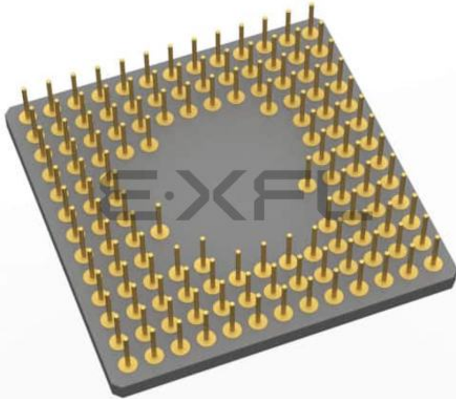


Welcome to E-XFL.COM

Understanding Embedded - Microprocessors

Embedded microprocessors are specialized computing chips designed to perform specific tasks within an embedded system. Unlike general-purpose microprocessors found in personal computers, embedded microprocessors are tailored for dedicated functions within larger systems, offering optimized performance, efficiency, and reliability. These microprocessors are integral to the operation of countless electronic devices, providing the computational power necessary for controlling processes, handling data, and managing communications.

Applications of Embedded - Microprocessors

Embedded microprocessors are utilized across a broad spectrum of applications, making them indispensable in

Details	
Product Status	Obsolete
Core Processor	68030
Number of Cores/Bus Width	1 Core, 32-Bit
Speed	40MHz
Co-Processors/DSP	-
RAM Controllers	-
Graphics Acceleration	No
Display & Interface Controllers	-
Ethernet	-
SATA	-
USB	-
Voltage - I/O	5.0V
Operating Temperature	0°C ~ 70°C (TA)
Security Features	-
Package / Case	128-BPGA
Supplier Device Package	128-PGA (34.55x34.55)
Purchase URL	https://www.e-xfl.com/pro/item?MUrl=&PartUrl=mc68030rc40c

Email: info@E-XFL.COM Address: Room A, 16/F, Full Win Commercial Centre, 573 Nathan Road, Mongkok, Hong Kong

TABLE OF CONTENTS (Continued)

Paragraph Number	Title	Page Number
6.3.1.2	Data Burst Enable	6-21
6.3.1.3	Clear Data Cache	6-21
6.3.1.4	Clear Entry in Data Cache	6-21
6.3.1.5	Freeze Data Cache	6-22
6.3.1.6	Enable Data Cache	6-22
6.3.1.7	Instruction Burst Enable	6-22
6.3.1.8	Clear Instruction Cache	6-22
6.3.1.9	Clear Entry in Instruction Cache	6-22
6.3.1.10	Freeze Instruction Cache	6-23
6.3.1.11	Enable Instruction Cache	6-23
6.3.2	Cache Address Register	6-23

Section 7
Bus Operation

Paragraph Number	Title	Page Number
7.1	Bus Transfer Signals	7-1
7.1.1	Bus Control Signals	7-3
7.1.2	Address Bus	7-4
7.1.3	Address Strobe	7-4
7.1.4	Data Bus	7-5
7.1.5	Data Strobe	7-5
7.1.6	Data Buffer Enable	7-5
7.1.7	Bus Cycle Termination Signals	7-5
7.2	Data Transfer Mechanism	7-6
7.2.1	Dynamic Bus Sizing	7-6
7.2.2	Misaligned Operands	7-13
7.2.3	Effects of Dynamic Bus Sizing and Operand Misalignment	7-19
7.2.4	Address, Size, and Data Bus Relationships	7-22
7.2.5	MC68030 versus MC68020 Dynamic Bus Sizing	7-24
7.2.6	Cache Filling	7-24
7.2.7	Cache Interactions	7-26
7.2.8	Asynchronous Operation	7-27
7.2.9	Synchronous Operation with $\overline{DSACKx}$	7-28
7.2.10	Synchronous Operation with $\overline{STERM}$	7-29
7.3	Data Transfer Cycles	7-30
7.3.1	Asynchronous Read Cycle	7-31
7.3.2	Asynchronous Write Cycle	7-37
7.3.3	Asynchronous Read-Modify-Write Cycle	7-43
7.3.4	Synchronous Read Cycle	7-48

TABLE OF CONTENTS (Continued)

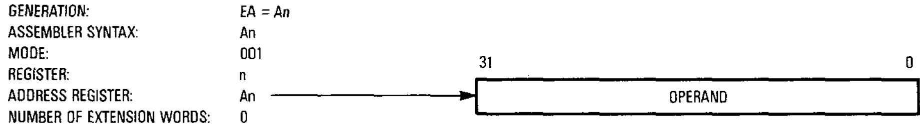
Paragraph Number	Title	Page Number
7.3.5	Synchronous Write Cycle	7-51
7.3.6	Synchronous Read-Modify-Write Cycle	7-54
7.3.7	Burst Operation Cycles	7-59
7.4	CPU Space Cycles	7-68
7.4.1	Interrupt Acknowledge Bus Cycles	7-69
7.4.1.1	Interrupt Acknowledge Cycle — Terminated Normally	7-70
7.4.1.2	Autovector Interrupt Acknowledge Cycle	7-71
7.4.1.3	Spurious Interrupt Cycle	7-74
7.4.2	Breakpoint Acknowledge Cycle	7-74
7.4.3	Coprocessor Communication Cycles	7-74
7.5	Bus Exception Control Cycles	7-75
7.5.1	Bus Errors	7-82
7.5.2	Retry Operation	7-89
7.5.3	Halt Operation	7-91
7.5.4	Double Bus Fault	7-94
7.6	Bus Synchronization	7-95
7.7	Bus Arbitration	7-96
7.7.1	Bus Request	7-98
7.7.2	Bus Grant	7-99
7.7.3	Bus Grant Acknowledge	7-100
7.7.4	Bus Arbitration Control	7-100
7.8	Reset Operation	7-103

Section 8
Exception Processing

Paragraph Number	Title	Page Number
8.1	Exception Processing Sequence	8-1
8.1.1	Reset Exception	8-5
8.1.2	Bus Error Exception	8-7
8.1.3	Address Error Exception	8-8
8.1.4	Instruction Trap Exception	8-9
8.1.5	Illegal Instruction and Unimplemented Instruction Exceptions	8-9
8.1.6	Privilege Violation Exception	8-11
8.1.7	Trace Exception	8-12
8.1.8	Format Error Exception	8-14
8.1.9	Interrupt Exceptions	8-14
8.1.10	MMU Configuration Exception	8-21
8.1.11	Breakpoint Instruction Exception	8-22

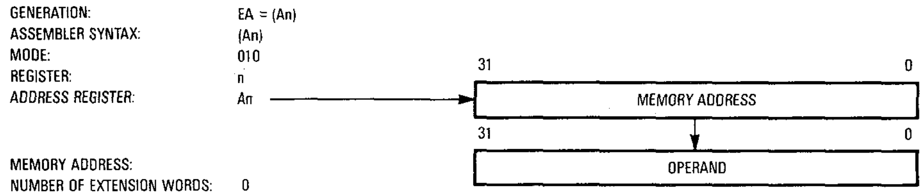
2.4.2 Address Register Direct Mode

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



2.4.3 Address Register Indirect Mode

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



2.4.4 Address Register Indirect with Postincrement Mode

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

SECTION 3
INSTRUCTION SET SUMMARY

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

3

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

3.1 INSTRUCTION FORMAT

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

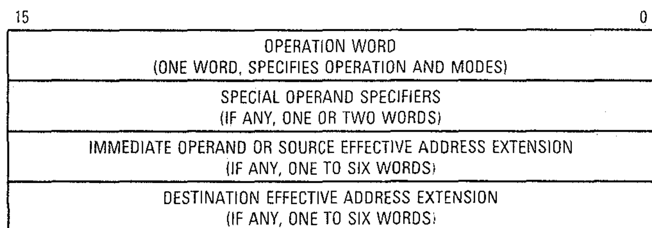


Figure 3-1. Instruction Word General Format

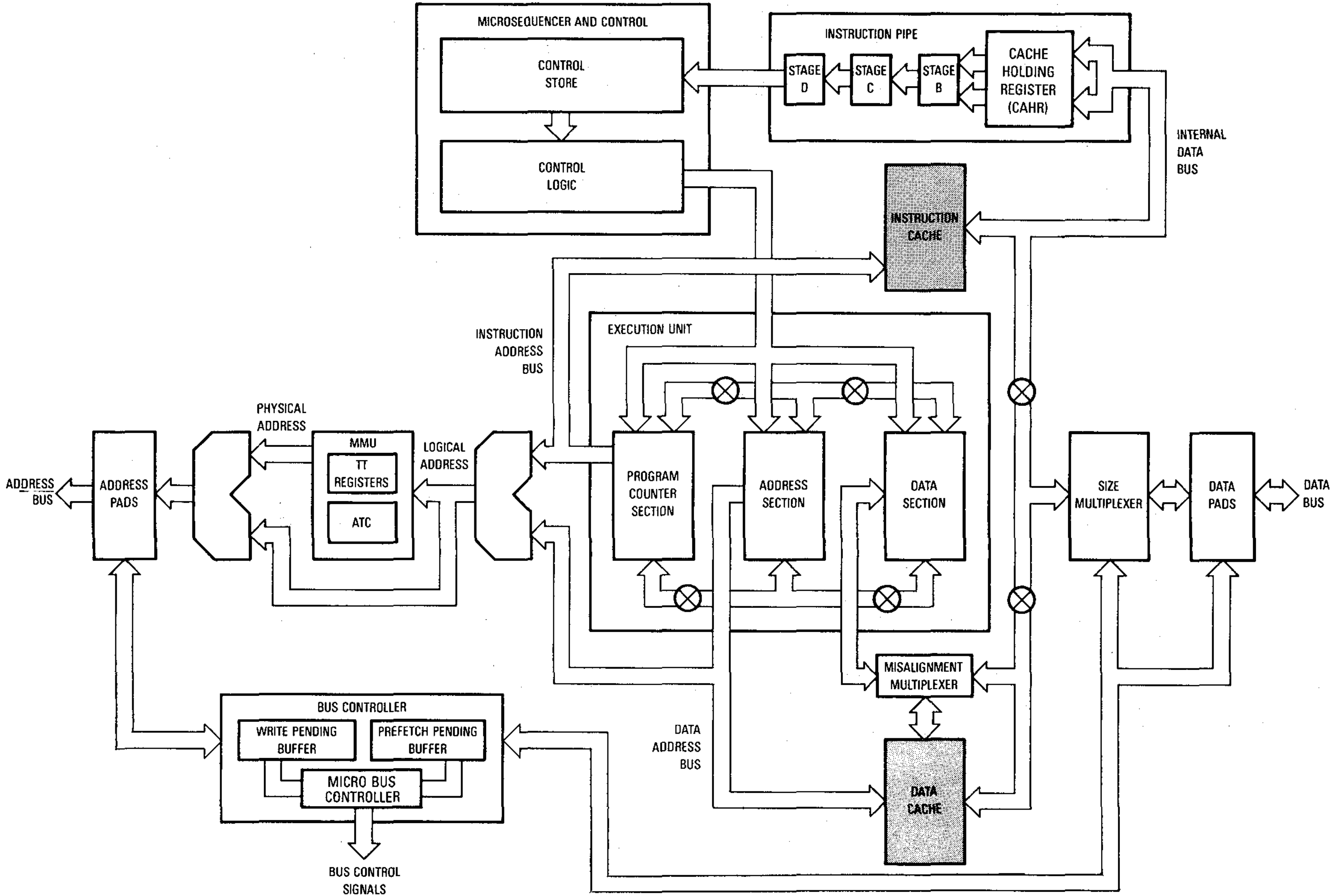


Figure 6-1. Internal Caches and the MC68030

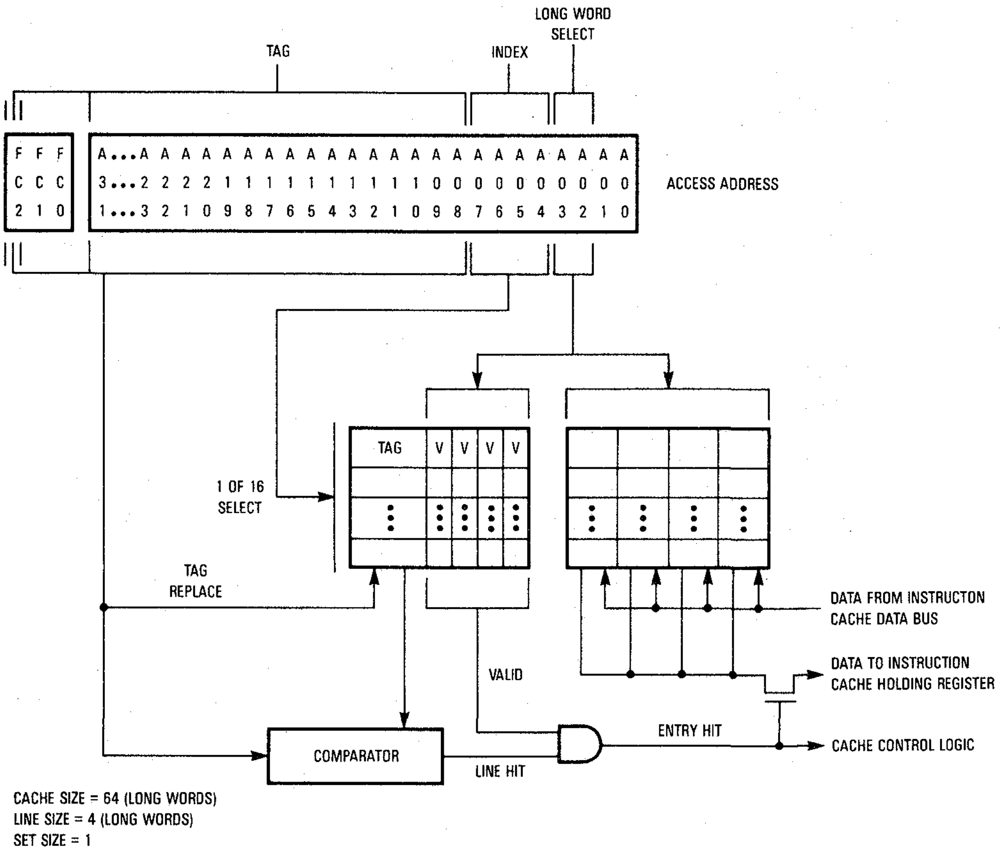


Figure 6-2. On-Chip Instruction Cache Organization

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

In the instruction cache, each of the 16 lines has a tag consisting of the 24 most significant logical address bits, the FC2 function code bit (used to distinguish between user and supervisor accesses), and the four valid bits (one

SECTION 7
BUS OPERATION

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

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

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

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

7.1 BUS TRANSFER SIGNALS

The bus transfers information between the MC68030 and an external memory, coprocessor, or peripheral device. External devices can accept or provide 8 bits, 16 bits, or 32 bits in parallel and must follow the handshake protocol described in this section. The maximum number of bits accepted or provided during a bus transfer is defined as the port width. The MC68030 contains an

State 6

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

State 7

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

State 8

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

State 9

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

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

State 10

The processor issues no new control signals during S10.

7

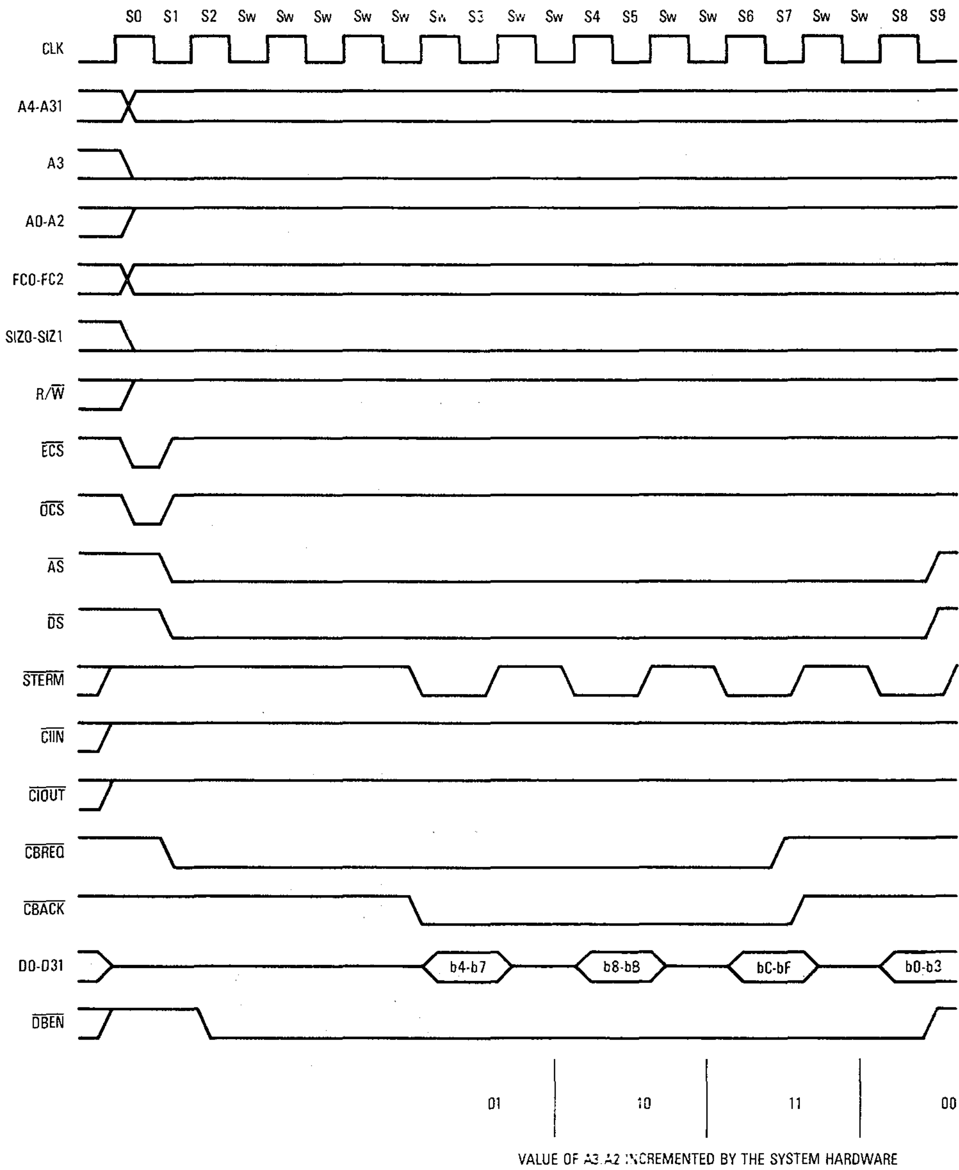


Figure 7-38. Long-Word Operand Request from $07 with Burst Request and Wait Cycle

7.4.1.1 INTERRUPT ACKNOWLEDGE CYCLE — TERMINATED NORMALLY.

When the MC68030 processes an interrupt exception, it performs an interrupt acknowledge cycle to obtain the number of the vector that contains the starting location of the interrupt service routine.

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

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

1. FC0–FC2 are set to seven (FC0/FC1/FC2 = 111) for CPU address space.
2. A1, A2, and A3 are set to the interrupt request level (the inverted values of $\overline{\text{IPL0}}$, $\overline{\text{IPL1}}$, and $\overline{\text{IPL2}}$, respectively).
3. The CPU space type field (A16–A19) is set to $F, the interrupt acknowledge code.
4. A20–A31, A4–A15, and A0 are set to one.

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

signal. When the requesting device receives BG and more than one external device can be bus master, the requesting device should begin whatever arbitration is required. The external device asserts BGACK when it assumes bus mastership and maintains BGACK during the entire bus cycle (or cycles) for which it is bus master. The following conditions must be met for an external device to assume mastership of the bus through the normal bus arbitration procedure:

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

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

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

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

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

9.7 REGISTERS

The registers of the MMU described here are part of the supervisor programming model for the MC68030.

The six registers that control and provide status information for address translation in the MC68030 are the CPU root pointer register (CRP), the supervisor root pointer register (SRP), the translation control register (TC), two independent transparent translation control registers (TT0 and TT1), and the MMU status register (MMUSR). These registers can be accessed directly by programs that execute only at the supervisor level.

9.7.1 Root Pointer Registers

The supervisor root pointer (SRP), used for supervisor accesses only, is enabled or disabled in software. The CPU root pointer (CRP) corresponds to the current translation table for user space (when the SRP is enabled) or for both user and supervisor space (when the SRP is disabled). The CRP is a 64-bit register that contains the address and related status information of the root of the translation table tree for the current task. When a new task begins execution, the operating system typically writes a new root pointer descriptor to the CRP. A new translation table address implies that the contents of the address translation cache (ATC) may no longer be valid. Therefore, the instruction that loads the CRP can optionally flush the ATC.

The SRP is a 64-bit register that optionally contains the address and related status information of the root of the translation table for supervisor area accesses. The SRP is used when operating at the supervisor privilege level only when the supervisor root pointer enable bit (SRE) of the translation control register (TC) is set. The instruction that loads the SRP can optionally flush the ATC. The format of the CRP and SRP is shown in Figure 9-35 and defines the following fields:

Lower/Upper (L/U)
: Specifies that the value contained in the limit field is to be used as the unsigned lower limit of indexes into the translation tables when this bit is set. When this bit is cleared, the limit field is the unsigned upper limit of the translation table indexes.

tection and maps the entire virtual operating system, physical I/O, and physical memory areas. This scheme avoids the requirement for extra lookup levels or pointer manipulations during a task switch to furnish correct access across the user/supervisor boundary. All the operating system has to do when creating the address table for a new task is to set the first upper level table entry to point to the common page table of the supervisor.

To solve the problem of accounting for virtual memory areas assigned to a user task, the operating system uses the existing translation tables to identify these areas. When a valid descriptor points to a given virtual address page, this 8K-byte page of memory has been allocated. This scheme provides areas of memory that are multiples of the 8K-byte page size. Due to the 8K granularity, this scheme would be inadequate for tasks that continually request and return virtual memory space. As a result, some other technique would be used (perhaps auxiliary tables to show virtual space availability). The tasks in this system seldom request additional memory space; any request made is for a large area. This scheme suffices. The application programs and utilities that run in the UNIX (r) environment have similar requirements for memory.

The operating system primitive GetVirtual allocates virtual memory space for tasks. The input parameter is a block size, in bytes; GetVirtual returns the virtual address for the new block. GetVirtual first checks that the requested size is not too large. Then it scans the translation tables looking for an unallocated virtual memory area large enough to hold the requested block. If it does not find enough space, GetVirtual attempts to increase the page table size to its maximum. If this does not provide the space, GetVirtual returns an error indication. When the routine finds enough virtual space for the block, it sets the page descriptors for the block to virgin status (invalid, but allocated). When these pages are first used, a page fault is generated. The operating system allocates a page frame for the page and replaces the descriptor with a valid page descriptor. The status (indicated by a software flag in the invalid descriptor) tells the operating system that the paging device does not have a page image for this page; no read operation from the paging device is required.

When the status of an invalid descriptor indicates that a page image must be read in, primitive SwapInPage, reads in the image. The input parameter for this routine is the invalid descriptor, which contains the disk address of the page image. Before returning, SwapInPage replaces the invalid descriptor with a valid page descriptor that contains the page address. The page is now ready for use.

are dedicated coprocessor instructions that utilize the coprocessor capabilities. The necessary interactions between the main processor and the coprocessor that provide a given service are transparent to the programmer. That is, the programmer does not need to know the specific communication protocol between the main processor and the coprocessor because this protocol is implemented in hardware. Thus, the coprocessor can provide capabilities to the user without appearing separate from the main processor.

In contrast, standard peripheral hardware is generally accessed through interface registers mapped into the memory space of the main processor. To use the services provided by the peripheral, the programmer accesses the peripheral registers with standard processor instructions. While a peripheral could conceivably provide capabilities equivalent to a coprocessor for many applications, the programmer must implement the communication protocol between the main processor and the peripheral necessary to use the peripheral hardware.

The communication protocol defined for the M68000 coprocessor interface is described in **10.2 COPROCESSOR INSTRUCTION TYPES**. The algorithms that implement the M68000 coprocessor interface are provided in the microcode of the MC68030 and are completely transparent to the MC68030 programmer's model. For example, floating-point operations are not implemented in the MC68030 hardware. In a system utilizing both the MC68030 and the MC68881 or MC68882 floating-point coprocessor, a programmer can use any of the instructions defined for the coprocessor without knowing that the actual computation is performed by the MC68881 or MC68882 hardware.

10.1.1 Interface Features

The M68000 coprocessor interface design incorporates a number of flexible capabilities. The physical coprocessor interface uses the main processor external bus, which simplifies the interface since no special-purpose signals are involved. With the MC68030, a coprocessor can use either the asynchronous or synchronous bus transfer protocol. Since standard bus cycles transfer information between the main processor and the coprocessor, the coprocessor can be implemented in whatever technology is available to the coprocessor designer. A coprocessor can be implemented as a VLSI device, as a separate system board, or even as a separate computer system.

Since the main processor and a M68000 coprocessor can communicate using the asynchronous bus, they can operate at different clock frequencies. The system designer can choose the speeds of a main processor and coprocessor

If the format word written to the restore CIR does not represent a valid coprocessor state frame, the coprocessor places an invalid format word in the restore CIR and terminates any current operations. The main processor receives the invalid format code, writes an abort mask (refer to **10.2.3.2.3 Invalid Format Word**) to the control CIR, and initiates format error exception processing (refer to **10.5.1.5 FORMAT ERRORS**).

The cpRESTORE instruction is a privileged instruction. When the main processor accesses a cpRESTORE instruction, it checks the supervisor bit in the status register. If the MC68030 attempts to execute a cpRESTORE instruction while at the user privilege level (status register bit [13] = 0), it initiates privilege violation exception processing without accessing any of the coprocessor interface registers (refer to **10.5.2.3 PRIVILEGE VIOLATIONS**).

10.3 COPROCESSOR INTERFACE REGISTER SET

The instructions of the M68000 coprocessor interface use registers of the CIR set to communicate with the coprocessor. These CIRs are not directly related to the coprocessor's programming model.

Figure 10-4 is a memory map of the CIR set. The registers denoted by asterisks (*) must be included in a coprocessor interface that implements coprocessor instructions in all four categories. The complete register model must be implemented if the system uses all of the coprocessor response primitives defined for the M68000 coprocessor interface.

The following paragraphs contain detailed descriptions of the registers.

10.3.1 Response CIR

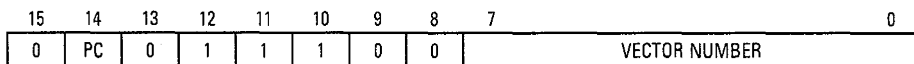
The coprocessor uses the 16-bit response CIR to communicate all service requests (coprocessor response primitives) to the main processor. The main processor reads the response CIR to receive the coprocessor response primitives during the execution of instructions in the general and conditional instruction categories. The offset from the base address of the CIR set for the response CIR is $00. Refer to **10.4 COPROCESSOR RESPONSE PRIMITIVES**.

The MC68030 discards any instruction words that have been prefetched beyond the current scanPC location when this primitive is issued with DR=1 (transfer to main processor). The MC68030 then refills the instruction pipe from the scanPC address in the address space indicated by the status register S bit.

If the MC68030 is operating in the trace on change of flow mode (T1:T0 in the status register contains 01) when the coprocessor instruction begins to execute and if this primitive is issued with DR=1 (from coprocessor to main processor), the MC68030 prepares to take a trace exception. The trace exception occurs when the coprocessor signals that it has completed all processing associated with the instruction. Changes in the trace modes due to the transfer of the status register to main processor take effect on execution of the next instruction.

10.4.18 Take Pre-Instruction Exception Primitive

The take pre-instruction exception primitive initiates exception processing using a coprocessor-supplied exception vector number and the pre-instruction exception stack frame format. This primitive applies to general and conditional category instructions. Figure 10-40 shows the format of the take pre-instruction exception primitive.



15	14	13	12	11	10	9	8	7–0
0	PC	0	1	1	1	0	0	VECTOR NUMBER

Figure 10-40. Take Pre-Instruction Exception Primitive Format

The primitive uses the PC bit as previously described. Bits [0–7] contain the exception vector number used by the main processor to initiate exception processing.

When the main processor receives this primitive, it acknowledges the coprocessor exception request by writing an exception acknowledge mask (refer to **10.3.2 Control CIR**) to the control CIR. The MC68030 then proceeds with exception processing as described in **8.1 EXCEPTION PROCESSING SEQUENCE**. The vector number for the exception is taken from bits [0–7] of the primitive, and the MC68030 uses the four-word stack frame format shown in Figure 10-41.

Note that for the head of fiea #<data>.L,D1, 4+op head, the resulting head of 6 is larger than the instruction-cache-case time of the fetch. A negative number for the execution time of that portion could result (e.g., 4 − min(6,6) = −2). This result would produce the correct execution time since the fetch was completely overlapped and the operation was partially overlapped by the same tail. No changes in the calculation for the operation execution time are required.

Many two-word instructions (e.g., MULU.L, DIV.L, BFSET, etc.) include the fetch immediate effective address (fiea) time or the calculate immediate effective address (ciea) time in the execution time calculation. The timing for immediate data of word length (#<data>.W) is used for these calculations. If the instruction has a source and a destination, the source EA is used for the table lookup. If the instruction is single operand, the effective address of that operand is used.

The following example includes multi-word instructions that refer to the fetch immediate effective address and calculate immediate effective address tables in **11.6 INSTRUCTION TIMING TABLES**.

	Instruction	
1.	MULU.L	(D7),D1:D2
2.	BFCLR	$6000{0:8}
3.	DIVS.L	#$10000,D3:D4

		Head	**Tail**	**CC**
1.	MULU.L (D7),D1:D2			
	fiea #<data>.W,Dn	2+op head	0	2
		4	0	2
	MUL.L EA, Dn	2(op head)	0	44
2.	BFCLR $6000{0:8}			
	fiea #<data>.W,$XXX.W	4	2	6
	BFCLR Mem(<5 bytes)	6	0	14
3.	DIVS.L #$10000,D3:D4			
	fiea #<data>.W,#<data>.L	6 − op head	0	6
		6	0	6
	DIVS.L EA,Dn	0(op head)	0	90

11.6.4 Calculate Immediate Effective Address (ciea) (Continued)

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
SINGLE EFFECTIVE ADDRESS INSTRUCTION FORMAT				
% #⟨data⟩.W,Dn	2 + op head	0	**2**(0/0/0)	**2**(0/1/0)
% #⟨data⟩.L,Dn	4 + op head	0	**4**(0/0/0)	**4**(0/1/0)
% #⟨data⟩.W,(An)	2 + op head	0	**2**(0/0/0)	**2**(0/1/0)
% #⟨data⟩.L,(An)	4 + op head	0	**4**(0/0/0)	**4**(0/1/0)
#⟨data⟩.W,(An) +	2	0	**4**(0/0/0)	**4**(0/1/0)
#⟨data⟩.L,(An) +	4	0	**6**(0/0/0)	**6**(0/1/0)
% #⟨data⟩.W, − (An)	2 + op head	0	**2**(0/0/0)	**2**(0/1/0)
% #⟨data⟩.L, − (An)	4 + op head	0	**4**(0/0/0)	**4**(0/1/0)
% #⟨data⟩.W,(d_{16},An)	4 + op head	0	**4**(0/0/0)	**4**(0/1/0)
% #⟨data⟩.L,(d_{16},An)	6 + op head	0	**6**(0/0/0)	**7**(0/2/0)
% #⟨data⟩.W,$XXX.W	4 + op head	0	**4**(0/0/0)	**4**(0/1/0)
% #⟨data⟩.L,$XXX.W	6 + op head	0	**6**(0/0/0)	**6**(0/2/0)
% #⟨data⟩.W,$XXX.L	6 + op head	0	**6**(0/0/0)	**6**(0/2/0)
% #⟨data⟩.L,$XXX.L	8 + op head	0	**8**(0/0/0)	**8**(0/2/0)
BRIEF FORMAT EXTENSION WORD				
% #⟨data⟩.W,(d_8,An,Xn) or (d_8,PC,Xn)	6 + op head	0	**6**(0/0/0)	**6**(0/2/0)
% #⟨data⟩.L,(d_8,An,Xn) or (d_8,PC,Xn)	8 + op head	0	**8**(0/0/0)	**8**(0/2/0)
FULL FORMAT EXTENSION WORD(S)				
#⟨data⟩.W,(d_{16},An) or (d_{16},PC)	4	0	**8**(0/0/0)	**8**(0/2/0)
#⟨data⟩.L,(d_{16},An) or (d_{16},PC)	6	0	**10**(0/0/0)	**10**(0/2/0)
% #⟨data⟩.W,(d_{16},An,Xn) or (d_{16},PC,Xn)	8 + op head	0	**8**(0/0/0)	**8**(0/2/0)
% #⟨data⟩.L,(d_{16},An,Xn) or (d_{16},PC,Xn)	10 + op head	0	**10**(0/0/0)	**10**(0/2/0)
#⟨data⟩.W,([d_{16},An]) or ([d_{16},PC])	4	0	**12**(1/0/0)	**12**(1/2/0)
#⟨data⟩.L,([d_{16},An]) or ([d_{16},PC])	6	0	**14**(1/0/0)	**14**(1/1/0)
#⟨data⟩.W,([d_{16},An],Xn) or ([d_{16},PC],Xn)	4	0	**12**(1/0/0)	**12**(1/2/0)
#⟨data⟩.L,([d_{16},An],Xn) or ([d_{16},PC],Xn)	6	0	**14**(1/0/0)	**14**(1/1/0)
#⟨data⟩.W,([d_{16},An],d_{16}) or ([d_{16},PC],d_{16})	4	0	**14**(1/0/0)	**15**(1/2/0)
#⟨data⟩.L,([d_{16},An],d_{16}) or ([d_{16},PC],d_{16})	6	0	**16**(1/0/0)	**17**(1/3/0)
#⟨data⟩.W,([d_{16},An],Xn,d_{16}) or ([d_{16},PC],Xn,d_{16})	4	0	**14**(1/0/0)	**15**(1/2/0)
#⟨data⟩.L,([d_{16},An],Xn,d_{16}) or ([d_{16},PC],Xn,d_{16})	6	0	**16**(1/0/0)	**17**(1/3/0)
#⟨data⟩.W,([d_{16},An],d_{32}) or ([d_{16},PC],d_{32})	4	0	**14**(1/0/0)	**16**(1/3/0)
#⟨data⟩.L,([d_{16},An],d_{32}) or ([d_{16},PC],d_{32})	6	0	**16**(1/0/0)	**17**(1/3/0)
#⟨data⟩.W,([d_{16},An],Xn,d_{32}) or ([d_{16},PC],Xn,d_{32})	4	0	**14**(1/0/0)	**15**(1/3/0)
#⟨data⟩.L,([d_{16},An],Xn,d_{32}) or ([d_{16},PC],Xn,d_{32})	6	0	**16**(1/0/0)	**17**(1/3/0)
% #⟨data⟩.W,(B)	8 + op head	0	**8**(0/0/0)	**8**(0/1/0)
% #⟨data⟩.L,(B)	10 + op head	0	**10**(0/0/0)	**10**(0/2/0)

SECTION 12
APPLICATIONS INFORMATION

This section provides guidelines for using the MC68030. First, it discusses the requirements for adapting the MC68030 to MC68020 designs. Then, it describes the use of the MC68881 and MC68882 coprocessors with the MC68030. The byte select logic is described next, followed by memory interface information. A description of external caches, the use of the STATUS and REFILL signals, and power and ground considerations complete the section.

12.1 ADAPTING THE MC68030 TO MC68020 DESIGNS

Perhaps the easiest way to first utilize the MC68030 is in a system designed for the MC68020. This is possible due to the complete compatibility of the asynchronous buses of the MC68020 and MC68030. This section describes how to configure an adapter for the MC68030 to allow insertion into an existing MC68020-based system. Software and architectural differences between the two processors are also discussed. The need for an adapter is absolute because the MC68020 and MC68030 are NOT pin compatible. Use of the adapter board provides the immediate capability for evaluating the programmer's model and instruction set of the MC68030 and for developing software to utilize the MC68030's additional enhanced features. This adapter board also provides a relatively simple method for increasing the performance of an existing MC68020 or MC68020/MC68851 system by insertion of a more advanced 32-bit MPU with an on-chip data cache and an on-chip MMU. Since the adapter board does not support of the synchronous bus interface of the MC68030, performance measurements for the MC68030 used in this manner may be misleading when compared to a system designed specifically for the MC68030.

The adapter board plugs into the CPU socket of an MC68020 target system, drawing power, ground, and clock signals through the socket and running bus cycles in a fashion compatible with the MC68030. The only support hardware necessary is a single 1K-ohm pullup resistor and two capacitors for decoupling power and ground on the adapter board.

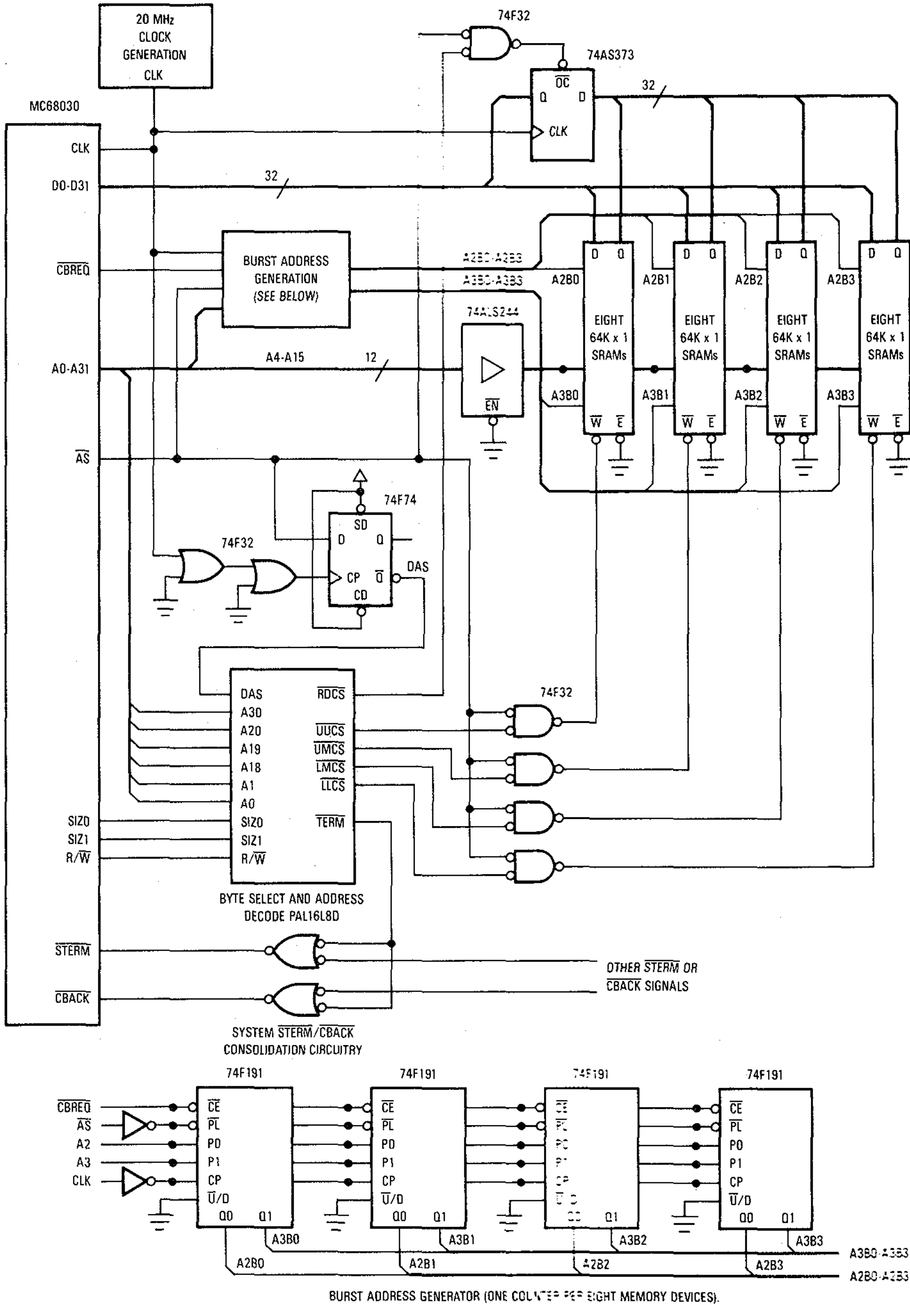


Figure 12-14. Example 2-1-1-1 Burst Mode Memory Bank at 20 MHz, 256K Bytes