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	0°C ~ 70°C (TA)
Security Features	-
Package / Case	114-BPGA
Supplier Device Package	114-PGA (34.55x34.55)
Purchase URL	https://www.e-xfl.com/pro/item?MUrl=&PartUrl=mc68020rc25e

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS (Continued)

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.

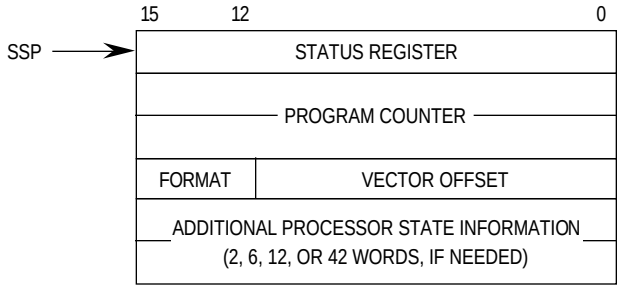
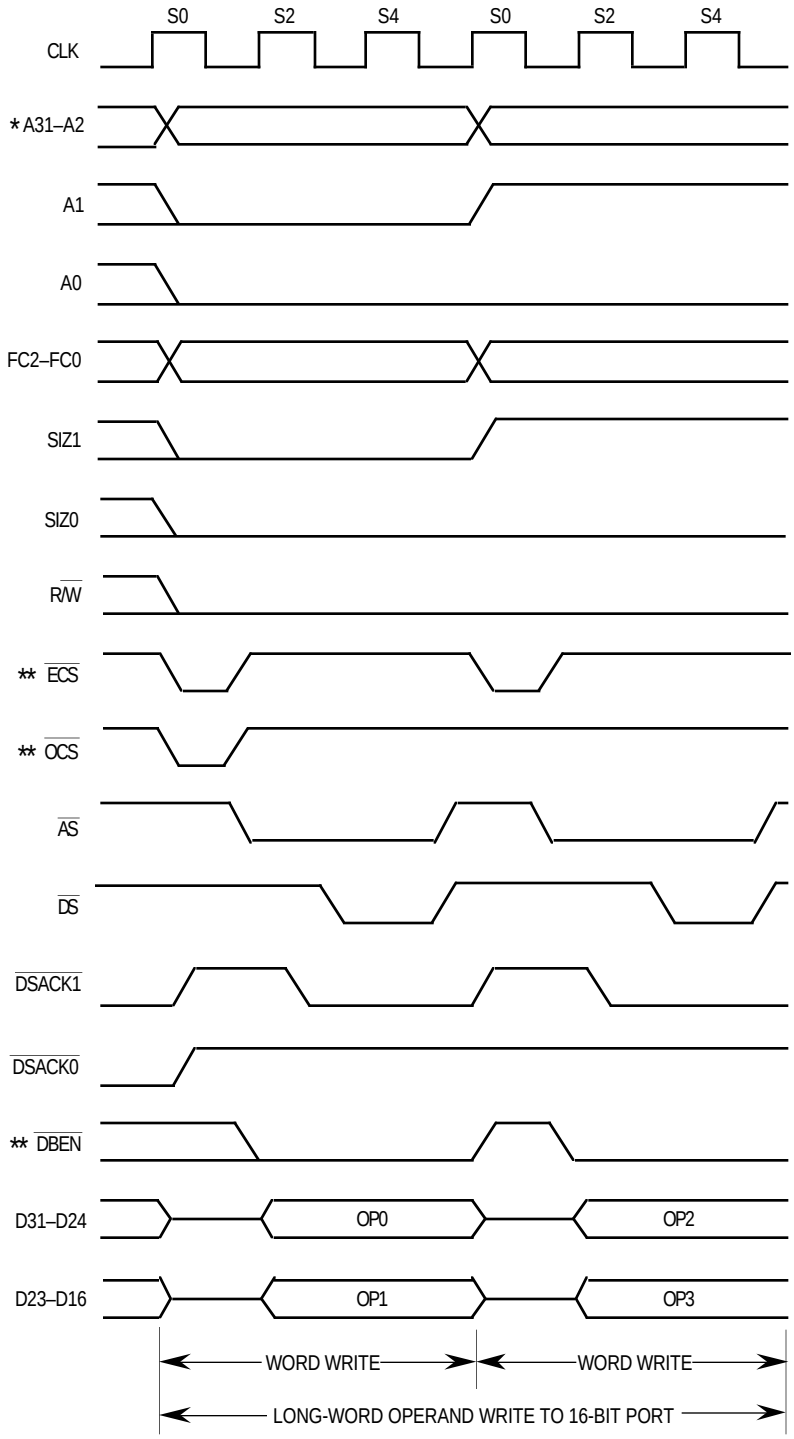


Figure 2-1. General Exception Stack Frame



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

Figure 5-6. Long-Word Operand Write to Word Port Timing

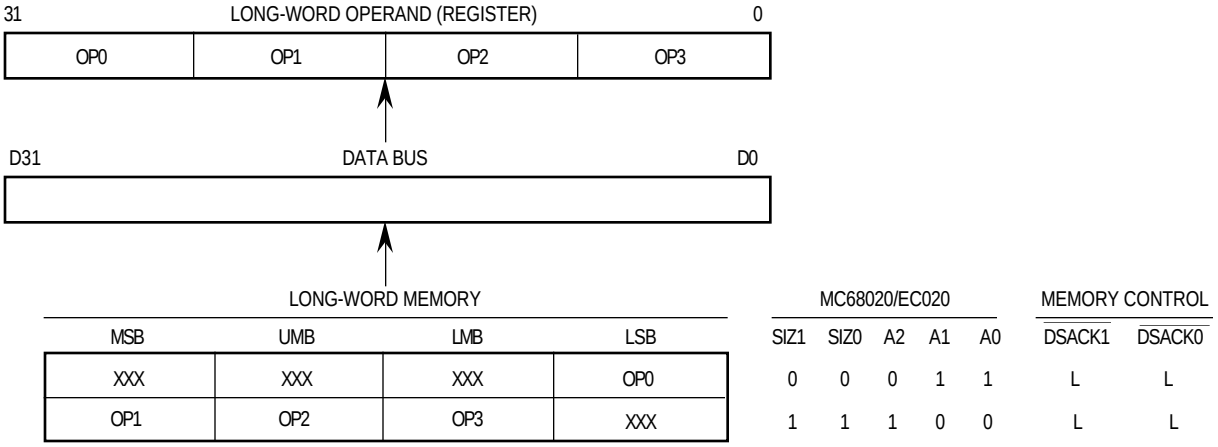


Figure 5-17. Misaligned Long-Word Operand Read from Long-Word Port Example

5.2.3 Effects of Dynamic Bus Sizing and Operand Misalignment

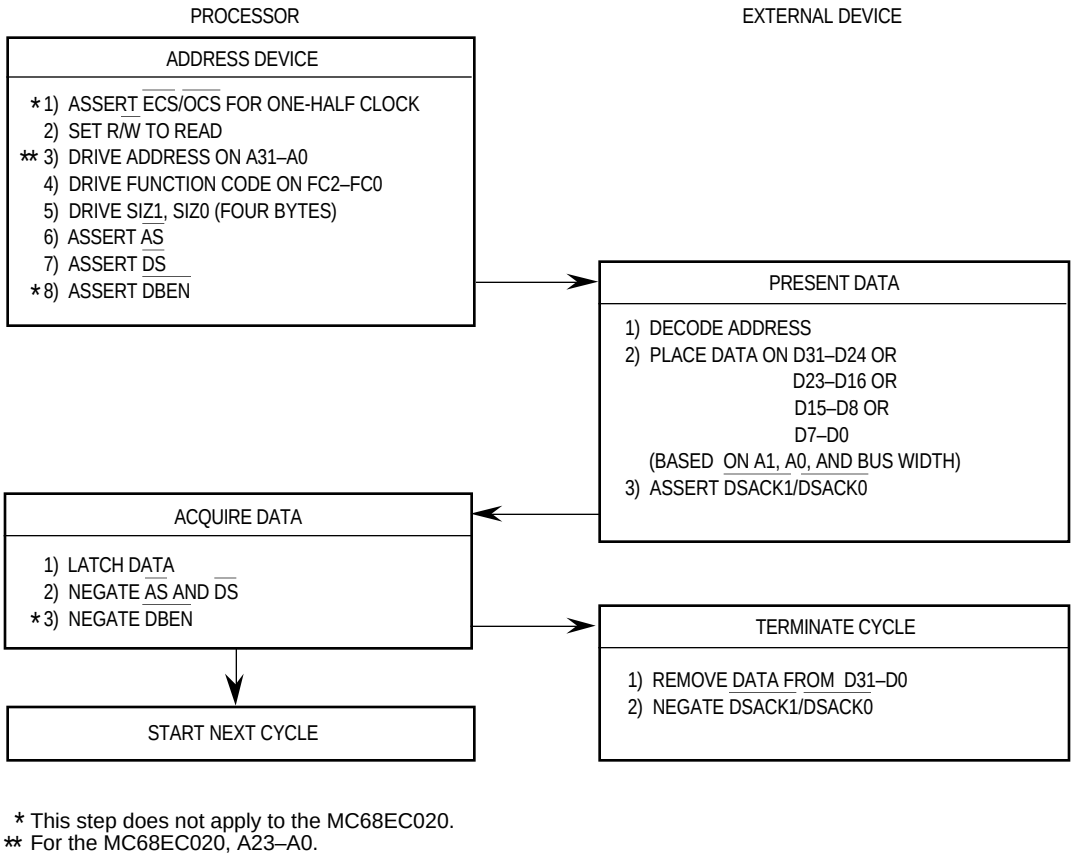
The combination of operand size, operand alignment, and port size determine the number of bus cycles required to perform a particular memory access. Table 5-6 lists the number of bus cycles required for different operand sizes to different port sizes with all possible alignment conditions for read/write cycles.

Table 5-6. Memory Alignment and Port Size Influence on Read/Write Bus Cycles

	Number of Bus Cycles (Data Port Size = 32 Bits:16 Bits:8 Bits)			
	A1, A0			
Operand Size	00	01	10	11
Instruction*	1:2:4	N/A	N/A	N/A
Byte Operand	1:1:1	1:1:1	1:1:1	1:1:1
Word Operand	1:1:2	1:2:2	1:1:2	2:2:2
Long-Word Operand	1:2:4	2:3:4	2:2:4	2:3:4

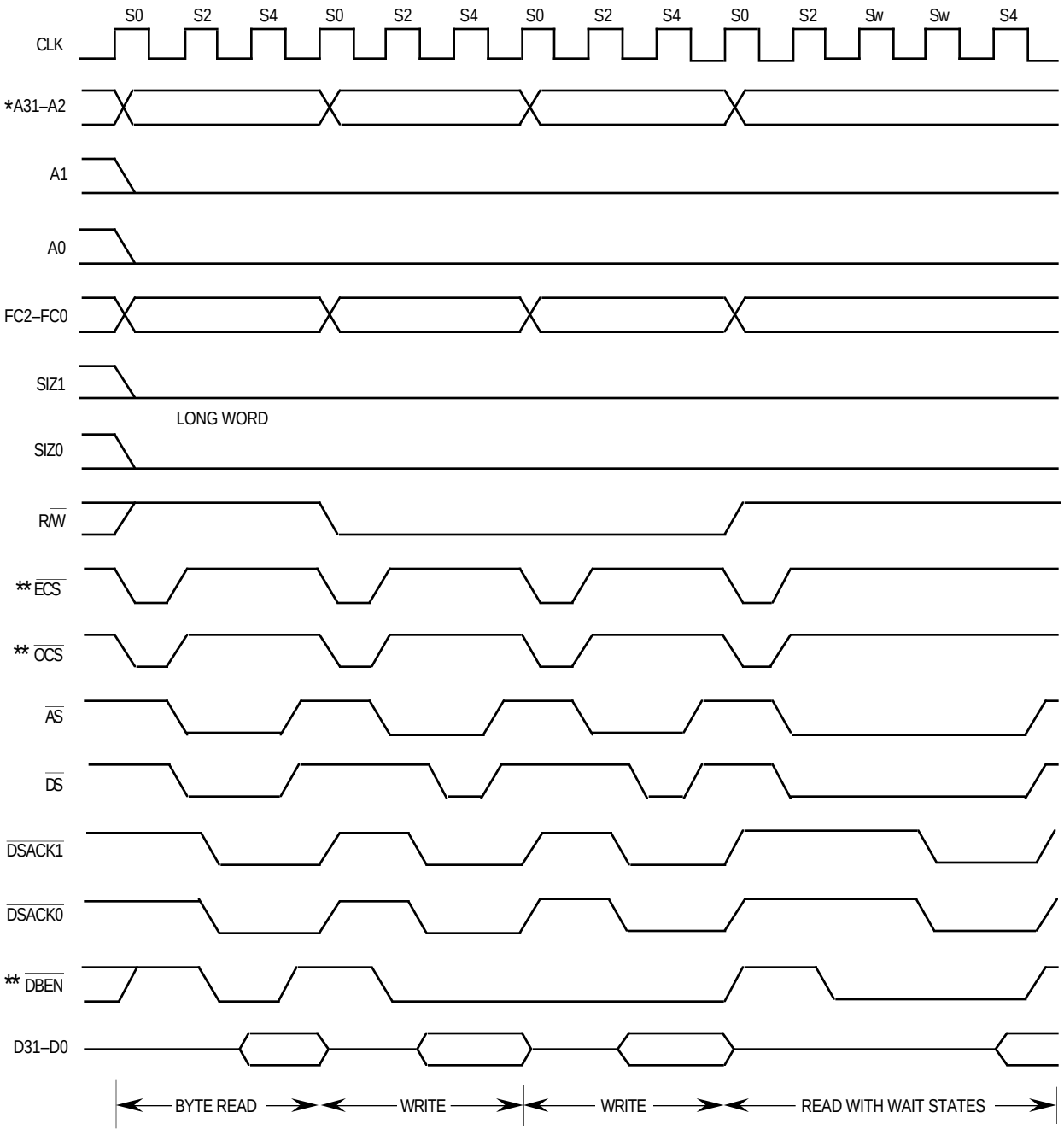
*Instruction prefetches are always two words from a long-word boundary

Table 5-6 reveals that bus cycle throughput is significantly affected by port size and alignment. The MC68020/EC020 system designer and programmer should be aware of and account for these effects, particularly in time-critical applications.



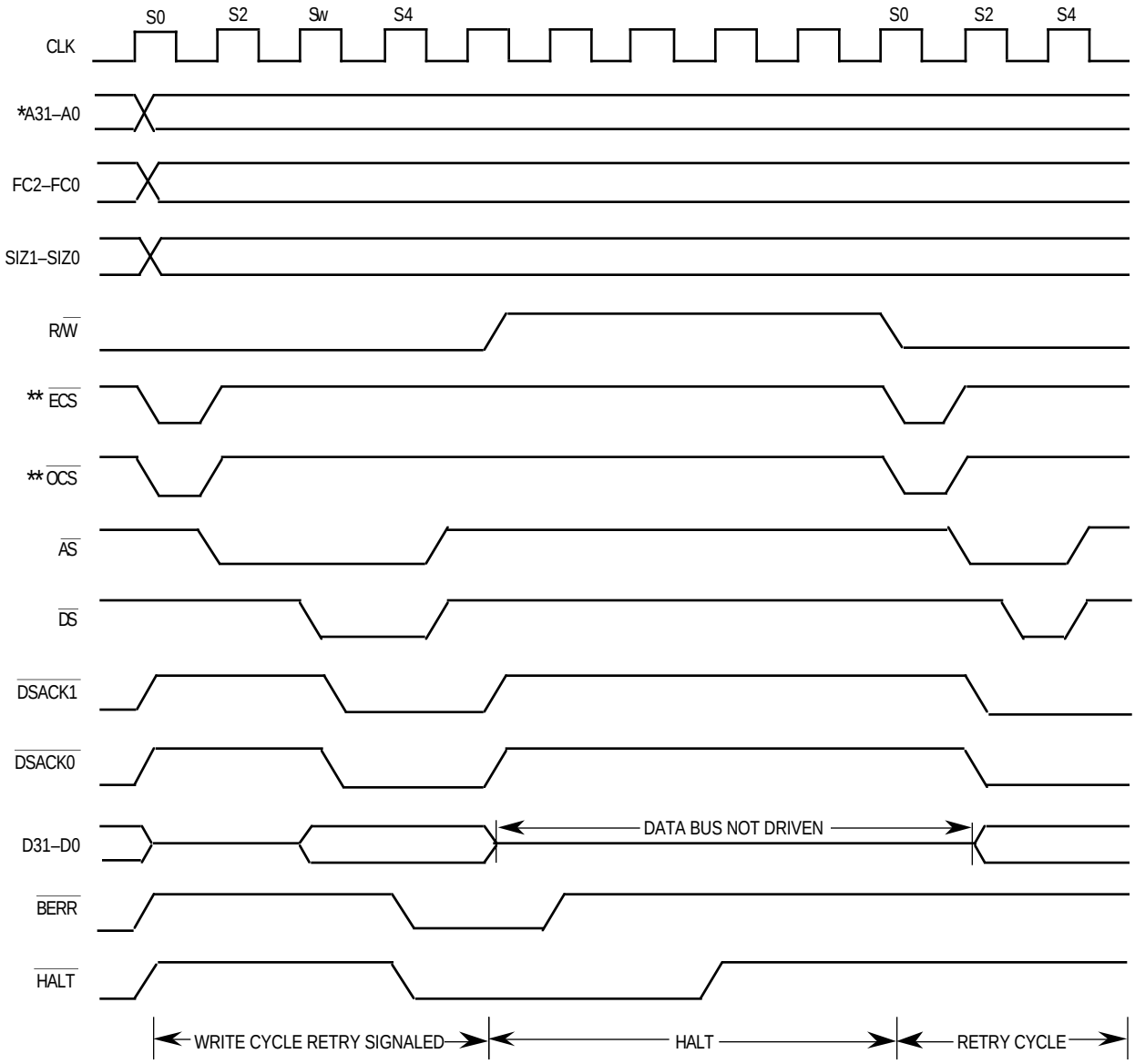
* This step does not apply to the MC68EC020.
** For the MC68EC020, A23–A0.

Figure 5-20. Byte Read Cycle Flowchart



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

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



* For the MC68EC020, A23–A0.
** This signal does not apply to the MC68EC020.

Figure 5-40. Late Retry

State changes occur on the next rising edge of the clock after the internal signal is recognized as valid. The $\overline{BG}$ signal transitions on the falling edge of the clock after a state is reached during which G changes. The bus control signals (controlled by T) are driven by the processor immediately following a state change when bus mastership is returned to the MC68EC020.

State 0, at the top center of the diagram, in which both G and T are negated, is the state of the bus arbiter while the processor is bus master. Request R keeps the arbiter in state 0 as long as it is negated. When a request R is received, both grant G and signal T are asserted (in state 1 at the top left). The next clock causes a change to state 2, at the lower left, in which G and T are held. The bus arbiter remains in that state until request R is negated. Then the arbiter changes to the center state, state 3, and negates grant G. The next clock takes the arbiter to state 4, at the upper right, in which grant G remains negated and signal T remains asserted. The arbiter returns to the original state, state 0, and negates signal T. This sequence of states follows the normal sequence of signals for relinquishing the bus to an external bus master. Other states apply to other possible sequences of R.

The MC68EC020 does not allow arbitration of the external bus during the read-modify-write sequence. For the duration of this sequence, the MC68EC020 ignores the $\overline{BR}$ input. If mastership of the MC68EC020 bus is required during a read-modify-write operation, $\overline{BERR}$ must be used to abort the read-modify-write sequence. The bus arbitration sequence while the bus is inactive (i.e., executing internal operations such as a multiply instruction) is shown in Figure 5-49.

execution of instructions. The processor also saves the contents of some of its internal registers. The information saved on the stack is sufficient to identify the cause of the bus fault and recover from the error.

For efficiency, the MC68020/EC020 uses two different bus error stack frame formats. When the bus error exception is taken at an instruction boundary, less information is required to recover from the error, and the processor builds the short bus fault stack frame as shown in Table 6-5. When the exception is taken during the execution of an instruction, the processor must save its entire state for recovery and uses the long bus fault stack frame shown in Table 6-5. The format code in the stack frame distinguishes the two stack frame formats. Stack frame formats are described in detail in **6.4 Exception Stack Frame Formats**.

If a bus error occurs during the exception processing for a bus error, address error, or reset or while the processor is loading internal state information from the stack during the execution of an RTE instruction, a double bus fault occurs and the processor enters the halted state. In this case, the processor does not attempt to alter the current state of memory. Only an external RESET can restart a processor halted by a double bus fault.

6.1.3 Address Error Exception

An address error exception occurs when the processor attempts to prefetch an instruction from an odd address. This exception is similar to a bus error exception but is internally initiated. A bus cycle is not executed, and the processor begins exception processing immediately. After exception processing commences, the sequence is the same as that for bus error exceptions described in the preceding paragraphs, except that the vector number is 3 and the vector offset in the stack frame refers to the address error vector. Either a short or long bus fault stack frame may be generated. If an address error occurs during the exception processing for a bus error, address error, or reset, a double bus fault occurs.

6.1.4 Instruction Trap Exception

Certain instructions are used to explicitly cause trap exceptions. The TRAP instruction always forces an exception and is useful for implementing system calls in user programs. The TRAPcc, TRAPV, cpTRAPcc, CHK, and CHK2 instructions force exceptions if the user program detects an error, which may be an arithmetic overflow or a subscript value that is out of bounds.

The DIVS and DIVU instructions force exceptions if a division operation is attempted with a divisor of zero.

When a trap exception occurs, the processor copies the SR internally, enters the supervisor privilege level (by setting the S-bit in the SR), and clears the T1 and T0 bits in the SR. If tracing is enabled for the instruction that caused the trap, a trace exception is taken after the RTE instruction from the trap handler is executed, and the trace corresponds to the trap instruction; the trap handler routine is not traced. The processor generates a vector number according to the instruction being executed; for the TRAP

tracing is enabled, the trace exception processing should also be emulated for the trace exception handler to account for the emulated instruction.

The exception processing for a trace starts at the end of normal processing for the traced instruction and before the start of the next instruction. The processor makes an internal copy of the SR and enters the supervisor privilege level (by setting the S-bit in the SR). It also clears the T0 and T1 bits of the SR, disabling further tracing. The processor supplies vector number 9 for the trace exception and saves the trace exception vector offset, PC value, and the copy of the SR on the supervisor stack. The saved value of the PC is the logical address of the next instruction to be executed. Instruction execution resumes after the required prefetches from the address in the trace exception vector.

The STOP instruction does not perform its function when it is traced. A STOP instruction that begins execution with T1, T0 = 10 forces a trace exception after it loads the SR. Upon return from the trace handler routine, execution continues with the instruction following the STOP instruction, and the processor never enters the stopped condition.

6.1.8 Format Error Exception

Just as the processor checks that prefetched instructions are valid, the processor (with the aid of a coprocessor, if needed) also performs some checks of data values for control operations, including the type and option fields of the descriptor for CALLM, the coprocessor state frame format word for a cpRESTORE instruction, and the stack frame format for an RTE or an RTM instruction.

The RTE instruction checks the validity of the stack format code. For long bus fault format frames, the RTE instruction also compares the internal version number of the processor to that contained in the frame at memory location SP + 54 (SP + $36). This check ensures that the processor can correctly interpret internal state information from the stack frame.

The CALLM and RTM both check the values in the option and type fields in the module descriptor and module stack frame, respectively. If these fields do not contain proper values or if an illegal access rights change request is detected by an external memory management unit, then an illegal call or return is being requested and is not executed. Refer to **Section 9 Applications Information** for more information on the module call/return mechanism.

The cpRESTORE instruction passes the format word of the coprocessor state frame to the coprocessor for validation. If the coprocessor does not recognize the format value, it signals the MC68020/EC020 to take a format error exception. Refer to **Section 7 Coprocessor Interface Description** for details of coprocessor-related exceptions.

If any of the checks previously described determine that the format of the stacked data is improper, the instruction generates a format error exception. This exception saves a short bus fault stack frame, generates exception vector number 14, and continues execution at the address in the format exception vector. The stacked PC value is the logical address of the instruction that detected the format error.

corresponding fault bit (FB or FC) is cleared, the associated prefetch cycle may or may not be run by the RTE instruction (depending on whether the stage is required).

If a fault occurs when the RTE instruction attempts to rerun the bus cycle(s), the processor creates a new stack frame on the supervisor stack after deallocating the previous frame, and address error or bus error exception processing starts in the normal manner.

The read-modify-write operations of the MC68020/EC020 can also be completed by the RTE instruction that terminates the handler routine. The rerun operation, executed by the RTE instruction with the DF bit of the SSW set, reruns the entire instruction. If the cause of the error has been corrected, the handler does not need to emulate the instruction but can leave the DF bit set and execute the RTE instruction.

6.3 COPROCESSOR CONSIDERATIONS

Exception handler programmers should consider carefully whether to save and restore the context of a coprocessor at the beginning and end of handler routines for exceptions that can occur during the execution of a coprocessor instruction (i.e., bus errors, interrupts, and coprocessor-related exceptions). The nature of the coprocessor and the exception handler routine determines whether or not saving the state of one or more coprocessors with the cpSAVE and cpRESTORE instructions is required. If the coprocessor allows multiple coprocessor instructions to be executed concurrently, it may require its state to be saved and restored for all coprocessor-generated exceptions, regardless of whether or not the coprocessor is accessed during the handler routine. The MC68882 floating-point coprocessor is an example of this type of coprocessor. On the other hand, the MC68881 floating-point coprocessor requires FSAVE and FRESTORE instructions within an exception handler routine only if the exception handler itself uses the coprocessor.

6.4 EXCEPTION STACK FRAME FORMATS

The MC68020/EC020 provides six different stack frames for exception processing. The set of frames includes the normal four- and six-word stack frames, the four-word throwaway stack frame, the coprocessor midinstruction stack frame, and the short and long bus fault stack frames.

When the MC68020/EC020 writes or reads a stack frame, it uses long-word operand transfers wherever possible. Using a long-word-aligned stack pointer with memory that is on a 32-bit port greatly enhances exception processing performance. The processor does not necessarily read or write the stack frame data in sequential order.

The system software should not depend on a particular exception generating a particular stack frame. For compatibility with future devices, the software should be able to handle any type of stack frame for any type of exception.

Table 6-5 summarizes the stack frames defined for the MC68020/EC020.

model of sequential, nonconcurrent instruction execution at the user level. Consequently, the programmer can assume that the images of registers and memory affected by a given instruction have been updated when the next instruction in the sequence accessing these registers or memory locations is executed.

The M68000 coprocessor interface provides full support of all operations necessary for nonconcurrent operation of the main processor and its associated coprocessors. Although the M68000 coprocessor interface allows concurrency in coprocessor execution, the coprocessor designer is responsible for implementing this concurrency while maintaining a programming model based on sequential nonconcurrent instruction execution.

For example, if the coprocessor determines that instruction B does not use or alter resources to be altered or used by instruction A, instruction B can be executed concurrently (if the execution hardware is also available). Thus, the required instruction interdependencies and sequences of the program are always respected. The MC68882 coprocessor offers concurrent instruction execution; whereas, the MC68881 coprocessor does not. However, the MC68020/EC020 can execute instructions concurrently with coprocessor instruction execution in the MC68881.

7.1.3 Coprocessor Instruction Format

The instruction set for a given coprocessor is defined by the design of that coprocessor. When a coprocessor instruction is encountered in the main processor instruction stream, the MC68020/EC020 hardware initiates communication with the coprocessor and coordinates any interaction necessary to execute the instruction with the coprocessor. A programmer needs to know only the instruction set and register set defined by the coprocessor to use the functions provided by the coprocessor hardware.

The instruction set of an M68000 coprocessor uses a subset of the F-line operation words in the M68000 instruction set. The operation word is the first word of any M68000 family instruction. The F-line operation word contains ones in bits 15–12 (refer to Figure 7-1); the remaining bits are coprocessor and instruction dependent. The F-line operation word may be followed by as many extension words as are required to provide additional information necessary for the execution of the coprocessor instruction.

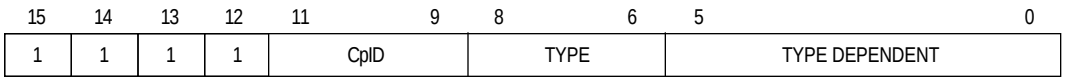
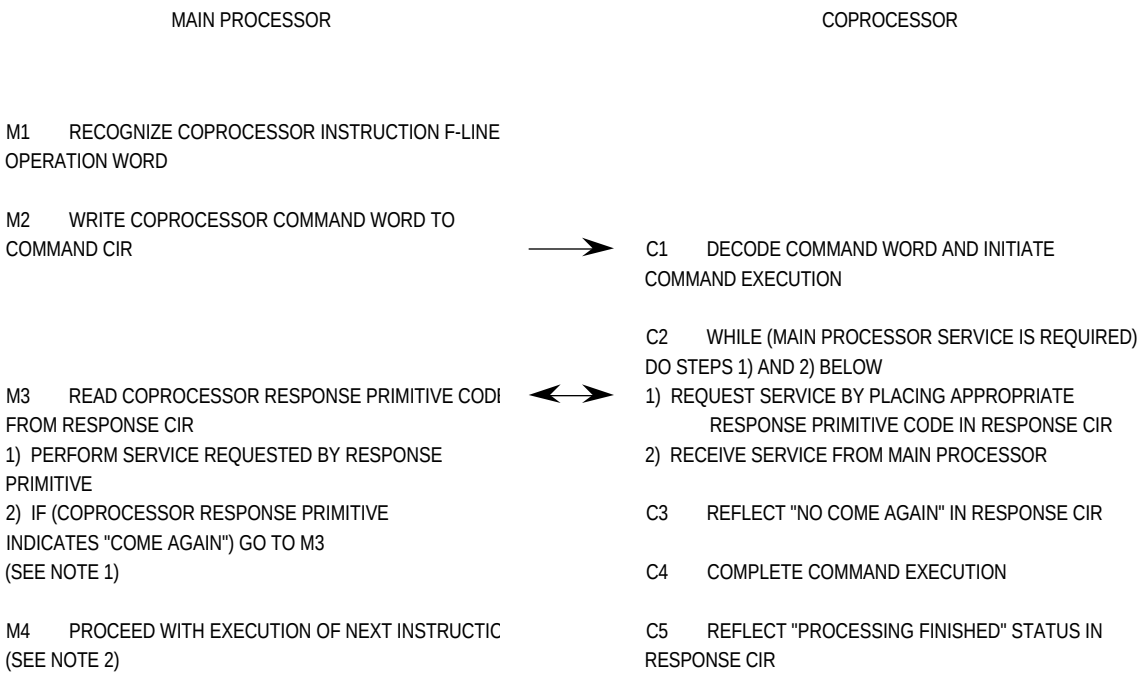


Figure 7-1. F-Line Coprocessor Instruction Operation Word

As shown in Figure 7-1, bits 11–9 of the F-line operation word encode the coprocessor identification (CpID) field. The MC68020/EC020 uses the CpID field to indicate the coprocessor to which the instruction applies. F-line operation words, in which the CpID is zero, are not coprocessor instructions for the MC68020/EC020. Instructions with a CpID of zero and a nonzero type field are unimplemented instructions that cause the



NOTES: 1. "Come Again" indicates that further service of the main processor is being requested by the coprocessor.
2. The next instruction should be the operation word pointed to by the ScanPC at this point. The operation of the MC68020/EC020 ScanPC is discussed in **7.4.1 ScanPC**.

Figure 7-7. Coprocessor Interface Protocol for General Category Instructions

7.2.2 Coprocessor Conditional Instructions

The conditional instruction category provides program control based on the operations of the coprocessor. The coprocessor evaluates a condition and returns a true/false indicator to the main processor. The main processor completes the execution of the instruction based on this true/false condition indicator.

The implementation of instructions in the conditional category promotes efficient use of both the main processor and the coprocessor hardware. The condition specified for the instruction is related to the coprocessor operation and is therefore evaluated by the coprocessor. However, the instruction completion following the condition evaluation is directly related to the operation of the main processor. The main processor performs the change of flow, the setting of a byte, or the TRAP operation, since its architecture explicitly implements these operations for its instruction set.

Figure 7-8 shows the protocol for a conditional category coprocessor instruction. The main processor initiates execution of an instruction in this category by writing a condition selector to the condition CIR. The coprocessor decodes the condition selector to determine the condition to evaluate. The coprocessor can use response primitives to request that the main processor provide services required for the condition evaluation.

After writing the format word to the restore CIR, the main processor continues cpRESTORE dialog by reading that same register. If the coprocessor returns a valid format word, the main processor transfers the number of bytes specified by the format word at the effective address to the operand CIR.

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 **7.2.3.2.3 Invalid Format Word**) to the control CIR, and initiates format error exception processing (refer to **7.5.1.5 Format Errors**).

The cpRESTORE instruction is a privileged instruction. When the MC68020/EC020 accesses a cpRESTORE instruction, it checks the S-bit in the SR. If the MC68020/EC020 attempts to execute a cpRESTORE instruction while at the user privilege level (S-bit in the SR is clear), it initiates privilege violation exception processing without accessing any of the CIRs (refer to **7.5.2.3 Privilege Violations**).

7.3 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 programming model.

Figure 7-4 is a memory map of the CIR set. The response, control, save, restore, command, condition, and operand registers must be included in a coprocessor interface that implements all four coprocessor instruction categories. The complete register model must be implemented if the system uses all coprocessor response primitives defined for the M68000 coprocessor interface.

The following paragraphs contain detailed descriptions of the registers.

7.3.1 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 **7.4 Coprocessor Response Primitives** for additional information.

7.3.2 Control CIR

The main processor writes to the 2-bit control CIR to acknowledge coprocessor-requested exception processing or to abort the execution of a coprocessor instruction. The offset from the base address of the CIR set for the control CIR is $02. The control CIR occupies the two least significant bits of the word at that offset. The 14 most significant bits of the word are undefined and reserved by Motorola. Figure 7-19 shows the format of this register.

The write to previously evaluated effective address primitive uses the CA and PC bits as described in **7.4.2 Coprocessor Response Primitive General Format**.

The length field of the primitive format specifies the length of the operand in bytes. The MC68020/EC020 transfers operands of 0–255 bytes in length.

When the main processor receives this primitive during the execution of a general category instruction, it transfers an operand from the operand CIR to an effective address specified by a temporary register within the MC68020/EC020. When a previous primitive for the current instruction has evaluated the effective address, this temporary register contains the evaluated effective address. Primitives that store an evaluated effective address in a temporary register of the main processor are the evaluate and transfer effective address, evaluate effective address and transfer data, and transfer multiple coprocessor registers primitive. If this primitive is used during an instruction in which the effective address specified in the instruction operation word has not been calculated, the effective address used for the write is undefined. Also, if the previously evaluated effective address was register direct, the address written to in response to this primitive is undefined.

The function code value during the write operation indicates either supervisor or user data space, depending on the value of the S-bit in the MC68020/EC020 SR when the processor reads this primitive. While a coprocessor should request writes to only alterable effective addressing modes, the MC68020/EC020 does not check the type of effective address used with this primitive. For example, if the previously evaluated effective address was PC relative and the MC68020/EC020 is at the user privilege level (S = 0 in SR), the MC68020/EC020 writes to user data space at the previously calculated program relative address (the 32-bit value in the temporary internal register of the processor).

Operands longer than four bytes are transferred in increments of four bytes (operand parts) when possible. The main processor reads a long-word operand part from the operand CIR and transfers this part to the current effective address. The transfers continue in this manner using ascending memory locations until all of the long-word operand parts are transferred, and any remaining operand part is then transferred using a one-, two-, or three-byte transfer as required. The operand parts are stored in memory using ascending addresses beginning with the address in the MC68020/EC020 temporary register, which is internal to the processor and not for user use.

The execution of this primitive does not modify any of the registers in the MC68020/EC020 programming model, even if the previously evaluated effective address mode is the predecrement or postincrement mode. If the previously evaluated effective addressing mode used any of the MC68020/EC020 internal address or data registers, the effective address value used is the final value from the preceding primitive. That is, this primitive uses the value from an evaluate and transfer effective address, evaluate effective address and transfer data, or transfer multiple coprocessor registers primitive without modification.

NOTE

This CC time is a maximum since the times given for the MULU.L and DIVS.L are maximums.

The MOVE instruction timing tables include all necessary timing for extension word fetch, address calculation, and operand fetch.

The instruction timing tables are used to calculate a best-case and worst-case bounds for some target instruction stream. Calculating exact timing from the timing tables is impossible because the tables cannot anticipate how the combination of factors will influence every particular sequence of instructions. This is illustrated by comparing the observed instruction timing from the prior four examples with instruction timing derived from the instruction timing tables.

Table 8-2 lists the original instruction stream and the corresponding clock timing from the appropriate timing tables for the best case, cache-only case, and worst case.

Table 8-2. Instruction Timings from Timing Tables

Instruction		Best Case	Cache Case	Worst Case
#1) MOVE.L	D4,(A1)+	4	4	6
#2) ADD.L	D4,D5	0	2	3
#3) MOVE.L	(A1),–(A2)	6	7	9
#4) ADD.L	D5,D6	0	2	3
Total		10	15	21

Table 8-3 summarizes the observed instruction timings for the same instruction stream as executed according to the assumptions of the four examples. For each example, Table 8-3 shows which entry (BC/CC/WC) from the timing tables corresponds to the observed timing for each of the four instructions. Some of the observed instruction timings cannot be found in the timing tables and appear in Table 8-3 within parentheses in the most appropriate column. These timings occur when instruction execution overlap dynamically alters what would otherwise be a BC, CC, or WC timing.

Table 8-3. Observed Instruction Timings

Instruction		Example 1			Example 2			Example 3			Example 4		
		BC	CC	WC	BC	CC	WC	BC	CC	WC	BC	CC	WC
#1) MOVE.L	D4,(A1)+			6	4				4			(5)	
#2) ADD.L	D4,D5	0					3	0			0		
#3) MOVE.L	(A1),–(A2)			9	6				7			(8)	
#4) ADD.L	D5,D6	(1)					3	(1)			0		
Total			(16)			(16)			(12)			(13)	

8.2.17 Exception-Related Instructions

The exception-related instructions table indicates the number of clock periods needed for the processor to perform the specified exception-related action. Footnotes specify when it is necessary to add the entry from another table to calculate the total effective execution time for the given instruction. 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
BKPT	**9**(1/0/0)	**10**(1/0/0)	**10**(1/0/0)
Interrupt (I-Stack)	**26**(2/0/4)	**26**(2/0/4)	**33**(2/2/4)
Interrupt (M-Stack)	**41**(2/0/8)	**41**(2/0/8)	**48**(2/2/8)
RESET Instruction	**518**(0/0/0)	**518**(0/0/0)	**519**(0/1/0)
STOP	**8**(0/0/0)	**8**(0/0/0)	**8**(0/0/0)
Trace	**25**(1/0/5)	**25**(1/0/5)	**32**(1/2/5)
TRAP #n	**20**(1/0/4)	**20**(1/0/4)	**27**(1/2/4)
Illegal Instruction	**20**(1/0/4)	**20**(1/0/4)	**27**(1/2/4)
A-Line Trap	**20**(1/0/4)	**20**(1/0/4)	**27**(1/2/4)
F-Line Trap	**20**(1/0/4)	**20**(1/0/4)	**27**(1/2/4)
Privilege Violation	**20**(1/0/4)	**20**(1/0/4)	**27**(1/2/4)
TRAPcc (Trap)	**23**(1/0/5)	**25**(1/0/5)	**32**(1/2/5)
TRAPcc (No Trap)	**1**(0/0/0)	**4**(0/0/0)	**5**(0/1/0)
TRAPcc.W (Trap)	**23**(1/0/5)	**25**(1/0/5)	**33**(1/3/5)
TRAPcc.W (No Trap)	**3**(0/0/0)	**6**(0/0/0)	**7**(0/1/0)
TRAPcc.L (Trap)	**23**(1/0/5)	**25**(1/0/5)	**33**(1/3/5)
TRAPcc.L (No Trap)	**5**(0/0/0)	**8**(0/0/0)	**10**(0/2/0)
TRAPV (Trap)	**23**(1/0/5)	**25**(1/0/5)	**32**(1/2/5)
TRAPV (No Trap)	**1**(0/0/0)	**4**(0/0/0)	**5**(0/1/0)

8.2.18 Save and Restore Operations

The save and restore operations table indicates the number of clock periods needed for the processor to perform the specified state save or return from exception, with complete execution times and stack length given. No additional tables are needed to calculate total effective execution time for these operations. 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.

Operation	Best Case	Cache Case	Worst Case
Bus Cycle Fault (Short)	**42**(1/0/10)	**43**(1/0/10)	**50**(1/2/10)
Bus Cycle Fault (Long)	**79**(1/0/24)	**79**(1/0/24)	**86**(1/2/24)
RTE (Normal)	**20**(4/0/0)	**21**(4/0/0)	**24**(4/2/0)
RTE (Six Word)	**20**(4/0/0)	**21**(4/0/0)	**24**(4/2/0)
RTE (Throwaway)*	**15**(4/0/0)	**16**(4/0/0)	**39**(4/0/0)
RTE (Coprocessor)	**31**(7/0/0)	**32**(7/0/0)	**33**(7/1/0)
RTE (Short Fault)	**42**(10/0/0)	**43**(10/0/0)	**45**(10/2/0)
RTE (Long Fault)	**91**(24/0/0)	**92**(24/0/0)	**94**(24/2/0)

*Add the time for RTE on second stack frame.

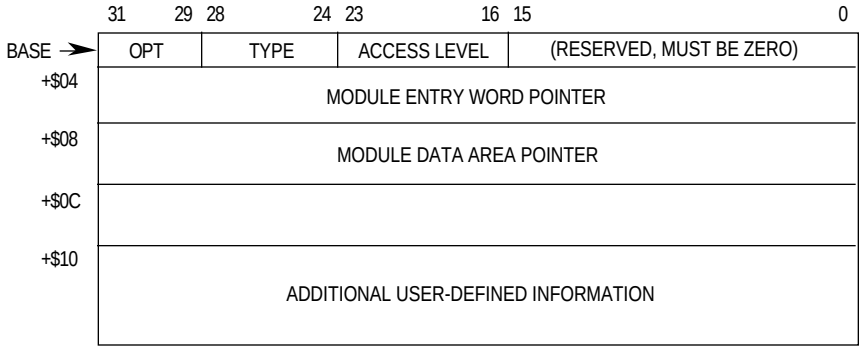


Figure 9-10. Module Descriptor Format

The opt field specifies how arguments are to be passed to the called module; the MC68020/EC020 recognizes only the options of 000 and 100; all others cause a format exception. The 000 option indicates that the called module expects to find arguments from the calling module on the stack just below the module stack frame. In cases where there is a change of stack pointer during the call, the MC68020/EC020 will copy the arguments from the old stack to the new stack. The 100 option indicates that the called module will access the arguments from the calling module through an indirect pointer in the stack of the calling module. Hence, the arguments are not copied, but the MC68020/EC020 puts the value of the stack pointer from the calling module in the module stack frame.

The type field specifies the type of the descriptor; the MC68020/EC020 only recognizes descriptors of type $00 and $01; all others cause a format exception. The $00 type descriptor defines a module for which there is no change in access rights, and the called module builds its stack frame on top of the stack used by the calling module. The $01 type descriptor defines a module for which there may be a change in access rights; such a called module may have a separate stack area from that of the calling module.

The access level field is used only with the type $01 descriptor and is passed to external hardware to change the access control.

The module entry word pointer specifies the entry address of the called module. The first word at the entry address (see Figure 9-11) specifies the register to be saved in the module stack frame and then loaded with the module descriptor data area pointer; the first instruction of the module starts with the next word. The module descriptor data area pointer field contains the address of the called module data area.

If the access change requires a change of stack pointer, the old value is saved in the module stack frame, and the new value is taken from the module descriptor stack pointer field. Any further information in the module descriptor is user defined.

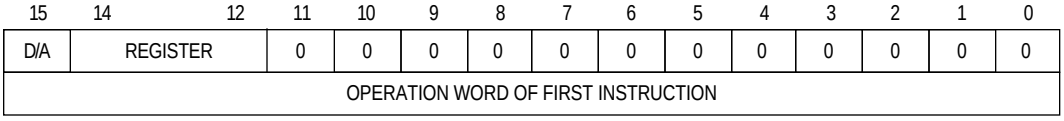


Figure 9-11. Module Entry Word