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	68020
Number of Cores/Bus Width	1 Core, 32-Bit
Speed	20MHz
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	132-BQFP Bumpered
Supplier Device Package	132-PQFP (46x46)
Purchase URL	https://www.e-xfl.com/product-detail/nxp-semiconductors/mc68020eh20e

LIST OF ILLUSTRATIONS (Concluded)

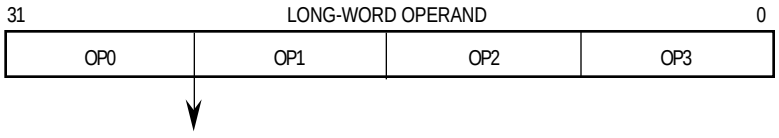
1.1 FEATURES

The main features of the MC68020/EC020 are as follows:

- Object-Code Compatible with Earlier M68000 Microprocessors
- Addressing Mode Extensions for Enhanced Support of High-Level Languages
- New Bit Field Data Type Accelerates Bit-Oriented Applications—e.g., Video Graphics
- An On-Chip Instruction Cache for Faster Instruction Execution
- Coprocessor Interface to Companion 32-Bit Peripherals—the MC68881 and MC68882 Floating-Point Coprocessors and the MC68851 Paged Memory Management Unit
- Pipelined Architecture with High Degree of Internal Parallelism Allowing Multiple Instructions To Be Executed Concurrently
- High-Performance Asynchronous Bus Is Nonmultiplexed and Full 32 Bits
- Dynamic Bus Sizing Efficiently Supports 8-/16-/32-Bit Memories and Peripherals
- Full Support of Virtual Memory and Virtual Machine
- Sixteen 32-Bit General-Purpose Data and Address Registers
- Two 32-Bit Supervisor Stack Pointers and Five Special-Purpose Control Registers
- Eighteen Addressing Modes and Seven Data Types
- 4-Gbyte Direct Addressing Range for the MC68020
- 16-Mbyte Direct Addressing Range for the MC68EC020
- Selection of Processor Speeds for the MC68020: 16.67, 20, 25, and 33.33 MHz
- Selection of Processor Speeds for the MCEC68020: 16.67 and 25 MHz

A block diagram of the MC68020/EC020 is shown in Figure 1-1.

Figure 5-5 shows the transfer (write) of a long-word operand to a word port. In the first bus cycle, the MC68020/EC020 places the four operand bytes on the external bus. Since the address is long-word aligned in this example, the multiplexer follows the pattern in the entry of Table 5-5 corresponding to SIZ0, SIZ1, A0, A1 = 0000. The port latches the data on D31–D16, asserts $\overline{\text{DSACK1}}$ ($\overline{\text{DSACK0}}$ remains negated), and the processor terminates the bus cycle. It then starts a new bus cycle with SIZ1, SIZ0, A1, A0 = 1010 to transfer the remaining 16 bits. SIZ1 and SIZ0 indicate that a word remains to be transferred; A1 and A0 indicate that the word corresponds to an offset of two from the base address. The multiplexer follows the pattern corresponding to this configuration of SIZ1, SIZ0, A1, and A0 and places the two least significant bytes of the long word on the word portion of the bus (D31–D16). The bus cycle transfers the remaining bytes to the word-sized port. Figure 5-6 shows the timing of the bus transfer signals for this operation.



WORD MEMORY		MC68020/EC020				MEMORY CONTROL	
MSB	LSB	SIZ1	SIZ0	A1	A0	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$
OP0	OP1	0	0	0	0	L	H
OP2	OP3	1	0	1	0	L	H

Figure 5-5. Long-Word Operand Write to Word Port Example

Figure 5-7 shows a word write to an 8-bit bus port. Like the preceding example, this example requires two bus cycles. Each bus cycle transfers a single byte. SIZ1 and SIZ0 for the first cycle specify two bytes; for the second cycle, one byte. Figure 5-8 shows the associated bus transfer signal timing.

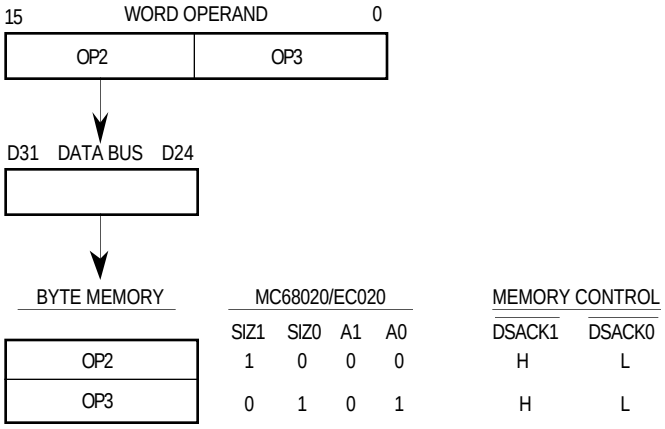


Figure 5-7. Word Operand Write to Byte Port Example

5.2.2 Misaligned Operands

Since operands may reside at any byte boundary, 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 MC68020/EC020 does not enforce any alignment restrictions for data operands (including PC relative data addresses), some performance degradation occurs when additional bus cycles are required for long-word or word operands that are misaligned. For maximum performance, data items should be aligned on their natural boundaries. All instruction words and extension words must reside on word boundaries. Attempting to prefetch an instruction word at an odd address causes an address error exception.

Figure 5-9 shows the transfer (write) of a long-word operand to an odd address in word-organized memory, which requires three bus cycles. For the first cycle, SIZ1 and SIZ0 specify a long-word transfer, and A2–A0 = 001. Since the port width is 16 bits, only the first byte of the long word is transferred. The slave device latches the byte and acknowledges the data transfer, indicating that the port is 16 bits wide. When the processor starts the second cycle, SIZ1 and SIZ0 specify that three bytes remain to be transferred with A2–A0 = 010. The next two bytes are transferred during this cycle. The processor then initiates the third cycle, with SIZ1 and SIZ0 indicating one byte remaining to be transferred with A2–A0 = 100. The port latches the final byte, and the operation is complete. Figure 5-10 shows the associated bus transfer signal timing. Figure 5-11 shows the equivalent operation for a data read cycle.

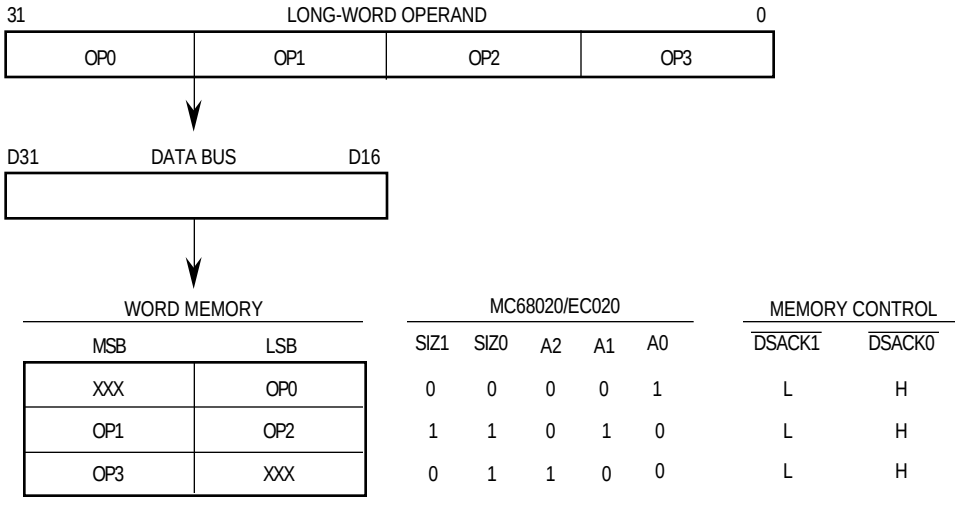


Figure 5-9. Misaligned Long-Word Operand Write to Word Port Example

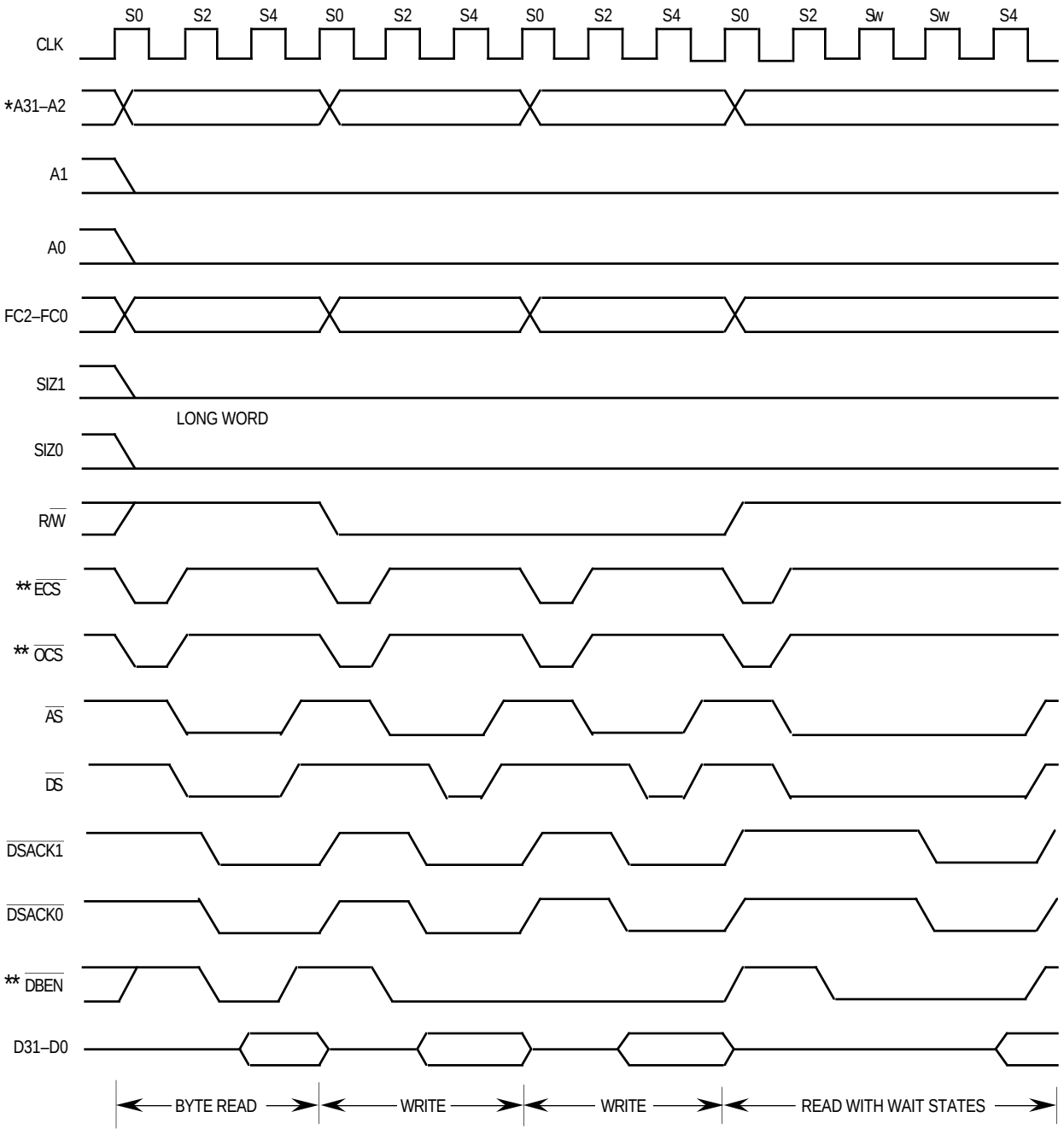
Table 5-6 demonstrates that the processor always prefetches instructions by reading a long word from a long-word address (A1, A0 = 00), regardless of port size or alignment. When the required instruction begins at an odd-word boundary, the processor attempts to fetch the entire 32 bits and loads both words into the instruction cache, if possible, although the second one is the required word. Even if the instruction access is not cached, the entire 32 bits are latched into an internal cache holding register from which the two instructions words can subsequently be referenced. Refer to **Section 8 Instruction Execution Timing** for a complete description of the cache holding register and pipeline operation.

5.2.4 Address, Size, and Data Bus Relationships

The data transfer examples show how the MC68020/EC020 drives data onto or receives data from the correct byte sections of the data bus. Table 5-7 shows the combinations of the SIZ1, SIZ0, A1, and A0 signals that can be used to generate byte enable signals for each of the four sections of the data bus for read and write cycles if the addressed device requires them. The port size also affects the generation of these enable signals as shown in the table. The four columns on the right correspond to the four byte enable signals. Letters B, W, and L refer to port sizes: B for 8-bit ports, W for 16-bit ports, and L for 32-bit ports. The letters B, W, and L imply that the byte enable signal should be true for that port size. A dash (—) implies that the byte enable signal does not apply.

The MC68020/EC020 always drives all sections of the data bus because, at the beginning of a write cycle, the bus controller does not know the port size.

Table 5-7 reveals that the MC68020/EC020 transfers the number of bytes specified by SIZ1, SIZ0 to or from the specified address unless the operand is misaligned or unless the number of bytes is greater than the port width. In these cases, the device transfers the greatest number of bytes possible for the port. For example, if the size is four and A1, A0 = 01, a 32-bit slave can only receive three bytes in the current bus cycle. A 16- or 8-bit slave can only receive one byte. The table defines the byte enables for all port sizes. Byte data strobes can be obtained by combining the enable signals with the $\overline{DS}$ signal. Devices residing on 8-bit ports can use the data strobe by itself since there is only one valid byte for every transfer. These enable or strobe signals select only the bytes required for write or read cycles. The other bytes are not selected, which prevents incorrect accesses in sensitive areas such as I/O.



* For the MC68EC020, A23–A2.
** This signal does not apply to the MC68EC020.

Figure 5-25. Read-Write-Read Cycles—32-Bit Port

The priority scheme is very important in determining the order in which exception handlers execute when several exceptions occur at the same time. As a general rule, the lower the priority of an exception, the sooner the handler routine for that exception executes. For example, if simultaneous trap, trace, and interrupt exceptions are pending, the exception processing for the trap occurs first, followed immediately by exception processing for the trace, and then for the interrupt. When the processor resumes normal instruction execution, it is in the interrupt handler, which returns to the trace handler, which returns to the trap exception handler. This rule does not apply to the reset exception; its handler is executed first even though it has the highest priority because the reset operation clears all other exceptions.

6.1.12 Return from Exception

After the MC68020/EC020 has completed exception processing for all pending exceptions, it 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 context 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. The following paragraphs describe the processing for each of the stack frame types; refer to **6.3 Coprocessor Considerations** for a description of the stack frame type formats.

For a normal four-word frame, 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 frame, 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 6-7. 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 or a ten-word coprocessor midinstruction 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.

For the six-word stack frame, the processor restores the SR and PC values from the stack, increments the active supervisor stack pointer by 12, and resumes normal instruction execution.

MC68020/EC020 to begin exception processing. The MC68020/EC020 never generates coprocessor interface bus cycles with the CpID equal to zero (except via the MOVES instruction).

CpID codes of 000–101 are reserved for current and future Motorola coprocessors, and CpID codes of 110–111 are reserved for user-defined coprocessors. The Motorola CpID code of 001 designates the MC68881 or MC68882 floating-point coprocessor. By default, Motorola assemblers will use a CpID code of 001 when generating the instruction operation codes for the MC68881 or MC68882.

The encoding of bits 8–0 of the coprocessor instruction operation word is dependent on the particular instruction being implemented (refer to **7.2 Coprocessor Instruction Types**).

7.1.4 Coprocessor System Interface

The communication protocol between the main processor and coprocessor necessary to execute a coprocessor instruction uses a group of interface registers, CIRs, resident within the coprocessor. By accessing one of the CIRs, the MC68020/EC020 hardware initiates coprocessor instructions. The coprocessor uses a set of response primitive codes and format codes defined for the M68000 coprocessor interface to communicate status and service requests to the main processor through these registers. The CIRs are also used to pass operands between the main processor and the coprocessor. The CIR set, response primitives, and format codes are discussed in **7.3 Coprocessor Interface Register Set** and **7.4 Coprocessor Response Primitives**.

7.1.4.1 COPROCESSOR CLASSIFICATION. M68000 coprocessors can be classified into two categories depending on their bus interface capabilities. The first category, non-DMA coprocessors, consists of coprocessors that always operate as bus slaves. The second category, DMA coprocessors, consists of coprocessors that operate as bus slaves while communicating with the main processor across the coprocessor interface. These coprocessors also have the ability to operate as bus masters, directly controlling the system bus.

If the operation of a coprocessor does not require a large portion of the available bus bandwidth or has special requirements not directly satisfied by the main processor, that coprocessor can be efficiently implemented as a non-DMA coprocessor. Since non-DMA coprocessors always operate as bus slaves, all external bus-related functions that the coprocessor requires are performed by the main processor. The main processor transfers operands from the coprocessor by reading the operand from the appropriate CIR and then writing the operand to a specified effective address with the appropriate address space specified on the FC2–FC0. Likewise, the main processor transfers operands to the coprocessor by reading the operand from a specified effective address (and address space) and then writing that operand to the appropriate CIR using the coprocessor interface. The bus interface circuitry of a coprocessor operating as a bus slave is not as complex as that of a device operating as a bus master.

information to the main processor during the execution of these instructions. These coprocessor format codes are discussed in detail in **7.2.3.2 Coprocessor Format Words**.

7.2.3.1 COPROCESSOR INTERNAL STATE FRAMES. The context save (cpSAVE) and context restore (cpRESTORE) instructions transfer an internal coprocessor state frame between memory and a coprocessor. This internal coprocessor state frame represents the state of coprocessor operations. Using the cpSAVE and cpRESTORE instructions, it is possible to interrupt coprocessor operation, save the context associated with the current operation, and initiate coprocessor operations with a new context.

A cpSAVE instruction stores a coprocessor internal state frame as a sequence of long-word entries in memory. Figure 7-14 shows the format of a coprocessor state frame. The format and length fields of the coprocessor state frame format comprise the format word. During execution of the cpSAVE instruction, the MC68020/EC020 calculates the state frame effective address from information in the operation word of the instruction and stores a format word at this effective address. The processor writes the long words that form the coprocessor state frame to descending memory addresses, beginning with the address specified by the sum of the effective address and the length field multiplied by four. During execution of the cpRESTORE instruction, the MC68020/EC020 reads the state frame from ascending addresses beginning with the effective address specified in the instruction operation word.

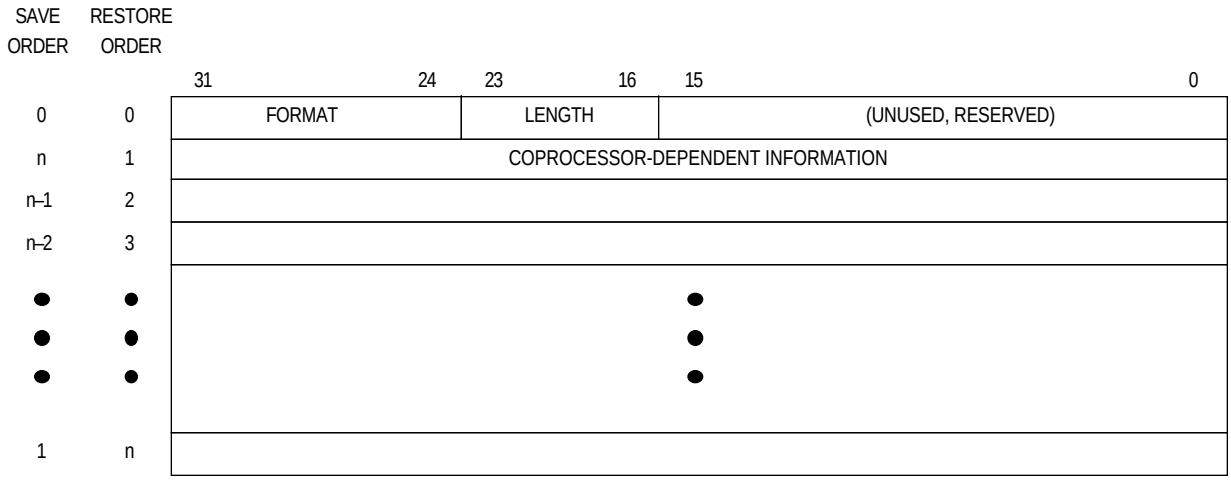


Figure 7-14. Coprocessor State Frame Format in Memory

The processor stores the coprocessor format word at the lowest address of the state frame in memory, and this word is the first word transferred for both the cpSAVE and cpRESTORE instructions. The word following the format word does not contain information relevant to the coprocessor state frame, but serves to keep the information in the state frame a multiple of four bytes in size. The number of entries following the format word (at higher addresses) is determined by the format word length for a given coprocessor state.

Figure 7-19. Control CIR Format

When the MC68020/EC020 receives one of the three take exception coprocessor response primitives, it acknowledges the primitive by setting the exception acknowledge bit (XA) in the control CIR. The MC68020/EC020 sets the abort bit (AB) in the control CIR to abort any coprocessor instruction in progress. (The 14 most significant bits of both masks are undefined.) The MC68020/EC020 aborts a coprocessor instruction when it detects one of the following exception conditions:

- An F-line emulator exception condition after reading a response primitive
- A privilege violation exception as it performs a supervisor check in response to a supervisor check primitive
- A format error exception when it receives an invalid format word or a valid format word that contains an invalid length

7.3.3 Save CIR

The coprocessor uses the 16-bit save CIR to communicate status and state frame format information to the main processor while executing a cpSAVE instruction. The main processor reads the save CIR to initiate execution of the cpSAVE instruction by the coprocessor. The offset from the base address of the CIR set for the save CIR is $04. Refer to **7.2.3.2 Coprocessor Format Words** for more information on the save CIR.

7.3.4 Restore CIR

The main processor initiates the cpRESTORE instruction by writing a coprocessor format word to the 16-bit restore register. During the execution of the cpRESTORE instruction, the coprocessor communicates status and state frame format information to the main processor through the restore CIR. The offset from the base address of the CIR set for the restore CIR is $06. Refer to **7.2.3.2 Coprocessor Format Words** for more information on the restore CIR.

7.3.5 Operation Word CIR

The main processor writes the F-line operation word of the instruction in progress to the 16-bit operation word CIR in response to a transfer operation word coprocessor response primitive (refer to **7.4.6 Transfer Operation Word Primitive**). The offset from the base address of the CIR set for the operation word CIR is $08.

7.3.6 Command CIR

The main processor initiates a coprocessor general category instruction by writing the instruction command word, which follows the instruction F-line operation word in the instruction stream, to the 16-bit command CIR. The offset from the base address of the CIR set for the command CIR is $0A.

7.4.3 Busy Primitive

The busy response primitive causes the main processor to reinitiate a coprocessor instruction. This primitive applies to instructions in the general and conditional categories. Figure 7-23 shows the format of the busy primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	PC	1	0	0	1	0	0	0	0	0	0	0	0	0	0

Figure 7-23. Busy Primitive Format

The busy primitive uses the PC bit as described in **7.4.2 Coprocessor Response Primitive General Format**.

Coprocessors that can operate concurrently with the main processor but cannot buffer write operations to their command or condition CIR use the busy primitive. A coprocessor may execute a cpGEN instruction concurrently with an instruction in the main processor. If the main processor attempts to initiate an instruction in the general or conditional instruction category while the coprocessor is executing a cpGEN instruction, the coprocessor can place the busy primitive in the response CIR. When the main processor reads this primitive, it services pending interrupts using a preinstruction exception stack frame (refer to Figure 7-41). The processor then restarts the general or conditional coprocessor instruction that it had attempted to initiate earlier.

The busy primitive should only be used in response to a write to the command or condition CIR. It should be the first primitive returned after the main processor attempts to initiate a general or conditional category instruction. In particular, the busy primitive should not be issued after program-visible resources have been altered by the instruction. (Program-visible resources include coprocessor and main processor program-visible registers and operands in memory, but not the scanPC.) The restart of an instruction after it has altered program-visible resources causes those resources to have inconsistent values when the processor reinitiates the instruction.

The MC68020/EC020 responds to the busy primitive differently in a special case that can occur during a breakpoint operation (refer to **Section 6 Exception Processing**). This special case occurs when a breakpoint acknowledge cycle initiates a coprocessor F-line instruction, the coprocessor returns the busy primitive in response to the instruction initiation, and an interrupt is pending. When these three conditions are met, the processor reexecutes the breakpoint acknowledge cycle after completion of interrupt exception processing. A design that uses a breakpoint to monitor the number of passes through a loop by incrementing or decrementing a counter may not work correctly under these conditions. This special case may cause several breakpoint acknowledge cycles to be executed during a single pass through a loop.

After reading a valid code from the register select CIR, if DR = 0, the main processor writes the long-word operand from the specified control register to the operand CIR. If DR = 1, the main processor reads a long-word operand from the operand CIR and places it in the specified control register.

7.4.15 Transfer Multiple Main Processor Registers Primitive

The transfer multiple main processor registers primitive transfers long-word operands between one or more of its data or address registers and the coprocessor. This primitive applies to general and conditional category instructions. Figure 7-35 shows the format of the transfer multiple main processor registers primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	0	1	1	0	0	0	0	0	0	0	0	0

Figure 7-35. Transfer Multiple Main Processor Registers Primitive Format

The transfer multiple main processor registers primitive uses the CA, PC, and DR bits as described in **7.4.2 Coprocessor Response Primitive General Format**. If the coprocessor issues this primitive with CA = 0 during a conditional category instruction, the main processor initiates protocol violation exception processing.

When the main processor receives this primitive, it reads a 16-bit register select mask from the register select CIR. The format of the register select mask is shown in Figure 7-36. A register is transferred if the bit corresponding to the register in the register select mask is set. The selected registers are transferred in the order D7–D0 and then A7–A0.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

Figure 7-36. Register Select Mask Format

If DR = 0, the main processor writes the contents of each register indicated in the register select mask to the operand CIR using a sequence of long-word transfers. If DR = 1, the main processor reads a long-word operand from the operand CIR into each register indicated in the register select mask. The registers are transferred in the same order, regardless of the direction of transfer indicated by the DR bit.

7.4.16 Transfer Multiple Coprocessor Registers Primitive

The transfer multiple coprocessor registers primitive transfers from 0–16 operands between the effective address specified in the coprocessor instruction and the coprocessor. This primitive applies to general category instructions. If the coprocessor issues this primitive during the execution of a conditional category instruction, the main processor initiates protocol violation exception processing. Figure 7-37 shows the format of the transfer multiple coprocessor registers primitive.

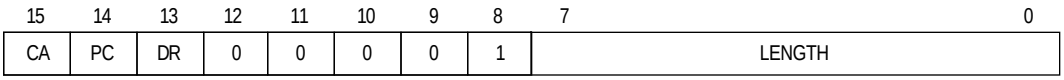


Figure 7-37. Transfer Multiple Coprocessor Registers Primitive Format

The transfer multiple coprocessor registers primitive uses the CA, PC, and DR bits as described in **7.4.2 Coprocessor Response Primitive General Format**.

The length field of the primitive format indicates the length in bytes of each operand transferred. The operand length must be an even number of bytes; odd length operands cause the MC68020/EC020 to initiate protocol violation exception processing (refer to **7.5.2.1 Protocol Violations**).

When the main processor reads this primitive, it calculates the effective address specified in the coprocessor instruction. The scanPC should be pointing to the first of any necessary effective address extension words when this primitive is read from the response CIR; the scanPC is incremented by two for each extension word referenced during the effective address calculation. For transfers from the effective address to the coprocessor (DR = 0), the control addressing modes and the postincrement addressing mode are valid. For transfers from the coprocessor to the effective address (DR = 1), the control alterable and predecrement addressing modes are valid. Invalid addressing modes cause the MC68020/EC020 to abort the instruction by writing an abort mask to the control CIR (refer to **7.3.2 Control CIR**) and to initiate F-line emulator exception processing (refer to **7.5.2.2 F-Line Emulator Exceptions**).

After performing the effective address calculation, the MC68020/EC020 reads a 16-bit register select mask from the register select CIR. The coprocessor uses the register select mask to specify the number of operands to transfer; the MC68020/EC020 counts the number of ones in the register select mask to determine the number of operands. The order of the ones in the register select mask is not relevant to the operation of the main processor. As many as 16 operands can be transferred by the main processor in response to this primitive. The total number of bytes transferred is the product of the number of operands transferred and the length of each operand specified in the length field of the primitive format.

If DR = 1, the main processor reads the number of operands specified in the register select mask from the operand CIR and writes these operands to the effective address specified in the instruction using long-word transfers whenever possible. If DR = 0, the main processor reads the number of operands specified in the register select mask from the effective address and writes them to the operand CIR.

For the control addressing modes, the operands are transferred to or from memory using ascending addresses. For the postincrement addressing mode, the operands are read from memory with ascending addresses also, and the address register used is incremented by the size of an operand after each operand is transferred. The address register used with the (An)+ addressing mode is incremented by the total number of bytes transferred during the primitive execution.

When the MC68020/EC020 detects a protocol violation, it does not automatically notify the coprocessor of the resulting exception by writing to the control CIR. However, the exception handling routine may use the MOVES instruction to read the response CIR and thus determine the primitive that caused the MC68020/EC020 to initiate protocol violation exception processing. The main processor initiates exception processing using the midinstruction stack frame (refer to Figure 7-43) and the coprocessor protocol violation exception vector number 13. If the exception handler does not modify the stack frame, the main processor reads the response CIR again following the execution of an RTE instruction to return from the exception handler. This protocol allows extensions to the M68000 coprocessor interface to be emulated in software by a main processor that does not provide hardware support for these extensions. Thus, the protocol violation is transparent to the coprocessor if the primitive execution can be emulated in software by the main processor.

7.5.2.2 F-LINE EMULATOR EXCEPTIONS. The F-line emulator exceptions detected by the MC68020/EC020 are either explicitly or implicitly related to the encodings of F-line operation words in the instruction stream. If the main processor determines that an F-line operation word is not valid, it initiates F-line emulator exception processing. Any F-line operation word with bits 8–6 = 110 or 111 causes the MC68020/EC020 to initiate exception processing without initiating any communication with the coprocessor for that instruction. Also, an operation word with bits 8–6 = 000–101 that does not map to one of the valid coprocessor instructions in the instruction set causes the MC68020/EC020 to initiate F-line emulator exception processing. If the F-line emulator exception is either of these two situations, the main processor does not write to the control CIR prior to initiating exception processing.

F-line exceptions can also occur if the operations requested by a coprocessor response primitive are not compatible with the effective address type in bits 5–0 of the coprocessor instruction operation word. The F-line emulator exceptions that can result from the use of the M68000 coprocessor response primitives are summarized in Table 7-6. If the exception is caused by receiving an invalid primitive, the main processor aborts the coprocessor instruction in progress by writing an abort mask (refer to **7.3.2 Control CIR**) to the control CIR prior to F-line emulator exception processing.

Another type of F-line emulator exception occurs when a bus error occurs during the CIR access that initiates a coprocessor instruction. The main processor assumes that the coprocessor is not present and takes the exception.

When the main processor initiates F-line emulator exception processing, it uses the four-word preinstruction exception stack frame (refer to Figure 7-41) and the F-line emulator exception vector number 11. Thus, if the exception handler does not modify the stack frame, the main processor attempts to restart the instruction that caused the exception after it executes an RTE instruction to return from the exception handler.

If the cause of the F-line exception can be emulated in software, the handler stores the results of the emulation in the appropriate registers of the programming model and in the status register field of the saved stack frame. The exception handler adjusts the program

CACHE CASE

Source Address Mode	Destination							
	An	Dn	(An)	(An)+	–(An)	(d_{16},An)	(xxx).W	(xxx).L
Rn	**2**(0/0/0)	**2**(0/0/0)	**4**(0/0/1)	**4**(0/0/1)	**5**(0/0/1)	**5**(0/0/1)	**4**(0/0/1)	**6**(0/0/1)
#<data>.B,W	**4**(0/0/0)	**4**(0/0/0)	**6**(0/0/1)	**6**(0/0/1)	**7**(0/0/1)	**7**(0/0/1)	**6**(0/0/1)	**8**(0/0/1)
#<data>.L	**6**(0/0/0)	**6**(0/0/0)	**8**(0/0/1)	**8**(0/0/1)	**9**(0/0/1)	**9**(0/0/1)	**8**(0/0/1)	**10**(0/0/1)
(An)	**6**(1/0/0)	**6**(1/0/0)	**7**(1/0/1)	**7**(1/0/1)	**7**(1/0/1)	**7**(1/0/1)	**7**(1/0/1)	**9**(1/0/1)
(An)+	**6**(1/0/0)	**6**(1/0/0)	**7**(1/0/1)	7(1/0/1)	**7**(1/0/1)	**7**(1/0/1)	**7**(1/0/1)	**9**(1/0/1)
–(An)	**7**(1/0/0)	**7**(1/0/0)	**8**(1/0/1)	**8**(1/0/1)	**8**(1/0/1)	**8**(1/0/1)	**8**(1/0/1)	**10**(1/0/1)
(d_{16},An) or (d_{16},PC)	**7**(1/0/0)	**7**(1/0/0)	**8**(1/0/1)	**8**(1/0/1)	**8**(1/0/1)	**8**(1/0/1)	**8**(1/0/1)	**10**(1/0/1)
(xxx).W	**6**(1/0/0)	**6**(1/0/0)	**7**(1/0/1)	**7**(1/0/1)	**7**(1/0/1)	**7**(1/0/1)	**7**(1/0/1)	**9**(1/0/1)
(xxx).L	**6**(1/0/0)	**6**(1/0/0)	**7**(1/0/1)	**7**(1/0/1)	**7**(1/0/1)	**7**(1/0/1)	**7**(1/0/1)	**9**(1/0/1)
(d_8,An,Xn) or (d_8,PC,Xn)	**9**(1/0/0)	**9**(1/0/0)	**10**(1/0/1)	**10**(1/0/1)	**10**(1/0/1)	**10**(1/0/1)	**10**(1/0/1)	**12**(1/0/1)
(d_{16},An,Xn) or (d_{16},PC,Xn)	**9**(1/0/0)	**9**(1/0/0)	**10**(1/0/1)	**10**(1/0/1)	**10**(1/0/1)	**10**(1/0/1)	**10**(1/0/1)	**12**(1/0/1)
(B)	**9**(1/0/0)	**9**(1/0/0)	**10**(1/0/1)	**10**(1/0/1)	**10**(1/0/1)	**10**(1/0/1)	**10**(1/0/1)	**12**(1/0/1)
(d_{16},B)	**11**(1/0/0)	**11**(1/0/0)	**12**(1/0/1)	**12**(1/0/1)	**12**(1/0/1)	**12**(1/0/1)	**12**(1/0/1)	**14**(1/0/1)
(d_{32},B)	**15**(1/0/0)	**15**(1/0/0)	**16**(1/0/1)	**16**(1/0/1)	**16**(1/0/1)	**16**(1/0/1)	**16**(1/0/1)	**18**(1/0/1)
([B],I)	**14**(2/0/0)	**14**(2/0/0)	**15**(2/0/1)	**15**(2/0/1)	**15**(2/0/1)	**15**(2/0/1)	**15**(2/0/1)	**17**(2/0/1)
([B],I,d_{16})	**16**(2/0/0)	**16**(2/0/0)	**17**(2/0/1)	**17**(2/0/1)	**17**(2/0/1)	**17**(2/0/1)	**17**(2/0/1)	**19**(2/0/1)
([B],I,d_{32})	**16**(2/0/0)	**16**(2/0/0)	**17**(2/0/1)	**17**(2/0/1)	**17**(2/0/1)	**17**(2/0/1)	**17**(2/0/1)	**19**(2/0/1)
([d_{16},B],I)	**16**(2/0/0)	**16**(2/0/0)	**17**(2/0/1)	**17**(2/0/1)	**17**(2/0/1)	**17**(2/0/1)	**17**(2/0/1)	**19**(2/0/1)
([d_{16},B],I,d_{16})	**18**(2/0/0)	**18**(2/0/0)	**19**(2/0/1)	**19**(2/0/1)	**19**(2/0/1)	**19**(2/0/1)	**19**(2/0/1)	**21**(2/0/1)
([d_{16},B],d_{32})	**18**(2/0/0)	**18**(2/0/0)	**19**(2/0/1)	**19**(2/0/1)	**19**(2/0/1)	**19**(2/0/1)	**19**(2/0/1)	**21**(2/0/1)
([d_{32},B],I)	**20**(2/0/0)	**20**(2/0/0)	**21**(2/0/1)	**21**(2/0/1)	**21**(2/0/1)	**21**(2/0/1)	**21**(2/0/1)	**23**(2/0/1)
([d_{32},B],I,d_{16})	**22**(2/0/0)	**22**(2/0/0)	**23**(2/0/1)	**23**(2/0/1)	**23**(2/0/1)	**23**(2/0/1)	**23**(2/0/1)	**25**(2/0/1)
([d_{32},B],I,d_{32})	**22**(2/0/0)	**22**(2/0/0)	**23**(2/0/1)	**23**(2/0/1)	**23**(2/0/1)	**23**(2/0/1)	**23**(2/0/1)	**25**(2/0/1)

8.2.8 Arithmetic/Logical Instructions

The arithmetic/logical instructions table indicates the number of clock periods needed for the processor to perform the specified arithmetic/logical operation using the specified addressing mode. It also includes, in worst case, the amount of time needed to prefetch the next instruction. Footnotes specify when to add either fetch address or fetch immediate effective address time. This sum gives the total effective execution time for the operation using the specified addressing mode. The total number of clock cycles is outside the parentheses; the number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). These cycles are included in the total clock cycle number.

Instruction			Best Case	Cache Case	Worst Case
*	ADD	EA,Dn	**0**(0/0/0)	**2**(0/0/0)	**3**(0/1/0)
*	ADDA	EA,An	**0**(0/0/0)	**2**(0/0/0)	**3**(0/1/0)
*	ADD	Dn,EA	**3**(0/0/1)	**4**(0/0/1)	**6**(0/1/1)
*	AND	EA,Dn	**0**(0/0/0)	**2**(0/0/0)	**3**(0/1/0)
*	AND	Dn,EA	**3**(0/0/1)	**4**(0/0/1)	**6**(0/1/1)
*	EOR	Dn,Dn	**0**(0/0/0)	**2**(0/0/0)	**3**(0/1/0)
*	EOR	Dn,Mem	**3**(0/0/1)	**4**(0/0/1)	**6**(0/1/1)
*	OR	EA,Dn	**0**(0/0/0)	**2**(0/0/0)	**3**(0/1/0)
*	OR	Dn,EA	**3**(0/0/1)	**4**(0/0/1)	**6**(0/1/1)
*	SUB	EA,Dn	**0**(0/0/0)	**2**(0/0/0)	**3**(0/1/0)
*	SUBA	EA,An	**0**(0/0/0)	**2**(0/0/0)	**3**(0/1/0)
*	SUB	Dn,EA	**3**(0/0/1)	**4**(0/0/1)	**6**(0/1/1)
*	CMP	EA,Dn	**0**(0/0/0)	**2**(0/0/0)	**3**(0/1/0)
*	CMPA	EA,An	**1**(0/0/0)	**4**(0/0/0)	**4**(0/1/0)
**	CMP2	EA,Rn	**16**(1/0/0)	**18**(1/0/0)	**18**(1/1/0)
*	MUL.W	EA,Dn	**25**(0/0/0)	**27**(0/0/0)	**28**(0/1/0)
**	MUL.L	EA,Dn	**41**(0/0/0)	**43**(0/0/0)	**44**(0/1/0)
*	DIVU.W	EA,Dn	**42**(0/0/0)	**44**(0/0/0)	**44**(0/1/0)
**	DIVU.L	EA,Dn	**76**(0/0/0)	**78**(0/0/0)	**79**(0/1/0)
*	DIVS.W	EA,Dn	**54**(0/0/0)	**56**(0/0/0)	**57**(0/1/0)
**	DIVS.L	EA,Dn	**88**(0/0/0)	**90**(0/0/0)	**91**(0/1/0)

*Add Fetch Effective Address Time

**Add Fetch Immediate Address Time

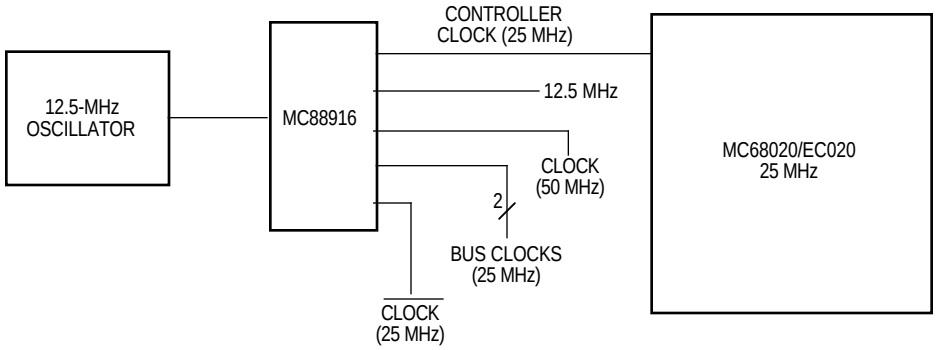


Figure 9-7. High-Resolution Clock Controller

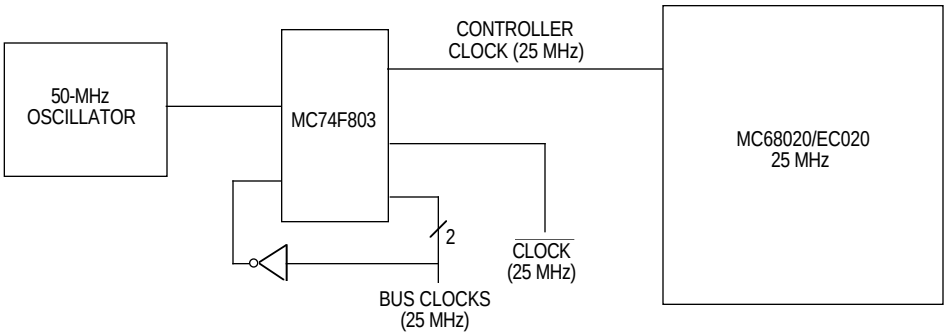


Figure 9-8. Alternate Clock Solution

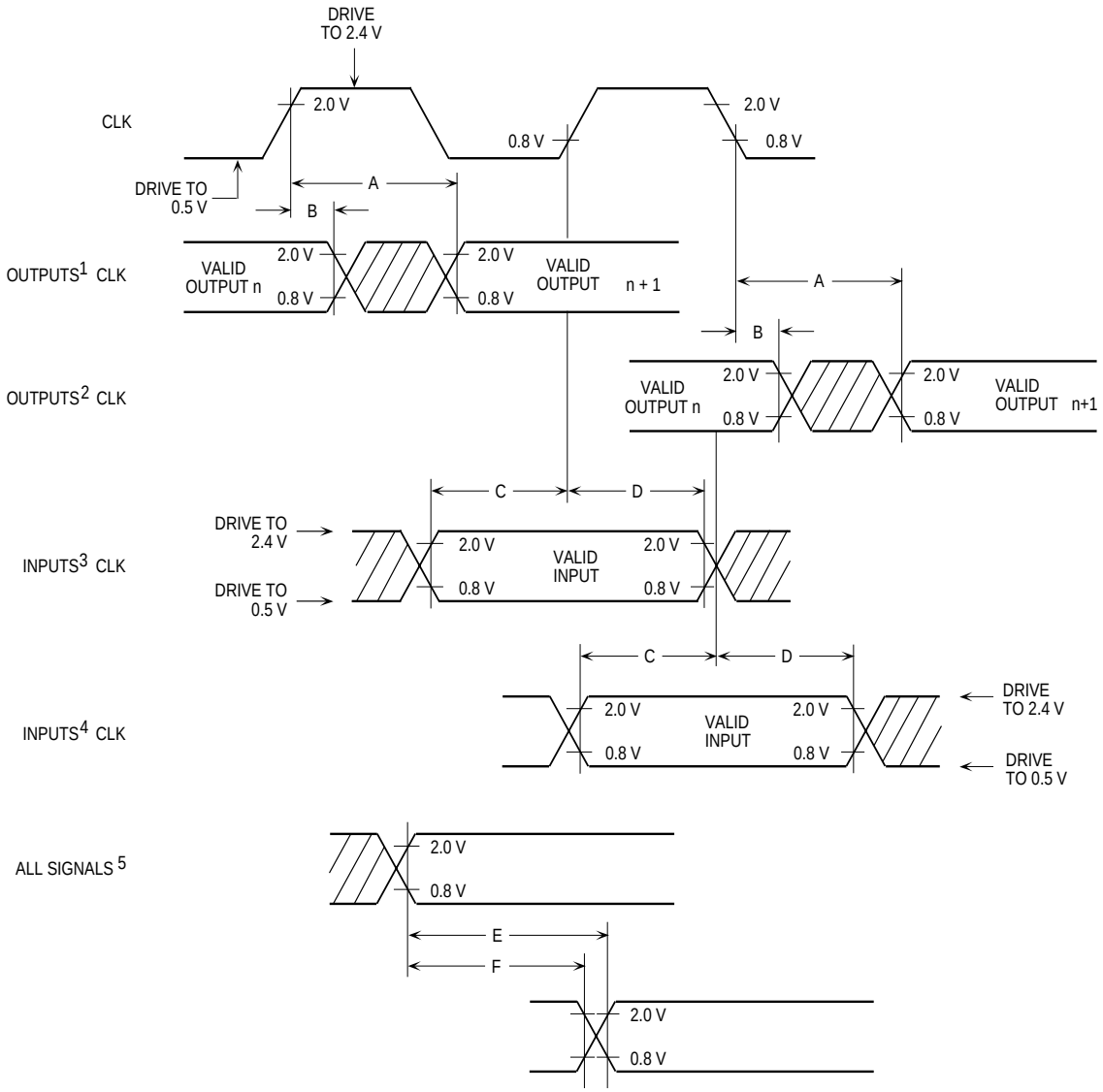
9.5 MEMORY INTERFACE

The MC68020/EC020 is capable of running an external bus cycle in a minimum of three clocks (refer to **Section 5 Bus Operation**). The MC68020/EC020 runs an asynchronous bus cycle, terminated by the DSACK1/DSACK0 signals, and has a minimum duration of three controller clock periods in which up to four bytes (32 bits) are transferred.

During read operations, the MC68020/EC020 latches data on the last falling clock edge of the bus cycle, one-half clock before the bus cycle ends. Latching data here, instead of the next rising clock edge, helps to avoid data bus contention with the next bus cycle and allows the MC68020/EC020 to receive the data into its execution unit sooner for a net performance increase.

Write operations also use this data bus timing to allow data hold times from the negating strobes and to avoid any bus contention with the following bus cycle. This MC68020/EC020 characteristic allows the system to be designed with a minimum of bus buffers and latches.

One benefit of the MC68020/EC020 on-chip instruction cache is that the effect of external wait states on performance is lessened because the caches are always accessed in fewer than “no wait states,” regardless of the external memory configuration.



NOTES:
1. This output timing is applicable to all parameters specified relative to the rising edge of the clock.
2. This output timing is applicable to all parameters specified relative to the falling edge of the clock.
3. This input timing is applicable to all parameters specified relative to the rising edge of the clock.
4. This input timing is applicable to all parameters specified relative to the falling edge of the clock.
5. This timing is applicable to all parameters specified relative to the assertion/negation of another signal.

LEGEND:
A. Maximum output delay specification.
B. Minimum output hold time.
C. Minimum input setup time specification.
D. Minimum input hold time specification.
E. Signal valid to signal valid specification (maximum or minimum).
F. Signal valid to signal invalid specification (maximum or minimum).

FIGURE 10-1
MC68020UM

Figure 10-1. Drive Levels and Test Points for AC Specifications

INDEX