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	25MHz
Co-Processors/DSP	-
RAM Controllers	-
Graphics Acceleration	No
Display & Interface Controllers	-
Ethernet	-
SATA	-
USB	-
Voltage - I/O	5.0V
Operating Temperature	-40°C ~ 85°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/mc68020ceh25e

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

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

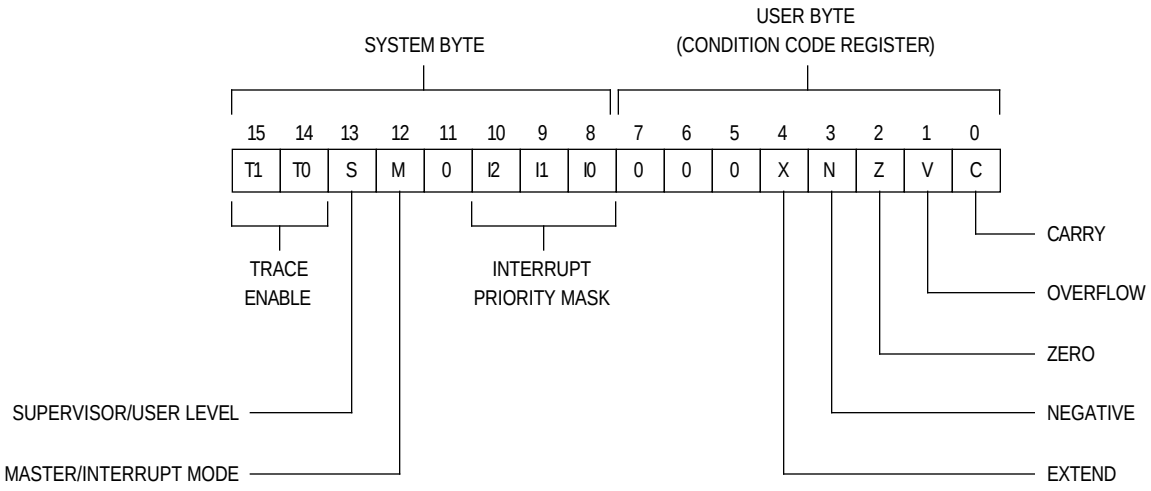


Figure 1-4. Status Register (SR)

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.

The alternate function code registers, SFC and DFC, contain 3-bit function codes. For the MC68020, function codes can be considered extensions of the 32-bit linear address that optionally provide as many as eight 4-Gbyte address spaces; for the MC68EC020, function codes can be considered extensions of the 24-bit linear address that optionally provide as many as eight 16-Mbyte address spaces. Function codes are automatically generated by the processor to select address spaces for data and program at the user and supervisor privilege levels and to select a CPU address space for processor functions (e.g., coprocessor communications). Registers SFC and DFC are used by certain instructions to explicitly specify the function codes for operations.

The CACR controls the on-chip instruction cache of the MC68020/EC020. The CAAR stores an address for cache control functions.

2.1 PRIVILEGE LEVELS

The processor operates at one of two privilege levels: the user level or the supervisor level. The supervisor level has higher privileges than the user level. Not all processor or coprocessor instructions are permitted to execute at the lower privileged user level, but all are available at the supervisor level. This arrangement allows a separation of supervisor and user so the supervisor can protect system resources from uncontrolled access. The S-bit in the SR is used to select either the user or supervisor privilege level and either the USP or an SSP for stack operations. The processor identifies a bus access (supervisor or user mode) via the function codes so that differentiation between supervisor level and user level can be maintained.

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

2.1.1 Supervisor Privilege Level

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

In a multitasking operating system, it is more efficient to have a supervisor stack space associated with each user task and a separate stack space for interrupt-associated tasks. The MC68020/EC020 provides two supervisor stacks, master and interrupt; the M bit of the SR selects which of the two is active. When the M-bit is set, references to the SSP implicitly or to address register seven (A7) explicitly, access the MSP. The operating system sets the MSP for each task to point to a task-related area of supervisor data space. This arrangement 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 other supervisor stack pointer, the ISP, can be used for interrupt control information and workspace area as interrupt handling routines require.

When the M-bit is clear, the MC68020/EC020 is in the interrupt mode of the supervisor privilege level, and operation is the same as supervisor mode in the MC68000, MC68HC001, MC68008, and MC68010. (The processor is in this mode after a reset operation.) All SSP references access the ISP in this mode.

2.3.2 Exception Stack Frame

Exception processing saves the most volatile portion of the current processor context on the top of the supervisor stack. This context is organized in a format called the exception stack frame. This information always includes a copy of the 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 format field to properly restore the information stored in the stack frame and to deallocate the stack space. The general form of the exception stack frame is illustrated in Figure 2-1. Refer to **Section 6 Exception Processing** for a complete list of exception stack frames.

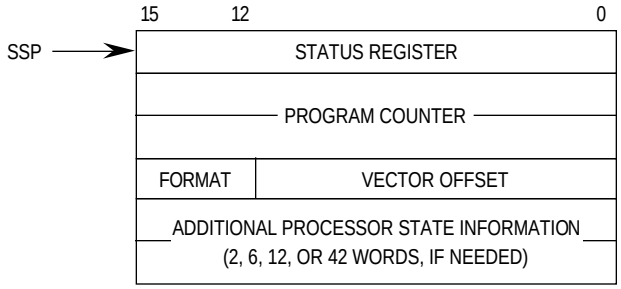


Figure 2-1. General Exception Stack Frame

4.2 CACHE RESET

During processor reset, the cache is cleared by resetting all of the valid bits. The E and F bits in the CACR are also cleared.

4.3 CACHE CONTROL

Only the MC68020/EC020 cache control circuitry can directly access the cache array, but a supervisor program can set bits in the CACR to exercise control over cache operations. The supervisor level also has access to the CAAR, which contains the address for a cache entry to be cleared.

System hardware can assert the CDIS signal to disable the cache. The assertion of CDIS disables the cache, regardless of the state of the E-bit in the CACR. CDIS is primarily intended for use by in-circuit emulators.

4.3.1 Cache Control Register (CACR)

The CACR, shown in Figure 4-2, is a 32-bit register than can be written or read by the MOVEC instruction or indirectly modified by a reset. Four of the bits (3–0) control the instruction cache. Bits 31–4 are reserved for Motorola definition. They are read as zeros and are ignored when written. For future compatibility, writes should not set these bits.

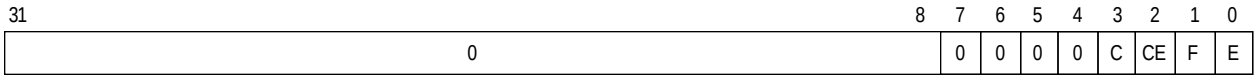


Figure 4-2. Cache Control Register

C—Clear Cache

The C-bit is set to clear all entries in the instruction cache. Operating systems and other software set this bit to clear instructions from the cache prior to a context switch. The processor clears all valid bits in the instruction cache when a MOVEC instruction sets the C-bit. The C-bit is always read as a zero.

CE—Clear Entry In Cache

The CE bit is set to clear an entry in the instruction cache. The index field of the CAAR (see Figure 4-3), corresponding to the index and long-word select portion of an address, specifies the entry to be cleared. The processor clears only the specified long word by clearing the valid bit for the entry when a MOVEC instruction sets the CE bit, regardless of the states of the E and F bits. The CE bit is always read as a zero.

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.

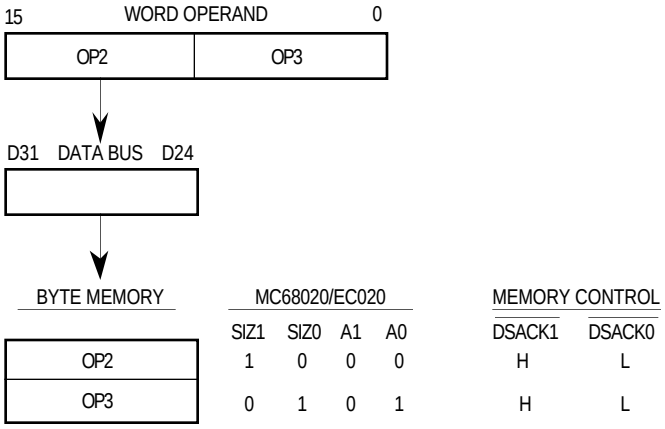


Figure 5-7. Word Operand Write to Byte Port Example

State 0

MC68020—The write cycle starts in S0. The processor negates $\overline{\text{ECS}}$, indicating the beginning of an external cycle. If the cycle is the first external cycle of a write operation, $\overline{\text{OCS}}$ is asserted simultaneously. During S0, the processor places a valid address on A31–A0 and valid function codes on FC2–FC0. The function codes select the address space for the cycle. The processor drives R/$\overline{\text{W}}$ low for a write cycle. SIZ1–SIZ0 become valid, indicating the number of bytes to be transferred.

MC68EC020—The write cycle starts in S0. During S0, the processor places a valid address on A23–A0 and valid function codes on FC2–FC0. The function codes select the address space for the cycle. The processor drives R/$\overline{\text{W}}$ low for a write cycle. SIZ1, SIZ0 become valid, indicating the number of bytes to be transferred.

State 1

MC68020—One-half clock later in S1, the processor asserts $\overline{\text{AS}}$, indicating that the address on the address bus is valid. The processor also asserts $\overline{\text{DBEN}}$ during S1, which can enable external data buffers. In addition, the $\overline{\text{ECS}}$ (and $\overline{\text{OCS}}$, if asserted) signal is negated during S1.

MC68EC020—One-half clock later in S1, the processor asserts $\overline{\text{AS}}$, indicating that the address on the address bus is valid.

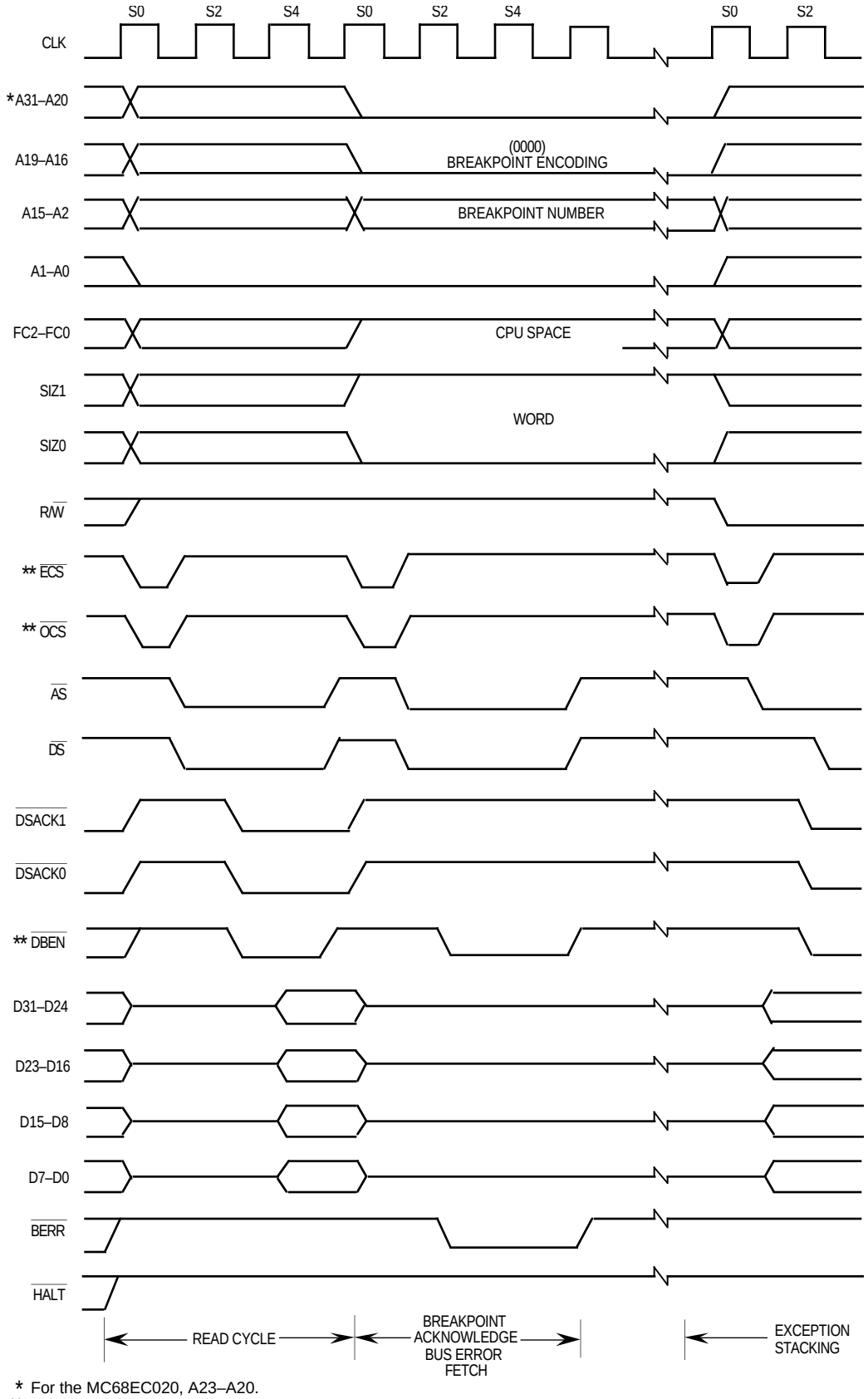
State 2

MC68020/EC020—During S2, the processor places the data to be written onto D31–D0. At the end of S2, the processor samples $\overline{\text{DSACK1}}$/$\overline{\text{DSACK0}}$.

State 3

MC68020/EC020—The processor asserts $\overline{\text{DS}}$ during S3, indicating that the data on the data bus is stable. As long as at least one of the $\overline{\text{DSACK1}}$/$\overline{\text{DSACK0}}$ signals is recognized by the end of S2 (meeting the asynchronous input setup time requirement), the cycle terminates one clock later. If $\overline{\text{DSACK1}}$/$\overline{\text{DSACK0}}$ is not recognized by the start of S3, the processor inserts wait states instead of proceeding to S4 and S5. To ensure that wait states are inserted, both $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$ must remain negated throughout the asynchronous input setup and hold times around the end of S2. If wait states are added, the processor continues to sample the $\overline{\text{DSACK1}}$/$\overline{\text{DSACK0}}$ signals on the falling edges of the clock until one is recognized.

The external device uses R/$\overline{\text{W}}$, $\overline{\text{DS}}$, SIZ1, SIZ0, A1, and A0 to latch data from the appropriate byte(s) of the data bus (D31–D24, D23–D16, D15–D8, and D7–D0). SIZ1, SIZ0, A1, and A0 select the bytes of the data bus. If it has not already done so, the device asserts $\overline{\text{DSACK1}}$/$\overline{\text{DSACK0}}$ to signal that it has successfully stored the data.

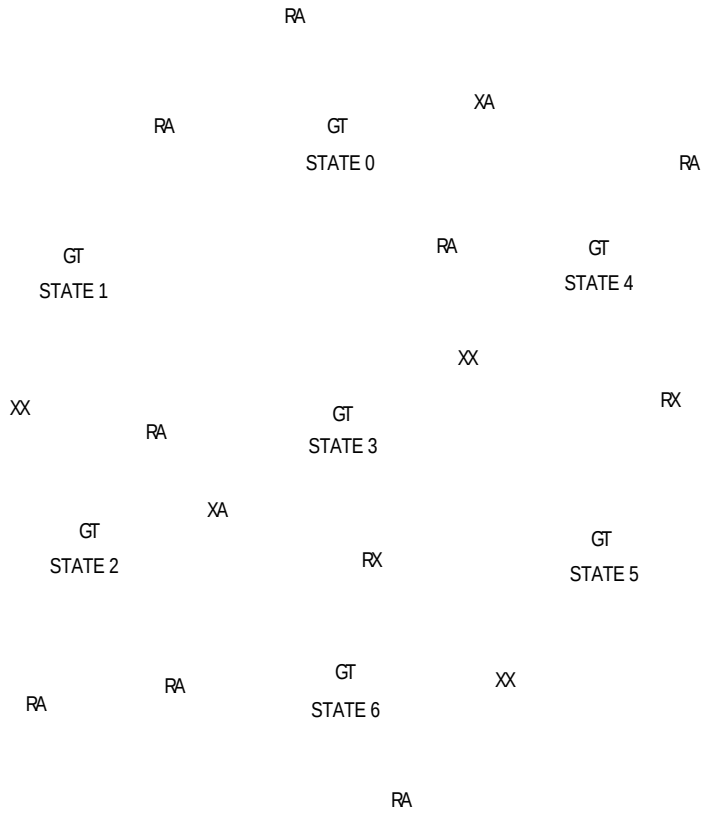


* For the MC68EC020, A23–A20.
** This signal does not apply to the MC68EC020.

Figure 5-38. Bus Error without DSACK1/DSACK0

5.7.1.4 BUS ARBITRATION CONTROL (MC68020). The bus arbitration control unit in the MC68020 is implemented with a finite state machine. As discussed previously, all asynchronous inputs to the MC68020 are internally synchronized in a maximum of two cycles of the processor clock.

As shown in Figure 5-44, input signals labeled R and A are internally synchronized versions of the $\overline{BR}$ and $\overline{BGACK}$ signals, respectively. The $\overline{BG}$ output is labeled G, and the internal high-impedance control signal is labeled T. If T is true, the address, data, and control buses are placed in the high-impedance state after the next rising edge following the negation of $\overline{AS}$ and $\overline{RMC}$. All signals are shown in positive logic (active high), regardless of their true active voltage level.



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

NOTE: The BG output will not be asserted while RMC is asserted.

Figure 5-44. MC68020 Bus Arbitration State Diagram

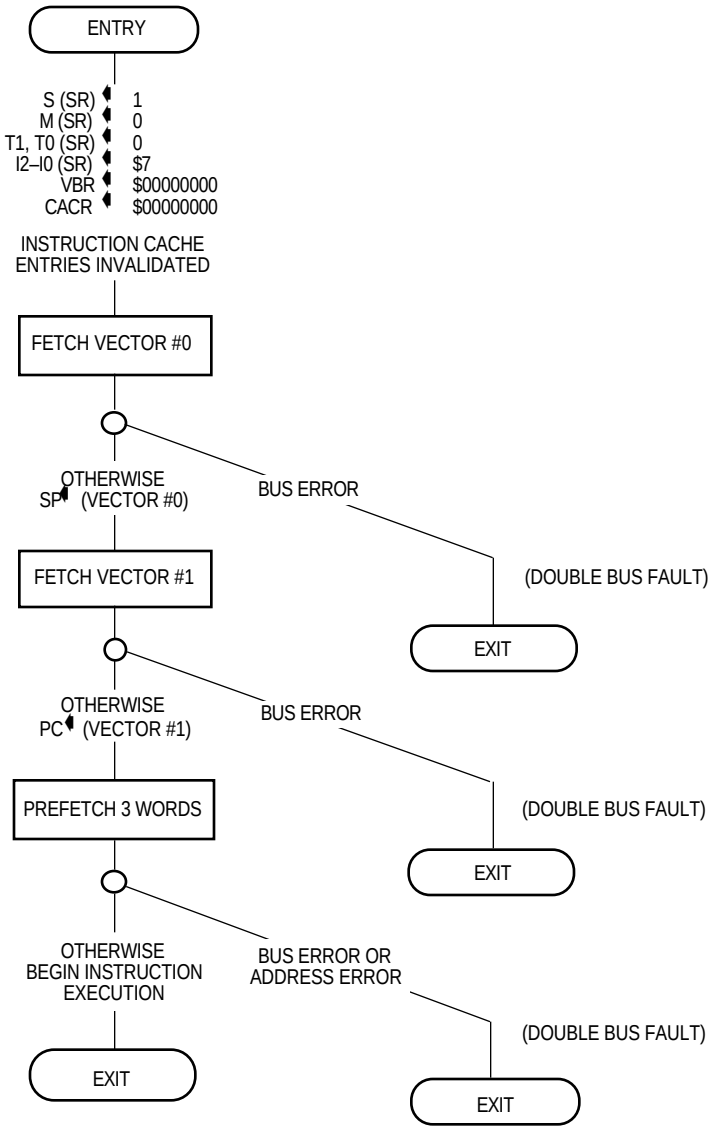


Figure 6-1. Reset Operation Flowchart

The processor begins exception processing for a bus error by making an internal copy of the current SR. The processor then enters the supervisor privilege level (by setting the S-bit in the SR) and clears the T1 and T0 bits in the SR. 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 active supervisor 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

level 6 interrupt is not processed. However, if the MC68020/EC020 is handling a level 7 interrupt (I2–I0 in the SR set to 111) and the external request is lowered to level 3 and then raised back to level 7, a second level 7 interrupt is processed. The second level 7 interrupt is processed because the level 7 interrupt is transition sensitive. A level 7 interrupt is also generated by a level comparison if the request level and mask level are at 7 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 6-3 for level 6 interrupt request level and mask level, this is the case for all interrupt levels.

Note that a mask value of 6 and a mask value of 7 both inhibit request levels of 1–6 from being recognized. In addition, neither masks a transition to an interrupt request level of 7. The only difference between mask values of 6 and 7 occurs when the interrupt request level is 7 and the mask value is 7. If the mask value is lowered to 6, a second level 7 interrupt is recognized.

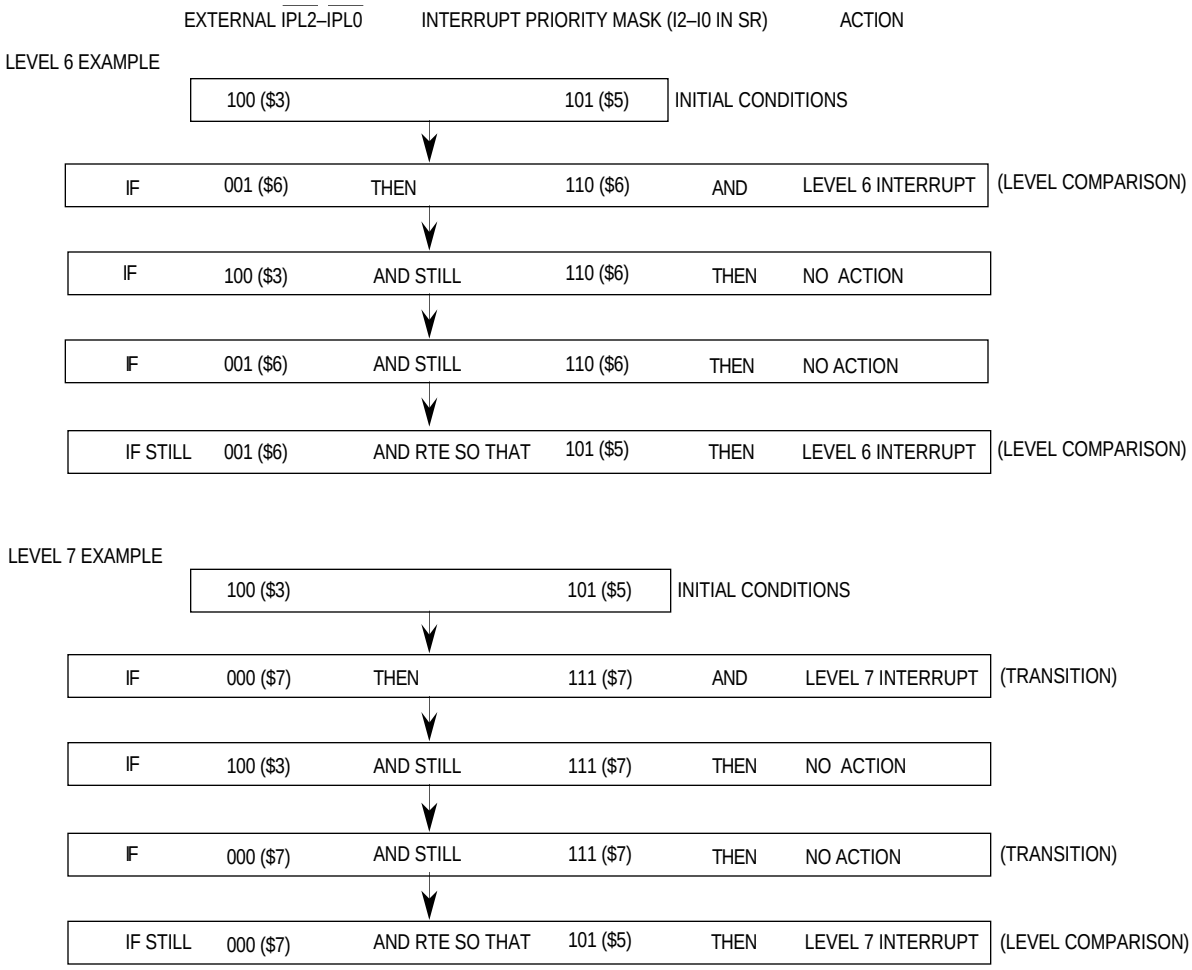


Figure 6-3. Interrupt Recognition Examples

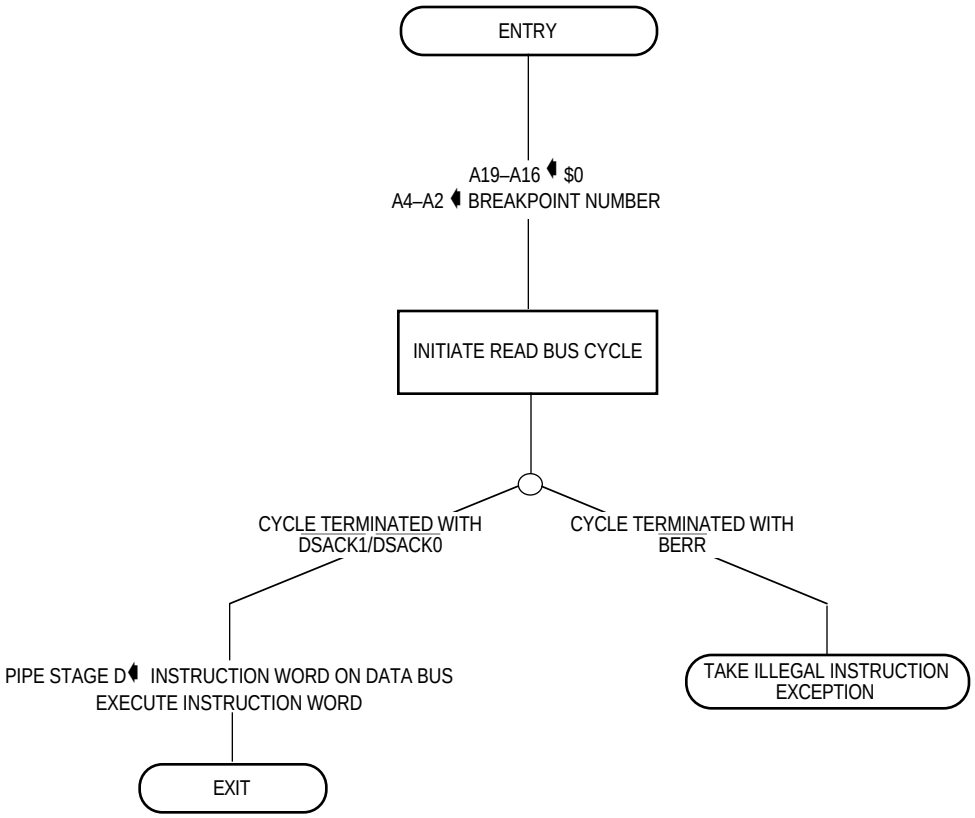


Figure 6-6. Breakpoint Instruction Flowchart

Table 6-4. Exception Priority Groups

Group/ Priority	Exception and Relative Priority	Characteristic
0	0.0—Reset	Aborts all processing (instruction or exception) and does not save old context.
1	1.0—Address Error 1.1—Bus Error	Suspends processing (instruction or exception) and saves internal context.
2	2.0—BKPT, CALLM, CHK, CHK2, cp Midinstruction, cp Protocol Violation, cpTRAPcc, Divide by Zero, RTE, RTM, TRAP, TRAPcc, TRAPV	Exception processing is part of instruction execution.
3	3.0—Illegal Instruction, Line A, Unimplemented Line F, Privilege Violation, cp Preinstruction	Exception processing begins before instruction is executed.
4	4.0—cp Postinstruction 4.1—Trace 4.2—Interrupt	Exception processing begins when current instruction or previous exception processing has completed.

NOTE: 0.0 is the highest priority; 4.2 is the lowest.

stack for stage B of the pipe are accepted as valid; the processor assumes that there is no prefetch pending for stage B and that software has repaired or filled the image of stage B, if necessary.

1 = Rerun faulted bus cycle or run pending prefetch
0 = Do not rerun bus cycle

Bits 11–9—Reserved by Motorola

DF—Fault/Rerun Flag

If the DF bit is set, a data fault has occurred and caused the exception. If the DF bit is set when the processor reads the stack frame, it reruns the faulted data access; otherwise, it assumes that the data input buffer value on the stack is valid for a read or that the data has been correctly written to memory for a write (or that no data fault occurred).

1 = Rerun faulted bus cycle or run pending prefetch
0 = Do not rerun bus cycle

RM—Read-Modify-Write

1 = Read-modify-write operation on data cycle
0 = Not a read-modify-write operation

RW—Read/Write

1 = Read on data cycle
0 = Write on data cycle

SIZE—Size Code

The SIZE field indicates the size of the operand access for the data cycle.

Bit 3—Reserved by Motorola

FC2–FC0—Specifies the address space for data cycle

6.2.2 Using Software to Complete the Bus Cycles

One method of completing a faulted bus cycle is to use a software handler to emulate the cycle. This is the only method for correcting address errors. The handler should emulate the faulted bus cycle in a manner that is transparent to the instruction that caused the fault. For instruction stream faults, the handler may need to run bus cycles for both the B and C stages of the instruction pipe. The RB and RC bits of the SSW identify the stages that may require a bus cycle; the FB and FC bits of the SSW indicate that a stage was invalid when an attempt was made to use its contents. Those stages must be repaired. For each faulted stage, the software handler should copy the instruction word from the proper address space as indicated by the S-bit of the copy of the SR saved on the stack to the image of the appropriate stage in the stack frame. In addition, the handler must clear the RB or RC bit associated with the stage that it has corrected. The handler should not change the FB and FC bits.

7.4.12 Transfer to/from Top of Stack Primitive

The transfer to/from top of stack primitive transfers an operand between the coprocessor and the top of the active system stack of the main processor. This primitive applies to general and conditional category instructions. Figure 7-32 shows the format of the transfer to/from top of stack primitive.

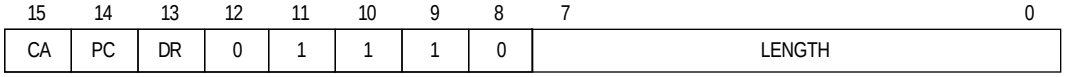


Figure 7-32. Transfer to/from Top of Stack Primitive Format

The transfer to/from top of stack 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.

The length field of the primitive format specifies the length in bytes of the operand to be transferred. The operand may be one, two, or four bytes in length; other length values cause the main processor to initiate protocol violation exception processing.

If DR = 0, the main processor transfers the operand from the active system stack to the operand CIR. The implied effective address mode used for the transfer is the (A7)+ addressing mode. A one-byte operand causes the stack pointer to be incremented by two after the transfer to maintain word alignment of the stack.

If DR = 1, the main processor transfers the operand from the operand CIR to the active system stack. The implied effective address mode used for the transfer is the –(A7) addressing mode. A one-byte operand causes the stack pointer to be decremented by two before the transfer to maintain word alignment of the stack.

7.4.13 Transfer Single Main Processor Register Primitive

The transfer single main processor register primitive transfers an operand between one of the main processor's data or address registers and the coprocessor. This primitive applies to general and conditional category instructions. Figure 7-33 shows the format of the transfer single main processor register primitive.

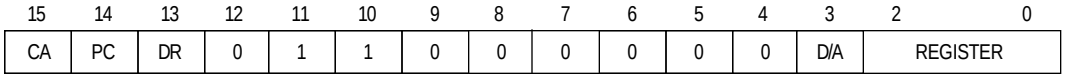


Figure 7-33. Transfer Single Main Processor Register Primitive Format

The transfer single main processor register 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.

8.1.5 Instruction Stream Timing Examples

A programming example allows a more detailed examination of these effects. The effect of instruction execution overlap on instruction timing is illustrated by the following example instruction stream.

Instruction	
#1) MOVE.L	D4,(A1)+
#2) ADD.L	D4,D5
#3) MOVE.L	(A1), –(A2)
#4) ADD.L	D5,D6

Example 1

For the first example, the assumptions are:

1. The data bus is 32 bits,
2. The first instruction is prefetched from an odd-word address,
3. Memory access occurs with no wait states, and
4. The instruction cache is disabled.

For example 1, the instruction stream is positioned in 32-bit memory as follows:

Address			
	n	•••	MOVE #1
	n + 4	ADD #2	MOVE #3
	n + 8	ADD #4	•••

Figure 8-3 shows processor activity on the first example instruction stream. It shows the activity of the external bus, the bus controller, the sequencer, and the attributed instruction execution time.

Example 3

Both Figures 8-3 and 8-4 show instruction execution without benefit of the MC68020/EC020 instruction cache. Figure 8-5 shows a third example for the same instruction stream executing in the cache. Note that once the instructions are in the cache, the original location in external memory is no longer a factor in timing.

The assumptions for Example 3 are:

1. The data bus is 32 bits,
2. The cache is enabled and instructions are in the cache, and
3. Memory access occurs with no wait states.

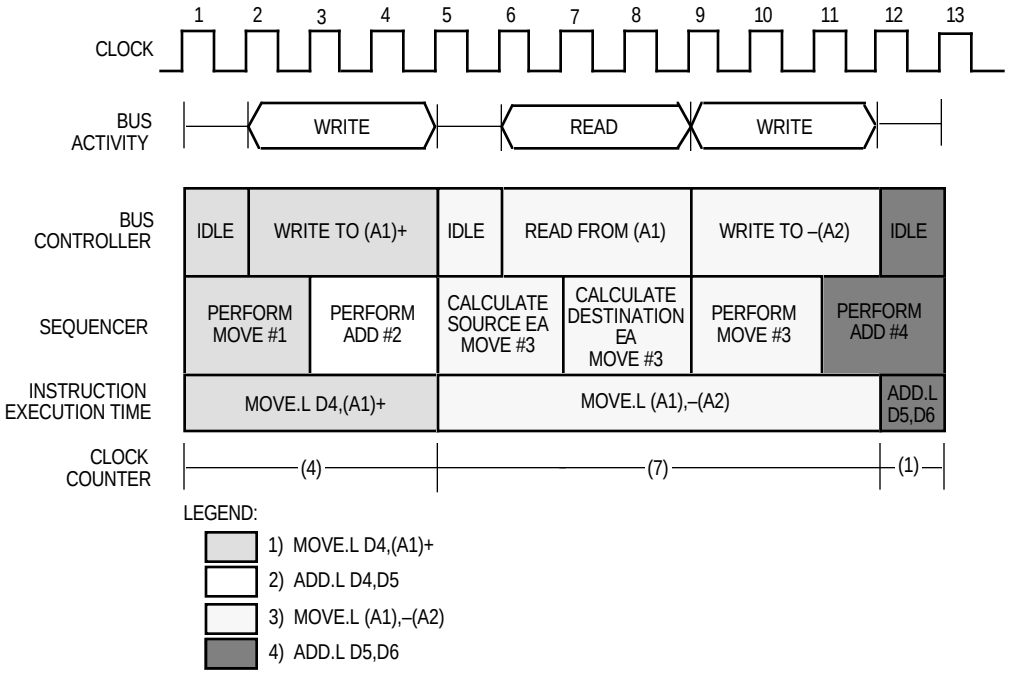


Figure 8-5. Processor Activity for Example 3

Figure 8-5 illustrates the benefits of the instruction cache. The total number of clock cycles is reduced from 16 to 12 clocks. Since the instructions are resident in the cache, the instruction prefetch activity does not require the bus controller to perform external bus cycles. Since prefetch occurs with no delay, the bus controller is idle more often.

Example 4

Idle clock cycles, such as those shown in example 3, are useful in MC68020/EC020 systems that require wait states when accessing external memory. This fact is illustrated in example 4 (see Figure 8-6) with the following assumptions:

1. The data bus is 32 bits,
2. The cache is enabled and instructions are in the cache, and
3. Memory access occurs with one wait state.

8.2.4 Calculate Immediate Effective Address

The calculate immediate effective address table indicates the number of clock periods needed for the processor to fetch the immediate source operand and calculate the specified destination effective address. Fetch time is only included for the first level of indirection on memory indirect addressing modes. 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.

Address Mode	Best Case	Cache Case	Worst Case
#<data>.W,Dn	**0**(0/0/0)	**2**(0/0/0)	**3**(0/1/0)
#<data>.L,Dn	**1**(0/0/0)	**4**(0/0/0)	**5**(0/1/0)
#<data>.W,(An)	**0**(0/0/0)	**2**(0/0/0)	**3**(0/1/0)
#<data>.L,(An)	**1**(0/0/0)	**4**(0/0/0)	**5**(0/1/0)
#<data>.W,(An)+	**2**(0/0/0)	**4**(0/0/0)	**5**(0/1/0)
#<data>.L,(An)+	**3**(0/0/0)	**6**(0/0/0)	**7**(0/1/0)
#<data>.W,(bd,An)	**1**(0/0/0)	**4**(0/0/0)	**5**(0/1/0)
#<data>.L,(bd,An)	**3**(0/0/0)	**6**(0/0/0)	**8**(0/2/0)
#<data>.W,xxx.W	**1**(0/0/0)	**4**(0/0/0)	**5**(0/1/0)
#<data>.L,xxx.W	**3**(0/0/0)	**6**(0/0/0)	**8**(0/2/0)
#<data>.W,xxx.L	**2**(0/0/0)	**4**(0/0/0)	**6**(0/2/0)
#<data>.L,xxx.L	**3**(0/0/0)	**8**(0/0/0)	**10**(0/2/0)
#<data>.W,(d_8,An,Xn) or (d_8,PC,Xn)	**0**(0/0/0)	**6**(0/0/0)	**8**(0/2/0)
#<data>.L,(d_8,An,Xn) or (d_8,PC,Xn)	**2**(0/0/0)	**8**(0/0/0)	**10**(0/2/0)
#<data>.W,(d_{16},An,Xn) or (d_{16},PC,Xn)	**3**(0/0/0)	**8**(0/0/0)	**10**(0/2/0)
#<data>.L,(d_{16},An,Xn) or (d_{16},PC,Xn)	**4**(0/0/0)	**10**(0/0/0)	**12**(0/2/0)
#<data>.W,(B)	**3**(0/0/0)	**8**(0/0/0)	**10**(0/1/0)
#<data>.L,(B)	**4**(0/0/0)	**10**(0/0/0)	**12**(0/2/0)
#<data>.W,(bd,PC)	**9**(0/0/0)	**14**(0/0/0)	**18**(0/3/0)
#<data>.L,(bd,PC)	**10**(0/0/0)	**16**(0/0/0)	**20**(0/3/0)
#<data>.W,(d_{16},B)	**5**(0/0/0)	**10**(0/0/0)	**13**(0/2/0)
#<data>.L,(d_{16},B)	**6**(0/0/0)	**12**(0/0/0)	**15**(0/2/0)
#<data>.W,(d_{32},B)	**9**(0/0/0)	**14**(0/0/0)	**18**(0/2/0)
#<data>.L,(d_{32},B)	**10**(0/0/0)	**16**(0/0/0)	**20**(0/3/0)
#<data>.W,([B],I)	**8**(1/0/0)	**13**(1/0/0)	**15**(1/2/0)
#<data>.L,([B],I)	**9**(1/0/0)	**15**(1/0/0)	**17**(1/2/0)
#<data>.W,([B],I,d_{16})	**10**(1/0/0)	**15**(1/0/0)	**18**(1/2/0)
#<data>.L,([B],I,d_{16})	**11**(1/0/0)	**17**(1/0/0)	**20**(1/2/0)
#<data>.W,([B],I,d_{32})	**10**(1/0/0)	**15**(1/0/0)	**19**(1/2/0)
#<data>.L,([d_{16},B],I,d_{32})	**11**(1/0/0)	**17**(1/0/0)	**21**(1/3/0)
#<data>.W,([d_{16},B],I)	**10**(1/0/0)	**15**(1/0/0)	**18**(1/2/0)
#<data>.L,([d_{16},B],I)	**11**(1/0/0)	**17**(1/0/0)	**20**(1/2/0)
#<data>.W,([d_{16},B],I,d_{16})	**12**(1/0/0)	**17**(1/0/0)	**21**(1/2/0)

CACHE CASE (Concluded)

Source Address Mode	Destination					
	([d_{16},B],I)	([d_{16},B],I,d_{16})	([d_{16},B],I,d_{32})	([d_{32},B],I)	([d_{32},B],I,d_{16})	([d_{32},B],I,d_{32})
Rn	**14**(1/0/1)	**16**(1/0/1)	**17**(1/0/1)	**18**(1/0/1)	**20**(1/0/1)	**21**(1/0/1)
#<data>.B,W	**14**(1/0/1)	**16**(1/0/1)	**17**(1/0/1)	**18**(1/0/1)	**20**(1/0/1)	**21**(1/0/1)
#<data>.L	**16**(1/0/1)	**18**(1/0/1)	**19**(1/0/1)	**20**(1/0/1)	**22**(1/0/1)	**23**(1/0/1)
(An)	**16**(2/0/1)	**18**(2/0/1)	**19**(2/0/1)	**20**(2/0/1)	**22**(2/0/1)	**23**(2/0/1)
(An)+	**16**(2/0/1)	**18**(2/0/1)	**19**(2/0/1)	**20**(2/0/1)	**22**(2/0/1)	**23**(2/0/1)
–(An)	**17**(2/0/1)	**19**(2/0/1)	**20**(2/0/1)	**21**(2/0/1)	**23**(2/0/1)	**24**(2/0/1)
(d_{16},An) or (d_{16},PC)	**17**(2/0/1)	**19**(2/0/1)	**20**(2/0/1)	**21**(2/0/1)	**23**(2/0/1)	**24**(2/0/1)
(xxx).W	**16**(2/0/1)	**18**(2/0/1)	**19**(2/0/1)	**20**(2/0/1)	**22**(2/0/1)	**23**(2/0/1)
(xxx).L	**16**(2/0/1)	**18**(2/0/1)	**19**(2/0/1)	**20**(2/0/1)	**22**(2/0/1)	**23**(2/0/1)
(d_8,An,Xn) or (d_8,PC,Xn)	**19**(2/0/1)	**21**(2/0/1)	**22**(2/0/1)	**23**(2/0/1)	**25**(2/0/1)	**26**(2/0/1)
(d_{16},An,Xn) or (d_{16},PC,Xn)	**19**(2/0/1)	**21**(2/0/1)	**22**(2/0/1)	**23**(2/0/1)	**25**(2/0/1)	**26**(2/0/1)
(B)	**19**(2/0/1)	**21**(2/0/1)	**22**(2/0/1)	**23**(2/0/1)	**25**(2/0/1)	**26**(2/0/1)
(d_{16},B)	**21**(2/0/1)	**23**(2/0/1)	**24**(2/0/1)	**25**(2/0/1)	**27**(2/0/1)	**28**(2/0/1)
(d_{32},B)	**25**(2/0/1)	**27**(2/0/1)	**28**(2/0/1)	**29**(2/0/1)	**31**(2/0/1)	**32**(2/0/1)
([B],I)	**24**(3/0/1)	**26**(3/0/1)	**27**(3/0/1)	**28**(3/0/1)	**30**(3/0/1)	**31**(3/0/1)
([B],I,d_{16})	**26**(3/0/1)	**28**(3/0/1)	**29**(3/0/1)	**30**(3/0/1)	**32**(3/0/1)	**33**(3/0/1)
([B],I,d_{32})	**26**(3/0/1)	**28**(3/0/1)	**29**(3/0/1)	**30**(3/0/1)	**32**(3/0/1)	**33**(3/0/1)
([d_{16},B],I)	**26**(3/0/1)	**28**(3/0/1)	**29**(3/0/1)	**30**(3/0/1)	**32**(3/0/1)	**33**(3/0/1)
([d_{16},B],I,d_{16})	**28**(3/0/1)	**30**(3/0/1)	**31**(3/0/1)	**32**(3/0/1)	**34**(3/0/1)	**35**(3/0/1)
([d_{16},B],I,d_{32})	**28**(3/0/1)	**30**(3/0/1)	**31**(3/0/1)	**32**(3/0/1)	**34**(3/0/1)	**35**(3/0/1)
([d_{32},B],I)	**30**(3/0/1)	**32**(3/0/1)	**33**(3/0/1)	**34**(3/0/1)	**36**(3/0/1)	**37**(3/0/1)
([d_{32},B],I,d_{16})	**32**(3/0/1)	**34**(3/0/1)	**35**(3/0/1)	**36**(3/0/1)	**38**(3/0/1)	**39**(3/0/1)
([d_{32},B],I,d_{32})	**32**(3/0/1)	**34**(3/0/1)	**35**(3/0/1)	**36**(3/0/1)	**38**(3/0/1)	**39**(3/0/1)

9.2 BYTE SELECT LOGIC FOR THE MC68020/EC020

The MC68020/EC020 architecture supports byte, word, and long-word operand transfers to any 8-, 16-, or 32-bit data port, regardless of alignment. This feature allows the programmer to write code that is not bus-width specific. When accessed, the peripheral or memory subsystem reports its actual port size to the controller, and the MC68020/EC020 then dynamically sizes the data transfer accordingly, using multiple bus cycles when necessary. The following paragraphs describe the generation of byte select control signals that enable the dynamic bus sizing mechanism, the transfer of differently sized operands, and the transfer of misaligned operands to operate correctly.

The following signals control the MC68020/EC020 operand transfer mechanism:

A1, A0 — Address signals. The most significant byte of the operand to be transferred is addressed directly.

SIZ1, SIZ0 — Transfer size signals. Output of the MC68020/EC020. These indicate the number of bytes of an operand remaining to be transferred during a given bus cycle.

R/W — Read/Write signal. Output of the MC68020/EC020. For byte select generation in MC68020/EC020 systems.

DSACK1, DSACK0 — Data transfer and size acknowledge signals. Driven by an asynchronous port to indicate the actual bus width of the port.

The MC68020/EC020 assumes that 16-bit ports are situated on data lines D31–D16, and that 8-bit ports are situated on data lines D31–D24. This ensures that the following logic works correctly with the MC68020/EC020's on-chip internal-to-external data bus multiplexer. Refer to **Section 5 Bus Operation** for more details on the dynamic bus sizing mechanism.

The need for byte select signals is best illustrated by an example. Consider a long-word write cycle to an odd address in word-organized memory. The transfer requires three bus cycles to complete. The first bus cycle transfers the most significant byte of the long word on D23–D16. The second bus cycle transfers a word on D31–D16, and the last bus cycle transfers the least significant byte of the original long word on D31–D24. To prevent overwriting those bytes that are not used in these transfers, a unique byte data strobe must be generated for each byte when using devices with 16- and 32-bit port widths.

For noncachable read cycles and all write cycles, the required active bytes of the data bus for any given bus transfer are a function of the SIZ1, SIZ0 and A1, A0 outputs (see Table 9-1). Individual strobes or select signals can be generated by decoding these four signals for every bus cycle. Devices residing on 8-bit ports can utilize DS or AS since there is only one valid byte for any transfer.

Table 9-6. Access Status Register Codes

Value	Validity	Processor Action
$00	Invalid	Format Error
$01	Valid	No Change in Access Rights
$02–$03	Valid	Change Access Rights with No Change of Stack Pointer
$04–$07	Valid	Change Access Rights and Change Stack Pointer
Other	Undefined	Undefined (Take Format Error Exception)

The processor uses the descriptor address registers during the CALLM instruction to communicate the address of the type $01 descriptor, allowing external hardware to verify that the address is a valid address for a type $01 descriptor. This validation prevents a module from creating a type $01 descriptor to increase its access rights.

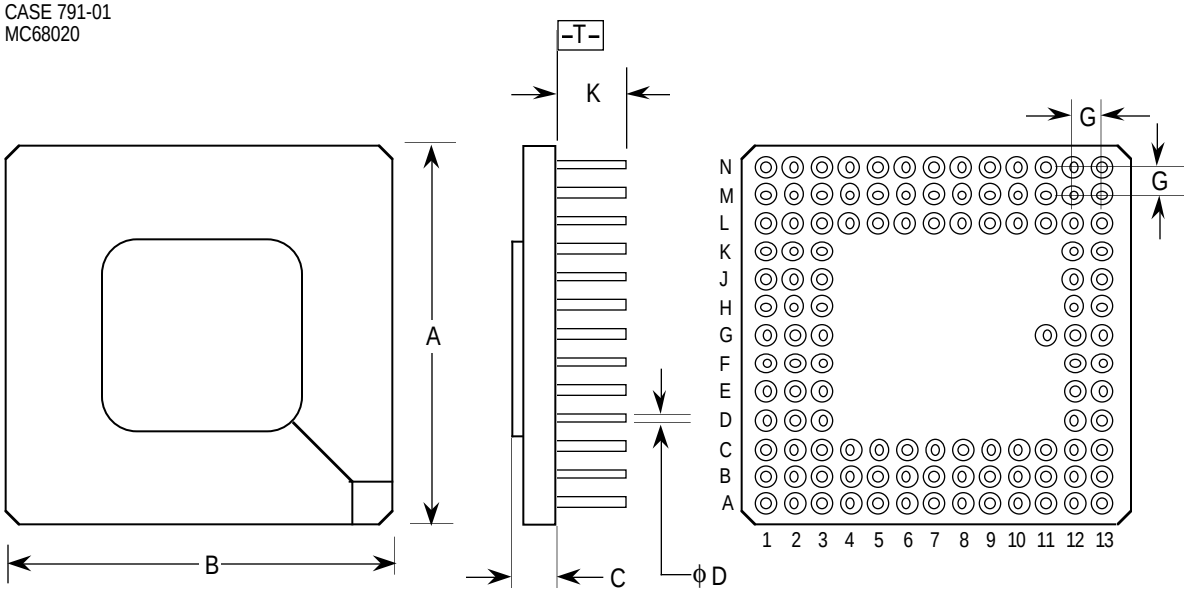
9.8.1 Module Call

The CALLM instruction is used to make the module call. For the type $00 module descriptor, the processor creates and fills the module stack frame at the top of the active system stack. The condition codes of the calling module are saved in the CCR field of the frame. If opt is equal to 000 (arguments passed on the stack) in the module descriptor, the MC68020/EC020 does not save the stack pointer or load a new stack pointer value. The processor uses the module entry word to save and load the module data area pointer register and then begins execution of the called module.

For the type $01 module descriptor, the processor must first obtain the current access level from external hardware. It also verifies that the calling module has the right to read from the area pointed to by the current value of the stack pointer by reading from that address. It passes the descriptor address and increase access level to external hardware for validation and then reads the access status. If external hardware determines that the change in access rights should not be granted, the access status is zero, and the processor takes a format error exception. No visible processor registers are changed, nor should the current access level enforced by external hardware be changed. If external hardware determines that a change should be granted, the external hardware changes its access level, and the processor proceeds. If the access status register indicates that a change in the stack pointer is required, the stack pointer is saved internally, a new value is loaded from the module descriptor, and arguments are copied from the calling stack to the new stack. Finally, the module stack frame is created and filled on the top of the current stack. The condition codes of the calling module are saved in the CCR field of the frame. Execution of the called module then begins as with a type $00 descriptor.

11.2.2 MC68020 RC Suffix—Package Dimensions

RC SUFFIX
CASE 791-01
MC68020



DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	34.04	35.05	1.340	1.380
B	34.04	35.05	1.340	1.380
C	2.54	3.81	0.100	0.150
D	0.43	0.55	0.017	0.022
G	2.54 BSC		0.100 BSC	
K	4.32	4.95	0.170	0.195

NOTES:
1. **A** AND **B** ARE DATUMS AND **T** IS A DATUM SURFACE.
2. POSITIONAL TOLERANCE FOR LEADS (114 PLACES).

⌖	0.13 (0.005) Ⓜ	T	A Ⓢ	B Ⓢ

3. DIMENSIONING AND TOLERANCING PER Y14.5M,1982.
4. CONTROLLING DIMENSION: INCH.