Welcome to E-XFL.COM

Understanding Embedded - Microprocessors

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

Applications of Embedded - Microprocessors

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

Details	
Product Status	Obsolete
Core Processor	68030
Number of Cores/Bus Width	1 Core, 32-Bit
Speed	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	132-BCQFP
Supplier Device Package	132-CQFP (24x24)
Purchase URL	https://www.e-xfl.com/pro/item?MUrl=&PartUrl=mc68030fe25c

TABLE OF CONTENTS (Continued)

PREFACE

The *MC68030 User's Manual* describes the capabilities, operation, and programming of the MC68030 32-bit second-generation enhanced microprocessor. The manual consists of the following sections and appendix. For detailed information on the MC68030 instruction set refer to M68000PM/AD, *M68000 Family Programmer's Reference Manual.*

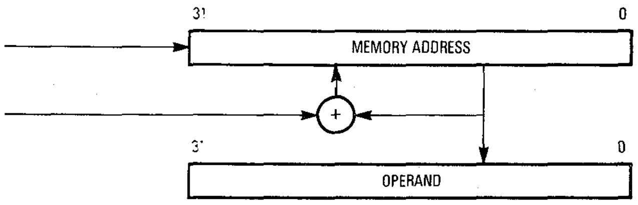
NOTE

In this manual, assertion and negation are used to specify forcing a signal to a particular state. In particular, assertion and assert refer to a signal that is active or true; negation and negate indicate a signal that is inactive or false. These terms are used independently of the voltage level (high or low) that they represent.

The audience of this manual includes systems designers, systems programmers, and applications programmers. Systems designers need some knowledge of all sections, with particular emphasis on Sections 1, 5, 6, 7, 13, 14, and Appendix A. Designers who implement a coprocessor for their system also need a thorough knowledge of Section 10. Systems programmers should

GENERATION:	EA = (An)
	An = An + SIZE

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

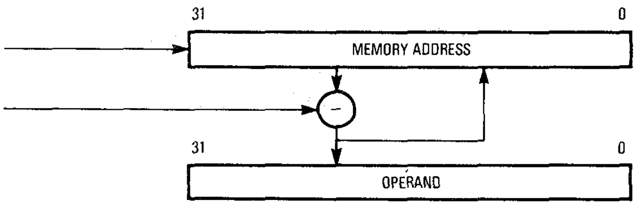


2.4.5 Address Register Indirect with Predecrement Mode

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

GENERATION:	An = An – SIZE
	EA = (An)

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



X	= extend (X) bit in CCR
N	= negative (N) bit in CCR
Z	= Zero (Z) bit in CCR
V	= overflow (V) bit in CCR
C	= carry (C) bit in CCR
+	= arithmetic addition or postincrement indicator
−	= arithmetic subtraction or predecrement indicator
×	= arithmetic multiplication
÷	= arithmetic division or conjunction symbol
~	= invert; operand is logically complemented
Λ	= logical AND
V	= logical OR
⊕	= logical exclusive OR
Dc	= data register, D7–D0 used during compare
Du	= data register, D7–D0 used during update
Dr, Dq	= data registers, remainder or quotient of divide
Dh, Dl	= data registers, high- or low-order 32 bits of product
MSW	= most significant word
LSW	= least significant word
MSB	= most significant bit
FC	= function code
{R/W}	= read or write indicator
[An]	= address extensions

3.2.1 Data Movement Instructions

The MOVE instructions with their associated addressing modes are the basic means of transferring and storing addresses and data. MOVE instructions transfer byte, word, and long-word operands from memory to memory, memory to register, register to memory, and register to register. Address movement instructions (MOVE or MOVEA) transfer word and long-word operands and ensure that only valid address manipulations are executed. In addition to the general MOVE instructions, there are several special data movement instructions: move multiple registers (MOVEM), move peripheral data (MOVEP), move quick (MOVEQ), exchange registers (EXG), load effective address (LEA), push effective address (PEA), link stack (LINK), and unlink stack (UNLK).

3.3.2 Conditional Tests

Table 3-13 lists the condition names, encodings, and tests for the conditional branch and set instructions. The test associated with each condition is a logical formula using the current states of the condition codes. If this formula evaluates to one, the condition is true. If the formula evaluates to zero, the condition is false. For example, the T condition is always true, and the EQ condition is true only if the Z bit condition code is currently true.

Table 3-13. Conditional Tests

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

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

*Not available for the Bcc instruction.

6.1.2.1 WRITE ALLOCATION. The supervisor program can configure the data cache for either of two types of allocation for data cache entries that miss on write cycles. The state of the write allocation (WA) bit in the cache control register specifies either no write allocation or write allocation with partial validation of the data entries in the cache on writes.

When no write allocation is selected (WA = 0), write cycles that miss do not alter the data cache contents. In this mode, the processor does not replace entries in the cache during write operations. The cache is updated only during a write hit.

When write allocation is selected (WA = 1), the processor always updates the data cache on cachable write cycles, but only validates an updated entry that hits or an entry that is updated with long-word data that is long-word aligned. When a tag miss occurs on a write of long-word data that is long-word aligned, the corresponding tag is replaced, and only the long word being written is marked as valid. The other three entries in the cache line are invalidated when a tag miss occurs on a misaligned long-word write or on a byte or word write, the data is not written in the cache, the tag is unaltered, and the valid bit(s) are cleared. Thus, an aligned long-word data write may replace a previously valid entry; whereas, a misaligned data write or a write of data that is not long word may invalidate a previously valid entry or entries.

Write allocation eliminates stale data that may reside in the cache because of either of two unique situations: multiple mapping of two or more logical addresses to one physical address within the same task or allowing the same physical location to be accessed by both supervisor and user mode cycles. Stale data conditions can arise when operating in the no-write-allocation mode and all the following conditions are satisfied:

- Multiple mapping (object aliasing) is allowed by the operating system.
- A read cycle loads a value for an "aliased" physical address into the data cache.
- A write cycle occurs, referencing the same aliased physical object as above but using a different logical address, causing a cache miss and no update to the cache (has the same page offset).
- The physical object is then read using the first alias, which provides stale data from the cache.

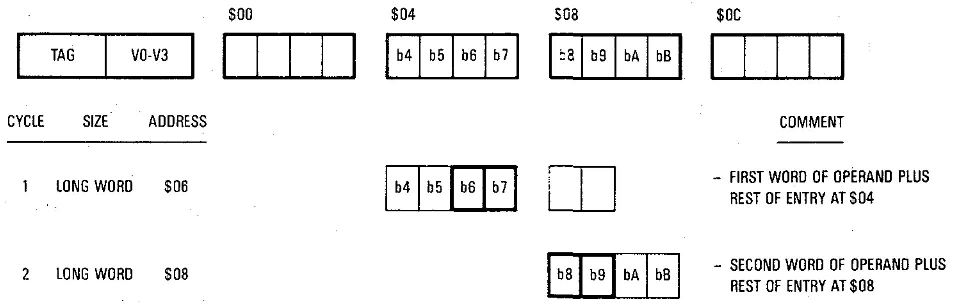

$00
$04
$08
$0C
TAG
V0-V3
b4
b5
b6
b7
b8
b9
bA
bB
CYCLE
SIZE
ADDRESS
COMMENT
1
LONG WORD
$06
– FIRST WORD OF OPERAND PLUS REST OF ENTRY AT $04
2
LONG WORD
$08
– SECOND WORD OF OPERAND PLUS REST OF ENTRY AT $08

Figure 6-10. Single Entry Mode Operation — Misaligned Long Word and 32-Bit DSACKx Port

If all bytes of a long word are cachable, CIIN must be negated for all bus cycles required to fill the entry. If any byte is not cachable, CIIN must be asserted for all corresponding bus cycles. The assertion of the CIIN signal prevents the caches from being updated during read cycles. Write cycles (including the write portion of a read-modify-write cycle) ignore the assertion of the CIIN signal and may cause the data cache to be altered, depending on the state of the cache (whether or not the write cycle hits), the state of the WA bit in the CACR, and the conditions indicated by the MMU.

The occurrence of a bus error while attempting to load a cache entry aborts the entry fill operation but does not necessarily cause a bus error exception. If the bus error occurs on a read cycle for a portion of the required operand (not the remaining bytes of the cache entry) to be loaded into the data cache, the processor immediately takes a bus error exception. If the read cycle in error is made only to fill the data cache (the data is not part of the target operand), no exception occurs, but the corresponding entry is marked invalid. For the instruction cache, the processor marks the entry as invalid, but only takes an exception if the execution unit attempts to use the instruction word(s).

6.1.3.2 BURST MODE FILLING. Burst mode filling is enabled by bits in the cache control register. The data burst enable bit must be set to enable burst filling of the data cache. Similarly, the instruction burst enable bit must be set to enable burst filling of the instruction cache. When burst filling is enabled and the corresponding cache is enabled, the bus controller requests a burst mode fill operation in either of these cases:

- A read cycle for either the instruction or data cache misses due to the indexed tag not matching.
- A read cycle tag matches, but all long words in the line are invalid.

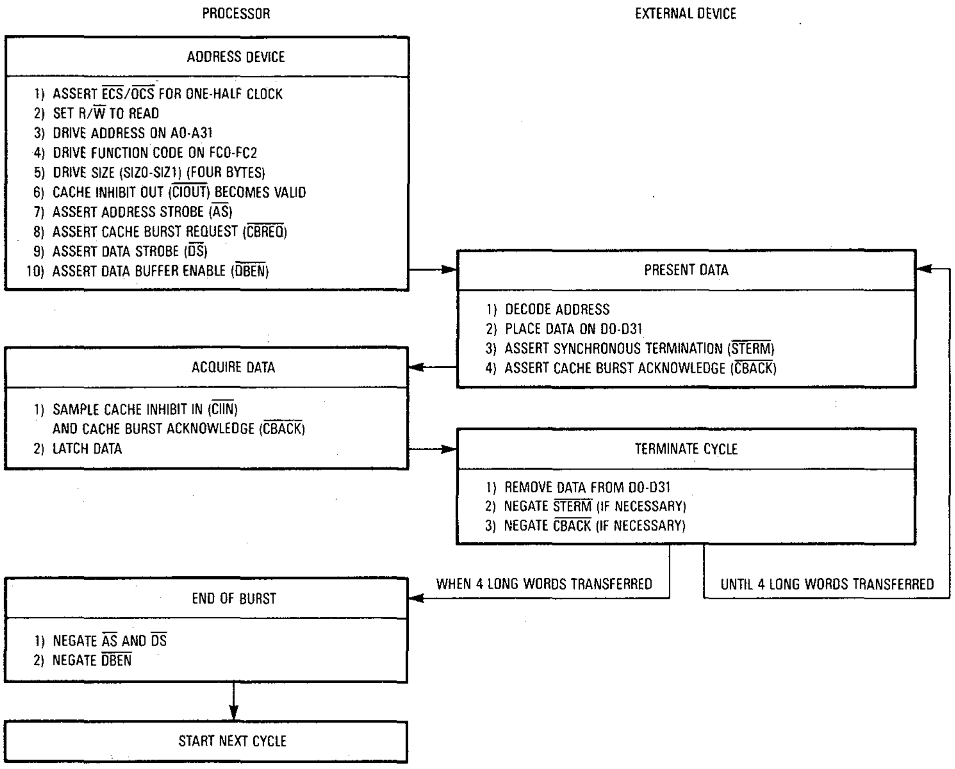
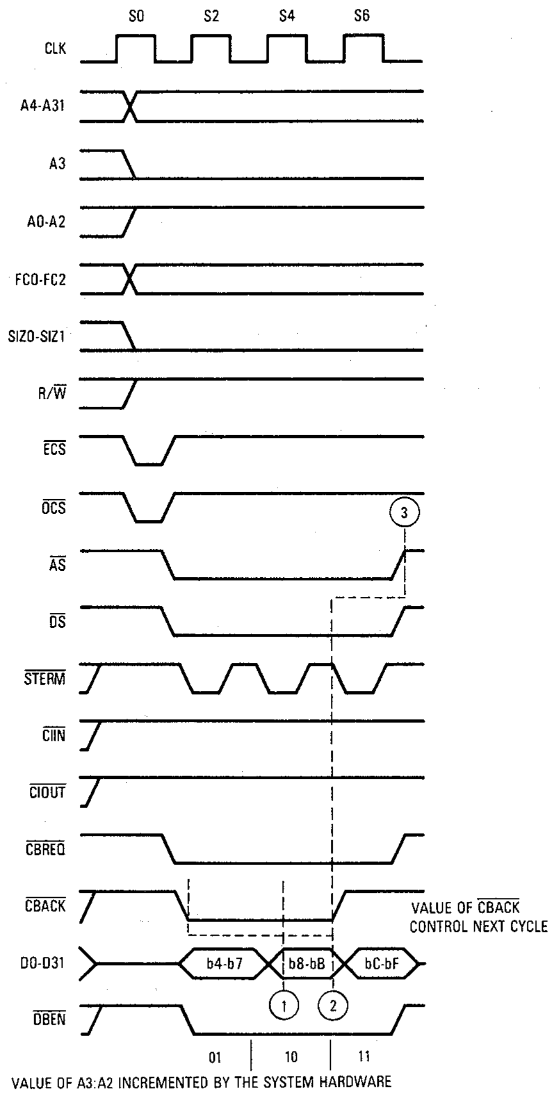

PROCESSOR
EXTERNAL DEVICE
ADDRESS DEVICE
1) ASSERT ECS/OCS FOR ONE-HALF CLOCK
2) SET R/W TO READ
3) DRIVE ADDRESS ON A0-A31
4) DRIVE FUNCTION CODE ON FC0-FC2
5) DRIVE SIZE (SIZ0-SIZ1) (FOUR BYTES)
6) CACHE INHIBIT OUT (CIOUT) BECOMES VALID
7) ASSERT ADDRESS STROBE (AS)
8) ASSERT CACHE BURST REQUEST (CBREQ)
9) ASSERT DATA STROBE (DS)
10) ASSERT DATA BUFFER ENABLE (DBEN)
PRESENT DATA
1) DECODE ADDRESS
2) PLACE DATA ON D0-D31
3) ASSERT SYNCHRONOUS TERMINATION (STERM)
4) ASSERT CACHE BURST ACKNOWLEDGE (CBACK)
ACQUIRE DATA
1) SAMPLE CACHE INHIBIT IN (CIIN) AND CACHE BURST ACKNOWLEDGE (CBACK)
2) LATCH DATA
TERMINATE CYCLE
1) REMOVE DATA FROM D0-D31
2) NEGATE STERM (IF NECESSARY)
3) NEGATE CBACK (IF NECESSARY)
WHEN 4 LONG WORDS TRANSFERRED
UNTIL 4 LONG WORDS TRANSFERRED
END OF BURST
1) NEGATE AS AND DS
2) NEGATE DBEN
START NEXT CYCLE

Figure 7-37. Burst Operation Flowchart — Four Long Words Transferred

State 2

The selected device uses R/$\overline{\text{W}}$, SIZ0–SIZ1, A0–A1, and $\overline{\text{CIOUT}}$ to place the data on the data bus. (The first cycle must supply the long word at the corresponding long-word boundary.) All of the byte sections (D24–D31, D16–D23, D8–D15, and D0–D7) of the data bus must be driven since the burst operation latches 32 bits on every cycle. During S2, the processor drives $\overline{\text{DBEN}}$ active to enable external data buffers. In systems that use two-clock synchronous bus cycles, the timing of $\overline{\text{DBEN}}$ may prevent its use. At the beginning of S2, the processor tests the level of $\overline{\text{STERM}}$. If $\overline{\text{STERM}}$ is recognized, the processor latches the incoming data at the end of S2. For the burst operation to proceed, $\overline{\text{CBACK}}$ must be asserted when $\overline{\text{STERM}}$ is recognized. If the data for the current cycle is not to be cached, $\overline{\text{CIIN}}$ must be asserted at the same time as $\overline{\text{STERM}}$. The assertion of $\overline{\text{CIIN}}$ also has the effect of aborting the burst operation.



NOTES:
1. Assertion of $\overline{CBACK}$ causes data to be placed on D0-D31.
2. Continued assertion of $\overline{CBACK}$ causes data to be placed on D0-D31.
3. Negation of $\overline{CBACK}$ cause $\overline{AS}$ to be negated.

Figure 7-39. Long-Word Operand Request from \$07 with Burst Request — $\overline{CBACK}$ Negated Early

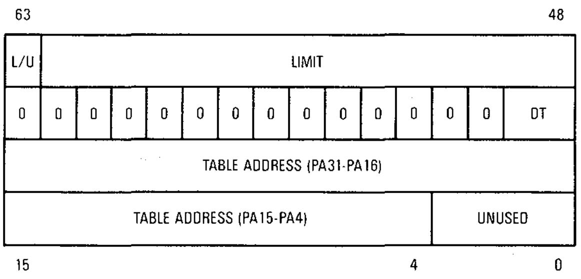
- 0–15 Upper Logical Address Bits Can Be Ignored (Using Initial Shift)
- Portions of Tables Can Be Undefined (Using Limits)
- Write Protection and Supervisor Protection
- History Bits Automatically Maintained in Page Descriptors
- Cache Inhibit Output ($\overline{\text{CIOUT}}$) Signal Asserted on Page Basis
- External Translation Disable Input Signal ($\overline{\text{MMUDIS}}$)
- Subset of Instruction Set Defined by MC68851

The MMU completely overlaps address translation time with other processing activity when the translation is resident in the ATC. ATC accesses operate in parallel with the on-chip instruction and data caches.

Figure 9-1 is a block diagram of the MC68030 showing the relationship of the MMU to the execution unit and the bus controller. For an instruction or operand access, the MC68030 simultaneously searches the caches and searches for a physical address in the ATC. If the translation is available, the MMU provides the physical address to the bus controller and allows the bus cycle to continue. When the instruction or operand is in either of the on-chip caches on a read cycle, the bus controller aborts the bus cycle before address strobe is asserted. Similarly, the MMU causes a bus cycle to abort before the assertion of address strobe when a valid translation is not available in the ATC or when an invalid access is attempted.

An MMU disable input signal ($\overline{\text{MMUDIS}}$) is provided that dynamically disables address translation for emulation, diagnostic, or other purposes.

The programming model of the MMU (see Figure 9-2) consists of two root pointer registers, a control register, two transparent translation registers, and a status register. These registers can only be accessed by supervisor programs. The CPU root pointer register points to an address translation tree structure in memory that describes the logical-to-physical mapping for user accesses or for both user and supervisor accesses. The supervisor root pointer register optionally points to an address translation tree structure for supervisor mappings. The translation control register is comprised of fields that control the translation operation. Each transparent translation register can define a block of logical addresses that are used as physical addresses (without translation). The MMU status register contains accumulated status information from a translation performed as a part of a PTEST instruction.



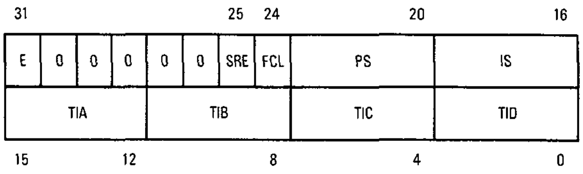
L/U — LOWER OR UPPER PAGE RANGE
DT — DESCRIPTOR TYPE
LIMIT — LIMIT ON TABLE INDEX FOR THIS TABLE ADDRESS
TABLE ADDRESS — ADDRESS OF TABLE AT NEXT LEVEL OR PAGE OFFSET IF DT = 1

Figure 9-35. Root Pointer Register (CRP, SRP) Format

9.7.2 Translation Control Register

The translation control register (TC) is a 32-bit register that contains the control fields for address translation. All unimplemented fields of this register are read as zeros and must always be written as zeros.

Writing to this register optionally causes a flush of the entire ATC. When written with the E bit (bit 31) set (translation enabled), a consistency check is performed on the values of PS, IS, and TIx as follows. The TIx fields are added together until a zero field is reached, and this sum is added to PS and IS. The total must be 32, or an MMU configuration exception (refer to **9.7.5.3 MMU CONFIGURATON EXCEPTION**) is taken. If an MMU configuration exception occurs, the TC register is updated with the data, and the E bit is cleared. The translation control register is shown in Figure 9-36.



E — ENABLE
SRE — SUPERVISOR ROOT POINTER ENABLE
FCL — FUNCTION CODE LOOKUP ENABLE
PS — PAGE SIZE
IS — INITIAL SHIFT
TIA, TIB, TIC, TID — TABLE INDICES

Figure 9-36. Translation Control Register (TC) Format

An operating system can use an early termination page descriptor to map a contiguous block of memory for each task (both program and data). The tasks can be relocated by changing the physical address portion of the descriptor. This scheme is useful when the tasks in a system consist of one or a few sequential blocks of memory that can be swapped as a group. The operating system memory map can treat the entire address space within these blocks as a uniform virtual space available for all tasks. The system only requires one translation table; by the use of limit fields and early termination page descriptors, it maps complete segments of memory.

9.9.3.5 INDIRECT DESCRIPTORS. An indirect descriptor is a table descriptor residing in a page table. It points to another page descriptor in the translation tree. Using an indirect descriptor for a page makes the page common to several tasks. History information for a common page is maintained in only one descriptor. Access to the page sets the used (U) bit, and a write operation to the page sets the M (modified) bit for that page. When the operating system is searching for an available page, it simply checks the page table containing the descriptor for the common page to determine its status. With other methods of page sharing, the system would have to check page tables for all sharing tasks to determine the status of the common page.

9.9.3.6 USING UNUSED DESCRIPTOR BITS. In general, the bits in the unused fields of many types of descriptors are available to the operating system for its own purposes. The invalid descriptor, in particular, uses only two bits of the 32 (short) or 64 (long) bits available with that format. An operating system typically uses these fields for the software flags, indicating whether the virtual address space is allocated and whether an image resides on the paging device. Also, these fields often contain the physical address of the image.

The operating system often maintains information in an unused field about a page resident in memory. This information may be an aging counter or some other indication of the page's frequency of use. This information helps the operating system to identify the pages that are least likely to impact system performance if they are reallocated. The system should first use physical page frames that are not allocated to a virtual page. Next it should use pages with the longest time since the most recent access. Pages that do not have the M (modified) bit set should be taken first, since they do not need to be copied to the paging device (the existing image remains valid).

the table size). By convention, the first entry maps the supervisor address space and has supervisor protection. The routine never modifies this first entry. The 31 entries after the first are available to be allocated as user address space.

A routine similar to this that linearly extends (grows) a previously allocated memory block could be written. A stack is a good example. The operating system can allocate the top of the memory (the thirty-second upper level table entry) as a stack that grows downward from the highest address. If a task needs several large stacks, a 16-Mbyte block can be used for each stack, with a software flag set to indicate growth in a downward direction.

The logic of Vallocate is:

1. Validate the request and calculate number of pages required.
2. Scan each upper table entry's lower page tables (where they exist) looking for an adequate group of unallocated pages.
3. If no space is found, see if the lower table is less than its maximum size and if the block can be allocated by expanding it at the end.
4. If still no space is found, use the next free upper table entry and initialize its new lower level page table to allocate the block here.
5. Set allocated page entries to indicate virgin status (allocated, invalid, and not swapped out).
6. Return status. If status is OK, also return virtual address.

The code for Vallocate is:

```
Vallocate (SizeInBytes, VirtualAddressReturned, Status);

/* The following are global to all routines                                              */

/* Symbolicly define the upper level pointer table                                       */

Declare Upper_Table[32] Record of
                Status=(unallocated, allocated),     /* lower table here or not          */
                Limit_Field=(0 to 4k),               /* limit for lower page table       */
                Pointer;                             /*address of lower page table if allocated */

/* Symbolicly define the lower level page table                                          */

Declare Lower_Table[0 to Limit_Field] Based Record of
                Status=(invalid_unallocated,         /*not allocated to User             */
                        invalid_paged_out,           /*allocated but paged out           */
                        invalid_virgin,              /*allocated but not yet used        */
                        valid_in_memory),            /*allocated and in memory           */
                Pointer;                             /*physical address or disk address of page */
```

10.4.1 ScanPC

Several of the response primitives involve the scanPC, and many of them require the main processor to use it while performing services requested. These paragraphs describe the scanPC and tell how it operates.

During the execution of a coprocessor instruction, the program counter in the MC68030 contains the address of the F-line operation word of that instruction. A second register, called the scanPC, sequentially addresses the remaining words of the instruction.

If the main processor requires extension words to calculate an effective address or destination address of a branch operation, it uses the scanPC to address these extension words in the instruction stream. Also, if a coprocessor requests the transfer of extension words, the scanPC addresses the extension words during the transfer. As the processor references each word, it increments the scanPC to point to the next word in the instruction stream. When an instruction is completed, the processor transfers the value in the scanPC to the program counter to address the operation word of the next instruction.

The value in the scanPC when the main processor reads the first response primitive after beginning to execute an instruction depends on the instruction being executed. For a cpGEN instruction, the scanPC points to the word following the coprocessor command word. For the cpBcc instructions, the scanPC points to the word following the instruction F-line operation word. For the cpScc, cpTRAPcc, and cpDBcc instructions, the scanPC points to the word following the coprocessor condition specifier word.

If a coprocessor implementation uses optional instruction extension words with a general or conditional instruction, the coprocessor must use these words consistently so that the scanPC is updated accordingly during the instruction execution. Specifically, during the execution of general category instructions, when the coprocessor terminates the instruction protocol, the MC68030 assumes that the scanPC is pointing to the operation word of the next instruction to be executed. During the execution of conditional category instructions, when the coprocessor terminates the instruction protocol, the MC68030 assumes that the scanPC is pointing to the word following the last of any coprocessor-defined extension words in the instruction format.

tions. If the coprocessor issues this primitive during the execution of a conditional category instruction, the main processor initiates protocol violation exception processing. Figure 10-37 shows the format of the transfer multiple coprocessor registers primitive.

15	14	13	12	11	10	9	8	7 — 0
CA	PC	DR	0	0	0	0	1	LENGTH

Figure 10-37. Transfer Multiple Coprocessor Registers Primitive Format

This primitive uses the CA, PC, and DR bits as previously described.

Bits [7–0] of the primitive format indicate the length in bytes of each operand transferred. The operand length must be an even number of bytes; odd length operands cause the MC68030 to initiate protocol violation exception processing (refer to **10.5.2.1 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 MC68030 to abort the instruction by writing an abort mask (refer to **10.3.2 Control CIR**) to the control CIR and to initiate F-line emulator exception processing (refer to **10.5.2.2 F-LINE EMULATOR EXCEPTIONS**).

After performing the effective address calculation, the MC68030 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 MC68030 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 bits [0–7] of the primitive format.

Table 10-6. Exceptions Related to Primitive Processing (Sheet 2 of 2)

Primitive	Protocol	F-Line	Other
Transfer Status and/or ScanPC Protocol: If Used with Conditional Instruction Other: 1. Trace — Trace Made Pending if MC68020 in "Trace on Change of Flow" Mode and DR = 1 2. Address Error — If Odd value Written to ScanPC	X		X
Take Pre-Instruction, Mid-Instruction, or Post-Instruction Exception Exception Depends on Vector Supplies in Primitive	X	X	X

*Use of this primitive with CA = 0 will cause protocol violation on conditional instructions.

Abbreviations:
EA = Effective Address
CP = Coprocessor

When the MC68030 detects a protocol violation, it does not automatically notify the coprocessor of the resulting exception by writing to the control CIR. The exception handling routine may, however, use the MOVES instruction to read the response CIR and thus determine the primitive that caused the MC68030 to initiate protocol violation exception processing. The main processor initiates exception processing using the mid-instruction stack frame (refer to Figure 10-43) 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.

11.6.1 Fetch Effective Address (fea)

The fetch effective address table indicates the number of clock periods needed for the processor to calculate and fetch the specified effective address. The effective addresses are divided by their formats (refer to **2.5 Effective Address Encoding Summary**). For instruction-cache case and for no-cache case, 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). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
SINGLE EFFECTIVE ADDRESS INSTRUCTION FORMAT				
% Dn	—	—	0(0/0/0)	0(0/0/0)
% An	—	—	0(0/0/0)	0(0/0/0)
(An)	1	1	3(1/0/0)	3(1/0/0)
(An)+	0	1	3(1/0/0)	3(1/0/0)
−(An)	2	2	4(1/0/0)	4(1/0/0)
(d_{16},An) or (d_{16},PC)	2	2	4(1/0/0)	4(1/1/0)
(xxx).W	2	2	4(1/0/0)	4(1/1/0)
(xxx).L	1	0	4(1/0/0)	5(1/1/0)
#(data).B	2	0	2(0/0/0)	2(0/1/0)
#(data).W	2	0	2(0/0/0)	2(0/1/0)
#(data).L	4	0	4(0/0/0)	4(0/1/0)
BRIEF FORMAT EXTENSION WORD				
(d_8,An,Xn) or (d_8,PC,Xn)	4	2	6(1/0/0)	6(1/1/0)

11.6.3 Calculate Effective Address (cea) (Continued)

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
SINGLE EFFECTIVE ADDRESS INSTRUCTION FORMAT				
% Dn	—	—	**0**(0/0/0)	**0**(0/0/0)
% An	—	—	**0**(0/0/0)	**0**(0/0/0)
(An)	2 + op head	0	**2**(0/0/0)	**2**(0/0/0)
(An) +	0	0	**2**(0/0/0)	**2**(0/0/0)
– (An)	2 + op head	0	**2**(0/0/0)	**2**(0/0/0)
(d_{16},An) or (d_{16},PC)	2 + op head	0	**2**(0/0/0)	**2**(0/1/0)
(xxx).W	2 + op head	0	**2**(0/0/0)	**2**(0/1/0)
(xxx).L	4 + op head	0	**4**(0/0/0)	**4**(0/1/0)
BRIEF FORMAT EXTENSION WORD				
(d_8,An,Xn) or (d_8,PC,Xn)	4 + op head	0	**4**(0/0/0)	**4**(0/1/0)
FULL FORMAT EXTENSION WORD(S)				
(d_{16},An) or (d_{16},PC)	2	0	**6**(0/0/0)	**6**(0/1/0)
(d_{16},An,Xn) or (d_{16},PC,Xn)	*6 + op head*	*0*	***6**(0/0/0)*	***6**(0/1/0)*
([d_{16},An]) or ([d_{16},PC])	2	0	**10**(1/0/0)	**10**(1/1/0)
([d_{16},An],Xn) or ([d_{16},PC],Xn)	2	0	**10**(1/0/0)	**10**(1/1/0)
([d_{16},An],d_{16}) or ([d_{16},PC],d_{16})	2	0	**12**(1/0/0)	**13**(1/2/0)
([d_{16},An],Xn,d_{16}) or ([d_{16},PC],Xn,d_{16})	2	0	**12**(1/0/0)	**13**(1/2/0)
([d_{16},An],d_{32}) or ([d_{16},PC],d_{32})	2	0	**12**(1/0/0)	**13**(1/2/0)
([d_{16},An],Xn,d_{32}) or ([d_{16},PC],Xn,d_{32})	2	0	**12**(1/0/0)	**13**(1/2/0)
(B)	6 + op head	0	**6**(0/0/0)	**6**(0/1/0)
(d_{16},B)	4	0	**8**(0/0/0)	**9**(0/1/0)
(d_{32},B)	4	0	**12**(0/0/0)	**12**(0/2/0)
([B])	4	0	**10**(1/0/0)	**10**(1/1/0)
([B],I)	4	0	**10**(1/0/0)	**10**(1/1/0)
([B],d_{16})	4	0	**12**(1/0/0)	**13**(1/1/0)
([B],I,d_{16})	4	0	**12**(1/0/0)	**13**(1/1/0)
([B],d_{32})	4	0	**12**(1/0/0)	**13**(1/2/0)
([B],I,d_{32})	4	0	**12**(2/0/0)	**13**(1/2/0)
([d_{16},B])	4	0	**12**(1/0/0)	**13**(1/1/0)
([d_{16},B],I)	4	0	**12**(1/0/0)	**13**(1/1/0)
([d_{16},B],d_{16})	4	0	**14**(1/0/0)	**16**(1/2/0)
([d_{16},B],I,d_{16})	4	0	**14**(1/0/0)	**16**(1/2/0)

INDEX