.

A multiplexed bus mode is selectable during processor reset (by the level on $\overline{\text{CDIS}}$) which allows the data bus and address bus to be physically tied together for multiplexed bus applications. A data latch mode is also selectable during processor reset (by the level on $\overline{\text{MDIS}}$) which allows the memory interface to specify when the processor should latch input data via the DLE signal. Refer to **SECTION 8 BUS OPERATION** for detailed information about the relationship of the data bus to bus operation, the multiplexed bus mode, and the data latch mode.

5.3 TRANSFER ATTRIBUTE SIGNALS

The following paragraphs describe the transfer attribute signals, which provide additional information about the bus transfer. Refer to **SECTION 8 BUS OPERATION** for detailed information about the relationship of the transfer attribute signals to bus operation.

5.3.1 Transfer Type (TT1,TT0)

These three-state bidirectional signals are driven by the processor to indicate the type of access for the current bus transfer. During bus transfers by an alternate master, these signals are sampled to determine if the processor should snoop the transfer; only normal and MOVE16 accesses can be snooped. The acknowledge access is used for both interrupt and breakpoint acknowledge transfers. Table 5-2 shows the definition of the transfer-type encodings.

Table 5-2. Transfer-Type Encoding

TT1	TT0	Transfer Type
0	0	Normal Access
0	1	MOVE16 Access
1	0	Alternate Logical Function Code Access
1	1	Acknowledge Access

5.3.2 Transfer Modifier (TM2–TM0)

These three-state output signals provide supplemental information for each transfer type. Table 5-3 shows the encodings for normal and MOVE16 transfers, and Table 5-4 shows the the encodings for alternate access transfers. For interrupt acknowledge transfers the TMx signals carry the interrupt level being acknowledged, and for breakpoint acknowledge transfers the TMx signals are low. When the MC68040 is not the bus master, the TMx signals are set to high impedance.

Table 5-3. Normal and MOVE16 Access TM Encoding

TM2	TM1	TM0	Transfer Modifier
0	0	0	Data Cache Push Access
0	0	1	User Data Access*
0	1	0	User Code Access
0	1	1	MMU Table Search Data Access
1	0	0	MMU Table Search Code Access
1	0	1	Supervisor Data Access*
1	1	0	Supervisor Code Access
1	1	1	Reserved

*MOVE16 accesses only use these encodings

Table 5-4. Alternate Access TM Encoding

TM2	TM1	TM0	Transfer Modifier
0	0	0	Logical Function Code 0
0	0	1	Reserved
0	1	0	Reserved
0	1	1	Logical Function Code 3
1	0	0	Logical Function Code 4
1	0	1	Reserved
1	1	0	Reserved
1	1	1	Logical Function Code 7

5.3.3 Transfer Line Number (TLN1,TLN0)

For normal push accesses and normal line read accesses, these three-state outputs indicate which line in the set of four instruction or data cache lines is being accessed. These signals are undefined for all other accesses, and are placed in a high-impedance state when the processor relinquishes the bus. Table 5-5 shows the definition of the encodings.

Table 5-5. TLN Encoding

TLN1	TLN0	Line
0	0	Zero
0	1	One
1	0	Two
1	1	Three

The TLN signals can be used in high-performance systems to build an external snoop filter with a duplicate set of cache tags. The TLN signals and address bus provide a direct indication of the state of the caches and can be used to help maintain the duplicate tag store.

5.3.4 User Programmable Attributes (UPA1,UPA0)

The UPA signals are three-state outputs whose levels are determined by the user programmable attribute bits in the address translation entry or transparent translation register that matches the logical address. These signals are defined only for normal code and data accesses, and for MOVE16 accesses. For all other accesses, including table search and cache line push accesses which may result from a normal access, the UPA signals are zero.

When the MC68040 is not the bus master, these signals are set to a high impedance state.

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

This bidirectional three-state signal defines the data transfer direction for the current bus cycle. A high level indicates a read cycle and a low level indicates a write cycle. This signal is examined by the bus snoop controller when the processor is not the bus master.

5.3.6 Transfer Size (SIZ1,SIZ0)

These bidirectional three-state signals indicate the data size for the bus transfer. Table 5-6 shows the definition of the bus request size encodings. These signals are examined by the bus snoop controller when the processor is not the bus master.

Table 5-6. Transfer Size Encoding

SIZ1	SIZ0	Requested Size
0	0	Long Word (4 Bytes)
0	1	Byte
1	0	Word (2 Bytes)
1	1	Line (16 Bytes)

5.3.7 Lock ($\overline{LOCK}$)

This three-state output indicates that the current transfer is part of a locked sequence of transfers for a read-modify-write operation. The external arbiter can use the $\overline{LOCK}$ signal to prevent an alternate master from gaining control of the bus and accessing the same operand between processor accesses for the locked sequence of transfers. Although the $\overline{LOCK}$ bus signal indicates the processor requests that the bus be locked, the processor will give up the bus if the bus grant ($\overline{BG}$) signal is negated by the external arbiter. When the MC68040 is not the bus master, the $\overline{LOCK}$ signal is set to a high impedance state.

5.3.8 Lock End ($\overline{LOCKE}$)

This three-state output indicates the current transfer is the last in a locked sequence of transfers for a read-modify-write operation. The external arbiter

can use $\overline{\text{LOCKE}}$ to support arbitration between unrelated locked transfer sequences while still maintaining the indivisible nature of each read-modify-write operation. When the MC68040 is not the bus master, the $\overline{\text{LOCKE}}$ signal is set to a high impedance state.

5.3.9 Cache Inhibit Out ($\overline{\text{CIOUT}}$)

This three-state output signal reflects the state of the CM field in the address translation cache entry logical address and is asserted for accesses to non-cachable pages to indicate that an external cache should ignore the bus transfer. When the referenced logical address is within an area specified for transparent translation, the cache modify (CM) field of the appropriate transparent translation register controls the state of $\overline{\text{CIOUT}}$. Refer to **SECTION 6 MEMORY MANAGEMENT** for more information about the address translation caches and transparent translation. When the MC68040 is not the bus master, the $\overline{\text{CIOUT}}$ signal is set to a high impedance state.

5.4 BUS TRANSFER CONTROL SIGNALS

The following signals provide control functions for bus transfers. Refer to **SECTION 8 BUS OPERATION** for detailed information about the relationship of the bus transfer control signals to bus operation.

5.4.1 Transfer Start ($\overline{\text{TS}}$)

This three-state bidirectional signal is asserted by the MC68040 for one clock period to indicate the start of each transfer. During alternate master accesses, this signal is monitored by the MC68040 to detect the start of each transfer to be snooped.

5.4.2 Transfer in Progress ($\overline{\text{TIP}}$)

This three-state output is asserted to indicate a bus transfer is in progress, and is negated during idle bus cycles if the bus is still granted to the processor. When the processor loses bus mastership, $\overline{\text{TIP}}$ negates after completion of the current transfer and then transitions to a high-impedance state.

5.4.3 Transfer Acknowledge ($\overline{\text{TA}}$)

This three-state bidirectional signal indicates the completion of a requested data transfer operation. During transfers by the MC68040, $\overline{\text{TA}}$ is an input

signal from the referenced slave device indicating the completion of the transfer. During alternate master accesses, $\overline{TA}$ is normally three-stated to allow the referenced slave device to respond, and is sampled by the MC68040 to detect the completion of each bus transfer. For alternate master accesses which reference modified (dirty) data in the MC68040's caches, the MC68040 can inhibit memory and intervene in the access to source or sink data in its internal caches, at which time $\overline{TA}$ is asserted to acknowledge the data transfer.

5.4.4 Transfer Error Acknowledge ($\overline{TEA}$)

This input signal is asserted by the current slave to indicate an error condition for the bus transaction. When asserted with $\overline{TA}$, this signal indicates that the processor should retry the access. During alternate master accesses, $\overline{TEA}$ is sampled by the MC68040 to detect the completion of each bus transfer.

5.4.5 Transfer Cache Inhibit ($\overline{TCI}$)

This input signal inhibits read data from being loaded into the MC68040 instruction or data caches. $\overline{TCI}$ is ignored during all writes and after the first data transfer for both burst line reads and burst-inhibited line reads. $\overline{TCI}$ is also ignored during all alternate bus master transfers.

5.4.6 Transfer Burst Inhibit ($\overline{TBI}$)

This input signal indicates to the processor that the accessed device can not support burst mode accesses, and that the requested line transfer should be broken up into individual long word transfers. If the first data transfer of a line access is terminated by asserting $\overline{TBI}$ with $\overline{TA}$, the processor terminates the burst and accesses the remaining data for the line as three successive long word transfers. During alternate master accesses, $\overline{TBI}$ is sampled by the MC68040 to detect the completion of each bus transfer.

5.4.7 Data Latch Enable (DLE)

This input signal is used in DLE mode to latch the input data bus on read transfers. DLE mode can be used to support asynchronous memory interfaces by allowing the interface to specify when data should be latched, instead of requiring data to be valid on the rising edge of BCLK.

5.5 SNOOP CONTROL SIGNALS

The following group of signals control the operation of the MC68040 on-chip snooping logic. Refer to **SECTION 7 INSTRUCTION AND DATA CACHES** for information about the relationship of the snoop control signals to the caches, and to **SECTION 8 BUS OPERATION** for detailed information about the relationship of the snoop control signals to bus operation.

5.5.1 Snoop Control (SC1,SC0)

These inputs signals specify the snoop operation to be performed by the MC68040 for an alternate master bus transfer. If the MC68040 is allowed to snoop an alternate master read transfer, it can intervene in the access to supply data from its data cache when the memory copy is stale, ensuring the alternate master receives valid data. Writes by an alternate master can also be snooped to either update the MC68040's internal data cache with the new data or invalidate the matching cache lines, ensuring subsequent reads by the MC68040 access valid data. Table 5-7 shows the general operation requested for each snoop control encoding. These signals are ignored when the processor is the bus master.

Table 5-7. Snoop Control Encoding

SC1	SC0	Requested Snoop Operation	
		Read Access	Write Access
0	0	Inhibit Snooping	Inhibit Snooping
0	1	Supply Dirty Data and Leave Dirty	Sink Byte/Word/Long-Word Data
1	0	Supply Dirty Data and Mark Line Invalid	Invalidate Line
1	1	Reserved (Snoop Inhibited)	Reserved (Snoop Inhibited)

5.5.2 Memory Inhibit ($\overline{MI}$)

This output signal inhibits memory from responding to an alternate master access when the MC68040 is snooping the access. When the snoop control signals indicate an access should be snooped, the MC68040 keeps $\overline{MI}$ asserted until it determines whether intervention in the access is required. If no intervention is required, $\overline{MI}$ is negated and memory is allowed to respond and complete the access; otherwise, $\overline{MI}$ remains asserted and the MC68040 completes the transfer as a slave, thereby updating its caches on a write or supplying data to the alternate master on a read. When the MC68040 is the bus master, $\overline{MI}$ is negated.

5.6 ARBITRATION SIGNALS

The following control signals support requests to an external arbiter for bus mastership. Refer to **SECTION 8 BUS OPERATION** for detailed information about the relationship of the arbitration signals to bus operation.

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

This output signal indicates to the external arbiter that the processor needs to become bus master for one or more bus transfers. $\overline{BR}$ is negated once the MC68040 begins an access to the external bus with no other accesses pending, and remains negated until another access is required.

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

This input signal from an external arbiter indicates the bus is available to the MC68040 as soon as the current bus access completes. The MC68040 must sample $\overline{BG}$ asserted and $\overline{BB}$ negated (indicating the bus is free) before it assumes ownership of the bus.

5.6.3 Bus Busy ($\overline{BB}$)

This three-state bidirectional signal indicates the bus is currently owned. $\overline{BB}$ is monitored as a processor input to determine when a prior bus master has released control of the bus. The MC68040 must sample $\overline{BG}$ asserted and $\overline{BB}$ negated (indicating the bus is free) before it asserts $\overline{BB}$ as an output to assume ownership of the bus. $\overline{BB}$ remains asserted by the processor until the external arbiter negates $\overline{BG}$ and the processor completes the bus transfer in progress. When releasing the bus the processor negates $\overline{BB}$, then sets it to a high impedance state for use again as an input.

5.7 PROCESSOR CONTROL SIGNALS

The following signals control disabling of the caches and memory management units (MMUs), and support processor and external device initialization.

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

The cache disable signal dynamically disables the on-chip caches on the next internal cache access boundary. $\overline{\text{CDIS}}$ does not flush the data and instruction caches; entries remain unaltered and become available again after $\overline{\text{CDIS}}$ is negated. Snooping is also unaffected by the assertion of $\overline{\text{CDIS}}$. During a processor reset the level on $\overline{\text{CDIS}}$ is latched and used to select the normal bus mode ($\overline{\text{CDIS}}$ high) or multiplexed bus mode ($\overline{\text{CDIS}}$ low). Refer to **SECTION 7 INSTRUCTION AND DATA CACHES** for information about the caches and to **SECTION 8 BUS OPERATION** for information about the multiplexed bus mode. Refer to MC68040DH/D, *MC68040 Design Handbook* for descriptions of the use of this signal by an emulator.

5

5.7.2 MMU Disable ($\overline{\text{MDIS}}$)

The MMU disable signal dynamically disables the translation of addresses by the MMUs. The assertion of $\overline{\text{MDIS}}$ does not flush the address translation (ATC) caches; ATC entries become available again when $\overline{\text{MDIS}}$ is negated. During a processor reset the level on $\overline{\text{MDIS}}$ is latched and used to select the normal data latch mode ($\overline{\text{MDIS}}$ high) or data latch enable (DLE) mode ($\overline{\text{MDIS}}$ low). Refer to **SECTION 6 MEMORY MANAGEMENT** for a description of address translation, **SECTION 8 BUS OPERATION** for information about DLE mode. Refer to MC68040DH/D, *MC68040 Design Handbook* for a description of the use of this signal by an emulator.

5.7.3 Reset In ($\overline{\text{RSTI}}$)

This input signal causes the MC68040 to enter reset exception processing. The $\overline{\text{RSTI}}$ signal is an asynchronous input that is internally synchronized to the next rising edge of the BCLK signal. All three-state signals are set to the high-impedance state, and all other outputs are negated when $\overline{\text{RSTI}}$ is recognized. The test pins are not affected by the assertion of $\overline{\text{RSTI}}$. Refer to **SECTION 8 BUS OPERATION** for a description of reset bus operation and to **SECTION 9 EXCEPTION PROCESSING** for information about the reset exception.

5.7.4 Reset Out ($\overline{\text{RSTO}}$)

This output is asserted by the MC68040 during execution of the RESET instruction to initialize external devices. Refer to **SECTION 8 BUS OPERATION** for a description of reset out bus operation.

5.8 INTERRUPT CONTROL SIGNALS

The following signals control the interrupt functions of the MC68040.

5.8.1 Interrupt Priority Level ($\overline{IPL2}$–$\overline{IPL0}$)

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 $\overline{IPLn}$ signals are active low, $\overline{IPL2}$–$\overline{IPL0}$ equal to $5 corresponds to an interrupt request at interrupt level 2.

During a processor reset the levels on the $\overline{IPLn}$ lines are latched and used to select the output driver characteristics for three signal groups, shown in Table 5-8. Refer to **SECTION 9 EXCEPTION PROCESSING** for information on MC68040 interrupts and to MC68040DH/D, *MC68040 Design Handbook* for information on the driver characteristics.

Table 5-8. Output Driver Control Groups

Signal	Output Buffers Controlled
$\overline{IPL2}$	Data Bus: D31–D0
$\overline{IPL1}$	Address Bus and Transfer Attributes: A31–A0, $\overline{CIOUT}$, $\overline{LOCK}$, $\overline{LOCKE}$, R/$\overline{W}$, SIZ1–SIZ0, TLN1–TLN0, TM2–TM0, TT1–TT0, UPA1–UPA0
$\overline{IPL0}$	Miscellaneous Control Signals: $\overline{BB}$, $\overline{BR}$, $\overline{IPEND}$, $\overline{MI}$, $\overline{PST3}$–$\overline{PST0}$, $\overline{RSTO}$, $\overline{TA}$, $\overline{TIP}$, $\overline{TS}$

NOTE:
High input level = small buffers enabled, low = large buffers enabled.

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

This output signal indicates 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 (other bus masters, for example) to predict processor operation on the following instruction boundaries. Refer to **SECTION 9 EXCEPTION PROCESSING** for interrupt information, and to **SECTION 8 BUS OPERATION** for bus information related to interrupts.

5.8.3 Autovector ($\overline{\text{AVEC}}$)

This input signal is asserted with $\overline{\text{TA}}$ during an interrupt acknowledge transfer to request internal generation of the vector number. Refer to **SECTION 8 BUS OPERATION** for more information about automatic vectors.

5.9 STATUS AND CLOCK SIGNALS

The following paragraphs explain the signals that provide timing, test control, and the internal status of the processor.

5

5.9.1 Processor Status (PST3–PST0)

These outputs indicate the internal execution unit status of the MC68040. The timing is synchronous with BCLK, and the status may have nothing to do with the current bus transfer. Table 5-9 shows the definition of the encodings. Refer to MC68040DH/D, *MC68040 Design Handbook* for a description of the use of these signals by an emulator.

Table 5-9. Processor Status Encoding

PST3	PST2	PST1	PST0	Internal Status
0	0	0	0	User Start/Continue Current Instruction
0	0	0	1	User End Current Instruction
0	0	1	0	User Branch Not Taken and End Current Instruction
0	0	1	1	User Branch Taken and End Current Instruction
0	1	0	0	User Table Search
0	1	0	1	Halted State (Double-Bus Fault)
0	1	1	0	Reserved
0	1	1	1	Reserved
1	0	0	0	Supervisor Start/Continue Current Instruction
1	0	0	1	Supervisor End Current Instruction
1	0	1	0	Supervisor Branch Not Taken and End Current Instruction
1	0	1	1	Supervisor Branch Taken and End Current Instruction
1	1	0	0	Supervisor Table Search
1	1	0	1	Stopped State (Supervisor Instruction)
1	1	1	0	RTE Executed
1	1	1	1	Exception Stacking

5.9.2 Bus Clock (BCLK)

The bus clock input is used as a reference for all bus timing. It is a TTL compatible signal and cannot be gated off. Refer to MC68040DH/D, *MC68040 Design Handbook* for suggestions on clock generation and to **SECTION 11 ELECTRICAL SPECIFICATIONS** for electrical specifications.

5.9.3 Processor Clock (PCLK)

The processor clock input is used to derive all internal timing. This clock is also TTL compatible, and cannot be gated off. Refer to MC68040DH/D, *MC68040 Design Handbook* for suggestions on clock generation and to **SECTION 11 ELECTRICAL SPECIFICATIONS** for electrical specifications.

5.10 TEST SIGNALS

The five test signals provide an interface that supports the IEEE P1149.1 *Test Access Port (TAP) for Boundary Scan Testing of Board Interconnects*. Refer to MC68040DH/D, *MC68040 Design Handbook* for a description of the use of these signals for board level testing.

5.10.1 Test Clock (TCK)

This input signal is used as a dedicated clock for the test logic. Since clocking of the test logic is independent of the normal operation of the MC68040, several other components on a board can share a common test clock with the processor even though each component may operate from a different system clock. The design of the test logic allows the test clock to run at low frequencies, or to be gated off entirely as required for test purposes.

5.10.2 Test Mode Select (TMS)

This input signal is decoded by the TAP controller and distinguishes the principle operations of the test-support circuitry.

5.10.3 Test Data In (TDI)

This input signal provides a serial data input to the TAP.

5.10.4 Test Data Out (TDO)

This three-state output signal provides a serial data output from the TAP. The TDO output can be placed in a high-impedance mode to allow parallel connection of board-level test data paths.

5.10.5 Test Reset ($\overline{\text{TRST}}$)

This input signal provides an asynchronous reset of the TAP controller.

5.11 POWER SUPPLY CONNECTIONS

The MC68040 requires connection to a V_{CC} power supply, positive with respect to ground. The V_{CC} and ground connections are grouped to supply adequate current to the various sections of the processor. **SECTION 12 ORDERING INFORMATION AND MECHANCIAL DATA** describes the groupings of V_{CC} and ground connections, and MC68040DH/D, *MC68040 Design Handbook* describes a typical power supply interface.

5.12 SIGNAL SUMMARY

Table 5-10 provides a summary of the electrical characteristics of the signals discussed in this section.

Table 5-10. Signal Summary (Sheet 1 of 2)

Signal Function	Signal Name	Type	Active	Three-State
Address Bus	A31–A0	Input/Output	High	Yes
Autovector	$\overline{\text{AVEC}}$	Input	Low	—
Bus Busy	$\overline{\text{BB}}$	Input/Output	Low	Yes
Bus Clock	BCLK	Input	—	—
Bus Grant	$\overline{\text{BG}}$	Input	Low	—
Bus Request	$\overline{\text{BR}}$	Output	Low	No
Cache Disable	$\overline{\text{CDIS}}$	Input	Low	—
Cache Inhibit	$\overline{\text{CIOUT}}$	Output	Low	Yes
Data Bus	D31–D0	Input/Output	High	Yes
Data Latch Enable	DLE	Input	High	—
Ground	GND	Input	—	—
Interrupt Pending	$\overline{\text{IPEND}}$	Output	Low	No
Interrupt Priority Level	$\overline{\text{IPL2}}$–$\overline{\text{IPL0}}$	Input	Low	—
Lock	$\overline{\text{LOCK}}$	Output	Low	Yes

Table 5-10. Signal Summary (Sheet 2 of 2)

Signal Function	Signal Name	Type	Active	Three-State
Lock End	$\overline{\text{LOCKE}}$	Output	Low	Yes
Memory Inhibit	$\overline{\text{MI}}$	Output	Low	No
MMU Disable	$\overline{\text{MDIS}}$	Input	Low	—
Processor Clock	PCLK	Input	—	—
Processor Status	PST3–PST0	Output	High	No
Read/Write	R/$\overline{\text{W}}$	Input/Output	High/Low	Yes
Reset In	$\overline{\text{RSTI}}$	Input	Low	—
Reset Out	$\overline{\text{RSTO}}$	Output	Low	No
Snoop Control	SC1,SC0	Input	High	—
Transfer Acknowledge	$\overline{\text{TA}}$	Input/Output	Low	Yes
Transfer Burst Inhibit	$\overline{\text{TBI}}$	Input	Low	—
Transfer Cache Inhibit	$\overline{\text{TCI}}$	Input	Low	—
Transfer Error Acknowledge	$\overline{\text{TEA}}$	Input	Low	—
Transfer In Progress	$\overline{\text{TIP}}$	Output	Low	Yes
Transfer Line Number	TLN1,TLN0	Output	High	Yes
Transfer Modifier	TM2–TM0	Output	High	Yes
Transfer Size	SIZ1,SIZ0	Input/Output	High	Yes
Transfer Start	$\overline{\text{TS}}$	Input/Output	Low	Yes
Transfer Type	TT1,TT0	Input/Output	High	Yes
Test Clock	TCK	Input	—	—
Test Data In	TDI	Input	High	—
Test Data Out	TDO	Output	High	Yes
Test Mode Select	TMS	Input	High	—
Test Reset	$\overline{\text{TRST}}$	Input	Low	—
User Programmable Attributes	UPA1,UPA0	Output	High	Yes
Power Supply	V_{CC}	Input	—	—

SECTION 6
MEMORY MANAGEMENT

The MC68040 includes independent instruction and data memory management units (MMUs) that support a demand-paged virtual memory environment. The memory management is "demand" in that programs do not specify required memory areas in advance but request them by accessing logical addresses. The physical memory is paged, meaning that it is divided into blocks of equal size, called page frames. The logical address space is divided into pages of the same size. The operating system assigns pages to page frames as they are required to meet the needs of programs.

The principle function of the MMUs is the translation of logical addresses to physical addresses using translation tables stored in memory. Each MMU contains an address translation cache (ATC) in which recently used logical-to-physical address translations are stored. As each MMU receives a logical address from the integer unit, it searches its ATC for the corresponding physical address. When the translation is not in the ATC, the processor searches the translation tables in memory for the translation information. The address calculations and bus cycles required for this search are performed by microcode and dedicated logic in the MC68040. In addition, each MMU contains two transparent translation registers that identify blocks of memory that can be accessed without translation. The MMUs include the following features:

- Independent Instruction and Data MMUs
- 32-Bit Logical Address Translated to 32-Bit Physical Address
- User-Defined 2-Bit Physical Address Extension
- Addresses Translated in Parallel with Indexing into Data or Instruction Cache
- 64-Entry Four-Way Set-Associative ATC for Each MMU (128 Total Entries)
- Global Bit Allows Flushing of All Nonglobal Entries from ATCs
- Selectable 4K or 8K Page Size
- Separate Supervisor and User Translation Trees Supported

- Two Independent Blocks for Each MMU Can Be Defined as Transparent (Untranslated)
- Three-Level Translation Tables with Optional Indirection
- Supervisor and Write Protections
- History Bits Automatically Maintained in Descriptors
- External Translation Disable Input Signal ($\overline{\text{MDIS}}$) for Emulator Support
- Caching Mode Selected on Page Basis

The MMUs completely overlap address translation time with other processing activity when the translation is resident in one of the ATCs. ATC accesses operate in parallel with indexing into the on-chip instruction and data caches.

The instruction memory unit (which supports instruction prefetches) and the data memory unit (which supports all other accesses) each contain an MMU to allow translation of the logical address used to access the memory unit (see Figure 6-1). Each MMU consists of control logic and an ATC that stores current translations. For an instruction or operand access, the corresponding MMU uses the upper logical address bits to check for a physical address in the ATC; the lower address bits are used by the cache controller to index into the cache. If the translation is available, the MMU provides the physical address to the cache controller, which determines if the data being accessed is cached. An external bus cycle is performed only when explicitly requested by the cache controller.

The MMU $\overline{\text{MDIS}}$ signal dynamically disables address translation for emulation and diagnostic support.

The programming model of the MMUs (see Figure 6-2) consists of two root pointer registers, four transparent translation registers, a status register, and a control register. These registers can only be accessed by supervisor programs. The user and supervisor root pointer registers point to address translation tree structures in memory that describe the logical-to-physical mapping for user and supervisor accesses, respectively. These pointers can also point to a common tree structure to support a merged supervisor and user address space. Each transparent translation register can define a block of logical addresses that are used as physical addresses without translation. The MMU status register (MMUSR) contains accumulated status information from a translation performed as a part of a PTEST instruction. The translation control (TC) register contains two bits: one bit enables/disables page address translation (independent of transparent translation); the other bit selects page size.

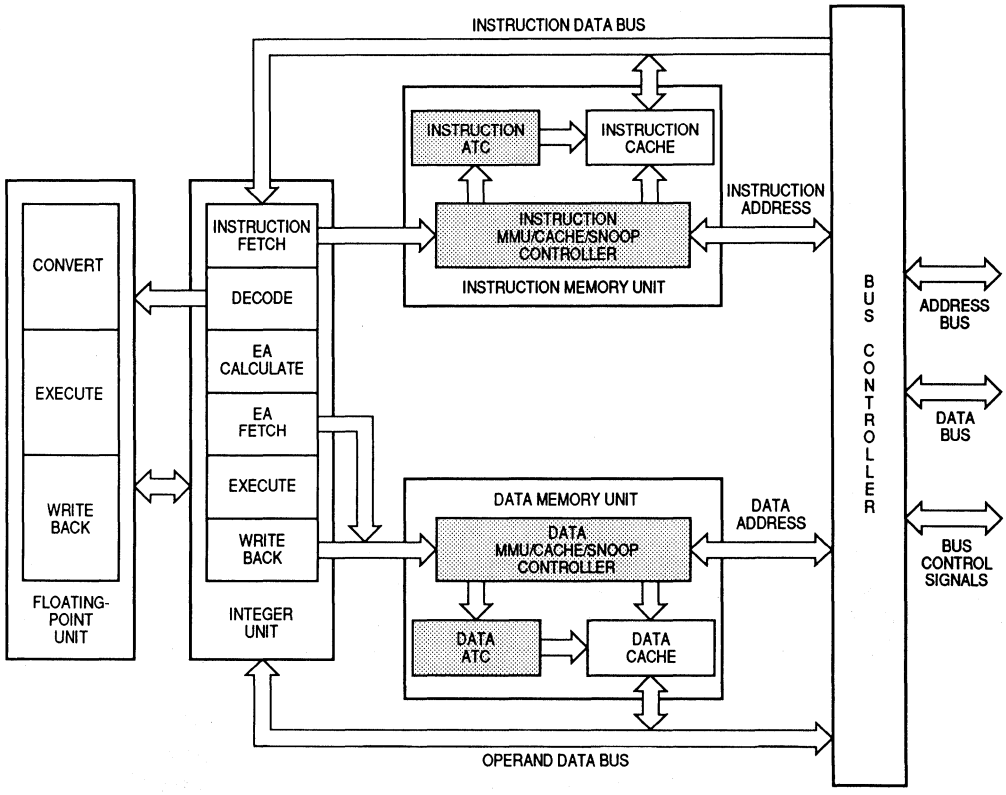


Figure 6-1. Memory Management Unit

ATCs in the MMUs are four-way set-associative caches that each store 64 logical-to-physical address translations and associated page information. For each access to a memory unit, the MMU uses the lower logical address bits to index into the ATC and compares the upper address bits and privilege mode (supervisor or user) with the tag for each of the four lines in the set. When the access address and privilege mode matches a tag in the set (a hit occurs) and no access violation is detected, the ATC outputs the corresponding physical address to the cache controller, which accesses the data within the cache and/or requests an external bus cycle. Each ATC entry contains a logical address, a physical address, and status bits. Among the status bits are the write-protect and cache-inhibit bits.

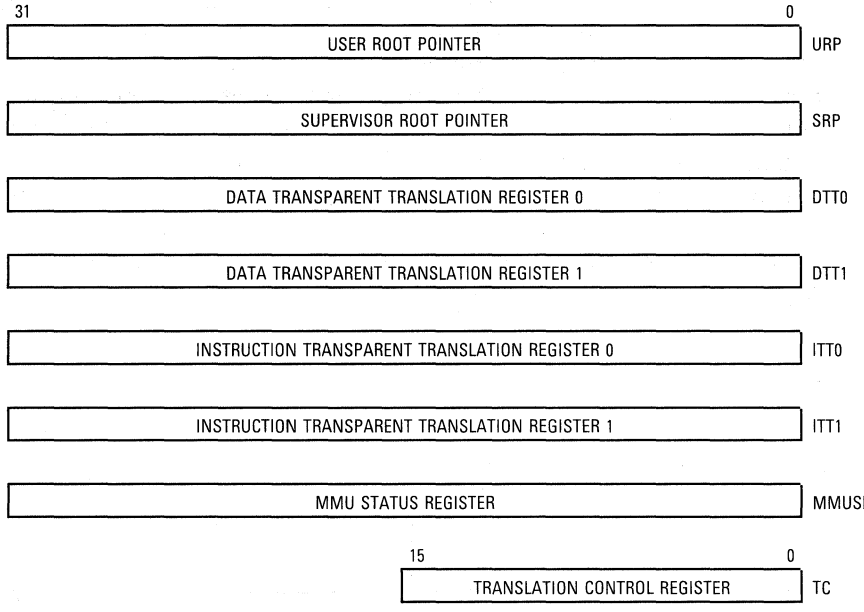


Figure 6-2. MMU Programming Model

When the ATC does not contain the translation for a logical address (a miss occurs), the MMU aborts the current access and searches the translation tables in memory for the correct translation. If the table search completes without any errors, the MMU stores the translation in the ATC and provides the physical address for the access, allowing the memory unit to retry the original access.

6.1 TRANSLATION TABLE STRUCTURE

The MC68040 uses the ATCs in the instruction and data memory units with translation tables stored in memory to perform the translations from logical to physical addresses. Translation tables for a program are loaded into memory by the operating system. Since the instruction and data MMUs access the same translation table for a specific privilege mode (user or supervisor), no distinction is made in the translation of instruction accesses versus data accesses. This lack of distinction results in a merged instruction and data address space.

The general translation table structure supported by the MC68040 is a three-level tree structure (see Figure 6-3). The pointer tables contain the base addresses of the tables at the next level. The page tables contain either the physical address for the translation or a pointer to the memory location containing the address. Only a portion of the translation table for the entire logical address space is required to be resident in memory at any time: specifically, only the portion of the table that translates the logical addresses of the currently executing process must be resident. Portions of translation tables can be dynamically allocated as the process requires additional memory.

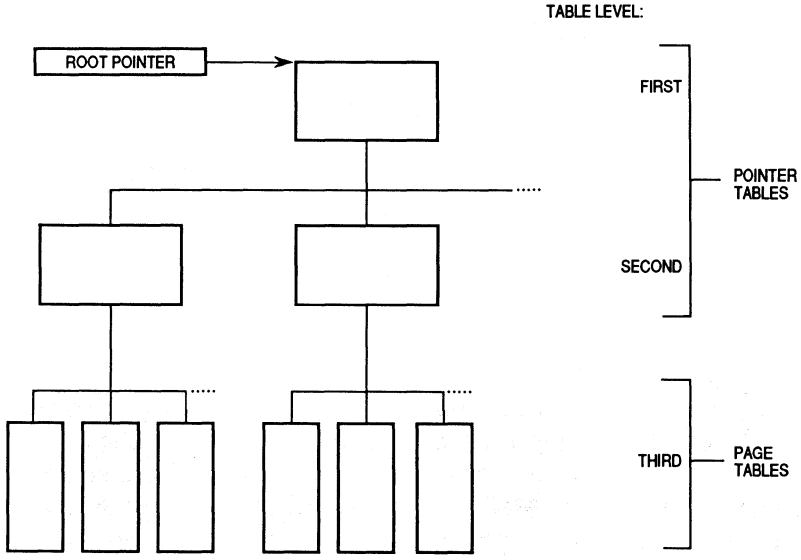


Figure 6-3. Translation Table Structure

The current privilege mode (either supervisor or user) selects the supervisor or user root pointer for translation of the access. Each root pointer contains the base address of the first-level table for a translation table tree. The base address for each table is indexed by a field (see Figure 6-4) extracted from the logical address. The table index A (TIA) field, which is seven bits wide, is used to index into the first-level pointer table and select one of 128 pointer descriptors. At this level, each descriptor corresponds to a 32 Mbyte block

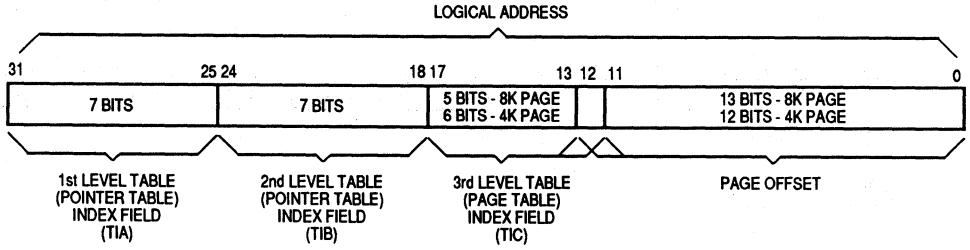


Figure 6-4. Table Index Fields

of memory and points to the base of a second-level table. Table index B (TIB) selects one of 128 pointer descriptors in the selected second-level table; each of these descriptors points to a page table in the third-level table and corresponds to a 256 Kbyte block of memory. Table index C (TIC) selects one of either 32 (for 8K pages) or 64 (for 4K pages) descriptors in the third-level page table. Descriptors in the page tables contain either a page descriptor for the translation or an indirect descriptor that points to a memory location containing the page descriptor. The page size, either 4K or 8K, is selected by a bit in the TC register.

Figure 6-5 shows an example of an access to address $76543210 in supervisor mode with a memory page size of 8K. The supervisor root pointer points to the base address of the level A table. The TIA field of the logical address, $3B, is mapped into bits 8–2 of the root pointer value to select a 32-bit descriptor at level A of the translation tree. The selected descriptor points to the base of a level B pointer table, and the TIB field of the logical address, $15, is mapped into bits 8–2 of this base address to select a descriptor within the table. This descriptor points to the base of a page table, and the TIC field of the logical address, $01, is mapped into bits 6–2 of this base address to select a descriptor within the table. A descriptor in a page table contains the physical base address of the page, user page attribute bits, caching mode selection bits, protection information, and history information for the page. Figure 6-6 shows a possible layout of this example translation tree in memory.

The address translation trees consist of tables of descriptors. The first- and second-level pointer table descriptors can be either resident or invalid. The third-level page table descriptors can be resident, indirect, or invalid. A page descriptor defines the physical address of a page frame in memory that corresponds to the logical address of a page. An indirect descriptor, which contains a pointer to the actual page descriptor, can be used when a single page descriptor is accessed by two or more logical addresses.

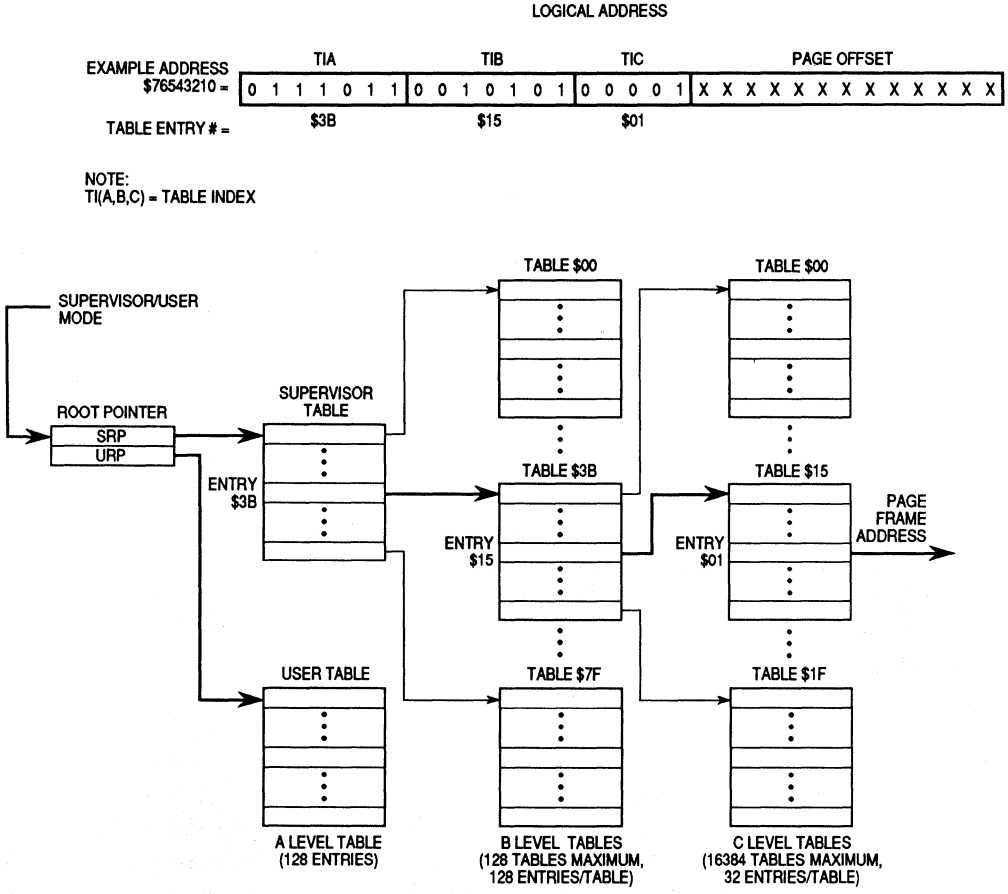


Figure 6-5. Translation Table Tree — Example

Invalid descriptors can be used at any level of the tree except the root. When a table search for a normal translation encounters an invalid descriptor, the processor takes a bus error exception. The invalid descriptor can be used to identify either a page or branch of the tree that has been stored on an external device and is not resident in memory or a portion of the translation table that has not yet been defined. In these two cases, the exception routine can either restore the page from disk or add to the translation table.

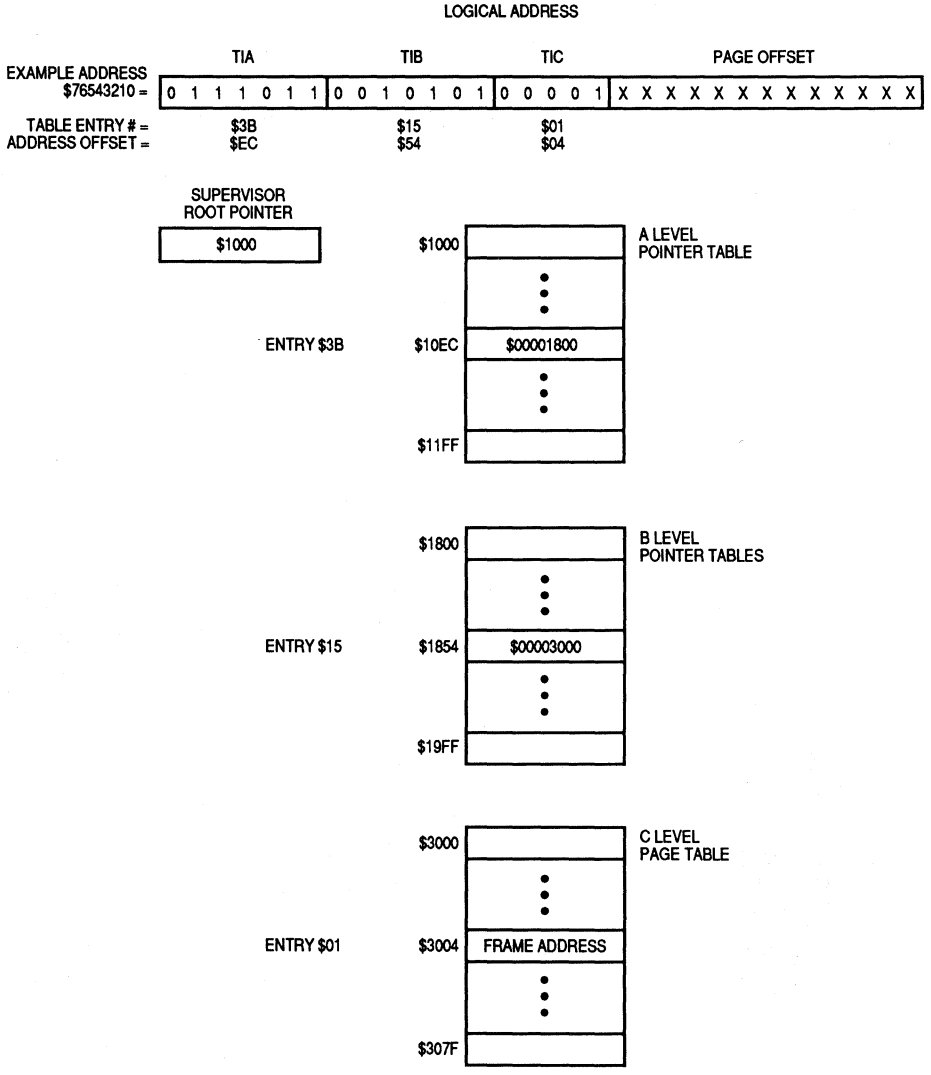


Figure 6-6. Translation Tree Layout in Memory — Example

6.2 ADDRESS TRANSLATION

The function of the MMUs is to translate logical addresses to physical addresses according to control information stored by the operating system in the MMU registers and in translation table trees resident in memory.

6.2.1 General Flow for Address Translation

For normal accesses, the translation process proceeds as follows for the accessed instruction or data memory unit:

1. Compare the logical address and privilege mode to the parameters in the transparent translation registers and use the logical address as a physical address for the access if one of the transparent translation registers match.
2. Compare the logical address and privilege mode to the tag portions of the entries in the ATC and use the corresponding physical address for the access when a match occurs.
3. When no transparent translation register nor valid ATC entry matches, initiate a table search operation to obtain the corresponding physical address from the translation tree, create a valid ATC entry for the logical address, and repeat step 2.

An alternate address space access is a special case that is immediately used as a physical address without translation.

Figure 6-7 provides a general flowchart for address translation. The top branch of the flowchart applies to transparent translation. The bottom three branches apply to ATC translation. If the requested access misses in the ATC, a table search operation proceeds. An ATC entry is created after the table search, and the access is retried. If an access hits in the ATC but a bus error or invalid descriptor was detected during the table search that created the ATC entry, the access is aborted, and a bus error exception is taken.

If a write or read-modify-write access results in an ATC hit but the page is write protected, the access is aborted, and a bus error exception is taken. If the page is not write protected and if the modified bit of the ATC entry is clear, a table search proceeds to set the modified bit in both the page descriptor in memory and in the ATC; the access is retried. If the modified bit of the ATC entry is set for a write or read-modify-write access to an unprotected page, if the resident bit is set (indicating the table search for the entry completed successfully), and if none of the mTTx registers (ITTx or DTTx, as appropriate) match, the ATC provides the address translation for the access.

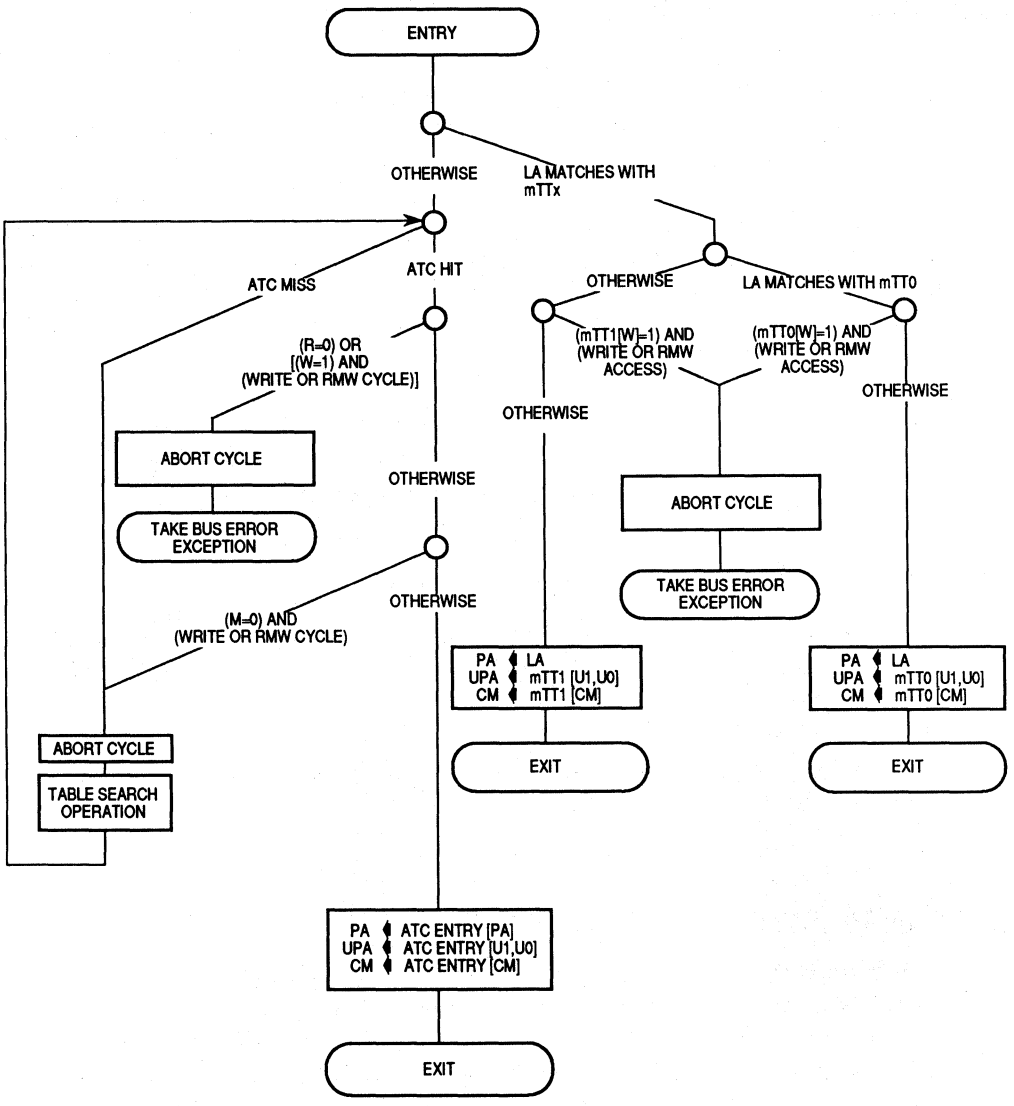


Figure 6-7. Address Translation General Flowchart

6.2.2 Affect of $\overline{\text{RSTI}}$ on the MMUs

When the MC68040 is reset by the assertion of the reset input ($\overline{\text{RSTI}}$) signal, the E bits of the TC, ITTx, and DTTx registers are cleared, disabling address translation. This reset causes logical addresses to be passed through as physical addresses, allowing an operating system to set up the translation tables and MMU registers, as required. After the translation tables and registers are initialized, the E bit of the TC register can be set, enabling paged address translation. While address translation is disabled, the attribute bits for an access that are normally supplied by an ATC entry or transparent translation register are zero, selecting writethrough cachable mode, no write protection, and user page attribute bits cleared.

A reset of the processor does not invalidate any entries in the ATCs. A PFLUSH instruction must be executed to flush all existing valid entries from the ATCs after a reset operation and before translation is enabled.

6.2.3 Affect of $\overline{\text{MDIS}}$ on Address Translation

The assertion of $\overline{\text{MDIS}}$ prevents the MMUs from performing searches of the ATCs and the execution unit from performing table searches. With address translation disabled, logical addresses are used as physical addresses. $\overline{\text{MDIS}}$ disables the MMUs on the next internal access boundary when asserted and enables the MMUs on the next boundary after the signal is negated. The assertion of this signal does not affect the operation of the transparent translation registers or execution of the PFLUSH or PTEST instructions.

6.3 TRANSPARENT TRANSLATION

Four independent transparent translation registers (DTT0 and DTT1 in the data MMU, ITT0 and ITT1 in the instruction MMU) optionally define four blocks of the logical address space that are directly translated to the physical address spaces. The blocks of addresses defined by the mTTx registers include at least 16 Mbytes of logical address space; the four blocks can overlap, or they can be separate.

The following description of the address comparison assumes that the mTTx registers are enabled; however, each mTTx register can be independently disabled. A disabled mTTx register is completely ignored.

When an MMU receives an address to be translated, the privilege mode and the eight high-order bits of the address are compared to the block of addresses defined by the two mTTx registers for the MMU. The address space

block for each mTTx register is defined by an S field, logical base address field, and logical address mask field. The S field allows matching either user or supervisor accesses or both accesses. When a bit in the logical address mask field is set, the corresponding bit of the logical base address is ignored in the address comparison and privilege mode. Setting successively higher order bits in the address mask increases the size of the transparently translated block.

The address for the current bus cycle and an mTTx register address match when the privilege mode and address bits (not including masked bits) are equal. Each mTTx register can specify write protection for the block. When write protection is enabled for a block, write or read-modify-write accesses to the block are aborted as if a nonresident table descriptor were encountered.

By appropriately configuring a transparent translation register, flexible transparent mappings can be specified (refer to **6.6.3 Transparent Translation Registers** for field identification). For instance, to transparently translate the user address space, the S field is set to $0, and the LOGICAL ADDRESS MASK is set to $FF in both an ITTx and DTTx register. To transparently translate supervisor accesses of addresses $00000000–$0FFFFFFF with write protection, the LOGICAL BASE ADDRESS field is set to $0x, the LOGICAL ADDRESS MASK is set to $0F, the W bit is set to one, and the S field is set to $1. The inclusion of independent TT registers in both the instruction and data MMUs provides an exception to the merged instruction and data address space, allowing different translations for instruction and operand accesses. Also, since the instruction memory unit is only used for instruction prefetches, different instruction and data TT registers can cause PC relative operand fetches to be translated differently from instruction prefetches.

Each mTTx register can specify the caching mode for logical addresses in its block. The four caching modes are cachable/writethrough, cachable/copyback, noncachable, and non-cachable/serialized. The writethrough and copyback caching modes force write accesses to either update the cache and write through to memory or to only update the cache, respectively. The noncachable mode forces matching entries in the cache to be pushed and invalidated and performs an external access with the cache inhibit out signal ($\overline{\text{CIOUT}}$) asserted to signal to external caches that the access should not be cached. The noncachable/serialized mode forces reads and writes within the block to occur in sequence to support I/O devices. Refer to **SECTION 7 INSTRUCTION AND DATA CACHES** for detailed information on caching modes.

Two user page attribute bits (U1 and U0) in each mTTx register are driven on the user page attribute (UPA1 and UPA0) signals if an external bus cycle

results from an access translated by the mTTx register. These bits can be programmed by the user to support extended addressing, bus snooping, or other applications.

If either of the mTTx registers match during an access to a memory unit (either instruction or data), the access is transparently translated. If both registers match, the mTT0 status bits are used for the access. Transparent translation can also be implemented by the translation tables of the translation trees if the physical addresses of pages are set equal to their logical addresses.

6.4 ADDRESS TRANSLATION CACHES (ATCs)

Each ATC is a 64-entry, four-way, set-associative cache that contains address translations similar in form to the corresponding page descriptors in memory. The purpose of the ATC is to provide a fast mechanism for address translation by avoiding the overhead associated with a table lookup of the logical-to-physical mapping of recently used logical addresses. Figure 6-8 shows the organization of the ATC.

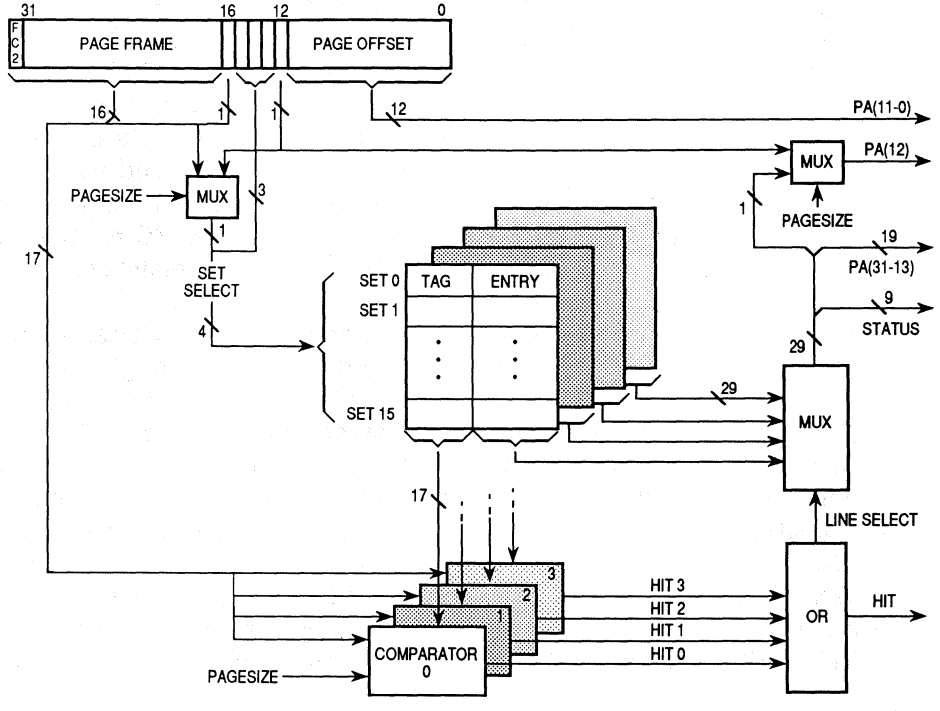


Figure 6-8. ATC Organization

The four bits of the logical address located just above the page offset (LA16–LA13 for 8K pages, LA15–LA12 for 4K pages) index into the ATC's 16 sets of entries. The tags are compared against the remaining upper bits of the logical address and FC2. If one of the tags matches and is valid, then the corresponding entry is chosen by the multiplexer to produce the physical address and status information. If no tag matches, then no mapping for the logical address exists in the ATC, and a table search is required.

There are some variations in the logical-to-physical mapping because of the two page sizes. If the page size is 4K, then logical address bit 12 is used to access the ATC's memory, bit 16 is used by the tag comparators, and physical address bit 12 is an ATC output. If the page size is 8K, then logical address bit 16 is used to access the ATC's memory and is ignored by the tag comparators, and physical address bit 12 is driven by logical address bit 12.

The MC68040 is organized such that the translation time of the ATCs is always completely overlapped by other operations; thus, no performance penalty is associated with ATC searches. The address translation occurs in parallel with indexing into the on-chip instruction and data caches.

When the ATC stores a new address translation, it replaces an invalid entry. When all entries in an ATC set are valid, the ATC selects a valid entry to be replaced, using a pseudo-random replacement algorithm. A two-bit counter, which is incremented for each ATC access, points to the entry to replace when an access misses in the ATC. ATC hit rates are application and page-size dependent, but hit rates ranging from 98% to greater than 99% can be expected. These high rates are achieved because the ATCs are relatively large (64 entries) and utilization efficiency is high with 8K and 4K page sizes.

Each ATC entry consists of a physical address, attribute information from a corresponding page descriptor, and a tag that contains a logical address and status information. Figure 6-9 shows the tag and entry fields.

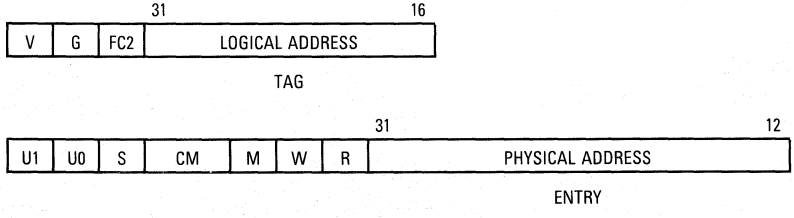


Figure 6-9. ATC Tag and Data

The following paragraphs define the bit fields shown in Figure 6-9.

V — VALID

When set, this bit indicates the validity of the entry. This bit is set when the MC68040 loads an entry. A flush operation by a PFLUSH or PFLUSHA instruction that selects this entry clears the bit.

G — GLOBAL

When set, G indicates the entry is global. Global entries are not invalidated by the PFLUSH instruction variants that specify nonglobal entries, even when all other selection criteria are satisfied. If these PFLUSH variants are not used, then this bit may be used by system software.

FC2 FUNCTION CODE BIT 2 (Supervisor/$\overline{\text{User}}$)

This bit contains the function code corresponding to the logical address in this entry. FC2 is set for supervisor mode accesses and cleared for user mode accesses.

LOGICAL ADDRESS

This 16-bit field contains the most significant logical address bits for this entry. All 16 bits of this field are used in the comparison of this entry to an incoming logical address when the page size is 4K bytes. For 8K pages, the least significant bit of this field is ignored.

U0,U1 USER PAGE ATTRIBUTES

These user-defined bits are not interpreted by the MC68040. U0 and U1 are echoed to the UPA0 and UPA1 signals, respectively, if an external bus transfer results from the access.

S SUPERVISOR PROTECTED

This bit identifies a pointer table or a page as a supervisor-only table or page. When the S bit is set, only programs operating in the supervisor privilege mode are allowed to access the portion of the logical address space mapped by this descriptor. If the bit is clear, both supervisor and user accesses are allowed.

CM CACHE MODE

This field selects the cache mode and access serialization for a page as follows:

00 CACHABLE, WRITETHROUGH

If the CM field indicates writethrough, then the access is considered cachable. A read access to a writethrough page is read from the

cache if matching data is found; otherwise, the data is read from memory and used to update the cache. Write accesses always write through to memory and update matching cache lines.

01 CACHABLE, COPYBACK
If the CM field indicates copyback, then the access is considered cachable. A read access to a copyback page reads from the cache if matching data is found; otherwise, the data is read from memory and used to update the cache. Write accesses that hit in the cache update the cache line and set the corresponding dirty status bits without an external bus access. If a write misses in the cache, the needed cache line is read from memory and updated in the cache.

10 CACHE INHIBITED, SERIALIZED

11 CACHE INHIBITED, NONSERIALIZED
If the CM field of a matching address indicates cache inhibited, the cache is bypassed, and an external bus transfer is performed. The data associated with the access is not cached internally, and the $\overline{\text{CIOUT}}$ signal is asserted during the bus transfer to indicate to external caches that the access should not be cached. If the data is already resident in an internal cache, then this data is pushed from the cache if dirty or invalidated if clean.

If the CM field indicates serialized, then the the sequence of read and write accesses to the page is guaranteed to match the sequence expected due to instruction ordering. Without serialization, the integer unit pipeline architecture can allow read accesses to occur before completion of a writeback for a prior instruction. Serialization also forces the operand read accesses for an instruction to occur only once by preventing the instruction from being interrupted after the operand fetch. Otherwise, the instruction is aborted, and the operand is accessed again when the instruction is later restarted. These guarantees apply only when the CM field is set to serialized and accesses are aligned.

Detailed information on caching modes is available in **7.2 CACHING MODES**, and information on serialization, in **SECTION 8 BUS OPERATION**.

M — MODIFIED
The modified bit is set when a valid write access to the logical address corresponding to the entry occurs. If the M bit is clear and a write access to this logical address is attempted, the MC68040 suspends the access, initiates a table search to set the M bit in the page descriptor, and writes over the old ATC entry with the current page descriptor information. The

MMU then allows the original write access to be performed. This procedure assures that the first write operation to a page sets the M bit in both the ATC and the page descriptor in the translation tables, even when a previous read operation to the page had created an entry for that page in the ATC with the M bit clear.

W — WRITE PROTECTED

This write-protect bit is set when a W bit is set in any of the descriptors encountered during the table search for this entry. Setting a W bit in a table descriptor write protects all pages accessed with that descriptor. When the W bit is set, a write access or a read-modify-write access to the logical address corresponding to this entry causes a bus error exception to be taken immediately.

R — RESIDENT

This bit is set if the table search successfully completes without encountering either a nonresident page or a transfer error acknowledge during the search.

PHYSICAL ADDRESS

The upper bits of the translated physical address are contained in this field.

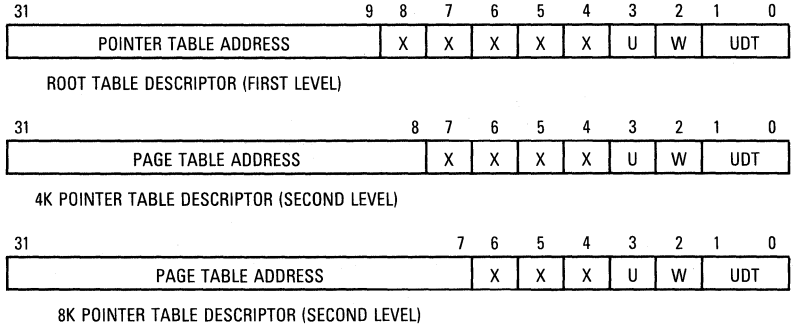
6.5 TRANSLATION TABLE DETAILS

The details of translation tables and their use include descriptions of the descriptors, table searching, translation table structure variations, and the protection techniques available with the MC68040 MMU.

6.5.1 Descriptor Details

The following paragraphs provide details on the table, page, and indirect descriptors, followed by a definition of the fields in the descriptors.

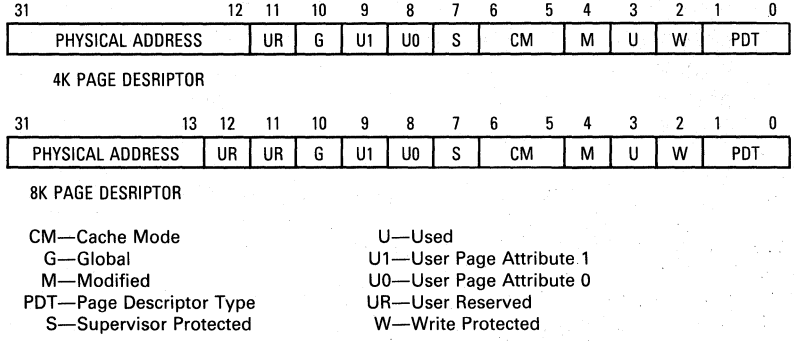
6.5.1.1 TABLE DESCRIPTORS. The formats of the first- and second-level table descriptors are shown in Figure 6-10. Two descriptor formats are possible at the second-level table to support 4K and 8K page sizes.



U—Used
UDT—Upper Level Descriptor Type
W—Write Protected
X—Motorola Reserved

Figure 6-10. Table Descriptors

6.5.1.2 PAGE DESCRIPTORS. The page descriptors for both 4K and 8K page sizes are shown in Figure 6-11.



CM—Cache Mode
G—Global
M—Modified
PDT—Page Descriptor Type
S—Supervisor Protected
U—Used
U1—User Page Attribute 1
U0—User Page Attribute 0
UR—User Reserved
W—Write Protected

Figure 6-11. Page Descriptors

6.5.1.3 INDIRECT DESCRIPTORS. The indirect descriptor format is shown in Figure 6-12.

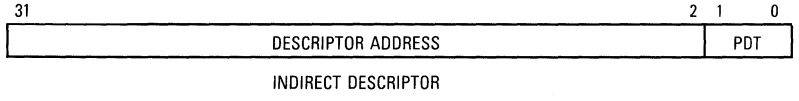


Figure 6-12. Indirect Descriptor

6.5.1.4 DESCRIPTOR FIELD DEFINITIONS. The field definitions for the table, page, and indirect descriptors are listed in alphabetical order:

CM — CACHE MODE

This field selects the cache mode and access serialization for a page as follows:

00 CACHABLE, WRITETHROUGH
If the CM field indicates writethrough, then the access is considered cachable. A read access to a writethrough page is read from the cache if matching data is found; otherwise, the data is read from memory and used to update the cache. Write accesses always write through to memory and update matching cache lines.

01 CACHABLE, COPYBACK
If the CM field indicates copyback, then the access is considered cachable. A read access to a copyback page reads from the cache if matching data is found; otherwise, the data is read from memory and used to update the cache. Write accesses that hit in the cache update the cache line and set the corresponding dirty status bits without an external bus access. If a write misses in the cache, the needed cache line is read from memory and updated in the cache.

10 CACHE INHIBITED, SERIALIZED

11 CACHE INHIBITED, NONSERIALIZED
If the CM field of a matching address indicates cache inhibited, the cache is bypassed, and an external bus transfer is performed. The data associated with the access is not cached internally, and the $\overline{\text{CIOUT}}$ signal is asserted during the bus transfer to indicate to external caches that the access should not be cached. If the data is already resident in an internal cache, then this data is pushed from the cache if dirty or is invalidated if clean.

If the CM field indicates serialized, then the the sequence of read and write accesses to the page is guaranteed to match the sequence expected due to instruction ordering. Without serialization, the integer unit pipeline architecture can allow read accesses to occur before completion of a writeback for a prior instruction. Serialization also forces the operand read accesses for an instruction to occur only once by preventing the instruction from being interrupted after the operand fetch. Otherwise, the instruction is aborted, and the operand is accessed again when the instruction is later restarted. These guarantees apply only when the CM field is set to serialized and accesses are aligned.

Detailed information on caching modes is available in **7.2 CACHING MODES**, and information on serialization, in **SECTION 8 BUS OPERATION**.

DESCRIPTOR ADDRESS

This 30-bit field, which contains the physical address of a page descriptor, is only used in indirect descriptors.

G — GLOBAL

When set, this bit indicates the entry is global. Global ATC entries are not invalidated by the PFLUSH instruction variants that specify nonglobal entries, even when all other selection criteria are satisfied. If these PFLUSH variants are not used, then this bit may be used by system software.

M — MODIFIED

This bit identifies a modified page. The MC68040 sets the M bit in the corresponding page descriptor before a write operation to a page for which the M bit is clear, except for write-protect supervisor violations. The read portion of a read-modify-write access is considered a write for updating purposes. The MC68040 never clears this bit.

PDT — PAGE DESCRIPTOR TYPE

This field identifies the descriptor as an invalid descriptor, a page descriptor for a resident page, or an indirect pointer to another page descriptor.

00,11 INVALID

These codes indicate that the descriptor is invalid. An invalid descriptor can represent a nonresident page or a logical address range that is out of bounds. All other bits in the descriptor are ignored. When an invalid descriptor is encountered, an ATC entry is created for the logical address with the resident (R) bit clear.

01 RESIDENT

This code indicates that the page is resident.

10 INDIRECT

This code indicates that the descriptor is an indirect descriptor. Bits 31–2 contain the physical address of the page descriptor. This encoding is invalid for a page descriptor pointed to by an indirect descriptor.

PHYSICAL ADDRESS

This 20-bit field contains the physical base address of a page in memory. The low-order bits of the address required to index into the page are supplied by the logical address. When the page size is 8K, the least significant bit of this field is not used.

S — SUPERVISOR PROTECTED

This bit identifies a page as supervisor only. When the S bit is set, only programs operating in the supervisor privilege mode are allowed to access the portion of the logical address space mapped by this descriptor. If the bit is clear, both supervisor and user accesses are allowed.

PAGE TABLE ADDRESS

This field contains the physical base address of a table of page descriptors. The low-order bits of the address required to index into the page table are supplied by the logical address.

U — USED

This bit is automatically set by the processor when a descriptor is accessed in which the U bit is clear. In a page descriptor table, this bit is set to indicate that the page corresponding to the descriptor has been accessed. In a pointer table, this bit is set to indicate that the pointer has been accessed by the MC68040 as part of a table search. Updates of the U bit are performed before the MC68040 allows a page to be accessed. The processor never clears this bit.

U0,U1 — USER PAGE ATTRIBUTES

These bits are user defined and are not interpreted by the MC68040. U0 and U1 are echoed to the UPA0 and UPA1 signals, respectively, if an external bus transfer results from the access. Applications for these bits include extended addressing and snoop protocol selection.

UDT — UPPER LEVEL DESCRIPTOR TYPE

These bits indicate whether or not the next level table is resident.

00,01 INVALID

These codes indicate that the table at the next level is not resident or that the logical address is out of bounds. All other bits in the descriptor are ignored. When an invalid descriptor is encountered, an ATC entry is created for the logical address with the resident (R) bit clear.

10,11 RESIDENT

These codes indicate that the page is resident.

UR — USER RESERVED

These bit fields are reserved for use by the user.

W — WRITE PROTECTED

Setting the write-protect (W) bit in a table descriptor write protects all pages accessed with that descriptor. When the W bit is set, a write access or a read-modify-write access to the logical address corresponding to this entry causes a bus error exception to be taken.

X — MOTOROLA RESERVED

These bit fields are reserved for future use by Motorola.

6.5.2 General Table Search

When an ATC does not contain a descriptor for the logical address of an access and when a translation is required, the MC68040 searches the translation tables in memory and obtains the physical address and status information for the page corresponding to the logical address. When a table search is required, the CPU suspends instruction execution activity and, at the end of a successful table search, stores the address mapping in the appropriate ATC and retries the access. The access then results in a match (it hits), and the translated address is transferred to the cache controller provided no exceptions were encountered.

The table search begins by selecting the translation tree, using internal function code bit FC2 for the access. FC2 is set for supervisor mode accesses and cleared for user mode accesses. The supervisor root point (SRP) is selected if FC2 is set; the user root pointer (URP) is selected if FC2 is cleared. A simplified flowchart of the table search procedure is shown in Figure 6-13.

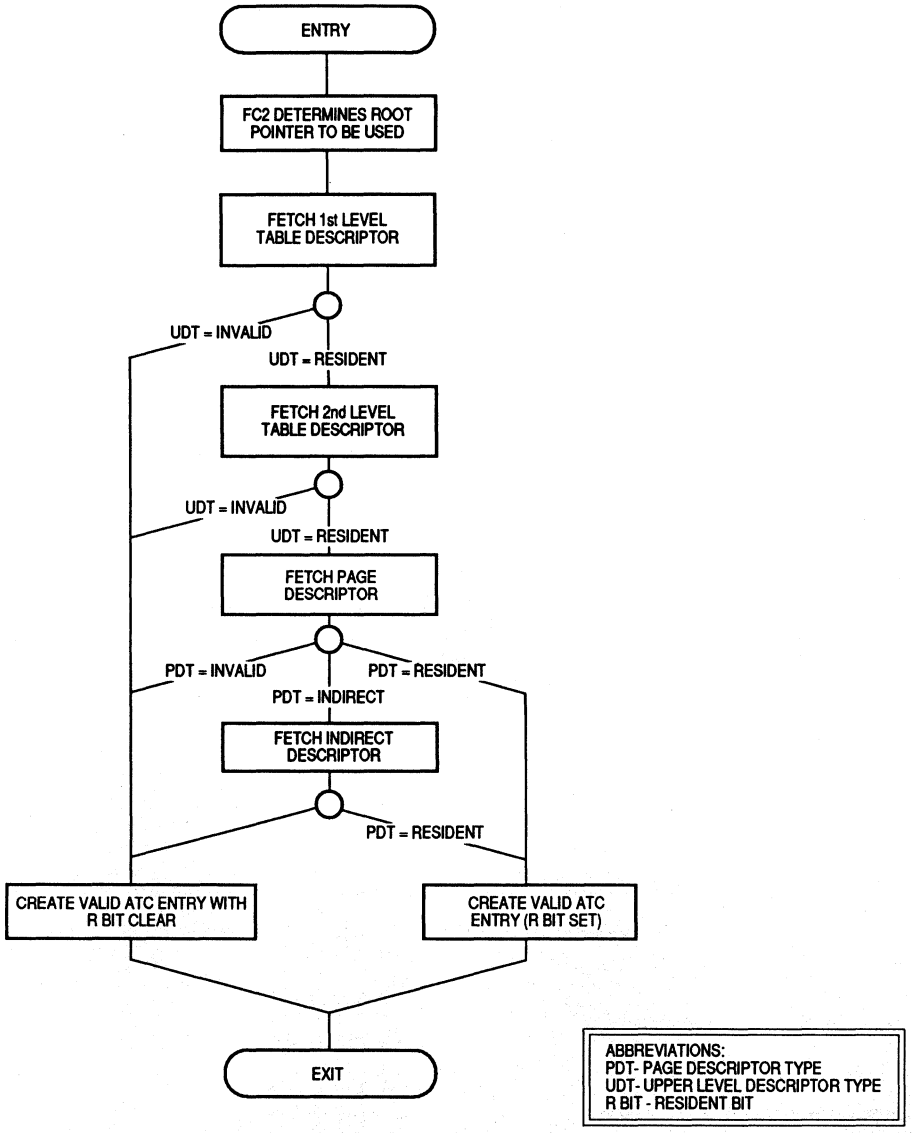


Figure 6-13. Simplified Table Search Flowchart

The table search uses physical addresses to access the translation tables. Table search accesses that are not read-modify-write accesses are treated by the cache as cachable/writethrough but do not allocate in the cache for misses. Read-modify-write table search accesses (which are required to update some descriptor U and M bit combinations) are treated as noncachable and force a matching cache line to be pushed and invalidated. Table search bus accesses are locked only for the specific portions of the table search that require a read-modify-write access.

The first access of the search uses the appropriate root pointer as the base address of the first table. The table is indexed by the TIA field of the logical address to access the first descriptor. If the descriptor is a resident descriptor, the table address field of the descriptor is used as a base address indexed by the TIB field of the logical address to access a descriptor in the second-level tables. The table address field of this descriptor is indexed by the TIC field of the logical address to fetch a descriptor from the page tables. If the descriptor from the page table is an indirect descriptor, the page descriptor pointed to by this descriptor is fetched. For a table search that successfully completes by accessing a valid page descriptor, the MC68040 creates an ATC entry, using the physical address and other information from the page descriptor, and retries the ATC lookup.

During a table search, the U bit in each descriptor that is encountered is checked and set if not already set. Similarly, when the table search is for a write access and the M bit of the page descriptor is clear, the processor sets the bit if the table search does not encounter a set W bit or a supervisor violation. Specific combinations of the U and M bits are updated by repeating the descriptor access as part of a read-modify-write access, allowing the external arbiter to prevent the update operation from being interrupted.

A table search terminates successfully when a page descriptor is encountered. The occurrence of an invalid descriptor or a transfer error acknowledge also terminates a table search, and the MC68040 takes an exception on the retry of the cycle because of these conditions. The exception routine should distinguish between anticipated conditions and true error conditions. The routine can correct an invalid descriptor that indicates a nonresident page or one that identifies a portion of the translation table yet to be allocated. A bus error due to a system malfunction may result in an error message and termination of the task.

Figure 6-14 shows how the various descriptors are fetched in a table search beginning with a root pointer and ending with an ATC entry and physical address. The example shown is for an 8K page size. The status bits of the ATC entry are derived by merging the status bits from the descriptors.

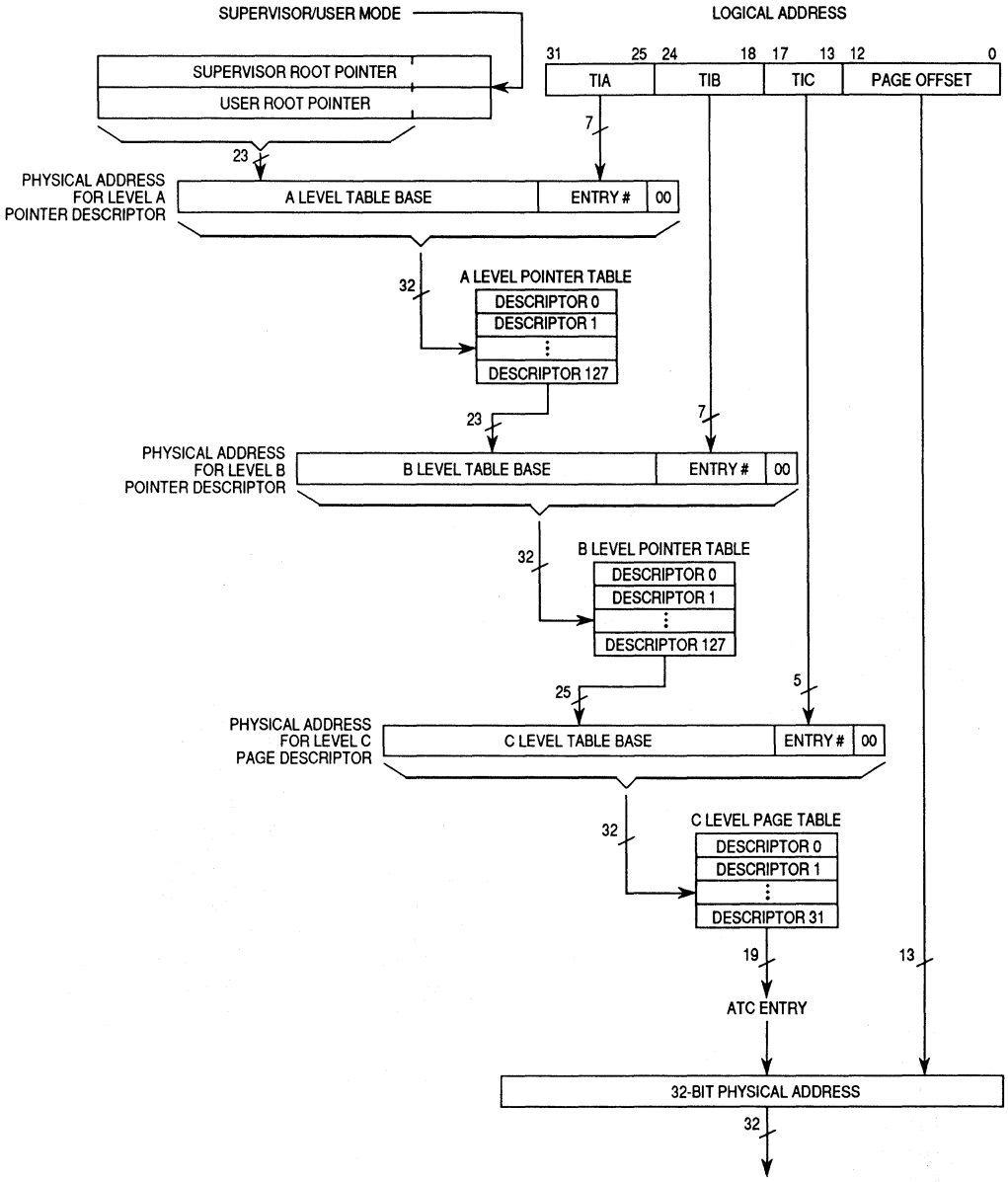


Figure 6-14. Physical Address Generation (8K Page Size)

The table search begins when an ATC miss is detected. The table address field of the appropriate root pointer is used as the base address of the first table. The table is indexed by the set of logical address bits defined by the table index fields TIA, TIB, and TIC (table index fields A, B, and C), as shown in Figure 6-14.

The upper 23 bits of the appropriate root pointer are concatenated with the seven bits of the TIA field of the logical address and multiplied by four (shifted to the left by two bits) to yield the physical address of the first-level table descriptor. The first-level table descriptor is fetched, and its upper 23 bits are concatenated with the seven bits of the TIB field of the logical address and multiplied by four to produce the physical address of the second-level table descriptor. The second-level table descriptor is fetched, and its upper 25 bits are concatenated with the TIC field (five bits for an 8K page, six bits for a 4K page) of the logical address to produce the physical address of the page descriptor. The upper 19 bits of the page descriptor become the page frame physical address. Write-protect status is accumulated from each descriptor level and combined with the status from the page descriptor to form the ATC entry status. The MC68040 creates the ATC entry from the page frame address and the associated status bits and retries the original bus access.

An indirect table search is identical to the preceding discussion except that the page table contains a pointer to a page descriptor rather than the descriptor itself.

6.5.3 Variations in Translation Table Structure

Several aspects of the MMU translation tree structure are software configurable, allowing the system designer flexibility to optimize the performance of the MMUs for a particular system. The following paragraphs discuss the variations of the tree structure from the general structure discussed previously.

6.5.3.1 INDIRECTION. The MC68040 provides the ability to replace an entry in a page table with a pointer to an alternate entry. The indirection capability allows multiple tasks to share a physical page while maintaining only a single set of history information for the page (i.e., the "modified" indication is maintained only in the single descriptor). The indirection capability also allows the page frame to appear at arbitrarily different addresses in the logical address spaces of each task.

Using the indirection capability, single entries or entire tables can be shared between multiple tasks. Figure 6-15 shows two tasks sharing a page using indirect descriptors.

When the MC68040 has completed a normal table search, it examines the descriptor type field of the last entry fetched from the page tables. If the PDT field contains an indirect ($2) encoding, this indicates that the address contained in the highest order 30 bits of the descriptor is a pointer to the page descriptor that is to be used to map the logical address. The processor then fetches the page descriptor from this address and uses the physical address field of the page descriptor as the physical mapping for the logical address.

The page descriptor located at the address given by the indirect descriptor must not have a PDT field with an indirect encoding (it must be either be a resident descriptor or invalid). Otherwise, the descriptor is treated as invalid, and the MC68040 creates an ATC entry with an error condition signaled (R bit clear).

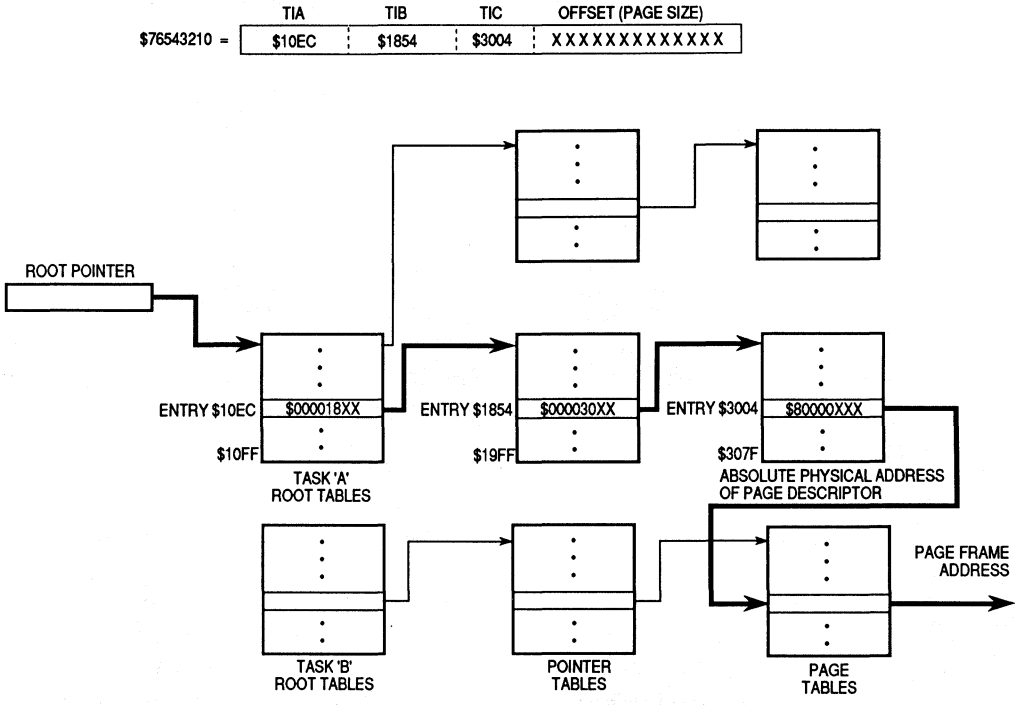


Figure 6-15. Translation Tree Using Indirect Descriptors — Example

6.5.3.2 TABLE SHARING BETWEEN TASKS. A page or pointer table can be shared between tasks by placing a pointer to the shared table in the address translation tables of more than one task. The upper (nonshared) tables can contain different write protection settings, allowing different tasks to use the memory areas with different write permissions. In Figure 6-16, two tasks share the memory translated by the table at the pointer table level. Task "A" cannot write to the shared area; task "B", however, has the W bit clear in its pointer to the shared table so it can read and write the shared area. Also, the shared area appears at different logical addresses for each task.

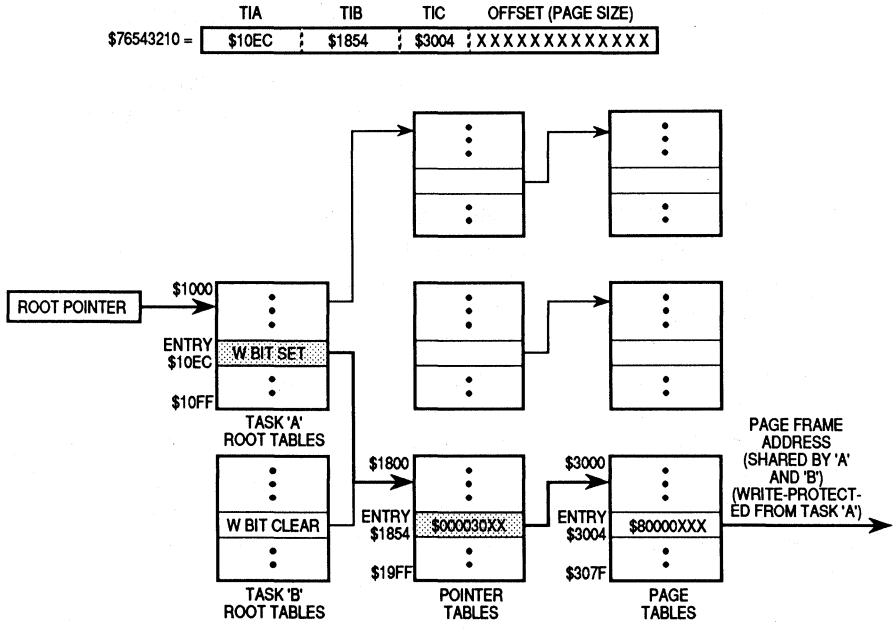


Figure 6-16. Translation Tree Using Shared Tables — Example

6.5.3.3 PAGING OF TABLES. The entire address translation tree for an active task need not be resident in main memory at once. In the same way that only the working set of pages must reside in main memory, only the tables that describe the resident set of pages need be available in main memory. This paging of tables is implemented by placing the "invalid" code ($0 or $1) in the UDT field of the table descriptor that points to the absent table(s). When a task attempts to use an address that would be translated by an absent

table, the MC68040 is unable to locate a translation and takes a bus error exception when the execution unit retries the bus access that caused the table search to be initiated.

System software determines that the "invalid" code in the descriptor corresponds to nonresident tables. This determination can be facilitated by using the unused bits in the descriptor to store status information concerning the invalid encoding. When the MC68040 encounters an "invalid" descriptor, it makes no interpretation (or modification) of any fields of this descriptor other than the UDT field, allowing the operating system to store system-defined information in the remaining bits. Typical stored information includes the reason for the "invalid" encoding (tables paged out, region not allocated, etc.) and possibly the disk address for nonresident tables.

Figure 6-17 shows an address translation table in which only a single page table (table $15) is resident and all other page tables are not resident.

6.5.3.4 DYNAMIC ALLOCATION OF TABLES. Similar to paged tables, a complete translation tree need not exist for an active task. The translation tree can be dynamically allocated by the operating system based on requests for access to particular areas.

As in demand paging, it is difficult, if not impossible, to predict the areas of memory that are used by a task over any extended period of time. Instead of attempting to predict the requirements of the task, the operating system performs no action for a task until a "demand" is made requesting access to a previously unused area or an area that is no longer resident in memory. This same technique can be used to efficiently create a translation tree for a task.

For example, consider an operating system that is preparing the system to execute a previously unexecuted task that has no translation tree. Rather than guessing what the memory-usage requirements of the task are, the operating system creates a translation tree for the task that maps one page corresponding to the initial value of the program counter for that task and one page corresponding to the initial stack pointer of the task. All other branches of the translation tree for this task remain unallocated until the task requests access to the areas mapped by these branches. This technique allows the operating system to construct a minimal translation tree for each task, conserving physical memory utilization and minimizing operating system overhead.

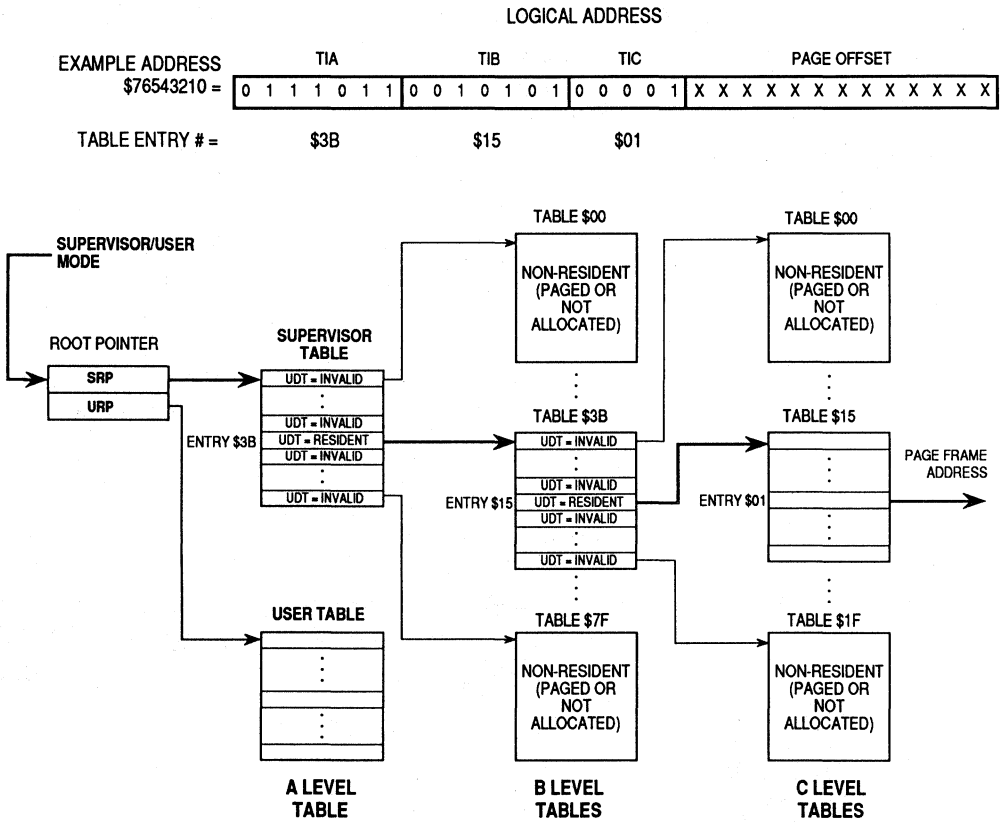


Figure 6-17. Translation Tree with Nonresident Tables — Example

6.5.4 Table Search Operation Details

The table search operations are described in detail in Figure 6-18, which shows a detailed flowchart of the table search operation, and in Figure 6-19, which shows the details of a descriptor fetch operation.

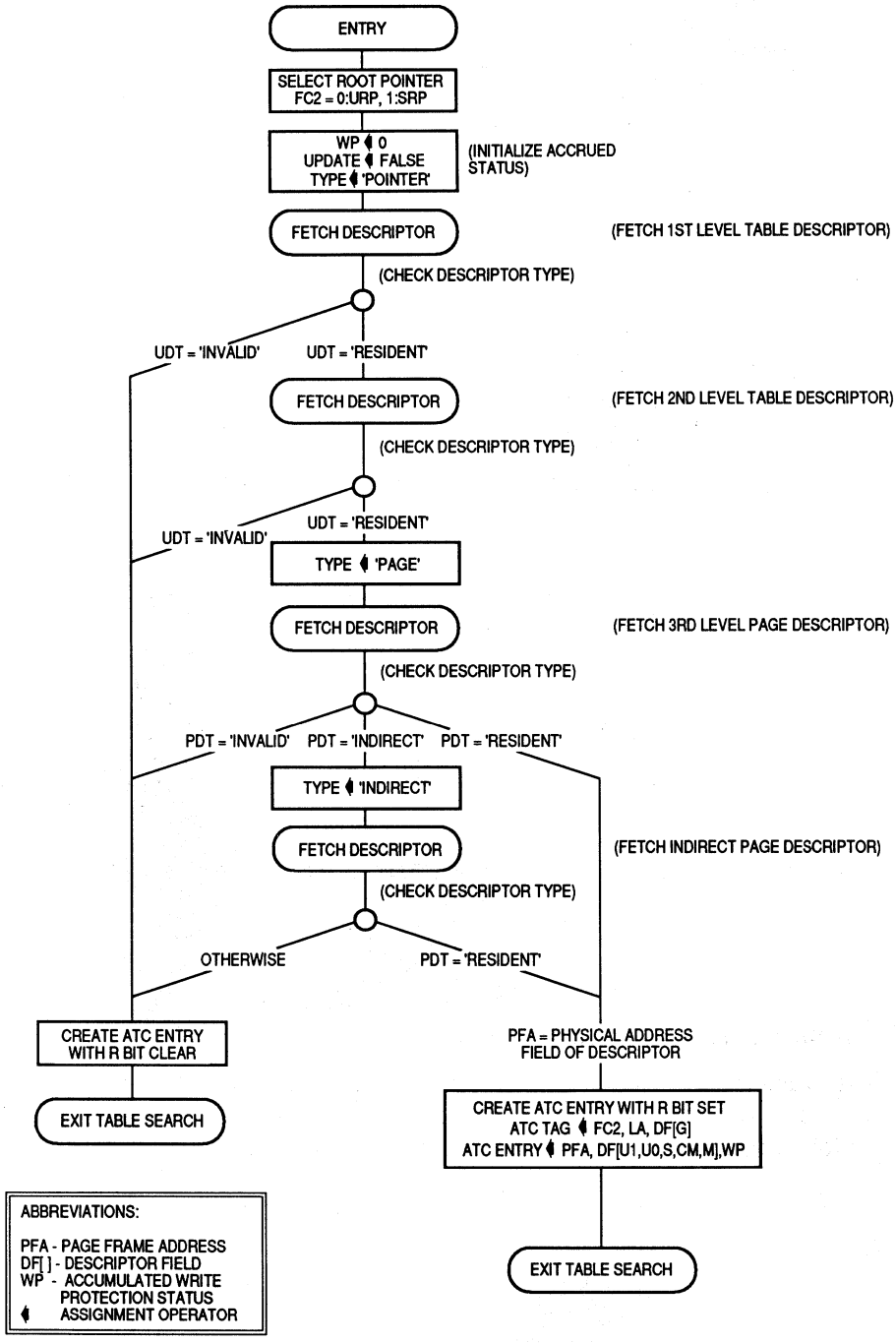


Figure 6-18. Detailed Flowchart of Table Search Operation

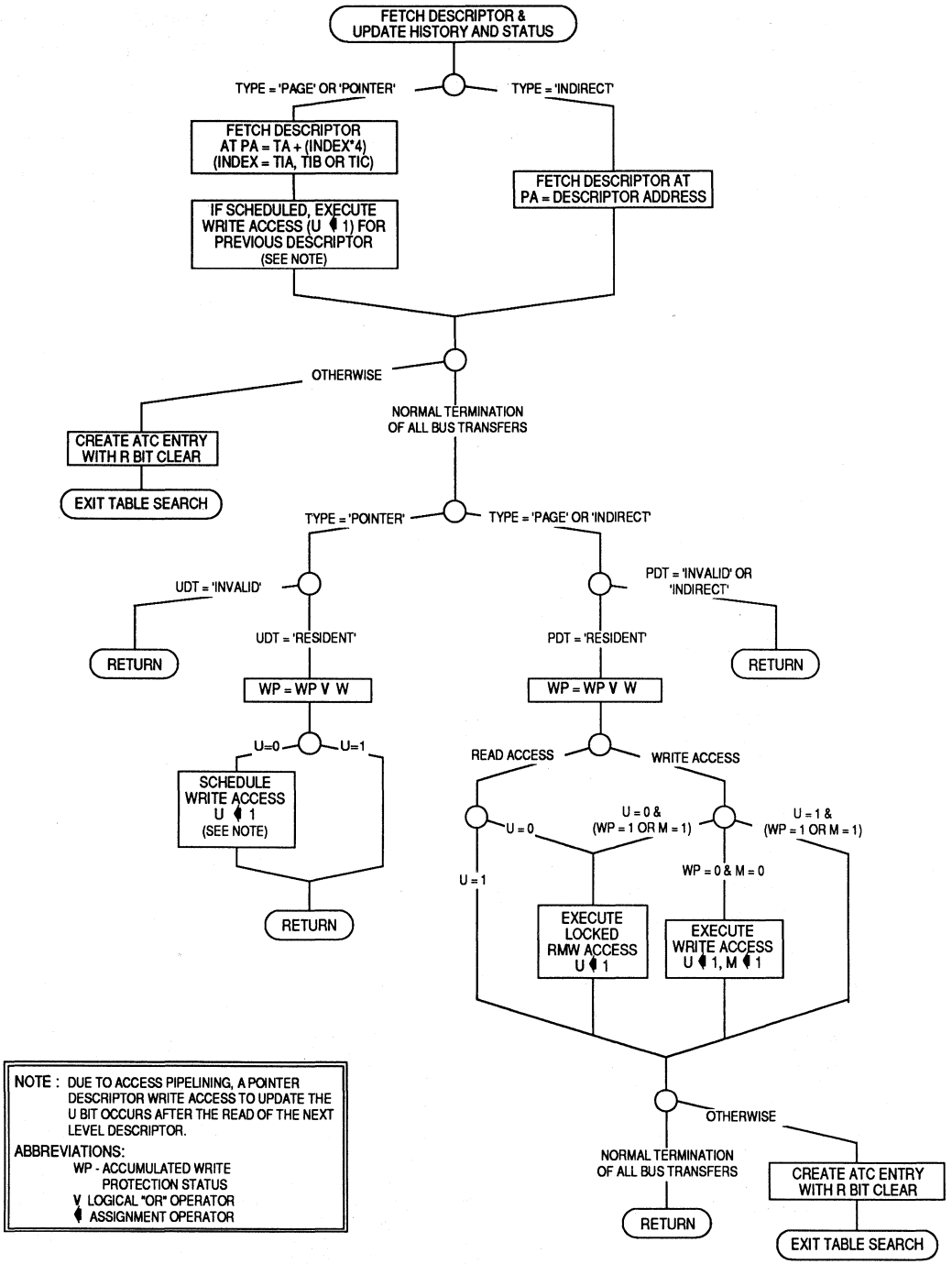


Figure 6-19. Detailed Flowchart of Descriptor Fetch Operation

As shown in Figure 6-19, the MC68040 asserts the $\overline{\text{LOCK}}$ signal during certain portions of the table search to assure proper maintenance of the U and M bits. The U and M bits are updated before the MC68040 allows a page to be accessed or written. As descriptors are fetched, the U and M bits are monitored. Write cycles modify these bits when required. For a table descriptor, a write cycle to set the U bit occurs only if the U bit was clear. Table 6-1 lists the page descriptor update operations for each combination of U bit, M bit, write protection, and read or write access type.

Table 6-1. Updating U and M Bits for Page Descriptors

Previous Status		WP	Access Type	Page Descriptor Update Operation	New Status	
U	M				U	M
0	0	X	Read	Locked RMW Access to Set U	1	0
0	1			Locked RMW Access to Set U	1	1
1	0			None	1	0
1	1			None	1	1
0	0	0	Write	Write to Set U and M	1	1
0	1			Locked RMW Access to Set U	1	1
1	0			Write to Set M	1	1
1	1			None	1	1
0	0	1		Locked RMW Access to Set U	1	0
0	1			Locked RMW Access to Set U	1	1
1	0			None	1	0
1	1			None	1	1

NOTE: WP = Accumulated write-protect status

6.5.5 Protection

The MC68040 MMUs provide separate translation trees for supervisor and user address spaces. The translation table trees contain both mapping and protection information. Each table and page descriptor includes a write-protect (W) bit that can be set to provide write protection at any level. Page descriptors also contain a supervisor-only (S) bit that can limit access to programs operating at the supervisor privilege level.

The protection mechanisms can be used individually or in any combination to protect:

- Supervisor address space from access by user programs.
- User address space from access by other user programs.

- Supervisor and user program spaces from write accesses (implicitly supported by designating all memory pages used for program storage as write protected).
- One or more pages of memory from write accesses.

6.5.5.1 USER AND SUPERVISOR TRANSLATION TREES. One way of protecting supervisor and user address spaces from unauthorized accesses is to use separate supervisor and user translation trees. Separate trees protect supervisor programs and data from access by user programs and user programs and data from access by supervisor programs. Access is granted to the supervisor programs that can access any area of memory with the move address space (MOVES) instruction. The translation tree pointed to by the SRP is selected for all other supervisor mode accesses. This translation tree can be common to all tasks. Figure 6-20 shows separate translation trees for supervisor accesses and for two user tasks that share the common supervisor space. Each user task has an address translation tree with unique mappings for the logical addresses in its user address space.

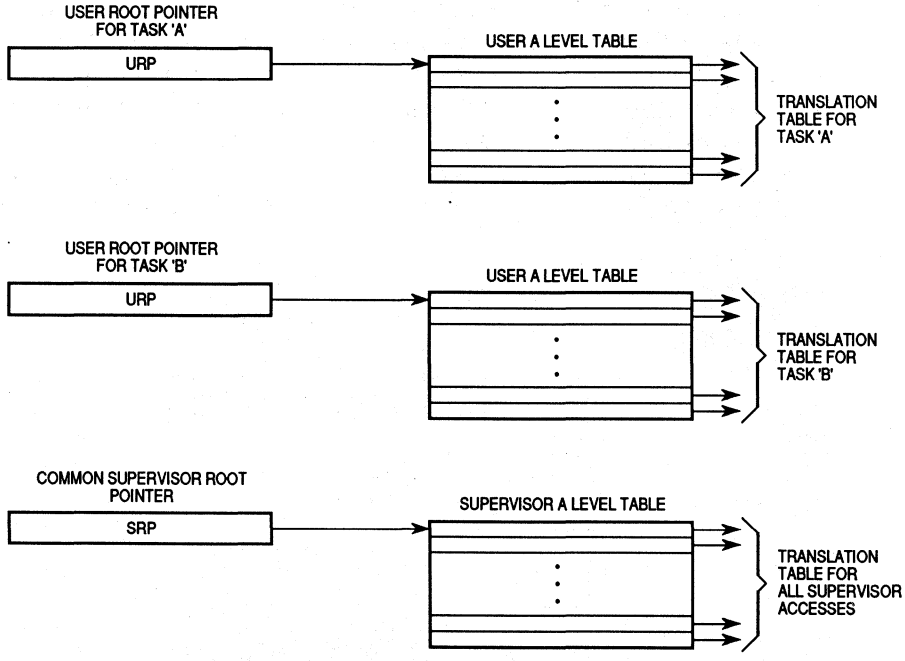


Figure 6-20. Translation Tree Structure for Two Tasks — Example

6.5.5.2 SUPERVISOR ONLY. A second mechanism protects supervisor programs and data without requiring segmenting of the logical address space into supervisor and user address spaces. Page descriptors contain S bits to protect areas of memory from access by user programs. When a table search for a user access encounters an S bit set in a page descriptor, the table search is completed, and an ATC descriptor corresponding to the logical address is created with the R bit clear. The subsequent retry of the user access results in a bus error exception being taken. The S bit can be used to protect one or more pages from user program access. Descriptors can be shared by supervisor and user mode accesses by using indirect descriptors or by sharing tables. The entire user and supervisor address spaces can be mapped together by loading the same root pointer address into both the SRP and URP registers.

6.5.5.3 WRITE PROTECT. The MC68040 provides write protection independent of the segmented address spaces for programs and data. All table and page descriptors contain W bits to protect areas of memory from write accesses of any kind, including supervisor writes. When a table search encounters a W bit set in any table or page descriptor, an ATC descriptor corresponding to the logical address is created with the W bit set after the table search is completed. The subsequent retry of the write access results in a bus error exception being taken. The W bit can be used to protect the entire area of memory defined by a branch of the translation tree or protect only one or more pages from write accesses. Figure 6-21 shows a memory map of the logical address space organized to use supervisor-only and write-protect bits for protection. Figure 6-22 shows an example translation tree for this technique.

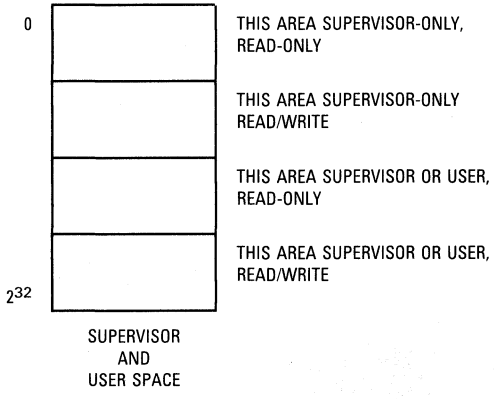


Figure 6-21. Logical Address Map with Shared Supervisor and User Address Spaces — Example

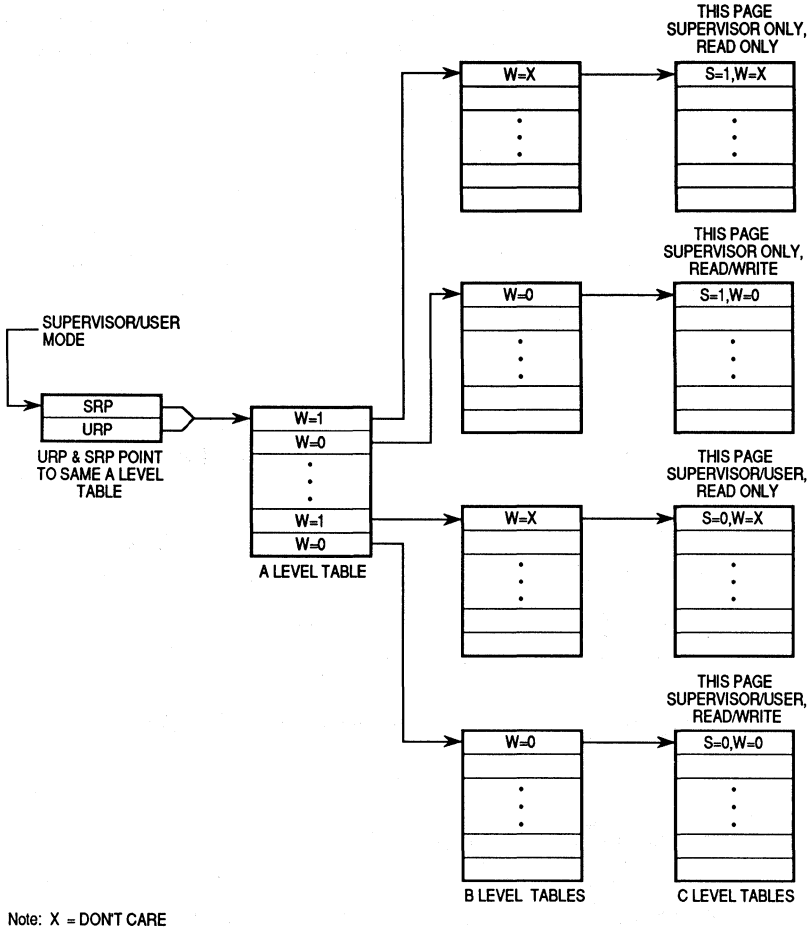


Figure 6-22. Translation Tree Using S and W Bits To Set Protection — Example

6.6 REGISTERS

The registers of the MMUs described here are part of the supervisor programming model for the MC68040.

The eight registers that control and provide status information for address translation in the MC68040 are the user root pointer register (URP), the supervisor root pointer register (SRP), the translation control register (TC), four independent transparent translation control registers (ITT0, ITT1, DTT0, and

DTT1), and the MMU status register (MMUSR). These registers can be directly accessed only by programs that execute in supervisor mode.

6.6.1 Root Pointer Registers

The SRP and URP registers each contain the physical address of the root of the translation tree used for supervisor and user accesses, respectively. The URP points to the translation tree for the current user task. When a new task begins execution, the operating system typically writes a new root pointer to the URP. A new translation table address implies that the contents of the ATCs may no longer be valid. A PFLUSH instruction should be executed to flush the ATCs before loading a new root pointer value, if necessary. Figure 6-23 shows the format of the 32-bit URP and SRP registers. Bits 8–0 of an address loaded into the URP or the SRP must be zero.

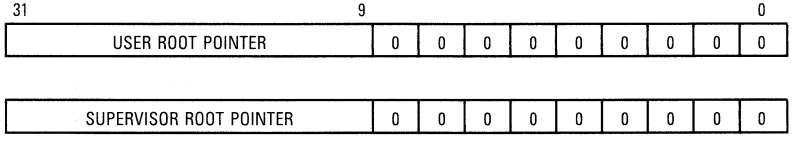
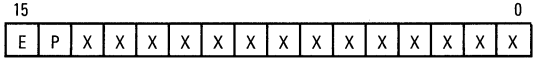


Figure 6-23. Root Pointer Register (URP, SRP) Format

6.6.2 Translation Control Register

The 16-bit TC register contains two control bits to enable paged address translation and to select page size. The operating system must flush the ATCs before enabling address translation since the ATCs are not flushed by TC register accesses or by reset. All unimplemented bits of this register are read as zeros and must always be written as zeros. The TC register is shown in Figure 6-24.



E — Enable Translations
P — Page Size (8–4K)
X — Undefined (Reserved)

Figure 6-24. Translation Control Register

The fields are as folows:

E — Enable

This bit enables and disables paged address translation.

0 = Disable

1 = Enable

A reset operation clears this bit. When translation is disabled, logical addresses are used as physical addresses. The MMU instructions (PTEST and PFLUSH) can be executed successfully regardless of the state of the E bit. If translation is disabled and an access does not match a transparent translation register, the access has the following default attributes: the caching mode is cachable/writethrough, write protection is disabled, and the user attribute signals (UPA1 and UPA0) are zero.

P — Page Size

This bit selects the memory page size.

0 = 4K bytes

1 = 8K bytes

A reset operation sets this bit, selecting 8K pages.

6.6.3 Transparent Translation Registers

The data transparent translation registers (DTT0 and DTT1) and instruction transparent translation registers (ITT0 and ITT1) are 32-bit registers that define blocks of logical address space that are transparently translated. Logical addresses in a transparently translated block are used as physical addresses with two user-defined page attributes and optional write protection. The minimum size block that can be defined by a transparent translation (TT) register is 16 Mbytes of logical address space. The TT registers can specify blocks that overlap. The TT registers operate independently of the E bit in the TC register and the state of the $\overline{\text{MDIS}}$ signal. If both a TT register and an ATC entry match a logical address, then the TT register is used for the translation, and the ATC entry is ignored. TT0 is used if both TT registers in a memory unit match. The format of the TT registers is shown in Figure 6-25.

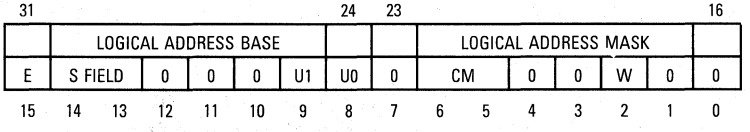


Figure 6-25. Transparent Translation Register Format

The fields of the transparent translation registers are as follows:

LOGICAL ADDRESS BASE

This 8-bit field is compared with address bits A31–A24. Addresses that match in this comparison (and are otherwise eligible) are transparently translated.

LOGICAL ADDRESS MASK

Since this 8-bit field contains a mask for the LOGICAL ADDRESS BASE field, setting a bit in this field causes the corresponding bit in the LOGICAL ADDRESS BASE field to be ignored. Blocks of memory larger than 16 Mbytes may be transparently translated by setting some of the logical address mask bits to ones. The low-order bits of this field are normally set to define contiguous blocks larger than 16 Mbytes, although this is not required.

E — ENABLE

This bit enables and disables transparent translation of the block defined by this register:

0 = Transparent translation disabled
1 = Transparent translation enabled

S — SUPERVISOR/USER MODE

This field specifies the way FC2 is used in matching an address:

00 = Match only if FC2 is 0 (user mode access)
01 = Match only if FC2 is 1 (supervisor mode access)
1x = Ignore FC2 when matching

U1, U2 — USER PAGE ATTRIBUTES

These bits are user defined and are not interpreted by the MC68040. U0 and U1 are echoed to the UPA0 and UPA1 signals, respectively, if an external bus transfer results from the access.

This field selects the cache mode and access serialization for the block as follows:

00 = Cachable, Writethrough
01 = Cachable, Copyback
10 = Cache Inhibited, Serialized
11 = Cache Inhibited, Not Serialized

Each mode is detailed in **6.5.1.5 DESCRIPTOR FIELD DEFINITIONS**.

W — WRITE PROTECT

This bit indicates if the transparent block is write protected. If set, write and read-modify-write accesses are aborted as if the resident (R) bit in a table descriptor were clear.

0 = Read and write accesses permitted
1 = Write accesses not permitted

6.6.4 MMU Status Register

The MMUSR is a 32-bit register that contains the status information returned by execution of the PTEST instruction. The PTEST instruction searches the translation tables to determine status information about the translation of a specified logical address. The MMUSR is shown in Figure 6-26.

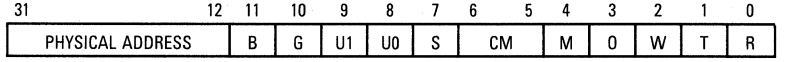


Figure 6-26. MMU Status Register

The fields of the MMUSR are as follows:

PHYSICAL ADDRESS
This 20-bit field contains the upper bits of the translated physical address. The actual physical address is formed by merging these bits with the lower bits of the logical address.

B — BUS ERROR
B is set if a transfer error is encountered during the table search for the PTEST instruction. If B is set, all other bits are zero.

G — GLOBAL
This bit is set if the G bit is set in the page descriptor.

U1, U0 — USER PAGE ATTRIBUTES
These bits are set if corresponding bits in the page descriptor are set.

S — SUPERVISOR PROTECTION
This bit is set if the S bit in the page descriptor is set. Setting this bit does not indicate that a violation has occurred.

CM — CACHE MODE
This 2-bit field is copied from the CM bits in the page descriptor.

M — MODIFIED
This bit is set if the M bit is set in the page descriptor associated with the address.

W — WRITE PROTECT
This bit is set if the W bit is set in any of the descriptors encountered during the table search. Setting this bit does not indicate that a violation occurred.

T — TRANSPARENT TRANSLATION REGISTER HIT
If T is set, then the PTEST address matched an instruction or data TTR, the R bit is set, and all other bits are zero.

R — RESIDENT
R is set if the PTEST address matches a TTR or if the table search completes by obtaining a valid page descriptor.

6.6.5 Register Programming Considerations

If the entries in the ATCs are no longer valid when a reset operation occurs (as is normally expected), an explicit flush operation must be specified by the system software. The assertion of $\overline{\text{RSTI}}$ disables translation by clearing the E bits of the TC, DTTx, and ITTx registers, but it does not flush the ATCs. Reading or writing any of the MMU registers (URP, SRP, TC, MMUSR, DTT0, DTT1, ITT0, ITT1) does not flush the ATCs. Since a write to these registers can cause some or all of the address translations to change, the write should be followed by a PFLUSH operation to flush the ATCs if necessary.

The status bits in the MMUSR indicate conditions to which the operating system should respond. In a typical bus error handler routine, the flow shown in Figure 6-27 can be used to determine the cause of an MMU fault. The PTEST instruction sets the bits in the MMUSR appropriately, and the program can branch to the appropriate code segment for the condition.

6.7 MMU INSTRUCTIONS

The MC68040 instruction set includes three privileged instructions that perform MMU operations. A brief description of each of these instructions follows. For detailed descriptions of these instructions, refer to M68000PM/AD, *M68000 Family Programmer's Reference Manual.*

The MOVEC instruction transfers data between an integer register or memory location and any of the MC68040 control and status registers. The operating system uses the MOVEC instruction to control and monitor MMU operation by manipulating and reading the eight MMU registers.

The PFLUSH instruction flushes (invalidates) address translation descriptors in the ATCs. PFLUSHA, a version of the PFLUSH instruction, flushes all entries. The PFLUSH instruction flushes a user or supervisor entry with a specified logical address. The PFLUSHAN and PFLUSHN instruction variants qualify entry selection further by flushing only entries that are nonglobal, indicated by a cleared G bit in the entry.

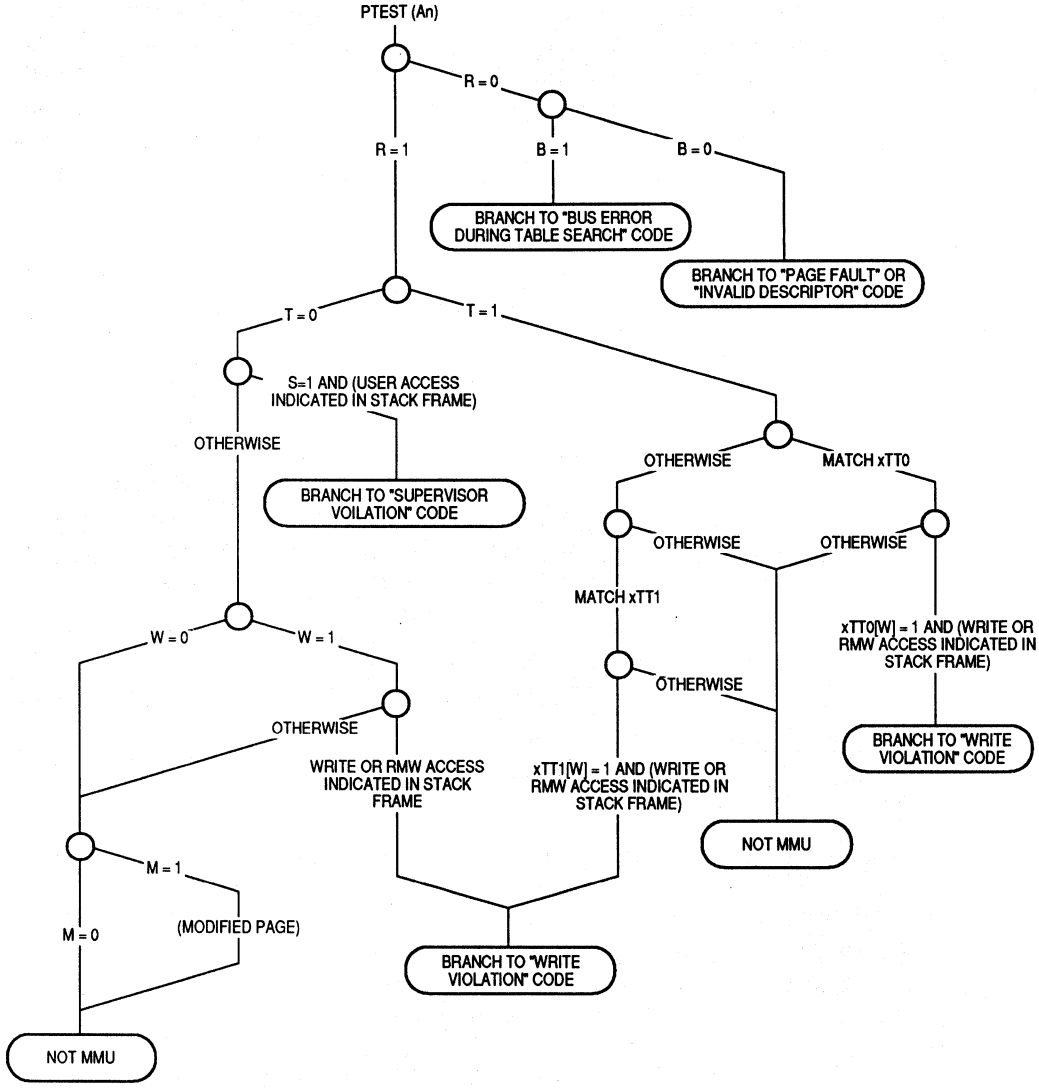


Figure 6-27. MMU Status Interpretation

The PTEST instruction performs a table search operation for a specified function code and logical address and sets the appropriate bit fields in the MMUSR to indicate conditions encountered during the search. PTEST automatically flushes the corresponding entry from the cache before searching the tables and loads the latest information from the translation tables into the ATC. The exception routines of the operating system can use this instruction to identify MMU faults.

This instruction is primarily used in bus error handler routines. For example, if a bus error has occurred, the handler can execute an instruction sequence such as follows:

```
MOVE.B (A7,offset1),D0    Copy transfer modifier field from stack frame
MOVEC D0,DFC                into DFC register
MOVEA.L (A7,offset2),A0   Copy fault address from stack frame into address register
PTESTW (A0)               Test address in A0 with function code in DFC registers
```


The transfer modifier field copied into the DFC register indicates whether the faulted access was a supervisor or user mode access and whether it was an instruction prefetch or data access. The PTEST instruction uses the DFC value to determine which translation tree (supervisor or user) to search and which ATC (data or instruction) to create the entry in. After executing this code sequence, the handler can examine the MMUSR for the source of the fault.

The MC68040 MMU instructions use opcodes that are different from those for the corresponding instructions in the MC68030 and MC68851. All MMU opcodes for the MC68030 and MC68851 cause F-line unimplemented instruction exceptions if executed in either supervisor or user mode by the MC68040.

SECTION 7
INSTRUCTION AND DATA CACHES

The MC68040 contains a 4K-byte on-chip instruction cache and a 4K-byte on-chip data cache, located in the physical address space. The caches improve system performance by providing cached data to the on-chip execution units with very low latency and 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 direct memory access (DMA) controller. 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.

The instruction and data caches (see Figure 7-1) are contained in the instruction and data memory units, respectively. Instruction prefetch requests and data requests from the integer unit are independently serviced by the appropriate memory unit, which translates the logical address in parallel with indexing into the memory unit's cache. If the translated address matches one of the cache entries, the access hits in the cache, and the memory unit supplies the data to the integer unit (for a read) or updates the cache (for a write). If the access misses in the cache or a write access must be written through to memory, the memory unit sends an external access request to the bus controller, which reads or writes the required data.

Cache coherency support in the MC68040 is optimized for use in multi-master applications that utilize the MC68040 as a caching master sharing memory with one or more noncaching masters (such as DMA controllers). The MC68040 implements a bus snooper that maintains coherency of the caches by monitoring the accesses by an alternate bus master and performing cache maintenance operations as requested by the alternate master. Matching cache lines can be invalidated during the alternate master access to memory, or memory can be inhibited to allow the MC68040 to respond to the access as a slave. By intervening in the access, the processor can update its internal caches for an external write (sink data) or supply cache data to the alternate bus master for an external read (source data). In this manner, the caches in the MC68040 are prevented from accumulating old or invalid copies of data (stale data), and external masters are allowed access to locally modified data

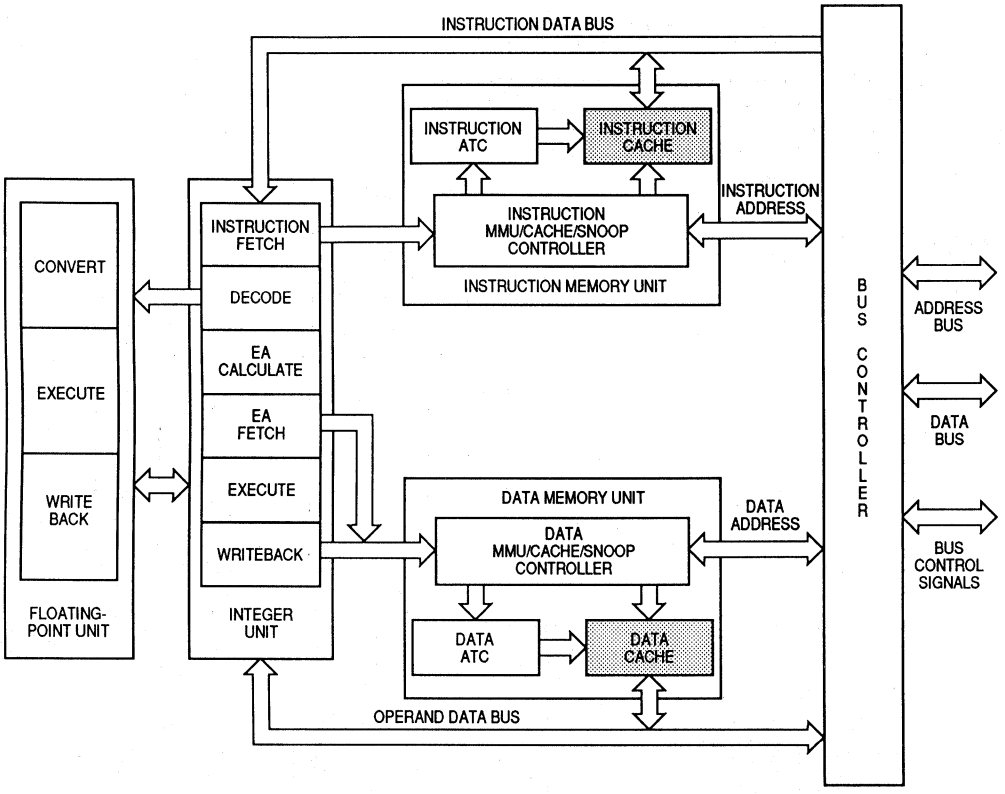


Figure 7-1. Overview of Internal Caches

within the caches that is no longer consistent with external memory (dirty data). Cache coherency is also supported by allowing memory pages to be specified as writethrough instead of copyback; processor writes to writethrough pages always update external memory via an external bus access after updating the cache, keeping memory and cache data consistent.

7.1 CACHE ORGANIZATION

Both four-way set-associative caches have 64 sets of four, 16-byte lines. Each cache line contains an address tag (TAG), status information, and four long words of data (D0–D3) (see Figure 7-2). The address tag contains the upper 22 bits of the physical address. The status information for the instruction cache consists of a single valid bit for the entire line. The status information for the data cache contains a valid bit, as well as four additional bits to indicate dirty status for each long word in the line. Since entry validity is provided

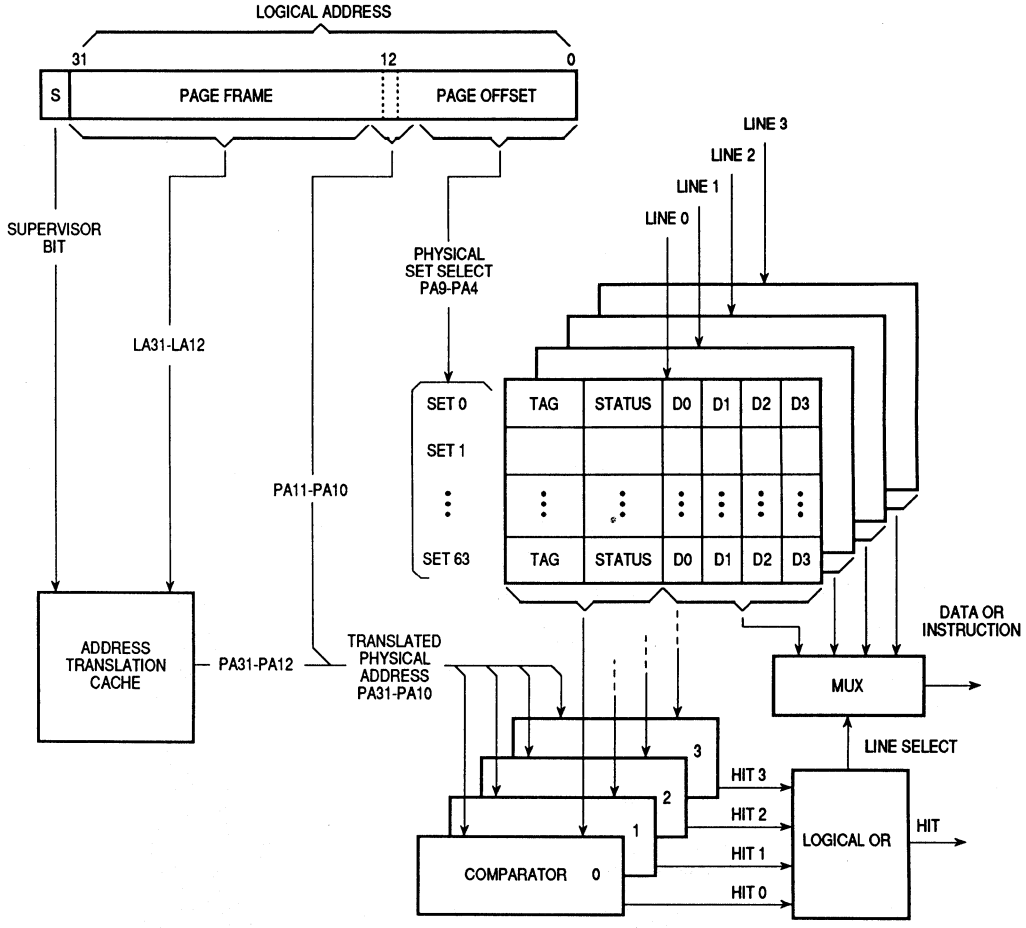


Figure 7-2. Internal Caches

only on a line basis, an entire line must be loaded from system memory in order for the cache to store an entry. Only burst mode accesses that successfully read four long words can be cached. Memory devices unable to support bursting can respond to line read or write accesses by asserting $\overline{\text{TBI}}$ (transfer burst inhibit), forcing the processor to complete the access as a sequence of long-word accesses.

Each memory unit access by the integer unit is translated from a logical address to a physical address to access the data in the cache. To minimize latency of the requested data, the lower untranslated bits of the logical address (which map directly to physical address bits) are used to access a set

of cache lines in parallel with the translation of the upper logical address bits. Bits PA9–PA4 are used to index into the cache and select one of the 64 sets of four lines. The four tags from the selected cache set are compared against the translated physical address bits PA31–PA12 from the MMU and bits PA11 and PA10 of the untranslated page offset. If any one of the four tags match and the tag status is either valid or dirty, then the cache has a hit. During read accesses, half of a line is accessed at a time, requiring two cache accesses for reads which cross a half-line boundary. Write accesses within a cache line require a single cache access.

7.2 CACHING MODES

Every cache access by the integer unit has an associated caching mode that determines how the access will be handled by the cache. The caching mode is normally specified on a page basis by the cache mode (CM) bits in the ATC entry or transparent translation register (TTR) corresponding to the logical address of the access. The CM bits select one of the following four modes: 1) cachable, writethrough, 2) cachable, copyback, 3) noncachable, or 4) special access. (If memory management is disabled, the default caching mode is writethrough.) In addition, some instructions and integer unit operations perform data accesses that have an implicit caching mode associated with them. The following paragraphs discuss the different caching modes in more detail.

7.2.1 Cachable, Writethrough Mode

Data accesses to pages specified as writethrough are always written to the external target address (although the cycle may be buffered), keeping memory and cache data consistent. Cache coherency for shared memory areas in a multiprocessing environment can be maintained by specifying writethrough mode for the shared pages. Writes in writethrough mode are handled with a no-writeallocate policy (i.e., writes that miss in the data cache are written to memory but do not cause the corresponding line in memory to be loaded into the cache).

Instruction or data read accesses which hit in the appropriate cache are supplied data by the cache; misses cause a new cache line to be loaded into the cache, replacing a cache line if necessary. Since instruction cache accesses are always reads, they are not affected by the selection of writethrough or copyback caching mode.

7.2.2 Cachable, Copyback Mode

Copyback pages used to minimize the bus bandwidth used by the processor are typically used for local data structures or stacks. A write hit updates the cache line, setting the dirty bits of any affected long words without an external write access. A write miss causes the needed cache line to be read from memory into the cache where the line is updated, setting the appropriate dirty bits. Read accesses are handled the same as those for writethrough mode — a read hit is supplied data by the cache, while a read miss causes a new cache line to be read in from memory.

For pages designated as copyback, writes will cause lines in the data cache to contain dirty data which has been locally modified and is no longer consistent with memory. When a miss causes one of these dirty cache lines to be selected for replacement, the line will be placed in an internal copyback buffer. The replacement line will be read into the cache, and memory will be updated by writing the dirty cache line back to memory.

7.2.3 Noncachable Mode

Regions of the address space containing noncachable targets, such as I/O devices and shared data structures in multiprocessing systems, can be designated noncachable. Accesses to a specific memory page may be explicitly identified as noncachable by setting the CM bits of the associated ATC entry or TTR to select either the noncachable or serialized noncachable caching modes. Cache operation is identical for both noncachable modes. Noncachable accesses that miss in the cache bypass the cache and do not allocate reads or writes in the cache. Accesses which hit in the cache invalidate a matching valid entry and force a matching dirty entry to be pushed to memory before the external access occurs.

Regardless of the selected cache mode, locked accesses are implicitly considered noncachable. Locked accesses are used by the TAS, CAS, and CAS2 instructions to access operands in memory, and to update translation table entries during table search operations.

7.2.4 Special Accesses

In addition to operations that have an implied noncachable access mode (locked instructions and table search operations), several other processor operations result in accesses with special caching characteristics.

Exception stack accesses, exception vector fetches, and table searches do not allocate entries in the data cache. These accesses (read or write) that miss in the cache do not allocate entries to prevent replacement of a cache line. Cache hits by these accesses are handled in the normal manner according to the caching mode specified for the access address.

Accesses by the MOVE16 instruction do not allocate entries in the data cache for either read or write misses. Read hits on either valid or dirty cache lines are read from the cache, and write hits invalidate a matching entry and perform an external access. By interacting with the cache in this manner, a large block move or block initialization implemented with MOVE16 is prevented from being cached since the data may not be needed immediately.

7.3 CACHE COHERENCY

Several different mechanisms are provided by the MC68040 to assist in maintaining cache coherency in multimaster systems. Both writethrough and copyback memory update techniques are supported to maintain coherency between the data cache and memory. For writethrough accesses, the cache controller always writes to both the data cache (for accesses which hit) and main memory, ensuring that cache data is always consistent with memory. In copyback accesses, the cache controller writes the data into the cache and sets the dirty bits for the affected entries, without performing an external access. The dirty cache data is only written to memory if 1) the line is replaced due to a miss, 2) a noncachable access matches the line, or 3) the line is explicitly pushed by the CPUSH instruction. Use of copyback pages minimizes external bus usage and reduces the latency of write accesses by the processor.

Accesses by an alternate bus master can reference data that is cached by the MC68040, causing coherency problems if the accesses are not handled appropriately. The MC68040 can watch the external processor bus during bus transfers by other masters (bus snooping), and can update its internal caches if a write access hits or can intervene in the access to supply dirty data if a read access hits. Snooping is controlled by the external bus master, indicating via the snoop control signals which accesses are to be snooped and the required operation for snoop hits. Table 7-1 shows the requested snoop operation for each encoding of the snoop control signals. Since the processor and the bus snooper must both access the caches, the snoop controller has priority over the processor for snoopable accesses to maintain cache coherency.

Table 7-1. Snoop Control Encoding

SC1	SC0	Requested Snoop Operation	
		Read Access	Write Access
0	0	Inhibit Snooping	Inhibit Snooping
0	1	Supply Dirty Data and Leave Dirty Data	Sink Byte/Word/Long-Word
1	0	Supply Dirty Data and Mark Line Invalid	Invalidate Line
1	1	Reserved (Snoop Inhibited)	Reserved (Snoop Inhibited)

The snooping protocol and caching mechanism supported by the MC68040 are optimized to support multimaster systems with the MC68040 as the single caching master. In systems implementing multiple MC68040s as bus masters, global data should be stored in writethrough pages. This procedure allows each processor to cache global data for read access while forcing a write to global data by any processor to appear as an external write to memory, which can be snooped by the other processors.

If shared data is stored in copyback pages, only one processor at a time can cache the data (since writes to copyback pages do not access the external bus). If a processor accesses shared data cached by another processor, the slave can source the data to the master without invalidating its own copy only if the transfer to the master is cache inhibited. In order for the master processor to cache the data it must force invalidation of the slave processor's copy of the data (by specifying mark invalid for the snoop operation), and the memory controller must monitor the data transfer between the processors and update memory with the transferred data. The memory update is required since the master processor is unaware of the source of the data (valid data from memory or dirty data from a snooping processor) and initially creates a valid cache line, losing dirty status if the data was supplied by a snooping processor.

Coherency between the instruction cache and the data cache must be maintained in software since the instruction cache does not monitor data accesses. Processor writes that modify code segments (i.e., resulting from self-modifying code or from code executed to load a new page from disk) access memory through the data memory unit. Because these data accesses are not monitored by the instruction cache stale data occurs in the instruction cache if the corresponding data in memory is modified. This coherency problem can be prevented by invalidating any lines in the instruction cache before writing to the corresponding lines in memory.

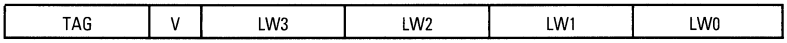
Another potential coherency problem exists due to the relationship between the cache state information and the translation table descriptors. Because each cache line reflects page state information, a page should be flushed from the caches before any of the page attributes are changed. The presence of a valid or dirty cache line implicitly indicates that accesses to the page containing the line are cachable. Presence of a dirty cache line also implies that the page is not write protected and that writes to the page are in copyback mode. Changing page attributes without flushing the corresponding page from the caches is considered a system programming error, which results in cache line states inconsistent with their page definitions. Even with these inconsistencies, the cache is defined and predictable.

7.4 CACHE OPERATION

The instruction and data caches function independently in servicing access requests from the integer unit. The following paragraphs discuss the operational details for the caches and present state diagrams depicting the state transitions for the cache lines. In general, a cache line is always in one of three states: INVALID, VALID, or DIRTY (capitalization indicates these are specifically line states). For invalid lines, the valid bit is clear, and the cache line is ignored during cache lookups. Valid lines have their valid bit set, dirty bits cleared, and all four long words in the line contain valid data consistent with memory. Dirty cache lines have the valid bit and one or more dirty bits set, indicating that the line is valid and contains long-word entries that have not been written to memory (long words whose dirty bit is set). Dirty cache lines are supported only by the data cache.

7.4.1 Instruction Cache

When enabled, the instruction cache is used to store instruction prefetches (instruction words and extension words) as they are requested by the integer unit. Instruction prefetches are normally requested from sequential memory locations except when a change of program flow occurs (e.g., a branch taken) or when an instruction is executed which can modify the status register (SR), in which case the instruction pipe is automatically flushed and refilled. Each instruction cache line consists of a tag, a single valid bit, and four long words (128 bits) of data (see Figure 7-3). The instruction cache supports a line-based protocol that allows individual cache lines to be in either the INVALID or VALID states.



TAG—22-Bit Physical Address Tag Information
V—Line VALID Bit
LWn—32-Bit Data Entry

Figure 7-3. Organization of Instruction Cache Line

For prefetch requests that hit in the cache, the cache half-line (two long words) selected by address bit PA3 is multiplexed onto the internal instruction data bus. When an access misses in the cache, the cache controller requests the memory line containing the required data from memory and places the line in the cache. If available, an invalid line in the selected set is updated with the tag address and data from memory, and the line transitions from the INVALID state to the VALID state by setting the valid bit. If all lines in the set are already valid, a pseudo-random replacement technique is used to select one of the four lines and replace the tag and data contents of the line with the new line information. Refer to **7.4.3 Line Replacement Algorithm** for further information.

A cache line transitions from VALID to INVALID if the cache line is explicitly invalidated by execution of the CINV or CPUSH instructions, if a snooped write access hits the cache line, or if the snoop control signals for a snooped read access indicate to invalidate the line. Both caches should be explicitly cleared after a hardware reset of the processor since the cache lines are not invalidated by the reset.

The state diagram in Figure 7-4 shows the instruction-cache-line state transitions resulting from CPU or snoop controller accesses. Table 7-2 lists the possible cache access cases and the resulting cache operation.

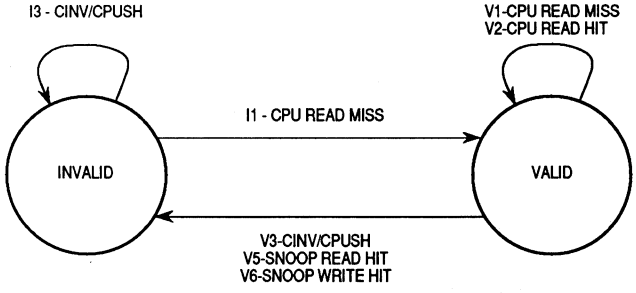


Figure 7-4. Instruction-Cache-Line State Diagram

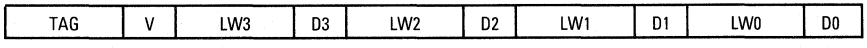
Table 7-2. Instruction-Cache-Line State Transitions

Cache Operation	Current State	
	Invalid	Valid
CPU Read Miss	Read Line from Memory [I1] Supply Data to CPU and Update Cache Go to VALID	Read Line from Memory [V1] Supply Data to CPU and Update Cache (Replace old line) Remain in Current State
CPU Read Hit	Not Possible [I2]	Supply Data to CPU [V2] Remain in Current State
Cache Invalidate or Push	No Action [I3] Remain in Current State	No Action [V3] Go to INVALID
Alternate Master Read Hit (Snoop Control = 01 — Leave Dirty)	Not Possible [I4] (Not Snooped)	Not Possible [V4] (Not Snooped)
Alternate Master Read Hit (Snoop Control = 10 — Invalidate)	Not Possible [I5]	No Action [V5] Go to INVALID
Alternate Master Write Hit (Snoop Control = 01 — Leave Dirty or Snoop Control = 10 — Invalidate)	Not Possible [I6]	No Action [V6] Go to INVALID

7

7.4.2 Data Cache

The data cache is used to cache operand accesses generated by the integer unit, and supports a line-based protocol allowing individual cache lines to be in one of three states: INVALID, VALID, or DIRTY. Figure 7-5 shows the tag, status, and data information contained in each data cache line as well as the equations defining the line state for the status bit combinations. The data cache supports both writethrough and copyback modes (specified by the caching mode bits for the page) to maintain coherency with memory.



TAG—22-Bit Physical Address Tag
V—Line VALID Bit
LWn—32-Bit Data Entry
Dn—DIRTY Bit for Long-Word n

$INVALID = \overline{V}$
$VALID = V \text{ and } (\overline{D3 + D2 + D1 + D0})$
$DIRTY = V \text{ and } (D3 + D2 + D1 + D0)$

Figure 7-5. Data Cache Line Organization

Read misses and write misses to copyback pages cause the cache controller to read a new cache line from memory into the cache. If available, an invalid line in the selected set is updated with the tag address and data from memory, and the line transitions from the INVALID state to the VALID state by setting the valid bit for the line. If all lines in the set are already valid or dirty, a pseudo-random replacement technique is used to select one of the four lines and replace the tag and data contents of the line with the new line information. Before replacement, dirty lines are temporarily buffered and later copied back to memory after the new line has been read from memory. If a snoop access occurs before the buffered line is written to memory, the snoop controller will snoop the buffer in addition to the caches.

The cache protocol for each processor and snooped access type is described in the following paragraphs. In all cases an external bus transfer will cause a line state transition only if the bus transfer is marked as "snoopable" on the bus. The protocols described in the following paragraphs assume that the data is cachable.

7.4.2.1 READ MISS. A processor read which misses in the cache causes the cache to request a bus transaction to read the needed line from main memory and supply the required data to the integer unit. The line is placed in the cache in the VALID state.

Snooped external reads which miss in the cache have no effect on the cache.

7.4.2.2 WRITE MISS. Processor writes which miss in the cache are handled based on the selected caching mode. Writes to a copyback page cause the processor to perform a bus transaction to get the needed cache line into its cache (in the same manner as for a read miss). The new cache line is then updated with the write data, and the dirty bits are set for each long word that has been modified, leaving the cache line in the DIRTY state. Write misses to writethrough pages do not allocate in the cache; the data is written to memory without loading the corresponding cache line.

Snooped external writes that miss in the cache have no effect on the cache.

7.4.2.3 READ HIT. Regardless of whether the page write mode is writethrough or copyback, data for a processor read that hits in the cache is supplied by the cache. No bus transaction is performed, and the state of the cache line does

not change. Physical address bit PA3 selects the upper or lower 64 bits (half-line) of the line containing the required operand; this half-line is driven onto the internal bus. If the required data resides entirely within the half-line, only one access into the cache is required. Because the organization of the cache does not allow selection of more than one half-line at a time, misalignment across a half-line boundary requires two accesses into the cache.

A snooped external read that hits in the cache is ignored if the cache line is valid. If the snooped access hits a dirty line, memory is inhibited from responding, and the data is sourced from the cache directly to the alternate bus master. A snooped read hit does not change the state of the cache line unless the snooped access also indicates mark INVALID, which causes the line to be invalidated after the access (even if dirty). Alternate bus masters should indicate mark INVALID only for line reads to ensure the entire line is transferred before invalidating.

7.4.2.4 WRITE HIT. Processor writes that hit in the cache are handled differently for writethrough and copyback pages. For writethrough accesses, a processor write hit causes the cache controller to update the affected long-word entries in the cache line and to request an external memory write transfer to update memory. The cache line state is not changed. Although dirty bits for the line are also unchanged, a writethrough access to a line containing dirty data constitutes a system programming error. This situation can be avoided by pushing cache entries when a page descriptor is changed and by ensuring that other bus masters indicate the appropriate snoop operation for writes to corresponding pages (i.e., mark invalid for writethrough pages and sink data for copyback pages).

If the access is marked as copyback, the cache controller updates the cache line and sets the dirty bit of the appropriate long words in the cache line. An external write is not performed, and the cache line transitions to (or remains in) the DIRTY state.

An alternate bus master can drive the snoop control signals for a write with an encoding that indicates to the MC68040 that it should sink the data (inhibit memory and respond as a slave) if the access hits in the data cache. Operation of the cache depends on the access size and current line state. A snooped line write or a snooped write that hits a valid line always causes the corresponding cache line to be invalidated. For snooped writes of byte, word, or long-word size that hit a DIRTY line, the processor inhibits memory and

responds to the alternate bus master as a slave, sinking the write data. Data received from the alternate bus master is written to the appropriate long word in the cache line, and the dirty bit is set for that entry.

For a snooped write in which the snoop control pins indicate that a matching cache line should be marked invalid, the line will be invalidated.

7.4.2.5 PROTOCOL STATE DIAGRAM. The state diagram in Figure 7-6 shows the three possible states for a data cache line, with the possible transitions caused by either processor accesses or snooped accesses. Transitions are labeled with a capital letter indicating the previous state followed by a number indicating the specific case. Table 7-3 shows the three states and all transitions between those states from both the processor and the bus.

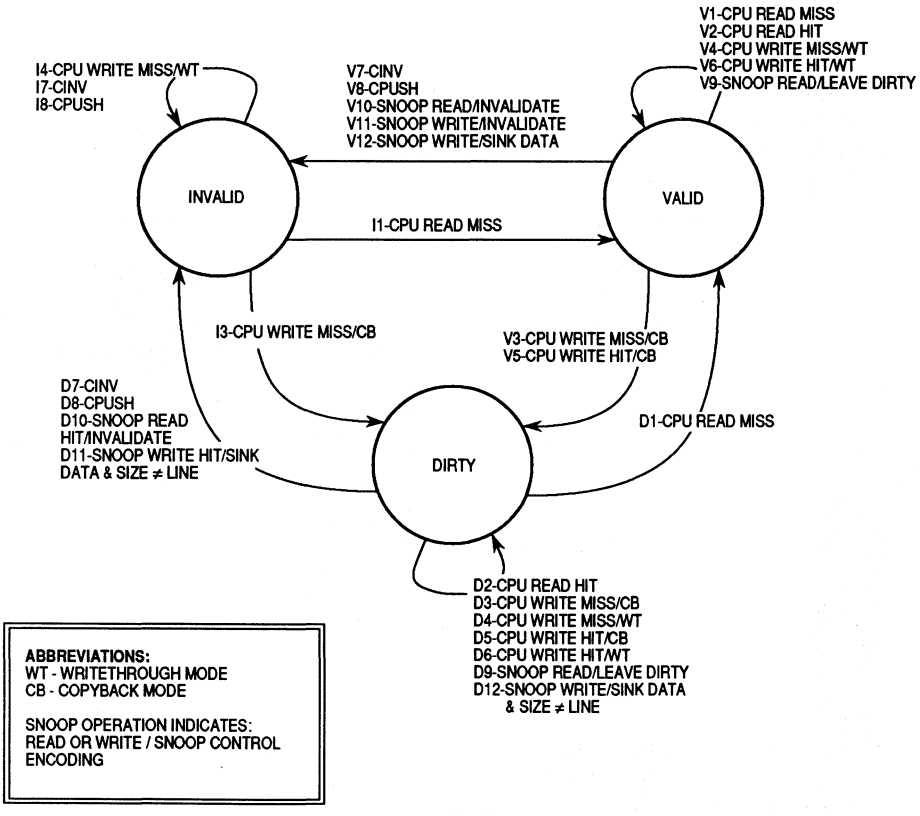


Figure 7-6. Data-Cache-Line State Diagram

Table 7-3. Data-Cache-Line State Transitions

Cache Operation	Current State		
	Invalid	Valid	Dirty
CPU Read Miss	Read Line from Memory I1 Supply Data to CPU and Update Cache Go to VALID	Read Line from Memory V2 Supply Data to CPU and Update Cache (line replaced in cache) Remain in current state	Buffer Dirty Cache Line Read New Line from Memory D1 Supply Data to CPU and Update Cache Write Buffered Dirty Data to Memory Go to VALID
CPU Read Hit	Not Possible I2	Supply Data to CPU Remain in Current State V2	Supply Data to CPU Remain in Current State D2
CPU Write Miss Cache Mode = Copyback	Read Line from Memory into Cache I3 Write Data to Cache Set Dirty Bits of Modified Long Words Go to DIRTY	Read Line from Memory into Cache V3 (line replaced in cache) Write Data to Cache and Set Dirty Bits Go to DIRTY	Buffer Dirty Cache Line Read New Line from Memory D3 Write Data to Cache and Set Dirty Bits Write Buffered Dirty Data to Memory Remain in Current State
CPU Write Miss Cache Mode = Writethrough	Write Data to Memory I4 Remain in Current State	Write Data to Memory V4 Remain in Current State	Write Data to Memory Remain in Current State (See Note) D4
CPU Write Hit Cache Mode = Copyback	Not Possible I5	Write Data into Cache V5 Set Dirty Bits of Modified Long Words Go to DIRTY	Write Data into Cache Set Dirty Bits of Modified Long Words D5 Remain in Current State
CPU Write Hit Cache Mode = Writethrough	Not Possible I6	Write Data to Cache Write Data to Memory V6 Remain in Current State	Write Data into Cache (No Change to Dirty Bits) D6 Write Data to Memory Remain in Current State (See Note)
Cache Invalidate	No Action I7 Remain in Current State	No Action V7 Go to INVALID	No Action Dirty Data Lost) D7 Go to INVALID
Cache Push	No Action I8 Remain in Current State	No Action V8 Go to INVALID	Write Dirty Data to Memory D8 Go to INVALID
Alternate Master Read Hit (Snoop Control = 01 — Leave Dirty)	Not Possible I9	No Action V9 Remain in Current State	Inhibit Memory and Source Data D9 Remain in Current State
Alternate Master Read Hit (Snoop Control = 10 — Invalidate)	Not Possible I10	No Action V10 Go to INVALID	Inhibit Memory and Source Data D10 Go to INVALID
Alternate Master Write Hit (Snoop Control = 10 — Invalidate or Size = Line)	Not Possible I11	No Action V11 Go to INVALID	No Action D11 Go to INVALID
Alternate Master Write Hit (Snoop Control = 01 — Sink Data and Size ≠ Line)	Not Possible I12	No Action V12 Go to INVALID	Inhibit Memory and Sink Data D12 Set Dirty Bits of Modified Long Words Remain in Current State

NOTE: While technically valid, DIRTY state transitions D4 and D6 are the result of a system programming error and should be avoided.

7.4.3 Line Replacement Algorithm

Both caches contain circuitry to automatically determine which cache line in a set to use for a new line. The algorithm operates as follows: locate the first invalid entry and use it; if no invalid entries are found, use a pseudo-random algorithm to select a valid entry and replace that entry.

To implement this replacement algorithm, each cache contains a 2-bit counter which is incremented for each access to the cache. The counter in the instruction cache is incremented once for each half-line accessed in the instruction cache. The counter in the data cache is incremented for each half-line accessed during reads, for each line accessed for writes in copyback mode, and for each bus transfer resulting from a write in writethrough mode. When a miss occurs and all four lines in the set are valid, the line pointed to by the current counter value is replaced, after which the counter is incremented.

7.4.4 Memory Accesses for Cache Maintenance

The cache controller in each memory unit performs all maintenance activity associated with supplying data from the cache to the execution units. This activity includes requesting accesses by the bus interface unit to read new lines and to write dirty cache data to memory when replacing lines. The following paragraphs describe the memory accesses resulting from cache fill operations (by both caches) and push operations (by the data cache). Refer to **SECTION 8 BUS OPERATION** for detailed information about the bus cycles required.

7.4.4.1 CACHE FILLING. When a new cache line is required due to a cachable read miss or write miss (in copyback mode), the cache controller requests a line read from the bus controller. The bus controller requests a burst read transfer by indicating a line access with the size signals (SIZ1,SIZ0) and indicates which line in the set is being loaded with the transfer line number signals (TLN1,TLN0). The responding device sequentially supplies four long words of data and may assert the transfer cache inhibit signal ($\overline{\text{TCI}}$) if the line is not cachable. If the responding device does not support the burst mode, it should assert the transfer burst inhibit signal ($\overline{\text{TBI}}$) for the first long word of the line access. The bus controller responds by terminating the line access and completes the remainder of the line read as three, sequential, long-word reads.

Line accesses by the bus controller implicitly request 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 as described in **SECTION 8 BUS OPERATION**. The device indicates its ability to support the burst access by acknowledging the initial long-word transfer with transfer acknowledge ($\overline{TA}$) asserted and $\overline{TBI}$ negated. This procedure causes the processor to continue to drive the address and bus control signals and to latch a new data value for the cache line at the completion of each subsequent cycle (as defined by $\overline{TA}$) for a total of four cycles. The bursting mechanism requires addresses to wrap around so that the entire four long words in the cache line are filled in a single operation.

When a cache line read is initiated, the first cycle attempts to load the cache entry corresponding to the instruction word or data item explicitly requested by the integer unit. The subsequent transfers are for the remaining entries in the cache line. In the case of a misaligned access in which 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.

The data from each cycle is temporarily stored by the cache controller in a 128-bit buffer, where it is immediately available to the integer unit. If a misaligned access spans two entries in the line, the second portion of the operand is available to the integer unit as soon as the second memory cycle completes. A new access by the integer unit that hits the cache line being filled is also supplied data as soon as the required long word has been received from the bus controller. During the period required to fill the buffer, other integer unit accesses that hit in the cache are supplied data.

The assertion of $\overline{TCI}$ during the first cycle of a burst read operation inhibits loading of the buffered line into the cache, but it does not cause the burst transfer (or pseudo-burst transfer if $\overline{TBI}$ is asserted with $\overline{TCI}$) to be terminated early. The data placed in the buffer is accessible by the integer unit until the last long word of the burst is transferred from the bus controller, after which the contents of the buffer are invalidated without being copied into the cache. The assertion of $\overline{TCI}$ is ignored during the second, third, or fourth cycle of a burst operation.

A bus error occurring during a burst operation causes the burst operation to abort. If the bus error occurs during the first cycle of a burst, the data from the bus is ignored. If the access is a data cycle, exception processing proceeds immediately. If the cycle is for an instruction prefetch, a bus error exception

is pending. The bus error is processed only if the integer unit attempts to use either instruction word. Refer to **SECTION 8 BUS OPERATION** for more information about pipeline operation.

For either cache, when a bus error occurs on the second cycle or later, the burst operation is aborted and the line buffer is invalidated. The processor may or may not take an exception, depending on the status of the pending data request. If the bus error cycle contains a portion of a data operand that the processor is specifically waiting for (e.g., the second half of a misaligned operand), the processor immediately takes an exception. Otherwise, no exception occurs, and the cache line fill is repeated the next time data within the line is required. In the case of an instruction cache line fill, the data from the aborted cycle is completely ignored.

On the initial access of a line read, a 'retry' (indicated by the assertion of $\overline{TA}$ and $\overline{TEA}$) causes the bus controller to retry the bus cycle. However, a retry signaled during the remaining cycles of the line access (either burst or pseudo-burst) is recognized as a bus error and is handled by the processor as described in the previous paragraphs.

A cache inhibit or bus error on a line read can change the state of the line being replaced, even though the new line is not copied into the cache. Before loading a new line, the cache line being replaced is copied to the push buffer if it is dirty, then the cache line is invalidated. If a cache inhibit or bus error occurs on a replacement line read a dirty line is restored to the cache from the push buffer. However, the line being replaced is not restored in the cache if it was originally valid and the cache line remains invalid. If the line read resulting from a write miss in copyback mode is cache inhibited, the write access misses in the cache and writes through to memory.

7.4.4.2 CACHE PUSHES. Dirty data cache lines are copied back to memory when selected by the cache controller for replacement by a new line, when explicitly selected by execution of a CPUSH instruction, and when hit by a noncachable access. When a dirty data cache line is selected for replacement, memory must be updated with the dirty data before the line is replaced. To reduce latency of the requested data in the new line, the dirty line being replaced is temporarily placed in a push buffer while the new line is fetched from memory. While a line resides in the push buffer, it can be snooped by an external bus master but can not be accessed by the execution units. After the bus transfer for the new line is successfully completed, the dirty cache line is copied back to memory, and the push buffer is invalidated. If the

operation to access the replacement line is abnormally terminated or signaled as noncachable, the line in the push buffer is copied back into its original position in the cache, and the processor continues operation as described in the previous paragraphs.

The size of the push transfer on the bus is determined by the number of dirty entries in the line to be pushed, minimizing bus bandwidth required for the push. If a single entry is dirty, that entry is written to memory using a long-word push transfer. A push transfer is distinguished from a normal write transfer by an encoding of '000' on the transfer modifier signals (TM2–TM0) for the push. The transfer can be retried by asserting $\overline{TA}$ and $\overline{TEA}$ or terminated by a bus error asserted $\overline{TEA}$. If a push transfer is terminated by bus error, an exception is immediately taken by the processor.

A line containing two or more dirty entries is copied back to memory using a line push transfer. For a line push, the bus controller requests a burst write transfer by indicating a line access with the size signals (SIZ1–SIZ0). The responding device sequentially accepts four long words of data. If the responding device does not support the burst mode, it should assert $\overline{TBI}$ for the first long word of the line access. The bus controller responds by terminating the line access and completes the remainder of the line push as three, sequential, long-word writes. The first cycle of the burst can be retried, but a retry for any of the three remaining cycles is interpreted by the bus controller as a bus error. If a bus error occurs in any cycle in the line push transfer, an exception is immediately taken by the processor.

A dirty cache line that is hit by a noncachable access is pushed before the external bus access occurs. If the access is part of a locked transfer sequence for TAS, CAS, or CAS2 operand accesses or translation table updates, the $\overline{LOCK}$ signal is also asserted for the push access

7.5 CACHE CONTROL AND MAINTENANCE

The caches are individually enabled using the MOVEC instruction to access the 32-bit cache control register (CACR) shown in Figure 7-7. The CACR contains two enable bits that allow the instruction and data caches to be independently enabled or disabled. Setting an enable bit enables the associated cache without affecting the state of any lines within the cache. A hardware reset clears the CACR, disabling both caches; however, the tags, state information, and data within the caches are not affected by reset and must be cleared by using the CINV instruction before enabling the caches.

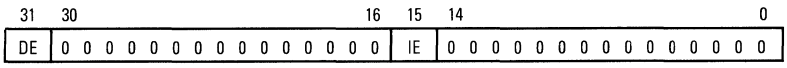


Figure 7-7. Cache Control Register

System hardware can assert the cache disable ($\overline{\text{CDIS}}$) signal to dynamically disable both caches, regardless of the state of the enable bits in the CACR. The caches are disabled immediately after the current access completes. If $\overline{\text{CDIS}}$ is asserted during the access for the first half of a misaligned operand spanning two cache lines, the data cache is disabled for the second half of the operand. Accesses by the execution units bypass the caches while they are disabled and do not affect their contents (with the exception of CINV and CPUSH instructions). Disabling the caches with $\overline{\text{CDIS}}$ does not affect snoop operations. $\overline{\text{CDIS}}$ is intended primarily for use by in-circuit emulators to allow swapping between the tags and emulator memories.

Cache management in the supervisor mode is supported by the CINV and CPUSH instructions. CINV allows selective invalidation of cache entries. CPUSH performs two operations: first, any selected data cache lines containing dirty data are pushed to memory; then, all selected cache lines are invalidated. This operation can be used to update a page in memory before swapping it out with snooping disabled or to push dirty data when changing a page caching mode to writethrough. Because of the size of the caches, pushing pages or an entire cache will incur a significant time penalty. Operation of CINV and CPUSH is not affected by the state of the $\overline{\text{CDIS}}$ signal or the cache enable bits in the CACR. Both instructions allow operation on a single cache line, all cache lines in a specific page, or an entire cache, and can select one or both caches for the operation. For line and page operations, the memory address is specified by a physical address in an address register.

SECTION 8
BUS OPERATION

The MC68040 bus interface supports synchronous data transfers between the processor and other devices in the system. This section provides a functional description of the bus, the signals that control the bus, and the bus cycles provided for data transfer operations. Operation of the bus is defined for transfers initiated by the processor as a bus master, and for transfers initiated by an alternate master that are snooped by the processor as a slave device. Descriptions of the error and halt conditions, bus arbitration, and the reset operation are also included. For exact timing specifications, refer to **SECTION 11 ELECTRICAL CHARACTERISTICS**.

Access requests by the processor and other potential bus masters in the system are arbitrated by an external arbiter that prioritizes the requests and determines which device is granted access to the bus. When the MC68040 is the bus master, it uses the bus to access instructions and data from memory which are not contained in its internal caches, and to write data to memory. Additional bus transfers are used to acknowledge interrupts and breakpoints.

Bus accesses by another bus master which has been granted control of the bus are monitored (snooped) by the processor when it is not the bus master to allow the processor to intervene in the access if required. Control inputs to the processor allow external logic to specify the required snoop operation to perform for each bus transfer by an alternate master. The processor allows memory to respond if no external action is required; otherwise, memory is inhibited and the processor responds to the access as a slave, supplying modified data from its data cache or writing data to an already modified cache line (for alternate master reads and writes, respectively). The snooping mechanism is optimized to support cache coherency in multi-master applications in which the MC68040 is the only caching master.

The 32-bit data bus supports byte, word, long-word, and line (16-byte) bus cycles using a handshaked transfer sequence. Line transfers are normally performed using an efficient burst transfer which provides an initial address and time-multiplexes the data bus to transfer four long words of information to or from the slave device. Slave devices which do not support bursting can burst-inhibit a line transfer, forcing the bus master to complete the access

using three additional long word bus cycles. All bus input and output signals are synchronous to the rising edge of the bus clock (BCLK) signal.

The MC68040 architecture supports byte, word, and long-word integer operands, as well as single, double, and extended precision floating point operands; these operands can be located in memory on any byte boundary. Misaligned accesses to the caches are supported with multiplex and alignment logic; misaligned memory accesses are completed by breaking up the access into a sequence of aligned byte or word bus transfers. The user should be aware that operand misalignment causes the MC68040 to perform multiple bus cycles for the operand transfer, and therefore, processor performance is optimized if operands are aligned to their natural boundaries (for example, long-word operands should be on long-word boundaries). Instruction words and their associated extension words must be aligned on word boundaries.

8.1 BUS CHARACTERISTICS

The MC68040 bus is a fully synchronous bus that uses the BCLK signal to clock transfers between the processor and memory devices or another bus master. Byte, word, long-word, and line burst transfers through a 32-bit data port are supported.

Unlike the MC68020 and MC68030 processors, the MC68040 does not support dynamic bus sizing and expects the referenced device to be able to accept the requested access width. Blocks of memory which must be contiguous, such as for code storage or program stacks, must be 32 bits wide. Byte and word sized I/O ports that return an interrupt vector during interrupt acknowledge cycles must be mapped into the low order 8 or 16 bits, respectively, of the data bus in order to pass the vector on data bus bits D0–D7.

The bus transfers information between the MC68040 and an external memory or peripheral device using a fixed 32-bit data port width. 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 MC68040 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 signals used to terminate the cycle.

The MC68040 uses two clocks to generate timing — a bus clock (BCLK) and a processor clock (PCLK). The PCLK signal is exactly twice the frequency of

the BCLK signal and is internally phase-locked to BLCK and distributed throughout the device to generate timing for all logic blocks. The BCLK signal is only used as the reference signal for the phase-lock-loop (PLL) which synchronizes the PCLK. The use of dual clock inputs allows the bus interface to operate at half the speed of the internal logic of the processor, requiring less stringent memory interface requirements. Since the rising edge of BCLK is used as the reference point for the PLL, all timing specs are referenced to this edge.

The general relationship between the clock signals and most input and output signals is shown in Figure 8-1. The rising edge of the internal phase-locked PCLK signal is aligned with the rising edge of BCLK, and the two PCLK cycles corresponding to each BCLK cycle are divided into four states, T1 through T4. Most outputs change during state T4, whether transitioning between a driven and high-impedance state, or switching between high and low logic levels. (The exceptions to this rule are the $\overline{TIP}$, $\overline{TA}$, and $\overline{BB}$ signals, which transition between logic levels during T4, but transition from a driven state to a high impedance state during the T1 state.)

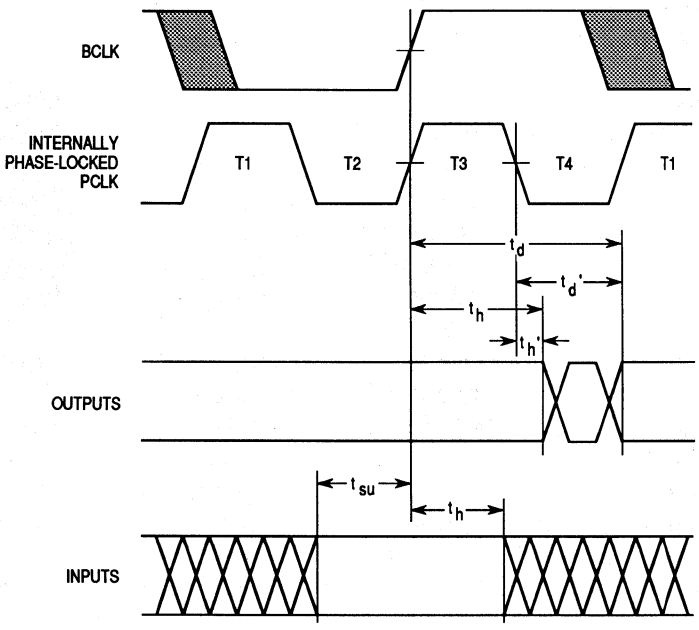


Figure 8-1. Signal Relationships to Clocks

Inputs to the MC68040 (other than the $\overline{IPL2}$–$\overline{IPL2}$ and $\overline{RSTI}$ signals) are synchronously sampled, and must be stable during the sample window defined by the input setup and hold times shown in Figure 8-1 to guarantee proper operation. The asynchronous $\overline{IPLn}$ and $\overline{RSTI}$ signals are also sampled on the rising edge of BCLK, but are internally synchronized to resolve the input to a high or low level before using it.

Since the timing specifications for the MC68040 are referenced to the rising edge of BCLK, they are valid only for the specified operating frequency, and must be scaled for lower operating frequencies. Refer to the MC68040DH/AD, *MC68040 Designer's Handbook* for further information.

8.2 DATA TRANSFER MECHANISM

The MC68040 architecture supports byte, word, long-word, and 16-byte integer operands, and single, double, and extended precision floating-point operands. The processor also supports the emulation of packed decimal real operands by fetching the operand from memory before trapping to the unimplemented data type handler. All operands other than 16-byte can be located on any byte boundary, but misaligned transfers may require additional bus cycles.

The bytes of operands are designated as shown in Figure 8-2. 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. Floating-point operands are handled by the integer unit as a sequence of related long-word operands. These designations are used in the figures and descriptions that follow.

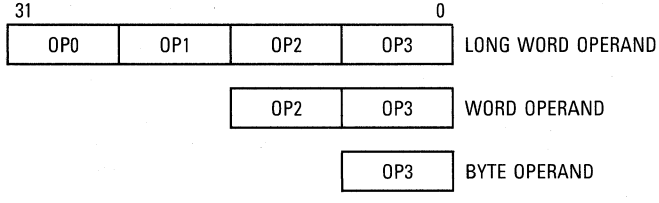


Figure 8-2. Internal Operand Representation

Figure 8-3 shows the general form of the multiplexing between the external bus and an internal register. The four bytes shown in Figure 8-3 are connected through the internal data bus and data multiplexer to the external data bus. This path is the means through which the MC68040 supports operand misalignment. Refer to **8.2.1 Misaligned Operands** for the definition of misaligned operand. The data multiplexer establishes the necessary connections for different combinations of address and data sizes.

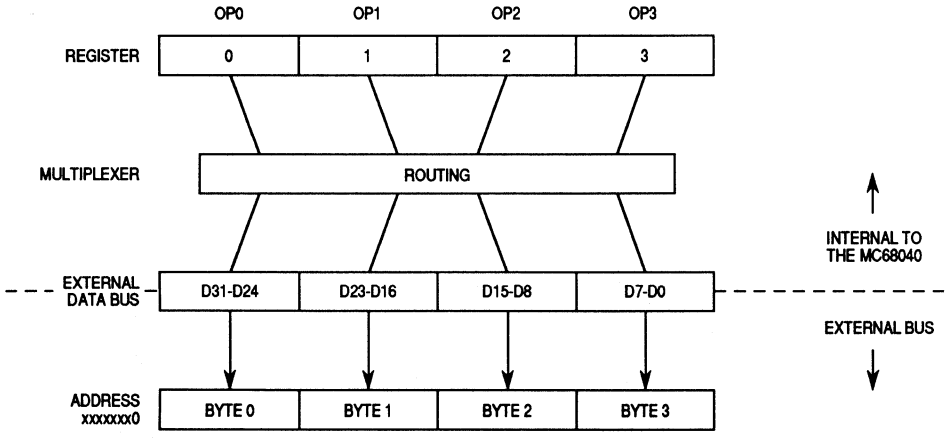


Figure 8-3. Data Multiplexing

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 in order 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 always indicate the number of bytes to be transferred during the current bus cycle, as shown in Table 8-1.

Table 8-1. Size Signal Encoding

SIZ1	SIZ0	Size
0	1	Byte
1	0	Word
0	0	Long Word
1	1	Line

The address lines 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 8-2 shows the encodings of A0 and A1 and the corresponding byte offsets from the long-word base.

Table 8-2. Address Offset Encodings

A1	A0	Size
0	0	+0 Bytes
0	1	+1 Bytes
1	0	+2 Bytes
1	1	+3 Bytes

Table 8-3 lists the valid bytes on the data bus for read and write cycles. 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. For line transfers, all bytes are valid as shown, and may correspond to either portions of the requested operand or to data required to fill the remainder of the cache line. The bytes labeled "—" are not required, and are ignored on read bus cycles and driven with undefined data on write bus cycles.

Table 8-3. Data Bus Requirements for Read and Write Cycles

Transfer Size	SIZ1	SIZ0	A1	A0	Data Bus Active Sections			
					D31:D24	D23:D16	D15:D8	D7:D0
Byte	0	1	0	0	OPn	—	—	—
	0	1	0	1	—	OPn	—	—
	0	1	1	0	—	—	OPn	—
	0	1	1	1	—	—	—	OPn
Word	1	0	0	0	OPn	OPn	—	—
	1	0	1	0	—	—	OPn	OPn
Long Word	0	0	X	X	OPn	OPn	OPn	OPn
Line	1	1	X	X	OPn	OPn	OPn	OPn

Additional information on the encodings for the MC68040 signals can be found in **SECTION 5 SIGNAL DESCRIPTION**. A brief summary of the bus signal encodings for each access type is shown in Table 8-4 below.

Table 8-4. Summary of Access Types versus Bus Signal Encodings

BUS SIGNAL	DATA CACHE PUSH ACCESS	NORMAL DATA/ CODE ACCESS	TABLEWALK ACCESS	MOVE16 ACCESS	ALTERNATE ACCESS	INTERRUPT ACKNOW-LEDGE	BREAKPOINT ACKNOW-LEDGE
A31–A0	Access Address	Access Address	Entry Address	Access Address	Access Address	$FFFFFFFF	$00000000
UPA1:0	$0	MMU Source[1]	$0	MMU Source[1]	$0	$0	$0
SIZ1:0	Line	B/W/L/Line	Long Word	Line	B/W/L	Byte	Byte
TT1:0	$0	$0	$0	$1	$2	$3	$3
TM2:04	$0	$1,2,5, or 6	$3 or 4	$1 or 5	Function Code	Int. Level $1-7	$0
TLN1:0	Cache Set Entry	Cache Set Entry[2]	Undefined	Undefined	Undefined	Undefined	Undefined
R/$\overline{W}$	Write	Read/Write	Read/Write	Read/Write	Read/Write	Read	Read
$\overline{LOCK}$ $\overline{LOCKE}$	Negated	Asserted/ Negated[3]	Asserted/ Negated[3]	Negated	Negated	Negated	Negated
$\overline{CIOUT}$	Negated	MMU Source[1]	Negated	MMU Source[1]	Asserted	Negated	Negate

NOTES:
1) The UPA1, UPA0 and XTO(CIOUT) signals are determined by the U1, U0, and CM bit fields, respectively, in the ATC entry or TT register corresponding to the access address.
2) The $\overline{TLNx}$ signals are defined only for normal push accesses and normal line read accesses.
3) The $\overline{LOCK}$ signal is asserted during TAS, CAS, and CAS2 operand accesses and for some tablewalk update sequences. $\overline{LOCKE}$ is asserted for the last transfer of each locked sequence of transfers.
4) Refer to **SECTION 5 SIGNAL DESCRIPTION** for definitions of the TMn signal encodings for normal, MOVE16, and alternate accesses.

8.2.1 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 MC68040 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.

Misaligned operand accesses that either miss in the data cache or are non-cachable are converted by the data memory unit in the MC68040 to a sequence of aligned accesses. These aligned access requests are sent to the bus controller for completion, resulting in bus transfers which are always aligned. The size indicated on the SIZn signals corresponds to the specific bus cycle, and does not indicate how many bytes may be remaining for the operand transfer.

Figure 8-4 shows the transfer of a long-word operand from an odd address, which requires three bus cycles. For the first cycle, the size signals specify a byte transfer, and the address offset (A2:A0) is 001. The slave device supplies the byte and acknowledges the data transfer. When the processor starts the second cycle, the size signals specify a word transfer 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 a byte transfer. The address offset (A2:A0) is now 100; the port supplies the final byte and the operation is complete. Figure 8-5 shows the associated bus transfer signal timing. For a long-word transfer from an odd-word address, only two word transfers are required.

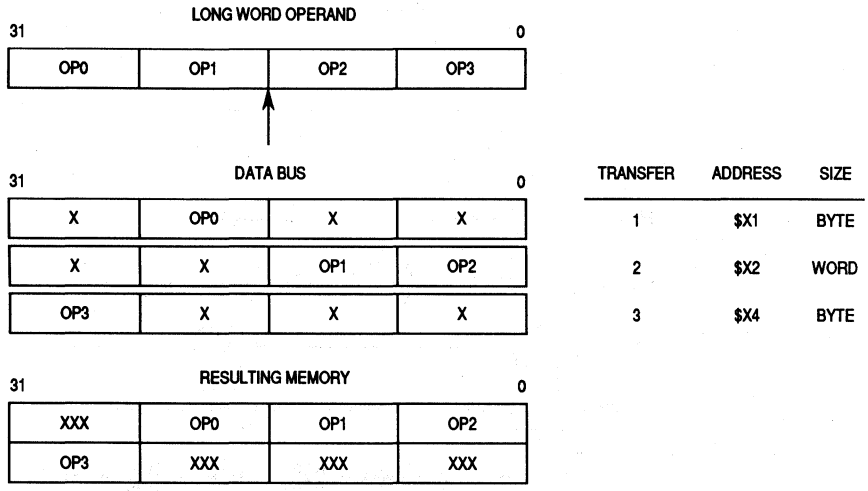


Figure 8-4. Example of a Misaligned Long-Word Read Transfer

Figures 8-6 shows a word transfer to an odd address. This example is similar to the one shown in Figures 8-4 and 8-5 except that the operand is word sized and the transfer requires only two bus cycles.

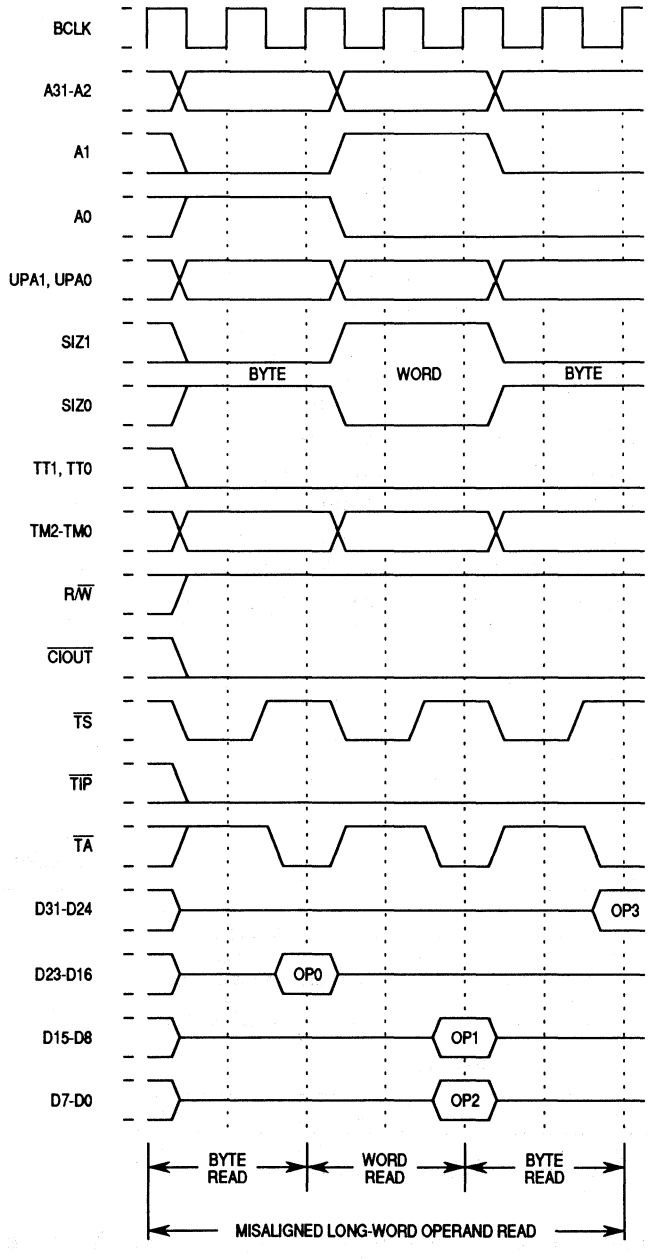


Figure 8-5. Long-Word Operand Read Timing

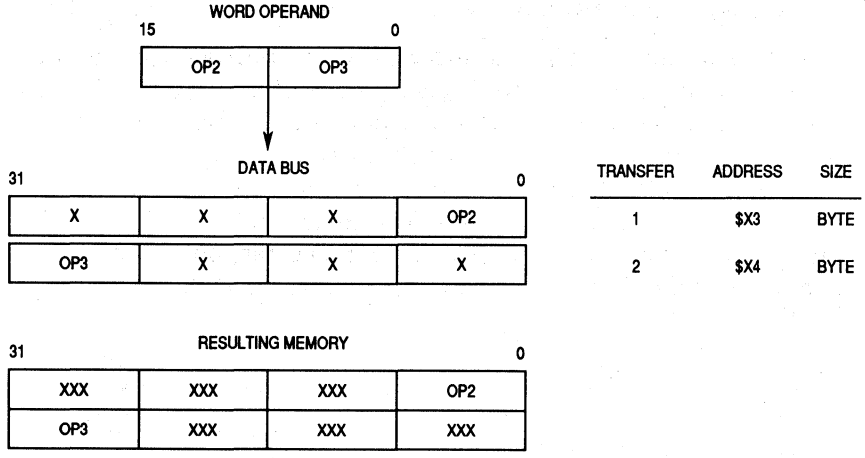


Figure 8-6. Example of a Misaligned Word Write Transfer

The combination of operand size and operand alignment determine the number of bus cycles required to perform a particular memory access. Table 8-5 shows the number of bus cycles required for different operand sizes with all possible alignment conditions for read and write cycles.

8

Table 8-5. Memory Alignment Influence on Noncachable and Writethrough Bus Cycles

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

*Instruction prefetches are always two words from a long word boundary.

This table shows that bus cycle throughput for non-cachable accesses is significantly affected by alignment. The MC68040 system designer and programmer should be aware of and account for this effects, particularly in time-critical applications.

Table 8-5 shows that the processor always prefetches instructions by reading a long word from a long-word address (A1:A0 = 00), regardless of alignment. When the required instruction begins at an odd word boundary, the processor attempts to fetch the entire 32 bits, although the second one is the required word.

8.2.3 Address, Size, and Data Bus Relationships

The data transfer examples show how the MC68040 drives data onto or receives data from the correct byte sections of the data bus. Table 8-6 shows the combinations of the size signals and address signals A0 and A1 that are used to generate byte enable signals for each of the four sections of the data bus for if the addressed device requires them. The four columns on the right correspond to the four byte enable signals. The letter A indicates the data bus section is active, and implies that the corresponding byte enable signal should be true. A hyphen (-) implies that the byte enable signal does not apply.

Table 8-6. Data Bus Byte Enable Signals

Transfer Size	SIZ1	SIZ0	A1	A0	Data Bus Active Sections			
					D31:D24	D23:D16	D15:D8	D7:D0
Byte	0	1	0	0	A	—	—	—
	0	1	0	1	—	A	—	—
	0	1	1	0	—	—	A	—
	0	1	1	1	—	—	—	A
Word	1	0	0	0	A	A	—	—
	1	0	1	0	—	—	A	A
Long Word	0	0	X	X	A	A	A	A
Line	1	1	X	X	A	A	A	A

These enable or strobe signals select only the bytes required for write cycles or for non-cacheable read cycles. The other bytes are not selected, which prevents incorrect accesses in sensitive areas such as I/O.

Figure 8-7 shows a logic diagram for one method for generating byte data enable signals from the size and address encodings and the read/write signal.

NOTE

These select lines can be combined with the address decode logic.

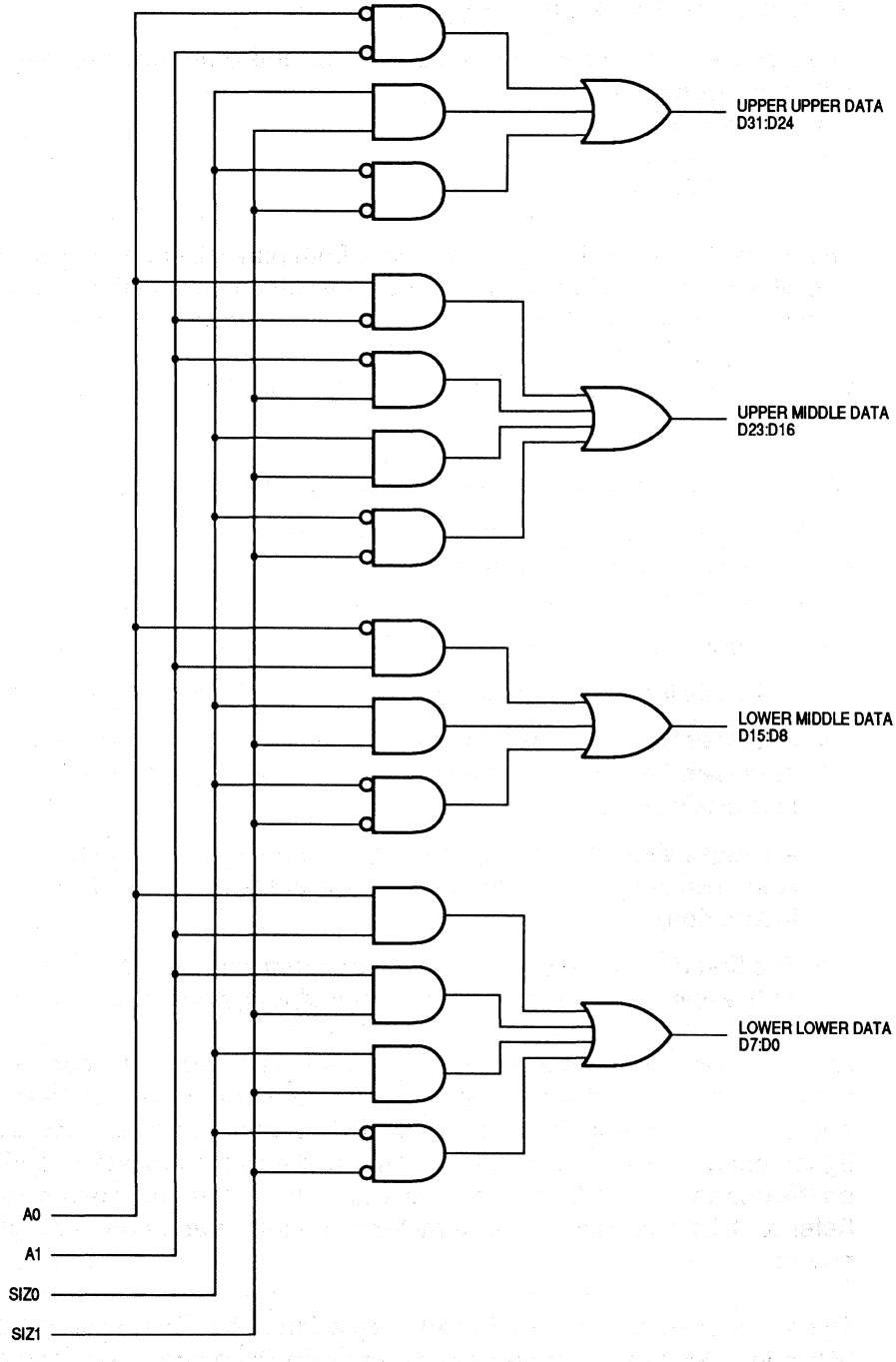


Figure 8-7. Byte Data Select Generation

8.3 PROCESSOR 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 normally both parallel, nonmultiplexed buses. The MC68040 moves data on the bus by issuing control signals and uses a handshake protocol to insure correct movement of data. The following paragraphs describe the bus cycles for byte, word, long-word, and line read and write cycles, and also the read-modify-write transfers.

8.3.1 Byte, Word, and Long-Word Read Cycles

During a read transfer, the processor receives data from a memory or peripheral device. Byte, word, and longword read transfers are performed by the bus controller for the following cases:

- Accesses to a disabled cache
- Accesses to a memory page that is specified non-cachable by the MMU
- Accesses that are implicitly non-cachable (locked read-modify-write accesses and accesses to an alternate logical address space via the MOVES instruction)
- Accesses that do not allocate in the data cache on a read miss (table searches, exception vector fetches, and exception unstacking for an RTE instruction)
- The first transfer of a line read is terminated with $\overline{\text{TBI}}$, forcing completion of the line access using three additional longword read transfers

All byte, word, and longword bus transfers are aligned to the corresponding memory boundary. Word transfers are performed on even address boundaries; long word transfers are performed on long word boundaries. Misaligned operand accesses are handled internally by the processor, and appear on the bus as a sequence of aligned accesses to perform the operand transfer. Refer to **8.2.1 Misaligned Operands** for more information on misaligned operands.

The processor properly positions each byte internally. The section of the data bus from which each byte is read depends on the operand size and address signals A0 and A1. A0 and A1 point to the specific byte required for byte

transfers. For word transfers, A0 is always low, and the address points to the first byte of the word. For long word transfers that are not the result of a burst-inhibited line transfer, both A0 and A1 are low and the address is that of the first byte of the long word. For a burst-inhibited line transfer, A0 and A1 for each of the four accesses (the burst-inhibited line transfer and three long word transfers) are copied from the lowest two bits of the access address that was used to initiate the line transfer.

Since the data read for a byte, word, or long word access is not placed in either of the internal caches, the level on the transfer cache inhibit signal ($\overline{TCI}$) is ignored by the processor when latching the data.

Figure 8-8 is a flowchart for byte, word, and long word read cycles. Bus operations is similar for each case and varies only in the size indicated and the portion of the data bus used for the transfer. Figure 8-9 is a functional timing diagram for a byte, word, and long word read cycle.

Clock 1

The read cycle starts in clock 1 (C1). During the first half of C1 the processor places valid values on the address bus and transfer attributes. For user and supervisor mode accesses that are translated by the corresponding

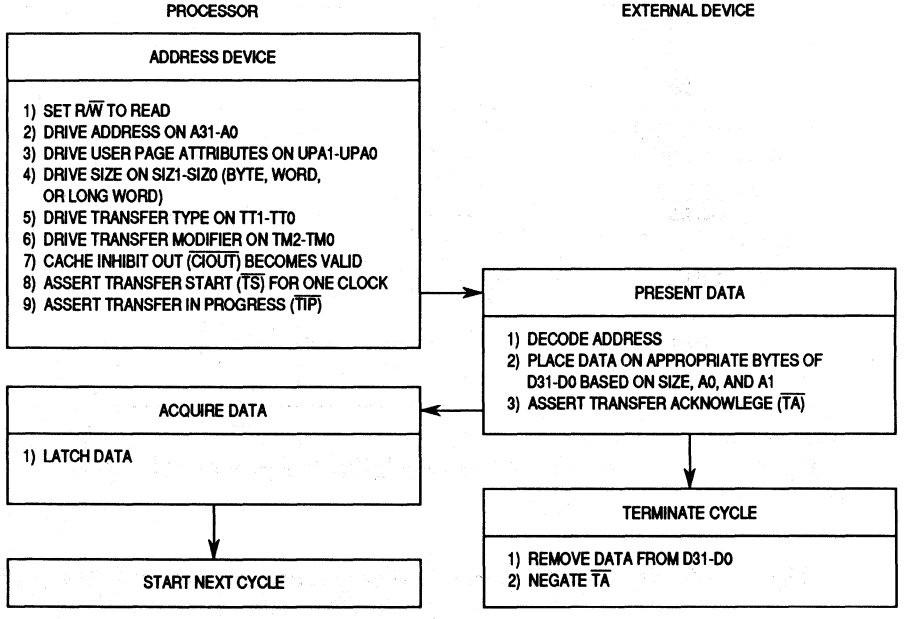


Figure 8-8. Byte, Word, and Long-Word Read Cycle Flowchart

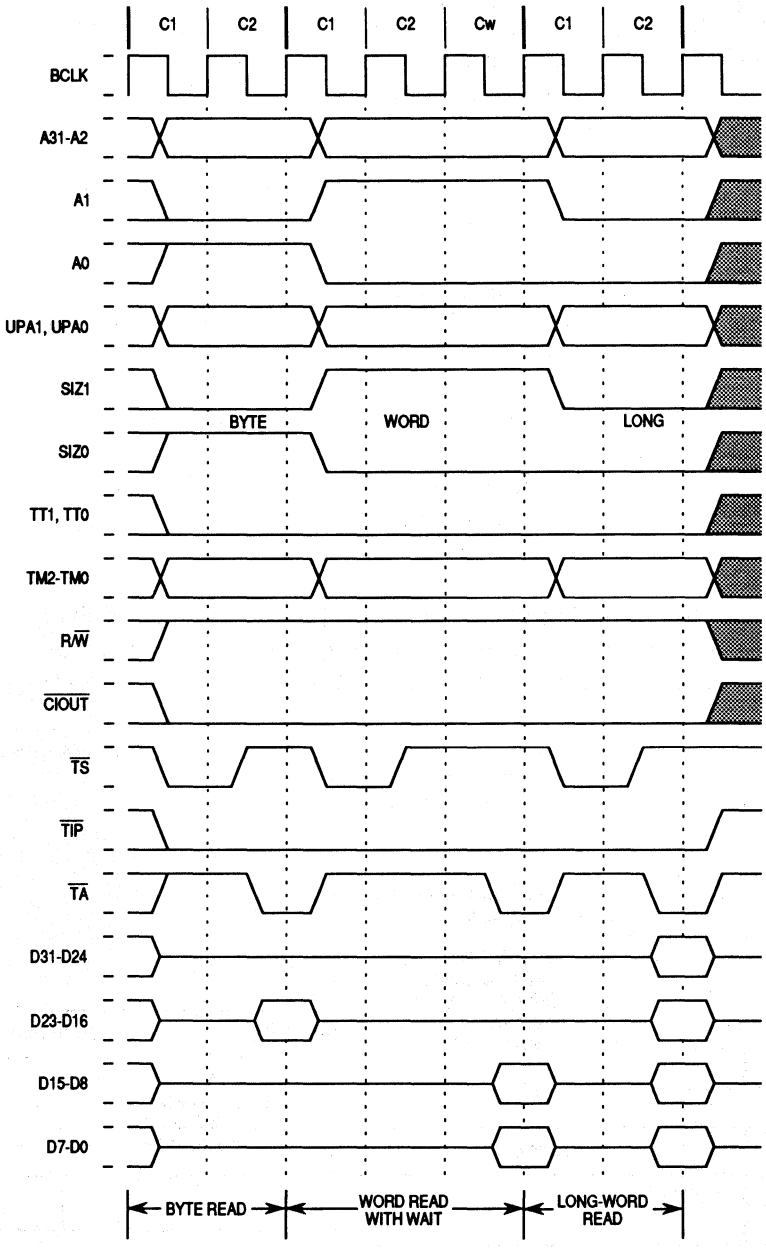


Figure 8-9. Non-Cachable Byte, Word, and Long-Word Read Transfers

MMU, the user programmable attribute signals (UPAn) are driven with the values from the matching ATC entry or transparent translation register user bits (U1 and U0). The transfer type (TTn) and transfer modifier (TMn) signals identify the specific access type. The read/write (R/$\overline{W}$) signal is driven high for a read cycle. Cache inhibit out ($\overline{CIOUT}$) is asserted if the access is identified as non-cachable in the corresponding ATC entry or transparent translation register, of if the access references an alternate address space.

The processor asserts transfer start ($\overline{TS}$) during C1 to indicate the beginning of a bus cycle. The transfer in progress ($\overline{TIP}$) signal is also asserted at this time, if not already asserted from a previous bus cycle, to indicate that a bus cycle is active.

Clock 2

During the first half of clock 2 (C2), the processor negates $\overline{TS}$. The selected device uses R/$\overline{W}$, SIZ0–SIZ1, and A0–A1 to place its information on the data bus. Any or all of the bytes (D24–D31, D16–D23, D8–D15, and D0–D7) are selected by the size signals and A0–A1. Concurrently, the selected device asserts the transfer acknowledge ($\overline{TA}$) signal. At the end of C2, the processor samples the level of $\overline{TA}$ and latches the current value on the data bus. If $\overline{TA}$ is asserted, the bus cycle terminates and the latched data is passed to the appropriate memory unit. If $\overline{TA}$ is not recognized at the end of clock 2, the processor ignores the latched data and appends a wait state (Cw) instead of terminating the transfer. The processor continues to sample the $\overline{TA}$ signal on successive rising edges of BCLK until it is recognized. The latched data is then passed to the appropriate memory unit.

When the processor recognizes $\overline{TA}$ at the end of a clock and terminates the bus cycle, $\overline{TIP}$ remains asserted if the processor is ready to begin another bus cycle. Otherwise, the processor negates $\overline{TIP}$ during the first half of the next clock.

8.3.2 Line Read Transfer

Line read bus cycles are used by the processor to access a 16-byte operand for a MOVE16 instruction, and to support cache line filling. A line read accesses a block of four long words, aligned to a 16-byte memory boundary. This is accomplished by supplying a starting address that points to one of the long words, and requiring the memory device to sequentially drive each long word on the data bus. The memory device internally increments address bits A2 and A3 of the supplied address for each transfer, causing the address to wrap around at the end of the block. The address and transfer attributes

supplied by the processor remain stable during the transfers, and the memory device terminates each transfer by driving the long word on the data bus and asserting $\overline{TA}$. A line transfer performed in this manner with a single address is referred to as a line burst transfer.

The MC68040 also supports "burst-inhibited" line transfers for memory devices that are unable to support bursting. For this type of bus cycle, the selected device supplies the first long word (pointed to by the processor address) and asserts transfer burst inhibit ($\overline{TBI}$) with $\overline{TA}$ for the first transfer of the line access. The processor responds by terminating the line burst transfer and accessing the remainder of the line using three long-word read bus cycles. Although the memory device can then treat the line transfer as four independent long-word bus cycles, the bus controller still handles the four transfers as a single line transfer, and does not allow other unreleated processor accesses or bus arbitration to intervene between the transfers. $\overline{TBI}$ is ignored after the first long-word transfer.

Line reads to support cache line filling can be cache inhibited by asserting transfer cache inhibit ($\overline{TCI}$) with $\overline{TA}$ for the first long-word transfer of the line. The assertion of $\overline{TCI}$ does not affect completion of the line transfer, but is latched by the bus controller and passed to the memory controller for use. $\overline{TCI}$ is ignored after the first long-word transfer of a line burst transfer, and during the three long-word bus cycles for a burst-inhibited line transfer.

The address placed on the address bus by the processor for line transfers does not necessarily point to the most significant byte of each long word, since address bits A1 and A0 for a line read are copied from the original operand address supplied to the memory unit by the integer unit. These two bits are also unchanged for the three long word bus cycles for a burst inhibited line transfer. Memory devices should ignore A1 and A0 for long-word and line bus cycles.

Figure 8-10 is a flowchart for line read bus cycles. Figures 8-11 is a functional diagram for a line burst read.

Clock 1

The line read cycle starts in clock 1 (C1). During the first half of C1 the processor places valid values on the address bus and transfer attributes. For user and supervisor mode accesses that are translated by the corresponding MMU, the user programmable attribute signals (UPAn) are driven with the values from the matching ATC entry or transparent translation register user bits (U1 and U0). The transfer type (TTn) and transfer modifier (TMn) signals identify the specific access type. The read/write (R/$\overline{W}$) signal

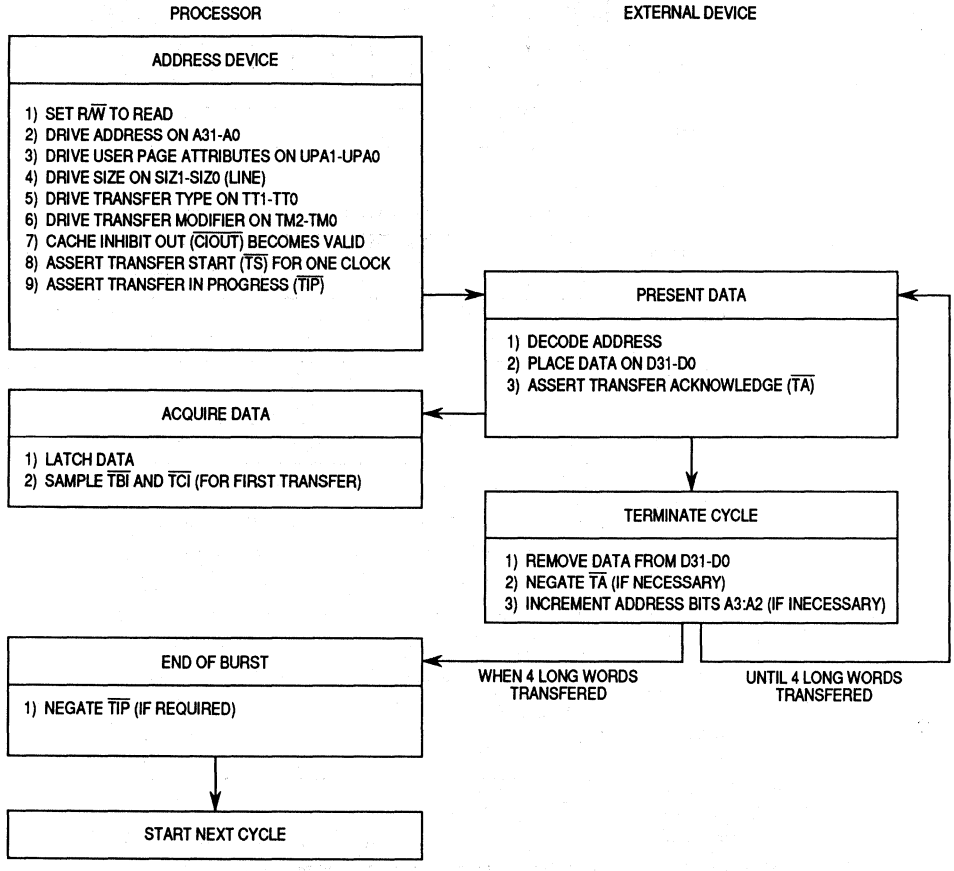


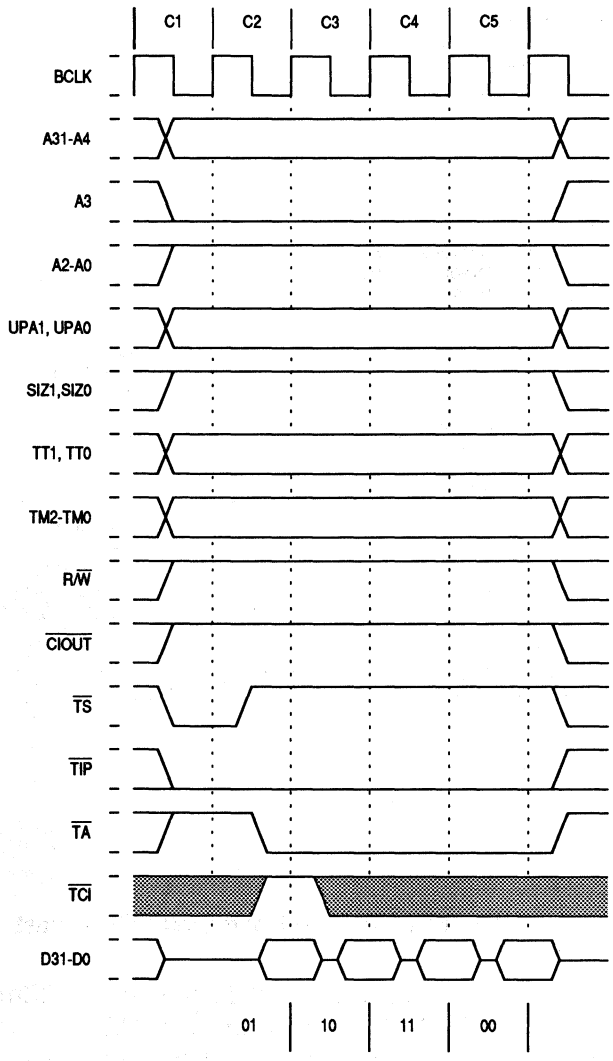
Figure 8-10. Line Read Cycle Flowchart

is driven high for a read cycle, and the size signals (SIZn) indicate size line. Cache inhibit out ($\overline{\text{CIOUT}}$) is asserted for a MOVE16 operand read if the access is identified as non-cachable in the corresponding ATC entry or transparent translation register.

The processor asserts transfer start ($\overline{\text{TS}}$) during C1 to indicate the beginning of a bus cycle. The transfer in progress ($\overline{\text{TIP}}$) signal is also asserted at this time, if not already asserted from a previous bus cycle, to indicate that a bus cycle is active.

Clock 2

During the first half of clock 2 (C2), the processor negates $\overline{\text{TS}}$. The selected device uses R/$\overline{\text{W}}$, and SIZ0–SIZ1 to place the data on the data bus. (The first transfer must supply the long word at the corresponding long-word



Note: Value of A3:A2 incremented by the system hardware.

Figure 8-11. Line Read for Operand Access to Address $07

boundary.) Concurrently, the selected device asserts the transfer acknowledge ($\overline{TA}$) signal, and either negates or asserts transfer burst inhibit ($\overline{TBI}$) to indicate it can or can not support a burst transfer. At the end of C2, the processor samples the level of (TA, $\overline{TBI}$, and $\overline{TCI}$, and latches the current value on the data bus. If $\overline{TA}$ is asserted, the transfer terminates and the

latched data is passed to the appropriate memory unit. If $\overline{TA}$ is not recognized at the end of clock 2, the processor ignores the latched data and inserts wait states instead of terminating the transfer. The processor continues to sample $\overline{TA}$, $\overline{TBI}$, and $\overline{TCI}$ on successive rising edges of BCLK until $\overline{TA}$ is recognized. The latched data and level on $\overline{TCI}$ are then passed to the appropriate memory unit.

If $\overline{TBI}$ was negated with $\overline{TA}$, the processor continues the cycle with clock 3. Otherwise, if $\overline{TBI}$ was asserted, the line transfer is burst-inhibited, and the processor reads the remaining three long words using long-word read bus cycles. Address bits A3:A2 are incremented for each read by the processor, and the new address placed on the address bus for each bus cycle. Refer to **8.3.1 Byte, Word, and Long-Word Read Cycles** for information on long-word reads. If no waits states are generated, a burst-inhibited line read completes in eight clocks instead of the five required for a burst read.

Clock 3

The processor holds the address and transfer attribute signals constant during clock 3. The selected device increments address bits A3:A2 to reference the next long word to transfer, places this data on the data bus, and asserts $\overline{TA}$. At the end of C3, the processor samples the level of $\overline{TA}$ and latches the current value on the data bus. If $\overline{TA}$ is asserted, the transfer terminates and the second lond word of data is passed to the appropriate memory unit. If $\overline{TA}$ is not recognized at the end of clock 3, the processor ignores the latched data and inserts wait states instead of terminating the transfer. The processor continues to sample the $\overline{TA}$ signal on successive rising edges of BCLK until it is recognized. The latched data is then passed to the appropriate memory unit.

Clock 4

This clock is identical to clock 3 except that once $\overline{TA}$ is recognized, the latched value corresponds to the third long word of data for the burst.

Clock 5

This clock is identical to clock 3 except that once $\overline{TA}$ is recognized, the latched value corresponds to the third long word of data for the burst. After the processor recognizes the last $\overline{TA}$ assertion and terminates the line read bus cycle, $\overline{TIP}$ remains asserted if the processor is ready to begin another bus cycle. Otherwise, the processor negates $\overline{TIP}$ during the first half of the next clock.

Figure 8-12 is a flowchart for a burst-inhibited line read. Figures 8-13 is a functional diagram for a burst-inhibited line read.

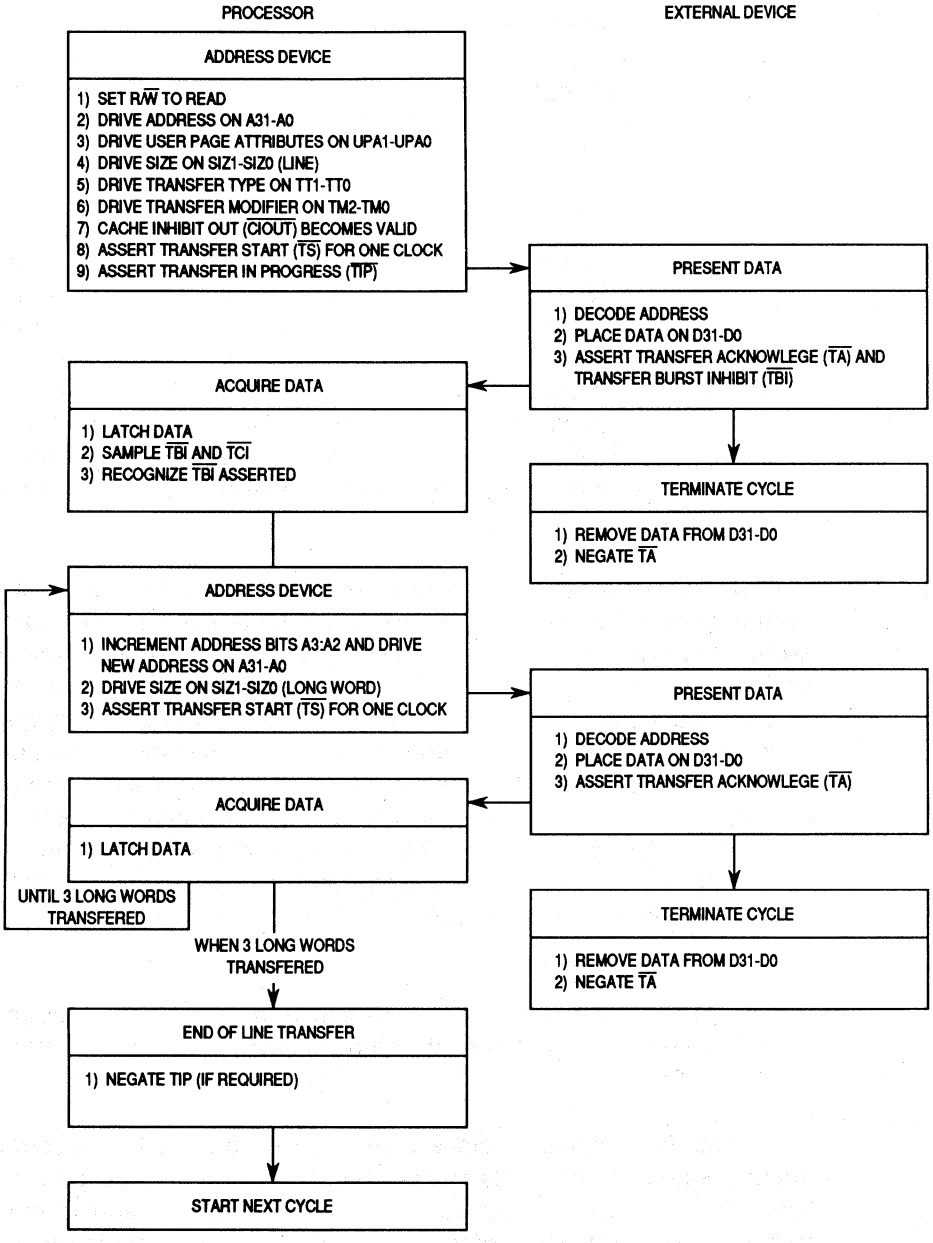


Figure 8-12. Burst-Inhibited Line Read Flowchart

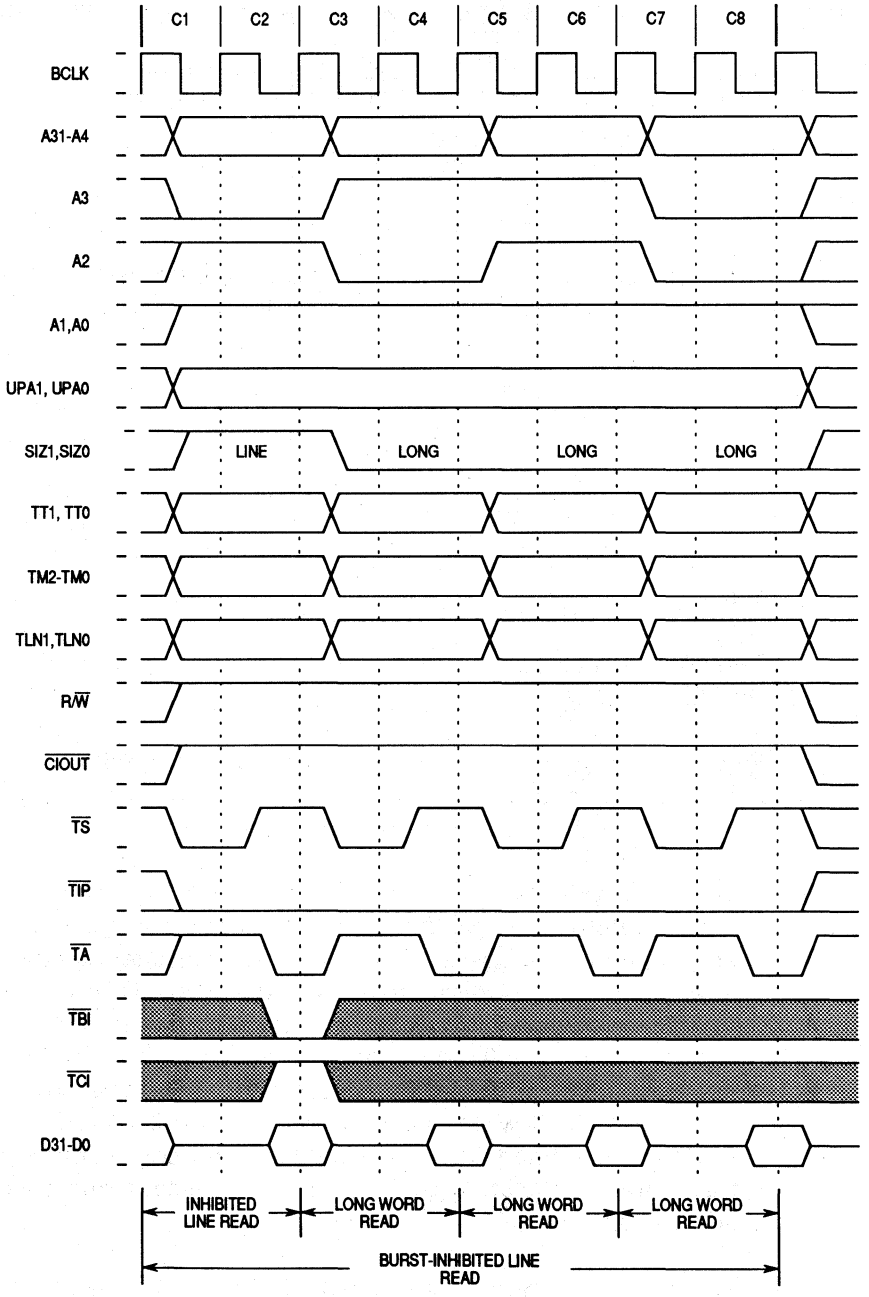


Figure 8-13. Burst-Inhibited Line Read

8.3.3 Byte, Word, and Long-Word Write Cycles

During a write transfer, the processor transfers data to a memory or peripheral device. Byte, word, and longword write transfers are performed by the bus controller for the following cases:

- Accesses to a disabled cache
- Accesses to a memory page that is specified non-cachable by the MMU
- Accesses that are implicitly non-cachable (locked read-modify-write accesses and accesses to an alternate logical address space via the MOVES instruction)
- Writes to writethrough pages
- Accesses that do not allocate in the data cache on a write miss (table updates and exception stacking)
- The first transfer of a line write is terminated with $\overline{TBI}$, forcing completion of the line access using three additional longword write transfers
- Cache line pushes for lines containing a single dirty long word

The level on the transfer cache inhibit signal ($\overline{TCI}$) is ignored by the processor during all write cycles.

Figure 8-14 is a flowchart for byte, word, and long word write cycles. Figure 8-15 is a functional timing diagram for a long word write cycle.

Clock 1

The write cycle starts in clock 1 (C1). During the first half of C1 the processor places valid values on the address bus and transfer attributes. For user and supervisor mode accesses that are translated by the corresponding MMU, the user programmable attribute signals (UPAn) are driven with the values from the matching ATC entry or transparent translation register user bits (U1 and U0). The transfer type (TTn) and transfer modifier (TMn) signals identify the specific access type. The read/write (R/$\overline{W}$) signal is driven low for a write cycle. Cache inhibit out ($\overline{CIOUT}$) is asserted if the access is identified as non-cachable in the corresponding ATC entry or transparent translation register, of if the access references an alternate address space.

The processor asserts transfer start ($\overline{TS}$) during C1 to indicate the beginning of a bus cycle. The transfer in progress ($\overline{TIP}$) signal is also asserted at this time, if not already asserted from a previous bus cycle, to indicate that a bus cycle is active.

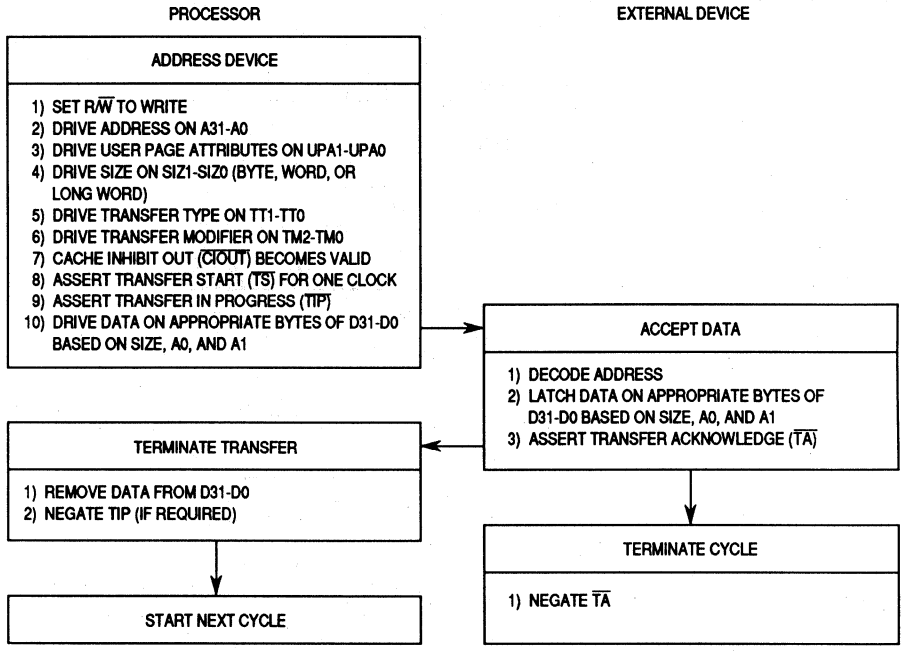


Figure 8-14. Byte, Word, and Long-Word Write Cycle Flowchart

Clock 2

During the first half of clock 2 (C2), the processor negates $\overline{TS}$ and drives the appropriate bytes of the data bus with the data to be written (based on size, A0, and A1. All other bytes are driven with undefined values. The selected device uses R/$\overline{W}$, SIZ0–SIZ1, A0–A1, and $\overline{CIOUT}$ to latch the information on the data bus. Any or all of the bytes (D24–D31, D16–D23, D8–D15, and D0–D7) are selected by the size signals and A0–A1. Concurrently, the selected device asserts the transfer acknowledge ($\overline{TA}$) signal. At the end of C2, the processor samples the level of $\overline{TA}$; if $\overline{TA}$ is asserted, the bus cycle terminates. If $\overline{TA}$ is not recognized at the end of clock 2, the processor appends a wait state instead of terminating the transfer. The processor continues to sample the $\overline{TA}$ signal on successive rising edges of BCLK until it is recognized.

When the processor recognizes $\overline{TA}$ at the end of a clock and terminates the bus cycle, $\overline{TIP}$ remains asserted if the processor is ready to begin another bus cycle. Otherwise, the processor negates $\overline{TIP}$ during the first half of the next clock. The processor also three-states the data bus during the first half of the next clock following termination of the write cycle.

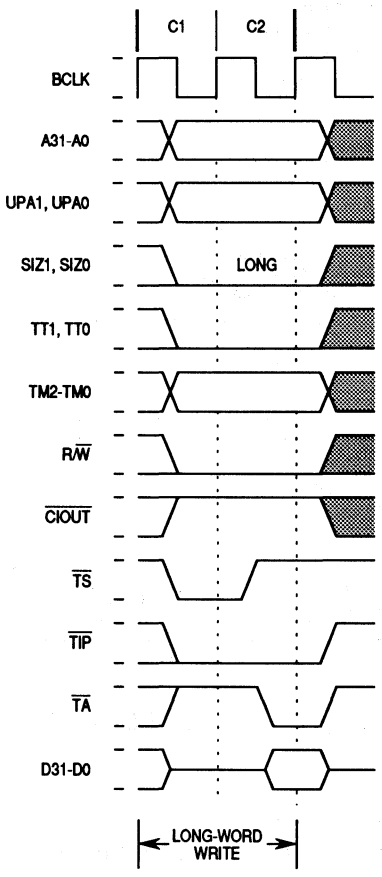


Figure 8-15. Long-Word Write Transfer

8.3.4 Line Write Transfer

Line write bus cycles are used by the processor to access a 16-byte operand for a MOVE16 instruction, and to support cache line pushes. Both burst and burst-inhibited transfers are supported.

Figure 8-16 is a flowchart for line write bus cycles. Figures 8-17 is a functional diagram for a line burst write.

Clock 1

The line write cycle starts in clock 1 (C1). During the first half of C1 the processor places valid values on the address bus and transfer attributes. For user and supervisor mode accesses that are translated by the corresponding MMU, the user programmable attribute signals (UPAn) are driven

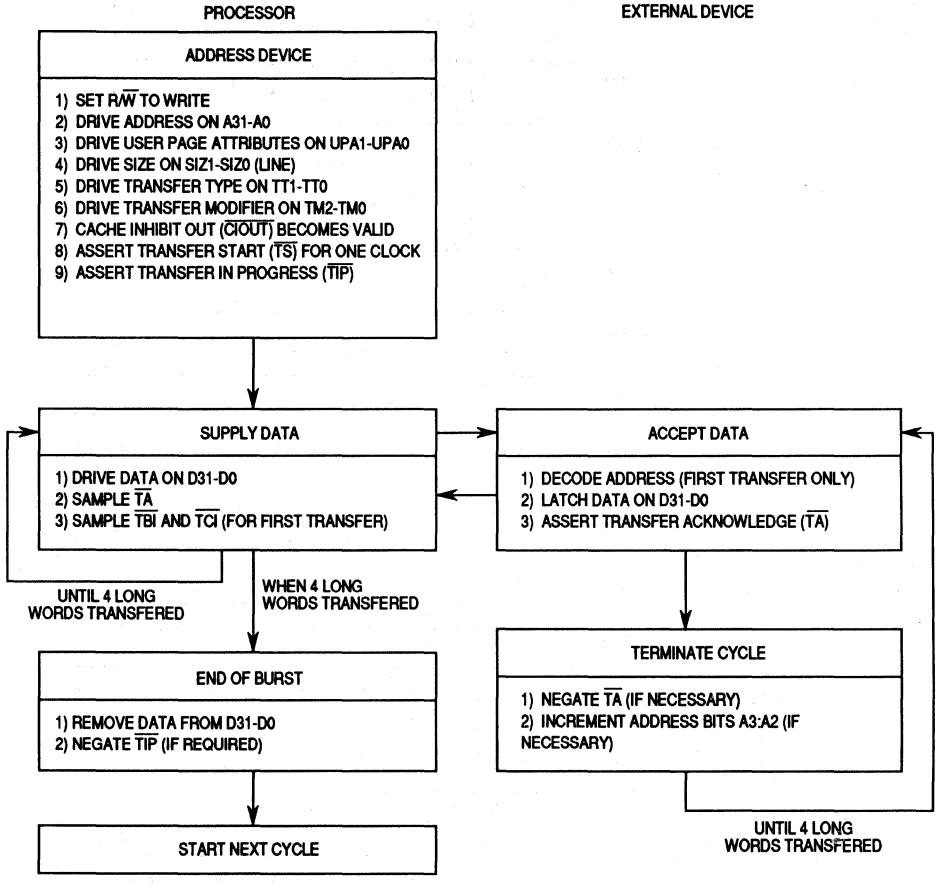
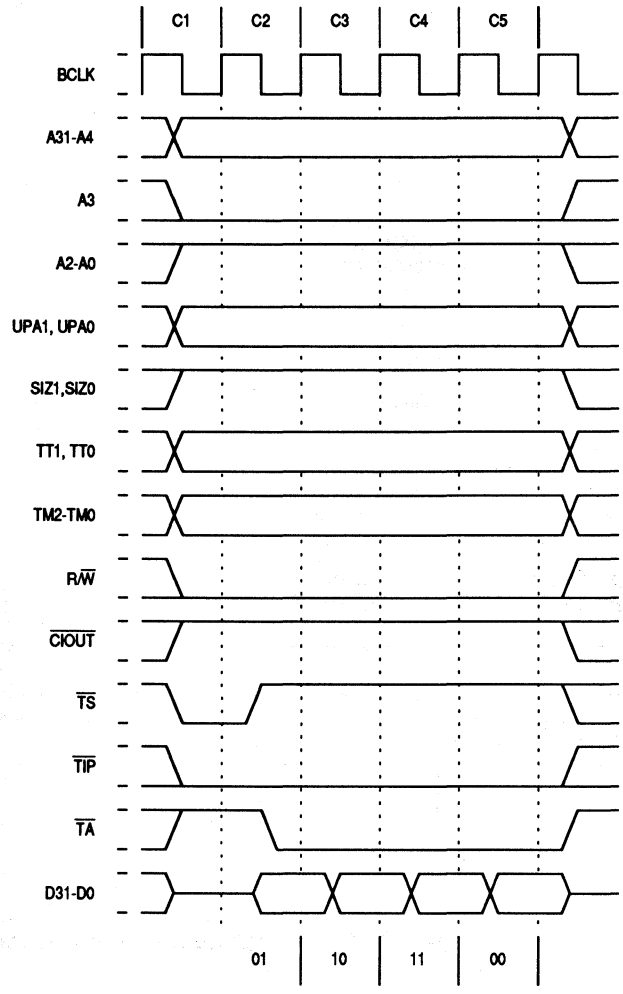


Figure 8-16. Line Write Cycle Flowchart

with the values from the matching ATC entry or transparent translation register user bits (U1 and U0). The transfer type (TTn) and transfer modifier (TMn) signals identify the specific access type. The read/write (R/$\overline{W}$) signal is driven low for a write cycle, and the size signals (SIZn) indicate size line. Cache inhibit out ($\overline{CIOUT}$) is asserted for a MOVE16 operand read if the access is identified as non-cachable in the corresponding ATC entry or transparent translation register.

The processor asserts transfer start ($\overline{TS}$) during C1 to indicate the beginning of a bus cycle. The transfer in progress ($\overline{TIP}$) signal is also asserted at this time, if not already asserted from a previous bus cycle, to indicate that a bus cycle is active.



Note: Value of A3:A2 incremented by the system hardware.

Figure 8-17. Line Write for Operand Access to Address $07

Clock 2

During the first half of clock 2 (C2), the processor negates $\overline{TS}$ and drives the data bus with the data to be written. The selected device uses R/$\overline{W}$, and SIZ0–SIZ1 to latch the data on the data bus. Concurrently, the selected device asserts the transfer acknowledge ($\overline{TA}$) signal, and either negates or asserts transfer burst inhibit ($\overline{TBI}$) to indicate it can or can not support a burst transfer. At the end of C2, the processor samples the level of $\overline{TA}$ and

of $\overline{TA}$ and $\overline{TBI}$. If $\overline{TA}$ is asserted, the transfer terminates. If $\overline{TA}$ is not recognized at the end of clock 2, the processor inserts wait states instead of terminating the transfer. The processor continues to sample $\overline{TA}$ and $\overline{TBI}$ on successive rising edges of BCLK until $\overline{TA}$ is recognized.

If $\overline{TBI}$ was negated with $\overline{TA}$, the processor continues the cycle with clock 3. Otherwise, if $\overline{TBI}$ was asserted, the line transfer is burst-inhibited, and the processor writes the remaining three long words using long-word write bus cycles. Address bits A3:A2 are incremented for each write by the processor, and the new address placed on the address bus for each bus cycle. Refer to **8.3.3 Byte, Word, and Long-Word Write Cycles** for information on long-word writes. If no waits states are generated, a burst-inhibited line write completes in eight clocks instead of the five required for a burst write.

Clock 3

The processor drive the second long word of data on the data bus, and holds the address and transfer attribute signals constant during clock 3. The selected device increments address bits A3:A2 to reference the next long word, latches this data from the data bus, and asserts $\overline{TA}$. At the end of C3, the processor samples the level of $\overline{TA}$; if $\overline{TA}$ is asserted, the transfer terminates. If $\overline{TA}$ is not recognized at the end of clock 3, the processor inserts wait states instead of terminating the transfer. The processor continues to sample the $\overline{TA}$ signal on successive rising edges of BCLK until it is recognized.

Clock 4

This clock is identical to clock 3 except that the value driven on the data bus corresponds to the third long word of data for the burst.

Clock 5

This clock is identical to clock 3 except that the value driven on the data bus corresponds to the third long word of data for the burst. After the processor recognizes the last $\overline{TA}$ assertion and terminates the line write bus cycle, $\overline{TIP}$ remains asserted if the processor is ready to begin another bus cycle. Otherwise, the processor negates $\overline{TIP}$ during the first half of the next clock. The processor also three-states the data bus during the first half of the next clock following termination of the write cycle.

8.3.5 Locked Transfer

The locked (or read-modify-write) cycle performs a read, conditionally modifies the data in the processor, and writes the data out to memory. In the MC68040 processor, this operation can be indivisible, providing semaphore capabilities for multi-processor systems. During the entire read-modify-write sequence the MC68040 asserts the $\overline{\text{LOCK}}$ signal to indicate that an indivisible operation is occurring, and asserts the $\overline{\text{LOCKE}}$ signal for the last transfer to indicate the completion of the locked sequence. The $\overline{\text{LOCK}}$ and $\overline{\text{LOCKE}}$ signals can be used by the external arbiter to prevent arbitration of the bus during locked processor sequences, by using the $\overline{\text{LOCKE}}$ signal to perform arbitration between two locked sequences. A read-modify-write operation is treated as noncachable. If the access hits in the data cache, it invalidates a matching valid entry and pushes a matching dirty entry. The locked transfer begins only after the line push (if required) has completed.

The test and set (TAS) and compare and swap (CAS and CAS2) instructions are the only MC68040 instructions that utilize locked transfers. Some page descriptor updates during translation table searches for the memory management units (MMUs) also use locked transfers. Refer to **SECTION 6 MEMORY MANAGEMENT** for information about the MMUs.

The locked transfer for the CAS and CAS2 instructions in the MC68040 differs from the read-modify-write bus cycles used by previous members of the M68000 Family, in order to support the $\overline{\text{LOCKE}}$ signal. If an operand does not match for one of these instructions, the MC68040 still executes a single write transfer to terminate the locked sequence with $\overline{\text{LOCKE}}$ asserted. For the CAS instruction the value read from memory is written back, while for the CAS2 instruction the second operand read is written back.

Figure 8-18 illustrates an example of a functional timing diagram for a TAS instruction.

Clock 1

The read cycle starts in clock 1 (C1). During the first half of C1 the processor places valid values on the address bus and transfer attributes. The lock signal ($\overline{\text{LOCK}}$) is asserted to identify a locked read-modify-write bus cycle. For user and supervisor mode accesses that are translated by the corresponding MMU, the user programmable attribute signals (UPAn) are driven with the values from the matching ATC entry or transparent translation register user bits (U1 and U0). The transfer type (TTn) and transfer modifier (TMn) signals identify the specific access type. The read/write (R/$\overline{\text{W}}$) signal is driven high for a read cycle. Cache inhibit out ($\overline{\text{CIOUT}}$) is asserted if the

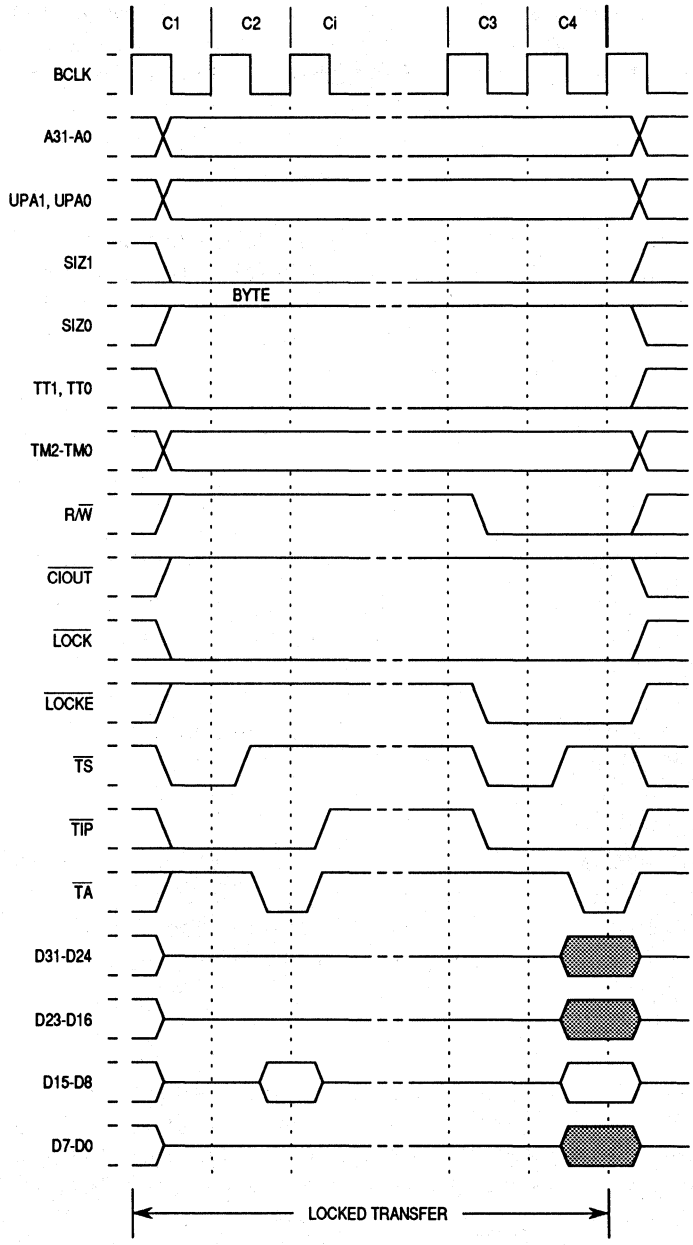


Figure 8-18. Locked Transfer for TAS Instruction

access is identified as non-cachable in the corresponding ATC entry or transparent translation register. The processor asserts transfer start ($\overline{TS}$) during C1 to indicate the beginning of a bus cycle. The transfer in progress ($\overline{TIP}$) signal is also asserted at this time, if not already asserted from a previous bus cycle, to indicate that a bus cycle is active.

Clock 2

During the first half of clock 2 (C2), the processor negates $\overline{TS}$. The selected device uses R/$\overline{W}$, SIZ0–SIZ1, and A1–A0 to place its information on the data bus. Any or all of the bytes (D31–D24, D23–D16, D15–D8, and D7–D0) are selected by the size signals and A1–A0. Concurrently, the selected device asserts the transfer acknowledge ($\overline{TA}$) signal. At the end of C2, the processor samples the level of $\overline{TA}$ and latches the current value on the data bus. If $\overline{TA}$ is asserted, the read transfer terminates and the latched data is passed to the appropriate memory unit. If $\overline{TA}$ is not recognized at the end of C2, the processor ignores the latched data and appends a wait state instead of terminating the transfer. The processor continues to sample the $\overline{TA}$ signal on successive rising edges of BCLK until it is recognized asserted. The latched data is then passed to the appropriate memory unit. If more than one read cycle is required to read in the operand(s), clock states C1 and C2 are repeated accordingly.

When the processor recognizes $\overline{TA}$ at the end of the last read transfer for the locked bus cycle, it negates $\overline{TIP}$ during the first half of the next clock.

Idle Clocks

The processor does not assert any new control signals during the idle clock 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 clock 3 to prevent bus conflicts with the preceding read portion of the cycle; the data bus is not driven until clock 4.

Clock 3

During the first half of C3 the processor places valid values on the address bus and transfer attributes, and drives the read/write (R/$\overline{W}$) signal low for a write cycle. The processor asserts transfer start ($\overline{TS}$) to indicate the beginning of a bus cycle. The transfer in progress ($\overline{TIP}$) signal is also asserted at this time to indicate that a bus cycle is active.

The lock end signal ($\overline{LOCKE}$) is asserted during C3 for the last write transfer of the locked sequence. If multiple writes transfers are required for misaligned operands or multiple operands, $\overline{LOCKE}$ is asserted only for the final write transfer. The external arbiter can use this indication to distinguish

between two back-to-back locked bus cycles and allow arbitration between them.

Clock 4

During the first half of clock 4 (C4), the processor negates $\overline{TS}$ and drives the appropriate bytes of the data bus with the data to be written (based on size, A0, and A1). All other bytes are driven with undefined values. The selected device uses R/$\overline{W}$, SIZ0–SIZ1, and A0–A1 to latch the information on the data bus. Any or all of the bytes (D31–D24, D23–D16, D15–D8, and D7–D0) are selected by the size signals and A0–A1. Concurrently, the selected device asserts the transfer acknowledge ($\overline{TA}$) signal. At the end of C4, the processor samples the level of $\overline{TA}$; if $\overline{TA}$ is asserted, the bus cycle terminates. If $\overline{TA}$ is not recognized asserted at the end of C4, the processor appends a wait state instead of terminating the transfer. The processor continues to sample the $\overline{TA}$ signal on successive rising edges of BCLK until it is recognized.

When the processor recognizes $\overline{TA}$ at the end of a clock, the bus cycle is terminated, but $\overline{TIP}$ remains asserted if the processor is ready to begin another bus cycle. Otherwise, the processor negates $\overline{TIP}$ during the first half of the next clock. The processor also three-states the data bus during the first half of the next clock following termination of the write cycle.

When the last write transfer is terminated, $\overline{LOCKE}$ is negated. The processor also negates $\overline{LOCK}$ if the next bus cycle is not a locked transfer.

8.4 Acknowledge Cycles

Bus transfers with transfer type signals TT1/TT0 = 11 are classified as acknowledge bus transfers. Interrupt acknowledge and breakpoint acknowledge bus cycles use this encoding, and are described in the following paragraphs.

8.4.1 Interrupt Acknowledge Bus Cycles

When a peripheral device signals the processor (with the $\overline{IPL2}$–$\overline{IPL0}$ signals) that service is required, 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 **SECTION 9 EXCEPTIONS** for details on the recognition of interrupts.

The MC68040 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. Table 8-7 provides a summary of the possible interrupt acknowledge terminations and resulting exception processing.

Table 8-7. Interrupt Acknowledge Termination Summary

$\overline{TA}$	$\overline{TEA}$	$\overline{AVEC}$	Termination Condition
N	N	X	Insert Waits
N	A	X	Take Spurious Interrupt Exception
A	N	N	Latch Vector Number on D7–D0 and Take Interrupt Exception
A	N	A	Take Autovectored Interrupt Exception
A	A	N	Retry Interrupt Acknowledge Cycle
A	A	A	Take Spurious Interrupt Exception

Legend: N = signal negated, A = signal asserted, X = don't care

8.4.1.1 INTERRUPT ACKNOWLEDGE CYCLE—TERMINATED NORMALLY. When the MC68040 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. Other interrupting conditions or devices cannot supply a vector number and use the autovector cycle described in **8.4.1.2 AUTOVECTOR INTERRUPT ACKNOWLEDGE CYCLE.**

The interrupt acknowledge cycle is a read cycle. It differs from a normal read cycle in the following respects:

1. The transfer type signals are set to three (TT1/TT0 = 11) to indicate an acknowledge bus cycle.
2. Address signals A31–A0 are set to all one's ($FFFFFFFF).
3. The transfer modifier signals TM2, TM1, and TM0 are set to the interrupt request level (the inverted values of $\overline{IPL2}$, $\overline{IPL1}$, and $\overline{IPL0}$, respectively).

The responding device places the vector number on the data bus during the interrupt acknowledge cycle, and the cycle is terminated normally with $\overline{TA}$. Figure 8-19 is the flowchart of the interrupt acknowledge cycle.

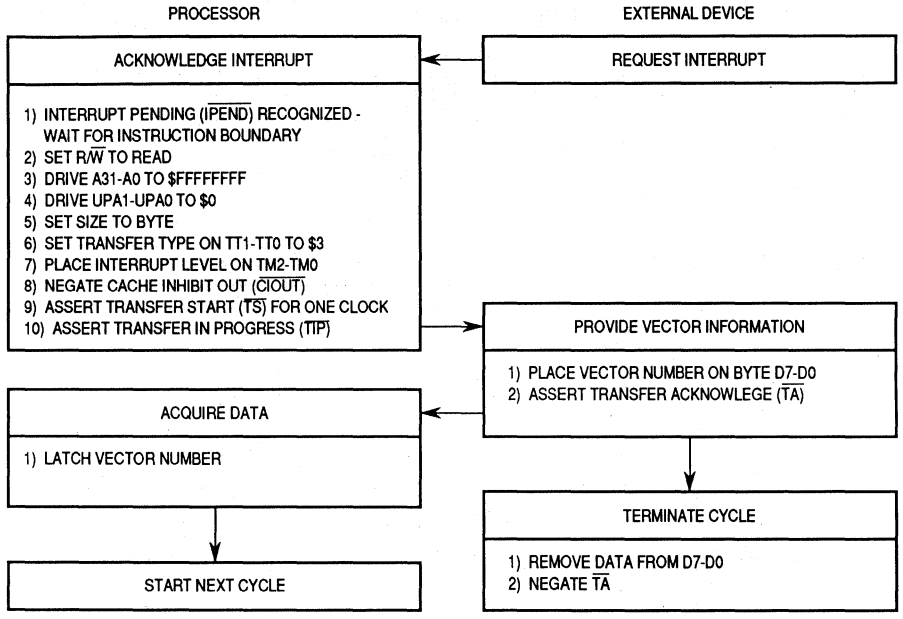


Figure 8-19. Interrupt Acknowledge Cycle Flowchart

Figure 8-20 shows the timing for an interrupt acknowledge cycle terminated with $\overline{TA}$.

8.4.1.2 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 the transfer acknowledge signal ($\overline{TA}$), the device asserts the autovector ($\overline{AVEC}$) signal with $\overline{TA}$ to terminate the cycle.

The vector number supplied in an autovector operation is derived from the interrupt level of the current interrupt. When the $\overline{AVEC}$ signal is asserted with $\overline{TA}$ during an interrupt acknowledge cycle, the MC68040 ignores the state of the data bus and internally generates the vector number, which is 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 $\overline{IPL0}$–$\overline{IPL2}$. Figure 8-21 shows the timing for an autovector operation.

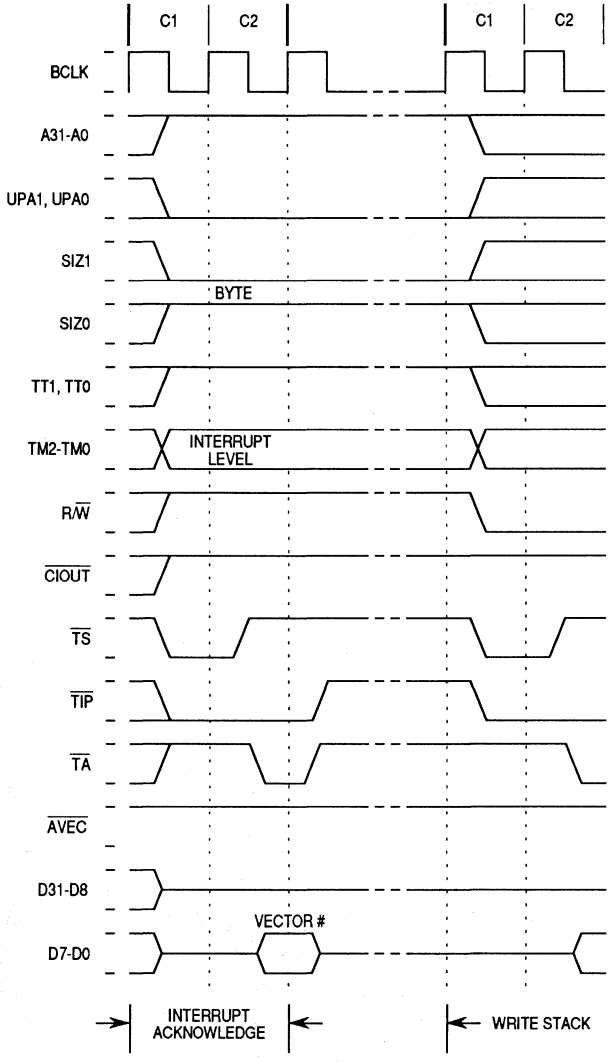


Figure 8-20. Interrupt Acknowledge Cycle Timing

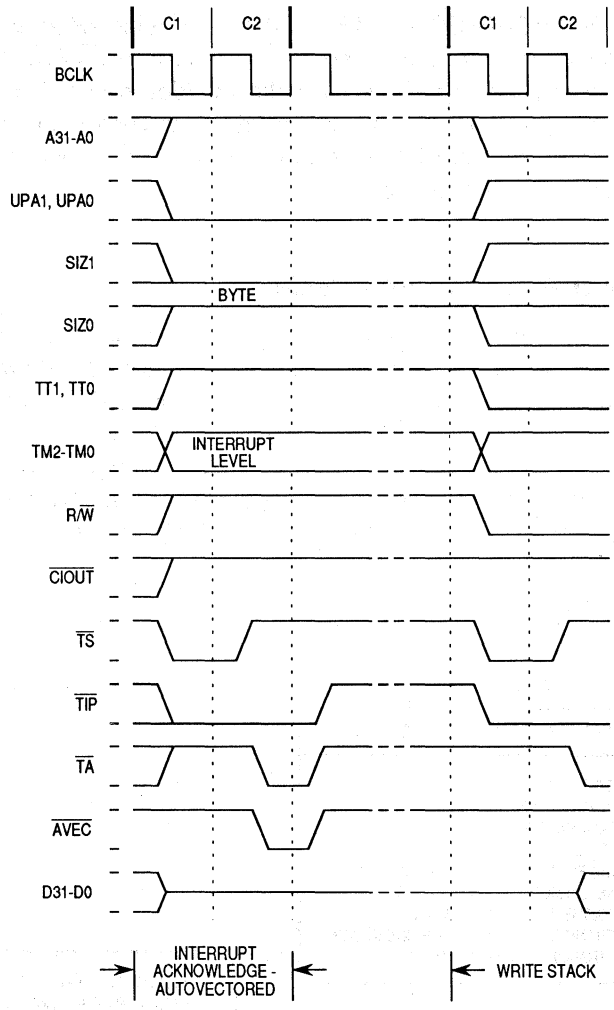


Figure 8-21. Autovector Operation Timing

8.4.1.3 SPURIOUS INTERRUPT CYCLE. When a device does not respond to an interrupt acknowledge cycle with $\overline{TA}$, or $\overline{TA}$ and $\overline{AVEC}$, the external logic typically returns the transfer error acknowledge signal ($\overline{TEA}$). The MC68040 automatically generates the spurious interrupt vector number 24, instead of the interrupt vector number in this case. If $\overline{TA}$ and $\overline{TEA}$ are both asserted, the processor retries the cycle.

8.4.2 Breakpoint Acknowledge Cycle

The breakpoint acknowledge cycle is generated by the execution of a breakpoint instruction (BKPT). An acknowledge access is indicated with transfer type signals TT1/TT0 = $3, address A31–A0 = $00000000, and transfer modifier signals TM2–TM0 = $0. When the external hardware terminates the cycle with either $\overline{TA}$ or $\overline{TEA}$, the processor takes an illegal instruction exception. Figure 8-22 is a flowchart of the breakpoint acknowledge cycle. Figure 8-23 shows the timing for a breakpoint acknowledge cycle.

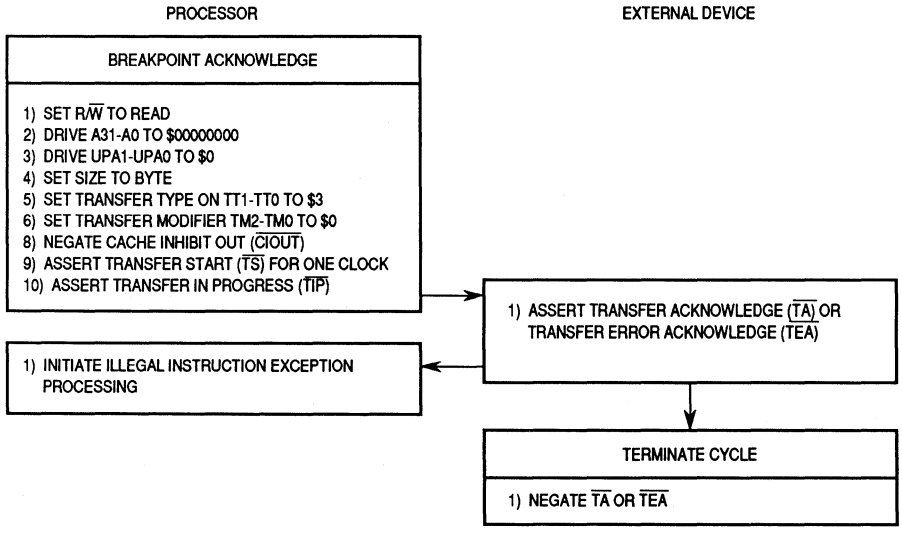


Figure 8-22. Breakpoint Operation Flow

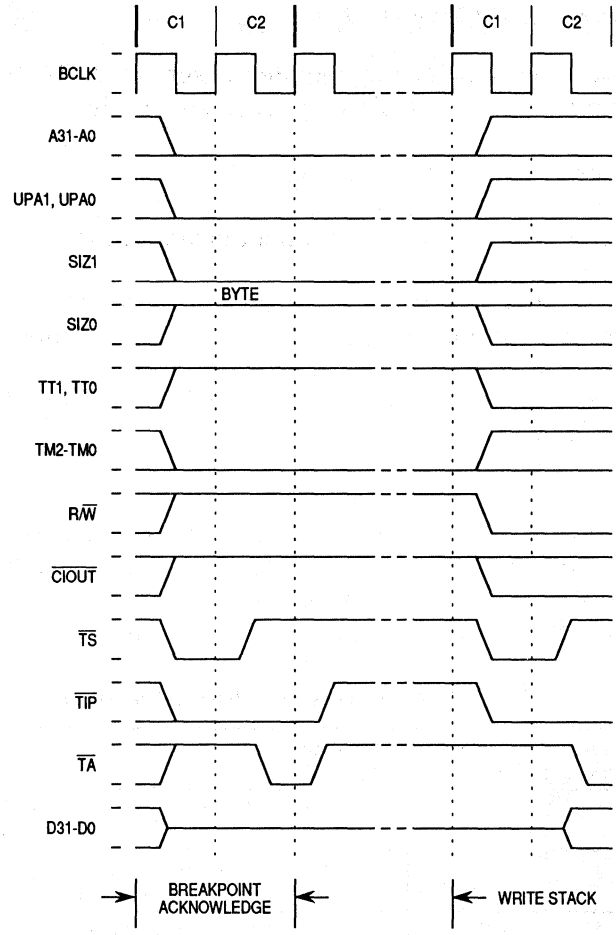


Figure 8-23. Breakpoint Acknowledge Cycle Timing

8.5 BUS EXCEPTION CONTROL CYCLES

The MC68040 bus architecture requires assertion of $\overline{TA}$ from an external device to signal that a bus cycle is complete. $\overline{TA}$ is not asserted in these cases:

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

External circuitry can provide the transfer error acknowledge signal ($\overline{TEA}$) when no device responds by asserting $\overline{TA}$ within an appropriate period of time after the processor begins the bus cycle. This allows the cycle to terminate and the processor to enter exception processing for the error condition. $\overline{TEA}$ can also be asserted in combination with $\overline{TA}$ to cause a retry of a bus cycle in error.

In order to properly control termination of a bus cycle for a retry or a bus error condition, $\overline{TA}$ and $\overline{TEA}$ must be asserted and negated for the same rising edge of the MC68040 bus clock (BCLK). Table 8-8 shows the control signal combinations and the resulting bus cycle terminations.

Table 8-8. $\overline{TA}$ and $\overline{TEA}$ Assertion Results

Case No.	Control Signal	Asserted/ Negated	Result
1	$\overline{TA}$ $\overline{TEA}$	A NA	Normal Cycle Terminate and Continue
2	$\overline{TA}$ $\overline{TEA}$	NA A	Terminate and Take Bus Eerror Exception, Possibly Deferred
3	$\overline{TA}$ $\overline{TEA}$	A A	Terminate and Retry

LEGEND:
A—Signal is asserted for this BLCK rising edge
NA—Signal is not asserted for this BLCK rising edge

Bus error and retry terminations during burst cycles operate as described in **8.8.3.2 LINE READ TRANSFER** and **8.3.4 Line Write Transfer.**

8.5.1 Bus Errors

The transfer error acknowledge ($\overline{TEA}$) signal can be used by the system hardware to abort the current bus cycle when a fault is detected. A bus error is recognized during a bus cycle when $\overline{TA}$ is negated and $\overline{TEA}$ is asserted.

When the processor recognizes a bus error condition for an access, the access is terminated immediately. A line access that has $\overline{TEA}$ asserted for one of the four longword transfers aborts without completing the remaining transfers, regardless of whether the line transfer uses a burst access or burst-inhibited access.

When $\overline{TEA}$ is asserted to terminate a bus cycle, the MC68040 may enter access error exception processing immediately following the bus cycle, or it may defer processing the exception. The instruction prefetch mechanism requests instruction words from the data memory unit 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 a task switch occurs, an access error exception does not occur. Similarly, if a bus error is detected on the second, third, or fourth longword transfer for a line read access, an access error exception is taken only if the execution unit is specifically requesting that long word. Otherwise, the bus errored line is not placed in the cache, and the processor repeats the line access when another access references the line. If a misaligned operand spans two long words in a line, a bus error on either the first or second transfers for the line causes exception processing to begin immediately. A bus error termination for any write accesses or for read accesses that reference data specifically requested by the execution unit causes the processor to begin exception processing immediately. Refer to **SECTION 9 EXCEPTION PROCESSING** for details of access error exception processing.

When an access is terminated by a bus error, the contents of the corresponding cache can be affected in different ways, depending on the type of access. For a cache line read to replace a valid instruction or data cache line, the cache line being filled is invalidated before the bus cycle begins and remains invalid if the replacement line access is terminated with a bus error. If a dirty data cache line is being replaced and the replacement line read is bus errored, the dirty line is restored from an internal push buffer into the cache to eliminate an unnecessary push access. For a data cache push which is bus errored, the corresponding cache line remains valid (with the new line data) if the line push follows a replacement line read, or is invalidated if the push is explicitly forced by a CPUSH instruction. Write accesses to memory pages specified as writethrough by the data MMU update the corresponding cache line before accessing memory. If the memory access is bus errored, the cache line remains valid with the new data.

Figure 8-24 shows the timing of a bus error on a word access which causes an access error exception. Figure 8-25 shows the timing of a bus error on a line read access which does not cause an access error exception.

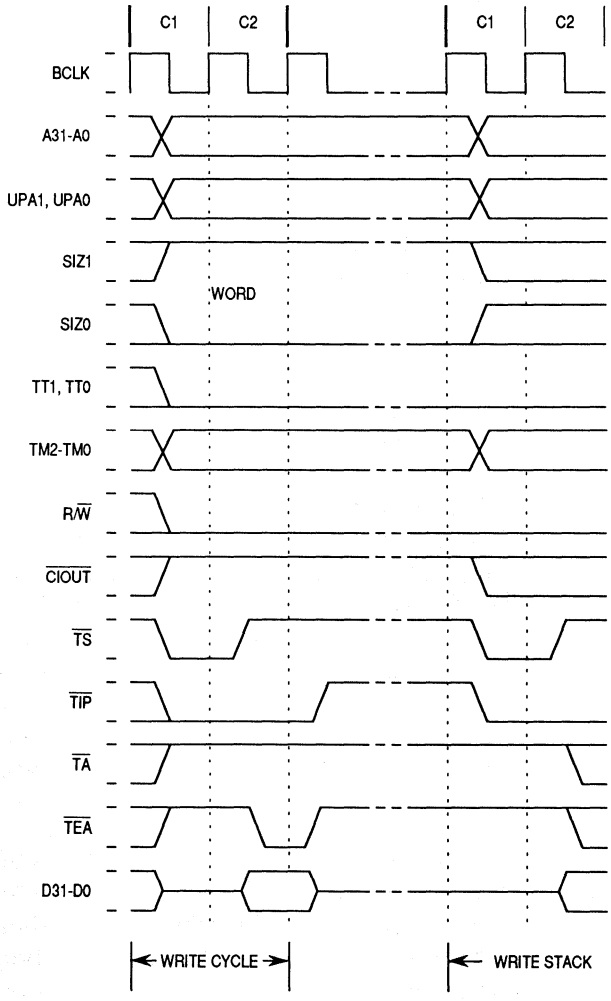
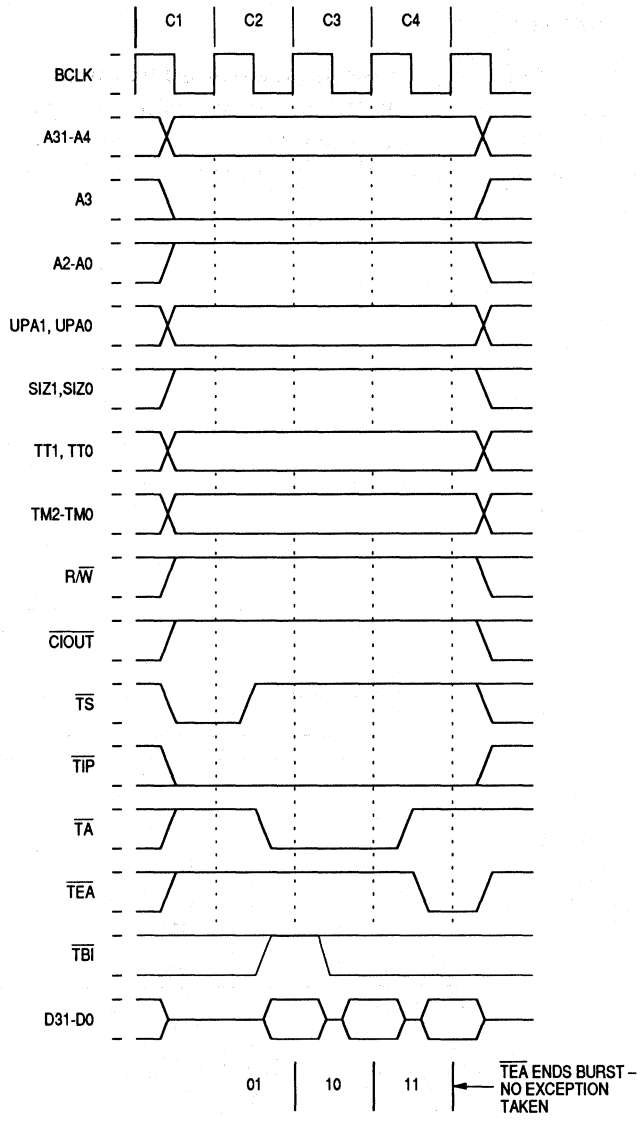


Figure 8-24. Word Write Access Terminated with TEA



Note: Value of A3:A2 incremented by the system hardware.

Figure 8-25. Line Read Access Terminated with TEA

8.5.2 Retry Operation

When the $\overline{TA}$ and $\overline{TEA}$ signals are both asserted by an external device during a bus cycle, the processor enters the retry sequence. The processor terminates the bus cycle and immediately retries the cycle using the same access information (address and transfer attributes). Figure 8-26 shows a retry of a read bus cycle.

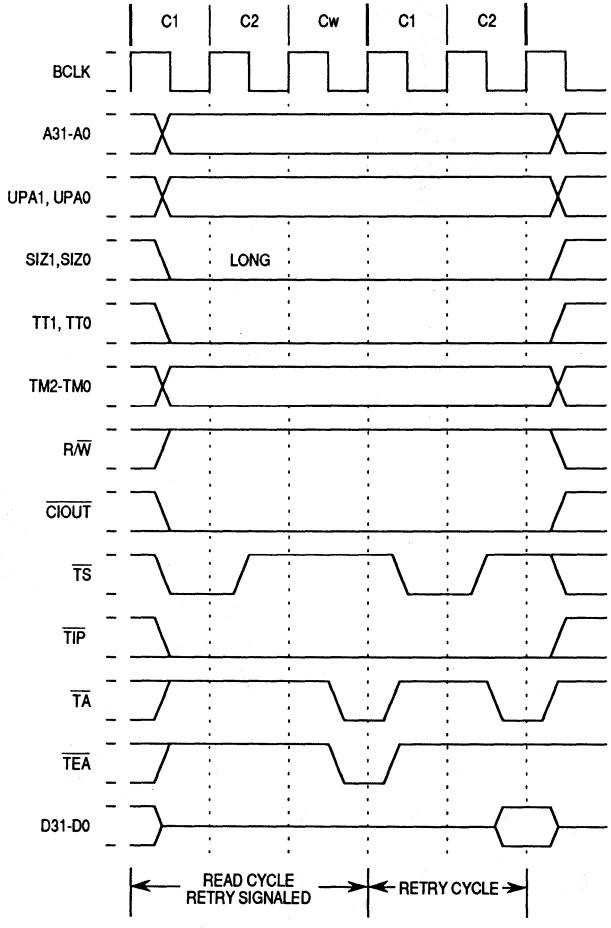


Figure 8-26. Read Cycle Retry

The processor retries any read or write cycle of a locked read-modify-write operation separately; the lock signal ($\overline{\text{LOCK}}$) remains asserted during the entire retry sequence. If the last bus cycle of a locked access is retried, the lock end signal ($\overline{\text{LOCKE}}$) remains asserted through the retry of the write cycle.

On the initial access of a line access, a retry causes the processor to retry the bus cycle as shown in Figure 8-27. However, a retry signaled during the second, third, or fourth cycle of a line transfer is recognized by the processor

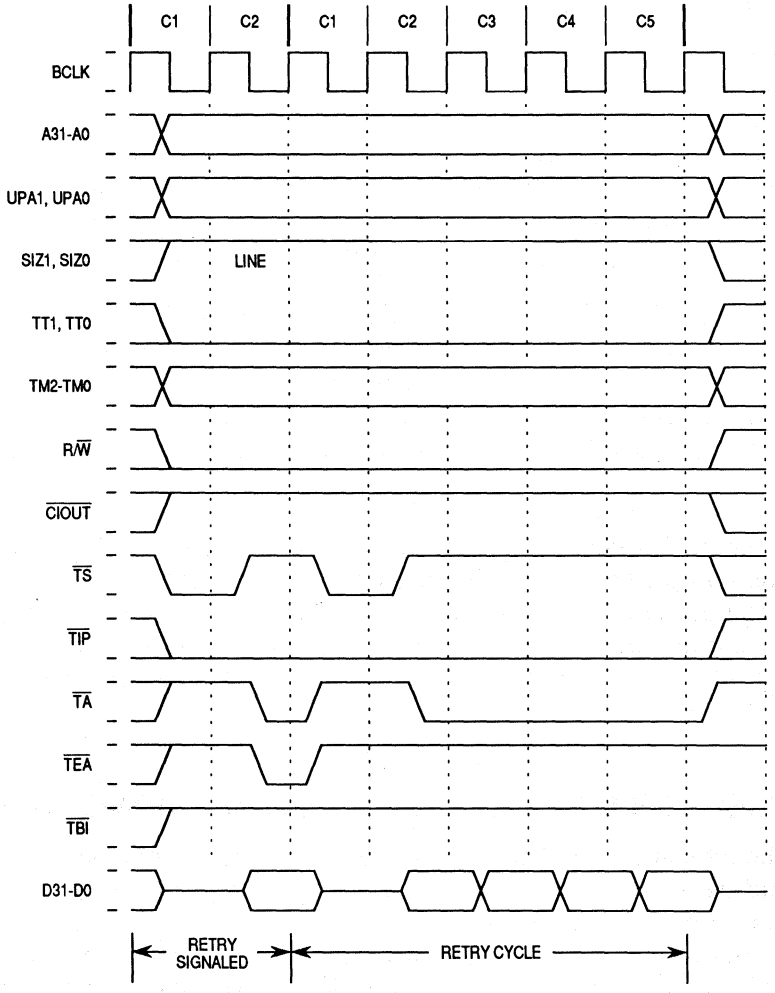


Figure 8-27. Retry Operation on Line Write

as a bus error, and causes the processor to abort the line transfer. A burst-inhibited line transfer can only be retried on the initial transfer, and aborts if a retry is signalled for any of the three longword transfers used to complete the line transfer.

Negating the bus grant signal ($\overline{BG}$) to the MC68040 while asserting both $\overline{TA}$ and $\overline{TEA}$ provides a relinquish and retry operation for any bus cycle that can be retried.

8.5.3 Double Bus Fault

When an access error or an address error occurs during the exception processing sequence for a previous access error, a previous address error, or a reset exception, the access 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 an access error exception. If a bus error occurs during the stacking operation, the second error is considered a double bus fault.

The MC68040 indicates that a double bus fault condition has occurred by continuously driving the processor status signals (PST3–PST0) with an encoded value of $5 until the processor is reset. Only an external reset operation can restart a halted processor. While the processor is halted, the external bus is released by negating $\overline{BR}$ and forcing all outputs to a high-impedance state.

A second access 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 either. The processor continues to retry the same bus cycle as long as the external hardware requests it.

8.6 ACCESS SERIALIZATION AND BUS SYNCHRONIZATION

The integer unit of the MC68040 generates access requests to the instruction and data memory units to support integer and floating-point operations. In the integer unit pipeline, accesses to the data memory unit are performed by both the ea (effective address) fetch and writeback pipeline stages, with ea fetches assigned a higher priority. This allows data read and write accesses to occur out of order, with a memory write access potentially delayed for many clocks while allowing reads generated by later instructions to complete.

The processor detects address collisions (a read access that references earlier data waiting to be written) and allows the corresponding write access to complete. A given sequence of read accesses or write accesses is completed in order, and reordering only occurs with writes relative to reads. The integer pipeline stages are shown in Figure 1-1 in **SECTION 1 INTRODUCTION**.

Another potential problem is a result of the instruction restart model used for exception processing in the MC68040. After the operand fetch for an instruction, an exception can occur that causes the instruction to be aborted, resulting in another access of the operand after the instruction is restarted. For example, an interrupt could occur after the read of the status register in an I/O device, which aborts the instruction and causes the register to be read again. If the status bits are cleared by the first read, the status information is lost and the instruction obtains incorrect data.

Both out-of-order and multiple accesses to devices that are sensitive to such accesses can be prevented by designating the memory page containing the device as noncachable I/O in the corresponding page descriptor. When the data memory unit detects an attempt to read an operand from a page designated as non-cachable I/O, it allows all pending writes to complete before beginning the external operand read. Only reads are affected by the definition of a page as noncachable versus noncachable I/O. When a write operation reaches the writeback pipeline stage (the last stage in the pipeline), all previous instructions are already complete. Once a read access to a noncachable I/O page begins, only a bus error exception on the operand read itself can cause the instruction to be aborted, preventing multiple reads.

Since write cycles can be deferred indefinitely, many subsequent instructions can be executed, resulting in seemingly non-sequential 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 because it freezes instruction execution until all pending bus cycles have completed.

An example of the NOP instruction, for this purpose, is 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 $\overline{\text{TEA}}$. 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 MC68040 cannot process the bus error until the end of the bus cycle, the external hardware can not successfully interrupted program execution. In order 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, access 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 the NOP instruction can also be used to force access serialization by placing a NOP before the instruction that reads an I/O device. This eliminates the need to specify the entire page as noncachable I/O, but does not prevent the possibility of the instruction from being aborted by an exception condition as noted earlier.

8.7 BUS ARBITRATION

The bus design of the MC68040 provides for a single bus master at any one time: either the processor or an external device. One or more of the devices on the bus can have the capability of becoming bus master. Bus arbitration is the protocol by which the processor or an external device becomes bus master. Unlike earlier members of the M68000 processor family, the MC68040 implements an arbitration method in which an external arbiter controls bus arbitration, and the processor acts as a slave device in requesting ownership of the bus from the arbiter. Since the functionality of the arbiter is defined by the user, it can be configured to support any desired priority scheme. For systems in which the processor is the only possible bus master, the bus can be continuously granted to the processor, and no arbiter is needed. Systems that include several devices that can become bus master require an arbiter to assign priorities to the devices, so that when two or more devices attempt to become bus master at the same time, the one having the highest priority becomes bus master first.

When the bus is owned by another bus master, the MC68040 is able to monitor the alternate master transfers and intervene when necessary to mantain cache coherency. This capability is discussed in more detail in **SECTION 8.8 BUS SNOOPING OPERATION.**

The bus controller in the MC68040 generates bus requests in reponse to internal requests from the instruction and data memory units, and asserts the bus arbitration signals using the protocol described in the following paragraphs. The arbitration protocol allows arbitration to be overlapped with bus activity, and requires a single dead clock when transfering bus ownership between bus masters to prevent bus contention. The three main signals used by the MC68040 for bus arbitration are bus request ($\overline{BR}$), bus grant ($\overline{BG}$), and bus busy ($\overline{BB}$). The bus arbitration unit in the MC68040 operates synchronously, and transitions between states on the rising edge of BLCK.

The MC68040 requests the bus from the arbiter by asserting the $\overline{BR}$ signal whenever an internal bus cycle request is pending, and continues to assert $\overline{BR}$ until the arbiter grants the bus to the processor, allowing the bus cycle to begin. After the bus cycle starts, the processor continues to assert $\overline{BR}$ if another bus cycle is pending, or negates it if no further accesses are required. If the bus is already granted to the processor when an internal bus cycle request is generated, $\overline{BR}$ is asserted at the same time the transfer start signal ($\overline{TS}$) asserts to indicate the start of the bus cycle, allowing the access to begin immediately. $\overline{BR}$ is always driven by the processor, and cannot be wire-ORed with other devices.

The bus grant signal ($\overline{BG}$) is asserted by the external arbiter to indicate to the processor that it has been granted the bus. If $\overline{BG}$ is negated while a bus cycle is in progress, the processor relinquishes the bus at the completion of the current bus cycle. Note that the bus controller considers the four bus cycles for a burst-inhibited line transfer to be a single bus cycle, and does not relinquish the bus until completion of the fourth transfer. The read and write portions of a locked read-modify-write sequence are divisible in the MC68040, allowing the bus to be arbitrated away during the locked sequence. For systems applications which must not allow locked sequences to be broken, the arbiter can use the bus lock signal ($\overline{LOCK}$) to detect locked accesses and prevent negation of $\overline{BG}$ to the processor during these sequences. The lock end signal ($\overline{LOCKE}$) is also provided by the processor to indicate the last write cycle of a locked sequence, allowing arbitration between back-to-back locked sequences. See **8.3.5 Locked Transfer** for a detailed description of locked bus transfers.

When the bus has been granted to the processor in response to the assertion of $\overline{BR}$, the processor monitors the bus busy signal ($\overline{BB}$) to determine when the bus cycle of the previous master has completed. After $\overline{BB}$ is negated by the alternate master, the processor asserts $\overline{BB}$ to indicate ownership of the bus and begins the bus cycle. The processor continues to assert $\overline{BB}$ until the arbiter negates $\overline{BG}$, after which $\overline{BB}$ is first negated at the completion of the current bus cycle, then forced to a high impedance state. As long as $\overline{BG}$ is asserted, $\overline{BB}$ remains asserted to indicate the bus is owned, and the processor continuously drives the bus signals. $\overline{BR}$ is negated when there are no pending accesses to allow the arbiter to grant the bus to another bus master if necessary.

Figure 8-28 is a timing diagram showing an example of the arbitration activity performed by the processor in requesting the bus from an alternate master for a single misaligned access. In clock 1, the MC68040 asserts $\overline{BR}$ to request the bus from the arbiter, which negates the alternate master's bus grant

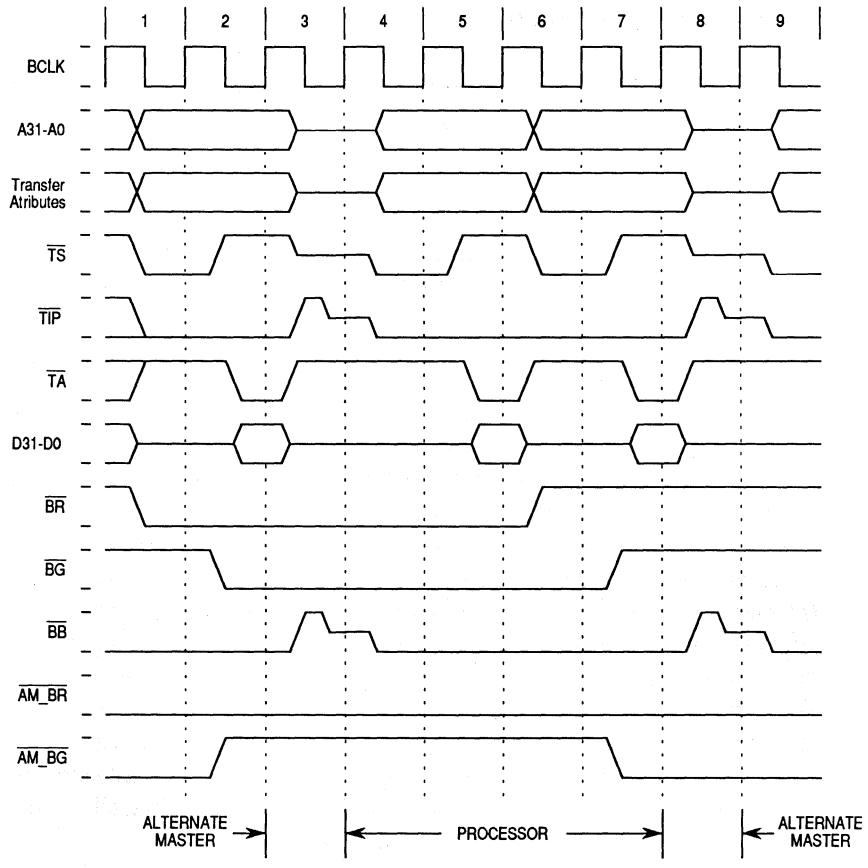


Figure 8-28. Processor Bus Request Example

signal (AM BG) in the figure) and grants the bus to the processor in clock 2 by asserting BG. During clock 3 the alternate master completes its current access and relinquishes the bus by three-stating all bus signals. Typically, the BB and TIP signals require a pullup resistor to maintain a logic "1" level between bus masters tenures, and should be actively negated by the alternate master before three-stating to minimize rise time of the signals and ensure the correct level is seen by the processor on the next BCLK rising edge. At the end of clock 3, the processor recognizes the bus grant and bus idle conditions (BG asserted and BB negated) and assumes ownership of the bus in clock 4 by asserting BB and immediately beginning a bus cycle. In clock

6 the processor begins the second bus cycle for the misaligned operand, and negates $\overline{BR}$ at this time, since no other accesses are pending. The arbiter grants the bus back to the alternate master in clock 7, which waits for the processor to relinquish the bus. The processor actively negates $\overline{BB}$ and $\overline{TIP}$ before three-stating these and all other bus signals during clock 8. Finally, the alternate master recognizes the bus grant and idle conditions at the end of clock 8 and is able to resume its bus activity in clock 9.

Figure 8-29 shows the arbitration timing for a relinquish and retry operation. The processor read access which begins in clock 1 is terminated at the end of clock 2 with a retry request and $\overline{BG}$ negated, forcing the processor to relinquish the bus and allow the alternate master to update the operand. Note that the processor reasserts $\overline{BR}$ in clock 3 since the original access is now pending again. After the operand update, the bus is granted back to the processor to allow it to retry the access beginning in clock 7.

A special case exists if $\overline{BG}$ is asserted and the processor neither owns the bus nor needs the bus for a pending access (i.e., $\overline{BB}$ is being sampled as an input, and $\overline{BR}$ is negated). If $\overline{BB}$ is negated, the processor assumes ownership of the bus and drives the address bus and transfer attribute signals with undefined values. Ownership is implicit since the processor does not drive either $\overline{BB}$ or the transfer in progress signal ($\overline{TIP}$), although $\overline{TS}$ remains negated. If $\overline{BB}$ is asserted by another bus master or $\overline{BG}$ is negated, the processor releases the bus. If an internal access request is generated, the processor assumes explicit ownership of the bus and immediately begins an access, asserting $\overline{BB}$, $\overline{BR}$, $\overline{TIP}$, and $\overline{TS}$ at the same time. Figure 8-30 shows an example of bus arbitration for a system in which the processor is the default bus master, and is granted an idle bus by the arbiter.

8.8 BUS SNOOPING OPERATION

The MC68040 has the capability of monitoring bus transfers by other bus masters and intervening in the access when required to maintain cache coherency. The process of bus monitoring and intervention is called snooping, and is controlled by the encoding of the snoop control signals (SC1–SC0) generated by the alternate master for each bus cycle, as shown in Table 8-9.

Snooping begins when the bus is granted to another bus master, and the MC68040 sees a $\overline{TS}$ assertion by the alternate master. The processor latches the level on the A31–A0, SIZ1–SIZ0, TT1–TT0, R/$\overline{W}$, and SC1–SC0 signals on the BCLK rising edge for which $\overline{TS}$ is first asserted, and evaluates the snoop

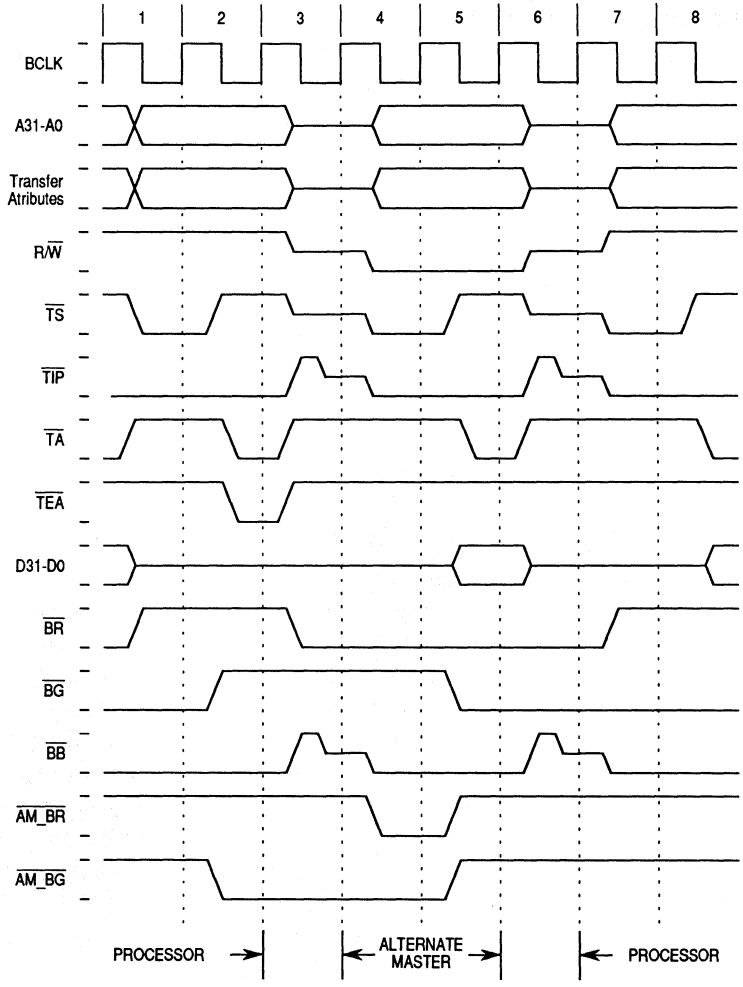


Figure 8-29. Arbitration During Relinquish and Retry

Table 8-9. Snoop Control Encoding

SC1	SC0	Requested Snoop Operation	
		Read Access	Write Access
0	0	Inhibit Snooping	Inhibit Snooping
0	1	Supply Dirty Data and Leave Dirty	Sink Byte/Word/Long-Word Data
1	0	Supply Dirty Data and Mark Line Invalid	Invalidate Line
1	1	Reserved (Snoop Inhibited)	Reserved (Snoop Inhibited)

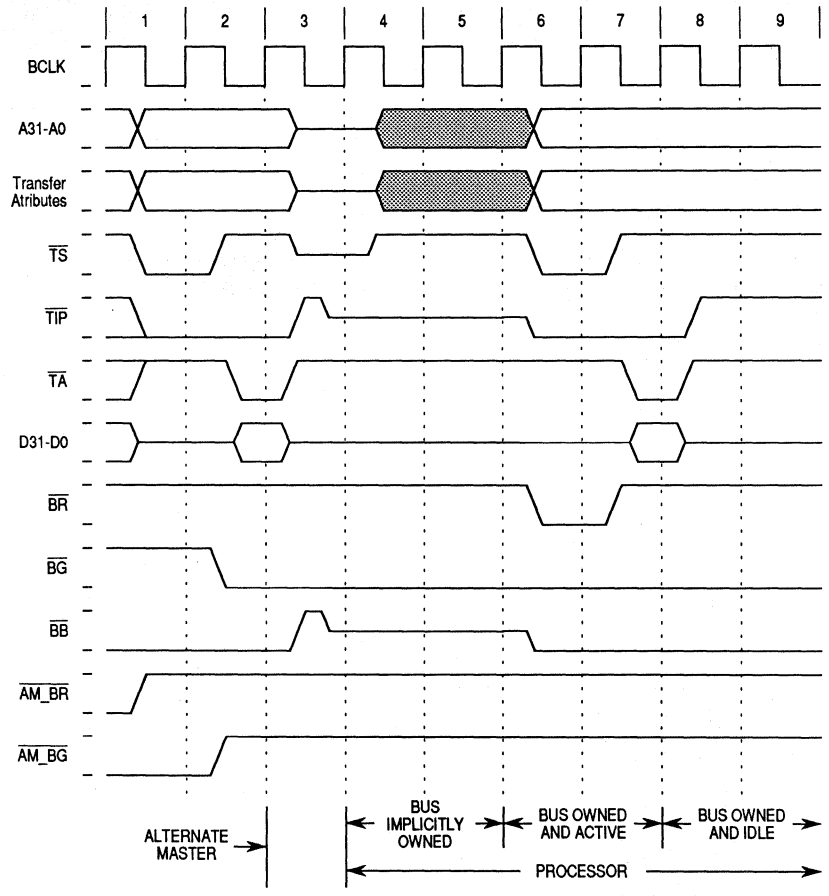


Figure 8-30. Implicit Bus Ownership

control and transfer type to determine if the access should be snooped. Only normal and MOVE16 bus transfers can be snooped. If snooping is enabled for the access, the processor inhibits memory from responding by continuing to assert the memory inhibit signal ($\overline{MI}$) while checking the internal caches for matching lines. The processor intervenes in the access only if the data cache contains a dirty line corresponding to the access, and the requested snoop operation indicates to sink data for a write or source datafor a read. If this occurs, the processor continues to inhibit memory and responds to the alternate master access as a slave device. Otherwise, $\overline{MI}$ is negated and

memory is allowed to respond and complete the access. The processor monitors the levels of $\overline{TA}$, $\overline{TEA}$, and $\overline{TBI}$ to detect normal, bus error, retry, and burst inhibit terminations. Note that for alternate master line transfers that are burst-inhibited, the MC68040 snoops each of the four resulting longword transfers individually.

In a system with multiple bus masters, memory must wait for each snooping bus master to negate its $\overline{MI}$ signal before responding to an access. Also, if the system contains multiple caching masters, then each processor must access shared data using writethrough pages to allow writes to the data to be snooped by other masters. Only one of the processors can access a given page of data using copyback cache mode, typically for data local to that processor. This also prevents multiple snooping processors from intervening in a specific access.

As a bus master, the MC68040 can be configured to request snooping operations on a page basis. The user programmable attribute signals (UPA1–UPA0) are connected to the SC1–SC0 inputs of the snooping processors. The required snooping operation is then selected for a page by appropriately programming the user attribute bits in the corresponding page descriptor.

Refer to **SECTION 6 MEMORY MANAGEMENT** for details on configuring the caching mode and user attribute bits for each memory page, and to **SECTION 7 INSTRUCTION AND DATA CACHES** for more information on the effects of snooping on the caches.

8.8.1 Snoop Inhibited Cycle

For alternate master accesses in which the SC1–SC0 signal encoding indicates that snooping is inhibited (SC1–SC0 = $0), the MC68040 immediately negates $\overline{MI}$ and allows memory to respond to the access. Snoop inhibited alternate master accesses do not affect performance of the processor, since no cache lookups are required. Figure 8-31 shows an example of snoop-inhibited operation in which an alternate master is granted the bus for an access.

8.8.2 Snoop Enabled Cycle — No Intervention Required

For alternate master accesses in which the SC1–SC0 signal encoding indicates that snooping is enabled (SC1–SC0 = $1 or $2), the MC68040 continues to assert $\overline{MI}$ while checking for a matching cache line. If intervention in the

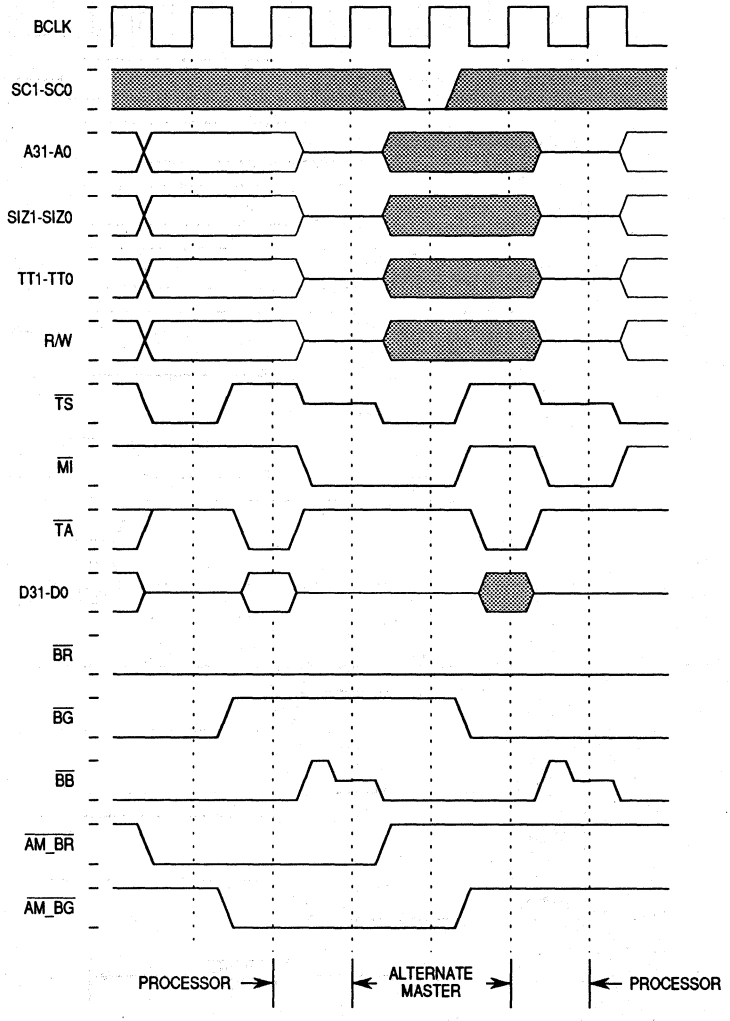


Figure 8-31. Snoop Inhibited Bus Cycle

alternate master access is not required, MI is then negated and memory is allowed to respond and complete the access. Figure 8-32 shows an example of snooping in which memory is allowed to respond. Best case timing is shown, which results in a memory access having the equivalent of two wait states. Variations in the timing required by the snooping logic to access the caches can delay the negation of MI by up to two additional clocks.

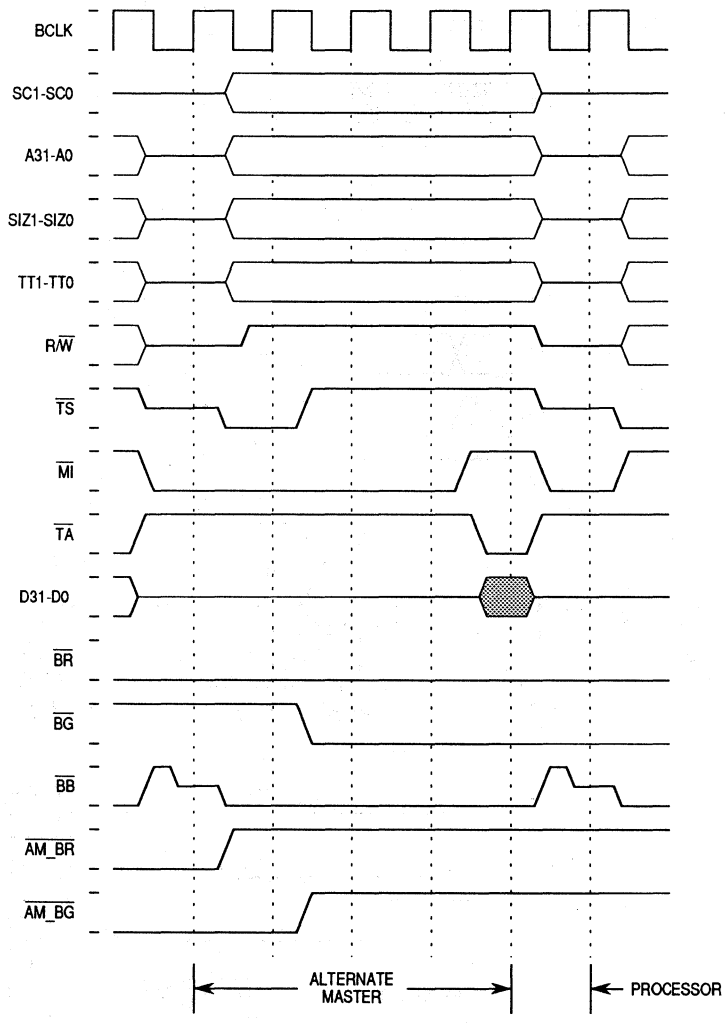


Figure 8-32. Snoop Access with Memory Response

8.8.3 Snoop Read Cycle Intervention

If snooping is enabled for a read access, and the corresponding data cache line contains dirty data, the MC68040 inhibits memory and responds to the access as a slave device to supply the requested read data. Intervention in a byte, word, or long word access is independent of which longword entry

in the cache line is dirty. Figure 8-33 shows an alternate master line read that hits a dirty line in the MC68040's data cache. $\overline{TA}$ is asserted by the processor to acknowledge the transfer of data to the alternate master, and the data bus is driven with the four longwords of data for the line. The timing shown is for best-case response time, and can include up to two additional clocks before the assertion of $\overline{TA}$ by the processor due to variations in the timing required by the snooping logic to access the caches.

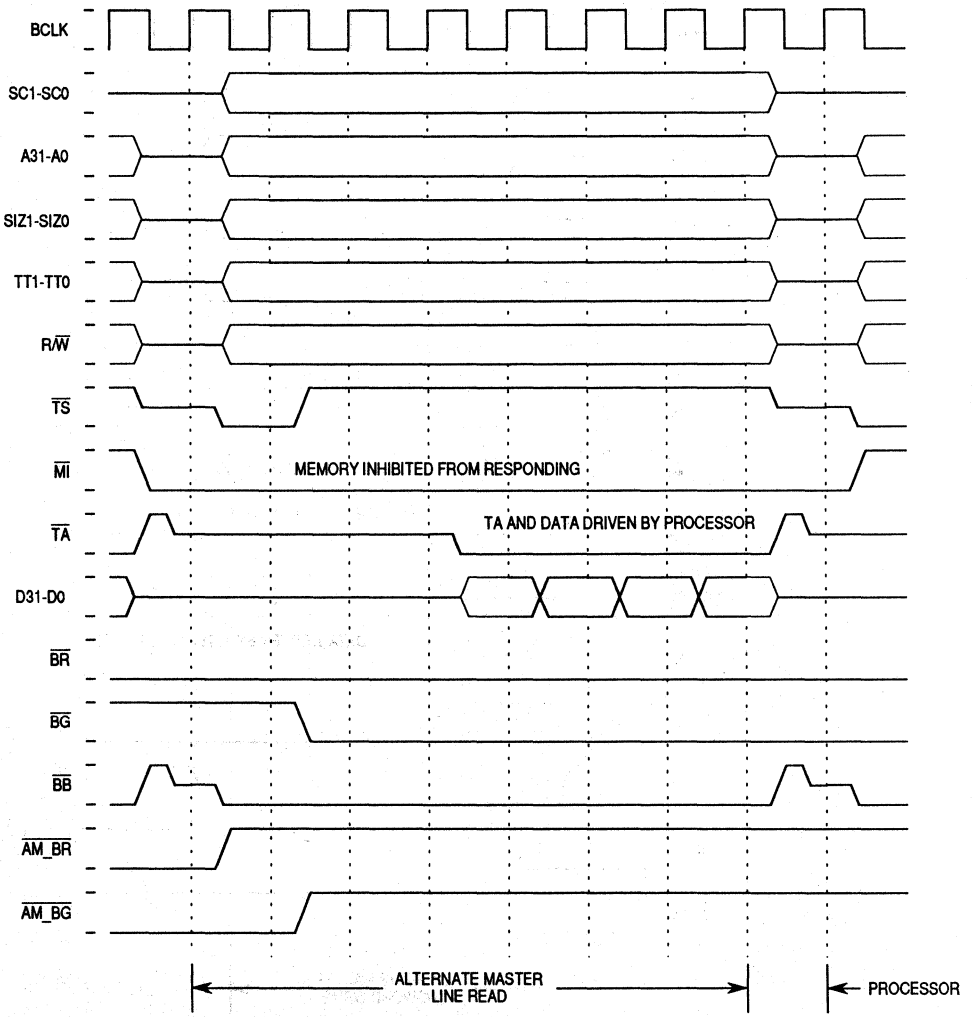


Figure 8-33. Snooped Line Read, Memory Inhibited

8.8.4 Snoop Write Cycle Intervention

If snooping with sink data is enabled for a byte, word, or longword write access, and the corresponding data cache line contains dirty data, the MC68040 inhibits memory and responds to the access as a slave device to read the data from the bus and update the data cache line. The dirty bit is set for the long word changed in the cache line. Figure 8-34 shows a longword write by an alternate master that hits a dirty line in the MC68040's data cache. $\overline{TA}$

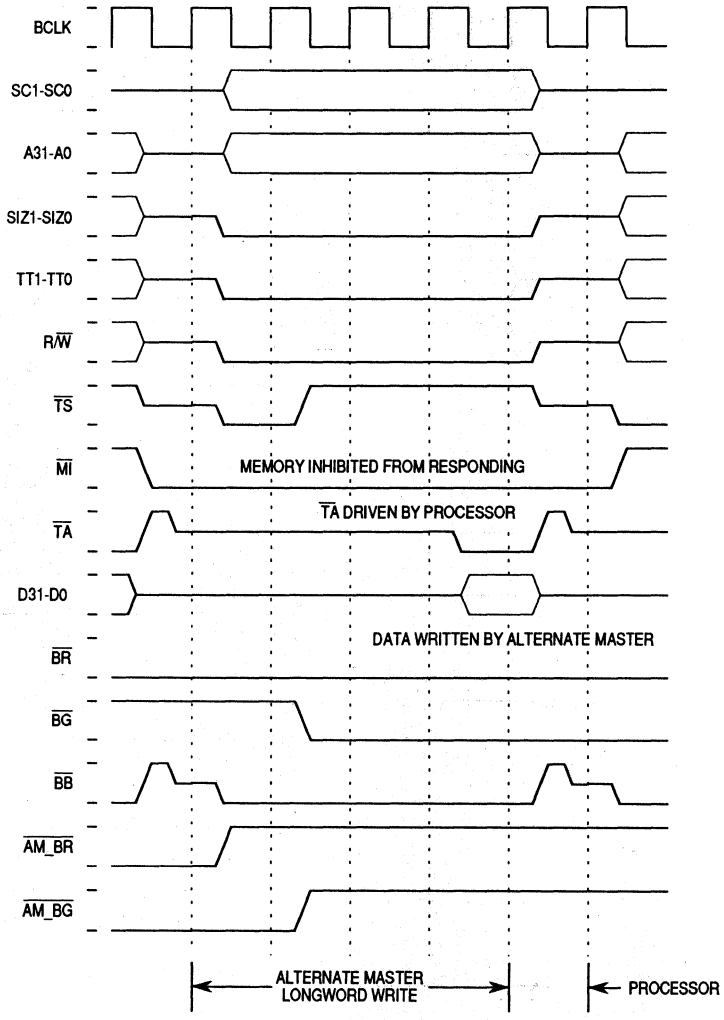


Figure 8-34. Snooped Longword Write, Memory Inhibited

is asserted by the processor to acknowledge the transfer of data from the alternate master, and the value on the data bus is read by the processor. The timing shown is for best-case response time, and can include up to two additional clocks before the assertion of $\overline{TA}$ by the processor due to variations in the timing required by the snooping logic to access the caches.

8.9 SPECIAL MODES OF OPERATION

The MC68040 supports the following three operation modes, which are selectively enabled during processor reset and remain in effect until the next processor reset. For further information refer to **8.10 RESET OPERATION**.

8.9.1 Output Buffer Impedance Selection

All output drivers in the MC68040 (with the exception of the test signal $\overline{TDO}$) can be configured to operate in either a large buffer mode (low impedance driver) or small buffer mode (high impedance driver). Large buffers have a nominal output impedance of 4 ohms for both high and low drive, and provide for minimum output delays. Signal traces driven by large buffers usually require transmission line effects to be considered in their design, including the use of signal termination. Small buffers have a nominal impedance of 30 ohms for high and low drive, resulting in longer output delays and less critical board design requirements. Refer to MC68040DH/D, *MC68040 Design Handbook* for further information on electrical specifications, buffer characteristics, and transmission line design examples.

The output drivers are configured in three groups as shown in Table 8-10. Each group of signals is configured as either large buffers or small buffers by a logic "0" or logic "1" level, respectively, on the corresponding interrupt priority level signal during processor reset.

8.9.2 Multiplexed Bus Mode

The multiplexed bus mode changes the timing of the three-state control logic for the address and data buses to support generation of a multiplexed address/data bus. When the MC68040 is operating in this mode, the address and data bus signals can be hardwired together to form a single 32-bit bus, with address and data information time-multiplexed on the bus. This minimizes the number of pins required to interface to peripheral devices without requiring additional discrete multiplexing logic. This mode is enabled during a processor reset by a logic "0" level on the cache disable signal ($\overline{CDIS}$).

Table 8-10. Output Buffer Impedance Control Groups

Signal	Output Buffers Controlled
$\overline{IPL2}$	Data Bus: D31–D0
$\overline{IPL1}$	Address Bus and Transfer Attributes: A31–A0, $\overline{CIOUT}$, $\overline{LOCK}$, $\overline{LOCKE}$, R/$\overline{W}$, SIZ1–SIZ0, TLN1–TLN0, TM2–TM0, TT1–TT0, UPA1–UPA0
$\overline{IPL0}$	Miscellaneous Control Signals: $\overline{BB}$, $\overline{BR}$, $\overline{IPEND}$, $\overline{MI}$, $\overline{PST3}$–$\overline{PST0}$, $\overline{RSTO}$, $\overline{TA}$, $\overline{TIP}$, $\overline{TS}$

NOTE:
High input level = small buffers enabled, low = large buffers enabled.

Figure 8-35 shows a line write with multiplexed bus mode enabled. The address bus drivers are enabled during clock 1 and disabled during clock 2. Later in clock 2 the data bus drivers are enabled to drive the data bus with the data to be written. The address bus is only driven for the BCLK rising edge at the start of each bus cycle.

8.9.3 Data Latch Enable Mode

The data latch enable (DLE) mode allows read data to be latched by the DLE signal, instead of by the BCLK rising edge at the end of each transfer. In some applications, this can reduce the number of clocks required to perform line burst reads. This mode is enabled during a processor reset by a logic "0" level on the MMU disable signal ($\overline{MDIS}$).

Figure 8-36 shows a conceptual block diagram of the logic used to latch the read data bus in DLE mode. Transparent latch A is controlled by the DLE signal and allows data to be latched before the rising edge of BCLK. Latch A operates transparently when DLE is at a high logic level, and latches the level on the data bus when DLE transitions to a low level. Note that the DLE signal only controls latching of the read data, and does not affect termination of the bus transfer. Edge-triggered latch B is clocked by the rising edge of BCLK, and latches the data from latch A for use by internal logic.

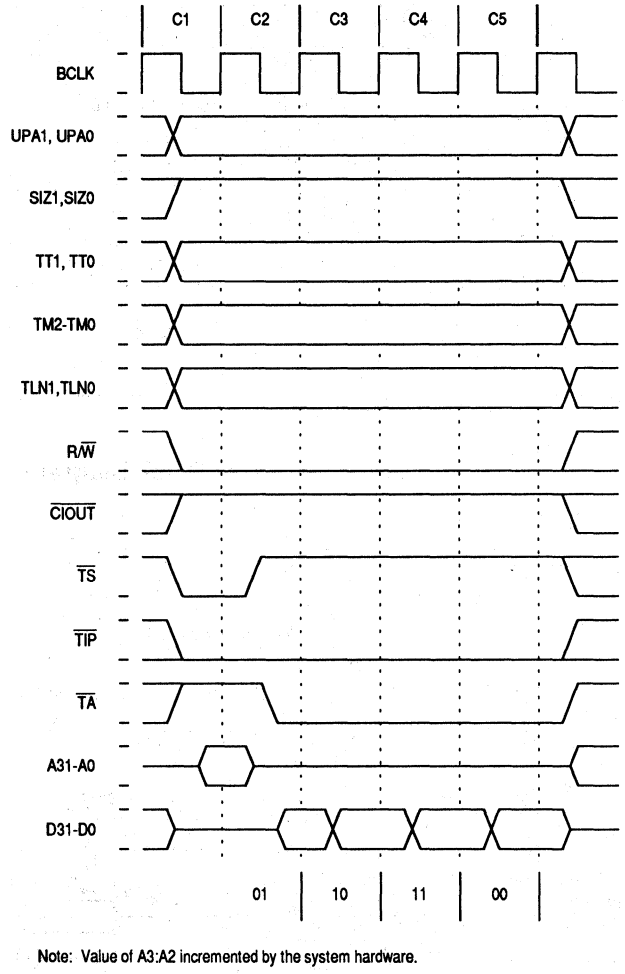


Figure 8-35. Multiplexed Address and Data Bus — Line Write

Figure 8-37 shows the data read timing for both normal operation and DLE mode. During normal operation (i.e. DLE mode disabled), latch A is always transparent, and read data is latched by the rising edge of BCLK. Data must meet setup and hold time specifications #15 and #16 in this case. When DLE mode is enabled, the data can be latched by the rising edge of BCLK or the falling edge of DLE, depending on the timing for DLE:

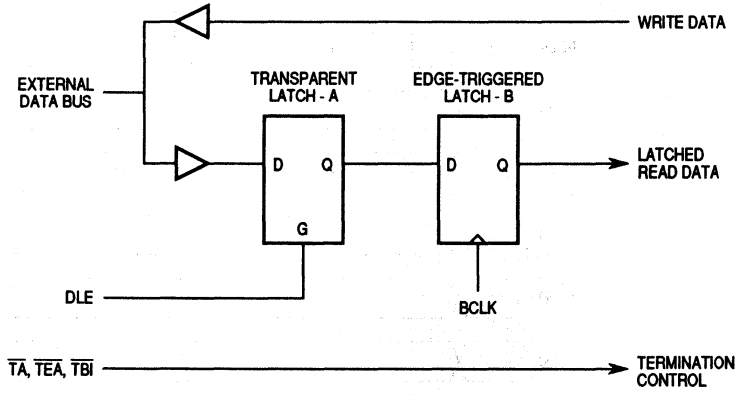


Figure 8-36. DLE Mode Block Diagram

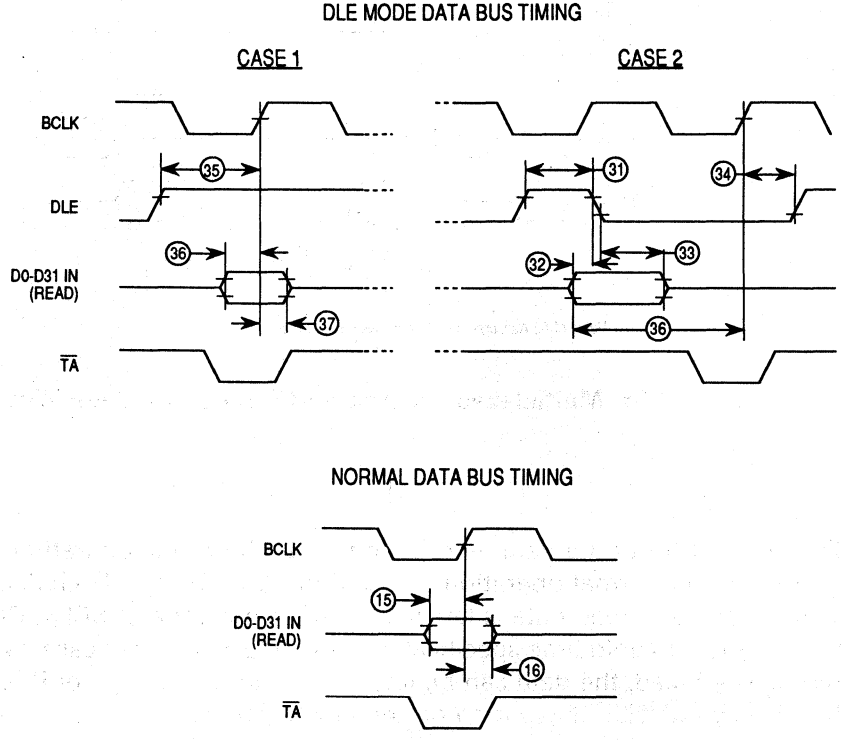


Figure 8-37. DLE versus Normal Data Read Timing

Case 1.

If DLE is negated high, and meets setup time #35 to the rising edge of BCLK when the bus read is terminated, latch A is transparent and the read data must meet setup and hold time specifications #36 and #37 to the rising edge of BCLK. Read timing is similar to normal timing for this case.

Case 2.

If DLE is asserted low, the data bus levels are latched and held internally. D31–D0 must meet setup and hold time specifications #32 and #33 to the falling edge of DLE, and can transition to a new level once DLE is asserted low. D31–D0 must still meet setup time #36 to BCLK, but not hold time #37, since the data is held valid as long as DLE remains asserted low.

8.10 RESET OPERATION

The reset input signal ($\overline{RSTI}$) is asserted by an external device to reset the processor. When power is applied to the system, external circuitry should assert $\overline{RSTI}$ for a minimum of ten BCLK cycles after V_{CC} is within tolerance. Figure 8-38 is a timing diagram of the power-on reset operation, showing

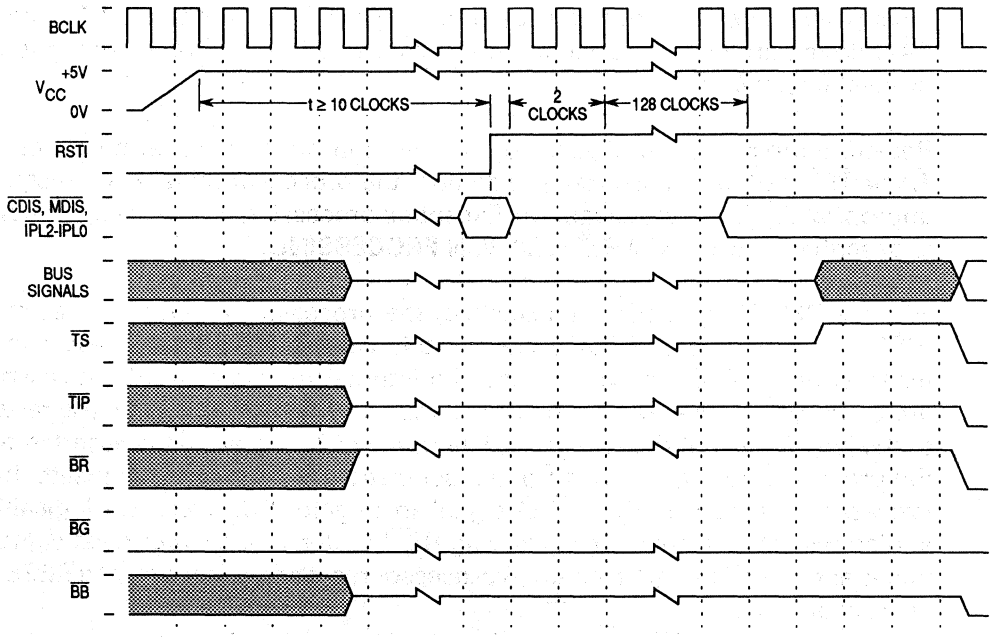


Figure 8-38. Initial Power-On Reset Timing

the relationships between V_{CC}, $\overline{RSTI}$, mode selects, and bus signals. The BCLK and PCLK clock signals are required to be stable by the time V_{CC} reaches the minimum operating specification. $\overline{RSTI}$ is internally synchronized for two BCLKS before being used, and must therefore meet the specified setup and hold times to BCLK (specifications #51 and #52) only if recognition by a specific BCLK rising edge is required.

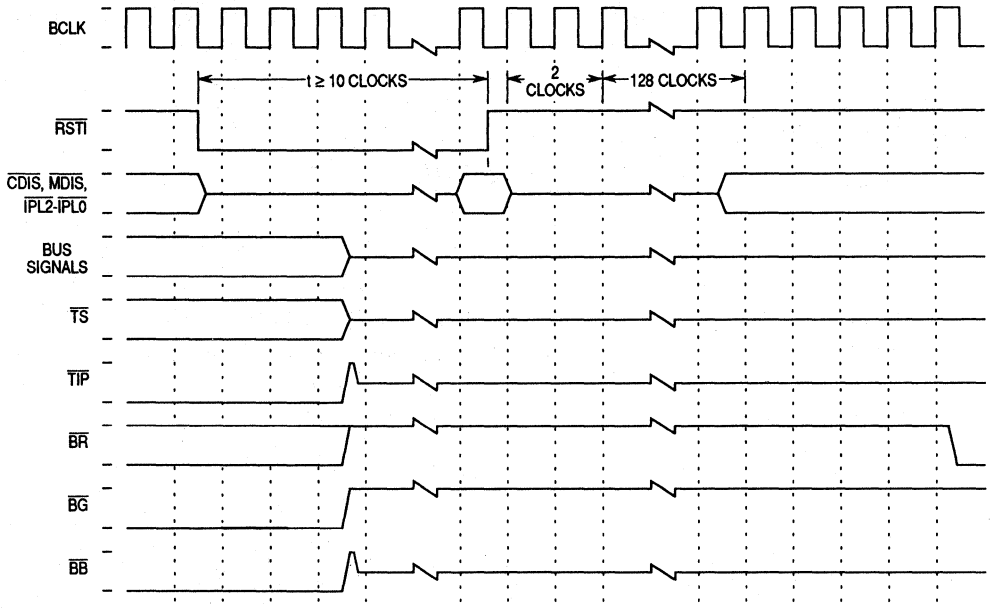
Once $\overline{RSTI}$ negates, the processor is internally held in reset for another 128 clock cycles. During the reset period, all three-statable signals three-state, and non-three-statable signals are driven to their inactive state. Once the internal reset signal negates, all bus signals continue to remain in a high-impedance state until the processor is granted the bus. After this, the first bus cycle for reset exception processing begins. In Figure 8-38 the processor assumes implicit ownership of the bus before the first bus cycle begins.

The levels on the $\overline{CDIS}$, $\overline{MDIS}$, and $\overline{IPL2}$–$\overline{IPL0}$ signals when $\overline{RSTI}$ negates are used to selectively enable the multiplexed bus mode, DLE mode, and large versus small buffer drivers. These signals should be driven to their normal levels before the end of the 128 clock internal reset period.

For processor resets after the initial power-on reset, $\overline{RSTI}$ should be asserted for at least ten clock periods. Figure 8-39 shows timing associated with a reset when the processor is executing bus cycles. Note that $\overline{BB}$ and $\overline{TIP}$ (and $\overline{TA}$ if driven during a snooped access) are asserted before transitioning to a three-state level.

Resetting the processor causes any bus cycle in progress to terminate as if $\overline{TA}$ or $\overline{TEA}$ had been asserted. In addition, the processor initializes registers appropriately for a reset exception. Exception processing for a reset operation is described in **SECTION 9 EXCEPTION PROCESSING**.

When a RESET instruction is executed, the processor drives the reset out ($\overline{RSTO}$) signal for 512 BCLK 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 $\overline{RSTO}$ signal are reset at the completion of the reset instruction. An $\overline{RSTI}$ signal that is asserted to the processor during execution of a reset instruction immediately resets the processor and causes the $\overline{RSTO}$ signal to negate. $\overline{RSTO}$ can be logically and'ed with the external signal driving $\overline{RSTI}$ to derive a system reset signal that is asserted for both an external processor reset and execution of a RESET instruction.



8

Figure 8-39. Normal Reset Timing

SECTION 9
EXCEPTION PROCESSING

Exception processing is defined as the activities performed by the processor in preparing to execute a handler routine for any condition that causes an exception. In particular, exception processing does not include execution of the handler routine itself. This section describes the processing for each type of exception, exception priorities, the return from an exception, and bus fault recovery. This section also describes the formats of the exception stack frames. For details of memory management unit (MMU) related exceptions, refer to **SECTION 6 MEMORY MANAGEMENT**.

9.1 EXCEPTION PROCESSING SEQUENCE

The MC68040 uses a restart exception processing model to minimize interrupt and instruction latency, and to reduce the size of the stack frame (compared to the frame required for a continuation model). Exceptions are recognized at each instruction boundary in the execute stage of the integer pipeline, and force later instructions which have not yet reached the execute stage to be aborted. Instructions which can not be interrupted, such as those that generate locked bus transfers or access serialized pages, are allowed to complete before exception processing begins.

Exception processing occurs in four functional steps. However, all individual bus cycles associated with exception processing (vector acquisition, stacking, etc.) are not guaranteed to occur in the order in which they are described in this section.

In the first step of exception processing, the processor makes an internal copy of the status register (SR). Then the processor sets the S bit in the SR, changing to the supervisor mode. Next, the processor inhibits tracing of the exception handler by clearing the trace enable (T1 and T0) bits. For the reset and interrupt exceptions, the processor also updates the interrupt priority mask.

In the second step, the processor determines the vector number of the exception. For interrupts, the processor performs an interrupt acknowledge cycle to obtain the vector number. For all other exceptions, internal logic

provides the vector number. This vector number is used in the last step to calculate the address of the exception vector. Throughout this section, vector numbers are given in decimal notation.

For all exceptions other than reset, the third step is to save the current processor context. The processor creates an exception stack frame on the active supervisor stack and fills it with context information appropriate for the type of exception. Other information may also be stacked, depending on which exception is being processed and the state of the processor prior to the exception. If the exception is an interrupt and the master/interrupt (M) bit of the SR is set, the processor clears the M bit in the SR, and builds a second stack frame on the interrupt stack.

The last step initiates execution of the exception handler. The processor multiplies the vector number by four to determine the exception vector offset. It adds the offset to the value stored in the vector base register (VBR) to obtain the memory address of the exception vector. Next, the processor loads the program counter (and the interrupt stack pointer (ISP) for the reset exception) from the exception vector table entry. After prefetching the first four longwords to fill the instruction pipe, the processor resumes normal processing at the address in the program counter.

All exception vectors are located in supervisor address space, and are accessed using data references. 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.

The MC68040 supports a 1024-byte vector table containing 256 exception vectors (See Table 9-1). The first 64 vectors are defined by Motorola and 192 vectors are reserved for interrupt vectors defined by the user. However, external devices may use vectors reserved for internal purposes at the discretion of the system designer.

9.2 STACK FRAMES

When the processor executes an RTE instruction, it examines the stack frame on top of the active supervisor stack to determine if it is a valid frame and what type of context restoration it requires. The MC68040 provides five different stack frames for exception processing. The set of frames include the four and six word stack frames, four word throwaway stack frame, floating-point post-instruction stack frame, and the access error stack frame. Table 9-2 summarizes the stack frames.

Table 9-1. Exception Vector Assignments

Vector Number(s)	Vector Offset (Hex)	Assignment
0	000	Reset Initial Interrupt Stack Pointer
1	004	Reset Initial Program Counter
2	008	Access Fault
3	00C	Address Error
4	010	Illegal Instruction
5	014	Integer Divide by Zero
6	018	CHK, CHK2 Instruction
7	01C	FTRAPcc, TRAPcc, TRAPV Instructions
8	020	Privilege Violation
9	024	Trace
10	028	Line 1010 Emulator (Unimplemented A-Line Opcode)
11	02C	Line 1111 Emulator (Unimplemented F-Line Opcode)
12	030	(Unassigned, Reserved)
13	034	Defined for MC68020 and MC68030, not used by MC68040
14	038	Format Error
15	03C	Uninitialized Interrupt
16–23	040–05C	(Unassigned, Reserved)
24	060	Spurious Interrpt
25	064	Level 1 Interrupt Autovector
26	068	Level 2 Interrupt Autovector
27	06C	Level 3 Interrupt Autovector
28	070	Level 4 Interrupt Autovector
29	074	Level 5 Interrupt Autovector
30	078	Level 6 Interrupt Autovector
31	07C	Level 7 Interrupt Autovector
32–47	080–0BC	TRAP #0–15 Instruction Vectors
48	0C0	FP Branch or Set on Unordered Condition
49	0C4	FP Inexact Result
50	0C8	FP Divide by Zero
51	0CC	FP Underflow
52	0D0	FP Operand Error
53	0D4	FP Overflow
54	0D8	FP Signaling NAN
55	0DC	FP Unimplemented Data Type
56	0E0	Defined for MC68030 and MC68851, not used by MC68040
57	0E4	Defined for MC68851, not used by MC68040
58	0E8	Defined for MC68851, not used by MC68040
59–63	0EC–0FC	(Unassigned, Reserved)
64–255	100–3FC	User Defined Vectors (192)

When the MC68040 writes or reads a stack frame, it uses long word operand transfers wherever possible. Using a long-word-aligned stack pointer greatly enhances exception processing performance. The processor does not necessarily read or write the stack frame data in sequential order. The system software should not depend on a particular exception generating a particular

Table 9-2. Exception Stack Frames (Sheet 1 of 2)

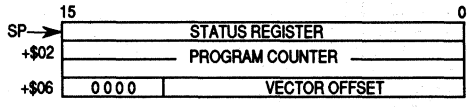
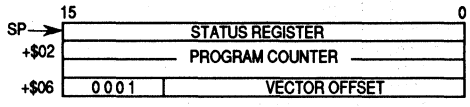
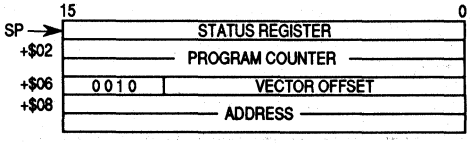
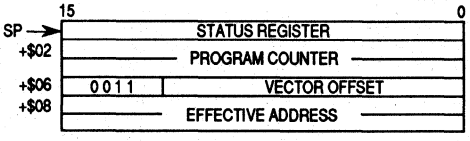
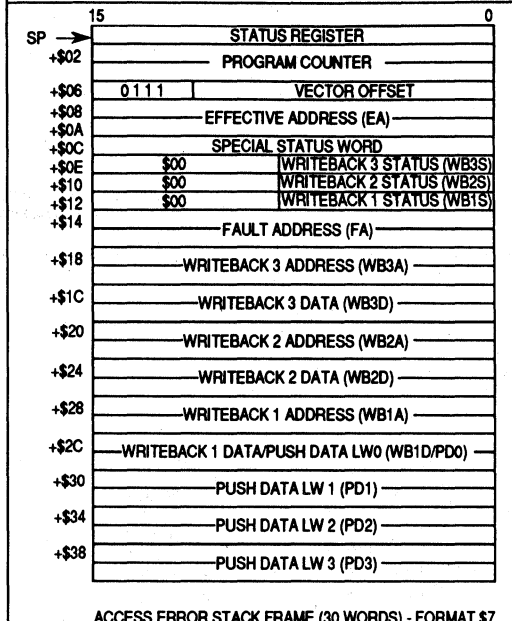
Stack Frames	Exception Types (Stacked PC Points to)
 FOUR WORD STACK FRAME - FORMAT $0	• Interrupt (Next Instruction) • Format Error RTE or FRESTORE instruction • TRAP #N (Next instruction) • Illegal Instruction (Illegal instruction) • A-Line Instruction (A-line Instruction) • F-Line Instruction (F-line Instruction) • Privilege Violation (First word of instruction causing Privilege Violation) • Floating Point Pre-Instruction (Floating-point instruction that returned pre-instruction exception)
 THROWAWAY FOUR WORD STACK FRAME - FORMAT $1	• Created on Interrupt Stack during interrupt exception processing when transition from master state to interrupt state occurs (Next instruction — same as on master stack)
 SIX WORD STACK FRAME - FORMAT $2	• CHK (Next instruction) • CHK2 • TRAPcc ADDRESS • FTRAPcc is the address of the • TRAPV instruction that caused the • Trace exception • Zero Divide • Unimplemented FP Instruction (Next instruction) ADDRESS is the calculated effective address for the FP instruction • Address Error (Instruction that caused the address error) ADDRESS is the reference address
 FLOATING-POINT POST-INSTRUCTION STACK FRAME - FORMAT $3	• Floating-point Post-Instruction (Next instruction) EFFECTIVE ADDRESS is the calculated effective address for the FP instruction

Table 9-2. Exception Stack Frames (Sheet 2 of 2)

Stack Frames	Exception Types (Stacked PC Points to)
	•Access Fault (Next instruction) ATC fault or bus error

9

9.3 INTEGER UNIT EXCEPTIONS

The following paragraphs describe the external interrupt exception and the different types of exceptions generated by the MC68040 integer unit. The following exceptions are discussed:

- Reset
- Access Fault
- Address Error
- Instruction Trap
- Illegal and Unimplemented Instructions
- Unimplemented Floating Point Instructions
- Privilege Violation
- Trace
- Format Error

9.3.1 Reset Exception

Assertion of the reset in ($\overline{\text{RSTI}}$) input signal causes a reset exception. The reset exception has the highest priority of any exception; it provides for system initialization and recovery from catastrophic failure. It aborts any processing in progress when it is recognized, and that processing cannot be recovered. Figure 9-1 is a flowchart of the reset exception, which performs the following operations:

1. Clears both trace bits (T1 and T0) in the SR to disable tracing.
2. Places the processor in the interrupt mode of the supervisor privilege mode by setting the supervisor (S) bit and clearing the master (M) bit in the SR.
3. Sets the processor interrupt priority mask to the highest priority level (level seven).
4. Initializes the VBR to zero ($00000000).
5. Clears the enable bits for the on-chip caches in the cache control register.
6. Clears the enable bit and sets the page size bit (selecting 8K pages) in the translation control register (TCR). Clears the enable bit in each of the four transparent translation registers.
7. Generates a vector number to reference the reset exception vector (two long words) at offset zero in the supervisor address space.
8. Loads the first long word of the reset exception vector into the ISP.
9. Loads the second long word of the reset exception vector into the program counter (PC).
10. Prefetches the first four longwords beginning at the memory location pointed to by the PC.

After the initial instruction prefetches, program execution begins at the address in the PC. The reset exception does not flush the address translation caches (ATCs), invalidate entries in the instruction or data caches, nor does it save the value of either the PC or the SR.

If a access fault or address error occurs during the exception processing sequence for a reset, a double bus fault is generated. The processor halts, and the processor status (PST3–PST0) signals indicate 0111.

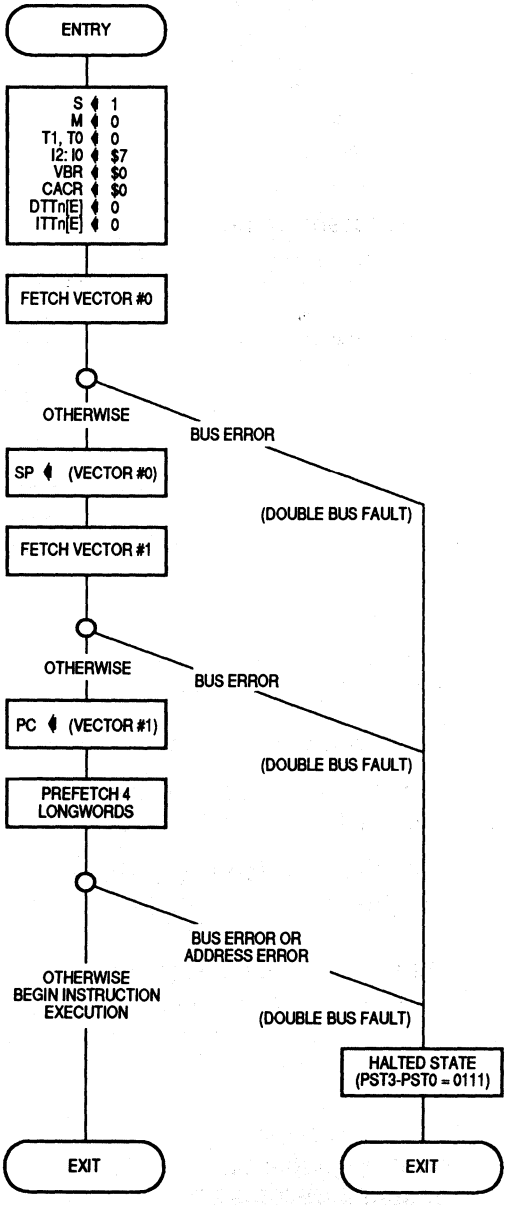


Figure 9-1. Reset Operation Flowchart

Execution of the reset instruction does not cause a reset exception, nor does it affect any internal registers, but it does cause the MC68040 to assert the reset out ($\overline{\text{RSTO}}$) signal, resetting all external devices.

9.3.2 Access Fault Exception

An access fault exception occurs when a data access or instruction prefetch access faults due to either an external bus error or an internal address translation fault. Both faults are treated identically, and can be distinguished by the access fault exception handler by a status bit in the access fault stack frame.

External logic can abort a bus cycle and signal a bus error by asserting the $\overline{\text{TEA}}$ input signal. An access fault exception may or may not be taken immediately, depending on whether the faulted access referenced data specifically required by the execution units, and whether any other exceptions occur in allowing the execution pipeline to idle. A bus error on a data write access always results in an access fault exception, and the processor begins exception processing immediately. A bus error on a data read also causes exception processing to begin immediately if the access is a byte, word, or long word access, or if the bus error occurs on the first transfer of a line read. Bus errors on the second, third, or fourth transfers for a data line read cause the transfer to be aborted, but result in a bus error only if the execution unit is specifically requesting the long word being transfered. For example, if a misaligned operand spans the first two long words in the line being read, a bus error on the second transfer causes an exception, but a bus error on the third or last transfer does not (unless the execution unit has generated another operand access which references data in these transfers).

Bus errors which occur during instruction prefetches are deferred until the processor attempts to use the prefetched information. For instance, if a bus error occurs while prefetching other instructions after a change of flow instruction (BRA, JMP, JSR, TRAP#n, etc.), the exception condition is cleared by execution of the new instruction flow. This also applies to the not-taken branch for a conditional branch instruction, even though both sides of the branch are decoded.

An access fault exception also occurs when the data MMU or instruction MMU detects that a successful address translation is not possible because the page is write protected, supervisor-only, or non-resident. Furthermore, when an ATC miss occurs, the processor searches the translation tables in memory for the mapping, and then retries the access. If a valid translation

for the logical address is not available due to a problem encountered during the table search, a bus error exception occurs when the aborted access is retried. The problem encountered could be either an invalid descriptor, or the assertion of the $\overline{TEA}$ signal during a bus cycle used to access the translation tables. A miss in the ATC causes the processor to automatically initiate a table search but does not cause a bus error exception unless one of the specific conditions mentioned above is encountered.

When an exception is detected, all parts of the execution unit are either forced or allowed to idle, at which time the highest priority exception is taken. Lower priority exceptions can be regenerated on the return from exception either by restarting the instruction or by the supervisor cleanup routine. Instruction ATC faults and bus errors are reported after all other pending integer instructions complete execution. If an exception is generated during completion of the earlier instructions, the pending instruction fault is cleared and the new exception is serviced first. The processor restarts the pending prefetch after completing exception handling for the earlier instructions, and takes a bus error exception if the access faults again. For data access faults the processor aborts current instruction execution, waits for the current instruction prefetch bus cycle to complete (if a data ATC fault was detected), then begins exception processing immediately.

The processor begins exception processing for a bus error by making an internal copy of the current SR. The processor then enters the supervisor mode , and clears T1 and T0. The processor generates exception vector number 2, for the bus error vector. It saves the vector offset, PC, and the internal copy of the SR on the stack. The saved PC value is the logical address of the instruction that was executing at the time the fault was detected. This is not necessarily the instruction that initiated the bus cycle, since the processor overlaps execution of instructions. The processor also saves information to allow continuation after a fault for a MOVEM instruction and to support other pending exceptions. The fault address and pending writeback information is saved. The information saved on the stack is sufficient to identify the cause of the bus fault, complete pending writebacks, and recover from the error. Exception handler must complete the pending writebacks. Up to three writebacks can be pending for push errors and data access errors.

If a bus error occurs during the exception processing for a bus error, address error, or reset, or while the processor is loading internal state information from the stack during the execution of an RTE instruction, a double bus fault occurs and the processor enters the halted state as indicated by the PST3–PST0 signals equaling 0111. In this case, the processor does not attempt to alter the current state of memory. Only an external reset can restart a processor halted by a double bus fault.

The supervisor stack has special requirements to ensure that exceptions can be stacked. The stack must be resident with correct protection in the direction of growth, to ensure that exception stacking never has a bus error or ATC fault. Memory pages allocated to the stack that are higher in memory(than the current stack pointer) may be nonresident since an RTE or FRESTORE instruction can check for residency and trap before restoring the state.

A special case exists for systems that allow arbitration of the processor bus during locked transfer sequences. If a locked translation table update could be bus errored by the arbiter due to an improperly broken lock, any pages touched by exception stack operations must have the U bit set in the corresponding page descriptor to prevent the occurrence of the locked access during translation table searches.

9.3.3 Address Error Exception

An address error exception occurs when the processor attempts to prefetch an instruction from an odd address. (This includes the case of a conditional branch instruction with an odd branch offset, that is not taken.) A prefetch bus cycle is not executed and the processor begins exception processing after the currently executing instructions complete. If another exception is generated by the completion of these instructions, the address error exception is deferred and the new exception is serviced. After exception processing for the address error commences, the sequence is the same as that for bus error exceptions , except that the vector number is 3 and the vector offset in the stack frame refers to the address error vector. A type $2 stack frame is generated that contains the address of the instruction that caused the address error and the address itself (with bit zero of the address cleared). If an address error occurs during the exception processing for a bus error, address error, or reset, a double bus fault occurs.

9.3.4 Instruction Trap Exception

Certain instructions are used to explicitly cause trap exceptions. The TRAP#n instruction always forces an exception, and is useful for implementing system calls in user programs. The TRAPcc, FTRAPcc, TRAPV, CHK, and CHK2 instructions force exceptions if the user program detects an error, which may be an arithmetic overflow or a subscript value that is out of bounds. The DIVS and DIVU instructions force exceptions if a division operation is attempted with a divisor of zero.

When a trap exception occurs, the processor copies the internally SR , enters the supervisor mode, and clears T1 and T0. The processor generates a vector number according to the instruction being executed. Vector 5 is for DIVx, vector 6 is for CHK and CHK2, and vector 7 is for FTRAPcc, TRAPcc, and TRAPV instructions. For the TRAP#n instruction, the vector number is 32 plus n. The stack frame saves the trap vector offset, the PC, and the internal copy of the SR on the supervisor stack. The saved value of the PC is the logical address of the instruction following the instruction that caused the trap. For all instruction traps other than TRAP#n, a pointer to the instruction that caused the trap is also saved. Instruction execution resumes at the address in the exception vector after the required instruction prefetches.

9.3.5 Illegal Instruction and Unimplemented Instruction Exceptions

An illegal instruction is an instruction that contains any bit pattern that does not correspond to the bit pattern of a valid MC68040 instruction, or a MOVEC instruction with an undefined register specification field in the first extension word. An illegal instruction exception corresponds to vector number 4, and occurs when the processor attempts to execute an illegal instruction.

An illegal instruction exception is also taken after a breakpoint acknowledge bus cycle is terminated, whether by the assertion of the transfer acknowlege ($\overline{TA}$) or the transfer error acknowlege ($\overline{TEA}$) signal.

Instruction word patterns with bits [15:12] equal to $A are referred to as unimplemented instructions with A-line opcodes. When the processor attempts to execute an unimplemented instruction with an A-line opcode, an exception is generated with vector number 10, permitting efficient emulation of unimplemented instructions.

Instructions that have bits [15:12] of the first word equal to $F and do not correspond to legal instructions for the MC68040 or the MC68881/MC68882 are treated as unimplemented instructions with F-line opcodes when execution is attempted. The exception vector number for an unimplemented instruction with an F-line opcode is number 11.

Exception processing for illegal and unimplemented instructions is similar to that for instruction traps. When the processor has identified an illegal or unimplemented instruction, it initiates exception processing instead of attempting to execute the instruction. The processor copies the SR, enters the supervisor mode, and clears the trace bits, disabling further tracing. The processor generates the vector number, either 4, 10, or 11, according to the

exception type. The illegal or unimplemented instruction vector offset, current PC, and copy of the SR are saved on the supervisor stack, with the saved value of the PC being the address of the illegal or unimplemented instruction. Instruction execution resumes at the address contained in the exception vector. It is the responsibility of the handling routine to adjust the stacked program counter if the instruction is emulated in software or is to be skipped on return from the handler.

9.3.6 Unimplemented Floating-Point Instruction Exception

Instructions that correspond to legal MC68881/MC68882 instructions but are not implemented in the MC68040 are defined as unimplemented floating point instructions. Like other unimplemented instructions, an F-line exception is generated when an unimplemented floating-point instruction is encountered. To aid in emulation of these instructions, the processor partially decodes the instruction to determine the effective address of the memory operand if required, and fetches the operand before taking the F-line exception. By performing an FSAVE to access the internal floating-point unit (FPU) state, the exception handler has access to all the information required to emulate the instruction without accessing user memory.

Exception processing for unimplemented floating-point instructions is slightly different from that for other unimplemented instructions. A longer stack frame is created that also contains the calculated effective address of the memory operand. The stacked PC points to the next instruction to be executed after the unimplemented floating point instruction, and the actual address of the unimplemented floating-point instruction is available to the exception handler in the FSAVE instruction state frame. More detailed information on unimplemented floating-point instructions is contained in **9.8.1 Unimplemented Floating-Point Instructions**.

9.3.7 Privilege Violation Exception

In order to provide system security, the instructions listed in Table 9-3 are privileged. An attempt to execute one of the privileged instructions while at the user mode causes a privilege violation exception.

Exception processing for privilege violations is similar to that for illegal instructions. When the processor identifies a privilege violation, it begins exception processing before executing the instruction. The processor copies the SR, enters the supervisor mode, and clears T1 and T0. The processor generates vector number 8, saves the privilege violation vector offset and

Table 9-3. Privileged Instructions

ANDI to SR	MOVEC
CINV	MOVES
CPUSH	ORI TO SR
EORI to SR	PFLUSH
FRESTORE	PTEST
FSAVE	RESET
MOVE from SR	RTE
MOVE to SR	STOP
MOVE USP	

the current PC value, and the internal copy of the SR on the supervisor stack. The saved value of the PC is the logical address of the first word of the instruction that caused the privilege violation. Instruction execution resumes after the required prefetches from the address in the privilege violation exception vector.

9.3.8 Trace Exception

To aid in program development, the M68000 processors include an instruction- by-instruction tracing capability. The MC68040 can be programmed to trace all instructions or only instructions that change program flow. In the trace mode, an instruction generates a trace exception after it completes execution, allowing a debugger program to monitor execution of a program.

The T1 and T0 bits in the supervisor portion of the SR control tracing. The state of these bits when an instruction begins execution determines whether the instruction generates a trace exception after the instruction completes. Clearing the T1 bit and setting the T0 bit causes an instruction that forces a change of flow to take a trace exception. Instructions that increment the PC normally do not take the trace exception. Instructions that are traced in this mode include all branches, jumps, instruction traps, and returns. This mode also includes SR manipulations, because the processor must re-prefetch instruction words to fill the pipe again any time an instruction that can modify the SR is executed. Table 9-4 shows the different trace modes.

Table 9-4. Tracing Control

T1	T0	Tracing Function
0	0	No Tracing
0	1	Trace on Change of Flow (BRA, JMP, etc.)
1	0	Trace on Instruction Execution (Any Instruction)
1	1	Undefined, Reserved

In general terms, a trace exception is an extension to the function of any traced instruction. The execution of a traced instruction is not complete until the trace exception processing is completed. If an instruction does not complete due to a bus error or address error exception, trace exception processing is deferred until after the execution of the suspended instruction is resumed and the instruction execution completes. If an interrupt is pending at the completion of an instruction, the trace exception processing occurs before the interrupt exception processing starts. If an instruction forces an exception as part of its normal execution, the forced exception processing occurs before the trace exception is processed.

When the processor is in the trace mode and attempts to execute an illegal or unimplemented instruction, that instruction does not cause a trace exception since it is not executed. This is of particular importance to an instruction emulation routine that performs the instruction function, adjusts the stacked program counter to skip the unimplemented instruction, and returns. Before returning, the trace bits of the SR on the stack should be checked. If tracing is enabled, the trace exception processing should be emulated also, in order for the trace exception handler to account for the emulated instruction.

The exception processing for a trace starts at the end of normal processing for the traced instruction, and before the start of the next instruction. The processor makes an internal copy of the SR, and enters the supervisor mode. It also clears the T0 and T1 bits of the SR, disabling further tracing. The processor supplies vector number 9 for the trace exception, and saves the trace exception vector offset, PC value, and the copy of the SR on the supervisor stack. The saved value of the PC is the logical address of the next instruction to be executed. Instruction execution resumes after the required prefetches from the address in the trace exception vector.

The STOP instruction does not perform its function when it is traced. A STOP instruction that begins execution with T1 = 1 and T0 = 0 forces a trace exception after it loads the SR. Upon return from the trace handler routine, execution continues with the instruction following the STOP, and the processor never enters the stopped condition.

9.3.9 Format Error Exception

Just as the processor checks that prefetched instructions are valid, the processor also performs some checks of data values for control operations. The RTE instruction checks the validity of the stack format code. For FPU state frames, the FRESTORE instruction compares the internal version number of

the processor to that contained in the state frame. This check ensures that the processor can correctly interpret internal FPU state information from the state frame.

If any of the checks previously described determine that the format of the data is improper, the instruction generates a format error exception. This exception saves a format $0 stack frame, generates exception vector number 14, and continues execution at the address in the format exception vector. The stacked PC value is the logical address of the instruction that detected the format error.

9.3.10 Interrupt Exceptions

When a peripheral device requires the services of the MC68040, or is ready to send information that the processor requires, it may signal the processor to take an interrupt exception. The interrupt exception transfers control to a routine that responds appropriately.

The peripheral device uses the active low interrupt priority level signals ($\overline{IPL0}$–$\overline{IPL2}$) to signal an interrupt condition to the processor and to specify the priority of that condition. The three signals encode a value of zero through seven ($\overline{IPL0}$ is the least-significant bit). High levels on all three signals correspond to no interrupt requested (level 0) and low levels on $\overline{IPL0}$–$\overline{IPL2}$ correspond to interrupt request level 7. Values one through seven specify one of seven levels of interrupts; level seven has the highest priority. External circuitry can chain or otherwise merge signals from devices at each level, allowing an unlimited number of devices to interrupt the processor.

The $\overline{IPL0}$–$\overline{IPL2}$ interrupt signals must maintain the interrupt request level until the MC68040 acknowledges the interrupt in order to guarantee that the interrupt is recognized. The MC68040 continuously samples the $\overline{IPL0}$–$\overline{IPL2}$ signals on consecutive rising edges of BCLK in order to synchronize and debounce these signals. An interrupt request that is held constant for two consecutive clock periods is considered a valid input. Although the protocol requires that the request remain until the processor runs an interrupt acknowledge cycle for that interrupt value, an interrupt request that is held for as short a period as two clock cycles could be recognized.

The SR of the MC68040 contains an interrupt priority mask (I2, I1, I0, bits 10–8). The value in the interrupt mask is the highest priority level that the processor ignores. When an interrupt request has a priority higher than the value in the mask, the processor makes the request an interrupt pending. Figure 9-2 is a flowchart of the procedure for making an interrupt pending.

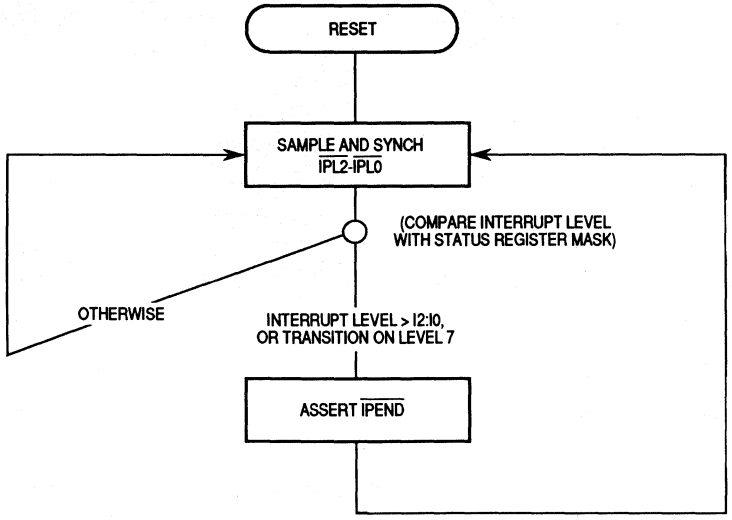


Figure 9-2. Interrupt Pending Procedure

When several devices are connected to the same interrupt level, each device should hold its interrupt priority level constant until its corresponding interrupt acknowledge cycle to ensure that all requests are processed.

9

Table 9-5 lists the interrupt levels, the states of $\overline{IPL2}$–$\overline{IPL0}$ that define each level, and the mask value that allows an interrupt at each level.

Table 9-5. Interrupt Levels and Mask Values

Requested Interrupt Level	Control Line Status			Interrupt Mask Level Required for Recognition
	IPL2	IPL1	IPL0	
0*	High	High	High	N/A*
1	High	High	Low	0
2	High	Low	High	0–1
3	High	Low	Low	0–2
4	Low	High	High	0–3
5	Low	High	Low	0–4
6	Low	Low	High	0–5
7	Low	Low	Low	0–7

*Indicates that no interrupt is requested.

Priority level seven, the non-maskable interrupt (NMI), is a special case. Level seven interrupts cannot be masked by the interrupt priority mask and they are transition sensitive. The processor recognizes an interrupt request each

time the external interrupt request level changes from some lower level to level seven, regardless of the value in the mask. Figure 9-3 shows two examples of interrupt recognitions, one for level six and one for level seven. When the MC68040 processes a level 6 interrupt, the SR mask is automatically updated with a value of 6 before entering the handler routine so that subsequent level six interrupts are masked. Provided that no instruction lowers the mask value is executed, the external request can be lowered to level three and then raised back to level six and a second level six interrupt is not processed. However, if the MC68040 is handling a level seven interrupt (status register mask set to 7) and the external request is lowered to level three and than raised back to level seven, a second level seven interrupt is processed. The second level seven interrupt is processed because the level seven interrupt is transition sensitive. A level seven interrupt is also generated by a level comparison if the request level and mask level are at seven and the priority mask is then set to a lower level (with the MOVE to SR or RTE instruction, for example). As shown in Figure 9- 3 for level six interrupt request level and mask level, this is the case for all interrupt levels.

	EXTERNAL $\overline{IPL2}$–$\overline{IPL0}$		SR MASK (12:10)		ACTION	
LEVEL 6 EXAMPLE						
	100 ($3)		101 ($5)			INITIAL CONDITIONS
IF	001 ($6)	THEN	110 ($6)	AND	LEVEL 6 INTERRUPT	(LEVEL COMPARISON)
IF	100 ($3)	AND STILL	110 ($6)	THEN	NO ACTION	
IF	001 ($6)	AND STILL	110 ($6)	THEN	NO ACTION	
IF	001 ($6)	AND RTE SO THAT	101 ($5)	THEN	LEVEL 6 INTERRUPT	(LEVEL COMPARISON)
LEVEL 7 EXAMPLE						
	100 ($3)		101 ($5)			INITIAL CONDITIONS
IF	001 ($7)	THEN	111 ($7)	AND	LEVEL 7 INTERRUPT	(TRANSITION)
IF	100 ($3)	AND STILL	111 ($7)	THEN	NO ACTION	
IF	000 ($7)	AND STILL	111 ($7)	THEN	LEVEL 7 INTERRUPT	(TRANSITION)
IF	000 ($7)	AND RTE SO THAT	101 ($5)	THEN	LEVEL 7 INTERRUPT	(LEVEL COMPARISON)

Figure 9-3. Interrupt Recognition Examples

Note that a mask value of six and a mask value of seven both inhibit request levels of one through six from being recognized. In addition, neither masks a transition to an interrupt request level of seven. The only difference between mask values of six and seven occurs when the interrupt request level is seven and the mask value is seven. If the mask value is lowered to six, a second level seven interrupt is recognized.

The MC68040 asserts $\overline{\text{IPEND}}$ when it makes an interrupt request pending. Figure 9-4 shows the assertion of $\overline{\text{IPEND}}$ relative to the assertion of an interrupt level on the $\overline{\text{IPL}}$ lines. $\overline{\text{IPEND}}$ signals to external devices that an interrupt exception will be taken at an upcoming instruction boundary (following any higher-priority exception).

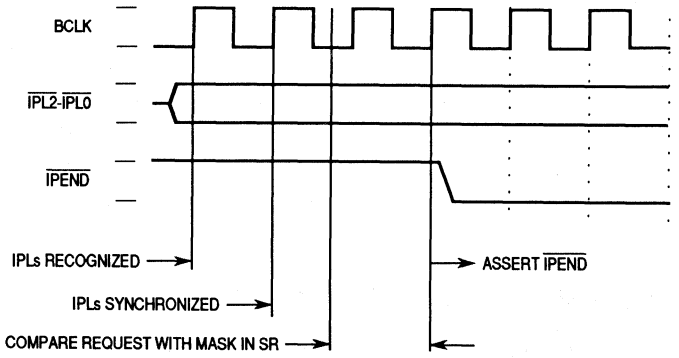


Figure 9-4. Assertion of $\overline{\text{IPEND}}$

When processing an interrupt exception, the processor first makes an internal copy of the SR, sets the mode to supervisor, suppresses tracing, and sets the processor interrupt mask level to the level of the interrupt being serviced. The processor attempts to obtain a vector number from the interrupting device using an interrupt acknowledge bus cycle with the interrupt level number output on the transfer modifier signals. For a device that cannot supply an interrupt vector, the autovector signal ($\overline{\text{AVEC}}$) can be asserted. The MC68040 uses an internally generated autovector, which is one of vector numbers 25–31, that corresponds to the interrupt level number. If external logic indicates a bus error during the interrupt acknowledge cycle, the interrupt is considered spurious, and the processor generates the spurious interrupt vector number, 24.

Once the vector number is obtained, the processor saves the exception vector offset, PC value, and the internal copy of the SR on the active supervisor stack. The saved value of the PC is the logical address of the instruction that would have been executed had the interrupt not occurred.

If the M bit of the SR is set, the processor clears the M bit and creates a throwaway exception stack frame on top of the interrupt stack as part of interrupt exception processing. This second frame contains the same PC value and vector offset as the frame created on top of the master stack, but has a format number of 1. The copy of the SR saved on the throwaway frame is exactly the same as that placed on the master stack except that the S bit is set in the version placed on the interrupt stack. (It may or may not be set in the copy saved on the master stack.) The resulting SR (after exception processing) has the S bit set and the M bit cleared.

The processor loads the address in the exception vector into the PC, and normal instruction execution resumes after the required prefetches for the interrupt handler routine.

Most M68000 Family peripherals use programmable interrupt vector numbers as part of the interrupt request/acknowledge mechanism of the system. If this vector number is not initialized after reset and the peripheral must acknowledge an interrupt request, the peripheral usually returns the vector number for the uninitialized interrupt vector, 15.

9.3.11 Breakpoint Instruction Exception

In order to use the MC68040 in a hardware emulator, it must provide a means of inserting breakpoints in the emulator code, and of performing appropriate operations at each breakpoint. For the MC68000 and MC68008, this can be done by inserting an illegal instruction at the breakpoint and detecting the illegal instruction exception from its vector location. However, since the VBR on the MC68010, MC68020, MC68030, and MC68040 allows arbitrary relocation of exception vectors, the exception address cannot reliably identify a breakpoint. The MC68020, MC68030, and MC68040 processors provide a breakpoint capability with a set of breakpoint instructions, $4848–$484F, for eight unique breakpoints.

When the MC68040 executes a breakpoint instruction, it performs a breakpoint acknowledge cycle (read cycle) with an acknowledge transfer type and transfer modifier value of $0. Refer to **SECTION 8 BUS OPERATION** for a description of the breakpoint acknowledge cycle. After external hardware terminates the bus cycle with either $\overline{TA}$ or $\overline{TEA}$, the processor performs illegal instruction exception processing.

9.4 EXCEPTION PRIORITIES

When several exceptions occur simultaneously, they are processed according to a fixed priority. Table 9-6 lists the exceptions, grouped by characteristics. Each group has a priority, from 0 through 7, with 0 as the highest priority.

Table 9-6. Exception Priority Groups

Group/ Priority	Exception and Relative Priority	Characteristics
0	Reset	Aborts all processing (instruction or exception) and does not save old context.
1	Data Access Error (ATC Fault or Bus Error)	Aborts current instructions — can have pendingtrace, FP post instruction, or unimplemented FP instruction exceptions.
2	Floating-Point Pre-Instruction	Exception processing begins before current floating-point instruction is executed. Instruction is restarted on return from exception.
3	BKPT #n, CHK, CHK2, Divide by Zero, FTRAPcc, RTE, TRAP #n, TRAPV	Exception processing is part of instruction execution.
	Illegal Instruction, Unimplemented Line A and Line F, Privilege Violation	Exception processing begins before instruction is executed.
	Unimplemented Floating-Point Instruction	Exception processing begins after memory operands are fetched and before instruction is executed.
4	Floating-Point Post-Instruction	Only reported for FMOVE to memory. Exception processing begins when FMOVE instruction and previous exception processing is completed.
5	Address Error	Reported after all previous instructions and associated exceptions complete.
6	Trace	Exception processing begins when current instruction or previous exception processing is completed.
7	Instruction Access Error (ATC Fault or Bus Error)	Reported after all previous instructions and associated exceptions complete.
8	Interrupt	Exception processing begins when current instruction or previous exception processing is completed.

The method used to process exceptions in the MC68040 is significantly different from that used in earlier members of the M68000 processor family, due to the restart exception model used. In general, when multiple exceptions are pending, the exception with the highest priority is processed first, and the remaining exceptions are regenerated when the current instruction is restarted. Note that the reset operation clears all other exceptions. Other exceptions to this are noted in the following paragraphs.

original exception condition. For example, if simultaneous interrupt and trap exceptions are pending, the exception processing for the trap exception occurs first, followed immediately by exception processing for the interrupt. When the processor resumes normal instruction execution, it is in the interrupt handler, which returns to the trap exception handler.

Exception processing for access error exceptions creates a type $7 stack frame that contains status information that can indicate a pending trace, floating-point post-instruction, or unimplemented floating-point instruction exception. The RTE instruction used to return from the access error exception handler checks the status bits for one of these pending exceptions. If one is indicated, the RTE changes the access error stack frame to match the pending exception and fetches the vector for the exception. Instruction execution then resumes in the new exception handler. If an access error, trace, and one of the two (mutually exclusive) floating-point exceptions occur simultaneously, the pending floating-point exception is indicated in the access error stack and the trace exception flag is undefined. The exception handler for the floating-point exception must check the trace bits on the stack and call the trace handler directly (after adjusting the stack frame to match the format for the trace exception).

Similarly, if a trace exception is pending at the same time a group 3 or floating-point post-instruction exception is pending, the trace exception is not reported and the exception handler for the other exception condition must check for the trace condition.

9.5 RETURN FROM EXCEPTIONS

After the processor has completed exception processing for all pending exceptions, the processor resumes normal instruction execution at the address in the vector for the last exception processed. Once the exception handler has completed execution, the processor must return to the system with the system context as it was prior to the exception (if possible). The RTE instruction returns from the handler to the previous system context for any exception.

When the processor executes an RTE instruction, it examines the stack frame on top of the active supervisor stack to determine if it is a valid frame and what type of context restoration it requires. This section describes the processing for each of the stack frame types; refer to **9.2 STACK FRAMES** for a description of the stack frame types.

For a normal four word frame (format $0), the processor updates the SR and PC with the data read from the stack, increments the stack pointer by eight, and resumes normal instruction execution.

For the throwaway four word stack (format $1), the processor reads the SR value from the frame, increments the active stack pointer by eight, updates the SR with the value read from the stack, and then begins RTE processing again, as shown in Figure 9-5. The processor reads a new format word from the stack frame on top of the active stack (which may or may not be the same stack used for the previous operation) and performs the proper operations corresponding to that format. In most cases, the throwaway frame is on the interrupt stack and when the SR value is read from the stack, the S and M bits are set. In that case, there is a normal four word frame on the master stack. However, the second frame may be any format (even another throwaway frame) and may reside on any of the three system stacks.

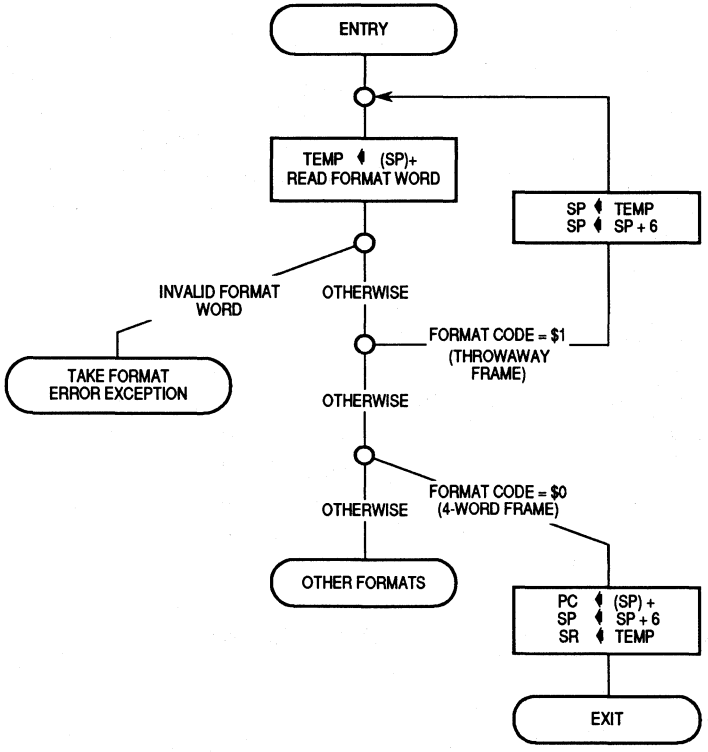


Figure 9-5. RTE Instruction for Throwaway Four-Word Frame

For the normal six word stack frame (format $2), the processor restores the SR and PC values from the stack, increments the active supervisor stack pointer by twelve, and resumes normal instruction execution.

For the floating-point post-instruction stack frame (format $3), the processor restores the SR and PC values from the stack, and increments the active supervisor stack pointer by twelve. If another pending floating-point post-instruction exception is pending, exception processing begins immediately for the new exception; otherwise, the processor resumes normal instruction execution.

For the access error stack frame (format $7), the processor restores the SR and PC values from the stack, and checks the four continuation status bits in the special status word (SSW) on the stack. If none of the bits are set, the processor increments the active supervisor stack pointer by 30, and resumes normal instruction execution. If the MOVEM continuation bit is set, the processor restores the calculated effective address (EA) from the stack frame, increments the active supervisor stack pointer by 30, and restarts the MOVEM instruction at a point after the EA calculation. All operand accesses for the MOVEM that occurred before the faulted access are repeated. If a continuation bit is set for a pending trace, unimplemented floating-point instruction, or floating-point post-instruction exception, the processor restores the calculated EA from the stack frame, increments the active supervisor stack pointer by 30, and immediately begins exception processing for the pending exception. The processor sets only one of the continuation bits when the access error stack frame is created. If multiple bits are set by the access error exception handler, operation of the RTE instruction is undefined.

If the frame format field in the stack frame contains an illegal format code, a format exception occurs. If a format error or access fault exception occurs during the frame validation sequence of the RTE instruction, the processor creates a normal four word or an access fault stack frame below the frame that it was attempting to use. In this way, the faulty stack frame remains intact. The exception handler can examine or repair the faulty frame. In a multiprocessor system, the faulty frame can be left to be used by another processor of a different type when appropriate.

9.6 ACCESS FAULT RECOVERY

Processor accesses of either data items or the instruction stream can result in bus errors. Bus error exceptions must be corrected to complete execution of the current context.

Push transfer bus errors are acted on when the execution unit is idle. The integer unit pipeline is frozen, the instruction cache and data cache requests are cancelled (however, writes are not lost), and pending writes are stacked.

Data ATC faults and bus errors are acted on when the bus controller and the execution unit are idle. A data access error freezes the pipeline and cancels any pending instruction cache accesses. Pending writes are stacked because the data cache will be deadlocked (because of the fault) until stacking transfers are initiated.

Instruction ATC faults and bus errors are acted on when the program counter section is deadlocked (the faulted data or another prefetch is required), the copy-back stage is empty, and the data cache and the bus controller are idle. Data ATC faults or bus error faults supersede the instruction ATC fault. Instruction access error faults are reset so that prefetch access faults can be ignored.

9.6.1 Access Error Stack Frame

A 30-word access error stack frame is created for data and instruction access faults other than instruction address errors. In addition to information about the current processor status and the faulted access, the stack frame also contains pending writebacks that must be completed by the bus error exception handler. The bus error stack frame is shown in Figure 9-6, followed by a description of the fields in the frame.

9.6.1.1 EFFECTIVE ADDRESS. The EA contains address information when one of the continuation flags CM, CT, CU or CP in the SSW is set.

9.6.1.2 SPECIAL STATUS WORD. The SSW is one of several registers saved as part of the access error stack frame. The SSW information indicates whether the fault was caused by an access to the instruction stream, data stream, or both and contains status information for the faulted access.

15	14	13	12	11	10	9	8	7	6 5	4 3	2 1 0
CP	CU	CT	CM	MA	ATC	LK	RW	X	SIZE	TT	TM

The first five fields listed below correspond to the TMn, TTn, SIZn, R/$\overline{W}$, and $\overline{LOCK}$ signals for the faulted access.

		OFFSET
STATUS REGISTER (SR)		$0
PROGRAM COUNTER (PC)		$2
0111	VECTOR OFFSET	$6
EFFECTIVE ADDRESS (EA)		$8
SPECIAL STATUS WORD (SSW)		$C
$00	WRITEBACK 3 STATUS (WB3S)	$E
$00	WRITEBACK 2 STATUS (WB2S)	$10
$00	WRITEBACK 1 STATUS (WB1S)	$12
FAULT ADDRESS (FA)		$14
WRITEBACK 3 ADDRESS (WB3A)		$18
WRITEBACK 3 DATA (WB3D)		$1C
WRITEBACK 2 ADDRESS (WB2A)		$20
WRITEBACK 2 DATA (WB2D)		$24
WRITEBACK 1 ADDRESS (WB1A)		$28
WRITEBACK 1 DATA/PUSH DATA LW0 (WB1D/PD0)		$2C
PUSH DATA LW1 (PD1)		$30
PUSH DATA LW2 (PD2)		$34
PUSH DATA LW3 (PC3)		$38

Figure 9-6. Access Error Stack Frame

TM — Transfer Modifier

TT — Transfer Type

SIZE — Transfer Size

The SIZE field corresponds to the original access size. If a data cache line read results from a read miss, and the line read is bus-errored, the SIZE field in the resulting stack frame indicates the size of the original read generated by the execution unit.

RW — Read/Write

LK — Locked Transfer

ATC — ATC Fault

This biit is set for an ATC fault due to a non-resident entry (bus error during tablewalk or invalid descriptor encountered) or privilege violation (write protected or supervisor-only). Cleared for a bus-errored instruction, data, or cache line push access.

MA — Misaligned Access
Set if an ATC fault occurs for the second page for an access which spans two pages in memory.

CM — Continuation — MOVEM Instruction Execution Pending
Set if a data access is bus errored for a MOVEM. Since the memory location or registers used to calculate the EA may get written over by the MOVEM operation, the MC68040 internally saves the EA after calculation. When MOVEM is bus errored, a stack frame is created with CM set, and the EA field contains the calculated EA for the instruction. When the RTE is executed, the MOVEM will restart using the EA on the stack (instead of repeating the EA calculate operation), if the address mode is PC relative (mode = 111, register = 010,011) or indirect with index (mode = 110).

CT — Continuation — Trace Exception Pending
CT is set for an access error with a pending trace exception. All pending accesses are allowed to complete after a trace condition is recognized — if any of these accesses fault, the resulting stack frame has the CT bit set and the EA field contains the address of the instruction being traced. When an RTE is executed with CT set, the MC68040 will move the words on the stack at offset $00–$0b from the current SP to offset $30–$3b, adjust the stack pointer by +$30, and change the stack frame format type to $2 before fetching the trace exception vector and jumping directly to trace exception handling. This stack adjustment creates the stack frame which normally would have been created for the trace exception if the pending access had not been bus errored.

CU — Continuation — Unimplemented Floating-Point -Instruction Exception Pending
CU is set for an access error with a pending exception for an unimplemented floating point instruction. Operation is the same as for the CT flag except the RTE fetches the Fline exception vector. The EA field contains the calculated EA determined by the EA field of the unimplemented instruction.

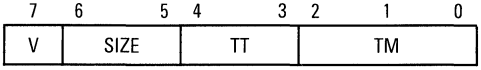
For the case where an unimplemented floating point instruction is traced, the unimplemented exception takes precedence, CU is set, and CT is undefined. The kernal must check for a trace condition using the stacked status register. If true, create required stack frame then jump directly to trace handler.

CP — Continuation — Floating-Point Post Exception Pending
CP is set for an access error with a floating point post exception pending. Operation is the same as for the CT flag except the RTE fetches the appropriate floating point post exception vector. For the case where a post

exception occurs during tracing, the post exception takes precedence, CP is set, and CT is undefined. The kernal must check for a trace condition using the stacked status register. The EA field contains the calculated EA determined by the EA field of the floating point instruction that caused the post-instruction exception.

X — Undefined

9.6.1.3 WRITEBACK STATUS. These 8-bit fields contain status information for the three possible writebacks which could be pending after the faulted access. For a data cache line push fault or a MOVE16 write fault, WB1S is zero (invalid).



TM — Transfer Modifier

TT — Transfer Type

SIZE — Transfer Size

V — Valid Write (writeback pending if set)

9.6.1.4 FAULT ADDRESS. The fault address (FA) is the initial address for the access which faulted. The fault address is a physical address only for cache pushes; FA is a logical address for all other cases. For a misaligned access which faults, the FA field contains the address of the first byte of the transfer, regardless of which of the two or three bus transfers for the misaligned access was faulted. For a push fault, the WB1A and FA addresses are the same.

9.6.1.5 WRITEBACK DATA. The writeback data in WB3D and WB2D are register-aligned with byte and word data contained in the least significant byte and word, respectively, of the field. Writeback data in WB1D is memory- aligned, and resides in the byte positions corresponding to the data bus lanes used in writing each byte to memory.

Table 9-7 show this explicitly for each combination of size and A1/A0.

Table 9-7. Writeback Data Alignment

Data Size	Address		Data Alignment	
	A1	A0	WB1D	WB2D,WB3D
Byte	0	0	31:24	7:0
	0	1	23:16	7:0
	1	0	15:8	7:0
	1	1	7:0	7:0
Word	0	0	31:16	15:0
	0	1	23:8	15:0
	1	0	15:0	15:0
	1	1	7:0, 31:24	15:0
Long	0	0	31:0	31:0
	0	1	23:0, 31:24	31:0
	1	0	15:0, 31:16	31:0
	1	1	7:0, 31:8	31:0

NOTE:
For a line transfer fault, the four long words of data i PD3–PD0 are already aligned with memory. Bits 31:0 of each field correspond to bits 31:0 of the memory location to be written to, regardless of the value of the address bits A1 and A0 for the writeback address.

9.6.2 Instruction ATC Faults and Bus Errors

The bus error exception handler can identify bus error exceptions due to instruction faults by examining the TM field in the SSW of the access error stack frame. For user and supervisor instruction faults the TM field contains 2 and 6, respectively. Since the processor allows all pending accesses to complete before reporting an instruction fault, the stack frame for an instruction fault will not contain any pending writebacks. The ATC bit of the SSW is used to distinguish between ATC faults and physical bus errors, and the FA field contains the logical address of the instruction prefetch. For ATC faults the handler can execute a PTEST instruction (using the FA and TM field from the SSW) to determine the specific cause of the address translation failure. After the handler corrects the cause of the fault, it executes an RTE instruction to restart execution of the instruction that contained the faulted prefetch.

9.6.3 Address Errors

For an address error fault, the processor saves a type 2 exception stack frame on the stack. This stack frame contains the PC pointing to the instruction that caused the address error, and the actual address referenced by the instruction. Note that bit zero of the referenced address is cleared on the stack frame. Address error faults must be repaired in software.

9.6.4 Data ATC Faults and Bus Errors

For a fault due to a data ATC fault or bus error, pending write-backs are also saved on the access error stack frame, and must be completed by the exception handler. For the faulted access, the fault address in the FA field combined with the transfer attribute information from the SSW can be used to identify the cause of the fault,. In identifying the fault, the system programmer should be aware that the read portion of locked transfers (for TAS, CAS, CAS2 and some translation table updates) is considered a write by the data MMU. This prevents both read and write accesses from occuring unless all pages touched by the instruction or table update are write enabled.

All accesses other than instruction prefetches go through the data memory unit, and the MC68040 treats the instruction and data address spaces as a single merged address space (the exception is the presence of separate transparent translation registers). The "function codes" for accesses such as PC-relative operand addressing and MOVES transfers to function codes 2 and 6 (user and supervisor instruction spaces in the MC68000) are converted to data references to go through the data memory unit, and appear in the TM field of the access error stack frame as data references.

After the fault is corrected, any pending writebacks on the stack frame must be completed. The writeback status fields should be checked for possible writebacks, which should be completed by the handler in the following order writeback 1, writeback 2, and then writeback 3. For a push fault, the push must be completed first, followed by two potential write-backs. Pending write-backs can occur in any combination of the three writeback registers (i.e. WB1S and WB2S may be invalid and WB3S valid). Completion of writeback 1 should not generate another access error, since this writeback corresponds to the faulted access that has been corrected by the handler. However, writebacks 2 and 3 can cause another bus error exception when the handler attempts to write to memory, and should be checked before attempting the write to prevent nesting of exceptions if required by the operating system. Some general bus fault examples follow which indicate the resulting contents of the access error stack frame fields:

1) Normal data access error (SSW — TT = $0, TM = $1 or $5). FA contains the logical address of the fault. For a write fault the addresses in FA and WB1A are the same, and WB1S and WB2S indicate up to two additional writebacks. For a read fault WB1S is zero, and WB2S and WB3S indicate up to two additional pending writebacks.
2) Data cache push fault (SSW — TT = $0, TM = $0, and RW = 0). WB1S is zero and the physical push address is contained in the Fault Address field. All four long-words of data for a line push are contained in push

data LW0–LW3, or a single long word is contained in LW0 for a long word push. Two write-backs may also be pending as indicated by the writeback 2 and 3 registers. Note that memory is now incoherent since the push buffer is invalidated after the fault — the only valid copy of the cache line now resides on the stack and cannot be snooped.

3) MOVE16 access error(SSW — TT=$1). WB1S is zero and the logical destination address is contained in the Fault Address field. For a faulted write, all four long-words of data for the line are contained in push data LW0–LW3. Two write-backs may also be pending for either a read or write fault.

9.6.5 Returning from Access Errors

After the bus error exception handler completes all pending operations and executes an RTE to return, the RTE reads only the stack information from offset $0–$d in the access error stack frame. For a pending trace exception, unimplemented floating-point instruction exception, or floating-point post exception, the RTE adjusts the stack to match the pending exception and immediately begins exception processing, without requiring the exception to re-occur.

9.7 FLOATING-POINT STATE FRAMES

An FSAVE instruction is executed to save the current floating-point internal state for context switches and floating-point exception handling. When an FSAVE is executed, the processor waits until the FPU either completes execution of all current instructions, or is unable to perform any further processing due to a pending exception that must be serviced. Any exceptions generated during this time are not reported, and are saved in the resulting busy state frame. Four state frames can be generated as a result of an FSAVE instruction: null, idle, busy, and unimplemented floating-point instruction.

A null state frame is saved if no floating-point instructions have been executed since the last hardware reset or FRESTORE of a null state frame. When an FRESTORE of a null state framed is performaed, all FPU operations are aborted, and the FPU enters the reset state. See Figure 9-7.

An idle state frame is saved if no exceptions are pending, and at least one instruction has been executed since the last hardware reset or FRESTORE of a null state frame. See Figure 9-7.

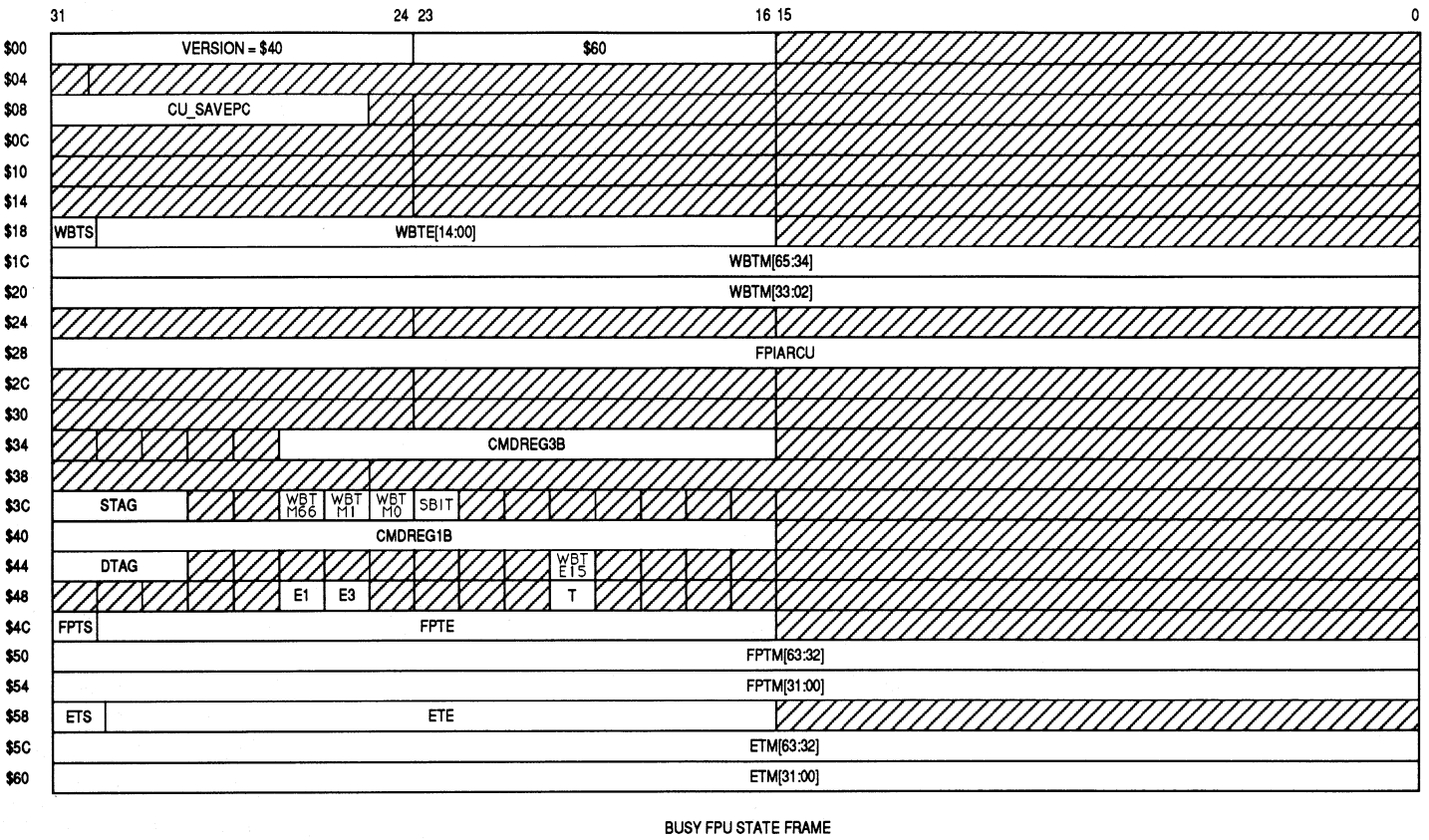


Figure 9-7. Floating-Point State Frames (Sheet 1 of 2)

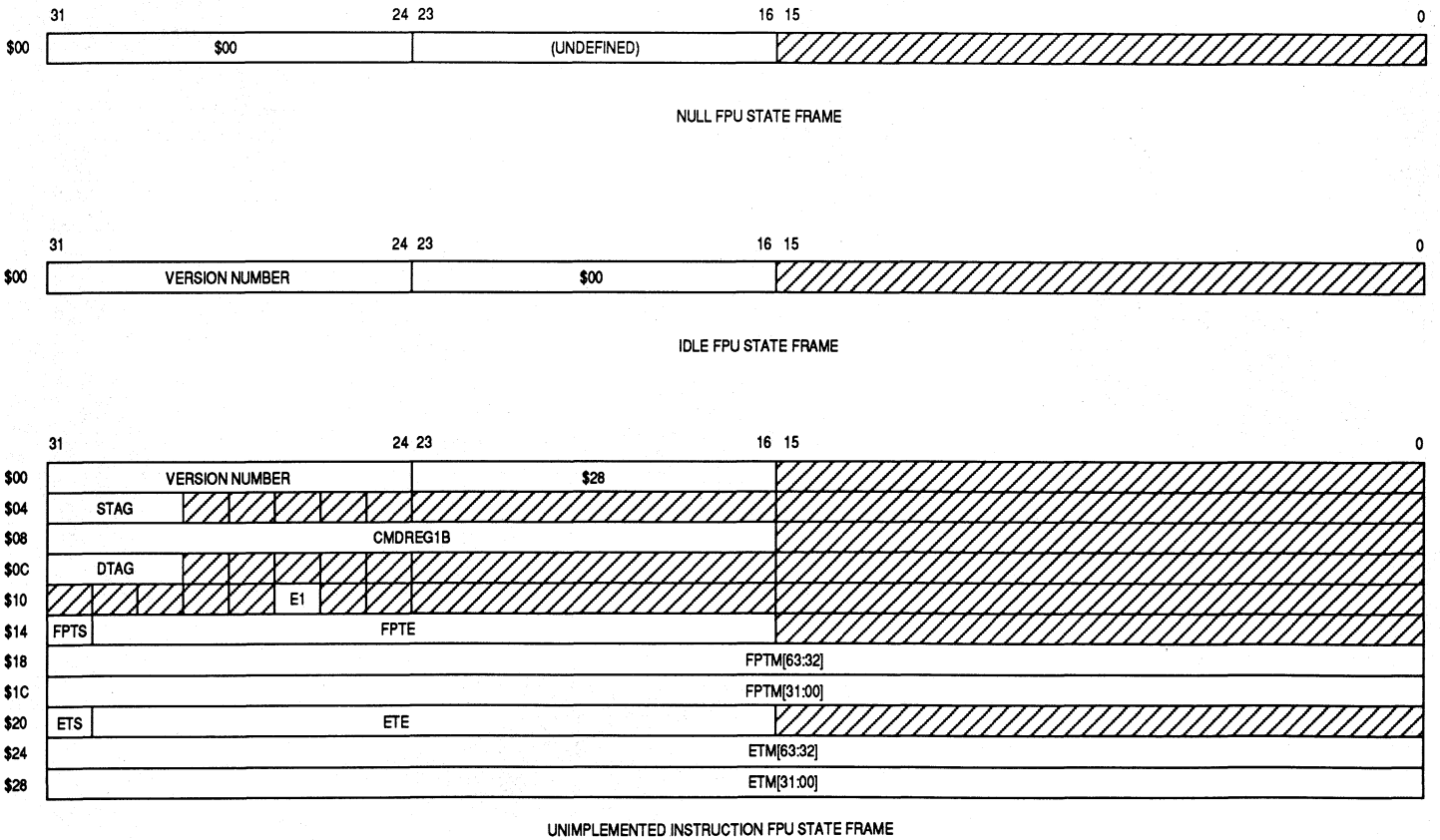


Figure 9-7. Floating-Point State Frames (Sheet 2 of 2)

A 50-word busy state frame is generated if any floating-point exceptions other than an unimplemented floating-point instruction exception are pending. See Figure 9-7.

A 22-word unimplemented floating-point instruction state frame is saved if the last instruction was an unimplemented floating-point instruction. See Figure 9-7.

For the busy and unimplemented instruction state frames, the following fields are defined for use by the exception handler:

CMDREG1B — This field contains the command word of the exceptional floating-point instruction for an E1 exception. For FSQRT, bits 6:0 are mapped from $04 for the instruction to $05 in CMDREG1B. All other instructions map directly.

CMDREG3B — Contains the encoded instruction command word for an E3 exception. The bit mapping between CMDREG1B and CMDREG3B is detailed below in Figure 9-8. For FSQRT, bits 6:0 are changed from $4 for the instruction to $5 for CMDREG1B, and therefore map to $21 for CMDREG3B.

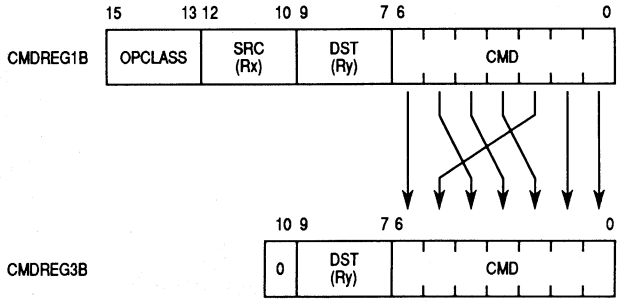


Figure 9-8. Mapping of Command Bits for CMDREG3B Field

CU_SAVEPC — This field contains the micro-PC for the conversion unit (CU).

E1 — If set, indicates an exception detected by the conversion unit (CU) pipeline stage. All exception types are possible. Check E3 (NU) first; if set, do E3 processing then RTE. If E1 is set it is handled later. For the unimplemented instruction state frame, if E1 is set then one or both of the operands are an unsupported data type.

E3 — If set, indicates an exception detected by the normalization unit (NU) pipeline stage. Only OVFL, UNFL, and INEX2 exceptions on opclass 0x0 (register-to-register and memory-to-register) for FADD, FSUB, FMUL, FDIV, FSQRT can take place. The exception handler must check for and handle an E3 exception first.

ETS, ETE, ETM — Collectively, these fields are refered to as the ETEMP register, and normally contain the source operand converted to extended precision (Sign, Exponent, Mantissa). For a packed decimal real source, bits [63:0] of the operand reside in ETM [63:00], and the ETS and ETE fields are undefined.

FPIARCU — Instruction address register for the conversion unit (CU).

FPTS, FPTE, FPTM — Collectively, these fields are refered to as the FPTEMP register, and normally contain the destination operand for diadic operations, converted to extended precision (Sign, Exponent, Mantissa). If the instruction specifies a packed decimal real source, bits [95:64] of the operand reside in FPTM [31:00], and the FPTS, FPTE, and FPTM [63:32] fields are undefined.

STAG, DTAG — These 3-bit fields specify the data type of the source and destination operands, respectively. STAG is undefined for a packed decimal real source operand. The encodings for STAG and DTAG are:

000 Normalized
001 Zero
010 Infinity
011 NAN
100 Extended precision denormalized or unnormalized input
101 Single or double precision denormalized input

T — If set, indicates a post-instruction exception occured.

WBTS, WBTE[15,14:00], WBTM[66,65:02,01,00], SBIT — Contain the exceptional operand in internal data format for E3 exceptions.

9.8 FLOATING-POINT EXCEPTIONS

There are eight "user" floating-point exceptions, of which seven can be generated by the MC68040. In order of priority, these exceptions are:

- Branch/Set on Unordered (BSUN)
- Signaling Not-a-Number (SNAN)

- Operand Error (OPERR)
- Overflow (OVFL)
- Underflow (UNFL)
- Divide by Zero (DZ)
- Inexact 2 (INEX2)

Each exception can be user disabled by clearing the corresponding bit in the enable byte of the FPCR. However, SNAN, OPERR, OVFL, and UNFL are non-maskable in some situations, and can cause a trap even if disabled by the user. This allows the supervisor exception handler to correct a default result generated by the MC68040 which is different from the result generated by an MC68881/MC68882 executing the same code. After correcting the result, the handler calls the user defined exception handler if the exception has been enabled in the floating-point control register (FPCR), or returns to the main program flow if the exception is disabled.

INEX1 (inexact result 1) is the condition that exists when a packed decimal operand cannot be converted exactly to extended precision in the current rounding mode. Since packed decimal real operands are not directly supported by the MC68040, INEX1 is never set by the processor, but is provided as a latch so that emulation software can report this exception.

All exception handlers (except format error) must have FSAVE as the first floating point instruction. All other floating-point instructions cause another exception to be reported. The trap handler should use only the FMOVEM instruction to read or write the floating-point data registers since FMOVEM cannot generate further exceptions or change the condition codes.

9.8.1 Unimplemented Floating-Point Instructions

Floating-point instructions that are supported by the MC68881 and MC68882 floating-point coprocessors, but are not directly supported by the MC68040 in hardware, are defined as unimplemented floating-point instructions. These instructions trap as an F-line exception and must be emulated in software by the F-line exception handler to maintain user object code compatibility.

The following MC68881/68882 instructions cause an unimplemented instruction exception when execution is attempted by the MC68040:

Monadic operations:

FACOS	FETOX	FLOG10	FSINH
FASIN	FETOXM1	FLOG2	FTAN
FATAN	FGETEXP	FLOGN	FTANH
FATANH	FGETMAN	FLOGNP1	FTENTOX
FCOS	FINT	FSIN	FTWOTOX
FCOSH	FINTRZ	FSINCOS	

Dyadic operations:

FMOD	FSGLDIV
FREM	FSGLMUL
FSCALE	

Miscellaneous operations:

FMOVECR

The MC68040 assists the emulation process by distinguishing unimplemented floating-point instructions from other unimplemented line-F instructions, and fetching any required source operands before taking the F-line exception. The memory operand (if required), floating-point instruction, and instruction address are passed to the FPU before taking the F-line exception, and the calculated EA is saved in the type $2 stack frame generated during exception processing for the unimplemented floating-point instruction. This simplifies and speeds up the emulation process by eliminating the need for the emulation routine to determine the EA, and providing all information required to emulate the instruction in either the exception stack frame or FSAVE state frame in the supervisor address space. The supervisor exception handler does not need to access the user address space, since none of the unimplemented floating-point instructions specify a memory or data register destination. (Implementations that choose to place the emulation software in the user address space may find it more efficient to pass the evaluated EA to user space than to pass the entire FSAVE state frame.)

In more detail, the followings processing steps occur for an unimplemented floating-point instruction:

1) When an unimplemented floating-point instruction is encountered, the processor waits for all previous floating-point instructions to complete execution. Any pre- or post-instruction exceptions which result are taken immediately, and the processor restarts the unimplemented floating-point instruction after returning from the exception handler.

2) Next, the instruction is partially decoded to allow fetching of the memory source operand, if required. When the operand fetch begins, all other read accesses for previous instructions are complete, and only the execution and writeback of results for previous integer instructions remains to be completed. If an access error occurs in fetching the operand (or in completing any other access before beginning the operand fetch), the unimplemented instruction is restarted after the processor returns from exception handling for the error.

3) The fetched source operand is passed to the FPU, which converts the operand to extended precision and saves the intermediate result. If the operand is an unsupported data type (denormalized, unnormalized, or packed decimal real), the unimplemented floating-point exception takes precedence, and the unsupported data type must be detected by the floating-point instruction emulation routine.

4) After the operand is fetched, the processor waits for all previous integer instructions, writebacks, and associated exception processing to complete before beginning exception processing for the unimplemented floating-point instruction. Any access error which occurs in completing the writebacks causes an access error exception, and the resulting stack frame indicates a pending unimplemented floating-point instruction exception. The writebacks are then completed in software by the access error exception handler, and exception processing for the unimplemented floating-point instruction exception begins immediately after return from the access error handler.

5) The processor begins exception processing for the unimplemented floating-point instruction by making an internal copy of the current SR. The processor then enters the supervisor mode , and clears the trace bits (T1, T0). The processor creates a type $2 stack frame, and saves the vector offset, PC, internal copy of the SR, and the calculated EA in the stack frame. The saved PC value is the logical address of the instruction that follows the unimplemented floating-point instruction. The processor generates exception vector number 11, for the unimplemented line-F instruction exception vector, fetches the address of the F-line exception handler from the exception vector table, and begins execution of the handler after prefetching instructions to fill the piepline.

The F-line exception handler can check for the format $2 stack frame type to distinguish an unimplemented floating point instruction from other F-line unimplemented instructions, which generate type $0 stack frames. When the exception handler for unimplemented floating point instructions executes an FSAVE, a 22-word unimnplemented instruction state frame is created (see Figure 9-7).

Note that unless the instruction specifies a packed decimal real sorce, the state frame contains both operands (if required). For packed decimal real, the handler can find the second operand in the designated destination register.

Additional information on floating-point instruction emulation can be found in the MC68040DH/AD, *MC68040 Designer's Handbook*.

9.8.2 Unimplemented Floating-Point Data Types

An unimplemented data type exception occurs when either operand to an implemented floating-point instruction is denormalized (for S, D, or X operands) or unnormalized (for X operands), or the source or destination data format is packed decimal real (P). These data types are unimplemented in the MC68040, and must be supported in software.

Unimplemented data types that are detected as operands for opclass 0x0 (register-to-register or memory-to-register) instructions cause a pre- instruction exception which is posted when the next floating point instruction is attempted. When an unimplemented data type is detected for opclass 011 (register-to-memory) instructions, a post-instruction exception is generated immediately. A type $0 (for the pre-instruction exception) or type $3 (for the post-instruction exception) stack frame is saved, and vector 55 is fetched.

A denormalized value generated as the result of a floating-point operation generates a non-maskable underflow exception instead of an unimplemented data type exception. Refer to **9.8.7 Underflow** for further information.

State Frame Information: For unimplemented data type exceptions resulting from the execution of opclass 0x0 (register-to-register or memory-to-register) instructions, the following FSAVE state frame fields are defined for use by the supervisor exception handler.

A denormalized or unnormalized extended precision source or destination operand is copied directly without modification to ETEMP or FPTEMP, respectively. If a packed decimal real source operand is specified, the upper 32 bits of the operand are copied to FPTEMP, and the lower 64 bits are copied to ETEMP. The destination operand in this case remains in the destination floating-point register, and may itself be either denormalized or unnormalized.

Denormalized single and double precision operands are stored in ETEMP as shown in Figure 9-9 and 9-10, respectively.

FSAVE State Frame Field	Contents
CMDREG1B	Exceptional instruction command word
ETEMP	Source operand converted to extended precision. If format is P, ETM [63:0] contains bits 63:0 of the packed decimal operand.
STAG	Source operand tag (undefined if format is P)
FPTEMP	Destination operand (if any) converted to extended precision. If format is P, FPTM [31:0] contains bits 95:64 of the packed decimal operand.
DTAG	Destination operand tag (if any)
E1	Set = CU exception
T-Flag	0 (pre-instruction exception)

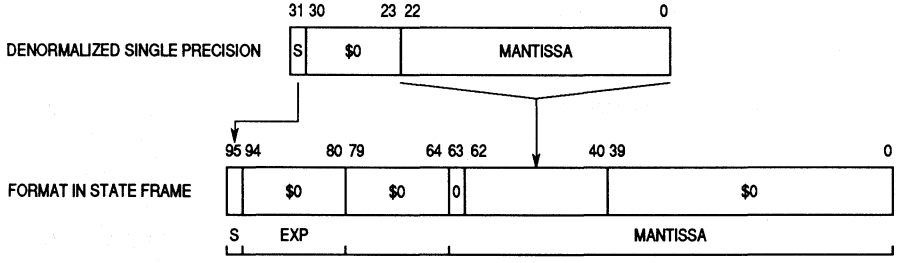


Figure 9-9. Format of Denormalized Single Precision Source Operand in State Frame

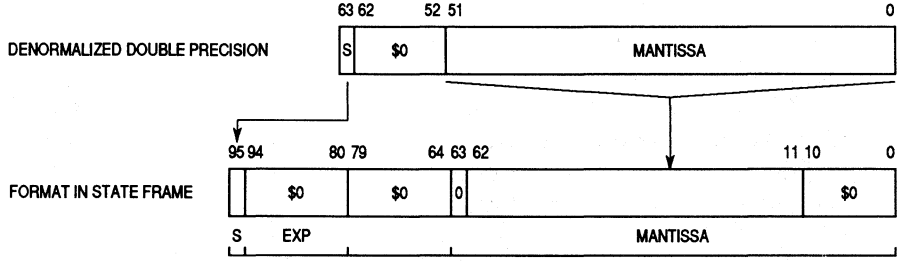


Figure 9.10. Format of Denormalized Double Precision Source Operand in State Frame

Single and double precision denormalized operands, and packed decimal real operands can be converted to normalized extended precision by the exception handler and restored into the state frame. (Unnormalized extended precision operands which can be normalized can also be restored.) If the operands for the instruction can be converted to normalized extended precision operands, the exception state frame can be restored into the processor for completion of the instruction, without the need for the handler to emulate the arithmetic operation itself. The exception handler must perform the following steps:

1) For a P format operand, emulate the conversion to extended precision and write the result into ETEMP. The destination operand (if required) is copied from the specified floating-point data register into FPTEMP.

2) Normalize the operands in ETEMP and FTEMP — if an operand can not be normalized, continuation of the instruction is not possible. For normalized operands, the corresponding STAG or DTAG field should be cleared, indicating a normalized operand.

3) Clear the E1 flag to clear the CU exception.

4) Write the CU_SAVEPC with $xxx to force the completion of the operand conversion in the CU execution unit. (Actual value To Be Defined.)

5) Execute an FRESTORE instruction to reload the state frame, and return to the main program flow. The CU pipeline stage of the FPU completes the conversion operation using the corrected operand values. If another E1 pre-instruction exception is generated at this time (such as overflow due to single or double precision rounding mode), that exception is taken immediately as a pre-instruction exception when the processor attempts to restart the last floating-point instruction.

For unimplemented data type exceptions resulting from execution of opclass 011 (register-to-memory) instructions, the following FSAVE state frame fields are defined for use by the supervisor exception handler.

FSAVE State Frame Field	Contents
CMDREG1B	Exceptional instruction command word
ETEMP	Source operand from floating-point data register, unrounded
STAG	Source operand tag
E1	Set = CU exception
T-Flag	1 (post-instruction exception)

If the source operand is unnormalized and can be converted to a normalized extended precision value, the floating-point instruction can be continued as noted above. Otherwise, the exception handler must perform the conversion and write the result to memory.

9.8.3 Branch/Set on Unordered (BSUN)

The BSUN exception is the result of performing a conditional test associated with the FBcc, FDBcc, FTRAPcc, and FScc instructions when an unordered condition is present. (An unordered condition occurs when an input to an arithmetic operation is a NAN.) The BSUN exception can only occur during floating-point conditional instructions with the following IEEE non-aware branch condition predicates:

GT	Greater Than	GL	Greater Than or Less Than
NGT	Not Greater Than	NGL	Not Greater Than or Less Than
GE	Greater Than or Equal	GLE	Greater Than or Less Than or Equal
NGE	Not Greater Than or Equal	NGLE	Not Greater Than or Equal Less Than or Equal
LT	Less Than	SF	Signaling False
NLT	Not Less Than	ST	Signaling True
LE	Less Than or Equal	SEQ	Signaling Equal
NLE	Not Less Than or Equal	SNE	Signaling Not Equal

If a floating-point exception is pending, a pre-instruction exception is taken. After the appropriate exception handler is executed, the conditional instruction is restarted. When the FPU pipeline is idle (all previous floating point instructions have been completed) and no exceptions are pending, the processor evaluates the conditional predicate and checks for a BSUN exception before executing the conditional instruction. A BSUN exception occurs if the conditional predicate is one of the IEEE non-aware branches, and the NAN condition code bit is set. When the processor detects this exception, it sets the BSUN bit in the floating-point status register (FPSR) exception status byte.

Trap Disabled Results:

The floating point condition is evaluated as if it were the equivalent aware conditional predicate.

Trap Enabled Results:

The processor takes a floating-point pre-instruction exception. A four-word type $0 stack frame is saved, and vector number 48 is generated to access the BSUN exception vector. For MC68881/MC68882 compatibility, the supervisor handler must update the floating-point instruction address register (FPIAR) register by copying the PC value in the pre-instruction stack frame to the FPIAR.

The BSUN exception is unique in that the trap is taken before the conditional predicate is evaluated. If the exception handler does not set the PC to the instruction following the one that caused BSUN exception when returning, the exception is re-executed. Therefore, it is the responsibility of the trap handler to prevent the conditional instruction from taking the BSUN trap again. Four ways are available to prevent taking the trap again.

The first way involves incrementing the stored PC in the stack to bypass the conditional instruction. This technique applies to situations where a fall-through is desired. Be aware that accurate calculation of the PC increment requires detailed knowledge of the size of the conditional instruction being bypassed.

The second method is to clear the NAN bit of the FPSR condition code byte. However, this alone cannot deterministically control the result indication (true or false) which would be returned when the conditional instruction re-executes.

The third method is to disable the BSUN trap. Like the second method, this method cannot control the result indication (true or false) which would be returned when the conditional instruction re-executes.

The fourth method involves examining the condition predicate and setting the condition code in the FPSR accordingly. This technique gives the most control since it is possible to pre-determine the direction of program flow. Bit 7 of the F-line operation word indicates where the conditional predicate is located. If bit 7 is set, the conditional predicate is the lower six bits of the F-line operation word. Otherwise, the conditional predicate is the lower six bits of the instruction word, which immediately follows the F-line operation word. Using the conditional predicate and the table for non-aware test in **4.4.2 Conditional Test Definitions**, the condition codes can be set to return a known result indication when the conditional instruction is re-executed.

9.8.4 Signaling Not-a-Number (SNAN)

An SNAN is used as an escape mechanism for a user defined, non-IEEE data type. The processor never creates an SNAN as a result of an operation; a NAN created by an operand error exception is always a non-signaling NAN.

When an SNAN is an operand involved in an arithmetic instruction, the SNAN bit is set in the FPSR exception byte. Since the FMOVEM, FMOVE FPcr, and FSAVE instructions do not modify the status bits, they cannot generate exceptions. Therefore, these instructions are useful for manipulating SNANs.

Trap Disabled Results:

If the destination data format is single (S), double (D), or extended (X) then the SNAN bit in the NAN is set to one and the resulting non-signaling NAN is transferred to the destination. No bits other than the SNAN bit of the NAN are modified, although the input NAN is truncated if necessary.

If the destination data format is byte (B), word (W), or long-word (L), then the data written to the destination is undefined, and an SNAN post-instruction exception is taken immediately. For MC68881/MC68882 compatibility, if the destination format is B,W, or L, the supervisor exception handler should store the most significant 8, 16, or 32 bits, respectively, of the SNAN mantissa, with the SNAN bit set, to the destination.

Trap Enabled Results:

For memory or integer data register destinations, the result is written in the same manner as if the trap were disabled, and then a post-instruction exception is taken immediately. For MC68881/MC68882 compatibility, if the destination format is B,W, or L, the supervisor exception handler should store the most significant 8, 16, or 32 bits, respectively, of the SNAN mantissa, with the SNAN bit set, to the destination. If desired, the user trap handler can overwrite the result.

For floating-point data register destinations, the floating-point data registers are not modified, and an SNAN pre-instruction exception is signaled. In this case, the SNAN trap handler should supply the result.

NOTE

The trap handler should use only the FMOVEM instruction to read or write the floating-point data registers, since FMOVEM cannot generate further exceptions. Also, only an FMOVEM instruction can write a SNAN into a floating-point data register.

State Frame Information:

For SNAN pre-instruction exceptions resulting from execution of opclass 0x0 (register-to-register or memory-to-register) instructions, the following FSAVE state frame fields are defined for use by the supervisor exception handler. A source or destination SNAN is stored in ETEMP or FPTEMP, respectively, with its SNAN bit set.

FSAVE State Frame Field	Contents
CMDREG1B	Exceptional instruction command word
ETEMP	Source operand converted to extended precision
STAG	Source operand tag
FPTEMP	Destination operand (if any) converted to extended precision
DTAG	Destination operand tag (if any)
T-Flag	0 (pre-instruction exception)

For SNAN post-instruction exceptions resulting from execution of an opclass 011 (register-to-memory) instruction, the following FSAVE state frame fields are defined for use by the supervisor exception handler:

FSAVE State Frame Field	Contents
CMDREG1B	FMOVE instruction command word
ETEMP	Source operand from FPn register, unrounded, with SNAN bit set
STAG	Source operand tag (indicates NAN)
T-Flag	1 (post-instruction exception)

9.8.5 Operand Error

The operand error category encompasses problems arising in a variety of operations, and includes those errors not frequent or important enough to merit a specific exception condition. Basically, an operand error occurs when an operation has no mathematical interpretation for the given operands. The possible operand errors are listed in Table 9-8. When an operand error occurs, the OPERR bit is set in the FPSR exception status byte.

Table 9-8. Possible Operand Errors

Instruction	Condition Causing Operand Error
FADD	(+infinity)+(−infinity) or (−infinity)+(+infinity)
FDIV	0/0 or infinity/infinity
FMOVE to B,W,or L	Integer Overflow, Source is Non-Signaling NAN, or Source is ±infinity
FMUL	One operand is 0, other operand is ±infinity
FSQRT	Source is <0, Source = −infinity
FSUB	(+infinity)−(+infinity) or (−infinity)−(−infinity)

Trap Disabled Results:

If the destination is a floating-point data register, an extended precision non-signaling NAN (with all ones mantissa) is stored in the destination floating-point data register.

For an operand error on an FMOVE to a B, W, or L memory or integer data register destination, the result stored is undefined, and a post-instruction exception is taken immediately. For MC68881/MC68882 compatibility, if the operand error is caused by an integer overflow or if the floating-point data register to be stored contains infinity, the supervisor exception handler should store the largest positive or negative integer that can fit in the specified destination format size. If the destination is integer (i.e. B, W, or L) and the floating-point number to be stored is a NAN, then the 8, 16, or 32 most significant bits of the NAN significand should be stored as the result.

The processor incorrectly reports an operand error for an FMOVE to memory or integer data register if the operand is equal to the largest negative integer representable in its format (-2^7 for B, -2^{15} for W, and -2^{31} for L). These are reported as operand errors even though no exception should be generated. The supervisor handler must detect these cases, store the proper result, clear the exception, and return to the main program flow.

Trap Enabled Results:

If the destination is a floating-point data register, the register is not modified, and a pre-instruction exception is reported. In this case, the trap handler should generate the appropriate result.

If an operand error occurs for an FMOVE FPn, ⟨EA⟩ instruction, then the results are the same as for trap disabled, and the result stored is undefined. The trap handler should store the appropriate result if required for MC68881/MC68882 compatibility.

State Frame Information:

For OPERR pre-instruction exceptions resulting from execution of opclass 0x0 (register-to-register or memory-to-register) instructions, the following FSAVE state frame fields are defined for use by the supervisor exception handler.

FSAVE State Frame Field	Contents
CMDREG1B	Exceptional instruction command word
ETEMP	Source operand converted to extended precision
STAG	Source operand tag
FPTEMP	Destination operand (if any) converted to extended precision
DTAG	Destination operand tag (if any)
T-Flag	0 (pre-instruction exception)

For an OPERR post-instruction exceptions resulting from execution of an FMOVE FPn, ⟨ea⟩ instruction, the following FSAVE state frame fields are defined for use by the supervisor exception handler. In addition, the FPIAR contains the address of the FMOVE instruction that caused the exception, and the EA field in the exception stack frame contains the destination address.

FSAVE State Frame Field	Contents
CMDREG1B	FMOVE instruction command word
ETEMP	Source operand from FPn register, unrounded
STAG	Source operand tag
WBTEMP	Contains the rounded integer, used to check for erroneous integer overflow
T-Flag	1 (post-instruction exception)

9.8.6 Overflow

An overflow occurs when the intermediate result of an arithmetic operation is too large to be represented in a floating-point data register using the selected rounding precision. A store to memory operation overflows when the value in the source floating-point data register is too large to be represented in the destination format.

Overflow is detected for arithmetic operations where the destination is a floating-point data register when the intermediate result exponent is greater than or equal to the maximum exponent value of the selected rounding precision. Overflow is detected for store to memory operations when the intermediate result exponent is greater than or equal to the maximum exponent value of the destination data format. Overflow can only occur when the destination is in the S, D, or X format. Overflows when converting to the B, W, or L integer and packed decimal formats are included as operand errors.

Refer to **3.6 DATA FORMAT DETAILS** for the maximum exponent value for each format. At the end of any operation that could potentially overflow, the intermediate result is checked for underflow, rounded, and checked for overflow before it is stored to the destination. If overflow occurs, the OVFL bit is set in the FPSR exception byte.

NOTE

An overflow can occur when the destination is a floating-point data register and the selected rounding precision is single or double even if the intermediate result is small enough to be represented as an extended precision number. The intermediate result is rounded to the selected precision (both the mantissa and the exponent), and then the rounded result is stored in extended precision format. If the magnitude of the intermediate result exceeds the range of the selected rounding precision format, an overflow occurs.

Trap Disabled Results:

If the destination is a floating-point data register, then the register is not affected, and either a pre-instruction or a post-instruction exception is reported, as described in later paragraphs. If the destination is a memory or integer data register destination, then an undefined result is stored, and a post-instruction exception is taken immediately. For MC68881/MC68882 compatibility, the supervisor exception handler should store a value determined by the rounding mode at the destination, as follows:

Rounding Mode	Result
RN	Infinity, with the sign of the intermediate result
RZ	Largest magnitude number, with the sign of the intermediate result
RM	For positive overflow, largest positive number For negative overflow, infinity
RP	For positive overflow, infinity For negative overflow, largest negative number

Trap Enabled Results:

Results are identical to the trap disabled case.

Stack Frame Information:

The following paragraphs outline specific cases and the information available to the supervisor exception handler to allow it to write the desired

value to the destination and determine the exceptional operand to pass to the user exception handler. For each case, the address of the instruction that causes the overflow is available to the trap handler in the FPIAR.

An overflow pre-instruction exception can occur for FMOVE to a floating-point register, FABS, and FNEG when the rounding mode is single precision and the source operand format is double or extended precision, or the rounding mode is double precision and the source operand format is extended. For MC68881/MC68882 compatibility, a bias of $6000 must be subtracted from the exponent of the intermediate result to create the exceptional operand that the user handler expects. The following information is available in the FSAVE state frame:

FSAVE State Frame Field	Contents
CMDREG1B	Exceptional instruction command word
ETEMP	Intermediate result with mantissa rounded to correct precision
STAG	Source operand tag = Normalized
E1	CU exception = Set
T-Flag	0 (pre-instruction exception)

An overflow exception can occur for FADD, FSUB, FMUL, and FDIV with any rounding precision, and for FSQRT with a single or double precision destiation. The exception is normally reported as a pre-instruction exception for the next floating-point instruction decoded by the integer unit; however, if a following FMOVE instruction is already in progress and generates a post-instruction exception, the overflow exception takes precedence and is reported as a post-instruction exception. This prevents out of order exception reporting. Note that the EA field for the post-instruction stack frame is undefined in this case. For MC68881/MC68882 compatibility, a bias of $6000 must be subtracted from the exponent of the intermediate result to create the exceptional operand that the user handler expects. The following information is available in the FSAVE state frame:

FSAVE State Frame Field	Contents
CMDREG3B	Exceptional instruction command word, encoded
WBTEMP	= WBTS, WBTE, and WBTM = Intermediate result with mantissa rounded to correct precision
WBTE15	Bit 15 of the intermediate result's 16-bit exponent = 0 for overflow
E3	NU exception = Set
T-Flag	0 (pre-instruction exception) or 1 (post-instruction exception)

An overflow pre-instruction exception can occur for FMOVE to memory when the destination format is S or D. The exception is reported as a post-instruction exception, with the evaluated destination EA in the stack frame. The following information is available in the FSAVE state frame:

FSAVE State Frame Field	Contents
CMDREG1B	FMOVE instruction command word
ETEMP	Intermediate result with mantissa rounded to correct precision
STAG	Source operand tag = Normalized
E1	CU exception = Set
T-Flag	1 (post-instruction exception)

By examining the instruction, the trap handler can determine the arithmetic operation type and destination location. The trap handler can execute an FSAVE instruction to obtain additional information. When an FSAVE is executed, the exceptional operand is stored in the state frame. Refer to **9.7 STATE FRAMES** for details of the FSAVE instruction state frames.

9.8.7 Underflow

An underflow occurs when the intermediate result of an arithmetic operation is too small to be represented as a normalized number in a floating-point data register using the selected rounding precision. A store to memory operation underflows when the value in the source floating-point data register is too small to be represented in the destination format as a normalized number. Underflow is detected for arithmetic operations where the destination is a floating-point data register when the intermediate result exponent is less than or equal to the minimum exponent value of the selected rounding precision.

Underflow is detected for store to memory operations when the intermediate result exponent is less than or equal to the minimum exponent value of the destination data format. Underflow is NOT detected for intermediate result exponents that are equal to the extended precision minimum exponent, since the explicit integer part bit of extended precision permits representation of normalized numbers with a minimum extended precision exponent.

Underflow can only occur when the destination format is S, D, or X. When the destination format is B, W, or L, the conversion underflows to zero without causing either an underflow or an operand error. See **3.2.3 Floating-Point Data Format Details** for the minimum exponent value for each format.

At the end of any operation that could potentially underflow, the intermediate result is checked for underflow, rounded, and checked for overflow before it is stored at the destination. If an underflow occurs, the UNFL bit is set in the FPSR exception status byte.

NOTE

An underflow can occur when the destination is a floating-point data register and the selected rounding precision is single or double even if the intermediate result is large enough to be represented as an extended precision number. The intermediate result is rounded to the selected precision (both the mantissa and the exponent), and then the rounded result is stored in extended precision format. If the magnitude of the intermediate result is too small to be represented in the selected rounding precision format, an underflow occurs.

Trap Disabled Results:

If the destination is a floating-point data register, then the register is not affected, and either a pre-instruction or a post-instruction exception is reported, as described in later paragraphs. If the destination is a memory or integer data register destination, then an undefined result is stored, and a post-instruction exception is taken immediately.

For MC68881/MC68882 compatibility, the supervisor exception handler should store the result in the destination as either a denormalized number or zero. Denormalization is accomplished by shifting the mantissa of the intermediate result to the right while incrementing the exponent until it is equal to the denormalized exponent value for the destination format. The denormalized intermediate result is rounded to the selected rounding precision or destination format.

If, in the process of denormalizing the intermediate result, all of the significant bits are shifted off to the right, the selected rounding mode determines the value to be stored at the destination, as follows:

Rounding Mode	Result
RN	Zero, with the sign of the intermediate result
RZ	Zero, with the sign of the intermediate result
RM	For positive underflow, +zero For negative underflow, smallest denormalized negative number
RP	For positive underflow, smallest denormalized positive number For negative underflow, −zero

Trap Enabled Results:

Results are identical to the trap disabled case.

State Frame Information:

The following paragraphs outline specific cases and the information available to the supervisor exception handler to allow it to write the desired value to the destination and determine the exceptional operand to pass to the user exception handler. For each case, the address of the instruction that causes the overflow is available to the trap handler in the FPIAR.

An underflow pre-instruction exception can occur for FMOVE to a floating-point register, FABS, and FNEG when the rounding mode is single precision and the source operand format is double or extended precision, or the rounding mode is double precision and the source operand format is extended. For MC68881/MC68882 compatibility, a bias of $6000 must be subtracted from the exponent of the intermediate result to create the exceptional operand that the user handler expects. The following information is available in the FSAVE state frame:

FSAVE State Frame Field	Contents
CMDREG1B	Exceptional instruction command word
FPTEMP	Intermediate result, extended precision, unrounded
STAG	Source operand tag = Normalized
E1	CU exception = Set
T-Flag	0 (pre-instruction exception)

An underflow exception can occur for FADD, FSUB, FMUL, FDIV, and FSQRT with any rounding precision. The exception is normally reported as a pre-instruction exception for the next floating-point instruction decoded by the integer unit; however, if following FMOVE instruction is already in progress and generates a post-instruction exception, the overflow exception takes precedence and is reported as a post-instruction exception. This prevents out of order exception reporting. Note that the EA field for the post-instruction stack frame is undefined in this case. For MC68881/MC68882 compatibility, a bias of $6000 must be subtracted from the exponent of the intermediate result to create the exceptional operand that the user handler expects. The following information is available in the FSAVE state frame:

FSAVE State Frame Field	Contents
CMDREG3B	Exceptional instruction command word, encoded
WBTEMP	= WBTS, WBTE, and WBTM = Intermediate result sign, biased 15-bit exponent, and 64-bit mantissa prior to rounding
WBTE15	Bit 15 of the intermediate result's 16-bit exponent = 1 for underflow
WBTM1, WBTM0, SBIT	Guard, round, and sticky of intermediate result's 67-bit mantissa
E3	NU exception = Set
T-Flag	0 (pre-instruction exception) or 1 (post-instruction exception)

An underflow pre-instruction exception can occur for FMOVE to memory when the destination format is S or D. The exception is reported as a post-instruction exception, with the evaluated destination EA in the stack frame. The following information is available in the FSAVE state frame:

FSAVE State Frame Field	Contents
CMDREG1B	FMOVE instruction command word
FPTEMP	Intermediate result with mantissa prior to rounding
STAG	Source operand tag = Normalized
E1	CU exception = Set
T-Flag	1 (post-instruction exception)

By examining the instruction, the trap handler can determine the arithmetic operation type and destination location. The trap handler can execute an FSAVE instruction to obtain additional information. When an FSAVE is executed, the exceptional operand is stored in the state frame. Refer to **9.7 STATE FRAMES** for details of the FSAVE instruction state frames. When an underflow occurs, the exceptional operand is defined differently for various destination types:

NOTE

The IEEE standard defines two causes of an underflow:

1. When a result is very small, the absolute value of the number is less than the minimum number that can be represented by a normalized number in a specific format
2. When loss of accuracy occurs while attempting to calculate a very small number (a loss of accuracy also causes an inexact exception)

The IEEE standard specifies that if the underflow trap is disabled, an underflow should only be signaled when both of these cases are satisfied (i.e., the result is too small to represent with a given format, and there is a loss of accuracy during the calculation of the final result). If the trap is enabled, the underflow should be signaled any time a tiny result is produced, regardless of whether accuracy is lost in calculating it.

The processor UNFL bit in the AEXC byte of the FPSR implements the IEEE trap disabled definition, since it is only set when a very small number is generated and accuracy has been lost when calculating that number. The UNFL bit in the EXC byte implements the IEEE trap enabled definition, since it is set anytime a tiny number is generated.

9.8.8 Divide by Zero

This exception occurs when a zero divisor occurs for an FDIV instruction. When a divide-by-zero is detected, the DZ bit is set in the FPSR exception status byte.

Trap Disabled Results:

An infinity with the sign set to the exclusive OR of the signs of the input operands is stored in the destination floating-point data register.

Trap Enabled Results:

The destination floating-point data register is not modified, and the exception is reported as a pre-instruction exception when the next floating-point instruction is attempted. The trap handler must generate a result to store in the destination.

To assist the trap handler in this function, the processor supplies the following iformation in the FSAVE state frame:

FSAVE State Frame Field	Contents
CMDREG1B	FDIV command word
ETEMP	Source operand converted to extended precision
STAG	Source operand tag
FPTEMP	Destination operand converted to extended precision
T-Flag	0 (Pre-Instruction exception)

9.8.9 Inexact Result

The processor provides two inexact bits (INEX1 and INEX2) to help distinguish between inexact results generated by emulated decimal input (INEX1) and other inexact results (INEX2). Two inexact bits are useful in instructions in which both types of inexacts can occur, such as:

```
FDIV.P   #7E-1,FP3
```

In this case, the packed decimal to extended precision conversion of the immediate source operand causes an inexact error to occur which is signaled as INEX1. Furthermore, the subsequent divide might also produce an inexact result and cause INEX2 to be set. Therefore, the processor provides two inexact bits in the FPSR exception status byte to distinguish these two cases.

Note that only one inexact exception vector number is generated by the processor. If either of the two inexact exceptions is enabled, the MPU fetches the inexact exception vector, and the exception handler routine is initiated. Refer to **9.8.10 Inexact Result on Decimal Input** for a discussion of INEX1.

In a general sense, INEX2 is the condition that exists when any operation, except the input of a packed decimal number, creates a floating-point intermediate result whose infinitely precise mantissa has too many significant bits to be represented exactly in the selected rounding precision or in the destination data format. If this condition occurs, the INEX2 bit is set in the FPSR exception status byte, and the infinitely precise result is rouned as described in the next paragraph.

The processor supports the four rounding modes specified by the IEEE standard. These modes are round to nearest (RN), round toward zero (RZ), round toward plus infinity (RP), and round toward minus infinity (RM). The rounding definitions are:

Rounding Mode	Result
RN	The representable value nearest to the infinitely precise intermediate value is the result. If the two nearest representable values are equally near (a tie), then the one with the least significant bit equal to zero (even) is the result. This is sometimes referred to as "round nearest, even."

Rounding Mode	Result
RZ	The result is the value closest to, and no greater in magnitude than, the infinitely precise intermediate result. This is sometimes referred to as the "chop mode," since the effect is to clear the bits to the right of the rounding point.
RM	The result is the value closest to and no greater than the infinitely precise intermediate result (possibly minus infinity).
RP	The result is the value closest to and no less than the infinitely precise intermediate result (possibly plus infinity).

The RM and RP rounding modes are often referred to as "directed rounding modes" and are useful in interval arithmetic. Rounding is accomplished usig the intermediate result format shown in Figure 9-12.

Depending on the selected rounding precision or destination data format in effect, the location of the least significant bit of the fraction and the locations of the guard, round, and sticky bits in the 67-bit intermediate result mantissa varies.

The guard and round bits are always calculated exactly. The sticky bit is used to create the illusion of an infinitely wide intermediate result mantissa. As shown by the arrow in Figure 9-11, the sticky bit is the logical OR of all the bits in the infinitely precise result to the right of the round bit. During the calculation stage of an arithmetic operation, any non-zero bits generated that are to the right of the round bit set the sticky bit (which is used in rounding) to one. Because of the sticky bit, the rounded intermediate result for all required IEEE arithmetic operations in the round-to-nearest mode is in error by no more than one half unit in the last place.

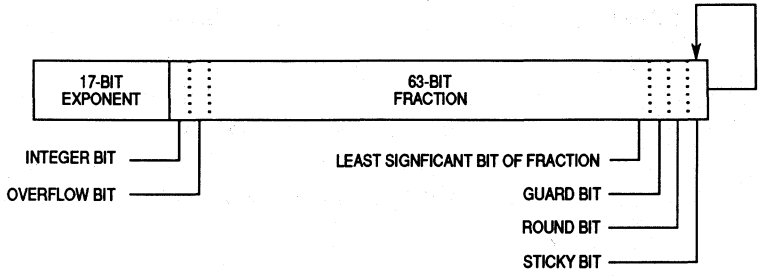


Figure 9-11. Intermediate Results Format

NOTE

When the FPU is programmed to operate in the single or double precision rounding mode, a method referred to as "range control" is used to assure correct emulation of a machine that only supports single or double precision arithmetic. When the processor performs any calculation, the intermediate result is in the format shown in Figure 6-2, and a rounded rest stored into a floating-point data register is always in the extended precision format. However, if the single or double precision rounding mode is in effect, the final result generated by the processor is within the range of the format.

Range control is accomplished by not only rounding the intermediate result mantissa to the specified precision, but also checking the 17-bit intermediate exponent to ensure that it is within the representable range of the selected rounding precision format. If the intermediate exponent exceeds the range of the selected precision, the exponent value appropriate for an underflow or overflow is stored as the result in the 16-bit extended precision format exponent. For example, if the rounding precision and mode is single/RM and the result of an arithmetic operation overflows the magnitude of the single precision format, the largest normalized single precision value is stored as an extended precision number in the destination floating-point data register (i.e., an unbiased 15-bit exponent of $00FF and a mantissa of $FFFFFF0000000000). If an infinity is the appropriate result for an underflow or overflow, the infinity value for the destination data type is stored as the result (i.e., an exponent with the maximum value and a mantissa of zero).

Figure 9-12 shows the algorithm that is used to round an intermediate result to the selected rounding precision or destination data format. If the destination is a floating-point register, the rounding boundary is determined by either the selected rounding precision in the FPCR or the precision specified by the instruction (for example, FSADD and FDADD specify single and double precision rounding regardless of the precision specified in the FPCR). If the destination is external memory or an MPU data register, the rounding boundary is determined by the destination data format. If the rounded result of an operation is not exact, then the INEX2 bit is set i teh FPSR exception status byte.

```
BEGIN
    IF GUARD, ROUND AND STICKY=0
        THE (RESULT IS EXACT)
            DON'T SET INEX2
            DON'T CHANGE THE INTEMEDIATE RESULT
        ELSE (RESULTS IS INEXACT)
            SET INEX2 IN THE FPSR EXEC BYTE
        SELECT THE ROUNDING MODE
            RM: IF INTERMEDIATE RESULTS IS NEGATIVE
                THEN ADD 1 TO LSB
            RN: IF GUARD=1 THEN
                IF ROUND AND STICKY=0 AND LSB=1 THEN
                    INCREMENT LSB
                ELSE IF ROUND OR STICKY=1 THEN
                    INCREMENT LSB
                  ENDIF
                ENDIF
            RP: IF INTERMEDIATE RESULT IS POSITIVE
                THEN ADD 1 TO LSB
            RZ: (FALL THROUGH, GUARD, ROUND, AND STICKY ARE
                CHOPPED)
        END SELECT
        IF OVERFLOW=1
            THEN
                SHIFT MANTISSA RIGHT BY ONE BIT
                ADD 1 TO EXPONENT
        ENDIF
        SET GUARD, ROUND, AND STICKY TO 0
    ENDIF
END
```

Figure 9-12. Rounding Algorithm

Trap Disabled Results:

The rounded result is stored to the destination

Trap Enabled Results:

The rounded result is stored in the destination, and an exception is reported. If the destination for an FMOVE instruction is memory or an MPU data register, a post-instruction exception is taken immediately, and the FSAVE state frame indicates an E1 (CU) exception. If the instruction is an FMOVE to a floating-point data register, FABS, or FNEG, a pre-instruction exception is reported, and the FSAVE state frame also indicates an E1 (CU) exception. For FADD, FSUB, FMUL, FDIV, and FSQRT instructions, a pre-instruction exception is normally reported as a pre-instruction exception for the next floating-point exception decoded by the integer unit. However, if an FMOVE instruction already in progress earlier in the floating-point pipeline generates a post-instruction exception, the inexact exception takes precedence and is reported as a post-instruction exception. This prevents out of order exception reporting. For both the pre- and the post-instruction exception, the FSAVE state frame indicates an E3 (CU) exception.

The address of the instruction that generated the inexact result is available to the trap handler in the FPIAR. The trap handler can determine the location of the operand(s) by examining the instruction. In the case of a memory destination, the evaluated EA of the operand is available i the EA field of the post-instruction stack frame. When an FSAVE is executed by an inexact trap handler, the value of the exceptional operand is not defined. An inexact exception differs from the other exception in this respect. If an inexact condition is the only exception that occurred during the execution of an instruction, the value of the exceptional operand is invalid. If multiple exceptions occur during an instruction, the exceptional operand value is related to a higher priority exception.

NOTE

The IEEE standard specifies that inexactness should be signaled on overflow as well as for rounding. The processor implements this via the INEX bit in the FPSR AEXC byte. However, the standard also indicates that the inexact trap should be taken if an overflow occurs with the overflow trap disabled and the inexact trap enabled. Therefore, the processor takes the inexact trap if this combination of conditions occurs, even though the INEX1 or INEX2 bits may not be set in the FPSR EXC byte. In this case, INEX is set in the AEXC byte and OVFL is set in both the EXC and AEXC bytes.

9.8.10 Inexact Result on Decimal Input

In a general sense, inexact result 1 (INEX1) is the condition that exists when a packed decimal operand cannot be converted exactly to extended precision in the current rounding mode. The processor provides two inexact bits (INEX1 and INEX2) to help distinguish between inexact results generated by decimal input conversios (INEX1) and other inexact results (INEX2). Since packed decimal real operands are not directly supported by the MC68040, INEX1 is never set by the processor, but is provided as a latch so that emulation software can report this exceptio if required for MC68881/MC68882 compatibility.

SECTION 10
INSTRUCTION TIMING

This section will contain brief information on the instruction timing for the MC68040 after the timing has been finalized. Detailed information on instruction timing will be furnished in the MC68040DH/AD, *MC68040 Designer's Handbook*.

Table 10-1 list the floating-point instruction timing that are directly supported by the MC68040.

Table 10-1. MC68040 Preliminary Floating-Point Unit Instruction Timings (Sheet 1 of 5)

Instruction	Opclass	Size	Round Precision	Operands	Time (in IU Cycles)
FADD,FSUB	0	—	any	norm,norm	2(3)/3/2(3)
	0	—	any	norm,zero	2(3)/3/2(3)
	0	—	any	zero,zero	4/0/0
	0	—	any	—,inf	4/0/0
	0	—	any	—,NAN	4/0/0
	2	S,D	any	norm,norm	2(3)/3/2(3)
	2	S,D	any	norm,zero	2(3)/3/2(3)
	2	S,D	any	zero,zero	4/0/0
	2	S,D	any	—,inf	4/0/0
	2	S,D	any	—,NAN	4/0/0
	2	X	any	norm,norm	3(4)/3/2(3)
	2	X	any	norm,zero	3(4)/3/2(3)
	2	X	any	zero,zero	5/0/0
	2	X	any	—,inf	5/0/0
	2	X	any	—,NAN	5/0/0
FMUL	0	—	any	norm,norm	2(3)/5/2(3)
	0	—	any	—,zero	4/0/0
	0	—	any	—,inf	4/0/0
	0	—	any	—,NAN	4/0/0

Table 10-1. MC68040 Preliminary Floating-Point Unit Instruction Timings (Sheet 2 of 5)

Instruction	Opclass	Size	Round Precision	Operands	Time (in IU Cycles)
FMUL	2	S,D	any	norm,norm	2(3)/5/2(3)
	2	S,D	any	—,zero	4/0/0
	2	S,D	any	—,inf	4/0/0
	2	S,D	any	—,NAN	4/0/0
	2	X	any	norm,norm	3(4)/5/2(3)
	2	X	any	—,zero	5/0/0
	2	X	any	—,inf	5/0/0
	2	X	any	—,NAN	5/0/0
FDIV	0	—	any	norm,norm	2(3)/37.5/2(3)
	0	—	any	—,zero	4/0/0
	0	—	any	—,inf	4/0/0
	0	—	any	—,NAN	4/0/0
	2	S,D	any	norm,norm	2(3)/37.5/2(3)
	2	S,D	any	—,zero	4/0/0
	2	S,D	any	—,inf	4/0/0
	2	S,D	any	—,NAN	4/0/0
	2	X	any	norm,norm	3(4)/37.5/2(3)
	2	X	any	—,zero	5/0/0
	2	X	any	—,inf	5/0/0
	2	X	any	—,NAN	5/0/0
FSQRT	0	—	any	norm	2(3)/103/2(3)
	0	—	any	(zero|inf|NAN)	4/0/0
	2	S,D	any	norm	2(3)/103/2(3)
	2	S,D	any	(zero|inf|NAN)	4/0/0
	2	X	any	norm	3(4)/103/2(3)
	2	X	any	(zero|inf|NAN)	5/0/0
FMOVE,FABS,FNEG	0	—	X	(norm|zero|inf)	2/0/0
	0	—	X	NAN	3/0/0
	0	—	S,D	norm	5/0/0
	0	—	S,D	(zero|inf)	3/0/0
	0	—	S,D	NAN	4/0/0

Table 10-1. MC68040 Preliminary Floating-Point Unit Instruction Timings (Sheet 3 of 5)

Instruction	Opclass	Size	Round Precision	Operands	Time (in IU Cycles)
FMOVE,FABS,FNEG	2	S	any	(norm|zero|inf)	3/0/0
	2	S	any	NAN	4/0/0
	2	D	D,X	(norm|zero|inf)	3/0/0
	2	D	D,X	NAN	4/0/0
	2	D	S	norm	5/0/0
	2	D	S	(zero|inf)	4/0/0
	2	D	S	NAN	5/0/0
	2	X	X	(norm|zero|inf)	4/0/0
	2	X	X	NAN	5/0/0
	2	X	S,D	norm	6/0/0
	2	X	S,D	(zero|inf)	5/0/0
	2	X	S,D	NAN	6/0/0
	2	B,W	any	(+norm|zero)	1.5(11)/4.5/2[1]
	2	L	D,X	(+norm|zero)	1.5(11)/4.5/2[1]
	2	L	S	(+norm|zero)	1.5(12.5)/4.5/2[1]
	2	B,W	any	−norm	1.5(11.5)/5/2[1]
	2	L	D,X	−norm	1.5(11.5)/5/2[1]
	2	L	S	−norm	1.5(13)/5/2[1]
FMOVE	3	S,D	any	any	3/0/0
	3	X	any	any	4/0/0
	3	B,W,L	any	+(norm|zero)	3(9)/1.5/3.5[2]
	3	B,W,L	any	−(norm|zero)	3(10)/1.5/4.5[2]
FMOVEM	4	—	—	—	2+(2 per reg)/0/0[3]
	5	—	-	—	2+(2 per reg)/0/0[3]
	6	—	—	—	2+(3 per reg)/0/0[3]
	7	—	—	—	2+(3 per reg)/0/0[3]
FCMP	0	—	any	norm,norm	2(3)/3/1
	0	—	any	norm,zero	2(3)/3/1
	0	—	any	zero,zero	4/0/0
	0	—	any	—,inf	4/0/0
	0	—	any	—,NAN	4/0/0

Table 10-1. MC68040 Preliminary Floating-Point Unit Instruction Timings (Sheet 4 of 5)

Instruction	Opclass	Size	Round Precision	Operands	Time (in IU Cycles)
FCMP	2	S,D	any	norm,norm	2(3)/3/1
	2	S,D	any	norm,zero	2(3)/3/1
	2	S,D	any	zero,zero	4/0/0
	2	S,D	any	—,inf	4/0/0
	2	S,D	any	—,NAN	4/0/0
	2	X	any	norm,norm	3(4)/3/1
	2	X	any	norm,zero	3(4)/3/1
	2	X	any	zero,zero	5/0/0
	2	X	any	—,inf	5/0/0
	2	X	any	—,NAN	5/0/0
FSAVE	null	frame	—	—	~2
	idle	frame	—	—	~2+(time for pipe to idle)
	unimp	frame	—	—	~18+(time for pipe to idle)
	busy	frame	—	—	~35+(time for pipe to idle)
FRESTORE	null	frame	—	—	~20
	idle	frame	—	—	~2+(time for pipe to idle)
	unimp	frame	—	—	~16+(time for pipe to idle)
	busy	frame	—	—	~30+(time for pipe to idle)
	error	frame	—	—	~2

Table 10-1. MC68040 Preliminary Floating-Point Unit Instruction Timings (Sheet 5 of 5)

NOTES:

1. Memory-to-Register of integer data types require a pass through the FPU pipe to convert the data to floating point format. The result of this conversion is presented to the CU (the first pipe stage) where the desired operation begins (possibly starting a second pass through the pipe). The integer unit (IU) is released and can execute other instructions once the data has been transferred to the FPU (during the first CU cycle).

   The data conversion (the first pass) executes with the following times:
   Positive norm or zero: 1.5(9)/4.5/2
   Negative norm: 1.5(9.5)/5/2
   To arbitrarily calculate the execution time of a mem-to-reg instructionwith integer data type (other than FMOVE, FNEG, or FABS), use the abovetimes for the first pass, then look up the opclass 0 version of the operation, subtract one (1) from the CU time, and you will have thesecond pass timings.

2. Register-to-Memory of integer data types require a pass through the FPU pipe to convert the data from floating point format to integer. Register-to-Memory instructions are normally handled entirely by the CU, so the result of the conversion is presented to the CU where the data move then completes. The IU is not released until it has received the converted data (during the last CU cycle).

3. All FMOVEM instructions wait for the pipe to idle before starting.

GENERAL NOTES:

Times in parentheses are the total time that that stage uses to execute an instruction even though the stage may pass data to the next stage earlier. So "2(3)/5/2(3)" means that the instruction takes 2+5+2 cycles to execute, but stages 1 and 3 were actually busy for 3 cycles each.

Different rounding modes (i.e. round to zero, etc) never incur a time penalty.

The order of operands is generally not significant (for timing purposes).

Preceding instructions with an S or D (i.e. FADD ➧ FSADD) has the same effect as setting the rounding precision to that precision.

SECTION 11
ELECTRICAL CHARACTERISTIC

The following paragraphs provide information on the maximum rating and thermal characteristics for the MC68040. Detail information on timing specifications for power considerations, DC electrical characteristics and AC timing specifications can be found in the MC68040EC/D, *MC68040 Electrical Specifications* and MC68040DH/D, *MC68040 Designer's Handbook*.

11.1 MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	−0.3 to +7.0	V
Input Voltage	V_{in}	−0.5 to +7.0	V
Maximum Operating Junction Temperature	T_J	110	°C
Storage Temperature Range	T_{stg}	−55 to 150	°C

11.2 THERMAL CHARACTERISTICS — PGA PACKAGE

Characteristic	Symbol	Value	Rating
Thermal Resistance — Junction to Case	R_{JC}		°C/W

11

SECTION 12
ORDERING INFORMATION AND MECHANICAL DATA

This section contains the pin assignments and package dimensions of the MC68040. In addition, detailed information is provided to be used as a guide when ordering.

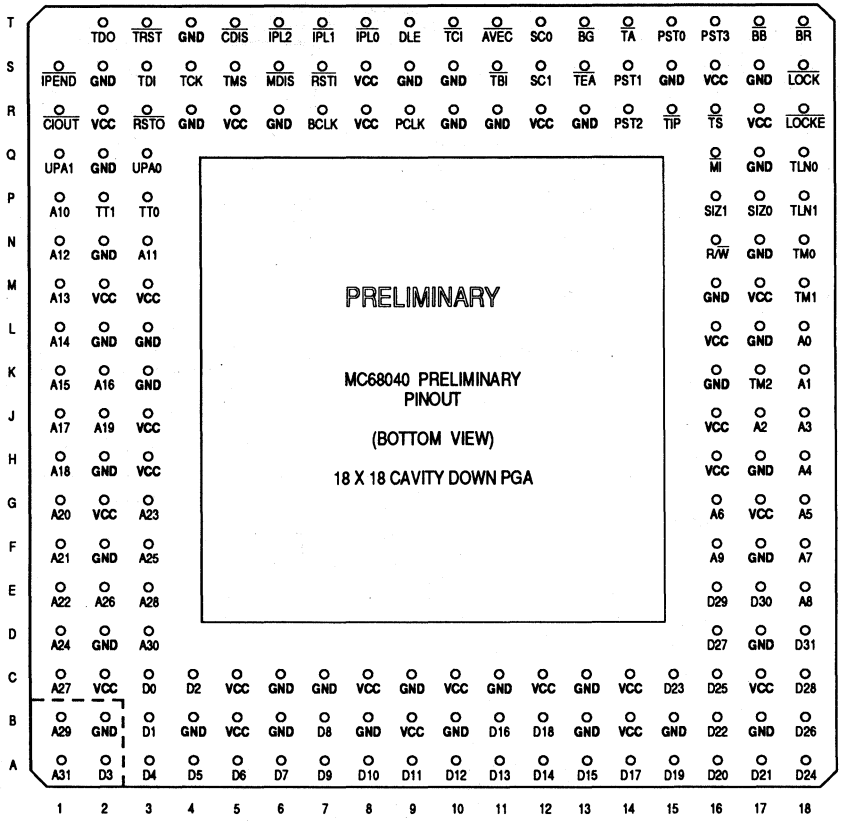
12.1 ORDERING INFORMATION

The following table provides ordering information pertaining to the package type frequency, temperature and Motorola order number for the MC68040.

Package Type	Frequency (MHz)	Temperature	Order Number
Pin Grid Array R Suffix	25.0	TBD	MC68040R25

12.2 PIN ASSIGNMENTS

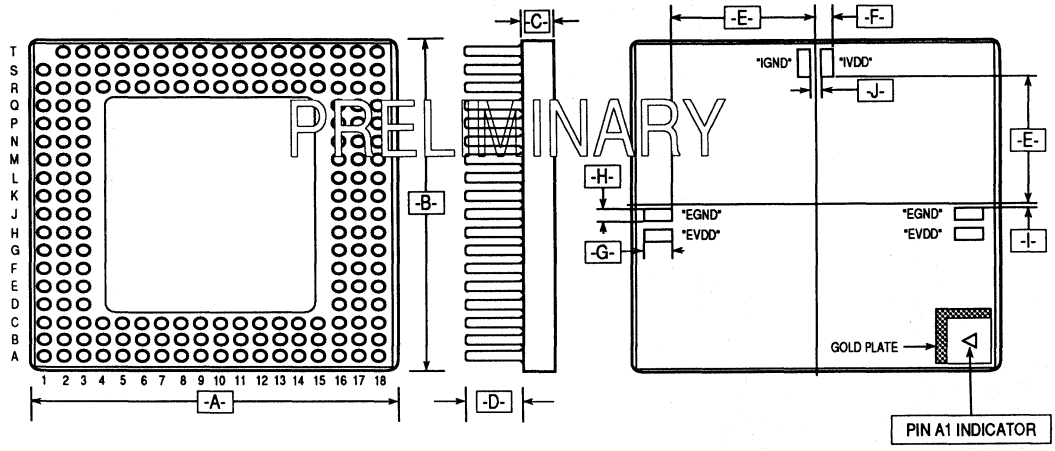
The MC68040 is available in an 179-pin package. The following figure shows the pin assignment of the MC68040.



	GND	VCC
PLL	S9, R6, R10	R8, S8
Internal Logic	C6, C7, C9, C11, C13, K3, K16, L3, M16, R4, R11, R13, S10, T4	C5, C8, C10, C12, C14, H3, H16, J3, J16, L16, M3, R5, R12
Output Drivers	B2, B4, B6, B8, B10, B13, B15, B17, D2, D17, F2, F17, H2, H17, L2, L17, N2, N17, Q2, Q17, S2, S15, S17	B5, B9, B14, C2, C17, G2, G17, M2, M17, R2, R17, S16

12.3 MECHANICAL DATA

The following figure provides the package dimensions for the MC68040.



DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	46.863	47.625	1.845	1.875
B	46.863	47.625	1.845	1.875
C	2.3876	2.9464	.094	.116
D	4.318	4.826	.170	.190
E	18.796		.740	
F	1.778		.070	
G	3.302		.130	
H	1.524		0.060	
I	1.016		0.040	
J	0.508		0.020	

APPENDIX A
M68000 FAMILY SUMMARY

This Appendix summarizes the characteristics of the microprocessors in the M68000 Family. The M68000PM/AD, *M68000 Programmer's Reference Manual* includes more detailed information on the M68000 Family differences.

Attribute	MC68000	MC68008	MC68010	MC68020	MC68030	MC68040
Data Bus Size (Bits)	16	8	16	8, 16, 32	8, 16, 32	32
Address Bus Size (Bits)	24	20	24	32	32	32
Instruction Cache (In Bytes)	—	—	3[1] (Words)	256	256	4096
Data Cache (In Bytes)	—	—	—	—	256	4096

NOTE 1: The MC68010 supports a 3-word cache for the loop mode.

Virtual Interfaces

MC68010, MC68020, MC68030	Virtual Memory/Machine
M68040	Virtual Memory
MC68010, MC68020, MC68030, MC68040	Provide Bus Error Detection, Fault Recovery
MC68030, MC68040	On-Chip MMU

Coprocessor Interface

MC68000, MC68008, MC68010	Emulated in Software
MC68020, MC68030	In Microcode
MC68040	Emulated in Software (On-Chip Floating-Point Unit)

Word/Long Word Data Alignment

MC68000, MC68008, MC68010	Word/Long Data, Instructions, and Stack Must be Word Aligned
MC68020, MC68030, MC68040	Only Instructions Must be Word Aligned (Data Alignment Improves Performance)

Control Registers

MC68000, MC68008	None
MC68010	SFC, DFC, VBR
MC68020	SFC, DFC, VBR, CACR, CAAR
MC68030	SFC, DFC, VBR, CACR, CAAR, CRP, SRP, TC, TT0, TT1, MMUSR
MC68040	SFC, DFC, VBR, CACR, URP, SRP, TC, DTT0, DTT1, ITT0, ITT1, MMUSR

Stack Pointer

MC68000, MC68008, MC68010	USP, SSP
MC68020, MC68030, MC68040	USP, SSP (MSP, ISP)

Status Register Bits

MC68000, MC68008, MC68010	T, S, I0/I1/I2, X/N/Z/V/C
MC68020, MC68030, MC68040	T0, T1, S, M, I0/I1/I2, X/N/Z/V/C

Function Code/Address Space

MC68000, MC68008	FC2–FC0=7 is Interrupt Acknowledge Only
MC68010, MC68020, MC68030, MC68040	FC2–FC0=7 is CPU Space
MC68040	User, Supervisor, and Acknowledge

Indivisible Bus Cycles

MC68000, MC68008, MC68010	Use $\overline{AS}$ Signal
MC68020, MC68030	Use $\overline{RMC}$ Signal
MC68040	Use $\overline{LOCK}$ and $\overline{LOCKE}$ Signal

Stack Frames

MC68000, MC68008	Supports Original set
MC68010	Supports Formats $0, $8
MC68020/MC68030	Supports Formats $0, $1, $2, $9, $A, $B
MC68040	Supports Formats $0, $1, $2, $3, $7

Addressing Modes

MC68020, MC68030, and MC68040 Extensions	Memory indirect addressing modes, scaled index, and larger displacements. Refer to specific data sheets for details.

MC68020, MC68030, and MC68040 Instruction Set Extensions		Applies To		
Instruction	**Notes**	**MC68020**	**MC68030**	**MC68040**
Bcc	Supports 32-Bit Displacements	✓	✓	✓
BFxxxx	Bit Field Instructions (BCHG, BFCLR, BFEXTS, BFEXTU, BFFFO, BFINS, BFSET, BFTST)	✓	✓	✓
BKPT	New Instruction Functionally	✓	✓	
BRA	Supports 32-Bit Displacements	✓	✓	✓
BSR	Supports 32-Bit Displacement	✓	✓	✓
CALLM	New Instruction	✓		
CAS, CAS2	New Instructions	✓	✓	✓
CHK	Supports 32-Bit Operands	✓	✓	✓
CHK2	New Instruction	✓	✓	✓
CINV	Cache Maintenance Instruction			✓
CMPI	Supports Program Counter Relative Addressing Modes	✓	✓	✓
CMP2	New Instruction	✓	✓	✓
CPUSH	Cache Maintenance Instruction			✓
cp	Coprocessor Instructions	✓	✓	
DIVS/DIVU	Supports 32-Bit and 64-Bit Operands	✓	✓	✓
EXTB	Supports 8-Bit Extend to 32-Bits	✓	✓	✓
FABS	New Instruction			✓
FADD	New Instruction			✓
FBcc	New Instruction			✓
FCMP	New Instruction			✓
FDBcc	New Instruction			✓
FDIV	New Instruction			✓
FMOVE	New Instruction			✓
FMOVEM	New Instruction			✓
FMUL	New Instruction			✓

— Continued —

MC68020, MC68030, and MC68040 Instruction Set Extensions		Applies To		
Instruction	**Notes**	**MC68020**	**MC68030**	**MC68040**
FNEG	New Instruction			✓
FRESTORE	New Instruction			✓
FSAVE	New Instruction			✓
FScc	New Instruction			✓
FSQRT	New Instruction			✓
FSUB	New Instruction			✓
FTRAPcc	New Instruction			✓
FTST	New Instruction			✓
LINK	Supports 32-Bit Displacement	✓	✓	✓
MOVE16	New Instruction			✓
MOVEC	Supports New Control Registers	✓	✓	✓
MULS/MULU	Supports 32-Bit Operands	✓	✓	✓
PACK	New Instruction	✓	✓	✓
PFLUSH	MMU Instruction		✓	✓
PLOAD	MMU Instruction		✓	
PMOVE	MMU Instruction		✓	
PTEST	MMU Instruction		✓	✓
RTM	New Instruction	✓		
TST	Supports Program Counter Relative Addressing Modes	✓	✓	✓
TRAPcc	New Instruction	✓	✓	✓
UNPK	New Instruction	✓	✓	✓

A

APPENDIX B
MC68040 FLOATING-POINT EMULATION

The MC68040 is user object code compatible with the MC68030 and MC68881/MC68882. The MC68040 floating-point unit is optimized to directly execute the most commonly used subset of the extensive MC68881/MC68882 instruction set.

The MC68040 provides specialized trap functions to facilitate high speed emulation of all indirectly supported floating-point instructions. These functions couple with Motorola's floating-point software package ensure complete object code compatibility .

The MC68040 directly supports portions of the MC68881/MC68882 instruction set through hardware, and the remainder by providing special traps and/or stack frames for the unimplemented instructions and data types. The intent of the MC68040 design is to provide full user code compatibility with the MC68881/MC68882 instruction set.

For all MC68040 floating point instructions the "coprocessor ID" field must be 001.

Table B-1 lists the floating-point instructions directly supported by the MC68040 and table B-2 lists the floating-point instructions indirectly supported.

Table B-1. Directly Supported Floating-Point Instructions

Mnemonic	Description
FABS	Floating-Point Absolute Value
FADD	Floating-Point Add
FBcc	Floating-Point Branch Conditionally
FCMP	Floating-Point Compare
FDBcc	Floating-Point Test Condition, Decrement, and Branch
FDIV	Floating-Point Divide
FMOVE	Move Floating-Point Data Register
FMOVE.L	Move Floating-Point System Control Register
FMOVEM	Move Multiple Floating-Point System Data Register
FMOVEM.L	Move Multiple Floating-Point Control Data Register
FMUL	Floating-Point Multiply
FNEG	Floating-Point Negate
FNOP	No Operation
FRESTORE	Restore Internal Floating-Point State
FSAVE	Save Internal Floating-Point State
FScc	Set According to Floating-Point Condition
FSQRT	Floating-Point Square Root
FSUB	Floating-Point Subtract
FTRAPcc	Trap on Floating-Point Condition
FTST	Test Floating-Point Operand

Table B-2. Indirectly Supported Floating-Point Instructions

Mnemonic	Description
FACOS	Floating-Point Arc Cosine
FASIN	Floating-Point Arc Sine
FATAN	Floating-Point Arc Tangent
FATANH	Floating-Point Hyperbolic Arc Tangent
FCOS	Floating-Point Cosine
FCOSH	Floating-Point Hyperbolic Cosine
FETOX	Floating-Point e^x
FETOXL	Floating-Point $e^x - 1$
FGETEXP	Floating-Point Get Exponent
FGETMAN	Floating-Point Get Mantissa
FINT	Floating-Point Integer Part
FINTRZ	Floating-Point Integer Part, Round-to-Zero
FLOG10	Floating-Point $\log_{10}$
FLOG10	Floating-Point $\log_2$
FLOGN	Floating-Point $\log_e$
FLOGNP1	Floating-Point $\log_e (x+1)$
FSQRT	Floating-Point Square Root
FMOD	Floating-Point Modulo Remainder
FMOVECR	Floating-Point Move Constant ROM
FREM	Floating-Point IEEE Remainder
FSCALE	Floating-Point Scale Exponent
FSGLDIV	Floating-Point Single Precision Divide
FSFLMUL	Floating-Point Single Precision Multiply
FSIN	Floating-Point Sine
FSINCOS	Floating-Point Simultaneous Sine and Cosine
FSINH	Floating-Point Hyperbolic Sine
FTAN	Floating-Point Tangent
FTANH	Floating-Point Hyperbolic Tangent
FTENTOX	Floating-Point 10^x
FTWOTOX	Floating-Point 2^x

Contact your local Motorola sales office or representative for information on the floating-point software package for the MC68040 and how to order it.

GLOSSARY

Aliasing
Mapping identical information to the same address-registers or memory locations which tells you what the op code acts on.

Big-Endian
A byte-ordering method in memory where the address n of a word corresponds to the most significant byte. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, and 3, with 3 being the most significant byte.

Breakpoint
A point in a program that allows a conditional interruption to permit visual checking, printouts, or other analysis.

Bus Snoop
CMMU operation in which M bus addresses marked global are compared to the data cache tags. If a tag matches the M bus address, the cache line is copied back to memory (if modified) and invalidated. Bus snooping is necessary to maintain cache coherency in a multimaster system.

Cache Coherency
Caches are coherent if a processor performing a read from its cache is supplied with data corresponding to the most recent value written to memory or to another processor's cache.

Cache
Small, high-speed memory containing recently accessed data and/or instructions.

Copyback
A CMMU operation is which a cache line is copied back to memory to enforce cache coherency. A copyback is done either on a snoop that hits on modified cache data or as a result of a copyback command initiated by the processor.

Denormalized Numbers
A floating-point number having all zeros in the exponent and a non-zero value in the fraction/mantissa.

Dirty Data
The data on line (in the chip cache) is valid but not consistent with memory (see stale data). Dirty data is the most recent data.

Dyadic
A mathematical operator indicated by writing the symbols of two vectors without a dot or cross between (as AB).

Dyadic Operations
An operation on two operands.

Little Endian
A byte-ordering method in memory where the address n of a word corresponds to the most significant byte. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, and 0, with 3 being the most significant byte.

Monadic Operation
An operation on one operand, for example, negation.

Orthogonally
Intersecting or lying at right angles.

Overflow
An error condition that occurs during arithmetic operations when the result cannot be stored accurately in the destination register(s). For example, if two 32-bit numbers are added, this sum may require 33 bits due to carry. Since the 32-bit register cannot represent this sum, an overflow condition occurs.

Pipelining
A technique that breaks instruction execution into distinct steps so that multiple steps can be performed at the same time.

Sink Data
To replace data in a cache line.

Stale Data
The data in external device (such as main memory or a DMA controller cache) is outdated and requires replacement from MPU write operation (see dirty data).

Three-Statable
Means the high impedance state of a three-state device.

Underflow
An error condition that occurs during arithmetic operations when the result cannot be stored accurately in the destination register(s). For example, an underflow can happen if two floating-point fractions are multiplied and the result is a single-precision number. The result may require a larger exponent and/or mantissa than the single-precision format makes available.

Writethrough
A memory update policy in which all processor write cycles are written to both the cache and memory.

INDEX

— A —

— B —

— C —

I

— D —

— E —

— F —

— I —

— L —

— M —

— N —

— O —

— P —

— Q —

— R —

I

— S —

— T —

— U —

— V —

— W —

— Z —

I

Literature Distribution Centers:

USA: Motorola Literature Distribution; P.O. Box 20912; Phoenix, Arizona 85036.
EUROPE: Motorola Ltd.; European Literature Center; 88 Tanners Drive, Blakelands, Milton Keynes, MK14 5BP, England.
ASIA PACIFIC: Motorola Semiconductors H.K. Ltd.; P.O. Box 80300; Cheung Sha Wan Post Office; Kowloon Hong Kong.
JAPAN: Nippon Motorola Ltd.; 3-20-1 Minamiazabu, Minato-ku, Tokyo 106 Japan.