

INTEL

80186/80286/80386

This chapter describes the internal architecture, addressing modes, instruction set, and I/O techniques associated with the 80186, 80286, and 80386 microprocessors. Interfacing capabilities to typical memory and I/O chips are also included. Finally, virtual memory concepts associated with the 80286 and 80386 are covered.

4.1 Intel 80186 and 80286

This section covers the two enhanced versions of the 8086 microprocessor: Intel 80186 and 80286. The Intel 80186 includes the Intel 8086 and six separate functional units in a single chip, while the 80286 has integrated memory protection and management into the basic 8086 architecture.

4.1.1 Intel 80186

The Intel 80186 family is fabricated using HMOS technology and contains two microprocessors: Intel 80186 and 80188. The only difference between them is that the 80186 has a 16-bit data bus, while the 80188 includes an 8-bit data bus. The 80186 is packaged in a 68-pin leadless package. The 80186 can be operated at three different clock speeds: 8 MHz, 10 MHz, 12.5 MHz, and other frequencies. The 80C186 and 80C188 are the Low-Power (HCMOS) versions of the 80186 and 80188 respectively. The 80C186 and the 80186 can be operated at the same frequencies while the 80C188, like the 80188 can be operated at 8- or 10-MHz. The 80186 can directly address one megabyte of memory. It contains the 8086 microprocessor and several additional functional units. The major on-chip circuits include a clock generator, two independent DMA channels, a programmable interrupt controller, three programmable 16-bit timers, and a chip select unit.

The 80186 provides double the performance of the standard 8086. The 80186 includes 10 new instructions beyond the 8086. The 80186 is completely object code compatible with the 8086. It contains all the 8086 registers and generates the 20-bit physical address from a 16-bit segment register and a 16-bit offset in the same way as the 8086. The 80186 does not have the 8086's $\overline{MN}/\overline{MX}$ pin. The 80186 has enough pins to generate the minimum mode-type pins. $S0$ - $S3$ status signals can be connected to external bus controller chips such as 8288 for generating the maximum mode type signals. Like the 8086, the 80186 fetches the first instruction from physical address $FFFF0H$ upon hardware reset.

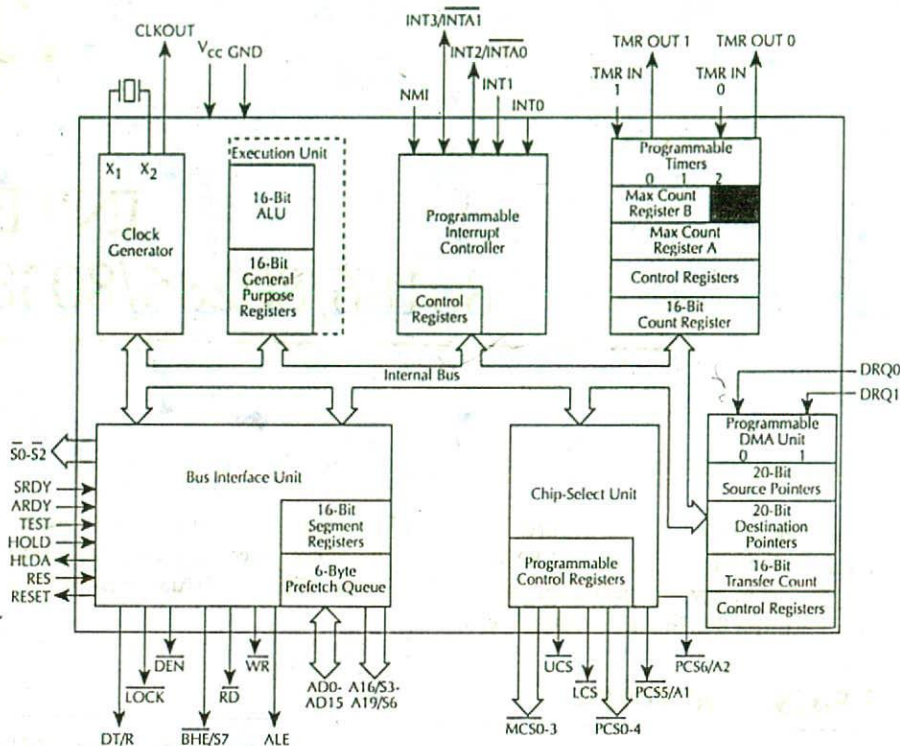


FIGURE 4.1 80186 functional block diagram.

Figure 4.1 shows the 80186 functional block diagram. The DMA unit provides two DMA request input signals, DRQ0 and DRQ1. One of these input signals can be used by external devices such as disk controller to request a data transfer between the memory and disk via direct memory access technique. Each 80186 DMA channel contains two 20-bit registers and a 16-bit counter. One of the 20-bit registers holds the destination address of the DMA transfers while the 16-bit counter stores the number of words or bytes to be transferred. DMA transfers can take place between memory and I/O or from memory to memory or from I/O to I/O. The DMA channels may be programmed such that one channel has priority over the other. DMA transfers will take place whenever the ST/STOP bit in the control register is set to one.

The 80186 contains three independent 16-bit timers/counters namely counter 0, 1, and 2. Counters 0 and 1 can be programmed to count external events. The inputs and outputs of these two counters are available on the 80186 pins. The third timer, counter 2, is not connected to any external pins. This timer only counts the 80186 clock cycles. Counter 2 is decremented every four 80186 clocks. One can connect the output of counter 2 to a DMA unit or to an interrupt input or to the input of counter 1 and/or 0 by setting or clearing the appropriate bits in a control word. Counter 2 can, therefore, be used to interrupt the 80186 after a programmed amount of time or to provide a pulse to the DMA unit after a specific amount of time.

The priorities of the four interrupt pins, INT0, INT1, INT2/INTA0 and INT3/INTA1 are programmable. If these four interrupt inputs are programmed in their internal mode, then the activation of one of them by an external signal will cause the 80186 to push the return address on the stack and vector directly to the interrupt address vector for that interrupt. The INT2/INTA0 and INT3/INTA1 pins have dual functions. They can be programmed as interrupt inputs or as interrupt acknowledge output signals (INTA0 for INT0 and INTA1 for INT1). These interrupt pins can be used to connect the 80186 to an external priority interrupt controller such as the 8259A. For example, suppose that the interrupt request line from an

external 8259A is connected to the 80186 INT1 input pin and the 80186 INT3/INTA1 pin is connected to the 8259A interrupt acknowledge input pin. When the 8259A receives an interrupt request, it activates the 80186 INT1 pin. In response, the 80186 sends the interrupt acknowledge signal via its INT3/INTA1 pin. The 8259A then places the desired interrupt type code on the 80186 data bus. The 80186 obtains the CS and IP values based on the type code and branches to the service routine.

The 80186 interrupt controller allows the 80186 to receive interrupts from internal or external sources. Internal interrupt sources (timers and DMA channels) can be disabled by their own control registers or by mask bits within the interrupt controller. The 80186 is provided with five dedicated pins for external interrupts. These pins are NMI, INT0, INT1, INT2/INTA0, and INT3/INTA1. NMI is the only nonmaskable interrupt.

In the master mode, the interrupt controller provides three modes of operation. These are fully nested mode, cascade mode, and special fully nested mode. In the fully nested mode, all four maskable interrupt pins are used as direct interrupt requests. The interrupt vectors are obtained by the 80186 internally.

Upon acceptance of an interrupt (hardware, INT instructions, or instruction exceptions such as divide by 0), the 80186 pushes CS, IP, and the status word onto the stack just like the 8086. Also, similar to the 8086, an interrupt pointer table with 256 entries provides interrupt address vectors (IP and CS) for each interrupt type. This type identifies the appropriate table entry. Nonmaskable interrupts use an internally supplied type, while the types for maskable interrupts are provided by the user via external hardware. The types for INT instructions and instruction exceptions are generated internally by the 80186.

The 80186 includes an on-chip clock generator/crystal oscillator circuit. Like the 8085, a crystal connected at the 80186 X1 X2 pins is divided by 2 internally. The built-in chip select unit is an address decoder. This unit can be programmed to generate six memory chip selects (LCS, UCS, and MCS0 — 3 pins) and seven I/O or peripheral chip selects (PCS0-4, PCS5/ A1, and PCS6/A2 pins). This unit can be programmed to generate an active low chip select when a memory or port address in a particular range is sent out. For example, the 80186 outputs low on the LCS (lower chip select) pin when it accesses an address between 00000H and a higher address (in the range of 1K to 256K) programmable by the user via a control word. On the other hand, the 80186 outputs low on the UCS (upper chip select) pin when it accesses an address between a user programmable lower address (by placing some bits in a control word via an instruction) and upper fixed address FFFFFH. The four middle chip select pins (MCS0-3) are activated low by the 80186 when it accesses an address in the mid range. Both the starting address and the size of the four blocks can be specified via the control word. The specified size of blocks can be from 2K to 128K. The memory areas assigned to different chip selects must not overlap; otherwise, two chip selects will be asserted and bus contention will occur. The built-in decoder allows the 80186 to select large memory blocks. For peripheral chip selects, a base address can be programmed via a control word. The 80186 sends low on the PCS0 when it accesses a port address located in a block from this base address to up to 128 bytes. The 80186 sends low on the other chip selects PCS1-6 when one of six contiguous 128-byte blocks above the block for PCS0 is selected. Like the 8086, memory for the 80186 is set up as odd (BHE = 0) and even (A0 = 0) memory banks.

The 80186 provides eight addressing modes. These include register, immediate, direct, register indirect (SI, DI, BX, or BP), based (BX or BP), indexed (SI or DI), based indexed, and based indexed with displacement modes.

Typical data types provided by the 80186 include signed integer, ordinal (unsigned binary number), pointer, string, ASCII, unpacked/packed BCD, and floating point. The 80186 instruction set is divided into seven types. These are data transfer, arithmetic, shift/rotate, string, control transfer, high level instructions (for example, the BOUND instruction detects values outside prescribed range), and processor control. As mentioned before, the 80186 includes 10 new instructions beyond the 8086. These 10 additional instructions are listed below:

Data Transfer

PUSHA	— Push all registers onto stack
POPA	— Pop all registers from stack
PUSH immediate	— Push immediate numbers onto stack

Arithmetic

IMUL destination register, source, immediate data means immediate data*source. → destination register.

Logical

SHIFT/ROTATE destination, immediate data or CL shifts/rotates register or memory contents by the number of times specified in immediate data or by the contents of CL.

String Instructions

INSB or INSW	— Input string byte or string word
OUTSB or OUTSW	— Output string byte or string word

High Level Instructions

ENTER	— Format stack for procedure entry
LEAVE	— Restore stack for procedure exit
BOUND	— Detect values outside predefined range

Let us explain some of these instructions.

- IMUL destination register, source, immediate data. This is a signed multiplication. This instruction multiplies signed 8- or 16-bit immediate data with 8- or 16-bit data in a specified source register or memory location and places the result in a general-purpose destination register. As an example, IMUL DX, CX, -3 multiplies the contents of CX by -3 and places the lower 16-bit result in DX. Note that the immediate 8-bit data of -3 are sign extended to 16-bit prior to multiplication. A 32-bit result is obtained but only the lower 16-bit is saved by this instruction.
- ROL/ROR/SAL/SAR destination, immediate data or CL. Shift count can be specified by immediate data (one) or in CL up to a maximum of 32₁₀.
- INSB DX or INSW DX respectively inputs a byte or a word from a port addressed by DX to a memory location in ES pointed to by DI. If DF = 0, DI will automatically be incremented (by 1 for byte and 2 for word) after execution of this instruction. On the other hand, if DF = 1, DI is automatically decremented (by 1 for byte and 2 for word) after execution of this instruction. The instructions INSB for byte and INSW for word are used. A typical example of inputting 50 bytes of I/O data via a port into a memory location is given below (assume ES is already initialized):

```

STD          ; Set DF to 1.
LEA DI, ADDR ; Initialize DI.
MOV DX, 0E124H ; Load port address.
MOV CX, 50    ; Initialize count.
REP INSB DX   ; Input port until CX = 0.
STOP JMP STOP ; Halt.

```

- OUTSB DX or OUTSW DX respectively provides outputting to a port addressed by DX from a source string in DS with offset in SI.
- The ENTER instruction is used at the beginning of an assembly language subroutine which is to be called by a high level language program such as Pascal. The main purpose of ENTER is to reserve space on the stack for variables used in the subroutine.

The ENTER instruction has two immediate operands:

ENTER imm16, imm8

The first operand imm16 specifies the total memory area allocated to the local variables, which is 16 bits wide (0 to 64K bytes). The second operand imm8, on the other hand, is 8 bits wide and specifies the number of nested subroutines.

For the main subroutine, imm8 = 0. Note that nested subroutines mean a subroutine calling another subroutine. For example, if there are three subroutines SUB1, SUB2, and SUB3 such that the main program M calls SUB1, SUB1 calls SUB2 and SUB2 calls SUB3, then imm8 = 0 for SUB1, 1 for SUB2, and 2 for SUB3. ENTER can be used to allocate temporary stack space for local variables for each subroutine.

In the second operand, if imm8 = 0, the ENTER instruction pushes the frame pointer BP onto the stack. ENTER then subtracts the first operand imm16 from the stack pointer and sets the frame pointer, BP, to the current stack pointer value.

The LEAVE instruction is used at the end of each subroutine (usually before the RET instruction). The LEAVE does not have any operand. The LEAVE instruction should be used with the ENTER instruction. The ENTER allocates space in stack for variables used in the subroutine, while the LEAVE instruction deallocates this space and ensures that SP and BP have the original values that they had prior to execution of the ENTER. The RET instruction then returns to the appropriate address in the main program.

As an example of application of ENTER and LEAVE instructions, suppose that a subroutine requires 16 bytes of stack for local variables. The instructions ENTER 16, 0 at the subroutine's entry point and a LEAVE before the RET instruction will accomplish this. The 16 local bytes may be accessed.

When the 80186 accesses an array, the BOUND instruction can be used to ensure that data outside the array are not accessed. When the BOUND is executed, the 80186 compares the content of a general-purpose register (initialized by the user with the offset of the array element currently being accessed) with the lower and upper bounds of the array (loaded by the user prior to BOUND). The format for BOUND is BOUND reg16, memory32. The first operand is the register containing the array index and the second operand is a memory location containing the array bounds. If the index value violates the array bounds, an exception (maskable interrupt 5) takes place. A service routine can be executed by the user to indicate that the array element being accessed is out of bounds. As an example, consider BOUND SI, ADDR. The lower bound of the array is contained in address ADDR and the upper bound is in address ADDR + 2. Both bounds are 16 bits wide. For a valid access content of SI must be greater than or equal to the content of the memory location with offset ADDR and less than or equal to the contents of the memory location with offset ADDR + 2; otherwise interrupt 5 occurs. The BOUND instruction is normally placed just before the array itself, making the array addressable via a constant from the start of the array.

The BOUND instruction is normally placed following the computation of an offset value to ensure that the limits of the array boundaries are not violated. This permits checking whether or not the offset of an array being accessed is within the boundaries when the based addressing mode is used to access an element in the array. For example, the instruction segment shown below will allow accessing of an array with base address in BX and array length of 50₁₀ bytes:

```
MOV    DS:ADDR, 1000;
MOV    DS:ADDR+2, 1049;
BOUND  BX, ADDR;
MOV    CL, [BX];
```

In the above, it is assumed that the offset of the lowest array element is 1000. With an array length of 50, the offset of the highest array element is 1049. It is assumed that BX contains the

offset of the array element currently being worked on. The BOUND instruction checks whether the contents of BX is between 1000 and 1049. If it is, the MOV instruction accesses the desired array element into CL; otherwise type 5 interrupt is generated.

The 80186/80188 is used in embedded control. In these applications, the microcomputer performs a dedicated control function. Embedded control applications are divided into two types. These are event control and data control.

In embedded control applications involving event control, the microprocessor initiates a timed sequence of events. An example of such an application is the industrial process control.

In embedded control applications involving data control, the microprocessor transfers volumes of data to be processed from secondary memory such as disk to main memory.

The 80186/80188 is, therefore, highly integrated to satisfy the requirements of data control applications. The 80186/80188 is also provided with added features such as string I/O instructions and DMA channels to better handle fast movement of data.

4.1.2 Intel 80286

The Intel 80286 is a high-performance 16-bit microprocessor with on-chip memory protection capabilities primarily designed for multiuser/multitasking systems. The IBM PC/AT and its clones capable of multitasking operations use the 80286 as their CPU. The 80286 can address 16 megabytes (2^{24}) of physical memory and 1 gigabyte (2^{30}) of virtual memory per task. The 80286 can be operated at several different clock speeds. These are 8 MHz (80286-4), 10 MHz (80286-6), 12.5 MHz, 16.67 MHz, and 20 MHz (80286).

The 80286 has two modes of operations. These are real address mode and protected virtual address mode (PVAM). In the real address mode, the 80286 is object code compatible with the Intel 8086/8088/80186/80188. In protected virtual address mode, the 80286 is source code compatible with the iAPX 86/88 family and may require some software modification to use virtual address features of the 80286. Note that the protected virtual address mode is not used by PC DOS.

The 80286 includes special instructions to support operating systems. For example, one instruction can end a current task execution, save its state, switch to a new task, load its state, and begin executing the new task.

The 80286's performance is up to six times faster than the standard 5-MHz 8086. The 80286 is housed in a 68-pin leadless flat package. Figure 4.2 shows a functional diagram of the 80286. It contains four separate processing units. These are the Bus Unit (BU), the Instruction Unit (IU), the Address Unit (AU), and the Execution Unit (EU). The BU provides all memory and I/O read and write operations. The BU also performs data transfer between the 80286 and

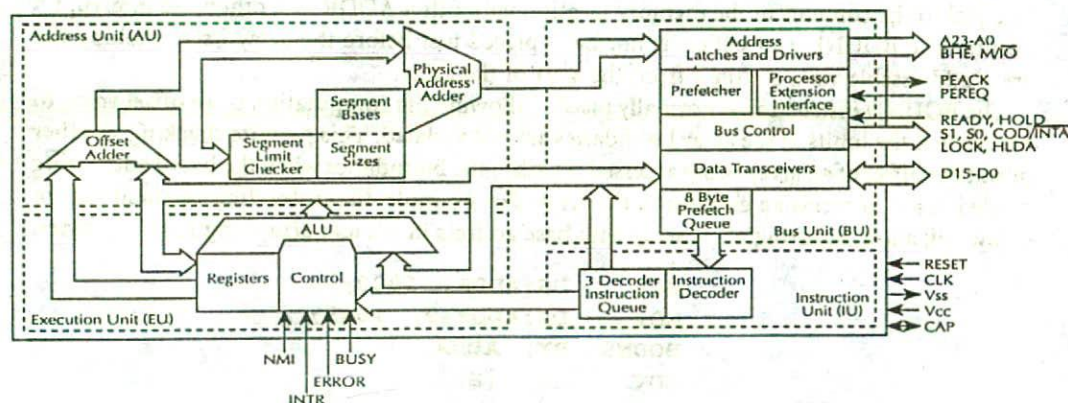


FIGURE 4.2 80286 internal block diagram.

coprocessors such as the 80287. The prefetcher in the BU prefetches instructions of up to 6 bytes and places them in a queue.

The Instruction Unit (IU) translates or decodes up to 3 prefetched instructions and places them in a queue for execution by the execution unit.

The Execution Unit (EU) executes instructions from the IU sequentially. The EU contains a 16-bit ALU, an 8086 flag register, general-purpose registers, pointer registers, index registers, and one 16-bit additional register called the machine status word (MSW) register. The lower four bits of the MSW are used. One bit places the 80286 into PVAM mode while the other three bits control the processor extension (coprocessor) interface. The LMSW and SMSW instructions can load and store MSW in real address mode.

The Address Unit (AU) calculates a 20-bit physical address based on the 16-bit contents of a segment register and a 16-bit offset just like the 8086. In this mode, the 80286 addresses one megabyte of physical memory. The 80286 has 24 address pins. However, in the real address mode, pins A23-A20 are ignored and A19-A0 pins are used. In the protected virtual address mode (PVAM), the AU operates as a memory management unit (MMU) and utilizes all 24 address lines to provide 16 megabytes of physical memory. The BU outputs memory or I/O addresses to devices connected to the 80286 after receiving them from the AU.

The 80286 does not have on-chip clock generator circuitry. Therefore, an external 82284 chip is required. The 80286 has a single CLK pin for a single-phase clock input. The 80286 divides its input clock by 2 internally and then provides the processor clock. The 82284 also provides the 80286 RESET and READY signals.

The 80286 external memory is configured as odd (BHE = 0) and even (A0 = 0) memory banks just like the 8086. The 80286 operates in a mode similar to the 8086 maximum mode. Some of the 80286 pins such as M/IO, S0, S1, HOLD, HLDA, READY, and LOCK have identical functions as the 8086. Two external interrupt pins (NMI and INTR) are provided. The nonmaskable interrupt NMI is serviced in the same way as the 8086. The INTR and COD/INTA are used together to provide an interrupt-type code on the data bus via external hardware. Note that the 80286 INTA is multiplexed with another function called the COD (code). This pin distinguishes instruction fetch cycles from memory data read cycles. Also, it distinguishes interrupt acknowledge cycles from I/O cycles. M/IO = HIGH and COD/INTA = HIGH define instruction fetch cycle. On the other hand, M/IO = LOW and COD/INTA = LOW specify interrupt acknowledge cycle.

A new pin called the CAP is provided on the chip. The 80286 MOS substrate must be applied with a negative voltage for maximum speed. The negative voltage is obtained from the +5V. An external capacitor must be connected to the CAP pin for filtering this bias voltage.

Four pins are provided to interface the 80286 with a coprocessor. These are PEREQ, PEACK, BUSY, and ERROR. The PEREQ (Processor Extension Request) input pin can be activated by the coprocessor to tell the 80286 to perform data transfer to or from memory for it. When the 80286 is ready, it activates the PEACK (Processor Extension Acknowledge) signal to inform the coprocessor of the start of the transfer. The 80286 BUSY signal input, when activated LOW by the coprocessor, stops 80286 program execution on WAIT and some ESC instructions until BUSY is HIGH. If a coprocessor finds some error during processing, it will activate the 80286 ERROR input pin. This will generate an interrupt. A service routine for this interrupt can be written to provide an indication to inform the user of the error.

Upon hardware reset, the 80286 operates in real (physical) address mode and starts executing programs at physical address FFFF0H like the 8086. In this mode, 20-bit physical addresses are generated by adding a 16-bit offset to the shifted (4 times to the left) segment register just like the 8086. Note that after hardware reset, the 80286 sets A23-A20 to all ones, CS = F000H, DS = 0000H, ES = 0000H, and SS = 0000H.

The 80286 on-chip MMU is disabled in the real address mode. In this mode, the 80286 acts functionally as a high-performance 8086. This mode averages 2 1/2 times the performance of an 8086 running at the same clock frequency. All instructions of the 8086, 80186, plus a few

more such as LMSW (Load Machine Status Word) and SMSW (Store Machine Status Word) are available with the 80286 in the real address mode. In this mode, the 80286 supports only the 8086 data types and can directly execute 8086 machine code programs with minor modifications. When interfaced with an 80287 floating point coprocessor, the 80286 supports 8087 floating point data types also. Upon hardware reset, the 80286 operates in real address mode unless the user sets a bit in the Machine States Word (MSW) register by using the LMSW instruction to change the 80286 mode to Protected Virtual Address Mode (PVAM). Note that before changing to PVAM, descriptor tables must be in memory. The 80286, in real address mode, can run 8086 or 8088 software. Now, to change the 80286 mode from real address to PVAM, the user should read the contents of MSW, set just the Protection Enable (PE) bit to 1 without changing the other bits, and then load the new data into the MSW. The following instruction sequence will accomplish this:

```

SMSW CX    ; Store MSW into a general register such as CX
OR CX, 1    ; Set only the PE bit (bit 0 in MSW)
LMSW CX     ; Load the new value back to MSW.

```

After the above instruction sequence is executed, the 80286 operates in PVAM with memory management capabilities. In the PVAM, the 80286 is compatible with the 8086/8088 at the source code level but not at the machine code level. This means that most 8086/8088 programs must be recompiled or reassembled. Note that the real address mode is normally used to initialize peripheral devices, transfer the main portion of the operating system from disk to main memory, initialize some registers, enable interrupts and place the 80286 into PVAM.

When the 80286 is in the protected mode, the on-chip MMU is enabled which expects several address-mapping tables to exist in memory. The 80286, in this mode, will automatically access these tables for translating the logical addresses used by the user to physical addresses. Once the 80286 is in PVAM, the only way to get back to the real mode is via hardware reset. This is intentionally done so that a malicious programmer cannot switch the mode from PVAM to real mode and thus the protection feature in PVAM is maintained.

The 80286 supports the following data types:

- 8-bit or 16-bit signed binary numbers (integers)
- Unsigned 8- or 16-bit numbers (ordinal)
- A 32-bit pointer comprised of a 16-bit segment selector and 16-bit offset
- A contiguous sequence of bytes or words (strings)
- ASCII
- Packed and unpacked BCD
- Floating point

The 80286 provides 8 addressing modes. These include register, immediate, direct, register indirect, based, indexed, based index, and based indexed with displacement modes. The new 80286 instructions are for supporting the PVAM of the 80286 via an operating system.

These instructions are listed in the following and are used by the operating system:

CTST	Clear task switch flag to zero located in the MSW register
LGDT	Load global descriptor table register from memory
SGDT	Store global descriptor table register into memory
LIDT	Load interrupt descriptor table register from memory
LLDT	Load selector and associated descriptor into LDTR (local descriptor table register)
SLDT	Store selector from LDTR in specified register or memory
LTR	Load task register and descriptor for TSS (task state segment)

STR	Store selector from task register in register or memory
LMSW	Load MSW register from register or memory
SMSW	Store MSW register in register or memory
LAR	Load access rights byte of descriptor into register or memory
LSL	Load segment limit from descriptor into register or memory
ARPL	Adjust register privilege Level of selector
VERR	Determine if segment addressed by a selector is readable
VERW	Determine if segment pointed to the selector is writable

Next, the 80286 will be considered from an operating systems point of view. In this context, the memory management capabilities protection and task switching features of the 80286 will be covered. Using these on-chip hardware features, a multitasking operating system can be implemented in the 80286-based microcomputer system.

The 80286 memory management features provide the operating system with the following capabilities:

- An operating system which can separate tasks from each other. This avoids an 80286 system failure due to task errors.
- As tasks begin and end, the operating system can optimize memory usage by moving them around, a process referred to as dynamic relocation. This is because a program can be executed in different parts of memory without being reassembled or recompiled.
- Use of virtual memory becomes easy. Note that virtual memory is a method for executing programs larger than the main memory by automatically transferring parts of the programs between main memory and disk.
- Controlled sharing of information between tasks.

The 80286 protection features allow:

- An operating system to protect itself from malicious users in a multiuser environment
- Critical subsystems such as disk I/O from being destroyed by program bugs under development

The 80286 task switching provides:

- Fast task switching due to 80286's hardware implementation for accomplishing this feature. This permits the 80286 to spend more time on task execution than switching. Real-time systems can thus be supported by the 80286 since they may require fast task switching. Note that an exception in a running task or an interrupt from a peripheral device requires task switching.

4.1.2.a 80286 Memory Management

The 80286 logical segments may be called virtual segments because all of them may not be resident in physical memory at the same time. A 80286 logical segment can be of any length from 1 byte to 64K bytes. During creation of each segment, a size or limit value is defined. This makes it easier for the 80286 to determine if a memory access is within bounds of a segment. The segments presently used by a task are stored in physical memory. In PVAM, all 24 address pins are used and therefore, the directly addressable (physical) memory is 16 megabytes. While writing 80286 programs in PVAM, one can refer to the current segment by the assigned names. For example, if a segment called JOHN is to be used as the current data segment, then the following instructions should be used to load JOHN into DS:

```
MOV BX, JOHN
MOV DS, BX.
```


When the 80286 executes the above instruction segment, it will use JOHN as the current data segment. If the segment JOHN is not currently resident in Physical memory, the 80286 will generate an interrupt. A service routine is written to load this segment from disk to physical memory and then resume execution of the main program.

In PVAM, when a program is assembled, a descriptor (8-byte wide) is assigned to each segment. The descriptor contains information such as segment length in bytes, 24-bit base address where the segment is located in physical memory and the privilege level.

The descriptors are held in tables in main memory and are read into the 80286 as needed. There are two main types of descriptor tables. These are the global and the local descriptor tables. A system can contain only one global descriptor table. The global descriptor table includes information such as the descriptors for the operating system segments and the descriptors for segments which are accessed by user tasks. A local descriptor is created in the system for each task. All tasks share a global descriptor table with the memory areas specified by its descriptors. Also, each task contains its own local descriptor table with the memory areas specified by its descriptors.

In PVAM, all programs are written using segments. Each segment is assigned with an 8-byte descriptor which includes the length, starting address, and access rights for that segment. Segment descriptors for programs are stored in memory either in the global descriptor table or in a local descriptor table.

The 80286 memory management is based on address translation. That is, the 80286 translates logical addresses (addresses used in programs) to physical addresses (addressing required by memory hardware). The 80286 memory pointer includes two 16-bit words: one word for a segment selector and the other as an offset into the selected segment. The real and virtual modes compute physical addresses from these selector and offset values in different ways.

In the real address mode, the 80286 computes the physical address from a 16-bit (selector) content and a 16-bit offset just like the 8086/80186. It shifts the 16-bit selector four times to the left and then adds the 16-bit offset to determine the 20-bit physical address. As mentioned before, even though the 80286 has 24 address pins (A0-A23), in the real address mode pins A0-A19 are used. Also, A20-A23 pins are only used at reset for CS and are zero otherwise.

In the Protected Virtual Address Mode (PVAM or virtual mode for short), the 32-bit address is called a virtual address. Just like the logical address, the virtual address includes a 16-bit selector and a 16-bit offset. The 80286 determines the 24-bit physical address by first obtaining a 24-bit value from a table in memory using the segment value (selector) as an index (rather than shifting the segment value 4 times to left as in the real mode) and then adding the 16-bit offset. Figure 4.3 shows the 80286 virtual address translation scheme.

The 16-bit selector is divided into a 13-bit index, one-bit Table Indicator (TI), and two-bit Requested Privilege Level (RPL). The 13-bit index is used as a displacement to access the selected table. Each entry in the table is termed a descriptor. An index can start from a value of 0 to a higher value. The index value refers to a descriptor in the table. For example, index value K refers to the descriptor K. Each descriptor is 8 bytes wide and contains the 24-bit base address required for physical address calculation. This address only occupies three bytes of the 8-byte descriptor. The meaning of the other bytes will be explained later. The single-bit TI tells the 80286 to select one of two tables: Global Descriptor Table (GDT) and Local Descriptor Table (LDT). All tasks in the 80286 share a common single table called the GDT, while each task has its own LDT. Therefore, the 24-bit base addresses required in physical address calculation for segments to be shared by all tasks are stored in the GDT and the base addresses for segments dedicated to a particular task are stored in its LDT. When TI = 0, the GDT is used as the look-up table, and when TI = 1, the LDT is used as the look-up table. The 2-bit RPL is used by the operating system for implementing the 80286's protection features. RPL is not used in physical address calculation.

The 24-bit physical address is then generated by the 80286 by adding the 24-bit base address of the selected descriptor and the 16-bit offset.

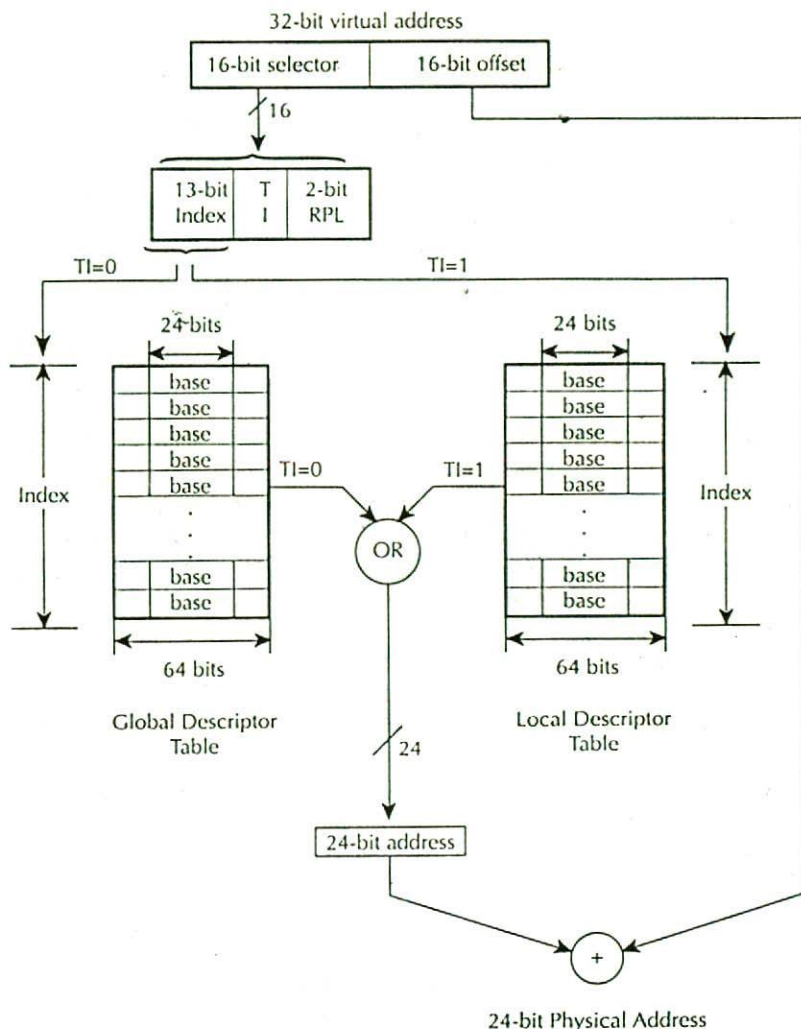


FIGURE 4.3 80286 virtual address translation.

Note that in the above, when $TI = 0$ (GDT selector) and $Index = 0$, a null selector is selected. The selector does not correspond to the 0th GDT descriptor. Null selectors can be loaded into a segment register, but use of null selectors in virtual address translation would generate an 80286 exception.

Figure 4.4 shows the 80286 address translation registers. The segment registers CS, DS, SS, and ES have already been discussed before. The GDT and LDT registers are only used in the 80286 virtual mode address translation. The GDT register stores the 24-bit base address and the length of the GDT (in bytes) minus one. During system initialization, the GDT is loaded and is usually kept unchanged after this. The 80286 generates an exception when indexing beyond the GDT limit is attempted.

The LDT register stores a 16-bit selector for a descriptor in the GDT which defines the location of the present LDT. The 80286 task switch operation is invoked by executing an inter-segment JMP or CALL instruction which refers to a task state segment (TSS) or task gate descriptor in the GDT or LDT. Each task must have a TSS associated with it. The current TSS is identified by a special register in the 80286 called the Task Register (TR). The TR register makes task switching automatic and very fast. For example, the 80286 local address space can be modified during task switching by updating the LDT register. Figure 4.5 shows flowchart for accessing memory.

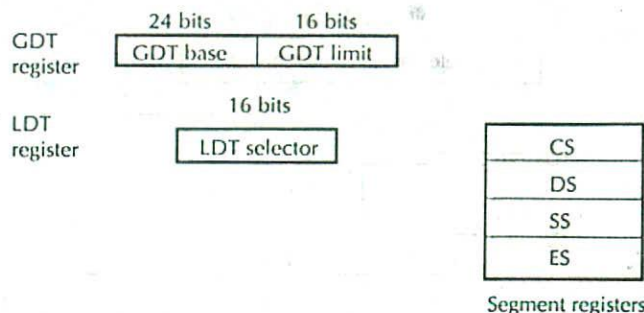


FIGURE 4.4 80286 address translation registers.

The 80286 contains a number of special registers called shadow registers which are provided to speed up memory references. The shadow registers are internal and cannot be accessed by instructions. Whenever a selector is moved into a segment register, the associated shadow register is updated with its descriptor automatically. Therefore, any memory accessed with respect to the segment register does not require referral to a look-up table, since the descriptor loaded into the shadow register contains the base address of the selected segment.

Note that in the virtual mode, the 80286 descriptor table can store a maximum of 2^{13} (13-bit index) descriptors and each segment can specify a segment of up to 2^{16} bytes. Therefore, a task can have its own LDT address space of up to $2^{13} \times 2^{16} = 2^{29}$ bytes and can share 2^{29} bytes (GDT) with all other tasks. Therefore, an address space of 1 gigabyte (2^{30} bytes) can be assigned to a task.

The following 80286 memory management instructions include loading and storing the address translation registers and checking the contents of descriptors:

LGDT	Load GDT register
SGDT	Store GDT register
LLDT	Load LDT register
SLDT	Store LDT register
LAR	Load Access Rights
LSL	Load Segment Limit

4.1.2.b Protection

In PVAM, the 80286 has built-in features for the following protection schemes:

1. Protecting system software such as the operating system from user programs.
2. Protecting one user task from another.
3. Protecting portions of memory from accidental access.

The 80286 protection mechanism is implemented by using the contents of the descriptors. Any access to memory is validated by checking the information in segment descriptors. The 80286 generates an interrupt if the memory access is invalid.

The 80286 provides protection mechanism for supporting multitasking and virtual memory features. The 80286 includes certain basic protection features such as segment limit and segment usage checking. These basic protections are useful even though multitasking and virtual memory may not be available in a system. The basic protection mechanism also allows assignment of privilege levels to virtual memory space in a hierarchical manner.

The 80286 privilege level mechanism uses certain rules to define the hierarchical order. This allows protection of the operating system independent of the user. The 80286 includes special descriptor table entries named call gates to permit CALLS to higher privilege code segments

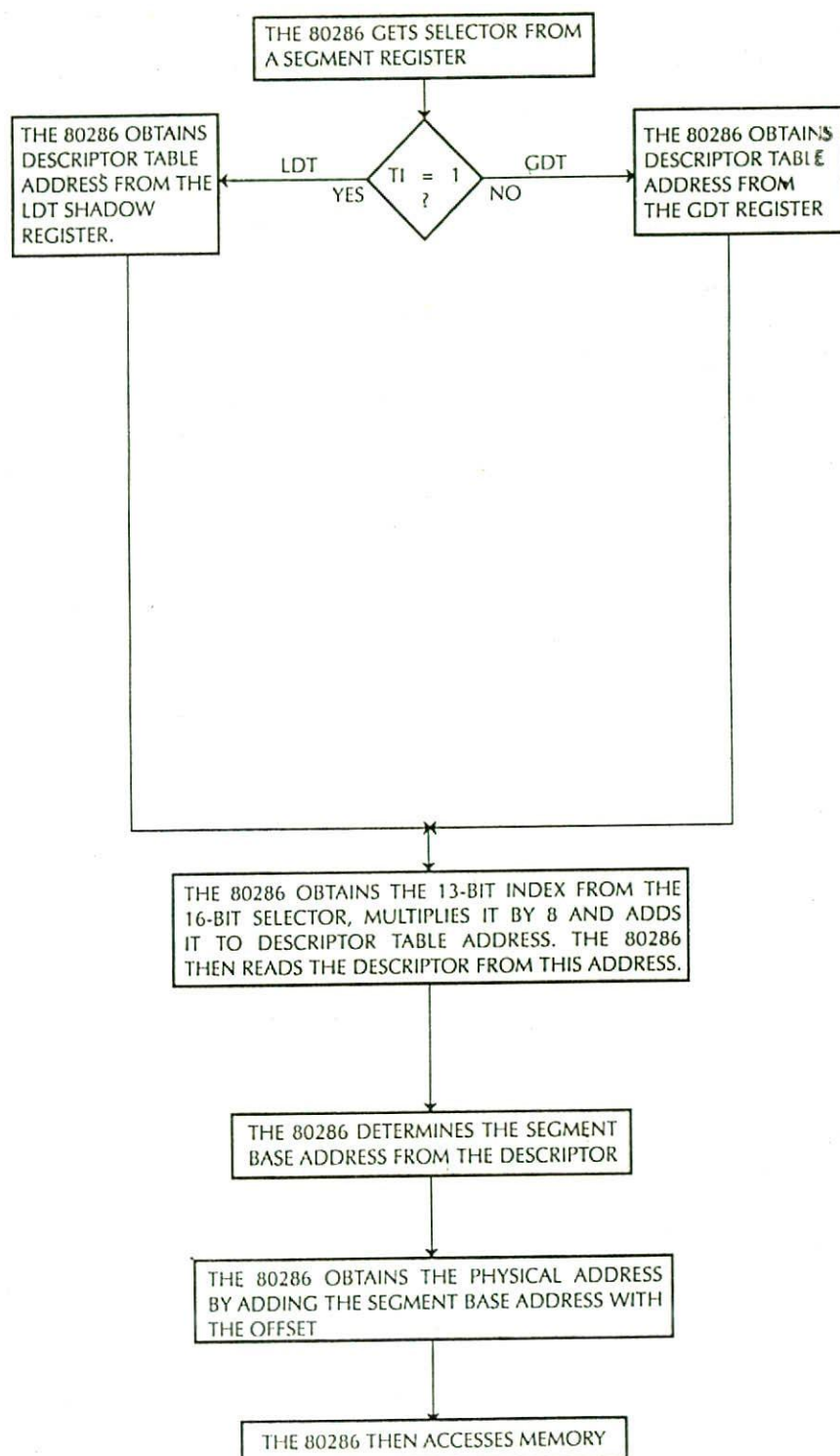
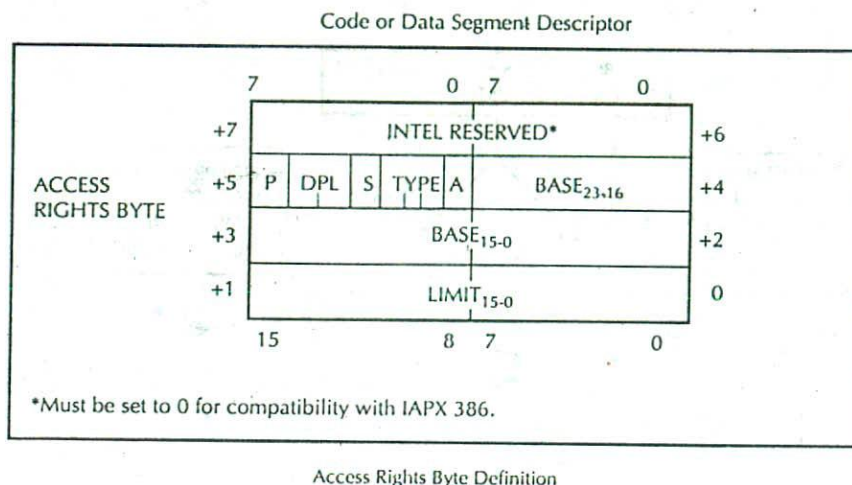


FIGURE 4.5 Flowchart showing steps for accessing memory.



Access Rights Byte Definition

Bit Position	Name	Function
7	Present (P)	P = 1 Segment is mapped into physical Memory. P = 0 No mapping to physical memory exists, base and limit are not used. Segment privilege attribute used in privilege test.
6-5	Descriptor Privilege Level (DPL)	
4	Segment Descriptor (S)	S = 1 Code or data (includes stacks) segment descriptor S = 0 System Segment Descriptor or Gate Descriptor
3	Executable (E)	E = 0 Data segment descriptor type is:
2	Expansion Direction (ED)	ED = 0 Expand up segment, offsets must be \leq limit. ED = 1 Expand down segment, offsets must be $>$ limit.
1	Writeable (W)	W = 0 Data segment may not be written into. W = 1 Data segment may be written into.
3	Executable (E)	E = 1 Code Segment Descriptor type is:
2	Conforming (C)	C = 1 Code segment may only be executed when $CPL \geq DPL$ and CPL remains unchanged.
1	Readable (R)	R = 0 Code segment may not be read. R = 1 Code segment may be read.
0	Accessed (A)	A = 0 Segment has not been accessed. A = 1 Segment selector has been loaded into segment register or used by selector test instructions.

FIGURE 4.6 Code and data segment descriptor contents.

at specific valid entry points. In order to protect the operating system, the 80286 does not allow accessing a higher privilege entry point without its access gate.

The 80286 provides some instructions controlled by its IOPL (Input/Output Privilege Level) feature to protect shared system resources. The I/O instructions are permitted at the highest privilege level.

The 80286 on-chip protection features handle basic violations such as trying to access code segment instead of stack segment by trapping into the operating systems's appropriate routine. Thus, the operating system recovers a faulty task by taking whatever actions are necessary. The 80286 protection hardware provides information such as stack status to inform the operating system of the fault type.

The 80286 on-chip protection hardware provides up to four privilege levels in a hierarchical manner which can be used to protect system software such as the operating system from unauthorized access. The 80286 can read specific bits in its segment descriptor to obtain privilege levels of each code and data segment. Figure 4.6 shows the format for a segment descriptor for a code or data segment.

The descriptors include four words. Bit 3 of the 3-bit-type field of the access rights byte is called the Executable (E) bit. E = 1 identifies a code segment descriptor, while E = 0 identifies the segment as a data segment descriptor. The two-bit DPL (Descriptor Privilege Level) provides the privilege level of the descriptor. The DPL field specifies a hierarchical privilege system of four levels 0 thru 3.

Level 0 is the highest level while level 3 is the lowest. Privilege level provides protection within a task. Operating system routines, interrupt handlers, and other system software can be included and protected within the virtual address space of each task using the four levels of privileges. Each task in the system has a separate stack for each of its privilege levels. Typical examples of privilege levels are summarized as follows:

- A single privilege level can be assigned to all the code of a dedicated 80286. In this case, all load/store and I/O instructions are available. The 80286 can be initialized by setting the PE bit in MSW to one and then loading the GDTR with appropriate values to address a valid global descriptor table.
- Privilege levels can be assigned to user and supervisor mode types of applications. In this case, all system software can be defined as level 0 (highest level) and all other programs at some lower privilege level.
- For large applications, the system software can be divided into critical and noncritical. All critical software can be defined as a kernel with the highest privilege level (level 0), the noncritical portion of the system software is defined with levels 1 and 2, while all user application programs are defined with the lowest level (level 3).

Upon enabling the protected mode bit, the 80286 basic protection features are available irrespective of a single 80286-type application or user/supervisor configuration or a large application. The descriptor's limit field (the maximum offset from the base) and the access rights byte provides the 80286's basic protection features.

Segment limit checking ensures that all memory accesses are physically available in the segment. For all read and write operations with memory, the 80286 in the protected mode automatically checks the offset of an effective address with the descriptor limit (predefined). This limit checking feature ensures that a software fault in a segment does not interfere with any other segments in the system.

The descriptor access rights byte complements the limit checking. It differentiates code segments from data segments. The access right byte along with limit checking ensures proper usage of the segments. At least three types of segments can be defined using the access byte. Data segments can be defined as read/write or read-only. Code segments can be designated as execute-only and can be defined as conforming segments. A code segment in a particular privilege level can be accessed by using 80286 CALL or JMP instruction without a privilege level transition. A segment of equal or lower privilege level than another segment (defined as conforming via the access byte) can access that segment.

The hierarchical protection levels have four logical rules. These are summarized below:

- The Current Privilege Level (CPL) at any instant of time represents the level of the code segment presently being executed. This is provided by the privilege level in the access rights byte of the descriptor. The 80286 gives the value of CPL in its code register.
- Since every privilege level has its own stack, a stack segment rule is implemented in the 80286 to ensure using the proper stack. According to this rule, the stack segment (stack addressed by the stack segment register) and the current code segment must have the identical privilege level.
- As far as the data segments are concerned, the DPL (Descriptor Privilege Level) of an accessed data segment must be lower than or equal to the CPL. This rule allows protection of privileged data segments from unprivileged codes.

- The 80286 is provided with a rule which pertains to accessing data segments. The 80286 can access data segments of equal or lower privilege with respect to the CPL. For example, if CPL is 1, the 80286 codes in current code segment can access data segments with privilege levels of 1, 2, or 3, but not 0.
- The 80286 allows CALLing a subroutine in a code segment with higher privilege level by using call gates, and returns to code with lower privilege code segments. This is called the flow control rule and it protects higher privilege code segments. For example, if the CPL is 2, all code segments with levels 2 and 3 can be accessed by the 80286; code segments of levels 0 or 1 cannot be accessed directly. However, lower or equal privilege level accesses can be done directly. Also, higher privilege level accesses can be controlled by special descriptor table entries known as call gates. A call gate is 8 bytes wide and is stored like a descriptor in a descriptor table.

The main difference between a descriptor and a call gate is that a descriptor's contents refer to a segment in memory. On the other hand, a gate refers to another descriptor.

A descriptor includes a 24-bit physical base address, while a gate contains a 32-bit virtual address. When the effective address of an intersegment CALL references a call gate, the 80286 redirects control to the destination address defined within the gate. The 32-bit virtual address (selector and offset) of the gate can be used by the 80286 to access a higher privilege code segment.

The 80286 controls the use of I/O instructions. The user may choose the level at which these I/O instructions can be used. This level is called the IOPL (Input Output Privilege Level).

IOPL is a two-bit flag whose value varies from 0 to 3. In a user/supervisor configuration in which all supervisor code is at level 0 (highest) and all user code is at lower levels, the IOPL should be 0. This zero value of IOPL allows the supervisor code to carry out I/O operations but ensures that the user code cannot execute these I/O instructions.

Protection of a task from unauthorized access by another is provided by the 80286 both in virtual and physical memory spaces. The 80286 provides a multitasking feature via its virtual memory capabilities. As mentioned before, the virtual memory space consists of two spaces: global and local. The local space is unique to the present task being executed. This uniqueness of the local spaces provides intertask protection in the virtual memory space. The 80286's limit checking feature in the physical memory space avoids illegal accesses of segments beyond the defined segment limits and thus provides protection.

The 80286 is especially designed to execute several different tasks simultaneously (appears to be simultaneous). This is called multitasking. If the present task needs to wait for some external data, the 80286 can be programmed to switch to another task until such data are available. This mechanism of switching from one task to another is called task switching. The 80286 automatically performs all the necessary steps in order to properly switch from one task to another. When a task switching takes place, the 80286 stores the state of the present task (typically most of the 80286 registers), loads the state of the new task, and starts executing the new task. If execution of the outgoing task is desired after completion of the incoming one, the 80286 can automatically go back to the right place where the task switch took place.

Task switching may occur due to hardware or software reasons. For example, task switching may take place due to 80286 external interrupt requests (hardware reason) or due to the operating system's desire to time-share the 80286 among multiple user tasks (software reason). The task to be executed due to interrupts is termed interrupt-scheduled, while the task to be executed due to time-sharing by the operating system is called software-scheduled.

As soon as an interrupt-scheduled or a software-scheduled task is ready to be run by the 80286, it becomes the currently active (incoming) task. All inactive tasks (outgoing) have code and data segments saved in memory or disk by the 80286. Each outgoing task has a Task State Segment (TSS) associated with it. The TSS holds the task register state of an inactive task.

The TSS includes a special access right byte in its descriptor in the GDT in order that the 80286 can identify it as code or data segments. TSSs are referenced by 16-bit selectors (each task has a unique selector) that identify a TSS descriptor in the GDT. The 80286 stores the TSS selector of the presently active task in its Task State Segment register (TR). The first 44 bytes of a TSS store the complete state of a task. Information such as selectors and 80286 registers is saved.

The 80286 provides protection for portions of memory from accidental access in the following ways. When a segment selector is to be loaded into a segment register, the 80286 automatically verifies whether the descriptor table indexed by the selector contains a valid descriptor for that selector. An interrupt is automatically generated by the 80286 if a valid descriptor is not present. However, if the descriptor is valid, the shadow registers are loaded with the base, limit, and access rights byte of the descriptor. The 80286 then verifies whether the segment for that descriptor is resident in physical memory. An interrupt is generated by the 80286 if it is not present (as indicated by the P-bit of the access rights byte of the descriptor). An interrupt service routine can be written to move the desired segment into physical memory and return execution to the main program. The 80286 also verifies whether the segment descriptor is of the right type to be loaded into the appropriate segment register. For example, the descriptor for a read-only data segment cannot be moved into a stack segment register since the stack is a read/write memory. After a segment selector and descriptor are loaded into a segment register, checks are made each time an address in the actual segment is accessed. For example, an attempt to write to a read-only data segment will generate an error. Also, the limit value in the segment descriptor is used to ensure that an address generated by program instructions is within the limit specified for the segment.

Example 4.1

Discuss the 80286's performance impact on memory management while executing the following program:

```

MOV DS, Segmentselector    ; Load data selector
MOV BX, Displ              ; Load offset
MOV CX, Count              ; Load loop count
BEGIN MOV DX, data         ; Move 16-bit data to
                           DX
    CMP DX, WORDPTR[BX]    ; Find match
    JZ DONE               ; If match
    JMP BEGIN             ; found, stop
                           ; else compare
DONE HLT

```

Solution

The 80286 memory management capabilities are only utilized when loading the selector value (first instruction in the above program). By loading the selector value into DS, the 80286 chooses a descriptor from a descriptor table and then automatically determines the physical address of that segment using the descriptor. Thus, by executing the first instruction `MOV DS, segmentselector`, the 80286 automatically determines the segment's physical address (transparent to the user).

After determining the segment's physical address, the 80286 executes all other instructions that follow. The 80286 reads data from the data segment every time it goes through the `BEGIN` loop. The 80286 does not refer to the descriptor table because of its on-chip cache.

Like any other memory management system, the 80286 will have some overhead for performing the virtual address to the physical address translation. This is transparent to the user's application programs and is automatically carried out by the 80286. The memory management overhead (virtual to physical address translation) is minimized due to on-chip cache. One of the main characteristics of on-chip MMU is that after the descriptor is read into the on-chip cache memory, no overhead occurs. Therefore, the overhead is kept at a minimum, thus providing good performance.

4.1.2.c 80286 Exceptions

In the real address mode, the 80286 exception mechanism is similar to the 8086 except a few more exceptions such as the 'invalid opcode exception' is included in the 80286. External interrupts such as the maskable (INTR) and nonmaskable (NMI) interrupts in the 80286 are serviced in the real address mode by using interrupt vector table with 256 vectors similar to the 8086.

In the protected mode, the 80286 detects exceptions essential to its protection model, and its support for multitasking and virtual memory. Some examples of the non-real mode exceptions include 'exception for code, data, or extra segment not present' and 'Privilege violation'. The protected mode pointers are actually gates, since gates in protected mode serve as a redirection mechanism. The interrupt table can contain task gates, interrupt gates, and trap gates. The interrupt table, in protected mode, is known as the IDT (Interrupt Descriptor Table). A special register called the IDTR (Interrupt Descriptor Table register) is used by the 80286 to locate the interrupt table both in real and protected modes. The IDTR is typically initialized by the system programmer once, to point at the desired area of physical memory.

4.2 INTEL 80386

In 1985, Intel introduced its first 32-bit microprocessor, the 80386DX. It initially ran at a clock frequency of 16 MHz and contained 275,000 transistors. In 1988, Intel introduced the 80386SX which was a 16-bit external data bus version of the 80386DX (32-bit data bus). The 80386DX is a full 32-bit microprocessor and is available in 16-, 20-, 25- and 33-Mhz speed versions. It can directly address a maximum of 4 gigabytes of memory.

The 80386SX, on the other hand, can operate at 16- and 20-Mhz. The 80386SL is similar to the 80386SX with 16-bit data bus and can operate at 20- and 25-Mhz. The 80386SX and 80386SL can directly address up to 16 megabytes and 32 megabytes of memory respectively. Compaq was the first major OEM (Original Equipment Manufacturer) to use the 80386DX in 1986 in its PC. The 80386SL, on the other hand, is used in notebook computers with built-in power management options.

The 80386DX will be covered in detail in this section. The 80386 family of microprocessors will be referred to as the 80386 in the following.

The 80386 is a logical extension of the Intel 80286.

The 80386 provides multitasking support, memory management, pipelined architecture, address translation caches, and a high-speed bus interface in a single chip.

The 80386 is software compatible at the object code level with the Intel 8086, 80186, and 80286. The 80386 includes separate 32-bit internal and external data paths along with eight general-purpose 32-bit registers. The processor can handle 8-, 16-, and 32-bit data types. It has separate 32-bit data and generates a 32-bit physical address. The chip has 132 pins and is housed in a Pin Grid Array (PGA) package. The 80386 is designed using high-speed CHMOS III technology.

The 80386 is highly pipelined and can perform instruction fetching, decoding, execution, and memory management functions in parallel. The on-chip memory management and protection hardware translates logical addresses to physical addresses and provides the protec-

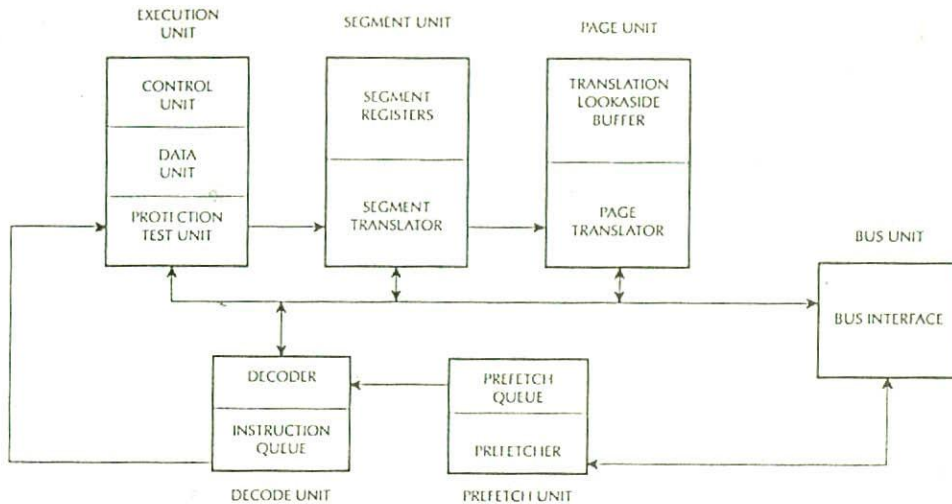


FIGURE 4.7 80386 functional units.

tion rules required in a multitasking environment. The 80386 includes special hardware for task switching. A single instruction or an interrupt is required for the 80386 to perform complete task switching. A 16-MHz 80386 can save the state of one task (all registers), load the state of another task (all registers, segment, and paging registers if needed), and resume execution in less than 16 microseconds. The 80386 contains a total of 129 instructions.

The 80386 protection mechanism, paging, and the instructions to support them are not present in the 8086. Also, the semantics of all instructions that affect segment registers (PUSH, POP, MOV, LES, LDS) and those affecting program flow (CALL, INTO, INT, IRET, JMP, RET) are quite different than the 8086 on the 80386 in protected mode.

The main differences between the 80286 and the 80386 are the 32-bit addresses and data types and paging and memory management. To provide these features and other applications, several new instructions are added in the 80386 instruction set beyond those of the 80286.

The internal architecture of the 80386 includes six functional units (Figure 4.7) that operate in parallel. The parallel operation is known as pipelined processing. Fetching, decoding, execution, memory management, and bus access for several instructions are performed simultaneously. The six functional units of the 80386 are

- Bus interface unit
- Code prefetch unit
- Decode unit
- Execution unit
- Segmentation unit
- Paging unit

The bus interface unit connects the 80386 with memory and I/O. Based on internal requests for fetching instructions and transferring data from the code prefetch unit, the 80386 generates the address, data, and control signals for the current bus cycles.

The code prefetch unit prefetches instructions when the bus interface unit is not executing bus cycles. It then stores them in a 16-byte instruction queue for decoding by the instruction decode unit.

The instruction decode unit translates instructions from the prefetch queue into microcodes. The decoded instructions are then stored in an instruction queue (FIFO) for processing by the execution unit.

The execution unit processes the instructions from the instruction queue. It contains a control unit, a data unit, and a protection test unit.

The control unit contains microcode and parallel hardware for fast multiply, divide, and effective address calculation.

The data unit includes a 32-bit ALU, 8 general-purpose registers, and a 64-bit barrel shifter for performing multiple bit shifts in one clock. The data unit carries out data operations requested by the control unit. The protection test unit checks for segmentation violations under the control of the microcode.

The segmentation unit translates logical addresses into linear addresses at the request of the execution unit.

The translated linear address is sent to the paging unit. Upon enabling of the paging mechanism, the 80386 translates these linear addresses into physical addresses. If paging is not enabled, the physical address is identical to the linear addresses and no translation is necessary.

Figure 4.8 shows a typical 80386 system block diagram.

The 80287 or 80387 numeric coprocessor can be interfaced to the 80386 to extend the 80386 instruction set to include instructions such as floating point operations. These instructions are executed in parallel by the 80287 or 80387 with the 80386 and thus off-load the 80386 of these functions.

The 82384 clock generator provides system clock and reset signals. The 82384 generates both the 80386 clock (CLK2) and a half-frequency clock (CLK) to drive the 80286-compatible devices that may be included in the system. It also generates the 80386 RESET signal. The internal frequency of the 80386 is 1/2 the frequency of CLK2.

The 8259A interrupt controller provides interrupt control and management functions. Interrupts from as many as eight external sources are accepted by one 8259A and up to 64 interrupt requests can be handled by connecting several 8259A chips. The 8259A manages priorities between several interrupts, then interrupts the 80386 and sends a code to the 80386 to identify the source of the interrupt.

The 82258 Advanced DMA (ADMA) controller performs DMA transfers between the main memory and the I/O device such as a hard disk or floppy disk without involving the 80386. It provides four channels and all signals necessary to perform DMA transfers.

The 80386 has three processing modes: protected mode, real-address mode, and virtual 8086 mode.

Protected mode is the normal 32-bit application of the 80386. All instructions and features of the 80386 are available in this mode.

Real-address mode (also known as the "real mode") is the mode of operation of the processor upon hardware RESET. This mode appears to programmers as a fast 8086 with a few new instructions. This mode is utilized by most applications for initialization purposes only.

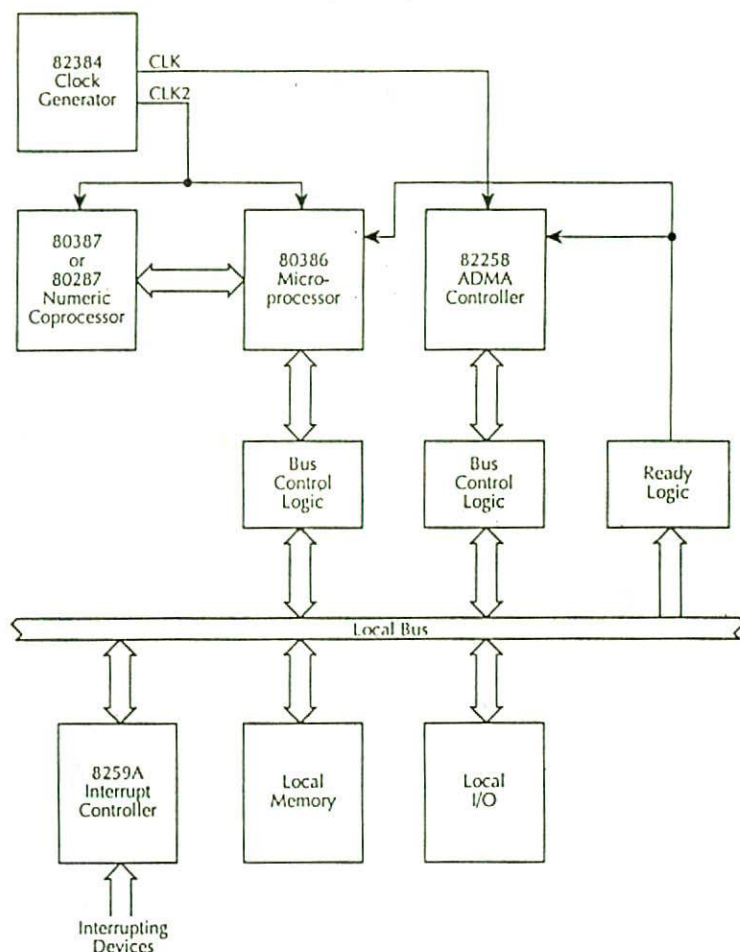
Virtual 8086 mode (also called V86 mode) is a mode in which the 80386 can go back and forth repeatedly between V86 mode and protected mode at a fast speed. The 80386, when entering into the V86 mode, can execute an 8086 program. The processor can then leave V86 mode and enter protected mode to execute an 80386 program.

As mentioned before, the 80386 enters real address mode upon hardware reset. In this mode, the Protection Enable (PE) bit in a control register called the Control Register 0 (CR0) is cleared to zero. Setting the PE bit in CR0 places the 80386 in protected mode. When in protected mode, setting the VM (Virtual Machine) bit in the flag register (called the EFLAGS register) will place the 80386 in V86 mode. Details of these modes are discussed later.

4.2.1 Basic 80386 Programming Model

The 80386 basic programming model includes the following aspects:

- a) Memory organization and segmentation
- b) Data types



Component	Description
80386 Microprocessor	32-bit high-performance microprocessor with on-chip memory management and protection; no on-chip cache
80287 or 80387 Numeric Coprocessor	Performs numeric instruction in parallel with 80386; expands instruction set
82384 Clock Generator	Generates system clock and RESET signal
8259A Programmable Interrupt Controller	Provides interrupt control and management
82258 Advanced DMA	Performs direct memory controller access (DMA)

FIGURE 4.8 80386 system block diagram.

- c) Registers
- d) Addressing modes

I/O is not included as part of the basic programming model. This is because systems designers may select to use I/O instructions for application programs or may select to reserve

them for the operating system. Therefore, 80386 I/O capabilities will be covered during the discussion of systems programming.

4.2.1.a Memory Organization and Segmentation

The 4-gigabyte physical memory of the 80386 is structured as 8-bit bytes. Each byte can be uniquely accessed by a 32-bit address.

The programmer can write assembly language programs without a knowledge of physical address space.

The memory organization model available to applications programmers is determined by the system software designers. The memory organization model available to the programmer for each task can vary between the following possibilities:

- A “flat” address space includes a single array of up to 4 gigabytes. Even though the physical address space can be up to 4 gigabytes, in reality it is much smaller. The 80386 maps the 4-gigabyte flat space into the physical address space automatically by using an address translation scheme transparent to the applications programmers.
- A segmented address space includes up to 16,383 linear address spaces of up to 4 gigabytes each. In a segmented model, the address space is called the logical address space and can be up to 2^{46} bytes (64 tetrabytes). The processor maps this address space onto the physical address space (up to 4 gigabytes) by an address translation technique.

To applications programmers, the logical address space appears as up to 16,383 one-dimensional subspaces, each with a specified length. Each of these linear subspaces is called a segment. A segment is a unit of contiguous address space with sizes varying from one byte up to a maximum of 4 gigabytes.

A pointer in the logical address space consists of a 16-bit segment selector identifying a segment and a 32-bit offset addressing a byte within a segment.

4.2.1.b Data Types

Data types can be byte (8-bit), word (16-bit with low byte address n and high byte by address $n + 1$), and double word (32-bit with byte 0 addressed by address n and byte 3 by address $n + 3$). All three data types can start at any byte address. Therefore, the words are not required to be aligned at even-numbered addresses and double words need not be aligned at addresses evenly divisible by 4. However, for maximum speed performance, data structures (including stacks) should be designed in such a way that, whenever possible, word operands are aligned at even addresses and double-word operands are aligned at addresses evenly divisible by 4.

Depending on the instruction referring to the operand, the following additional data types are available: integer (signed 8-, 16-, or 32-bit), ordinal (unsigned 8-, 16-, or 32-bit), near pointer (a 32-bit logical address which is an offset within a segment), far pointer (a 48-bit logical address consisting of a 16-bit selector and a 32-bit offset), string (8-, 16-, or 32-bit data from 0 bytes to $2^{32} - 1$ bytes), bit field (a contiguous sequence of bits starting at any bit position of any byte and may contain up to 32 bits), bit string (a contiguous sequence of bits starting at any position of any byte and may contain up to $2^{32} - 1$ bits), and packed/unpacked BCD and ASCII-type data. When the 80386 is interfaced to a coprocessor such as the 80287 or 80387, then floating point numbers (signed 32-, 64-, or 80-bit real numbers) are supported.

4.2.1.c 80386 Registers

Figure 4.9 shows 80386 registers. The 80386 has 16 registers classified as general, segment, status, and instruction.

The eight general registers are the 32-bit registers EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. The low-order word of each of these eight registers has the 8086/80186/80286 register

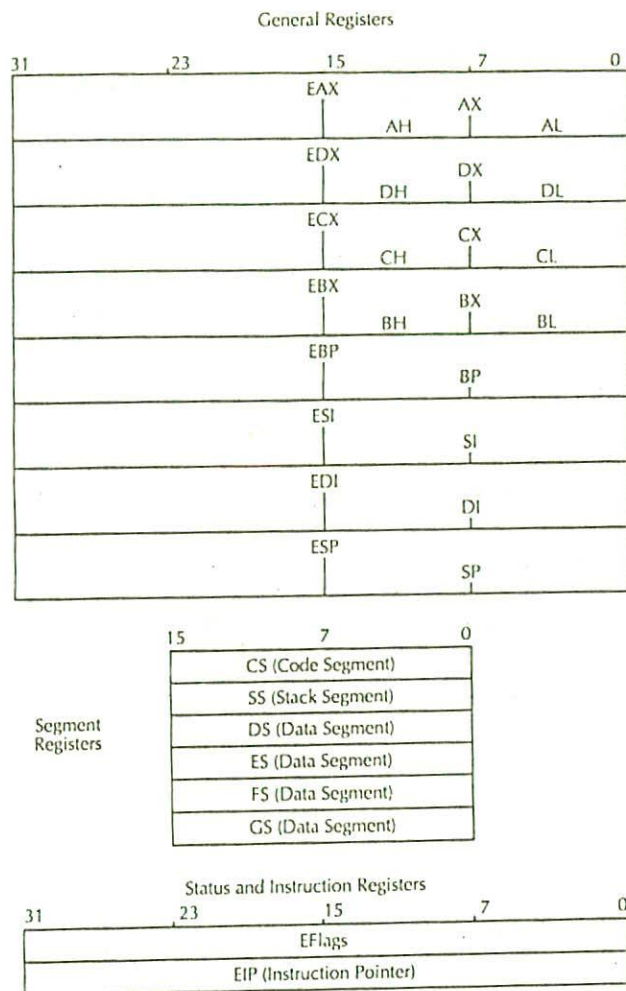


FIGURE 4.9 80386 applications register set.

names AX (AH or AL), BX (BH or BL), CX (CH or CL), DX (DH or DL), BP, SP, SI, and DI. They are useful for making the 80386 compatible with the 8086, 80186, and 80286 processors.

The six 16-bit segment registers (CS, SS, DS, ES, FS, and GS) allow systems software designers to select either a flat or segmented model of memory organization. The purpose of CS, SS, DS, and ES is obvious. Two additional data segment registers FS and GS are included in the 80386. The four data segment registers (DS, ES, FS, GS) can access four separate data areas and allow programs to access different types of data structures. For example, one data segment register can point to the data structures of the current module, another to the exported data of a higher level module, another to a dynamically created data structure, and another to data shared with another task.

The flag register is a 32-bit register named EFLAGS. Figure 4.10 shows the meaning of each bit in this register. The low-order 16 bits of EFLAGS is named FLAGS and can be treated as

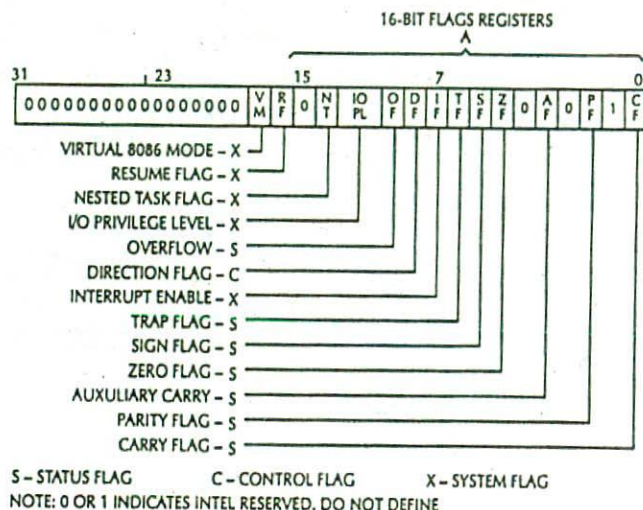


FIGURE 4.10 EFLAGS register.

a unit. This is useful when executing 8086/80186/80286 code, because this part of EFLAGS is the same as the FLAGS register of the 80286/80186/80286. The 80386 flags are grouped into three types: the status flags, the control flags, and the system flags.

The status flags include CF, PF, AF, ZF, SF, and OF as in the 8086/80186/80286. The control flag DF is used by strings as in the 8086/80186/80286. The system flags control I/O, maskable interrupts, debugging, task switching, and enabling of virtual 8086 execution in a protected, multitasking environment. The purpose of IF and TF is identical to the 8086/80186/80286. Let us explain the other flags:

- **IOPL (I/O Privilege Level)** — This is a 2-bit field and supports the 80386 protection feature. The IOPL field defines the privilege level needed to execute I/O instructions. If the present privilege level is less than or equal to IOPL (privilege level is specified by numbers), the 80386 can execute I/O instructions; otherwise it takes a protection exception.
- **NT (Nested Task)** — The NT bit controls the IRET operation. If NT = 0, a usual return from interrupt is taken by the 80386 by popping EFLAGS, CS, and EIP from the stack. If NT = 1, the 80386 returns from an interrupt via task switching.
- **RF (Resume Flag)** — If RF = 1, the 80386 ignores debug faults and does not take another exception so that an instruction can be restarted after a normal debug exception. If RF = 0, the 80386 takes another debug exception to service debug faults.
- **VM (Virtual 8086 Mode)** — When VM bit is set to one, the 80386 executes 8086 programs. When VM bit is zero, the 80386 operates in the protected mode.

The RF, NT, DF, and TF can be set or reset by an 80386 program executing at any privilege level. The VM and IOPL bits can be modified by a program running at only privilege level 0 (the highest privilege level). An 80386 with I/O privilege level can only modify the IF bit. The IRET instruction or a task switch can set or reset the RF and VM bits. The other control bits can also be modified by the POPF instruction.

The instruction pointer register (EIP) contains the offset address relative to the start of the current code segment of the next sequential instruction to be executed. The EIP is not directly accessible by the programmer; it is controlled implicitly by control-transfer instructions, interrupts, and exceptions. The low-order 16 bits of EIP is called IP and is useful when the 80386 executes 8086/80186/80286 instructions.

4.2.1.d 80386 Addressing Modes

The 80386 has 11 addressing modes which are classified into register/immediate and memory addressing modes. Register/immediate type includes two addressing modes, while the memory addressing type contains the other nine modes.

1. Register/Immediate Modes — Instructions using these register or immediate modes operate on either register or immediate operands.
 - i) Register Mode — The operand is contained in one of the 8, 16, or 32-bit general registers. An example is DEC ECX which decrements the 32-bit register ECX by one.
 - ii) Immediate Mode — The operand is included as part of the instruction. An example is MOV EDX, 5167812FH which moves the 32-bit data 5167812F₁₆ to EDX register. Note that the source operand in this case is in immediate mode.
2. Memory Addressing Modes — The other 9 addressing modes specify the effective memory address of an operand. These modes are used when accessing memory. An 80386 address consists of two parts: a segment base address and an effective address. The effective address is computed by adding any combination of the following four components:
 - Displacement: 8- or 32-bit immediate data following the instruction; 16-bit displacements can be used by inserting an address prefix before the instruction.
 - Base: The contents of any general-purpose register can be used as base. Compilers normally use these base registers to point to the beginning of the local variable area.
 - Index: The contents of any general-purpose register except ESP can be used as an index register. The elements of an array or a string of characters can be accessed via the index register.
 - Scale: The index register's contents can be multiplied (scaled) by a factor of 1, 2, 4, or 8. Scaled index mode is efficient for accessing arrays or structures.

The nine memory addressing modes are a combination of the above four elements. Of these nine modes, eight of them are executed with the same number of clock cycles, since the Effective Address calculation is pipelined with the execution of other instructions; the mode containing base, index, and displacement components requires additional clocks.

As shown in Figure 4.11, the Effective Address (EA) of an operand is computed according to the following formula:

$$EA = \text{Base reg} + (\text{Index Reg} * \text{Scaling}) + \text{Displacement}$$

1. Direct Mode — The operand's effective address is included as part of the instruction as an 8-, 16-, or 32-bit displacement. An example is DEC WORDPTR [4000H].
2. Register Indirect Mode — A base or index register contains the operand's effective address. An example is MOV EBX, [ECX].
3. Based Mode — The contents of a base register are added to a displacement to obtain the operand's effective address. An example is MOV [EDX + 16], EBX.
4. Index Mode — The contents of an index register are added to a displacement to obtain the operand's effective address. An example is ADD START [EDI], EBX.
5. Scaled Index Mode — The contents of an index register are multiplied by a scaling factor (1, 2, 4, or 8) which is added to a displacement to obtain the operand's effective address. An example is MOV START [EBX * 8], ECX.
6. Based Index Mode — The contents of a base register are added to the contents of an index register to obtain the operand's effective address. An example is MOV ECX, [ESI] [EAX].
7. Based Scaled Index Mode — The contents of an index register are multiplied by a scaling factor (1, 2, 4, or 8) and the result is added to the contents of a base register to determine the operand's effective address. An example is MOV [ECX * 4] [EDX], EAX.

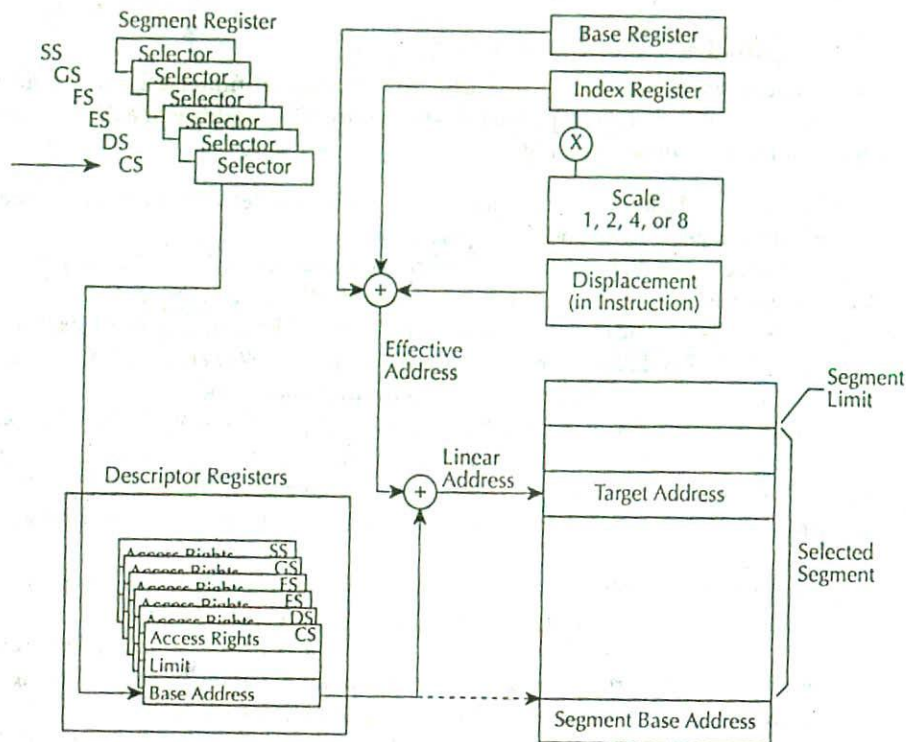


FIGURE 4.11 Addressing mode calculations.

8. Based Index Mode with Displacement — The operand's effective address is obtained by adding the contents of a base register and an index register with a displacement. An example is `MOV [EBX] [EBP + 0F24782AH], ECX`.
9. Based Scaled Index Mode with Displacement — The contents of an index register are multiplied by a scaling factor, and the result is added to the contents of a base register and a displacement to obtain the operand's effective address. An example is `MOV [ESI*8] [EBP + 60H], ECX`.

The 80386 can execute 8086/80186/80286 16-bit instructions in real and protected modes. This is provided in order to make the 80386 software compatible with the 80286, 80186, and the 8086. The 80386 uses the D bit in the segment descriptor register (8 bytes wide) to determine whether the instruction size is 16 or 32 bits wide. If $D = 0$, the 80386 uses all operand lengths and effective addresses as 16 bits long. On the other hand, if $D = 1$, then the default length for operands and addresses is 32 bits. Note that in the protected mode, the operating system can set or reset the D bit using proper instructions. In real mode, the default size for operands and addresses is 16 bits. Note that real address mode does not use descriptors.

Irrespective of the D-bit definition, the 80386 can execute either 16- or 32-bit instructions via the use of two override prefixes such as operand size prefix and address length prefix. These prefixes override the D bit on an individual instruction basis. These prefixes are automatically included by Intel assemblers. For example, if $D = 1$ and the 80386 wants to execute `INC WORD PTR [BX]` to increment a 16-bit memory location, the assembler automatically adds the operand length prefix to specify only a 16-bit value.

The 80386 uses either 8- or 32-bit displacements and any register as base or index register while executing a 32-bit code. However, the 80386 uses either 8- or 16-bit displacements with the base and index registers conforming to the 80286 while executing 16-bit code. The base and index registers utilized by the 80386 for 16- and 32-bit addresses are given in the following:

	16-bit addressing	32-bit addressing
Base Register	BX, BP	Any 32-bit general-purpose register
Index Register	SI, DI	Any 32-bit general-purpose register except ESP
Scale Factor	None	1, 2, 4, 8
Displacement	0, 8, 16 bits	0, 8, 32 bits

4.2.2 80386 Instruction Set

The 80386 extends the 8086/80186/80286 instruction set in two ways: 32-bit forms of all 16-bit instructions are included to support the 32-bit data types and 32-bit addressing modes are provided for all memory reference instructions. The 32-bit extension of the 8086/80186/80286 instruction set is accomplished by the 80386 via the default bit (D) in the code segment descriptor and by having 2 prefixes to the instruction set.

The 80386 instruction set is divided into nine types:

- Data transfer
- Arithmetic
- String
- Logical
- Bit manipulation
- Program Control
- High-level language
- Protection Model
- Processor control

These instructions are listed in Table 4.1.

TABLE 4.1 80386 Instructions

Data Transfer Instructions	
General purpose	
MOV	Move operand
PUSH	Push operand onto stack
POP	Pop operand off stack
PUSHA	Push all registers on stack
POPA	Pop all registers off stack
XCHG	Exchange operand, register
XLAT	Translate
Conversion	
MOVZX	Move Byte or Word, Dword, with zero extension
MOVSX	Move Byte or Word, Dword, sign extended
CBW	Convert Byte to Word, or Word to Dword
CDW	Convert Word to Dword
CDQE	Convert Word to Dword extended
CDQ	Convert Dword to Qword
Input/output	
IN	Input operand from I/O space
OUT	Output operand to I/O space
Address object	
LEA	Load effective address
LDS	Load pointer into D segment register
LES	Load pointer into E segment register
LFS	Load pointer into F segment register
LGS	Load pointer into G segment register
LSS	Load pointer into S (stack) segment register
Flag manipulation	
LAHF	Load AH register from flags

TABLE 4.1 80386 Instructions (*continued*)

Data Transfer Instructions	
SAHF	Store AH register in flags
PUSHF	Push flags onto stack
POPF	Pop flags off stack
PUSHFD	Push Eflags onto stack
POPPD	Pop Eflags off stack
CLC	Clear carry flag
CLD	Clear direction flag
CMC	Complement carry flag
STC	Set carry flag
STD	Set direction flag
Arithmetic Instructions	
Addition	
ADD	Add operand
ADC	Add with carry
INC	Increment operand by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
Subtraction	
SUB	Subtract operand
SBB	Subtract with borrow
DEC	Decrement operand by 1
NEG	Negate operand
CMP	Compare operands
AAS	ASCII adjust for subtraction
Multiplication	
MUL	Multiply double/single precision
IMUL	Integer multiply
AAM	ASCII adjust after multiply
Division	
DIV	Divide unsigned
IDIV	Integer divide
AAD	ASCII adjust after division
String Instructions	
MOVS	Move Byte or Word, Dword string
INS	Input string from I/O space
OUTS	Output string to I/O space
CMPS	Compare Byte or Word, Dword string
SCAS	Scan Byte or Word, Dword string
LODS	Load Byte or Word, Dword string
STOS	Store Byte or Word, Dword string
REP	Repeat
REPE/REPZ	Repeat while equal/zero
RENE/REPZ	Repeat while not equal/not zero
Logical Instructions	
Logicals	
NOT	"NOT" operand
AND	"AND" operand
OR	"Inclusive OR" operand
XOR	"Exclusive OR" operand
TEST	"Test" operand
Shifts	
SHL/SHR	Shift logical left or right
SAL/SAR	Shift arithmetic left or right
SHLD/SHRD	Double shift left or right

TABLE 4.1 80386 Instructions (*continued*)

Rotates	
ROL/ROR	Rotate left/right
RCL/RCR	Rotate through carry left/right
Bit Manipulation Instructions	
Single bit instructions	
BT	Bit test
BTS	Bit test and set
BTR	Bit test and reset
BTC	Bit test and complement
BSF	Bit scan forward
BSR	Bit scan reverse
Bit string instructions	
IBTS	Insert bit string
XBTS	Exact bit string
Program Control Instructions	
Conditional transfers	
SETCC	Set byte equal to condition code
JA/JNBE	Jump if above/not below nor equal
JAЕ/JNB	Jump if above or equal/not below
JB/JNAE	Jump if below/not above nor equal
JBE/JNA	Jump if below or equal/not above
JC	Jump if carry
JE/JZ	Jump if equal/zero
JG/JNLE	Jump if greater/not less nor equal
JGE/JNL	Jump if greater or equal/not less
JL/JNGE	Jump if less/not greater nor equal
JLE/JNG	Jump if less or equal/not greater
JNC	Jump if not carry
JNE/JNZ	Jump if not equal/not zero
JNO	Jump if not overflow
JNP/JPO	Jump if not parity/parity odd
JNS	Jump if not sign
JO	Jump if overflow
JP/JPE	Jump if parity/parity even
JS	Jump if sign
Unconditional transfers	
CALL	Call procedure/task
RET	Return from procedure/task
JMP	Jump
Iteration controls	
LOOP	Loop
LOOPE/LOOPZ	Loop if equal/zero
LOOPNE/	Loop if not equal/not zero
LLOPNZ	
JCXZ	JUMP if register CX = 0
Interrupts	
INT	Interrupt
INTO	Interrupt if overflow
IRET	Return from interrupt
CLI	Clear interrupt enable
STI	Set interrupt enable
High Level Language Instructions	
BOUND	Check array bounds

TABLE 4.1 80386 Instructions (*continued*)

ENTER	Setup parameter block for entering procedure
LEAVE	Leave procedure
Protection Model	
SGDT	Store global descriptor table
SIDT	Store interrupt descriptor table
STR	Store task register
SLDT	Store local descriptor table
LGDT	Load global dedcriptor table
LIDT	Load interrupt descriptor table
LTR	Load task register
LLDT	Load local descriptor table
ARPL	Adjust requested privilege level
LAR	Load access rights
LSL	Load segment limit
VERR/VERW	Verify segment for reading or writing
LMSW	Load machine status word (lower 16 bits of CR0)
SMSW	Store machine status word
Processor Control Instructions	
HLT	Halt
WAIT	Wait until BUSY # negated
ESC	Escape
LOCK	Lock bus

The 80386 instructions include zero-operand, single-operand, two-operand, and three-operand instructions. Most zero-operand instructions such as STC occupy only one byte. Single operand instructions are usually two bytes wide. The two-operand instructions usually allow the following types of operations:

- Register-to-register
- Memory-to-register
- Immediate-to-register
- Memory-to-memory
- Register-to-memory
- Immediate-to-memory

The operands can be either 8, 16, or 32 bits wide. In general, operands are 8 or 32 bits long when the 80386 executes the 32-bit code. On the other hand, operands are 8 or 16 bits wide when the 80386 executes the existing 80286 or 8086 code (16-bit code). Prefixes can be added to all instructions which override the default length of the operands. That is, 32-bit operands for 16-bit code or 16-bit operands for 32-bit code can be used.

The 80386 various instructions affecting the status flags are summarized in Table 4.2.

Table 4.3 lists the various conditions referring to the relation between two numbers (signed and unsigned) for Jcond and SETcond instructions.

All new 80386 instructions along with those which have minor variations from the 80286 are listed in alphabetical order below in Table 4.4.

A detailed description of most of the new 80386 instructions is given in the following.

TABLE 4.2 Status Flag Summary

Status Flag Functions		
Bit	Name	Function
0	CF	Carry flag — Set on high-order bit carry or borrow; cleared otherwise
2	PF	Parity flag — Set if low-order eight bits of result contain an even number of 1 bits; cleared otherwise
4	AF	Adjust flag — Set on carry from or borrow to the low-order four bits of AL; cleared otherwise; used for decimal arithmetic
6	ZF	Zero flag — Set if result is zero; cleared otherwise
7	SF	Sign flag — Set equal to high-order bit of result (0 is positive, 1 is negative)
11	OF	Overflow flag — Set if result is too large a positive number or too small a negative number (excluding sign-bit) to fit in destination operand; cleared otherwise

Key to Codes	
T	Instruction tests flag
M	Instruction modifies flag (either sets or resets depending on operands)
0	Instruction resets flag
—	Instruction's effect on flag is undefined
Blank	Instruction does not affect flag

Instruction	OF	SF	ZF	AF	PF	CF
AAS	—	—	—	TM	—	M
AAD	—	M	M	—	M	—
AAM	—	M	M	—	M	—
DAA	—	M	M	TM	M	TM
DAS	—	M	M	TM	M	TM
ADC	M	M	M	M	M	TM
ADD	M	M	M	M	M	M
SBB	M	M	M	M	M	TM
SUB	M	M	M	M	M	M
CMP	M	M	M	M	M	M
CMPS	M	M	M	M	M	M
SCAS	M	M	M	M	M	M
NEG	M	M	M	M	M	M
DEC	M	M	M	M	M	
INC	M	M	M	M	M	
IMUL	M	—	—	—	—	M
MUL	M	—	—	—	—	M
RCL/RCR 1	M					TM
RCL/RCR count	—					TM
ROL/ROR 1	M					M
ROL/ROR count	—					M
SAL/SAR/SHL/SHR 1	M	M	M	—	M	M
SAL/SAR/SHL/SHR count	—	M	M	—	M	M
SHLD/SHRD	—	M	M	—	M	M
BSF/BSR	—	—	M	—	—	—
BT/BTS/BTR/BTC	—	—	—	—	—	M
AND	0	M	M	—	M	0
OR	0	M	M	—	M	0
TEST	0	M	M	—	M	0
XOR	0	M	M	—	M	0

TABLE 4.3 Condition Codes
(For Conditional Instructions Jcond, and SETcond)

Mnemonic	Meaning	Condition tested
O	Overflow	OF = 1
NO	No overflow	OF = 0
B	Below	
NAE	Neither above nor equal	CF = 1
NB	Not below	CF = 0
AE	Above or equal	
E	Equal	ZF = 1
Z	Zero	
NE	Not equal	ZF = 0
NZ	Not zero	
BE	Below or equal	(CF or ZF) = 1
NA	Not above	
NBE	Neither below nor equal	(CF or ZF) = 0
A	Above	
S	Sign	SF = 1
NS	No sign	SF = 0
P	Parity	PF = 1
PE	Parity even	
NP	No parity	PF = 0
PO	Parity odd	
L	Less	(SF \oplus OF) = 1
NGE	Neither greater nor equal	
NL	Not less	(SF \oplus OF) = 0
GE	Greater or equal	
LE	Less or equal	((SF \oplus OF) or ZF) = 1
NG	Not greater	
NLE	Neither less nor equal	((SF \oplus OF) or ZF) = 0
G	Greater	

Note: The terms "above" and "below" refer to the relation between two unsigned values (neither SF nor OF is tested). The terms "greater" and "less" refer to the relation between two signed values (SF and OF are tested).

TABLE 4.4 80386 Instructions (New Instructions beyond Those of 8086/80186/80286)

Instruction	Comment
ADC EAX, imm32	Add CF with sign-extended immediate byte and 32-bit data in reg32 or mem32
ADC reg32/mem32, imm32	
ADC reg32/mem32, imm8	
ADC reg32/mem32, reg32	
ADC reg32, reg32/mem32	Immediate data byte is sign-extended before addition
ADD EAX, imm32	
ADD reg32/mem32, imm32	
ADD reg32/mem32, imm8	
ADD reg32/mem32, reg32	
ADD reg32, reg32/mem32	
AND EAX, imm32	
AND reg32/mem32, imm32	
AND reg32/mem32, imm8	
AND reg32/mem32, reg32	
AND reg32, reg32/mem32	
BOUND reg32, mem64	
BSF	Bit scan forward
BSR	Bit scan reverse

TABLE 4.4 80386 Instructions (New Instructions beyond Those of 8086/80186/80286) (continued)

Instruction	Comment
BT	Bit test
BTC	Bit test and complement
BTR	Bit test and reset
BTS	Bit test and set
CALL label	There are several variations of the label such as disp16, disp32, reg16, reg32, mem16, mem32, ptr16:16, ptr16:32, mem16:16, and mem16:32; NEAR CALLS use reg16/mem16, reg32/mem32, and disp16/disp32 and the CALL is in the same segment with CS unchanged; CALL disp16 or disp32 adds a signed 16- or 32-bit offset to the address of the next instruction (current EIP) and the result is stored in EIP; when disp16 is used, the upper 16 bits of EIP are cleared to zero; CALL reg16/mem16 or reg32/mem32 specifies a register or memory location from which the offset is obtained; near-return instruction should be used for these CALL instructions; the far calls CALL ptr16:16 or ptr16:32 uses a four-byte or six-byte operand as a long pointer to the procedure called; the CALL mem16:16 or mem16:32 reads the long pointer from the memory location specified (indirection); in real-address mode or virtual 86 mode, the long pointer provides 16 bits for CS and 16 or 32 bits for EIP depending on operand size; the far-call instruction pushes CS and IP (or EIP) and return addresses; in protected mode, both long pointers have different meaning; consult Intel manuals for details
CDQ	Convert doubleword to quadword EDX: EAX ← sign extend EAX
CMP reg32/mem32, imm32	<div style="display: inline-block; vertical-align: middle;"> <div style="border-left: 1px solid black; border-right: 1px solid black; padding: 0 5px; margin: 0 5px;">reg32 or mem32</div> <div style="vertical-align: middle;">- imm32 ⇒ affects flags</div> </div>
CMP EAX, imm32	
CMP reg32/mem32, imm8	
CMP reg32/mem32, reg32	
CMP reg32, reg32/mem32	
CMPSD	
CMPSD	
CMPD	
CWDE	EAX ← sign extend AX
DEC reg32/mem32	
DIV EAX, reg32/mem32	Unsigned divide EDX:EAX by reg32 or mem32 (EAX = quotient, EDX = remainder)
ENTER imm16, imm8	Create a stack frame before entering a procedure
IDIV EAX, reg32/mem32	Signed divide; everything else is same as DIV
IMUL reg32/mem32	Signed multiply EDX: EAX ← EAX * reg32 or mem32
IMUL reg32, reg32/mem32	reg32 ← reg32 * reg32 or mem32; upper 32 bits of the product are discarded
IMUL reg16, reg16/mem16	reg16 ← reg16 * reg16/mem16; result is low 16 bits of the product
IMUL reg16, reg16/mem16, imm8	reg16 ← reg16/mem16 * (sign extended imm8); result is low 16 bits of the product
IMUL reg32, reg32/mem32, imm8	reg32 ← reg32/mem32 * (sign extended imm8); result is low 32 bits of the product
IMUL reg16, imm8	In all IMUL instructions, size of the result is defined by the size of the destination (first) operand.
IMUL reg32, imm8	
IMUL reg16, reg16/mem16,	
IMUL reg32, reg32/mem32,	
IMUL reg16, imm16	
IMUL reg32, imm32	
IN EAX, imm8	Input 32 bits from immediate port into EAX
IN EAX, DX	Input 32 bits from port DX into EAX
INC reg32/mem32	
INS reg32/mem32, DX or INSD	

TABLE 4.4 80386 Instructions (New Instructions beyond Those of 8086/80186/80286) (continued)

Instruction	Comment
IRET/IRETD	In real-address mode, IRET pops IP, CS, and FLAGS, and IRETD POPS EIP, CS, and EFLAGS; for protection mode, consult Intel manuals
Jcc label	cc can be any of the 31 conditions, including the flag settings and less, greater, above, equal, etc.; label can be disp8 as with the 80286; in 80386, however, one can have disp16 and disp32 as label; all these are signed displacements; the 80386 includes the 80286 JCXZ disp8 instruction; furthermore, the 80386 includes a new instruction called JECXZ disp8; (Jump if ECX = 0)
JMP label	The label can be specified in the same way as the CALL label instruction; see CALL label explanation for near-jump and far-jump explanations
LAR reg32, reg32/mem32	Load access right byte; reg32 ← reg32 or mem32 masked by 00FF00H
LEA reg32, m	Calculate the effective address (offset part) m and store it in reg32
LEAVE	LEAVE releases the stack space used by a procedure for its local variables; LEAVE reverses the action of ENTER instruction; LEAVE sets ESP to EBP and then POPS EBP
LGS/LSS/LDS/LES/LFS	Load full pointer; explained later
LODS mem32 or LODSD	To be explained later
LOOP disp8	To be explained later
LOOP cond disp8	To be explained later
MOV reg32/mem32, reg32	
MOV reg32, reg32/mem32	
MOV EAX, mem32	Move 32 bits from mem32 to EAX
MOV mem32, EAX	
MOV reg32, imm32	
MOV reg32/mem32, imm32	
MOV reg32, CR0/CR2/CR3	CRs are control registers
MOV CR0/CR2/CR3, reg32	
MOV reg32, DR0/DR1/DR2/DR3	DRs are debug registers
MOV reg32, DR6/DR7	
MOV DR0/DR1/DR2/DR3, reg32	
MOV DR6/DR7, reg32	
MOV reg32, TR6/TR7	TRs are test registers
MOV TR6/TR7, reg32	
MOVS mem32, mem32 or MOVSD	To be explained later
MOVSB	Move with sign extend; to be discussed later
MOVZB	Move with zero extend; to be discussed later
MUL EAX, reg32/mem32	Unsigned multiply; EDI: EAX ← EAX * (reg32 or mem32)
NEG reg32/mem32	Two's complement. Negate 32 bits in reg32 or mem32
NOT reg32/mem32	Ones complement
OR d,s	The definitions for d and s are same as AND
OUT imm8, EAX	Output 32-bit EAX to immediate port number
OUT DX, reg32/mem32 or OUTSD	To be explained later
POP reg32	To be explained later
POP mem32	To be explained later
POP FS	To be explained later
POP GS	To be explained later
POPAD	To be explained later
POPF	To be explained later
PUSH reg32	To be explained later
PUSH mem32	To be explained later
PUSH FS	To be explained later
PUSH GS	To be explained later
PUSHAD	To be explained later
PUSHFD	To be explained later
RCL reg32/mem32, 1	Rotate reg32 or mem32 thru CF once to left

TABLE 4.4 80386 Instructions (New Instructions beyond Those of 8086/80186/80286) (continued)

Instruction	Comment
RCL reg32/mem32, CL	Rotate reg32 or mem32 thru CF to left CL times
RCL reg32/mem32, imm8	
RCR reg32/mem32, 1	
RCR reg32/mem32, CL	
RCR reg32/mem32, imm8	
ROL reg32/mem32, 1	
ROL reg32/mem32, CL	
ROL reg32/mem32, imm8	
ROR reg32/mem32, 1	
ROR reg32/mem32, CL	
ROR reg32/mem32, imm8	d and n have same definitions as RCL/RCR/ROL/ROR d and s have same definitions as ADD d,s To be explained later To be explained later To be explained later To be explained later To be explained later d and s have same definitions as ADD
SAL/SAR/SHL/SHR d, n	
SBB d,s	
SCAS mem32 or SCASD	
SET cc	
SHLD	
SHRD	
STOS mem32 or STOSD	
SUB d,s	
TEST EAX, imm32	
TEST reg32/mem32, imm32	Exchange 32-bit register contents with EAX
TEST reg32/mem32, reg32	
XCHG reg32, EAX	
XCHG EAX, reg32	
XCHG reg32/mem32, reg32	Set AL to memory byte DS: [EBX + unsigned AL]. DS cannot be overridden. In XLAT, DS can be overridden. d and s have same definitions as AND
XCHG reg32, reg32/mem32	
XLATB	
XOR d,s	

Note: All MUL operands are same as IMUL operands.

4.2.2.a Sign-Extension Instructions

There are two new instructions beyond those of 80286. These are CWDE and CDQ. CWDE sign extends the 16-bit contents of AX to a 32-bit doubleword in EAX. CDQ instruction sign extends a doubleword (32 bits) in EAX to a quadword (64 bits) in EDX: EAX.

4.2.2.b Bit Manipulation Instructions

The following lists the 80386 six-bit manipulation instructions:

BSF	Bit scan forward
BSR	Bit scan reverse
BT	Bit test
BTC	Bit test and complement
BTR	Bit test and reset
BTS	Bit test and set

The above instructions are discussed in the following:

- BSF (Bit Scan Forward)

```
BSF  d , s
      reg16, reg16
      reg16, mem16
      reg32, reg32
      reg32, mem32
```


The 16-bit (word) or 32-bit (doubleword) number defined by *s* is scanned (checked) from right to left (bit 0 to bit 15 or bit 31). The bit number of the first one found is stored in *d*. If the whole 16-bit or 32-bit number is zero, the zero flag is set to one; if a one is found, the zero flag is reset to zero. For example, consider BSF EBX, EDX. If [EDX] = 01241240₁₆, then [EBX] = 00000006₁₆ and ZF = 0. This is because the bit number 6 in EDX (contained in second nibble of EDX) is the first one when EDX is scanned from the right.

- BSR (Bit Scan Reverse)

```
BSR  d , s
      reg16, reg16
      reg16, mem16
      reg32, reg32
      reg32, mem32
```

BSR scans or checks a 16-bit or 32-bit number specified by *s* from the most significant bit (bit 15 or bit 31) to the least significant bit (bit 0). The destination operand *d* is loaded with the bit index (bit number) of the first set bit. If the bits in the number are all zero, the ZF is set to one and operand *d* is undefined; ZF is reset to zero if a one is found.

- BT (Bit Test)

```
BT   d , s
      reg16, reg16
      mem16, reg16
      reg16, imm8
      mem16, imm8
      reg32, reg32
      mem32, reg32
      reg32, imm8
      mem32, imm8
```

BT assigns the bit value of operand *d* (base) specified by operand *s* (the bit offset) to the carry flag. Only the CF is affected. If operand *s* is an immediate data, only eight bits are allowed in the instruction. This operand is taken modulo 32, so the range of immediate bit offset is from 0 to 31. This permits any bit within a register to be selected. If *d* is a register, the bit value assigned to CF is defined by the value of the bit number defined by *s* taken modulo the register size (16 or 32). If *d* is a memory bit string, the desired 16-bit or 32-bit can be determined by adding *s* (bit index) divided by operand size (16 or 32) to the memory address of *d*. The bit within this 16- or 32-bit word is defined by *d* modulo the operand size (16 or 32). If *d* is a memory operand, the 80386 may access four bytes in memory starting at Effective address + (4 * [bit offset divided by 32]). As an example, consider BT CX, DX. If [CX] = 081F₁₆, [DX] = 0021₁₆, then since the content of DX is 33₁₀, the bit number one (remainder of 33/16 = 1) of CX (value 1) is reflected in the CF and therefore CF = 1.

- BTC (Bit Test and Complement)

```
BTC  d , s
```

d and *s* have the same definitions as the BT instruction. The bit of *d* defined by *s* is reflected in the CF. After CF is assigned, the same bit of *d* defined by *s* is ones

complemented. The 80386 determines the bit number from *s* (whether *s* is immediate data or register) and *d* (whether *d* is register or memory bit string) in the same way as the BT instruction.

- BTR (Bit Test and Reset)

BTR *d* , *s*

d and *s* have the same definitions as for the BT instruction. The bit of *d* defined by *s* is reflected in CF. After CF is assigned, the same bit of *d* defined by *s* is reset to zero. Everything else that is applicable to the BT instruction also applies to BTR.

- BTS (Bit Test and Set)

BTS *d* , *s*

Same as BTR except the specified bit in *d* is set to one after the bit value of *d* defined by *s* is reflected into CF. Everything else applicable to the BT instruction also applies to BTS.

4.2.2.c Byte-Set-On Condition Instructions

These instructions set a byte to one or reset a byte to zero depending on any of the 16 conditions defined by the status flags. The byte may be located in memory or in a one-byte general register. These instructions are very useful in implementing Boolean expressions in high level languages such as Pascal. The general structure of this instruction is SETcc (set byte on condition cc) which sets a byte to one if condition cc is true; or else, reset the byte to zero. The following is a list of these instructions:

Instruction	Condition codes	Description
SETA/SETNBE reg8/mem8	CF = 0 and ZF = 0	Set byte if above or not below/equal
SETAE/SETNB/SETNC reg8/mem8	CF = 0	Set if above/equal, set if not below, or set if not carry
SETB/SETNAE/SETC reg8/mem8	CF = 1	Set if below, set if not above/equal, or set if carry
SETBE/SETNA reg8/mem8	CF = 1 or ZF = 1	Set if below/equal or set if not above
SETE/SETZ reg8/mem8	ZF = 1	Set if equal or set if zero
SETG/SETNLE reg8/mem8	ZF = 0 or SF = OF	Set if greater or set if not less/equal
SETGE/SETNL reg8/mem8	SF = OF	Set if greater/equal or set if not less
SETL/SETNGE reg8/mem8	SF ≠ OF	Set if less or set if not greater/equal
SETLE/SETNG reg8/mem8	ZF = 1 and SF ≠ OF	Set if less/equal or set if not greater
SETNE/SETNZ reg8/mem8	ZF = 0	Set if not equal or set if not zero
SETNO reg8/mem8	OF = 0	Set if no overflow
SETNP/SETPO reg8/mem8	PF = 0	Set if no parity or set if parity odd
SETNS reg8/mem8	SF = 0	Set if not sign
SETO reg8/mem8	OF = 1	Set if overflow
SETP/SETPE reg8/mem8	PF = 1	Set if parity or set if parity even
SETS reg8/mem8	SF = 1	Set if sign

As an example, consider SETB BL. If [BL] = 52₁₆ and CF = 1, then after this instruction is executed [BL] = 01₁₆ and CF remains at 1; all other flags (OF, SF, ZF, AF, PF) are undefined. On the other hand, if CF = 0, then after execution of SETB BL, BL contains 00₁₆, CF = 0 and ZF = 1; all other flags are undefined. Similarly, the other SETcc instructions can be explained.

4.2.2.d Conditional Jumps and Loops

JECXZ disp8 jumps if ECX is zero. disp8 means a relative address range from 128 bytes before the end of the instruction. JECXZ tests the contents of ECX register for zero and not the flags.

If $[ECX] = 0$, then after execution of JECXZ instruction, the program branches with signed 8-bit relative offset ($+127_{10}$ to -128_{10} with 0 being positive) defined by disp8 .

JECXZ instruction is useful at the beginning of a conditional loop that terminates with a conditional loop instruction such as LOOPNE label. The JECXZ prevents entering the loop with $ECX = 0$, which would cause the loop to execute up to 2^{32} times instead of zero times.

LOOP Instructions

Instruction	Description
LOOP disp8	Decrement CX/ECX by one and jump if CX/ECX $\neq 0$
LOOPE/LOOPZ disp8	Decrement CX/ECX by one and jump if CX/ECX $\neq 0$ and ZF = 1
LOOPNE/LOOPNZ disp8	Decrement CX/ECX by one and jump if CX/ECX $\neq 0$ and ZF = 0

The 80386 LOOP instructions are similar to those of 8086/80186/80286, except that if the counter is more than 16 bits, ECX rather than CX register is used as the counter.

4.2.2.e Data Transfer

- Move instructions description

```

MOVSB  d,s  Move and sign extend
MOVZX  d,s  Move and zero extend

```

The d and s operands are defined as follows:

```

MOVSB  d,s
or
MOVZX  reg16, reg8
        reg16, mem8
        reg32, reg8
        reg32, mem8
        reg32, reg16
        reg32, mem16

```

MOVSB reads the contents of the effective address or register as a byte or a word from the source and sign-extends the value to the operand size of the destination (16 or 32 bits) and stores the result in the destination. No flags are affected. MOVZX, on the other hand, reads the contents of the effective address or register as a byte or a word and zero-extends the value to the operand size of the destination (16 or 32 bits) and stores the result in the destination. No flags are affected. For example, consider MOVSB BX, CL. If $CL = 81_{16}$ and $[BX] = 21AF_{16}$, then after execution of MOVSB BX, CL, register BX will contain $FF81_{16}$ and CL contents do not change. Also, consider MOVZX CX, DH. If $CX = F237_{16}$ and $[DH] = 85_{16}$, then after execution of this MOVZX, CX register will contain 0085 and DH contents do not change.

- PUSHAD and POPAD Instructions — There are two new PUSH and POP instructions in the 80386 beyond those of 80286. These are PUSHAD and POPAD. PUSHAD saves all 32-bit general registers (the order is EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI) onto the 80386 stack. PUSHAD decrements the stack pointer (ESP) by 32_{10} to hold the eight 32-bit values. No flags are affected. POPAD reverses a previous PUSHAD. It pops the eight 32-bit registers (the order is EDI, ESI, EBP, ESP, EBX, EDX, ECS, and EAX). The ESP value is discarded instead of loading onto ESP. No flags are

affected. Note that ESP is actually popped but thrown away, so that [ESP], after popping all the registers, will be incremented by 32₁₀.

- **Load Pointer Instruction** — There are five instructions in this category. These are LDS, LES, LFS, LGS, and LSS. The first two instructions LDS and LES are available in the 80286. However, the 80286 loads 32 bits from a specified location (16-bit offset and DS) into a specified 16-bit register such as BX and the other into DS for LDS or ES for LES. The 80386, on the other hand, can have four versions of these instructions as follows:

```
LDS    reg16, mem16: mem16
LDS    reg32, mem16: mem32
LES    reg16, mem16: mem16
LES    reg32, mem16: mem32
```

Note that mem16: mem16 or mem16: mem32 defines a memory operand containing four pointers composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to the offset. These instructions read a full pointer from memory and store it in the selected segment register: specified register. The instruction loads 16 bits into DS (for LDS) or into ES (for LES). The other register loaded is 32 bits for 32-bit operand size and 16 bits for 16-bit operand size. The 16- and 32-bit registers to be loaded are determined by reg16 or reg32 register specified.

The three new instructions LFS, LGS, and LSS associated with segment registers FS, GS, and SS can similarly be explained.

4.2.2.f Flag Control

There are two new 80386 instructions beyond those of the 80286. These are PUSHFD and POPFD. PUSHFD decrements the stack pointer by 4 and saves the 80386 EFLAGS register to the new top of the stack. No flags are affected. POPFD, on the other hand, pops the 32-bit (doubleword) from the stack-top and stores the value in EFLAGS. All flags except VM and RF are affected.

4.2.2.g Logical

There are two new 80386 logical instructions beyond those of 80286. These are SHLD and SHRD.

Instruction	Description
SHLD d,s, count	Shift left double
SHRD d,s, count	Shift right double

The operands are defined as follows:

d	s	count
reg16	reg16	imm8
mem16	reg16	imm8
reg16	reg16	CL
mem16	reg16	CL
reg32	reg32	CL
mem32	reg32	imm8
reg32	reg32	CL
mem32	reg32	CL

For both SHLD and SHRD, the shift count is defined by the low five bits and, therefore, shifts up to 0 to 31 can be obtained.

SHLD shifts by the specified shift count the contents of *d:s* with the result stored back into *d*; *d* is shifted to the left by the shift count with the low-order bits of *d* being filled from the high-order bits of *s*. The bits in *s* are not altered after shifting. The carry flag becomes the value of the last bit shifted out of the most significant bit of *d*.

If the shift count is zero, the instruction works as a NOP. For a specified shift count, the SF, ZF, and PF flags are set according to the result in *d*. CF is set to the value of the last bit shifted out. OF and AF are undefined.

SHRD, on the other hand, shifts the contents of *d:s* by the specified shift count to the right with the result being stored back into *d*. The bits in *d* are shifted right by the shift count with the high-order bits being filled from the low-order bits of *s*. The bits in *s* are not altered after shifting.

If the shift count is zero, this instruction operates as a NOP. For the specified shift count, the SF, ZF, and PF flags are set according to the value of the result. CF is set to the value of the last bit shifted out. OF and AF are undefined.

As an example, consider SHLD BX, DX, 2. If $[BX] = 183F_{16}$, $[DX] = 01F1_{16}$, then after this SHLD, $[BX] = 60FC_{16}$, $[DX] = 01F1_{16}$, CF = 0, SF = 0, ZF = 0, and PF = 1.

Similarly, the SHRD instruction can be illustrated.

4.2.2.h String

- **Compare String** — There is a new instruction CMPS mem32, mem32 (or CMPSD) beyond the compare string instruction available with the 80286. This instruction compares 32-bit words ES:EDI (second operand) with DS:ESI and affects the flags. The direction of subtraction of CMPS is $[[ESI]] - [[EDI]]$. The left operand ESI is the source and the right operand EDI is the destination. This is a reverse of the normal Intel convention in which the left operand is the destination and the right operand is the source. This is true for byte (CMPSB) or word (CMPSW) compare instructions. The result of subtraction is not stored; only the flags are affected. For the first operand (ESI), the DS is used as segment unless a segment override byte is present, while the second operand (EDI) must use ES as the segment register and cannot be overridden. ESI and EDI are incremented by 4 if DF = 0, while they are decremented by 4 if DF = 1. CMPSD can be preceded by the REPE or REPNE prefix for block comparison. All flags are affected.
- **Load and Move Strings** — There are two new 80386 instructions beyond those of 80286. These are LODS mem32 (or LODSD) and MOVS mem32, mem32 (or MOVSD). LODSD loads the doubleword (32-bit) from a memory location specified by DS:ESI into EAX. After the load, ESI is automatically incremented by 4 if DF = 0, while ESI is automatically decremented by 4 if DF = 1. No flags are affected. LODS can be preceded by REP prefix. LODS is typically used within a loop structure because further processing of the data moved into EAX is normally required. MOVSD copies the doubleword (32-bit) at memory location addressed by DS:ESI to the memory location at ES:EDI. DS is used as the segment register for the source and may be overridden. ES must be used as the segment register and cannot be overridden. After the move, ESI and EDI are incremented by four if DF = 0, while they are decremented by 4 if DF = 1. MOVS can be preceded by the REP prefix for block movement of ECX doublewords. No flags are affected.
- **String I/O Instructions** — There are two new 80386 string I/O instructions beyond those of the 80286. These are INS mem32, DX (or INSD) and OUTS DX, mem32 (or OUTSD). INSD inputs 32-bit data from a port addressed by the content of DX into a memory location specified by ES:EDI. ES cannot be overridden. After data transfer, EDI

is automatically incremented by 4 if DF = 0, while it is decremented by 4 if DF = 1. INSD can be preceded by the REP prefix for block input of ECX doublewords. No flags are affected. OUTSD instruction outputs 32-bit data from a memory location addressed by DS:ESI to a port addressed by the content of DX. DS can be overridden. After data transfer, ESI is incremented by 4 if DF = 0 and decremented by 4 if DF = 1. OUTSD can be preceded by the REP prefix for block output of ECX doublewords.

- **Store and Scan Strings** — There is a new 80386 STOS mem32 (or STOSD) instruction. STOS stores the contents of the EAX register to a doubleword addressed by ES and EDI. ES cannot be overridden. After storing, EDI is automatically incremented by 4 if DF = 0 and decremented by 4 if DF = 1. No flags are affected. STOS can be preceded by the REP prefix for a block fill of ECX doublewords. There is a new scan instruction called the SCAS mem32 (or SCASD) in the 80386. SCASD performs the 32-bit subtraction [EAX] – [memory addressed by ES and EDI]. The result of subtraction is not stored, and the flags are affected. SCASD can be preceded by the REPE or REPNE prefix for block search of ECX doublewords. All flags are affected.

4.2.2.i Table Look-Up Translation Instruction

There is a modified version of the 80286 XLAT instruction available in the 80386.

XLAT mem8 (or XLATB) replaces the AL register from the table index to the table entry. AL should be the unsigned index into a table addressed by DS:BX for 16-bit address (available in 80286 and 80386) and DS:EBX for 32-bit address (available only in 80386). DS can be overridden. No flags are affected.

4.2.2.j High-Level Language Instructions

The three instructions ENTER, LEAVE, and BOUND (also available with 80186/80286) in this category have been enhanced in the 80386.

Before a subroutine is called by a main program, it is required quite often to pass some parameters to the subroutine by the main program. Normally these parameters are pushed onto the stack before calling the subroutine and then they are used by the subroutine by popping them from stack during its execution. In the 80386, a portion of the stack called the stack frame is used to store these parameters. Two 80386 instructions, namely, ENTER and LEAVE, are included for allocating and deallocating stack frames.

The ENTER imm16, imm8 instruction creates a stack frame. The data imm8 defines the nesting depth (also called the lexical level) of the subroutine and can be from 0 to 31. The value 0 specifies the first subroutine only. The data imm16 defines the number of stack frame pointers copied into the new stack frame from the preceding frame.

After the instruction is executed, the 80386 uses EBP as the current frame pointer and ESI as the current stack pointer. The data imm8 specifies the number of bytes of local variables for which the stack space is to be allocated.

ENTER can be used for either nested or non-nested subroutines or procedures. For example, if the lexical level imm8 is zero, the non-nested form is used. If imm8 is zero, ENTER pushes the frame pointer EBP onto the stack; ENTER then subtracts the first operand imm16 from the ESP and sets EBP to the current ESP.

For example, a procedure with 28 bytes of local variables would have an ENTER 28, 0 instruction at its entry point and a LEAVE instruction before every RET. The 28 local bytes would be addressed as offset from EBP. Note that the LEAVE instruction sets ESP to EBP and then pops EBP. For the 80186 and 80286, ENTER and LEAVE instructions use BP and SP instead of EBP and ESP. The 80386 uses BP (low 16 bits of EBP) and SP (low 16 bits of ESP) for 16-bit operands, and EBP and ESP for 32-bit operands.

The formal definition of the ENTER instruction is given in the following:

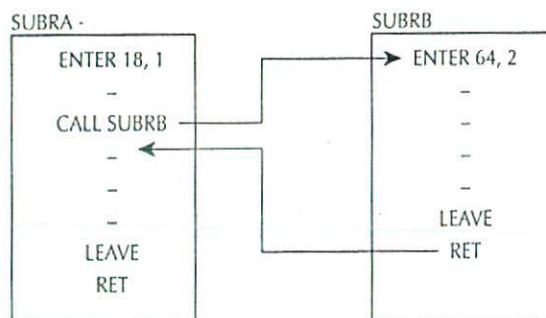
```

LEVEL denotes the value of the second operand, imm8:
Push EBP
Set a temporary value FRAME-PTR: = ESP
If LEVEL > 0 then
    Repeat (LEVEL-1) times:
        EBP: = EBP-4
        Push all EBP for the previous subroutines
        and then EBP for the present subroutine.
    End repeat
Push FRAME-PTR
End if
EBP: = FRAME-PTR
ESP: = ESP - first operand, imm16
  
```

The LEAVE instruction performs the following:

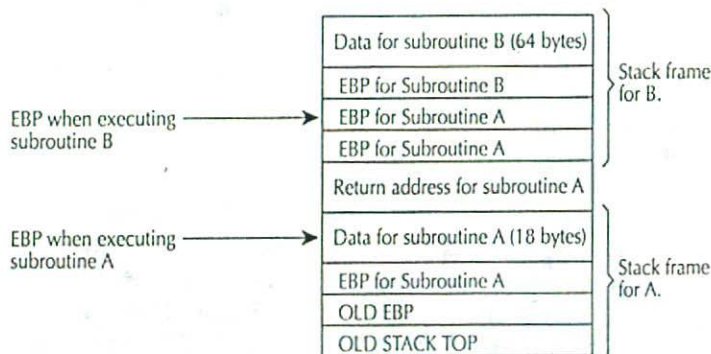
- (ESP) \leftarrow (EBP)
- POP into EBP.

In order to illustrate the Enter and Leave instructions, consider the following:



In the above, subroutine A (SUBRA) calls subroutine B. It is assumed that the nesting depth for these subroutines are 1 and 2 respectively.

The stack frames created after execution of the ENTER instructions in the two subroutines are shown below:



As the ENTER instruction in subroutine A is executed, the old EBP from the subroutine that called SUBRA is pushed onto the stack. EBP is loaded from ESP to point to the location of the

old EBP. The lexical level for the ENTER for subroutine A is pushed onto the stack. Finally, in order to allocate 18 bytes, 12H is subtracted from the current ESP.

After entering into the subroutine B, a second ENTER instruction ENTER 64, 2 is executed. Since the lexical level in this case is 2, the Old EBP (EBP for subroutine A) is first pushed onto the stack. The EBP for subroutine A previously stored on the stack frame is pushed onto the stack. Finally, the current EBP for subroutine B is pushed onto the stack. This mechanism provides access to the stack frame for subroutine A from subroutine B. Next, local storage of 64 bytes as specified in the ENTER instruction are allocated for local storage.

The BOUND instruction checks to determine if the contents of a register called the array index lie within the minimum (lower bound) and maximum (upper bound) limits of an array.

The 80386 provides two forms of the BOUND instruction:

```
BOUND    reg16, mem32
BOUND    reg32, mem64
```

The first form is for 16-bit operands and is also available with the 80186 and 80286. The second form is for 32-bit operands and is included in the 80386 instruction set. For example, consider BOUND EDI, ADDR. Suppose [ADDR] = 32-bit lower bound, d_l and [ADDR + 4] = 32-bit upper bound d_u . If, after execution of this instruction, [EDI] < d_l and > d_u , the 80386 traps to interrupt 5; otherwise the array is accessed.

The BOUND instruction is usually placed following the computation of an index value to ensure that the limits of the index value are not violated. This allows checking whether or not the address of an array being accessed is within the array boundaries when the address register indirect with index mode is used to access an element in the array. For example, the following instruction segment will allow accessing an array with base address in ESI, the index value in EDI, and an array length of 200₁₀ bytes. Assume the 32-bit contents of memory location 52070422₁₆ and 52070426₁₆ are 0 and 199₁₀, respectively:

```

-
-
-
BOUND    EDI, 52070422H
MOV      EAX, [EDI][ESI]
```

In the above, if the contents of EDI are not within the array boundaries of 0 and 199₁₀, the 80386 will trap to interrupts and the MOV instruction will not be executed.

In the following 80386 programming examples, the 80386 is assumed to be real mode. The 80386 assembler directives are not included. These directives are similar to those of the 8086.

Example 4.2

Determine the effect of each one of the following 80386 instructions:

- i) CDQ
- ii) BTC CX, BX
- iii) MOVSX ECX, E7H

Assume [EAX] = FFFFFFFFH, [ECX] = F1257124H, [EDX] = EEEEEEEEH, [BX] = 0004H, [CX] = 0FA1H, prior to execution of each of the above instructions.

- i) After CDQ,
[EAX] = FFFF FFFFH
[EDX] = FFFF FFFFH
- ii) After BTC CX, BX, bit 4 of register CX is reflected in the ZF and then ones complemented in CX.

Before

[CX] = 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Bit no.
 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 1₂

After BTC CX, BX,

[CX] = 0 0 0 0 1 1 1 1 1 0 1 1 0 0 0 1₂

0
F
B
1₁₆

CF=0

1's complement

Hence, [CX] = 0FB1H and [BX] = 0004H

- ii) MOVSX ECX, E7H copies the 8-bit data E7H into low byte of ECX and then sign-extends to 32 bits. Therefore, after MOVSX ECX, E7H, [ECX] = FFFFFFFE7H.

Example 4.3

Write an 80386 assembly language program to multiply a signed 8-bit number by a signed 32-bit number in ECX. Store result in EAX. Assume that the segment registers are already initialized and also that the result fits within 32 bits.

Solution

```
IMUL ECX, data8 ; Perform signed
                  ; multiplication
MOV EAX, ECX    ; Store result
HLT              ; in EAX and stop
```

Example 4.4

Write an 80386 assembly program to move two columns of 10,000 32-bit numbers from A (i) to B (i). In other words, move A (1) to B (1), A (2) to B (2), and so on. Initialize DS to 2000H, ES to 3000H, ESI to 0100H and EDI to 0200H.

Solution

```
MOV ECX, 10000 ; Initialize counter
MOV BX, 2000H  ; Initialize DS
MOV DS, BX     ; register
MOV BX, 3000H  ; Initialize ES
MOV ES, BX     ; register
MOV ESI, 0100H ; Initialize ESI
MOV EDI, 0200H ; Initialize EDI
CLD            ; Clear DF to Autoincrement
REPMOVSD       ; MOV A (i) to
                ; B (i) until ECX = 0
HLT
```

Example 4.5

Identify the addressing modes and the size of the operation for the following instructions:

- i) `MOV EAX, [EBX*4]`
- ii) `ADD AH, [EBX+35][ESI]`

Solution

- | | | |
|-----|---------------|--------------------|
| i) | <u>Source</u> | <u>Destination</u> |
| | Scaled Index | Register |
| | 32-bit | |
| ii) | <u>Source</u> | <u>Destination</u> |
| | Based Indexed | Register |
| | 8-bit | |

Example 4.6

Compute the physical address for the specified operands of the following instructions. Assume $DS = 0300_{16}$, $ESI = 00005000_{16}$, $EBX = 00000200_{16}$.

- i) `MOV BH, [SI]`
- ii) `ADD [BX+50H], CX`

Solution

- i) Physical address for the source $= 03000_{16} + 5000_{16} = 08000_{16}$
Physical address for the source $= 03000_{16} + 0250_{16}$
 $= 03250_{16}$

Example 4.7

Write an 80386 instruction sequence to compute the following:

$$\text{INTEGER} = (\text{INTEGER1} \oplus \text{EDX}) \vee (\text{ECX} \cdot \text{EDX})$$

Assume that the contents of locations INTEGER and INTEGER1 are 32-bit wide.

Solution

```

MOV EAX, EDX
NOT EDX           ; EDX ← NOT EDX
XOR EDX, [INTEGER1] ; EDX ← (INTEGER1) ⊕ EDX
AND ECX, EAX      ; ECX ← ECX · EDX
OR ECX, EDX       ; ECX ← EDX ∨ ECX
MOV [INTEGER], ECX ; [INTEGER] ← ECX

```

Example 4.8

Write an 80386 assembly language program to add two 64-bit binary numbers. The numbers are pointed to by ESI and EDI, respectively. Store result in location pointed to by EDI. Assume data are already loaded in memory locations.

Solutions

```

MOV EAX, [ESI]    ; Load first 64-bit
MOV EBX, [ESI+4]  ; number
ADD [EDI], EAX    ; Add with
ADC [EDI+4], EBX  ; second 64-bit number
HLT

```

Example 4.9

Write an 80386 assembly language program to check whether the 64-bit number stored in memory pointed to by ESI is zero. If it is zero, store 0 in AL; otherwise store 1 in AL.

Solution

```

MOV     EBX, [ESI]    ; Move the upper 32-bit into
                        ; EBX
OR      EBX, [ESI+4]  ; Check whether both halves
                        ; are zero.
JNZ     ZERO          ;
MOV     AL, 1         ; The number is not zero
HLT
ZERO:   MOV AL, 0      ; The number is zero
HLT

```

Example 4.10

Assume the content of physical memory location 21010_{16} is $2F_{16}$, $[ADDR] = 2000_{16}$ and double word stored at location $ADDR+2$ is 02340110_{16} . Find the contents of EAX register after execution of the following 80386 instruction sequence:

```

LDS     EAX, [ADDR]
MOV     EBX, 00001000H
XLAT

```

Solution

LDS instruction in the above loads DS with 2000_{16} and EAX with 02340110_{16} . MOV loads EBX with $00001000H_{16}$. The XLAT instruction loads AL with the contents of memory location addressed by $DS + BX + AL$ which is $20000_{16} + 1000_{16} + 10_{16} = 21010_{16}$. Therefore, $[EAX] = 0234012F_{16}$.

Example 4.11

Write an 80386 assembly language program to multiply an unsigned 32-bit number in EBX by an unsigned 16-bit number in CX. Store result in EDX:EAX.

Solution

```

MOVZX   ECX, CX      ; Zero Extend CX.
MOV     EAX, EBX      ; Move to EAX for multiplication
MUL     EAX, ECX      ; Unsigned multiplication
HLT

```

Example 4.12

Write an 80386 assembly language instruction sequence to extract the bit field EAX[31:24] and store it in EBX[7:0], so that EBX[31:8] is zero and the original EAX is not affected.

Solution

```
MOV    EBX, EAX
MOV    CL, 8
ROL    EBX, CL
AND    EBX, 000000FFH
```

Example 4.13

Write 80386 assembly language program to compute the following: $X = Y + Z - 1F20_{16}$ where X, Y, and Z are 64-bit variables. The lower 32 bits of Y and Z are stored starting at locations NUMBER and NUMBER + 8, each followed by the upper 32 bits. Store the lower 32-bit of the 64-bit result in EAX followed by the upper 32 bits in ECX.

Solution

```
MOV    EAX, [NUMBER]      ; Load EAX with low 32-bit of Y
MOV    EBX, [NUMBER+4]    ; Load EBX with high 32-bit of Y
ADD    EAX, [NUMBER+8]    ; Add low 32 bits
MOV    ECX, [NUMBER+12]   ; Load ECX with high 32-bit of Z
ADC    ECX, EBX           ; Add high 32 bits and carry
SUB    EAX, 1F20H         ; Subtract 1F20H from 32-bit of
                        ; result
MOV    EBX, 0             ; If borrow,
SBB    ECX, EBX           ; Subtract from high 32-bit of result
HLT
```

The 80386 assembly language programs can be assembled by using 80386 assemblers on IBM PC's.

A typical program structure is given below:

```
        PAGE 55, 132      ; Set page dimensions
        .386
STACK   SEGMENT 'STACK' STACK
        DW 50 DUP(?)
STACK   ENDS
PROG    SEGMENT PARA 'CODE' PUBLIC USE16
        ASSUME CS:PROG, DS:DATA, SS:STACK
BEGIN:  MOV BX, 3000H
        MOV DS, BX
        (Specify the constants and variables here)
        (Enter program here)
        MOV AX, 4C00H     ; RETURN
        INT 21H           ; to DOS
PROG    ENDS
        END BEGIN
```


In the above, the PAGE directive in the first line specifies the number of lines on a page and the width of each line for the assembler. The notation .386 in the second line indicates to the assembler that the program includes the 80386 instructions. The next three lines define the stack segment with 50 bytes.

The code segment where the actual program starts is defined next. The logical name of the code segment in the above is PROG. The code segment is public and tells the assembler to use 16-bit registers.

The 80386 uses a new directive USE16 or USE32. Programs that are developed to run on the 8086/80286 must use the USE16 to specify that all operand and address sizes are 16 bits. This automatically limits segment size to 64K. With the USE32, programs are assembled with an operand and address sizes of 32 bits. This allows access to up to 4 gigabytes of memory.

INT 21H with 4C00H in AX returns control to DOS operating system. The 80386 assembly language programs can be assembled with an assembler such as the PHAR LAP '386' assembler. All examples in this chapter are written without the complete 80386 directives. The main purpose is to illustrate use of 80386 instructions for writing assembly language programs.

4.2.3 Memory Organization

Memory on the 80386 is structured as 8-bit (byte), 16-bit (word), or 32-bit (doubleword) quantities. Words are stored in two consecutive bytes with low byte at the lower address and high byte at the higher address. The byte address of the low byte addresses the word. Doublewords are stored in four consecutive bytes in memory with byte 0 at the lowest address and byte 3 at the highest address. The byte address of byte 0 addresses the doubleword.

Memory on the 80386 can also be organized as pages or segments. The entire memory can be divided into one or more variable length segments which can be shared between programs or swapped to disks. Memory can also be divided into one or more 4K-byte pages. Segmentation and paging can also be combined in a system. The 80386 includes three types of address spaces. These are logical or virtual, linear, and physical. A logical or virtual address contains a selector (contents of a segment register) and offset (effective address) obtained by adding the base, index, and displacement components discussed earlier. Since each task on the 80386 can have a maximum of 16K selectors and offsets can be 4 gigabytes (2^{32} bits), the programmer views a virtual address space of 2^{46} or 64 tetrabytes of logical address space per task.

The 80386 on-chip segment unit translates the logical address space to 32-bit linear address space. If the paging unit is disabled, then the 32-bit linear address corresponds to the physical address. On the other hand, if the paging unit is enabled, the paging unit translates the linear address space to the physical address space. Note that the physical addresses are generated by the 80386 on its address pins.

The main difference between real and protected modes is how the 80386 segment unit translates logical addresses to linear addresses. In real mode, the segment unit shifts the selector to the left four times and adds the result to the offset to obtain the linear address. In protected mode, every selector has a linear base address. The linear base address is stored in one of two operating system tables (local descriptor table or global descriptor table). The selector's linear base address is summed with the offset to obtain the linear address. Figure 4.12 shows the 80386 address translation mechanism.

There are three main types of 80386 segments. These are code, data, and stack segments. These segments are of variable size and can be from 1 byte to 4 gigabytes (2^{32} bits) in length.

Instructions do not explicitly define the segment type. This is done in order to obtain compact instruction encoding. A default segment register is automatically selected by the 80386 according to Table 4.5.

In general, DS, SS, and CS use the selectors for data, stack, and code. Segment override prefixes can be used to override a given segment register as per Table 4.5.

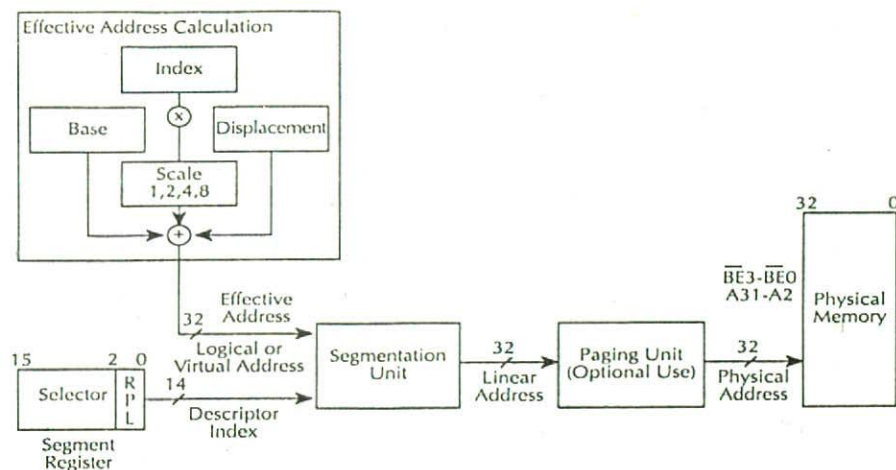


FIGURE 4.12 Address translation mechanism.

4.2.4 I/O Space

The 80386 supports both standard and memory-mapped I/O. The I/O space contains 64K 8-bit ports, 32K 16-bit ports, 16K 32-bit ports, or any combination of ports up to 64K bytes. I/O instructions do not go through the segment or paging units. Therefore, the I/O space refers to physical memory. The M/IO pin distinguishes between the memory and I/O.

The 80386 includes IN and OUT instructions to access I/O ports with port address provided by DL, DX, or EDX registers. All 8- and 16-bit port addresses are zero-extended on the upper address lines. The IN and OUT instructions drive the 80386 M/IO pin to low.

I/O port addresses 00F8H through 00FFH are reserved by Intel. The coprocessors in the I/O space are at locations 800000F8H through 800000FFH.

4.2.5 80386 Interrupts

Earlier interrupts and exceptions which are of interest to application programmers were discussed. In this section, details of these interrupts and exceptions are covered. The difference between interrupts and exceptions is that interrupts are used to handle asynchronous external events and exceptions handle instruction faults. The 80386 also treats software interrupts such as INT n as exceptions.

TABLE 4.5 Segment Register Selection Rules

Type of memory reference	Implied (default) segment use	Segment override prefixes possible
Code Fetch	CS	None
Destination of PUSH, PUSHA instructions	SS	None
Source of POP, POPA instructions	SS	None
Other data references, with effective address using base register of:		
[EAX]	DS	CS,SS,ES,FS,GS
[EBX]	DS	CS,SS,ES,FS,GS
[ECX]	DS	CS,SS,ES,FS,GS
[EDX]	DS	CS,SS,ES,FS,GS
[EBX]	DS	CS,SS,ES,FS,GS
[ESI]	DS	CS,SS,ES,FS,GS
[EDI]	DS	CS,SS,ES,FS,GS
[EBP]	SS	CS,DS,ES,FS,GS
[ESP]	SS	CS,DS,ES,FS,GS

The 80386 interrupts and exceptions are similar to those of the 8086.

There are three types of interrupts/exceptions. These are hardware interrupts, exceptions, and software interrupts.

Hardware interrupts can be of two types. The 80386 provides the NMI pin for the nonmaskable interrupt. When the NMI pin encounters a LOW to HIGH transition by an external device such as an A/D converter, the 80386 services the interrupt via the internally supplied instruction INT2. The INT2 instruction does not need to be provided via external hardware.

The 80386 services a maskable interrupt when its INTR pin is activated HIGH and the IF bit is set to one. An 8-bit vector can be supplied by the user via external hardware which identifies the interrupt source.

The IF bit in the EFLAG registers is reset when an interrupt is being serviced. This, in turn, disables servicing additional interrupts during an interrupt service routine.

When an interrupt occurs, the 80386 completes execution of the current instruction. The 80386 then pushes the EIP, CS, and flags onto the stack. Next, the 80386 obtains an 8-bit vector via either external hardware (maskable) or internally (nonmaskable) which identifies the appropriate entry in the interrupt table. The table contains the starting address of the interrupt service routine. At the end of the interrupt service routine, IRET can be placed to resume the program at the appropriate place in the main program.

The software interrupt due to execution of INT *n* has the same effect as the hardware interrupt. A special case of the software interrupt INT *n* is the INT3 or breakpoint interrupt. Like the 8086, the single-step interrupt is enabled by setting the TF bit. The TF bit is set by altering the stack image and executing a POPF or IRET instruction. The single step uses INT1. Exceptions are classified as faults, traps, or aborts depending on the way they are reported and whether or not the instruction causing the exception is restarted. Faults are exceptions that are detected and serviced before the execution of the faulting instruction. A fault can occur in a virtual memory system when the 80386 references a page or a segment not present in the main memory. The operating system can execute a service routine at the fault's interrupt address vector to fetch the page or segment from disk. Then the 80386 restarts the instruction traps and immediately reports the cause of the problem via the execution of the instruction. Typical examples of traps are user-defined interrupts. Aborts are exceptions which do not allow the exact location of the instruction causing the exception to be determined. Aborts are used to report severe errors such as a hardware error or illegal values in system tables.

Therefore, upon completion of the interrupt service routine, the 80386 resumes program execution at the instruction following the interrupted instruction. On the other hand, the return address from an exception fault routine will always point to the instruction causing the exception. Table 4.6 lists the 80386 interrupts along with to where the return address points.

The 80386 can handle up to 256 different interrupts/exceptions. For servicing the interrupts, a table containing up to 256 interrupt vectors must be defined by the user. These interrupt vectors are pointers to the interrupt service routine. In real mode, the vectors contain two 16-bit words: the code segment and a 16-bit offset. In protected mode, the interrupt vectors are 8-byte quantities, which are stored in an interrupt descriptor table. Of the 256 possible interrupts, 32 are reserved by Intel and the remaining 224 are available to be used by the system designer.

If there are several interrupts/exceptions occurring at the same time, the 80386 handles them according to the following priorities:

Priority	Interrupt/exception
1. (Highest)	Exception faults
2.	TRAP instructions
3.	Debug traps for this instruction
4.	Debug faults for next instruction
5.	NMI
6. (Lowest)	INTR

TABLE 4.6 Interrupt Vector Assignments

Function	Interrupt number	Instruction which can cause exception	Return address points to faulting instruction	Type
Divide error	0	DIV, IDIV	Yes	Fault
Debug exception	1	Any instruction	Yes	Trap ^a
NMI interrupt	2	INT 2 or NMI	No	NMI
One-byte interrupt	3	INT	No	Trap
Interrupt on overflow	4	INTO	No	Trap
Array bounds check	5	BOUND	Yes	Fault
Invalid OP-code	6	Any illegal instruction	Yes	Fault
Device not available	7	ESC, WAIT	Yes	Fault
Double fault	8	Any instruction that can generate an exception		Abort
Coprocessor segment	9	Coprocessor tries to access data past the end of a segment	No	Trap ^b
Invalid TSS	10	JMP, CALL, IRET, INT	Yes	Fault
Segment not present	11	Segment register instructions	Yes	Fault
Stack fault	12	Stack references	Yes	Fault
General protection fault	13	Any memory reference	Yes	Fault
Page fault	14	Any memory access or code fetch	Yes	Fault
Coprocessor error	16	ESC, WAIT	Yes	Fault
Intel reserved	17—32			
Two-byte interrupt	0—255	INT n	No	Trap

^aSome debug exceptions may report both traps on the previous instruction and faults on the next instruction.

^bException 9 no longer occurs on the 80386 due to the improved interface between the 80386 and its coprocessors.

As an example, suppose an instruction causes both a debug exception (interrupt no. 1) and page fault (interrupt vector 14). According to the built-in priority mechanism, the 80386 will first service the page fault by executing the exception 14 handler. The exception 14 handler will be interrupted by the debug exception handler (1). An address in the page fault handler will be pushed onto the stack and the service routine for the debug handler (1) will be completed. After this, the exception 14 handler will be executed. This permits the system designer to debug the exception handler.

In real mode, the 80386 obtains the values of IP and CS similar to the 8086 by using a table called the interrupt pointer table. In protected mode, this table is called Interrupt descriptor table. The 80386 real-mode interrupt pointer table is shown on the following page.

4.2.6 80386 Reset and Initialization

When the 80386 is initialized and reset, the 80386 will start executing instructions near the top of physical memory, at location FFFFFFF0H. When the first Intersegment Jump or call is executed, address lines A20-A31 will drop low, and the 80386 will only execute instructions in the lower one megabyte of physical memory. This allows the system designer to use a ROM at the top of physical memory to initialize the system and take care of Resets. Driving the RESET input pin HIGH for at least 78 CLK2 periods resets the 80386.

Upon hardware reset, the 80386 registers contain the values as shown in Table 4.7.

		Physical Memory Address
Vector Number 255	CS	003FEH
	IP	003FCH
	⋮	
32	CS	00082H
	IP	00080H
	⋮	
Coprorocessor not present	CS	0001EH
	IP	0001CH
Invalid opcode 6	CS	0001AH
	IP	00018H
Bound check 5	CS	00016H
	IP	00014H
OVERFLOW 4	CS	00012H
	IP	00010H
BREAKPOINT #	CS	0000EH
	IP	0000BH
NMI 2	CS	0000AH
	IP	00008H
DEBUG 1	CS	00006H
	IP	00004H
DIVIDE BY 0 ERROR 0	CS	00002H
	IP	00000H

4.2.7 Testability

The 80386 provides capability to perform self-test. The self-test checks all of the control ROM and the associate nonrandom logic inside the 80386. The self-test feature is performed when the 80386 RESET pin goes from HIGH to LOW and the BUSY # pin is LOW. The self-test takes above 30 milliseconds with a 16-MHz clock. After self-test, the 80386 performs reset and begins program execution. If the self-test is successful, the contents of both EAX and EDX are zero; otherwise the contents of EAX and EDX are not zero, indicating a faulty chip.

4.2.8 Debugging

In addition to the software breakpoint and single-stepping features, the 80386 also includes six program-accessible 32-bit registers for specifying up to four distinct breakpoints. Unlike the INT3 which only allows instruction breakpointing, the 80386 debug registers permit breakpoints to be set for data accesses. Therefore, a breakpoint can be set up if a variable is accidentally being overwritten. Thus, the 80386 can stop executing the program whenever the variable's contents are being changed.

4.2.9 80386 Pins and Signals

As mentioned before, the 80386 is a 132-pin ceramic Pin Grid Array (PGA). Pins are arranged 0.1 inch (2.54 mm) center-to-center, in a 14 × 14 matrix, three rows around.

A number of sockets are available for low insertion force or zero insertion force mountings. Three types of terminals include soldertail, surface mount, or wire wrap. These application

TABLE 4.7 Register Values after Reset

Flag word (EFLAGS)	00000002H	Note 1
Machine status word (CR0)	UUUUUUU0H	Note 2
Instruction pointer (IP)	0000FFF0H	
Code segment	F000H	Note 3
Data segment	0000H	
Stack segment	0000H	
Extra segment (ES)	0000H	
Extra segment (FS)	0000H	
Extra segment (GS)	0000H	
All other registers	Undefined	

Note: U means undefined.

Note 1: The upper 14 bits of the EFLAGS registers are zero, VM (Bit 17) and RF (Bit 16) and other defined flag bits are zero.

Note 2: All of the defined fields in the CR0 are 0 (PG Bit 31, TS Bit 3, EM Bit 2, MP Bit 1, and PE Bit 0) except for ET Bit 4 (processor Extension type). The ET Bit is set during Reset according to the type of coprocessor in the system. If the coprocessor is an 80387 then ET will be 1, if the coprocessor is an 80287 or no coprocessor is present then ET will be 0. All other bits are undefined.

Note 3: The code segment Register (CS) will have its base address set to FFF00000H and Limit set to 0FFFFH. All undefined bits are reserved and should not be used.

sockets are manufactured by Amp, Inc. of Harrisburg, PA, Advanced Interconnections of Warwick, RI, and Textool Products of Irving, TX.

Figure 4.13 shows the 80386 pinout as viewed from the pin side of the chip. Table 4.8 provides the 80386 pinout functional grouping description.

Figure 4.14 shows functional grouping of the 80386 pins. A brief description of the 80386 pins and signals is provided in the following. The # symbol at the end of the signal name or the — symbol above a signal name indicates the active or asserted state when it is low. When the symbol # is absent after the signal name or the symbol — is absent above a signal name, the signal is asserted when high.

The 80386 has 20 Vcc and 21 GND pins for power distribution. These multiple power and ground pins reduce noise. Preferably, the circuit board should contain Vcc and GND planes.

CLK2 pin provides the basic timing for the 80386. This clock is divided by 2 internally to provide the internal clock used for instruction execution. The CLK2 signal is specified at CMOS-compatible voltage levels and not at TTL levels.

There are two phases (phase one and phase two) of the internal clock. Each CLK2 period defines a phase of the internal clock. Figure 4.15 shows the relationship. The 80386 is reset by activating the RESET pin for at least 15 CLK2 periods. The RESET signal is level-sensitive. When the RESET pin is asserted, the 80386 ignores all input pins and drives all other pins to idle bus state. The 82384 clock generator provides system clock and reset signals.

D0-D31 provides the 32-bit data bus. The 80386 can transfer 16- or 32-bit data via the data bus.

The address pins A2-A31 along with the byte enable signals BE0# thru BE3# to generate physical memory or I/O port addresses. Using these pins, the 80386 can directly address 4

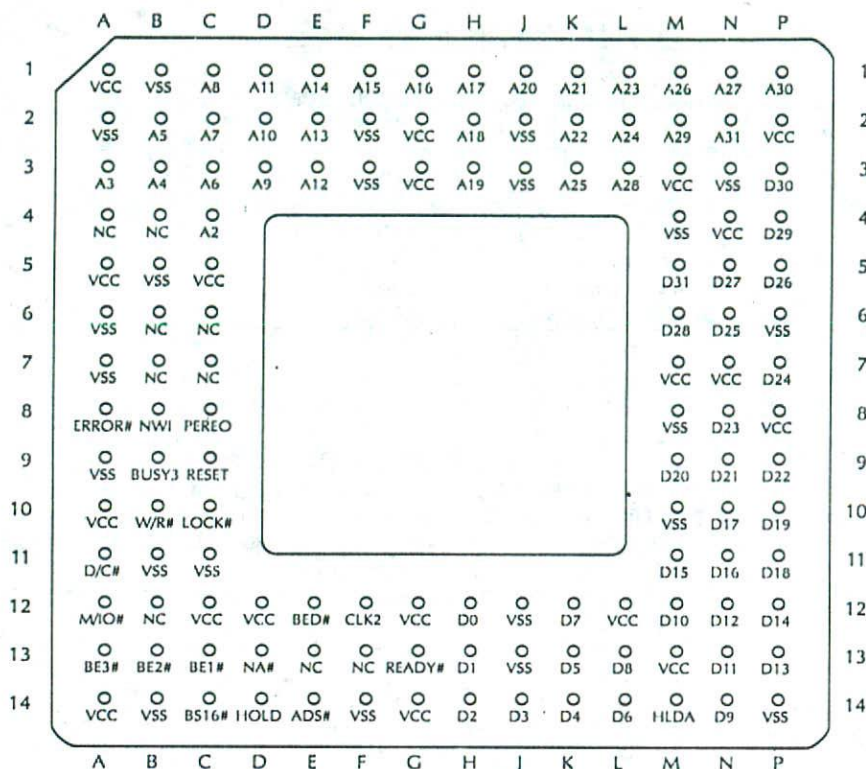


FIGURE 4.13 80386 PGA pinout view from pin side.

gigabytes by physical memory (00000000H thru FFFFFFFFH) and 64 kilobytes of I/O addresses (00000000H thru 0000FFFFH). The coprocessor addresses range from 800000F8H thru 800000FFH. Therefore, coprocessor select signal is generated by the 80386 when M/IO # is LOW and A31 is HIGH.

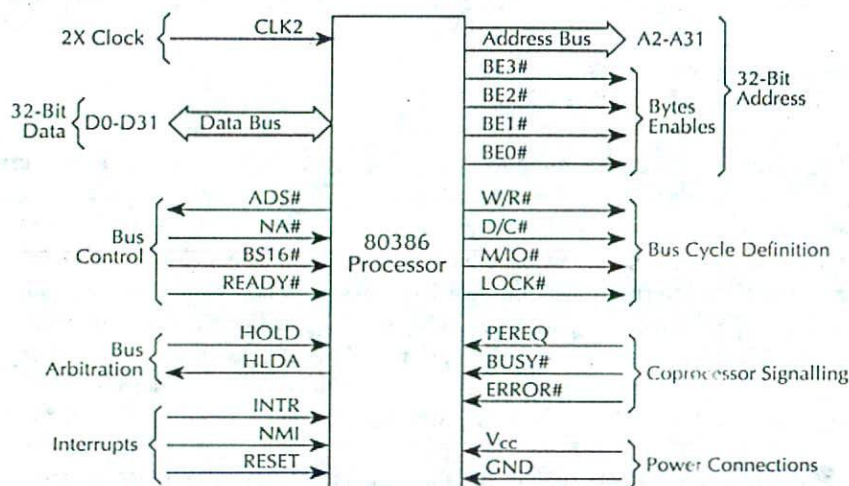


FIGURE 4.14 Functional signal groups.

The byte enable outputs, BE0# thru BE3# by the 80386, define which bytes of D0-D31 are utilized in the current data transfer. These definitions are given below:

BE0# is low when data is transferred via D0-D7
 BE1# is low when data is transferred via D8-D15
 BE2# is low when data is transferred via D16-D23
 BE3# is low when data is transferred via D24-D31

The 80386 asserts one or more byte enables depending on the physical size of the operand being transferred (1, 2, 3, or 4 bytes).

When the 80386 performs a word memory write or word I/O write cycle via D16-D31 pins, it duplicates this data on D0-D15.

W/R#, D/C#, M/IO#, and LOCK# output pins specify the type of bus cycle being performed by the 80386. W/R# pin, when HIGH, identifies write cycle and, when LOW, indicates read cycle. D/C# pin, when HIGH, identifies data cycle and, when LOW, indicates control cycle. M/IO# differentiates between memory and I/O cycles. LOCK# distinguishes between locked and unlocked bus cycles. W/R#, D/C#, and M/IO# pins define the primary bus cycle. This is because these signals are valid when ADS# (address status output) is asserted. LOCK# output is valid as soon as the bus cycle begins, but due to address pipelining LOCK# may be valid later when ADS# is asserted. Table 4.9 defines the bus cycle definitions.

The 80386 bus control signals include ADS# (address status), READY# (transfer acknowledge), NA# (next address request), and BS16# (bus size 16).

The 80386 outputs LOW on the ADS# pin to indicate a valid bus cycle (W/R#, D/C#, M/IO#) and address (BE0#-BE3#, A2-A31) signals.

When READY# input is asserted during a read cycle or an interrupt acknowledge cycle, the 80386 latches the input data on the data pins and ends the cycle. When READY# is low during a write cycle, the 80386 ends the bus cycle.

The NA# input pin is activated low by external hardware to request address pipelining. A low on this pin means that the system is ready to receive new values of BE0#-BE3#, A2-A31, W/R#, D/C#, and M/IO# from the 80386 even if the completion of the present cycle is not acknowledged on the READY# pin.

BS16# input pin permits the 80386 to interface to 32- and 16-bit data buses. When the BS16# input pin is asserted low by an external device, the 80386 uses the low-order half (D0-D15) of the data bus corresponding to BE0# and BE1# for data transfer. If the 80386 asserts BE2# or BE3# during a bus cycle, then assertion of BS16# by an external device in this cycle

TABLE 4.9 Bus Cycle Definition

M/IO#	D/C#	W/R#	Bus cycle type	Locked?
Low	Low	Low	INTERRUPT ACKNOWLEDGE	Yes
Low	Low	High	Does not occur	—
Low	High	Low	I/O DATA READ	No
Low	High	High	I/O DATA WRITE	No
High	Low	Low	MEMORY CODE READ	No
High	Low	High	HALT: SHUTDOWN:	No
			Address = 2 Address = 0 (BE0# High (BE0# Low BE1# High BE1# High BE2# Low BE2# High BE3# High BE3# High A2-A31 Low) A2-A31 Low)	
High	High	Low	MEMORY DATA READ	Some cycles
High	High	High	MEMORY DATA WRITE	Some cycles

will automatically cause the 80386 to transfer the upper byte(s) via only D0-D15. For 32-bit data operands with BS16# asserted, the 80386 will automatically execute two consecutive 16-bit bus cycles to accomplish this.

HOLD (input) and HLDA (output) pins are 80386 bus arbitration signals. These signals are used for DMA transfers. PEREQ, BUSY#, and ERROR# pins are used for interfacing coprocessors such as 80287 or 80387 to the 80386. A HIGH on PEREQ (coprocessor request) input pin indicates that a coprocessor is requesting the 80386 to transfer data to or from memory. The 80386 thus transfers data between the coprocessor and memory. This signal is level-sensitive. A LOW on the BUSY# (coprocessor Busy) input pin means that the coprocessor is still executing an instruction and is not capable of accepting another instruction. The BUSY# pin avoids interference with a previous coprocessor instruction.

ERROR# (coprocessor error) input pin, when asserted LOW by the coprocessor, indicates that the previous coprocessor instruction generated a coprocessor error of a type not masked by the coprocessor's control register. This input pin is automatically sampled by the 80386 when a coprocessor instruction is encountered and, if asserted, the 80386 generates exception 7 for executing the error-handling routine.

There are two interrupt pins on the 80386. These are INTR (maskable) and NMI (nonmaskable) pins. INTR is level-sensitive. When INTR is asserted and if the IF bit in the EFLAGS is 1, the 80386 (when ready) responds to the INTR by performing two interrupt acknowledge cycles and at the end of the second cycle latches an 8-bit vector on D0-D7 to identify the source of interrupt. To ensure INTR recognition, it must be asserted until the first interrupt acknowledge cycle starts.

NMI is leading-edge sensitive. It must be negated for at least 8 CLK2 periods and then be asserted for at least 8 CLK2 periods to assure recognition by the 80386. The servicing of NMI was discussed earlier.

Table 4.10 summarizes the characteristics of all 80386 signals.

TABLE 4.10 80386 Signal Summary

Signal name	Signal function	Active state	Input/output	Input synch or asynch to CLK2	Output high impedance during HDLA?
CLK2	Clock	—	I	—	—
D0-D31	Data bus	High	I/O	S	Yes
BE0#-BE3#	Byte enables	Low	O	—	Yes
A2-A31	Address bus	High	O	—	Yes
W/R#	Write-read indications	High	O	—	Yes
D/C#	Data-control indication	High	O	—	Yes
M/IO#	Memory-I/O indication	High	O	—	Yes
LOCK#	Bus lock indication	Low	O	—	Yes
ADS#	Address status	Low	O	—	Yes
NA#	Next address request	Low	I	S	—
BS16#	Bus size 16	Low	I	S	—
READY#	Transfer acknowledge	Low	I	S	—
HOLD	Bus hold request	High	I	S	—
HLDA	Bus hold acknowledge	High	O	—	No
PEREQ	Coprocessor request	High	I	A	—
BUSY#	Coprocessor busy	Low	I	A	—
ERROR#	Coprocessor error	Low	I	A	—
INTR	Maskable interrupt request	High	I	A	—
NMI	Nonmaskable interrupt request	High	I	A	—
RESET	Reset	High	I	A (note)	—

Note: If the phase of the internal processor clock must be synchronized to external circuitry, RESET falling edge must meet setup and hold times t_{25} and t_{26} .

4.2.10 80386 Bus Transfer Technique

The 80386 uses one or more bus cycles to perform all data transfers.

The 32-bit address is generated by the 80386 from BE0#-BE3# and A2-A32 as follows:

80386 address signals							
Physical base address			80386 address pins and BE0#-BE3# signals				
A31—A2	A1	A0	A31—A2	BE3#	BE2#	BE1#	BE0#
A31—A2	0	0	A31—A2	X	X	X	Low
A31—A2	0	1	A31—A2	X	X	Low	High
A31—A2	1	0	A31—A2	X	Low	High	High
A31—A2	1	1	A31—A2	Low	High	High	High

Dynamic bus sizing feature connects the 80386 with 32-bit or 16-bit data buses for memory or I/O. A single 80386 can be connected to both 16- and 32-bit buses. During each bus cycle, the 80386 dynamically determines bus width and then transfers data to or from 32- or 16-bit devices. During each bus cycle, the 80386 BS16# pin can be asserted for 16-bit ports or negate BS16# for 32-bit ports by the external device. With BS16# asserted all transfers are performed via D0-D15 pins. Also, with BS16# asserted, the 80386 automatically performs data transfers larger than 16 bits or misaligned 16 bits transfers in multiple cycles as needed. Note that 16-bit memory or I/O devices must be connected on D0-D15 pins.

Asserting BS16# only affects the 80386 when BE2# and/or BE3# are asserted during the cycle. Assertion of BS16# does not affect the 80386 if data transfer is only performed via D0-D15. On the other hand, the 80386 is affected by assertion of the BS16# pin, depending in which byte enable pins are asserted during the current bus cycle. For example, asserting BS16# during "upper half only" reads causes the 80386 to read data on the D0-D15 pins and ignores data on the D16-D31. Data that would have been read from D16-D31 (as indicated by BE2# and BE3#) will instead be read from D0-D15.

A 32-bit-wide memory can be interfaced to the 80386 by utilizing its BS16#, BE0#-BE3#, and A2-A31 pins. Each 32-bit memory word starts at a byte address that is a multiple of 4. BS16# is connected to HIGH (negated) for all bus cycles for 32-bit transfers. A2-A31 and BE0#-BE3# are used for addressing the memory.

For 16-bit memories, each 16-bit memory word starts at an address which is a multiple of 2. The address is decoded to assert BS16# only during bus cycles for 16-bit transfers.

A2-A31 can be used to address 16-bit memory also. A1 and two-byte enable signals are also required.

To obtain A1 and two-byte enables for 16-bit transfers, BE0#-BE3# should be decoded as in Table 4.11.

Figure 4.16 shows a block diagram interfacing 16- and 32-bit memories to 80386.

Finally, if an operand is not aligned such as a 32-bit doubleword operand beginning at an address not divisible by 4, then multiple bus cycles are required for data transfer.

4.2.11 80386 Read and Write Cycles

The 80386 performs data transfer during bus cycles (also called read or write cycles).

Two choices of address timing are dynamically selectable. These are nonpipelined and pipelined. One of these timing choices is selectable on a cycle-by-cycle basis with the Next Address (NA#) input.

After a bus idle state, the 80386 always uses nonpipelined address timing. However, the NA# may be asserted by an external device to select pipelined address timing, for the next cycle is made available before the present bus cycle is terminated by the 80386 by asserting READY#.

TABLE 4.11 Generating A1, BHE#, and BLE# for Addressing 16-Bit Devices

80386 signals				16-bit bus signals			Comments
BE3#	BE2#	BE1#	BE0#	A1	BHE#	BLE# (A0)	
H*	H*	H*	H*	x	x	x	x — no active bytes
H	H	H	L	L	H	L	
H	H	L	H	L	L	H	
H	H	L	L	L	L	L	
H	L	H	H	H	H	L	
H*	L*	H*	L*	x	x	x	x — not contiguous bytes
H	L	L	H	L	L	H	
H	L	L	L	L	L	L	
L	H	H	H	H	L	H	
L*	H*	H*	L*	x	x	x	x — not contiguous bytes
L*	H*	L*	H*	x	x	x	x — not contiguous bytes
L*	H*	L*	L*	x	x	x	x — not contiguous bytes
L	L	H	H	H	L	L	
L*	L*	H*	L*	x	x	x	x — not contiguous bytes
L	L	L	H	L	L	H	
L	L	L	L	L	L	L	

Note: BLE# asserted when D0-D7 of 16-bit bus is active; BHE# asserted when D8-D15 of 16-bit bus is active; A0 low for all even words; A0 high for all odd words.

Key: x = don't care

H = high voltage level

L = low voltage level

* = a nonoccurring pattern of Byte Enables; either none are asserted, or the pattern has Byte Enables asserted for noncontiguous bytes

In general, the 80386 samples NA# input during each bus cycle to select the desired address timing for the next bus cycle.

Physical data bus width (16- or 32-bit) is selected by the 80386 by sampling the BS16# (bus size 16) input pin near the end of the bus cycle. Assertion of BS16# indicates a 16-bit data bus, while negation of BS16# means a 32-bit data bus.

A read or write cycle is terminated by the 80386 on a low READY# (assertion) from the external device. Until the READY# is asserted, the 80386 inserts wait states to permit adjustment

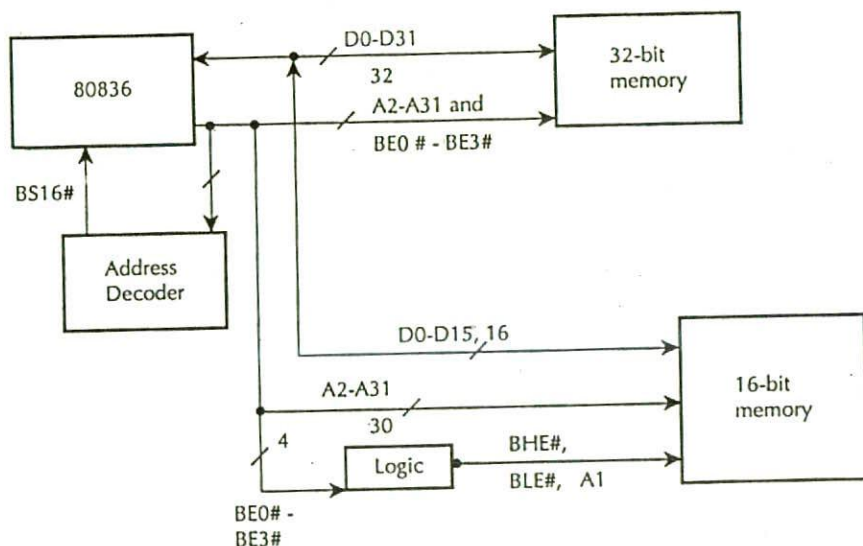


FIGURE 4.16 Interfacing 80386 16- and 32-bit memories.

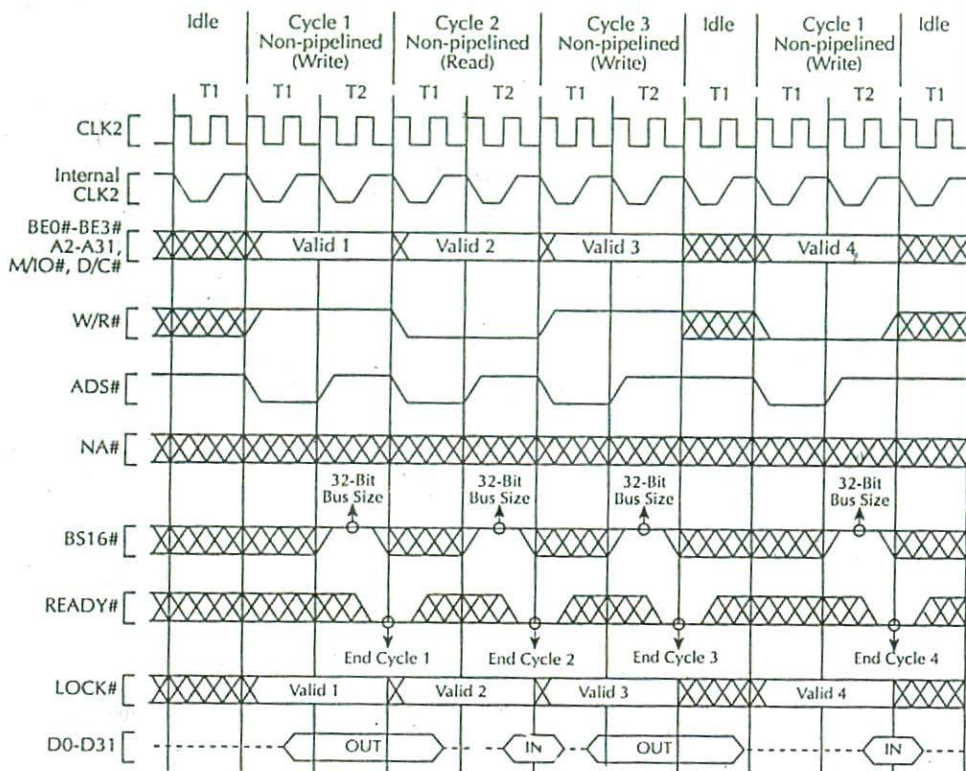


FIGURE 4.17 Bus cycles with nonpipelined address (zero wait states).

for the speed of any external device when a read cycle is terminated, and the 80386 latches the information present at its data pins. When a write cycle is acknowledged, the 80386 write data remain valid throughout phase one of the next bus state, to provide write data hold time.

To illustrate the concept of 80386 bus cycle timing, a mixture of read and write cycles with nonpipelined address timing is shown in Figure 4.17.

This diagram shows the fastest possible cycles with nonpipelined address timing having two bus states (T1 and T2) per bus cycle. In phase one T1, the address signals and bus control signals are valid and the 80386 activates ADS# low to indicate their availability.

During read cycle, the 80386 tristates its data signals to permit driving by the external device being addressed. During write cycle, the 80386 places data on the data bus starting in phase two of T1 until phase one of the bus state following cycle acknowledgment.

4.2.12 80386 Modes

The 80386 can be operated in real, protected, or virtual 8086 mode. These modes are described below.

4.2.12.a 80386 Real Mode

Upon reset or power-up, the 80386 operates in real mode. In real mode, the 80386 can access all the 8086 registers along with the 80386 32-bit registers. The memory addressing, memory size, and interrupts of 80386 in this mode are the same as those of the 80286 in real mode.

The 80386 can execute all the instructions in real mode. The main purpose of real mode is to initialize the 80386 for protected mode operation.

In real mode, the 80386 can directly address up to one megabyte of memory. The address lines A2-A19, BE0#-BE3# are used by the 80386 in this mode. Paging is not provided in real

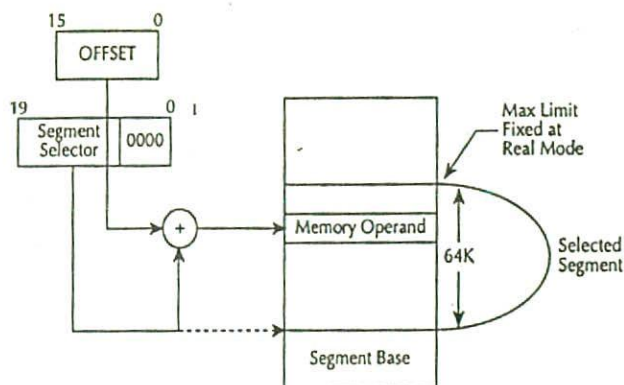


FIGURE 4.18 Real address mode addressing.

mode. Therefore, linear addresses are identical to physical addresses. The 20-bit physical address is formed by adding the shifted (four times to the left) segment registers to an offset as shown in Figure 4.18.

All segments in real mode are exactly 64K bytes wide. Segments can be overlapped in this mode. There are two memory areas which are reserved in real mode for system initialization and interrupt pointer table. Addresses 00000H thru 002FFH are reserved for the interrupt pointer table, while addresses FFFFFFF0H thru FFFFFFFFH are reserved for system initialization. Many of the exceptions listed in Table 4.6 are not applicable to real mode. Exceptions 10, 11, 12, and 14 will never occur in this mode. Also, other exceptions have minor variations as follows:

Function	Interrupt number	Related instructions	Return address location
Interrupt table Limit too small	8	INT vector is not within table limit	Before instruction
Segment overrun exception	13	Word memory reference with offset = FFFFH or an attempt to execute an instruction past the end of a segment	Before instruction

4.2.12.b Protected Mode

The total 80386 capabilities are available when the 80386 operates in protected mode. This mode increases the linear space to four gigabytes (2^{32} bytes) and permits the execution of virtual memory programs of 64 tetra-bytes (2^{46} bytes). Also, in protected mode, the 80386 can run all existing 8086 and 80286 programs with on-chip memory management and protection features. The protected mode includes new instructions to support multitasking operating systems. The main difference between protected mode and real mode from a programmer's viewpoint is the increased memory space and a differing addressing mechanism. Similar to real mode, protected mode also includes two elements (16-bit selector for determining a segment's base address and a 32-bit offset or effective address) to obtain a 32-bit linear address. This 32-bit linear address is either used as the 32-bit physical address or, if paging is enabled, the paging mechanism translates this 32-bit linear address to a 32-bit physical address. Figure 4.19 shows the protected mode addressing mechanism. The selector is used to specify an index into a table defined by the operating system. The table includes the 32-bit base address of a given segment. The physical address is obtained by summing the base address obtained from the table with the offset.

With the paging mechanism enabled, the 80386 provides an additional memory management mechanism. The paging feature manages large 80386 segments.

The paging mechanism translates the protected linear addresses from the segmentation unit into physical addresses. Figure 4.20 shows this translation scheme.

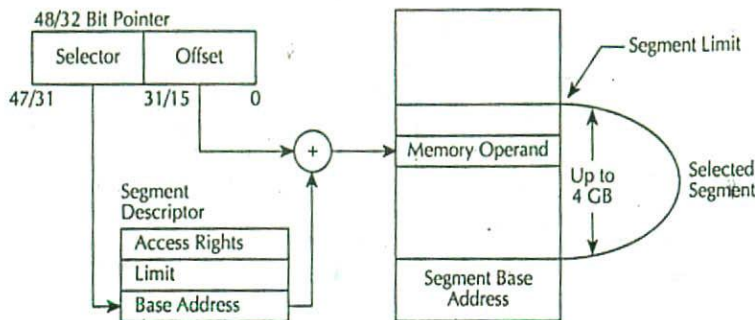


FIGURE 4.19 Protected mode addressing.

Let us now discuss 80386 segmentation, protection, and paging features.

Segmentation provides both memory management and protection. All information about the segments is stored in an 8-byte data structure called a descriptor. All the descriptors are stored in tables identified by the 80386 hardware. There are three types of tables holding 80386 descriptors: global descriptor table (GDT), local descriptor table (LDT), and interrupt descriptor table (IDT). These tables are memory arrays of variable lengths. Their sizes can vary from 8 bytes to 64K bytes. Each table can store up to 8192 8-byte descriptors. The upper 13 bits of a selector are used as an index into the descriptor table. The tables have associated registers which store a 32-bit linear base address and a 16-bit limit for each table. Each table has a set of registers, namely, GDTR (32-bit), LDTR (16-bit), and IDTR (32-bit), associated with it. The 80386 instructions LGDT, LLDT, and LIDT are used to load the base and 16-bit limit of the global, local, and interrupt descriptor tables into the appropriate registers. The SGDT, SLDT, and SIDT instructions store the base and limit values.

The GDT contains descriptors which are available to all the tasks in the system. In general, the GDT contains code and data segments used by the operating system, task state segments, and descriptors for LDTs in a system.

LDTs store descriptors for a given task. Each task has a separate LDT, while the GDT contains descriptors for segments which are common to all tasks.

The IDT contains the descriptors which point to the location of up to 256 interrupt service routines. Every interrupt used by a system must have an entry into the IDT. The IDT entries are referenced via INT instructions, external interrupt vectors, and exceptions.

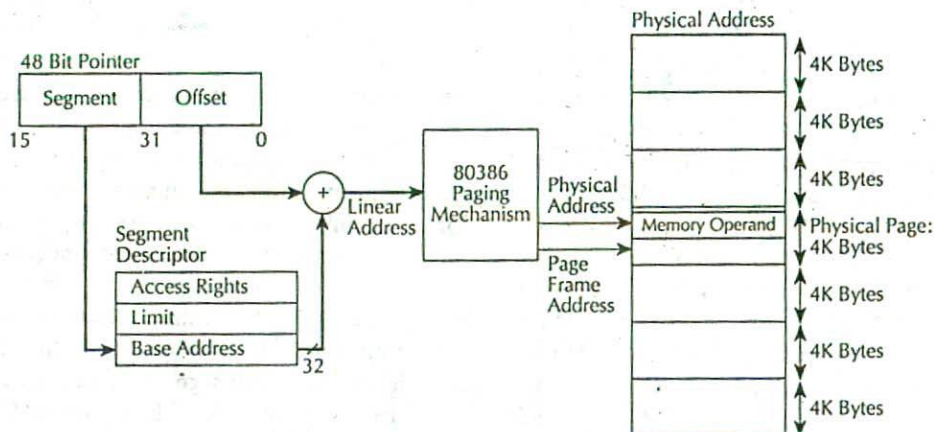


FIGURE 4.20 Paging and segmentation.

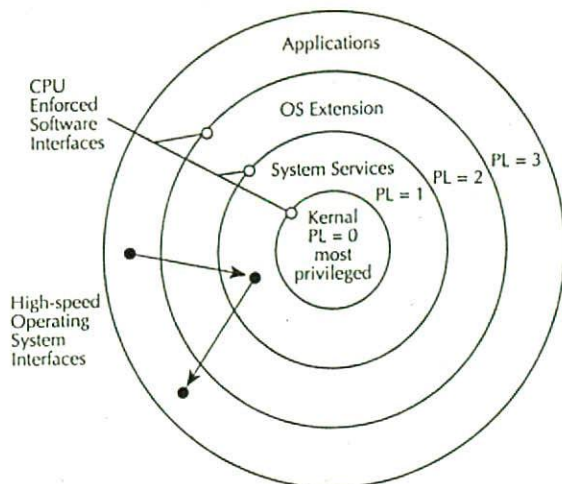


FIGURE 4.21 Four-level hierarchical protection.

The object to which the selector points is called a descriptor. Descriptors are eight bytes wide containing attributes about a given segment. These attributes contain the 32-bit base linear address of the segment, segment length, protection level, read/write/execute privileges, the default operand size (16 or 32 bits), and segment type.

In order to provide operating system compatibility between the 80286 and 80386, the 80386 supports all of the 80286 segment descriptors. The only differences between the 80286 and 80386 formats are that the values of the type fields and the limit and base address fields have been expanded for the 80386.

The 80286 system segment descriptors contain a 24-bit base address and 16-bit limit, while the 80386 system segment descriptors have a 32-bit base address, a 20-bit limit field, and a granularity bit. Note that the segment length is page granular if the granularity bit is one; otherwise, the segment length is byte granular.

By supporting 80286 segments the 80386 is able to execute 80286 application programs on an 80386 operating system. This is possible because the 80386 automatically can differentiate between the 80286-type and 80386-type descriptors. In particular, if the upper word of a descriptor is zero, then that descriptor is an 80286-type descriptor.

The only other differences between the 80286 and 80386 descriptors are the interpretation of the word count field of call gates and the B bit. The word count field specifies the number of 16-bit quantities to copy for 80286 call gates and 32-bit quantities for 80386 call gates. The B bit controls the size of pushes when using a call gate. If $B = 0$, then pushes are 16 bits, while pushes are 32 bits for $B = 1$.

The 80386 provides four protection levels for supporting a multitasking operating system to isolate and protect user programs from each other and the operating system. The privilege level controls the use of privileged instructions, I/O instructions, and access to segments and segment descriptors. The 80386 includes the protection as part of its memory management unit. The 80386 also provides an additional type of protection when paging is enabled.

The four-level hierarchical privilege system is shown in Figure 4.21. It is an extension of the user/supervisor privilege mode used by minicomputers. Note that the user/supervisor mode is supported by the 80386 paging mechanism. The Privilege Levels (PL) are numbered 0 thru 3. Level 0 is the most privileged level.

The 80386 provides the following rules of privilege to control access to both data and procedures between levels of a task:

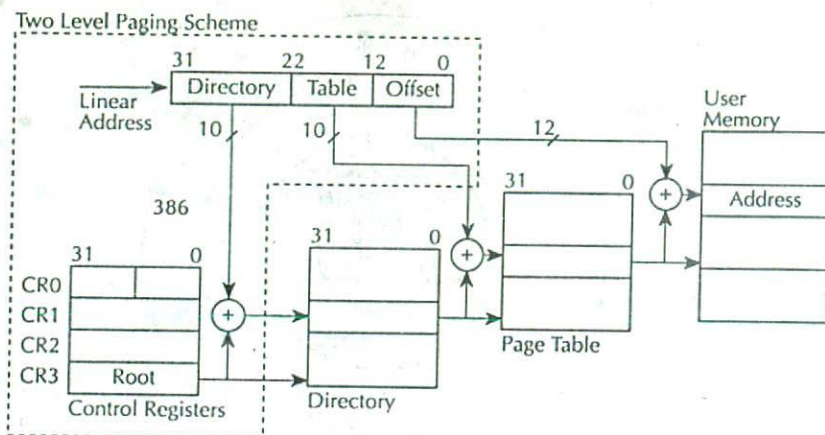


FIGURE 4.22 Paging mechanism.

- Data stored in a segment with a privilege level x can be accessed only by code executing at a privilege level at least as privileged as x .
- A code segment/procedure with privilege level x can only be called by a task executing at the same or a higher privilege level than x .

The 80386 supports task gates (protected indirect calls) to provide a secure method of privilege transfers within a task.

The 80386 also supports a rapid task switch operation via hardware. It saves the entire state of the machine (all of the registers, address space, and a link to the previous task), loads a new execution state, performs protection checks, and commences execution in the new task in approximately 17 microseconds.

Paging is another type of memory management for virtual memory multitasking operating systems. The main difference between paging and segmentation is that paging divides programs/data into several equal-sized pages, while segmentation divides programs/data into several variable-sized segments.

There are three elements associated with the 80386 paging mechanism. These are page directory, page tables, and the page itself (page frame). The paging mechanism does not have memory fragmentation since all pages have the same size of 4K bytes. Figure 4.22 shows the 80386 paging mechanism.

There are four 32-bit control registers (CR0-CR3) associated with the paging mechanism. CR2 is the page fault linear address register and contains the 32-bit linear address which caused the last page fault detected.

CR3 is the page directory physical base address register and contains the physical starting address of the page directory. The lower 12 bits of CR3 are always zero to ensure that the page directory is always page aligned. CR1 is reserved for future Intel processors. CR0 contains 6 defined bits for control and status purposes. The low-order 16 bits of CR0 are known as the machine status word and include special control bits such as the enable bit and the protection enable bit.

The page directory is 4K bytes long and permits up to 1024 page directory entries. Each page directory entry contains the address of the next level of tables, page tables, and information about the page table. The upper 10 bits of the linear address (A22-A31) are used as an index to select the correct page directory entry.

Each page table is 4K bytes long and holds up to 1024 page table entries. Page table entries contain the starting address of the page frame and statistical information about the page such as whether the page can be read or written in supervisor or user mode. Address bits A12-A21

are used as index to select one of the 1024 page table entries. The 20 upper-bit page frame address is concatenated with the lower 12 bits of the linear address to form the physical address. Page tables can be shared between tasks and swapped to disks.

The lower 12 bits of the page table entries and page directory entries contain statistical information about pages and page tables, respectively. As an example, the P (present) bit indicates whether a page directory or page table entry can be used in address translation. If $P = 1$, the entry can be used in address translation, and if $P = 0$, the entry cannot be used for translation and all other 31 bits are available for use by the software. These 31 bits can be used to indicate where on a disk the page is located.

The 80386 provides a set of protection attributes for paging systems. The paging mechanism provides two levels of protection: user and supervisor. The user level corresponds to level 3 of the segmentation-based protection and the supervisor level combines all of the other protection levels (0, 1, 2). Programs executing at level 0, 1, or 2 bypass the page protection, although segmentation-based protection is still enforced by hardware.

The 80386 takes care of the page address translation process, relieving the burden from an operating system in a demand-paged system. The operating system is responsible for setting up the initial page tables and the handling of any page faults. The operating system initializes the tables by loading CR3 with the address of the page directory and allocates space for the page directory and the page tables. The operating system also implements a swapping policy and handles all of the page faults.

4.2.12.c Virtual 8086 Mode

The virtual 8086 mode permits the execution of 8086 applications while taking full advantage of the 80386 protection mechanism. In particular, the 80386 permits concurrent execution of 8086 operating systems and applications, an 80386 operating system, and both 80286 and 80386 applications. For example, in a multiuser 80386-based microcomputer, one person can run an MD-DOS spreadsheet, another person can use MS-DOS, and a third person can run multiple UNIX utilities and applications.

One of the main differences between 80386 real and protected modes is how the segment selectors are interpreted. In virtual 8086 mode, the segment registers are used in the same way as the real mode. The contents of the segment register are shifted 4 times to the left and added to the offset to obtain the linear address.

The paging hardware permits the simultaneous execution of several virtual mode tasks and provides protection.

The paging hardware allows the 20-bit linear address produced by a virtual mode program to be divided up into 256 pages. Each one of the pages can be located anywhere within the maximum 4-gigabyte physical address space of the 80386.

The paging hardware also permits sharing of the 8086 operating system code by several 8086 applications. All virtual mode programs execute at privilege level 3. Therefore, virtual mode programs are subject to all of the protection checks defined in protected mode. This is different from real mode which executes programs in level 0.

4.3 80386 System Design

In this section, the 80386 is interfaced to typical memory and I/O chips. As mentioned in the last section the 80386 address and data lines are not multiplexed. There is a total of thirty address pins (A2-A31) on one chip. A0 and A1 are decoded internally to generate four byte enable outputs, BE0#, BE1#, BE2# and BE3#. In real mode, the 80386 utilizes 20-bit addresses and A2 through A19 address pins are active and the address pins A20 through A31 are used in real mode at reset, high for CS-based accesses, low for others, and always low after CS changes. In the protected mode, on the other hand, all address pins A2 through A31 are active.

In both modes, A0 and A1 are decoded internally. In all modes, the 80386 outputs on the byte enable pins to activate appropriate portions of the data bus to transfer byte (8-bit), word (16-bit), and double-word (32-bit) data as follows:

Byte Enable Pins	Data Bus
BE0#	D0-D7
BE1#	D8-D15
BE2#	D16-D23
BE3#	D24-D31

The 80386 supports dynamic bus sizing. This feature connects the 80386 with 32-bit or 16-bit data buses for memory or I/O. The 80386 32-bit data bus can be dynamically switched to a 16-bit bus by activating the BS16# input from high to low by a memory or I/O device. In this case, all data transfers are performed via D0-D15 pins. 32-bit transfers take place as two consecutive 16-bit transfers over data pins D0 through D15. On the other hand, the 32-bit memory or I/O device can activate the BS16# pin HIGH to transfer data over D0-D31 pins.

For reading a byte, the 80386 makes one of BE0#-BE3# active. For a word read (aligned:even address), the 80386 makes two byte enable outputs (BE0#-BE3#) active. On the other hand, for a 32-bit aligned read, the 80386 activates all byte enable outputs BE0#-BE3#.

The 80386 duplicates data on some 16-bit write operations in order to enhance performance of the data bus. This is illustrated in the table below:

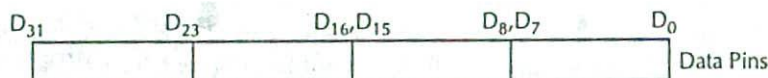
Transfer Size							Data Pins			
	A1	A0	BE3	BE2	BE1	BE0	D ₃₁ -D ₁₆	D ₂₃ -D ₁₆	D ₁₅ -D ₈	D ₇ -D ₀
Byte	0	0	1	1	1	0				X
Byte	0	1	1	1	0	1			X	
Byte	1	0	1	0	1	1		X		DD
Byte	1	1	0	1	1	1	X		DD	
Word	0	0	1	1	0	0			X	X
Word	0	1	1	0	0	1		X	X	
Word	1	0	0	0	1	1	X	X	DD	DD
Dword*	0	0	1	0	0	0		X	X	X
Dword*	0	1	0	0	0	1	X	X	X	
Dword	0	0	0	0	0	0	X	X	X	X

Note: DD = Data Duplication. Assumes 32-bit bus.

*For 32-bit misaligned transfers for addresses 3, 7, ... or addresses 1, 5, ..., these three-byte transfers along with one-byte from above are used to complete the 32-bit transfers in two cycles.

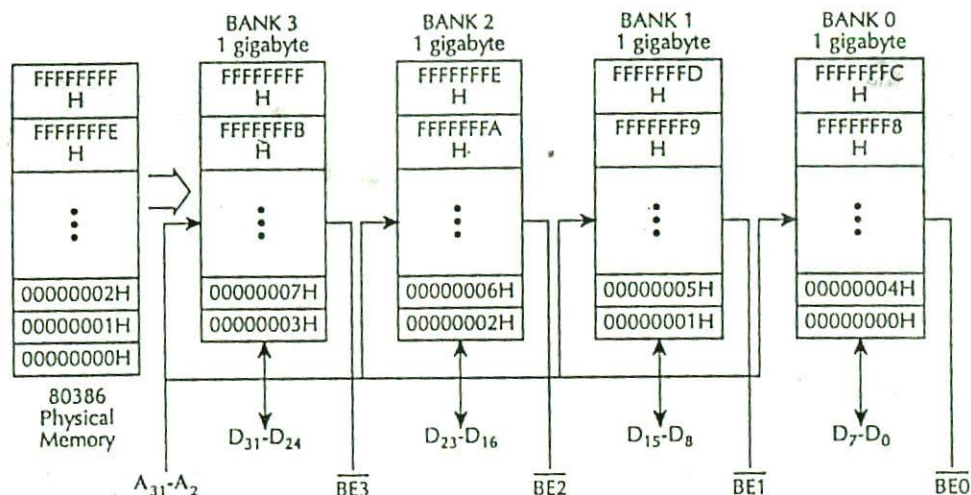
In the above table, the 80386 duplicates data for three cases. For example, in case of write cycle for byte when data is written on D₂₃-D₁₆ pins, the same data is duplicated on D₇-D₀ pins. On the other hand, writing a byte on D₃₁-D₂₄ pins, the same data is written on D₈-D₁₅ pins. Finally when 16-bit data is written on D₃₁-D₁₆ pins, the same data is duplicated by the 80386 on D₁₅-D₀ pins.

The 80386 address pins A1 and A0 specify the four addresses of a four byte (32-bit) word. Consider the following:



The contents of the memory addresses which include 0, 4, 8, ... with A1A0 = 00₂ are transferred by D₀-D₇. Similarly, the contents of addresses which include 1, 5, 9, ..., with A1A0 = 01₂ are transferred over D₈-D₁₅. On the other hand, the contents of memory addresses 2, 6, 10, ... with A1A0 = 10₂ are transferred over D₁₆-D₂₃ while contents of addresses 3, 7, 11, ... with

A1A0 = 11₂ are transferred over D₂₄-D₃₁. Note that A1A0 is encoded from BE3 - BE0 pins. The following figure depicts this:



In each bank, a byte can be accessed by enabling one of the byte enables, BE0 - BE3. For example, in response to execution of a byte-MOVE instruction such as MOV (00000006H), BL, the 80386 outputs low on BE2 and high on BE0, BE1 and BE3 and the content of BL is written to address 00000006H. On the other hand, when the 80386 executes a MOVE instruction such as MOV (00000004H), AX, the 80386 drives BE0 and BE1 to low. The contents of 00000004H and 00000005H are transferred from AL and AH via D₀-D₇ and D₈-D₁₅ respectively. For 32-bit transfer, the 80386 executing a MOVE instruction from an aligned address such as MOV (00000004H), EAX, the 80386 drives all bus enable pins (BE0 - BE3) to low and writes the contents of four bytes (00000004H through 00000007H) from EAX. Byte (8-bit), aligned word (16-bit), and aligned double-word (32-bit) are transferred by the 80386 in a single bus cycle.

The 80386 performs misaligned transfers in two bus cycles. For example, the 80386 executing a misaligned word MOVE instruction such as MOV (00000003H), AX drives BE3 to low in the first bus cycle and writes the contents of 00000003H from bank 3 into AL in the first bus cycle. The 80386 then drives BE0 to low in the second bus cycle and writes the contents of 00000004H from bank 0 into AH in the second bus cycle. This transfer takes two bus cycles.

A 32-bit misaligned transfer such as MOV (00000002H), EAX, on the other hand, takes two bus cycles. In the first bus cycle, the 80386 enables BE2 and BE3, and writes the contents of address 00000002H and 00000003H from banks 2 and 3 respectively into low 16-bits of EAX. In the second cycle, the 80386 enables BE0 and BE1 to low and then writes the contents of address 00000004H and 00000005H into the upper 16 bits of EAX.

4.3.1 80386 Memory Interface

Figure 4.23 shows the basic memory interface block diagram.

The bus control logic provides the control signals for READY#, NA#, address latches, data buffers, and memory chips. The bus control logic activates READY# to LOW to signal the completion of the 80386 bus cycle and also outputs low on NA# (Next Address Request) to

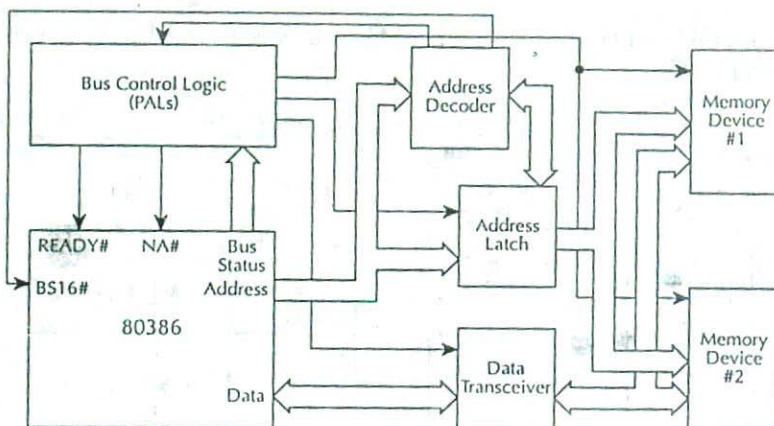


FIGURE 4.23 Basic memory interface block diagram.

activate address pipelining. The address decoder generates chip select signals and BS16# signal based on the address outputs of the 80386. These conventional memory interfacing concepts can be used to interface the 80386 to ROMs, EPROMs, and static RAMs.

The bus control logic decodes the 80386 status outputs (W/R#, M/IO#, and D/C#) and sends a command signal for the type of bus cycle according to Table 4.9 as per the following:

1. Memory read command (MRDC#) signal is generated from memory data read (M/IO#=1, D/C#=1, W/R#=0) or memory code read (M/IO#=1, D/C#=0, W/R#=0) cycle. MRDC# commands the selected memory device to output data.
2. I/O read cycle (M/IO#=0, D/C#=1, W/R#=0) generates the I/O read command (IORC#) output. IORC# commands the selected I/O device to output data.
3. Memory write command (MWTC#) is obtained from memory write cycle (M/IO#=1, D/C#=1, W/R#=1). MWTC# tells the selected memory device to receive data on the data bus.
4. I/O write command (IOWC#) is obtained from the IO write cycle (M/IO#=0, D/C#=1, W/R#=1). IOWC# tells the selected I/O device to receive data on the data bus.
5. INTA# is generated from Interrupt Acknowledge cycle (M/IO#=0, D/C#=0, W/R#=0). INTA# is sent back to the Interrupt controller such as the 80259A. A second INTA# cycle tells a device such as the 8257A to place the interrupt vector on the bus. PALs can be used to design bus control logic.

Address latches maintain the 80386 address for the duration of the bus cycle and are required to pipeline address since the address for the next bus cycle appears on the address lines before the end of the current bus cycle. Latches such as 74 × 373 can be used. Note that the 80386 can be run without address pipelining to eliminate the need for address latching but the system will run inefficiently.

Standard 8-bit transceivers (74 × 245) provide isolation and additional drive currents for the 80386 data bus. Transceivers are necessary to prevent the contention on the data bus that occurs if some devices are slow to remove data from the data bus after a read cycle. Transceiv-

ers can be omitted if the data float time of the device is short enough and the load on the 80386 data pins meets device specifications. Figure 4.24 shows memory interface of a typical 80386-based microcomputer.

Three 74AS373 octal latches are used to maintain the address for the duration of the bus cycle. The 74AS373 latch Enable (LE) input is controlled by the 80386 ALE output. The 74AS373 output Enable (OE#) is always active.

Two 74F139 decoders (2 to 4) are used. The top 74F139 decoder has A31 and M/IO as inputs. When the 80386 executes a memory-oriented instruction, it outputs HIGH on the M/IO pin and if A31 is one, then the Y2 output of the decoder goes to LOW. The Y2 output of the decoder is connected to the chip select 1 wait state (CS1WS) of the PAL 16R8B. The CS1WS is also latched by the 74AS373 and is used as the chip select for the 27128 EPROMs. Note that the 80386 reset vector is included in high memory (FFFFFF0H). Therefore, the memory maps for EPROMs must include these high memory addresses. Also, when M/IO=1 and A31=0, the upper decoder generates chip select 0 wait state (CS0WS). This signal is latched and used as chip select for 651628, 2KX8 static RAMs.

The address pins, byte enable signals (BE3-BE0), chip select outputs of the decoder and W/R are latched by the 74AS373 at the falling edge of ALE. Note that 80386 address lines A1 through A14 are used to address each EPROM.

The PAL 1 and PAL2 devices are used to design the bus control logic. PAL1 follows the 80386 bus cycles and generates the overall bus cycle timing. PAL2 generates most of the control signals required for memory interface. The PAL equations are given in Intel 80386 hardware reference manual. These equations can be assembled by a PAL assembler program. The assembled code to the PAL is then applied by using a standard PROM programmer with a special PAL enhancement. The write control logic for the SRAMs are implemented by using 74F32 quad OR gates. The memory write command (MWTC) is applied to one input of each one of the OR gates. The other input to each one of the OR gates is the appropriate byte enable (BE3-BE0) signal. For example, when both MWTC and BE3 are low, then WE input of the left most static RAM is enabled and data can then be written to that RAM.

Four 74AS245 transceivers are used to provide buffering of the data lines. Then DEN output generated by PAL2 is used to enable the transceivers and the latched 80386 W/R is used to control the direction of data transfer.

Two 27128's are used to provide 16K × 16 of EPROM. These chips are connected to the 80386 D0-D15 pins. The 27128 requires 14-bit address, A0-A13. MRDC (Memory Read Command) output generated by PAL2 is used to enable the 27128 OE Chip select signal (CS1WS) from the input of PAL1 is latched and then converted to CS inputs of the 27128's. CS1WS is also connected to the 80386 BS16 input to indicate to the 80386 that all read cycles are 16-bit.

The 27128 memory map can be determined as follows:

Memory Map for 27128-1 ODD

A31	A30	A15	A14	A1	A0
1							1
└──────────────────┘				└──────────────────┘			
don't cares				all zeros			
assume 1's				to ones			
= FFFF8001H, FFFF8003H, , FFFFFFFFH.							

Memory Map for 27128-1 EVEN

FFFF8000H, FFFF8002H, , FFFFFFFEH

Four $2K \times 8$ static RAM chips are used to provide a total of 8K bytes of RAM storage. Each RAM has 11-bit address input connected to the 80386 A12-A2 pins. When data is read from SRAMs, the MRDC signals activate all four RAMs and the 32-bit data from the selected location is placed on the 80386 D₃₁-D₀ pins. The 80386 then reads a byte, word, or double word from the bus.

During write to SRAMs, the 80386 may not output a 32-bit word. The 80386 in this case outputs the appropriate data size (byte, word or double word). It also sends appropriate BE0-BE3 signals to indicate which portion of the data bus carries the data. MWTC and WE signals ensure that data are written into the appropriate SRAM. Note that SRAMs are enabled by the CS0WS signal. This means that data read and write bus cycle do not require any wait states.

Let us determine the memory map for RAMs:

BANK 0	A31 A30 . . . A13	A12 . . . A2 A1 A0
	0 <u> </u>	<u> </u> 0 0
	don't cares	all zeros to ones
	assume 1's	
	= 7FFFE000H, 7FFFE004H,	
BANK 1	7FFFE001H, 7FFFE005H,	
BANK 2	7FFFE002H, 7FFFE006H,	
BANK 3	7FFFE003H, 7FFFE007H,	

When an 80386-based microcomputer system uses a large main memory of several megabytes, it is usually designed with high capacity, slow-speed dynamic RAMs and EPROMs. Although access times of DRAMs and EPROMs can be as fast as 60ns and 120ns respectively, the 80386 microcomputer system with these chips is expensive and too slow for running with zero wait states. The details of the 80386's interfacing to DRAMs can be found in Intel manuals and is beyond the scope of this chapter.

An 80386 microprocessor at a speed of 25MHz would require DRAMs at 40ns access time for zero wait-state. This is why, for large memory, wait states are introduced in all bus cycles while accessing memory. These degrade the overall performance of the 80386-based microcomputer.

A cache memory subsystem can be implemented in the 80386-based microcomputer to improve overall system performance while utilizing inexpensive, slow-speed DRAMs in main memory.

A cache memory subsystem includes a small amount of fast static RAM and a large amount of DRAM. The cache memory subsystem contains a fast SRAM between the 80386 and the slower main memory (DRAMs) along with a cache controller. The cache controller such as Intel 82385 includes the logic to implement the cache memory. Cache sizes are either 32K bytes or 64K bytes.

One of the two most widely used cache organizations, namely the direct-mapped cache or two-way set associative cache, is utilized in an 80386 system. Details of the 80386 cache implementation are provided in the 80386 hardware reference manual.

4.3.2 80386 I/O

The 80386 can use either a standard I/O or a memory-mapped I/O technique.

The address decoding required to generate chip selects for devices using standard I/O is often simpler than that required for memory-mapped devices. But, memory-mapped I/O offers more flexibility in protection than standard I/O does.

The 80386 can operate with 8-, 16-, and 32-bit peripherals.

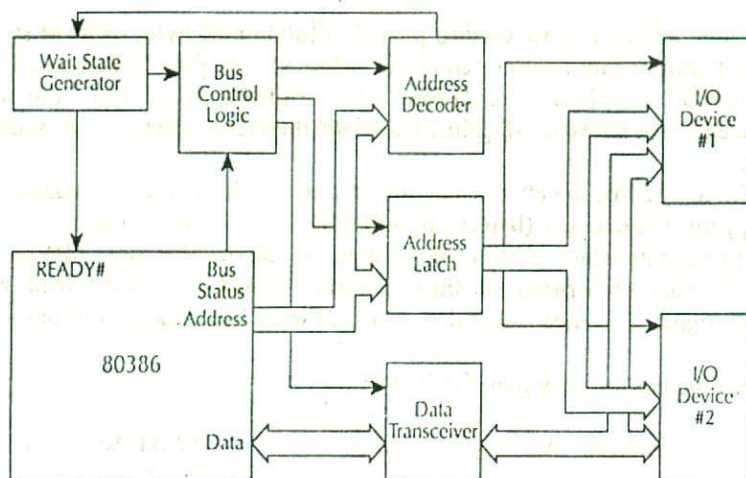


FIGURE 4.25 Basic I/O interface block diagram.

Eight-bit I/O devices can be connected to any of the four 8-bit sections of the data bus. Figure 4.25 shows 80386 I/O interface block diagram.

In the figure, the 80386's 32-bit data bus is multiplexed onto an 8-bit bus.

For efficient operation, 32-bit I/O devices should be assigned to addresses that are even multiples of four. If I/O devices are located on adjacent word boundaries, address decoding must generate the BS16# signal so that the 80386 performs a 16-bit bus cycle.

The block diagram is very similar to the 80386 memory interface block diagram. The purpose of various blocks in the figure has already been explained earlier in this section.

For standard I/O, the 80386 includes three types of I/O instructions. These are direct, indirect, and string I/O instructions which include the following:

Direct

For 8-bit: IN AL, PORT
OUT PORT, AL
For 16-bit: IN AX, PORT
OUT PORT, AX
For 32-bit: IN EAX, PORT
OUT PORT, EAX

Indirect

For 8-bit: IN AL, DX
OUT DX, AL
For 16-bit: IN AX, DX
OUT DX, AX
For 32-bit: IN EAX, DX
OUT DX, EAX

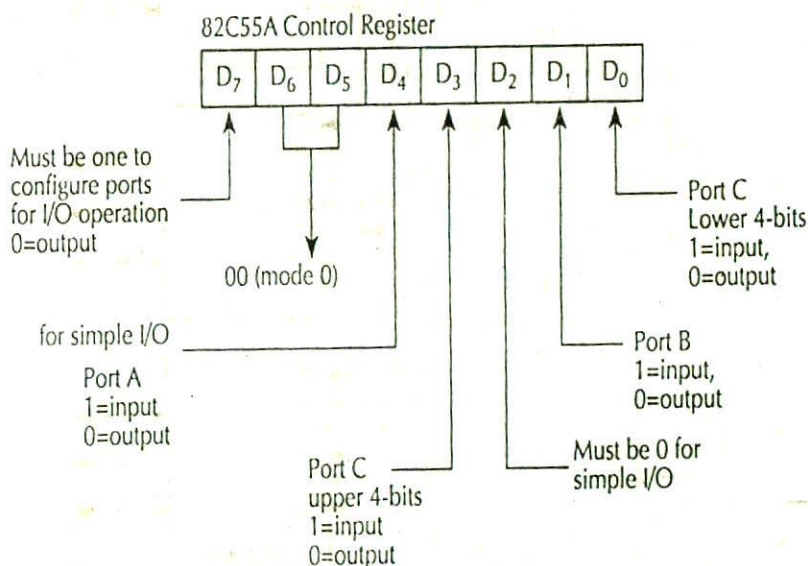
String

For 8-bit: INSB, (ES:DI) ← ((DX))
DI ← DI ± 1
OUTSB ((DX)) ← (ES:SI)
SI ← SI ± 1
For 16-bit: INSW, (ES:DI) ← ((DX))
(DI) ← DI ± 2
OUTSW (ES:SI) ← ((DX))

$(SI) \leftarrow SI \pm 2$
 For 32-bit: $INSD, (ES:EDI) \leftarrow (DX)$
 $EDI \leftarrow EDI \pm 4$
 $OUTSD, (DX) \leftarrow (ES:ESI)$
 $ESI \leftarrow ESI \pm 4$

The 82C55A Programmable Peripheral Interface (PPI) can be interfaced with the 80386 for obtaining parallel ports for either standard or memory-mapped I/O. The pin diagram and the features provided by the 82C55A are same as the 8255 which were described in Chapter 3. The 82C55A will be interfaced to the 80386 for simple I/O operation.

In summary, the 82C55A contains three 8-bit parallel ports namely Port A, Port B, and Port C. These ports can be configured as input or output ports by writing 1 or 0 respectively in the corresponding bits in the control register. The control register bits can be defined as follows:



For example, outputting 89H to the control register will configure Ports A and B as output ports and Port C as an input port.

The 82C55A ports and control register are selected by the two-bit register select inputs (A0 and A1 inputs of the 82C55A) as follows:

A1	A0	
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Control Register

Figure 4.26 shows 82C55A-80386 interface using standard I/O. Now let us determine the I/O port addresses in the 82C55A-4.

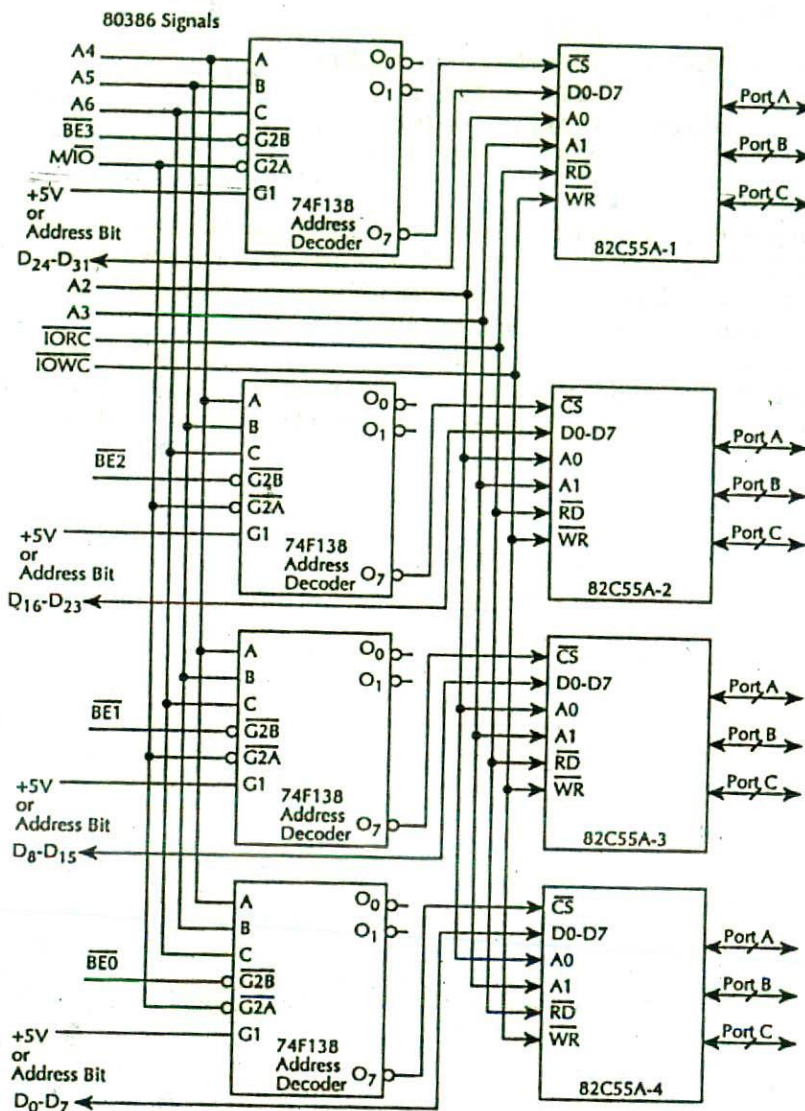


FIGURE 4.26 80386-82C55A interface using standard I/O.

A7	A6	A5	A4	A3	A2	A1	A0
X	1	1	1	0	0	0	0
↑							
don't care assume 1				Port A =F0H			

Note that, since BE0 is used to select the 82C55A-4, 80386 A₁ and A₀ pins will be zeros. Therefore in the 82C55A-4 the port addresses can be obtained as follows:

8-bit Address	Port Name
F0H	Port A
F4H	Port B
F8H	Port C
FCH	Control Register

The seven other unused outputs of each address decoder can be used to enable three 82C55As in each category. Similarly, the port addresses for the other 82C55A's can be obtained as follows:

	8-bit Address	Port Name
82C55A-3	F1H	Port A
	F5H	Port B
	F9H	Port C
	FDH	Control Register
82C55A-2	F2H	Port A
	F6H	Port B
	FAH	Port C
	FEH	Control Register
82C55A-1	F3H	Port A
	F7H	Port B
	FBH	Port C
	FFH	Control Register

Note that $\overline{\text{IORC}}$ and $\overline{\text{IOWC}}$ are the outputs of PAL2 of the 80386 memory interface of Figure 4.24.

Using memory-mapped I/O, a read-mode 80386 can be interfaced to 82C55's in Figure 4.26 by connecting $\overline{\text{MRDC}}$ and $\overline{\text{MWTC}}$ outputs of PAL2 of Figure 4.24 to RD and WR of the 82C55A respectively. The G2A input line of the 74F138 can be connected through an inverter to unused 80386 address line such as A19, G1 can be connected to the 80386 M/IO pin through an inverter so that when M/IO=0 and A19=1, I/O ports are selected. The unused outputs 0₀ through 0₆ of the 74F138 decodes can be connected to other peripherals. The schematic of Figure 4.26 can be changed to memory-mapped I/O by connecting each of the 82C55's as above. Note that M/IO=1 and A19=0 can be used to select memory chips.

Example 4.14

Write an 80386 assembly language program to input two 32-bit data via Ports 5274H and 5270H, logically OR them together and then output the 32-bit result to Port 5270H.

Solution

```

MOV  DX, 5274H  ; Initialize DX
IN   EAX, DX    ; Input first 32-bit data
MOV  EBX, EAX   ; Save data
MOV  DX, 5270H  ; Initialize DX
IN   EAX, DX    ; Input second 32-bit data

```



```

OR    EAX, EBX    ; Or the two data
OUT   DX, EAX     ; Output to Port 5270H
HLT

```

Example 4.15

Prior execution of the 80386 INSD instruction, assume the following data:

```

EDI = 00000032H
Contents of address [ES:EDI] = 37124426H
DX = 0020H, DF = 0
Contents of PORT 0020H = EEEF2752H

```

What are the contents of EDI, [ES:EDI], DX, DF, and PORT 0020H after execution of the INSD?

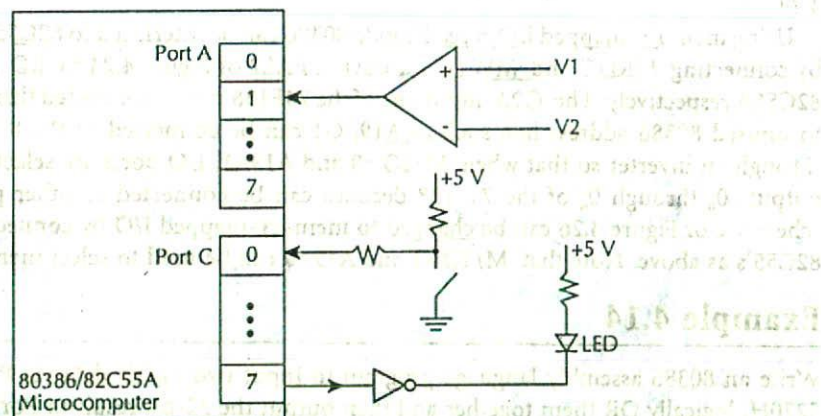
Solution

```

DF=0, EDI=00000036H
Contents of [ES:EDI]=EEEF2752H
DX=0020H
Contents of PORT 0020H=EEEF2752H

```

Example 4.16



For the above, write an 80386 assembly language program such that if $V_1 > V_2$, input the switch via bit 0 of 82C55A Port C and if the switch is closed, turn the LED ON.

Solution

```

MOV    AL, 90H    ; Configure Port A as input, Port C
                ; lower

```

```

OUT CNTRL, AL ; 4-bit as input and upper 4-bit as
               ; output
CHK: IN AL, PORTA ; Input Port A
    AND AL, 02H ; check  $V_1 > V_2$ 
    CMP AL, 02H ; check if  $V_1 > V_2$ 
    JNZ CHK ; Loop if  $V_1 < V_2$ 
    IN AL, PORT C ; Input switch
    BTC AL, 0 ; complement bit 0 of Port C
    RCR AL, 2
    OUT PORT C, AL ; output to Port C
    HLT

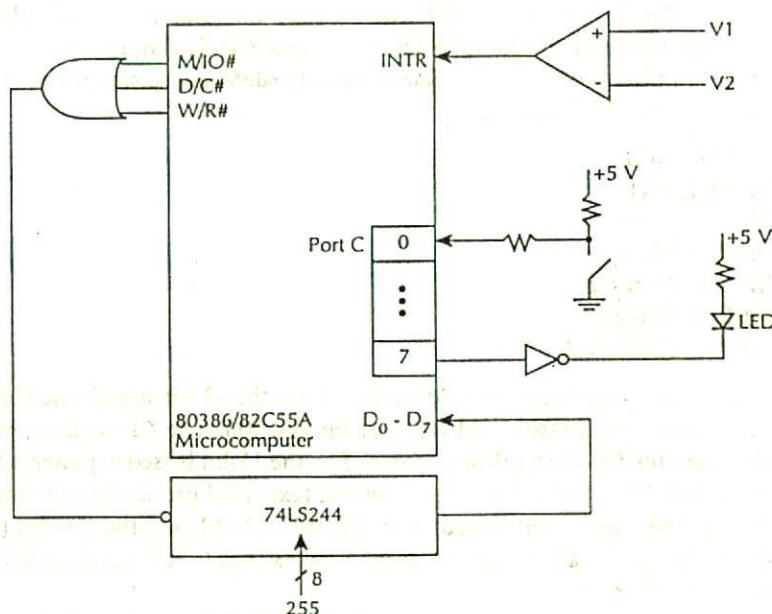
```

Example 4.17

Repeat Example 4.16 using interrupt I/O. Assume that the comparator output is connected to 80386 INTR pin. Use INT255.

Solution

The block diagram for servicing interrupt is as follows:



Note that M/I/O#=0, D/C#=0, and W/R#=0 indicate Interrupt Acknowledge

Assume all segment registers are initialized. Write service routine at IP=3000H and CS=0200H.

Main Program

```

MOV SP, VALUE ; Initialize SP
STI             ; Enable Interrupts
MOV AL, 90H    ; Configure

```



```

OUT    CNTRL, AL    ; Port C lower 4-bit as input,
                   ; upper 4-bit as output
WAIT   JMP    WAIT   ; wait for interrupt
HLT
Service Routine
ORG    30000200H
BEGIN: IN    AL, PORT C ; Input switch
      BTC    AL, 0      ; complement switch
      RCR    AL, 2
      OUT    PORT C, AL ; output to LED
      IRET
      ORG    000003FCH
      DD     02003000H ; Initialize IP = 0200H and
                   ; CS = 3000H

```

4.4 Coprocessor Interface

The performance of an 80386 system is enhanced by addition of a numeric coprocessor such as the 80287 or the 80387. The numeric coprocessor executes numeric instructions in parallel with the 80386. The 80386 automatically passes on these instructions to the coprocessor as it encounters them through I/O ports 800000F8H and 800000FCH.

The 80287 performs 16-bit transfers with a 16-bit data bus while the 80387 performs 32-bit transfers with a 32-bit data bus. The 80387 provides additional enhancements to the 80287 and includes full compatibility with the IEEE Floating-Point Standard draft 10.

The 80387 can utilize seven types of data using any 80386 addressing mode. These 80387 data types are listed in the following:

1. 32-bit Short real
2. 64-bit Long real
3. 80-bit Temporary real
4. 16-bit Word Integer
5. 32-bit Short Integer
6. 64-bit Long Integer
7. 80-bit Packed decimal

The 32-bit short real (single precision) format contains the 23-bit significand (bits 0-22), 8-bit biased exponent (bits 23-30), and the sign bit (bit 31). The 64-bit Long real (double precision) includes the 52-bit significand (bits 0-51), the 11-bit biased exponent (bits 52-62), and the sign bit (bit 63). The 80-bit Temporary real (extended precision) contains the 64-bit significand (bits 0-63), the 15-bit biased exponent (bits 64-78), and the sign bit (bit 79).

The 80-bit packed decimal contains 18 decimal digits (bits 0-71) and the sign bit (bit 79). Bits 72 thru 78 are undefined.

The 16-bit word integer includes the 15-bit integer (bits 0-14) along with the sign bit (bit 15) while the 32-bit short integer contains the 31-bit integer (bits 0-30) and the sign bit (bit 31). Finally, the 64-bit Long integer contains the 63-bit integer (bits 0-62) along with the sign bit (bit 63).

4.4.1 Coprocessor Hardware Concepts

The 80386 samples its ERROR# input during initialization to determine which coprocessor is present. The 80287 and 80387 require different interfaces and therefore, slightly different protocols.

For coprocessor cycles, the address pin A_{31} is HIGH. I/O addresses 800000F8H and 800000FCH are used for transferring data to and from a coprocessor. The 80386 automatically generates these addresses for coprocessor instructions. Also, the 80386 provides chip-select signals for the coprocessor using $A_{31}=1$ and $M/IO\#=0$.

The 80386 utilizes three input signals for controlling data transfers with coprocessors. These are $BUSY\#$, Coprocessor Request ($PEREQ$), and $ERROR\#$.

The $BUSY\#$ indicates that the coprocessor is presently executing an instruction and therefore cannot accept another instruction. A new instruction, therefore, cannot overrun the execution of the current coprocessor instruction.

$PEREQ$ indicates that the coprocessor needs to perform data transfer to or from memory. Since the coprocessor is never a bus master, all I/O transfers are performed by the 80386.

If a coprocessor math instruction results in an error that is not masked by the coprocessor's control register, the $ERROR\#$ signal is asserted. The coprocessor data sheets describe these errors and explain how to mask them by writing programs.

The interfacing characteristics of the 80386 with the 80387 Numeric coprocessor is described in the following.

The 80387 runs at an internal clock frequency of up to 16MHz. It is designed to run either fully synchronously or pseudo synchronously with the 80386. In the pseudo synchronous mode, the interface logic of the 80387 runs with the 80386 clock signal while internal logic runs with a different clock signal.

Figure 4.27 shows a typical 80386-80387 interface schematic. The main interfaces are described below:

- The 80386 and 80387 $BUSY\#$, $ERROR\#$, and $PEREQ$ signals are directly connected.
- The 82384 $RESET$ output is connected to the 80386 $RESET$ and 80387 $RESET IN$ signals.
- The 80387 chip select inputs, $NPS1\#$ and $NPS2$ are respectively connected to the 80386 $M/IO\#$ and A_{31} . With $M/IO\#=0$ and $A_{31}=1$, the 80386 selects the coprocessor.
- The command input ($CMD\#$) of the 80387 distinguishes data from commands. The 80386 A_2 is connected directly to the 80387 $CMD\#$. The 80386 outputs address 800000FCH when reading or writing data while address 800000F8H is used when writing a command or reading status.
- The 80387 uses $READY\#$ and $ADS\#$ pins to track bus activity and determine when $W/R\#$, $NPS1\#$, $NPS2$ and status Enable ($STEN$) can be sampled. $STEN$ is an 80387 chip select and is pulled HIGH. If multiple 80387's are used by one 80386, $STEN$ can be used to activate one 80387 at a time.
- Ready out ($READY0\#$) is an optional signal that can be used to generate the wait states required by a coprocessor. When the 80386 encounters a coprocessor instruction, it automatically generates one or more I/O cycles to addresses 800000F8H and 800000FCH. The 80386 performs all bus cycles to memory and transfers data to and from the 80387. All 80387 transfers are 32-bit wide. For 16-bit memories, the 80386 automatically performs the necessary conversion before transferring data with the 80387.

Read cycles (transfer from 80387 to the 80386) require at least one wait state while write cycles to the 80386 require no wait states. This requirement is automatically reflected in the state of the 80387 $READY0\#$ output which can be used to generate the required wait states.

Upon hardware reset, the 80386 before executing the first instruction, checks its $ERROR\#$ input to determine the type of coprocessor present. If the 80386 samples $ERROR\#$ low, it assumes that an 80387 is present. On the other hand, a high $ERROR\#$ input indicates either an 80287 is present or no coprocessor is used.

The coprocessor type can be determined via software by checking the ET bit in the machine status word. The 80387 is present if $ET=1$.

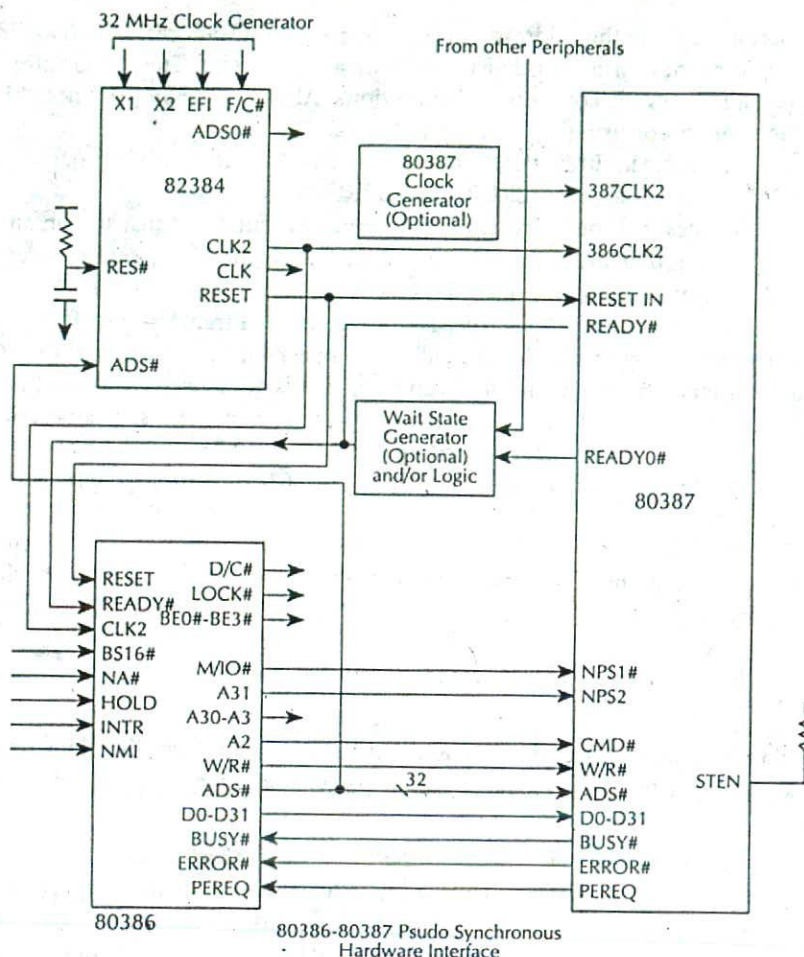


FIGURE 4.27 80386 system with 80387 coprocessor.

If the 80386 finds that an 80387 is present, an 80386 program must be written to execute the FNINT instruction to reset 80387 ERROR# output before any coprocessor transaction occurs.

When 80386 finds that either an 80287 is present or a coprocessor is not used, an 80386 program must be executed to determine the presence of an 80287 in order to set an internal status. An example of an 80386 instruction sequence to check the presence of an 80287 is given below:

;Initialization Routine to Detect An 80287 Numeric Processor

```
FND_287:  FNINIT          ; INITIALIZE NUMERIC
          ; PROCESSOR
```

```

FSTSW  AX      ; RETRIEVE 80287 STATUS WORD
OR     AL,AL    ; TEST LOW-BYTE 80287
                ; EXCEPTION FLAGS. IF ALL
                ; ZERO, THEN 80287 PRESENT AND
                ; PROPERLY INITIALIZED.
                ; IF NOT ALL ZERO, THEN 80287
JZ     GOT_287  ; ABSENT. BRANCH IF
                ; 80287 PRESENT
SMSW   AX      ; NO NUMERIC PROCESSOR
OR     AX,04H   ; SET EM BIT IN MACHINE
LMSW   AX      ; STATUS WORD TO ENABLE
                ; SOFTWARE EMULATION OF 80287

JMP     CONTINUE
GOT_287: SMSW   AX      ; NUMERIC PROCESSOR PRESENT
OR     AX,02H   ; SET MP BIT IN MACHINE
LMSW   AX      ; STATUS WORD TO PERMIT
                ; NORMAL 80287 OPERATION
CONTINUE:      ; AND OFF WE GO ....

```

In the above instruction sequence, the 80386 assumes that the 80287 is present. Therefore, it executes an FNINT instruction. Next, the 80386 reads the 80287 status word. If an 80287 is present, the lower 8 bits of this word (the exception flags) are all zeros. If an 80287 is not present, these data-lines are floating. If a pull-up resistor is connected to at least one of these lines, the absence of the 80287 is confirmed by at least one high bit in the lower eight bits of the status word. The routine then sets or resets the Emulate Coprocessor (EM) bit of the CR0 register of the 80386, depending on whether or not the 80287 is present.

4.4.2 Coprocessor Registers

Figure 4.28 shows the 80387 registers.

The 80387 contains three types of registers:

1. Eight 80-bit floating point stack registers
2. One 16-bit status word, one 16-bit control word and one 16-bit tag register
3. Four 32-bit error-pointer registers namely FIP, FCS, FOO, and FOS which store the instruction and memory operand causing an exception

The 80386 floating-point instructions consider the eight 80-bit registers as a stack of accumulators. The current stack top is called ST or ST(0). After a LOAD or PUSH, ST(0) becomes ST(1) and all other ST(i)'s are incremented by one. After a STORE or POP, ST(1) before the POP becomes the new ST(0), and all ST(i)'s are decremented by one. A 3-bit field name TOP in the status word defines the register number ST(0) of the current stack top. If TOP=000₂, a push will decrement TOP to 111₂ and store a new value into ST(7). ST(7) is the present stack top. On the other hand, if TOP=111₂, a POP will read a value from ST(7) and then increment top to 000₂ so that ST(0) becomes the new top of stack.

The 16-bit tag register includes eight 2-bit fields—one for each physical floating point register. This two-bit field indicates whether the corresponding physical floating-point register (0-7)

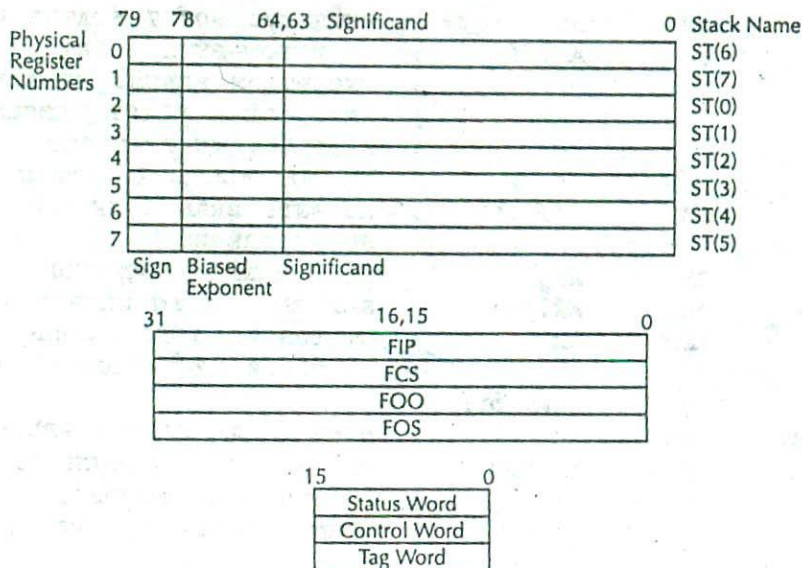


FIGURE 4.28 80387 registers.

rather than ST-relative names contains a valid, zero or special floating-point number (infinity, NAN) or is empty. The tag fields are used to indicate overflow or underflow of ST(i)'s. A stack overflow occurs if a PUSH decrements TOP to point to an ST(i) that is not empty. A stack underflow occurs if an empty ST(i) is popped. Stack underflow or overflow generates an invalid operation exception.

The control word register can be written by the program to control the 80387 operation. The control word contains exception mask bits for situations such as invalid operation, zero divide, overflow, and underflow. If the 80387 encounters an exception, the mask bit for that exception is checked to determine if the exception should be sent to a program error handler if the mask bit=0 or handled by an 80387 error handler if the mask bit=1.

The control word also contains information such as precision control and rounding control. The two-bit precision control field specifies that the results of arithmetic operations are rounded to 24-bit (short real), 53 bits (long real), or 64-bits (temporary real) precision before storing into the destination. All other operations use temporary real precision or a precision specified in the instruction.

On the other hand, the two bit rounding control field specifies that the results of floating-point operations are approximated or rounded to nearest toward minus infinity, plus infinity, or truncate toward zero.

The 80387 updates the bits of the status word register that can be checked by the program to check for special conditions. Bits 11, 12, and 13 specify the TOP field containing the three bit address of the physical register stack (0-7) corresponding to ST(0). The other bits in the status register include information such as indication of exception due to floating-point operations and floating-point condition code bits.

The four 32-bit 80387 Error Pointer register (FIP, FCS, FOO, and FOS) hold pointers to the last 80387 instruction executed along with its data. FCS stores the CS selector while FIP stores the IP offset. FOS, on the other hand, stores operand selector while FOO contains the operand offset.

4.4.3 80387 Instructions

The 80387 floating point instructions can be classified into six groups. These are data transfer, arithmetic, comparison, transcendental, loading, and control instructions. Some of these instructions are described in the following:

i) Data Transfer Instructions

- FLD mem or ST(n) mem can be short, long or temporary real. This instruction first decrements stack pointer by one and then loads the specified real number onto the new stack top.
Example:
Consider FLD mem32. If prior to execution of this FLD, contents of mem32 = 4.1572×10^3 with TOP = 4 then after FLD, contents of mem32 = 4.1572×10^3 , ST = 4.1572×10^3 with TOP = 3
- FXCH ST(n) This instruction exchanges the contents of ST(n) with the stack top.
Example:
Consider FXCH ST(5). If prior to execution of this instruction, ST = 5.71252×10^{-87} , ST(5) = -2.78164×10^{82} with TOP = 5 then after this FXCH, ST = -2.78164×10^{82} , ST(5) = 5.71252×10^{-87} with TOP = 5.
- FST mem or ST(n) The stack top is stored into specified memory or ST(n). FTSP mem or ST(n) is identical to FST except that the stack is popped after transfer.

ii) Arithmetic Instructions

- FABS This instruction converts the stack top to its absolute value.
Example:
If prior to execution of FABS, ST = -5.372×10^{-300} then after FABS, ST = 5.372×10^{-300}
- FADD
FADD ST(n) Performs real addition.
• FADD; ST \leftarrow ST + ST(1)
• FADD ST(n);
ST \leftarrow ST + ST(n)
Example:
If prior to execution of FADD,
ST = 3.17300×10^4 , ST(1) = 2.167×10^3
then after FADD,
ST = 3.38970×10^4 , ST(1) = 2.167×10^3
- FDIV
FDIV: ST \leftarrow ST/ST(1)
FDIVP ST(n),ST: ST(n) \leftarrow ST(n)/ST. Stack is popped.
FDIV ST(n): ST \leftarrow ST/ST(n)
FIDIV src: ST \leftarrow ST/src. Integer divide
src may be ST(n) or integer
Example:
If prior execution of FDIV,
ST = 4.240×10^{-4} , ST(1) = $8.480 \times 10^{+3}$
then after FDIV,
ST = 5.000×10^{-8} , ST(1) = $8.480 \times 10^{+3}$

-FIMUL src Integer multiply. $ST \leftarrow ST * src$
 -FMUL: Performs multiplication of real numbers.
 FMUL: $ST \leftarrow ST(1) * ST$
 FMUL src $ST \leftarrow ST * src$; src may be $ST(n)$ or a real number
 FMUL $ST(n), src$ $ST(n) \leftarrow ST(n) * src$; src may be $ST(n)$ or a real number
 Example:

If prior to execution of FMUL,
 $ST = 2.500 * 10^{-10}$, $ST(1) = 2.500 * 10^7$
 then after FMUL,
 $ST = 6.250 * 10^{-3}$, $ST(1) = 2.500 * 10^7$

-FSUB Performs real subtraction
 FSUB: $ST(1) \leftarrow ST(1) - ST$
 FSUB $ST(n)$: $ST \leftarrow ST - ST(n)$

Example:
 If prior execution of FSUB,
 $ST = 7.140 * 10^{10}$, $ST(1) = 8.217 * 10^{11}$
 then after FSUB,
 $ST(1) = 7.503 * 10^{11}$

-FSQRT $ST \leftarrow \sqrt{ST}$
 Example
 If prior execution of FSQRT,
 $ST = 2.25 * 10^6$
 then after FSQRT,
 $ST = 1.5 * 10^3$

iii) Comparison Instructions

-FCOM $ST(n)$ Compares $ST(n)$ numerically with top of stack. The condition codes C3, C2, and C0 (bits 14, 10, and 8 in the status word) are set according to the following:

	C3	C2	C0
$ST > ST(n)$	0	0	0
$ST < ST(n)$	0	0	1
$ST = ST(n)$	1	0	0

iv) Transcendental Instructions

-F2XMI: $ST \leftarrow 2^{ST} - 1$
 The range of the values of ST prior execution of F2XMI is -0.5 to $+0.5$.
 -FCOS This instruction computes the cosine of ST . ST is assumed to contain real numbers in radians. The result replaces the original ST .
 -FSIN This instruction computes the sine of ST . ST is assumed to contain real number in radians. The result replaces ST .
 -FSINCOS $ST \leftarrow \cos(ST)$
 $ST(1) \leftarrow \sin(ST)$
 -FYL2X $ST = ST(1) * \log_2(ST)$
 The stack is popped and the new stack top is replaced with the result.
 -FYL2XP1 $ST = ST(1) * \log_2(ST + 1.0)$
 The stack is popped and the new stack top is replace with the result of this computation.

v) Loading Instructions

-FLD n	The stack is pushed. The constant value, n is loaded into the new top of stack (ST) according to the following:
FLD1	Load 1.0
FLDL2E	Load $\log_2 e$
FLDL2T	Load $\log_2 10$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_2 2$
FLDPI	Load π
FLDZ	Load 0.0

vi) Control Instructions

-FLDCW mem16	Loads the control word with the contents of mem 16.
Example:	
	If prior execution of FLDCW mem16,
	CW = 2547H, mem16 = 25F2H
	then after FLDCW mem16,
	CW = 25F2H, mem16 = 25F2H.

Example 4.18

Write an 80386 assembly language program using 80387 floating point instructions to compute $Y = \sqrt{X^2 - Z^2}$.

Solution

```

FLD    Z           ; Load Z onto stack top
FMUL   ST, ST      ; Compute Z2
FLD    X           ; Load X onto stack
FMUL   ST, ST      ; compute X2
FSUB   ST, ST(1)   ; Compute X2 - Z2
FSQRT  ST           ; Y = SQRT(X2 - Z2)
HLT

```

Example 4.19

Write an 80386 assembly language program using 80387 floating point instructions to compute the volume of a sphere = $(4/3) * \pi * r^3$ where r is the radius of the sphere.

Solution

```

FLD    r           ; Load r to stack top
FST    ST(1)       ; Make a copy of r on stack
FMUL   ST, ST      ; compute r2
FMUL   ST, ST(1)   ; compute r3
FMUL   NUM         ; Compute 4*r3
FDIV   NUM1        ; Compute (4/3) * r3
FLDPI  ST           ; Load  $\pi$ 
FMUL   ST, ST(1)   ; Compute volume
HLT

```

The above program assumes NUM and NUM1, respectively store real numbers 4.0 and 3.0. Also, memory location r stores the radius.

QUESTIONS AND PROBLEMS

- 4.1 Write an 80186 assembly program to multiply a 16-bit signed number in BX by 00F3H. Assume that the result is 16 bits wide.
- 4.2 Identify the peripheral functional blocks integrated into the 80186.
- 4.3 What is the relationship between internal and external clocks of the 80186?
- 4.4 Identify the basic differences between 8086 and 80186.
- 4.5 Identify the main differences between the 80186 and 80286.
- 4.6 How much physical and virtual memory can the 80286 address?
- 4.7 What is the difference between the 80286 real address mode and PVAM? Explain how these two modes can be switched back and forth.
- 4.8 Explain how the 80286 determines where in memory the global descriptor table and the present local descriptor table are located.
- 4.9 Discuss briefly the 80286 protection mechanism.
- 4.10 Explain the meaning of 80286 call gates.
- 4.11 What is the purpose of 80286 CAP, COD/INTA pins?
- 4.12 Identify the 80286 pins used for interfacing it to a coprocessor.
- 4.13 Discuss the issues associated with isolating a user program from a supervisor program and then describe the 80286's protection features for protection. Assume that no task switching is involved. Also, assume that the supervisor program will perform all I/O operations and be present in the virtual memory space.
- 4.14 Compare the features of the 80386 with those of the 80286 from the following point of view: registers, clock rate, number of pins, number of instructions, modes of operation, memory management, and protection mechanism.
- 4.15 What are the basic differences between the 80386 real, protected, and virtual 8086 modes?
- 4.16 Assume the following register contents:

[EBX]	=	0000	2000H
[ECX]	=	0500	0000H
[EDX]	=	5000	5000H

prior to execution of each of the 80386 instructions listed below. Determine the effective address after execution of each instruction and identify the addressing modes of both source and destination:

- i) `MOV [EBX * 2] [ECX], EDX`
- ii) `MOV [EBX * 4] [ECX + 20H], EDX`

4.17 Determine the effect of each of the following 80386 instructions:

- i) **MOVZX ECX, BX**
 assume [ECX] = F1250024H
 [BX] = F130H

prior to execution of the MOVZX instruction.

- ii) **SHLD CX, BX, 0**
 if [CX] = 0025H, [BX] = 0F27H

prior to execution of the SHLD instruction.

4.18 Given the following data prior to the execution of each of the following instructions, determine the contents of the specified registers and carry flag after execution:

- i) **Prior to execution: [EAX]=0527AF12H**
 [EBX]=0071FFD1H
 CF=1

After execution of BTC EAX, EBX

- ii) **Prior execution: [BX]=F216H**
 CF=1

After execution of BTR BX, 20H

- iii) **Prior to execution: [EDX]=F214721FH**
 CF=1

After execution of: BTS EDX, 35H

4.19 Write an 80386 assembly language program to divide a signed 64-bit number in EBX:EAX by a 16-bit signed number in AX. Store the 32-bit quotient and remainder in memory locations.

4.20 Write an 80386 assembly program to compute X_i^2/N where $N = 100$ and X_i 's are signed 32-bit numbers. Assume that X_i^2 can be stored as a 32-bit signed number without overflow.

4.21 Write an 80386 assembly program to input 100 32-bit string data via a port addressed by DX. The program will then store the data in memory locations addressed by [DS] and [ESI].

4.22 Find a single 80386 MOV instruction with the appropriate addressing mode to replace the following 80386 instruction sequence:

```
MOV CL, 3
SAL ESI, CL
MOV EAX, [ESI]
```

4.23 Write an 80386 assembly language program to compute the following: $X = Y + Z - 20EEH$ where X, Y, and Z are 64-bit variables. The upper 32 bits of Y and Z are stored at 3000H and 3008H each followed by the lower 32 bits. Store the upper 32-bit of the 64-bit result at location at 4000H followed by the lower 32 bits.

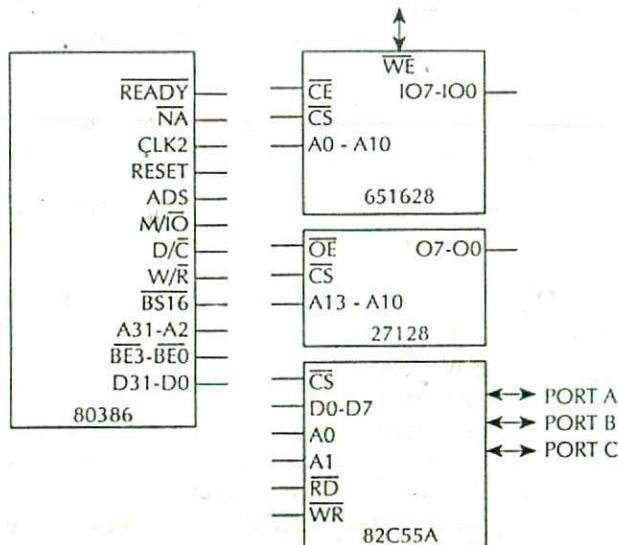
4.24 Assume that the 80386 registers BL, CX, and EDX contain a signed byte, a signed word, and a signed 32-bit word in two's complement form respectively. Write an 80386 assembly language program that will generate the signed result of operation $BL + CX - EDX \rightarrow EDX$.

4.25 Write an 80386 assembly language program to compute:

ECX=6*BX+EDX/EAX

where all numbers are signed numbers. Discard the remainder of EDX/EAX.

- 4.26 Write an 80386 assembly language program that checks all 32 bits of EAX from right to left (bit 0 to bit 31). The program will set the corresponding bit in EBX based on the location of the first one bit found in EAX. If the entire 32 bits of EAX are zero, clear EBX to all zeros.
- 4.27 Discuss how the following situation will be handled by the 80386: The 80386 executing an instruction causes both a general protection fault (interrupt 13) and coprocessor segment overrun (interrupt 9).
- 4.28 How does the 80386 generate the 32-bit physical address from A2-A31 and BE0#-BE3#?
- 4.29 What are the purposes of NA#, D/C#, BS16#, and ERROR# pins?
- 4.30 For 16- and 32-bit transfers, what is the logic level of the BS16# pin?
- 4.31 Discuss briefly the 80386 segmentation unit, paging unit, and protection.
- 4.32 How many bits are required for the address in real and protected modes?
- 4.33 Discuss the basic differences between the 80286 and 80386 descriptors.
- 4.34 What is the four-level hierarchical protection in protected mode?
- 4.35 Discuss briefly the 80386 virtual mode.
- 4.36 Consider the following pins and signals:



Draw a simplified diagram connecting the above chips to include the following:

- The 651628 to contain addresses
XXXXXXXX1H, XXXXXXXX5H, XXXXXXXX9H
.
- The 27128 to include addresses
XXXXXXXX0H, XXXXXXXX4H, XXXXXXXX8H,
.

-The 82C55A to contain port addresses

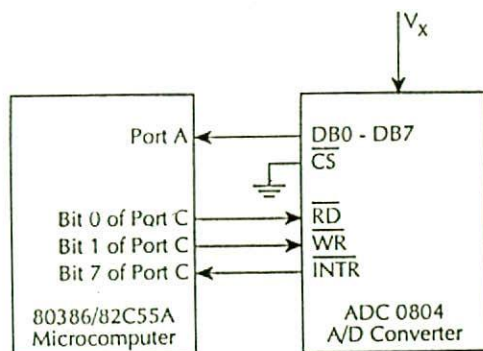
Port A = X1H

Port B = X5H

Port C = X9H

Assume all don't cares (X) to be zeros. Use PALs, latches, and other components as required. Determine memory and I/O maps.

4.37



The ADC 0804 can be started by sending a HIGH to LOW transition at the \overline{WR} pin. The conversion is completed when INTR is LOW. The 8-bit data at the DB0-DB7 pins can be read by the microcomputer by sending a LOW at the RD pin.

Write an 80386 assembly language program to start the 0804 by the microcomputer and input the converted data via Port A.

4.38 Repeat problem 4.37 by using:

- i) **NMI**
- ii) **INTR**

4.39 Assume 80386/80387 system. Write an 80386 assembly language program using floating point instructions to compute the mass of a sphere with radius r and density of Log_e^2 . Mass is equal to density times volume.

4.40 Assume 80386/80387 system. Write an 80386 assembly language program to compute the logarithm with a base other than 2; or log base 'n' of X ($\text{Log}_n X$).

MOTOROLA MC68000

This chapter describes the details of the Motorola 68000 microprocessor. The basic architecture, addressing modes, instruction set, and interfacing features of the Motorola 68000 are included.

5.1 Introduction

The MC68000 is Motorola's first 16-bit microprocessor. All 68000 address and data registers are 32 bits wide and its ALU is 16 bits wide. The 68000 is designed using HMOS technology. The 68000 requires a single 5V supply. The processor can be operated from a maximum internal clock frequency of 25 MHz. The 68000 is available in several frequencies. These include 6 MHz, 8 MHz, 10 MHz, 12.5 MHz, 16.67 MHz, and 25 MHz. The 68000 does not have on-chip clock circuitry and, therefore, requires a crystal oscillator or external clock generator/driver circuit to generate the clock.

The 68000 has several different versions. These include 68008, 68010, and 68012. The 68000 and 68010 are packaged in a 64-pin DIP (Dual In-line Package) with all pins assigned or in a 68-pin quad pack or Pin Grid Array (PGA) with some unused pins. The 68000 is also packaged in 68-terminal chip carrier. The 68008 is packaged in a 48-pin dual in-line package while the 68012 is packaged in 84-pin grid array. The 68008 provides the basic 68000 capabilities with inexpensive packaging. It has an 8-bit data bus which facilitates interfacing of this chip to inexpensive 8-bit peripheral chips.

The 68010 provides hardware-based virtual memory support and efficient looping instructions. Like the 68000, it has a 16-bit data bus and a 24-bit address bus.

The 68012 includes all the 68010 features with a 31-bit address bus.

The clock frequencies of the 68008, 68010, and 68012 are the same as the 68000.

The following table summarizes the basic differences among the 68000 family members:

	68000	68008	68010	68012
Data size (bits)	16	8	16	16
Address bus size (bits)	24	20	24	31
Virtual memory	No	No	Yes	Yes
Directly addressable	16 Mbytes	1 Mbyte	16 Mbytes	2 gigabytes memory

In order to implement operating systems and protection features, the 68000 can be operated in two modes. These are supervisor and user modes. The supervisor mode is also called the operating system mode. In this mode, the 68000 can execute all instructions. The 68000

TABLE 5.1 68000 User and Supervisor Modes

	Supervisor mode	User mode
Enter mode by	Recognition of a trap, reset, or interrupt	Clearing status bit S
System stack pointer	Supervisor stack pointer	User stack pointer registers A0-A6
Other stack pointers	User stack pointer and registers A0-A6	
Instructions available	All including STOP RESET MOVE to/from SR ANDI to/from SR ORI to/from SR EORI to SR MOVE USP to (An) MOVE to USP RTE	All except those listed under supervisor mode
Function code pin FC2	1	0

operates in one of these modes based on the S-bit of the Status register. When the S-bit is one, the 68000 operates in the supervisor mode. On the other hand, the 68000 operates in the user mode when $S = 0$.

Table 5.1 lists the basic differences between 68000 user and supervisor modes.

From Table 5.1 it can be seen that the 68000 executing a program in supervisor mode can enter the user mode by modifying the S-bit of the Status register to zero via an instruction. Instructions such as MOVE to SR, ORI to SR, EORI to SR can be used to accomplish this. On the other hand, the 68000 executing a program in user mode can enter the supervisor mode only via recognition of a trap, reset, or interrupt. Note that upon hardware reset, the 68000 operates in the supervisor mode and can execute all instructions. An attempt to execute privileged instructions (instructions that can only be executed in supervisor mode) in user mode will automatically generate an internal interrupt (trap) by the 68000.

The logical level in the 68000 Function Code pin (FC2) indicates to the external devices whether the 68000 is currently operating in user or supervisor mode. The 68000 has three function code pins (FC2, FC1, and FC0) which indicate to the external devices whether the 68000 is accessing supervisor program/data, user program/data, or performing an interrupt acknowledge cycle. These three pins are used for memory protection and for enabling an external chip such as the 74LS244 to provide an interrupt address vector.

The 68000 can operate on six different data types. These are bit, 4-bit BCD digit, 8-bit packed BCD, 8-bit byte, 16-bit word, and 32-bit long word.

The 68000 provides 56 basic instructions. With 14 addressing modes, 56 instructions, and 6 data types, the 68000 includes more than 1000 op codes. The fastest instruction is MOVE reg, reg and is executed in 500 ns at 8-MHz clock. The slowest instruction is 32-bit by 16-bit divide, which is executed in 21.25 μ s at 8-MHz clock.

Like the 8-bit Motorola microprocessors such as Motorola 6800 and 6809, the 68000 supports memory-mapped I/O. Thus, the 68000 instruction set does not include any IN or OUT instructions. This allows the full instruction set to be used by I/O.

The 68000 is a general-purpose register-based microprocessor since any data register can be used as an accumulator or as a scratch pad register. Even though the 68000 program counter is 32 bits wide, only the low-order 24 bits are used for PC. With 24 bits as address, the 68000 can directly address 16 megabytes (2^{24}) of memory.

5.2 68000 Programming Model

The register architecture of the 68000 is shown in Figure 5.1.

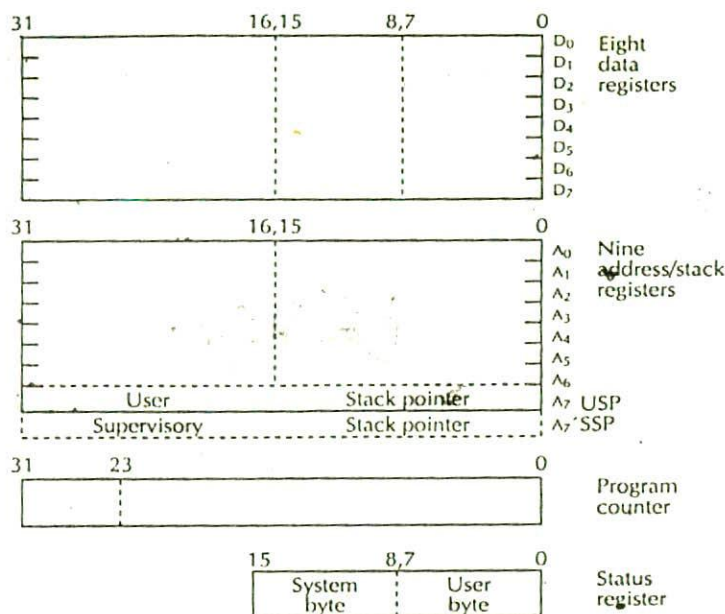


FIGURE 5.1 MC68000 programming model.

The 68000 chip contains eight 32-bit data registers (D0-D7) and nine 32-bit address registers (A0-A7, A7'). The 68000 uses A7 or A7' as the user or supervisor stack pointer, depending on the mode of operation. Data items such as bytes (8 bits), words (16 bits), long words (32 bits), and BCD numbers (8 bits) are usually stored in the data registers. On the other hand, the address of the operand is usually stored in an address register. Since the address sizes used by 68000 instructions can be either 16 or 24 bits, the address registers can only be used as 16- or 32-bit registers. While using the 32-bit address registers as addresses, the 68000 discards the uppermost eight bits (bits 24 thru 31).

The 68000 status register consists of two bytes. These are a user byte and a system byte (Figure 5.2). The user byte includes the usual condition codes such as C, V, N, Z, and X. The meaning of C, V, N, and Z flags is obvious. However, the X-bit (extend bit) has a special meaning. The 68000 does not have any ADDC or SUBC instructions; rather, it has ADDX or SUBX instructions. For arithmetic operations, the carry flag C and the extend flag X are affected in an identical manner. This means that one can use ADDX or SUBX to include carries or borrows while adding or subtracting high-order long words (32 bits) in multiprecision additions or subtractions.

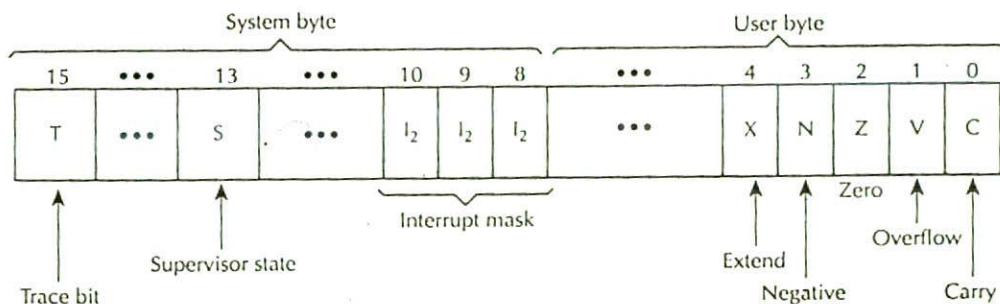
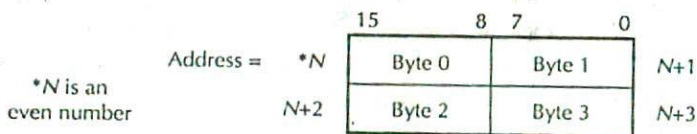


FIGURE 5.2 68000 status register.

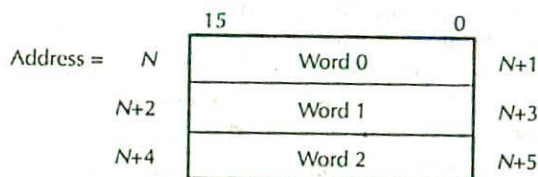
The system byte contains a 3-bit interrupt mask (I2, I1, I0), a supervisor flag (S), and a trace flag (T). Interrupt mask is set according to interrupt level recognized. A3-A1 pins of the 68000 provide the current interrupt level being serviced. The 68000 interrupt pins IPL2 IPL1 IPL0 = 000 indicate nonmaskable interrupt while IPL2 IPL1 IPL0 = 111 means no interrupt. The other combinations of IPL2, IPL1, and IPL0 provide the 68000 maskable interrupt levels. It should be pointed out that signals on IPL2, IPL1, and IPL0 pins are inverted and then reflected on I2, I1, and I0, respectively. When the S-bit in SR is 1, the 68000 operates in the supervisor mode. As mentioned before, when S = 0, the 68000 assumes user mode of operation. When the TF (trace flag) is set to one, the 68000 generates an internal interrupt (trap) after execution of each instruction. A debugging routine can be written at the interrupt address vector to display registers and/or memory after execution of each instruction. This provides single-stepping facility. The 68000 can be placed in the single-step mode by setting the TF bit in SR to one by executing a logical privileged instruction such as ORI # \$8000, SR in the supervisor mode.

5.3 68000 Addressing Structure

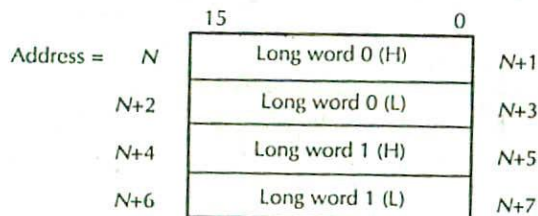
The 68000 supports 8-bit bytes, 16-bit words, and 32-bit long words as shown in Figure 5.3. Byte addressing contains both odd and even addresses (0, 1, 2, 3, 4,); and word and long-word addressing includes only even addresses in increments of 2 (0, 2, 4, 6,). As an example of the addressing structure, consider MOVE.L D0, \$102050. If [D0] = \$12345678, then after this MOVE, [\$102050] = \$12, [\$102051] = \$34, [\$102052] = \$56, and [\$102053] = \$78.



(a) 68000 Words Stored as Bytes (4 bytes).



(b) 68000 Word Structure (3 Words).



(c) 68000 Long Word Structure (2 long words).

For byte addressing (not shown in this figure), each byte can be uniquely addressed with bit 0 as the least significant bit and bit 7 as the most significant bit.

FIGURE 5.3 68000 addressing structure.

TABLE 5.2 68000 Addressing Modes

Mode	Generation	Assembler syntax
1. Register direct addressing		
Data register direct	$EA = Dn$	Dn
Address register direct	$EA = An$	An
2. Address register indirect addressing		
Register indirect	$EA = (An)$	(An)
Postincrement register indirect	$EA = (An), An \leftarrow An + N$	$(An)+$
Predecrement register indirect	$An \leftarrow An - N, EA = (An)$	$-(An)$
Register indirect with offset	$EA = (An) + d16$	$d(An)$
Indexed register indirect with offset	$EA = (An) + (Ri) + d8$	$d(An, Ri)$
3. Absolute data addressing		
Absolute short	$EA = (\text{Next word})$	$xxxx$
Absolute long	$EA = (\text{Next two words})$	$xxxxxxxx$
4. Program counter relative addressing		
Relative with offset	$EA = (PC) + d16$	d
Relative with index and offset	$EA = (PC) + (Ri) + d8$	$d(Ri)$
5. Immediate data addressing		
Immediate	$DATA = \text{Next word(s)}$	$\#xxxx$
Quick immediate	Inherent data	$\#xx$
6. Implied addressing		
Implied register	$EA = SR, USP, SP, PC$	

Notes: EA = effective address; An = address register; Dn = data register; Ri = address or data register used as index register; SR = status register; PC = program counter; USP = user stack pointer; SP = active system stack pointer (user or supervisor); d8 = 8-bit offset (displacement); d16 = 16-bit offset (displacement); N = 1 for byte, 2 for words, and 4 for long words; () = contents of; and \leftarrow = replaces

5.4 68000 Addressing Modes

Table 5.2 lists the 14 addressing modes of the 68000. The addressing modes are divided into six basic groups. These are register direct, address register indirect, absolute, program counter relative, immediate, and implied.

The 68000 contains three types of instructions: zero-operand, single-operand, and two-operand. The zero-operand instructions have no operands in the operand field. A typical example is CLC (Clear carry) instruction. The single-operand instructions contain the effective address, EA, in the operand field. The EAs of these instructions are calculated by the 68000 using the addressing mode specified for this operand. For two-operand instructions, one of the operands usually contains the EA and the operand is usually a register or memory location. The EAs in these instructions are calculated by the 68000 based on the addressing modes used for the EAs. Some two-operand instructions have an EA in both operands. This means that the operands in these instructions can use two different addressing modes.

As mentioned before, the 68000 address registers do not support byte-sized operands. Therefore, when an address register is used as a source, either the low-order word or the entire long-word operand is used depending on the operation size. On the other hand, when an address register is used as the destination, the entire register is affected regardless of the operation size. If the operation size is a word, and the destination is an address register, then the 68000 performs the 16-bit operation, places the result in the low 16 bits of the address register, and then sign-extends the address register to 32 bits. An example is MOVEA.W #8050, A5. In this case, the source operand is 16-bit immediate data 8050_{16} which is moved to the low 16 bits of A5, and the result is then sign-extended to 32 bits so that $[A5] = \text{FFFF}8050_{16}$. Data registers support data operands of byte, word, and long-word size.

5.4.1 Register Direct Addressing

In this mode, the eight data registers (D0-D7) or seven address registers (A0-A6) contain the data operand. For example, consider `ADD $005000, D0`. The destination of this instruction is in data register (direct mode).

Now, if $[005000] = 0002_{16}$, $[D0] = 0003_{16}$, then after execution of `ADD $005000, D0` the contents of $D0 = 0002 + 0003 = 0005$. Note that in the above instruction, the \$ symbol is used to represent hexadecimal numbers by Motorola. Also, note that instructions using address registers are not available for byte operations. In addition, in the 68000, the first operand of a two-operand instruction is the source and the second operand is the destination. Recall that in the 8086, the first operand is the destination while the second operand is the source.

5.4.2 Address Register Indirect Addressing

There are five different types of address register indirect mode. In the register indirect mode, an address register contains the effective address. For example, consider `CLR (A1)`. If $[A1] = \$003000$, then after execution of `CLR (A1)`, the contents of memory location $\$003000$ will be cleared to zero.

The postincrement address register indirect mode increments an address register by 1 for byte, 2 for word, and 4 for long word after it is used. For example, consider `CLR.L (A0) +`. If $[A0] = 005000_{16}$, then after execution of `CLR.L (A0) +`, the contents of locations 005000_{16} through 005003_{16} are cleared to zero and $[A0] = 005004_{16}$. The postincrement mode is typically used with memory arrays stored from LOW to HIGH memory locations. For example, in order to clear 1000_{16} words starting at memory location 003000_{16} , the following instruction sequence can be used:

```

MOVE.W #$1000,D0    ; Load length of data
                     ; into D0
MOVEA.L#$3000,A0    ; Load starting address
                     ; into A0
REPEAT CLR.W (A0)+   ; Clear a location
                     ; pointed to by A0 and
                     ; increment A0 by 2
SUBQ#1,D0            ; Decrement D0 by 1
BNE REPEAT           ; Branch to REPEAT if Z = 0, else
                     ; go to next instruction
-
-
-

```

Note that in the above, `CLR.W (A0)+` automatically points to the next location by incrementing $A0$ by 2 after clearing a memory location.

The predecrement address register indirect mode, on the other hand, decrements an address register by 1 for byte, 2 for word, and 4 for long word before using a register. For example, consider `CLR.W - (A0)`. If $[A0] = 002004_{16}$, then during execution of `CLR.W - (A0)`, the content of $A0$ is first decremented by 2; that is, $[A0] = 002002_{16}$, and the 16-bit contents of memory location 002002_{16} are then cleared to zero.

The predecrement mode is used with arrays stored from HIGH to LOW memory locations. For example, in order to clear 1000_{16} words starting at memory location 4000_{16} and below, the following instruction sequence can be used:

```

MOVE.W #$1000,D0    ; Load length of data into D0
MOVEA.L #$4002, A0  ; Load starting address plus 2
                     ; into A0

```



```

REPEAT CLR.W - (A0)      ; Decrement A0 by 2 and clear
                          ; the memory location addressed
                          ; by A0
      SUBQ#1, D0          ; Decrement D0 by one
      BNE REPEAT          ; If Z = 0, branch to REPEAT;
      -                  ; otherwise, go to next
      -                  ; instruction
      -

```

In the above, CLR.W - (A0) first decrements A0 by 2 and then clears the location. Since the starting address is 004000₁₆, A0 must initially be initialized with 004002₁₆.

It should be pointed out that the predecrement and postincrement modes can be combined in a single instruction. A typical example is MOVE.W(A5) +, -(A3).

The two other address register modes provide accessing of the tables by allowing offsets and indexes to be included with an indirect address pointer. The address register indirect with offset mode determines the effective address by adding a 16-bit signed integer to the contents of an address register. For example, consider MOVE.W \$10(A5), D3. If [A5] = 00002000₁₆, [002010₁₆] = 0014₁₆, then after execution of MOVE.W \$10(A5), D3, register D3 will contain 0014₁₆. A5 is unchanged.

The indexed register indirect with offset determines the effective address by adding an 8-bit signed integer and the contents of a register (data or address register) to the contents of an address (base) register. This mode is usually used when the offset from the base address register needs to be varied during program execution. The size of the index register can be a 16-bit or a 32-bit value.

As an example, consider MOVE.W \$10(A4, D3.W), D4. Note that in this instruction A4 is the base register and D3.W is the 16-bit index register (sign extended to 32 bits). This register can be specified as 32 bits by using D3.L in the instruction, and 10₁₆ is the 8-bit offset which is sign-extended to 32 bits. If [A4] = 00003000₁₆, [D3] = 0200₁₆, [003210₁₆] = 0024₁₆, then the above MOVE instruction will load 0024₁₆ into low 16 bits of register D4. A4 and D3 are unchanged.

The address register indirect with offset mode can be used to access one dimensional array such as a table where the offset (maximum 16 bits) can be the starting address of the table (fixed number) and the address register can hold the index number in the table to be accessed. Note that the starting address, plus the index number, provides the address of the element to be accessed in the table. For example, consider MOVE.W \$3400(A5), D1. If A5 contains 04, then this move instruction transfers the contents of 3404 (i.e., the fifth element, 0 being the first element) into low 16 bits of D1. The indexed register indirect with offset, on the other hand, can be used to access two dimensional arrays such as matrices.

5.4.3 Absolute Addressing

In this mode, the effective address is part of the instruction. The 68000 has two absolute addressing modes: absolute short addressing in which a 16-bit address is used (the address is sign-extended to 32 bits before use) and absolute long addressing in which a 24-bit address is used. For example, consider ADD \$2000, D2 as an example of absolute short mode. If [\$2000] = 0012₁₆, [D2] = 0010₁₆, then after execution of ADD \$2000, D2, the address \$2000 is sign-extended to 32 bits, whose low 24 bits are used as the address. Register D2 will then contain 0022₁₆. The absolute long addressing is used when the address size is more than 16 bits. For example, MOVE.W \$240000, D5 loads the 16-bit contents of location \$240000 into low 16 bits of D5. The absolute short mode includes an address ADDR in the range $0 \leq \text{ADDR} \leq \$7FFF$ or $\$FF8000 \leq \text{ADDR} \leq \$FFFFFF$. Note that a single instruction may use both short and long absolute modes, depending on whether the source or destination address is less than, equal to, or greater than the 16-bit address. A typical example is MOVE.W \$500002, \$1900.

5.4.4 Program Counter Relative Addressing

The 68000 has two program counter relative addressing modes: relative with offset, and relative with index and offset. In relative with offset, the effective address is obtained by adding the contents of the current PC with a sign-extended 16-bit displacement. This mode can be used when the displacement needs to be fixed during program execution. Typical branch instructions such as BEQ, BRA, and BLE use relative mode with offset. This mode can also be used by some other instructions. For example, consider $\text{ADD}^*+\$30, \text{D5}$ in which the source operand is relative to the offset mode. Note that typical assemblers use the symbol * to indicate offset. Now suppose that the current PC contents are 002000_{16} , the contents of 002030_{16} are 0005_{16} , and the low 16 bits of D5 contain 0010_{16} ; after execution of this ADD instruction, D5 will contain 0015_{16} .

In relative with index and offset, the effective address is obtained by adding the contents of the current PC, a signed 8-bit displacement (sign-extended to 32 bits), and the contents of an index register (address or data register). The size of the index register can be 16 or 32 bits wide. For example, consider $\text{ADD.W } \$4(\text{PC}, \text{D0.W}), \text{D2}$. If $[\text{D2}] = 00000012_{16}$, $[\text{PC}] = 002000_{16}$, $[\text{D0}]_{\text{low 16 bits}} = 0010_{16}$, and $[002014] = 0002_{16}$, then after this ADD, $[\text{D2}]_{\text{low 16 bits}} = 0014_{16}$. This mode is used when the displacement needs to be changed during program execution.

5.4.5 Immediate Data Addressing Mode

There are two immediate modes available with the 68000. These are the immediate and quick immediate modes. In the immediate mode, the operand data are constant data, which is part of the instruction. For example, consider $\text{ADD } \#\$0005, \text{D0}$. If $[\text{D0}] = 0002_{16}$, then after this ADD instruction, $[\text{D0}] = 0002_{16} + 0005_{16} = 0007_{16}$. Note that the # symbol is used by Motorola to indicate the immediate mode.

The quick immediate mode allows one to increment or decrement a register by a number from 0 to 7. For example, $\text{ADDQ } \#1, \text{D0}$ increments the contents of D0 by 1. Note that the data is inherent in the op code with the op code length of one word (16-bit). Data 0 to 7 are represented by three bits in the op code.

5.4.6 Implied Addressing

The instructions using this implicit mode do not require any operand, and registers such as PC, SP, or SR are implicitly referenced in these instructions. For example, RTE returns from an exception routine to the main program by using implicitly the PC and SR.

All 68000 addressing modes of Table 5.2 can further be divided into four functional categories as follows:

- **Data Addressing Mode.** An addressing mode is said to be a data addressing mode if it references data objects. For example, all 68000 addressing modes, except the address register direct mode, fall into this category.
- **Memory Addressing Mode.** An addressing mode that is capable of accessing a data item stored in the memory is classified as memory addressing mode. For example, the data and address register direct addressing modes cannot satisfy this definition.
- **Control Addressing Mode.** This refers to an addressing mode that has the ability to access a data item stored in the memory without the need to specify its size. For example, all 68000 addressing modes except the following are classified as control addressing modes:
 - Data register direct
 - Address register direct
 - Address register indirect with postincrement

Addressing Mode	Addressing Categories			
	Data	Memory	Control	Alterable
Data register direct	X	-	-	X
Address register direct	-	-	-	X
Address register indirect	X	X	X	X
Address register indirect with postincrement	X	X	-	X
Address register indirect with predecrement	X	X	-	X
Address register indirect with displacement	X	X	X	X
Address register indirect with index	X	X	X	X
Absolute short	X	X	X	X
Absolute long	X	X	X	X
Program counter with displacement	X	X	X	-
Program counter with index	X	X	X	-
Immediate	X	X	-	-

FIGURE 5.4 68000 addressing-functional categories.

- Address register indirect with predecrement
- Immediate
- Alterable Addressing Mode. If the effective address of an addressing mode is written into, then that mode is called alterable addressing mode. For example, the immediate and the program counter relative addressing modes will not satisfy this definition.

The addressing modes are classified into the four functional categories as shown in Figure 5.4.

5.5 68000 INSTRUCTION SET

The 68000 instruction set contains 56 basic instructions. Table 5.3 lists them in alphabetical order. Table 5.4 lists those affecting the condition codes. The repertoire is very versatile and offers an efficient means to handle high-level language data structures (such as arrays and linked lists). Note that in order to identify the operand size of an instruction, the following is placed after a 68000 mnemonic: .B for byte, .W or none for word, .L for long word. For example:

```
ADD.B      D0, D1 ; [D1]8 ← [D0]8 + [D1]8
ADD.W or ADD D0, D1 ; [D1]16 ← [D0]16 + [D1]16
ADD.L      D0, D1 ; [D1]32 ← [D0]32 + [D1]32
```


TABLE 5.3 68000 Instruction Set

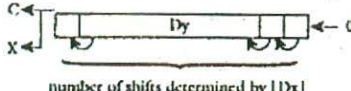
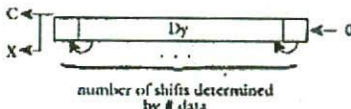
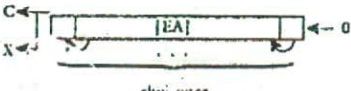
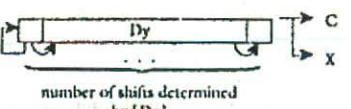
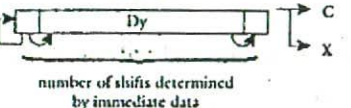
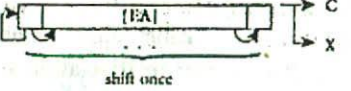
Instruction	Size	Length (words)	Operation
ABCD $-(Ay), -(Ax)$	B	1	$-\{Ay\}10 + -\{Ax\}10 + X \rightarrow \{Ax\}$
ABCD Dy, Dx	B	1	$\{Dy\}10 + \{Dx\}10 + X \rightarrow Dx$
ADD $(EA), (EA)$	B, W, L	1	$\{EA\} + \{EA\} \rightarrow EA$
ADDA $(EA), An$	W, L	1	$\{EA\} + An \rightarrow An$
ADDI $\#data, (EA)$	B, W, L	2 for B, W 3 for L	$data + \{EA\} \rightarrow EA$
ADDQ $\#data, (EA)$	B, W, L	1	$data + \{EA\} \rightarrow EA$
ADDX $-(Ay), -(Ax)$	B, W, L	1	$-\{Ay\} + -\{Ax\} + X \rightarrow \{Ax\}$
ADDX Dy, Dx	B, W, L	1	$Dy + Dx + X \rightarrow Dx$
AND $(EA), (EA)$	B, W, L	1	$\{EA\} \wedge \{EA\} \rightarrow EA$
ANDI $\#data, (EA)$	B, W, L	2 for B, W 3 for L	$data \wedge \{EA\} \rightarrow EA$
ANDI $\#data8, CCR$	B	2	$data8 \wedge \{CCR\} \rightarrow CCR$
ANDI $\#data16, SR$	W	2	$data16 \wedge \{SR\} \rightarrow SR$ if $s = 1$; else trap
ASL Dx, Dy	B, W, L	1	 <p>number of shifts determined by $\{Dx\}$</p>
ASL $\#data, Dy$	B, W, L	1	 <p>number of shifts determined by $\#data$</p>
ASL (EA)	B, W, L	1	 <p>shift once</p>
ASR Dx, Dy	B, W, L	1	 <p>number of shifts determined by $\{Dx\}$</p>
ASR $\#data, Dy$	B, W, L	1	 <p>number of shifts determined by immediate data</p>
ASR (EA)	B, W, L	1	 <p>shift once</p>
BCC d	B, W	1 for B 2 for W	Branch to $PC + d$ if carry = 0; else next instruction
BCHG $Dn, (EA)$	B, L	1	$\{bit\ of\ \{EA\},\ specified\ by\ Dn\}' \rightarrow Z$ $\{bit\ of\ \{EA\},\ specified\ by\ Dn\}' \rightarrow bit\ of\ \{EA\}$
BCHG $\#data, (EA)$	B, L	2	Same as BCHG $Dn, \{EA\}$ except bit number is specified by immediate data
BCLR $Dn, (EA)$	B, L	1	$\{bit\ of\ \{EA\}\}' \rightarrow Z$ $0 \rightarrow bit\ of\ \{EA\}\ specified\ by\ Dn$
BCLR $\#data, (EA)$	B, L	2	Same as BCLR $Dn, \{EA\}$ except the bit is specified by immediate data
BCS d	B, W	1 for B 2 for W	Branch to $PC + d$ if carry = 1; else next instruction
BEQ d	B, W	1 for B 2 for W	Branch to $PC + d$ if $Z = 1$; else next instruction
BGE d	B, W	1 for B 2 for W	Branch to $PC + d$ if greater than or equal; else next instruction

TABLE 5.3 68000 Instruction Set (continued)

Instruction	Size	Length (words)	Operation
BGT d	B, W	1 for B 2 for W	Branch to PC + d if greater than; else next instruction
BHI d	B, W	1 for B 2 for W	Branch to PC + d if higher; else next instruction
BLE d	B, W	1 for B 2 for W	Branch to PC + d if less or equal; else next instruction
BLS d	B, W	1 for B 2 for W	Branch to PC + d if low or same; else next instruction
BLT d	B, W	1 for B 2 for W	Branch to PC + d if less than; else next instruction
BMI d	B, W	1 for B 2 for W	Branch to PC + d if N = 1; else next instruction
BNE d	B, W	1 for B 2 for W	Branch to PC + d if Z = 0; else next instruction
BPL d	B, W	1 for B 2 for W	Branch to PC + d if N = 0; else next instruction
BRA d	B, W	1 for B 2 for W	Branch always to PC + d
BSET Dn, (EA)	B, L	1	[bit of (EA)]' \rightarrow Z 1 \rightarrow bit of (EA) specified by Dn
BSET #data, (EA)	B, L	2	Same as BSET Dn, (EA) except the bit is specified by immediate data
BSR d	B, W	1 for B 2 for W	PC \rightarrow -[SP] PC + d \rightarrow PC
BTST Dn, (EA)	B, L	1	[bit of (EA) specified by Dn]' \rightarrow Z
BTST #data, (EA)	B, L	2	Same as BTST Dn, (EA) except the bit is specified by data
BVC d	B, W	1 for B 2 for W	Branch to PC + d if V = 0; else next instruction
BVS d	B, W	1 for B 2 for W	Branch to PC + d if V = 1; else next instruction
CHK (EA), Dn	W	1	If Dn < 0 or Dn > [EA], then trap
CLR (EA)	B, W, L	1	0 \rightarrow EA
CMF (EA), Dn	B, W, L	1	Dn - [EA] \rightarrow Affect all condition codes except X
CMF (EA), An	W, L	1	An - [EA] \rightarrow Affect all condition codes except X
CMPI #data, (EA)	B, W, L	2 for B, W 3 for L	[EA] - data \rightarrow Affect all flags except X-bit
CMPM (Ay) +, (Ax) +	B, W, L	1	[Ax] + - [Ay] + \rightarrow Affect all flags except X; update Ax and Ay
DBCC Dn, d	W	2	If condition false, i.e., C = 1, then Dn - 1 \rightarrow Dn; if Dn \neq -1, then PC + d \rightarrow PC; else PC + 2 \rightarrow PC
DBCS Dn, d	W	2	Same as DBCC except condition is C = 1
DBEQ Dn, d	W	2	Same as DBCC except condition is Z = 1
DBF Dn, d	W	2	Same as DBCC except condition is always false
DBGE Dn, d	W	2	Same as DBCC except condition is greater or equal
DBGT Dn, d	W	2	Same as DBCC except condition is greater than
DBHI Dn, d	W	2	Same as DBCC except condition is high
DBLE Dn, d	W	2	Same as DBCC except condition is less than or equal
DBLS Dn, d	W	2	Same as DBCC except condition is low or same
DBLT Dn, d	W	2	Same as DBCC except condition is less than
DBMI Dn, d	W	2	Same as DBCC except condition is N = 1
DBNE Dn, d	W	2	Same as DBCC except condition Z = 0
DBPL Dn, d	W	2	Same as DBCC except condition N = 1
DBT Dn, d	W	2	Same as DBCC except is always true
DBVC Dn, d	W	2	Same as DBCC except condition is V = 0
DBVS Dn, d	W	2	Same as DBCC except condition is V = 1

TABLE 5.3 68000 Instruction Set (continued)

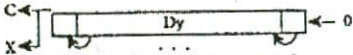
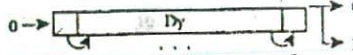
Instruction	Size	Length (words)	Operation
DIVS (EA), Dn	W	1	Signed division [Dn]32/[EA]16 → [Dn]0-7 = quotient [Dn]8-15 = remainder
DIVU (EA), Dn	W	1	Same as DIVS except division is unsigned
EOR Dn, (EA)	B, W, L	1	$Dn \oplus [EA] \rightarrow EA$
EORI #data, (EA)	B, W, L	3 for L 2 for B, W	$data \oplus [EA] \rightarrow EA$
EORI #d8, CCR	B	2	$d8 \oplus CCR \rightarrow CCR$
EORI #d16, SR	W	2	$d16 \oplus SR \rightarrow SR$ if S = 1; else trap
EXG Rx, Ry	L	1	$Rx \leftrightarrow Ry$
EXT Dn	W, L	1	Extend sign bit of Dn from 8-bit to 16-bit or from 16-bit to 32-bit depending on whether the operand size is B or W
JMP (EA)	Unsize	1	[EA] → PC
JSR (EA)	Unsize	1	Unconditional jump using address in operand PC → -[SP]; [EA] → PC
LEA (EA), An	L	1	Jump to subroutine using address in operand [EA] → An
LINK An, #-d	Unsize	2	$An \leftarrow -[SP]; SP \rightarrow An; SP - d \rightarrow SP$
LSL Dx, Dy	B, W, L	1	
LSL #data, Dy	B, W, L	1	Same as LSL Dx, Dy except immediate data specify the number of shifts from 0 to 7
LSL (EA)	B, W, L	1	Same as LSL Dx, Dy except left shift is performed only once
LSR Dx, Dy	B, W, L	1	
LSR #data, Dy	B, W, L	1	Same as LSR except immediate data specifies the number of shifts from 0 to 7
LSR (EA)	B, W, L	1	Same as LSR, Dx, Dy except the right shift is performed once only
MOVE (EA), (EA)	B, W, L	1	$[EA]_{source} \rightarrow [EA]_{destination}$
MOVE (EA), CCR	W	1	$[EA] \rightarrow CCR$
MOVE CCR, (EA)	W	1	$CCR \rightarrow [EA]$
MOVE (EA), SR	W	1	If S = 1, then $[EA] \rightarrow SR$; else TRAP
MOVE SR, (EA)	W	1	If S = 1, then $SR \rightarrow [EA]$; else TRAP
MOVE An, USP	L	1	If S = 1, then $An \rightarrow USP$; else TRAP
MOVE USP, An	W, L	1	$[USP] \rightarrow An$
MOVEM register list, (EA)	W, L	2	Register list → [EA]
MOVEM (EA), register list	W, L	2	[EA] → register list
MOVEP Dx, d(Ay)	W, L	2	$Dx \rightarrow d[AY]$
MOVEP d(Ay), Dx	W, L	2	$d[AY] \rightarrow Dx$
MOVEQ #d8, Dn	L	1	d8 sign extended to 32-bit → Dn
MULS (EA)16, (Dn)16	W	1	Signed 16 × 16 multiplication $[EA]16 * [Dn]16 \rightarrow [Dn]32$
MULU (EA)16, (Dn)16	W	1	Unsigned 16 × 16 multiplication $[EA]16 * [Dn]16 \rightarrow [Dn]32$
NBCD (EA)	B	1	$0 - [EA]10 - X \rightarrow EA$
NEG (EA)	B, W, L	1	$0 - [EA] \rightarrow EA$
NEGX (EA)	B, W, L	1	$0 - [EA] - X \rightarrow EA$
NOP	Unsize	1	No operation
NOT (EA)	B, W, L	1	$[EA]^c \rightarrow EA$
OR (EA), (EA)	B, W, L	1	$[EA] \vee [EA] \rightarrow EA$
ORI #data, (EA)	B, W, L	2 for B, W 3 for L	$data \vee [EA] \rightarrow EA$

TABLE 5.3 68000 Instruction Set (continued)

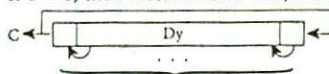
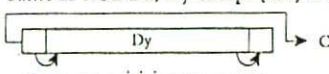
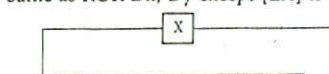

Instruction	Size	Length (words)	Operation
ORI #d8, CCR	B	2	d8VCCR → CCR
ORI #d16, SR	W	2	If S = 1, then d16VSR → SR; else TRAP
PEA (EA)	L	1	[EA]16 sign extend to 32 bits → [SP]
RESET	Unsize	1	If S = 1, then assert RESET line; else TRAP
ROL Dx, Dy	B, W, L	1	
ROL #data, Dy	B, W, L	1	Same as ROL Dx, Dy except immediate data specifies number of times to be rotated from 0 to 7
ROL (EA)	B, W, L	1	Same as ROL Dx, Dy except [EA] is rotated once
ROR Dx, Dy	B, W, L	1	
ROR #data, Dy	B, W, L	1	Same as ROR Dx, Dy except the number of rotates is specified by immediate data from 0 to 7
ROR (EA)	B, W, L	1	Same as ROR Dx, Dy except [EA] is rotated once
ROXL Dx, Dy	B, W, L	1	
ROXL #data, Dy	B, W, L	1	Same as ROXL Dx, Dy except immediate data specifies number of rotates from 0 to 7
ROXL (EA)	B, W, L	1	Same as ROXL Dx, Dy except [EA] is rotated once
ROXR Dx, Dy	B, W, L	1	
ROXR #data, Dy	B, W, L	1	Same as ROXR Dx, Dy except immediate data specifies number of rotates from 0 to 7
ROXR (EA)	B, W, L	1	Same as ROXR Dx, Dy except [EA] is rotated once
RTE	Unsize	1	If S = 1, then [SP]+ → SR; [SP]+ → PC, else TRAP
RTR	Unsize	1	[SP] +→ CC; [SP] +→ PC
RTS	Unsize	1	[SP] +→ PC
SBCD -(Ay), -(Ax)	B	1	-(Ax)10 - (Ay)10 - X → (Ax)
SBCD Dy, Dx	B	1	[Dx]10 - [Dy]10 - X → Dx
SCC (EA)	B	1	If C = 0, then 1s → [EA] else 0s → [EA]
SCS (EA)	B	1	Same as SCC except the condition is C = 1
SEQ (EA)	B	1	Same as SCC except if Z = 1
SF (EA)	B	1	Same as SCC except condition is always false
SGE (EA)	B	1	Same as SCC except if greater or equal
SGT (EA)	B	1	Same as SCC except if greater than
SHI (EA)	B	1	Same as SCC except if high
SLE (EA)	B	1	Same as SCC except if less or equal
SLS (EA)	B	1	Same as SCC except if low or same
SLT (EA)	B	1	Same as SCC except if less than
SMI (EA)	B	1	Same as SCC except if N = 1
SNE (EA)	B	1	Same as SCC except if Z = 0
SPL (EA)	B	1	Same as SCC except if N = 0
ST (EA)	B	1	Same as SCC except condition always true
STOP #data	Unsize	2	If S = 1, then data → SR and stop; TRAP if executed in user mode
SUB (EA), (EA)	B, W, L	1	[EA] - [EA] → EA
SUBA (EA), An	W, L	1	An - [EA] → An
SUBI #data, (EA)	B, W, L	2 for B, W 3 for L	[EA] - data → EA
SUBQ #data, (EA)	B, W, L	1	[EA] - data → EA

TABLE 5.3 68000 Instruction Set (continued)

Instruction	Size	Length (words)	Operation
SUBX - (Ay), - (Ax)	B, W, L	1	$-(Ax) \leftarrow -(Ay) - X \rightarrow [Ax]$
SUBX Dy, Dx	B, W, L	1	$Dx \leftarrow Dx - X \rightarrow Dx$
SVC (EA)	B	1	Same as SCC except if V = 0
SVS (EA)	B	1	Same as SCC except if V = 1
SWAP Dn	W	1	$Dn [31:16] \leftrightarrow Dn [15:0]$
TAS (EA)	B	1	[EA] tested; N and Z are affected accordingly; 1 \rightarrow bit 7 of [EA]
TRAP #vector	Unsize	1	$PC \rightarrow -[SSP]$, $SR \rightarrow -[SSP]$, (vector) $\rightarrow PC$; 16 TRAP vectors are available
TRAPV	Unsize	1	If V = 1, then TRAP; else next instruction
TST (EA)	B, W, L	1	[EA] - 0 \rightarrow condition codes affected; no result provided
UNLK An	Unsize	1	$An \rightarrow SP$; $[SP] \rightarrow An$

All 68000 instructions may be classified into eight groups, as follows:

- i) Data movement instructions
- ii) Arithmetic instructions
- iii) Logical instructions
- iv) Shift and rotate instructions
- v) Bit manipulation instructions
- vi) Binary-coded decimal instructions

TABLE 5.4 68000 Instructions Affecting the Condition Codes

Instruction	X	N	Z	V	C
ABCD	+	U	+	U	—
ADD, ADDI, ADDQ, ADDX	+	+	+	+	+
AND, ANDI	—	+	+	0	0
ASL, ASR	+	+	+	+	+
BCHG, BCLR, BSET, BTST	—	—	+	—	—
CHK	—	+	U	U	U
CLR	—	0	1	0	0
CMP CMPA, CMPI, CMPM	—	+	+	+	+
DIVS, DIVU	—	+	+	+	0
EOR, EORI	—	+	+	0	0
EXT	—	+	+	0	0
LSL, LSR	+	+	+	0	+
MOVE (EA), (EA)	—	+	+	0	0
MOVE TO CC	+	+	+	+	+
MOVE TO SR	+	+	+	+	+
MOVEQ	—	+	+	0	0
MULS, MULU	—	+	+	0	0
NBCD	+	U	+	U	+
NEG, NEGX	+	+	+	+	+
NOT, OR, ORI	—	+	+	0	0
ROL, ROR	—	+	+	0	+
ROXL, ROXR	+	+	+	0	+
RTE, RTR	+	+	+	+	+
SBCD	+	U	+	U	+
STOP	+	+	+	+	+
SUB, SUBI, SUBQ, SUBX	+	+	+	+	+
SWAP	—	+	+	0	0
TAS	—	+	+	0	0
TST	—	+	+	0	0

Note: +, affected; —, not affected; U, undefined.

TABLE 5.5 68000 Data Movement Instructions

Instruction	Size	Comment
EXG Rx, Ry	L	Exchange the contents of two registers; Rx or Ry can be any address or data register; no flags are affected
LEA (EA), An	L	The effective address [EA] is calculated using the particular addressing mode used and then loaded into the address register; [EA] specifies the actual data to be loaded into An
LINK An, #displacement	Unsize	The current contents of the specified address register are pushed onto the stack; after the push, the address register is loaded from the updated SP; finally, the 16-bit sign-extended displacement is added to the SP; a negative displacement is specified to allocate stack
MOVE (EA), (EA)	B, W, L	[EA]s are calculated by the 68000 using the specific addressing mode used; [EA]s can be register or memory location; therefore, data transfer can take place between registers, between a register and a memory location, and between different memory locations; flags are affected; for byte-size operation, address register direct is not allowed; An is not allowed in the destination [EA]; the source [EA] can be An for word or long-word transfers
MOVEM reg list, (EA) or (EA), reg list	W, L	Specified registers are transferred to or from consecutive memory locations starting at the location specified by the effective address
MOVEP Dn, d (Ay) or d (Ay), Dn	W, L	Two [W] or four [L] bytes of data are transferred between a data register and alternate bytes of memory, starting at the location specified and incrementing by 2; the high-order byte of data is transferred first, and the low-order byte is transferred last. This instruction has the address register indirect with displacement-only mode
MOVEQ #data, Dn	L	This instruction moves the 8-bit inherent data into the specified data register; the data are then sign-extended to 32 bits
PEA (EA)	L	Computes an effective address and then pushes the 32-bit address onto the stack
SWAP Dn	W	Exchanges 16-bit halves of a data register
UNLK An	Unsize	An → SP; [SP] → An

- [EA] in LEA [EA], An can use all addressing modes except Dn, An, [An] +, −[An], and immediate.
- Destination [EA] in MOVE [EA], [EA] can use all modes except An, relative, and immediate.
- Source [EA] in MOVE [EA], [EA] can use all modes.
- Destination [EA] in MOVEM reg list, [EA] can use all modes except Dn, An, [An] +, relative, and immediate.
- Source [EA] in MOVEM [EA], reg list can use all modes except Dn, An, −[An], and immediate.
- [EA] in PEA [EA] can use all modes except Dn, An, [An] +, −[An], and immediate.

vii) Program control instructions

viii) System control instructions

5.5.1 Data Movement Instructions

These instructions allow data transfers from register to register, register to memory, memory to register, and memory to memory. In addition, there are also special data movement instructions such as MOVEM (Move multiple registers). Typically, byte, word, or long-word data can be transferred. Table 5.5 lists the 68000 data movement instructions.

5.5.1.a MOVE Instructions

The format for the basic MOVE instruction is MOVE.S (EA), (EA), where S = .L, .W, or .B. (EA) can be a register or memory location depending on the addressing mode used. Consider MOVE.B D2, D0 which uses a data register direct mode for both the source and destination. Now if [D2] = 03₁₆, [D0] = 01₁₆, then after execution of this MOVE instruction [D2] = 03₁₆ and [D0] = 03₁₆.

There are several variations of the MOVE instruction. For example, MOVE.W CCR, (EA) moves the content of the low-order byte of the SR, i.e., CCR, to the low-order byte of the

destination operand, and the upper byte of SR is considered as zero. The source operand is a word. Similarly, MOVE.W (EA), CCR moves an 8-bit immediate number or low-order 8-bit data from a memory location or register into the condition code register; the upper byte is ignored. The source operand is a word. Data can also be transferred between (EA) and SR or USP (in the supervisor mode when S = 1) using the following privileged instructions:

```
MOVE.W (EA), SR
MOVE.W SR, (EA)
MOVE.L USP, An
MOVE.L An, USP
```

MOVEA.W or .L (EA), An can be used to load an address into an address register. Word size source operands are sign-extended to 32 bits. Note that (EA) is obtained using an addressing mode. As an example, MOVEA.W #\$8000, A0 moves the 16-bit word 8000_{16} into the low 16 bits of A0 and then sign-extends 8000_{16} to the 32-bit number $FFFF8000_{16}$.

Note that sign extension means extending bit 15 of 8000_{16} from bit 16 through bit 31. As mentioned, sign extension is required when an arithmetic operation between two signed binary numbers of different sizes is performed. (EA) in MOVEA can use all addressing modes. MOVEM instruction can be used to PUSH or POP multiple registers to or from the stack. For example, MOVEM.L D0-D7/A0-A6, - (SP) saves the contents of all of the 8 data registers and 7 address registers in the stack. This instruction stores address registers in the order A6-A0 first, followed by data registers in the order D7-D0, regardless of the order in the list. MOVEM.L (SP)+, D0-D7/A0-A6 restores the contents of the registers in the order D0-D7, A0-A6 regardless of the order in the list. MOVEM instruction can also be used to save a set of registers in memory. In addition to the above predecrement and postincrement modes for the effective address, MOVEM instruction allows all the control modes. If the effective address uses one of the control modes, such as absolute short, then the registers are transferred starting at the specified address and up through higher addresses. The order of transfer is from D0 to D7 and then from A0 to A7. For example, MOVEM.W A4/D1/D3/A0-A2, \$8000 transfers the low 16-bit contents of D1, D3, A0, A1, A2, and A4 to locations \$8000, \$8002, \$8004, \$8006, \$8008, and \$800A, respectively.

The MOVEQ.L #d8, Dn moves the immediate 8-bit data into the low byte of Dn. The 8-bit data are then sign-extended to 32 bits. This is a one-word instruction. For example, MOVEQ.L #\$FF, D0 moves $FFFFFFF_{16}$ into D0.

In order to transfer data between the 68000 data registers and 6800 (8-bit) peripherals, the MOVEP instruction can be used. This instruction transfers two to four bytes of data between a data register and alternate byte locations in memory, starting at the location specified in increments of 2. Register indirect with displacement is the only addressing mode used with this instruction. If the address is even, all the transfers are made on the high-order half of the data bus. The high-order byte from the register is transferred first and the low-order byte is transferred last. For example, consider MOVEP.L \$0050 (A0), D0. If $[A0] = 00003000_{16}$, $[003050] = 01_{16}$, $[003052] = 03_{16}$, $[003054] = 02_{16}$, $[003056] = 04_{16}$, then after the execution of the above MOVEP instruction, D0 will contain 01030204_{16} .

5.5.1.b EXG and SWAP Instructions

The EXG.L Rx, Ry instruction exchanges the 32-bit contents of Rx with that of the Ry. The exchange is between two data registers, two address registers, or between an address and a data register. The EXG instruction exchanges only 32-bit-long words. The data size (L) does not have to be specified after the EXG instruction since this instruction has only one data size. No flags are affected.

The SWAP.W Dn instruction, on the other hand, exchanges the low 16 bits of Dn with the high 16 bits of Dn. All condition codes are affected.

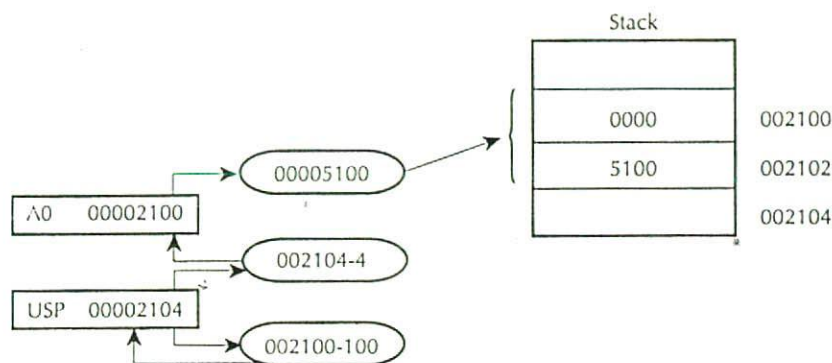


FIGURE 5.5 Execution of the link instruction.

5.5.1.c LEA and PEA Instructions

The LEA.L (EA), An moves an effective address (EA) into the specified address register. The (EA) can be calculated based on the addressing mode of the source. For example, LEA \$00456074, A1 moves \$00456074 to A1. This instruction is equivalent to MOVEA.L #\$00456074, A1. Note that \$00456074 is contained in the PC.

LEA instruction is very useful when address calculation is desired during program execution. (EA) in LEA specifies the actual data to be loaded into An, whereas (EA) in MOVEA specifies the address of actual data. For example, consider LEA \$06(A2, D5.W), A0. If [A2] = 00003000₁₆, [D5] = 0044₁₆, then the LEA instruction moves 0000304A₁₆ into A0. On the other hand, MOVEA \$06(A2, D5.W), A0 moves the contents of 0000304A₁₆ into A0. Therefore, it is obvious that if address calculation is required, the instruction LEA is very useful.

The PEA (EA) computes an effective address and then pushes it onto the stack. This instruction can be used when the 16-bit address used in absolute short mode is required to be pushed onto the stack. For example, consider PEA \$8000 in the user mode. If [USP] = \$00005004, then \$8000 is sign-extended to 32 bits and pushed to stack. The low-order 16 bits (\$8000) are pushed at \$005002₁₆ and the high-order 16 bits (\$FFFF) are pushed at 005000₁₆.

5.5.1.d LINK and UNLK Instructions

Before calling a subroutine, the main program quite often transfers values of certain parameters to the subroutine. It is convenient to save these variables onto the stack before calling the subroutine. These variables can then be read from the stack and used by the subroutine for computations. The 68000 LINK and UNLK instructions are used for this purpose. In addition, the 68000 LINK instruction allows one to reserve temporary storage for the local variables of a subroutine. This storage can be accessed as needed by the subroutine and be released using UNLK before returning to the main program. The LINK instruction is usually used at the beginning of a subroutine to allocate stack space for storing local variables and parameters for nested subroutine calls. The UNLK instruction is usually used at the end of subroutine before the RETURN instruction to release the local area and restore the stack pointer contents so that it points to the return address.

The LINK An, #-displacement causes the current contents of the specified An to be pushed onto the system (user or supervisor) stack. The updated SP contents are then loaded into An. Finally, a sign-extended 2's complement displacement value is added to the SP. No flags are affected. For example, consider LINK A0, #-\$100. If [A0] = 00005100₁₆, [USP] = 00002104₁₆ then after execution of the LINK instruction the situation shown in Figure 5.5 occurs.

This means that after the LINK instruction, [A0] = 00005100₁₆ is pushed onto the stack, the [updated USP] = 002100₁₆ is loaded into A0. USP is then loaded with 002000₁₆ and, therefore,

100₁₆ locations are allocated to subroutine at the beginning of which the above LINK instruction can be used. Note that A0 cannot be used in the subroutine.

The UNLK instruction at the end of this subroutine before the RETURN instruction releases the 100₁₆ locations and restores the contents of A0 and USP to those prior to using the LINK instruction. For example, UNLK A0 will load [A0] = 00002100₁₆ into USP, the two stack words 00005100₁₆ into A0, and USP is then incremented by 4 to contain 00002104₁₆. Therefore, the contents of A0 and USP prior to using the LINK are restored. In the above example, after execution of the LINK, addresses 001FFF₁₆ and below can be used as the stack. Sixty-four (\$100) locations starting at 002000₁₆ and above can be reserved for storing the local variables of the subroutine. These variables can then be accessed with an address register such as A0 as a base pointer using the address register indirect with displacement mode such as MOVE.W d(A0), D2 for read and MOVE.W D2, d(A0) for write.

5.5.2 Arithmetic Instructions

These instructions allow:

- 8-, 16-, or 32-bit additions and subtractions
- 16-bit by 16-bit multiplication (both signed and unsigned)
- Compare, clear, and negate instructions
- Extended arithmetic instructions for performing multiprecision arithmetic
- Test (TST) instruction for comparing the operand with zero
- Test and set (TAS) instruction which can be used for synchronization in multiprocessor system

The 68000 arithmetic instructions are summarized in Table 5.6.

5.5.2.a Addition and Subtraction Instructions

- Consider ADD.W \$245000, D0. If [245000]₁₆ = 2014₁₆ and [D0] = 1004₁₆, then after execution of this ADD instruction, the low 16 bits of D0 will contain 3018₁₆.
- ADDI instruction can be used to add immediate data to register or memory location. The immediate data follows the instruction word. For example, consider ADDI.W #50062, \$500000. If [500000]₁₆ = 1000₁₆, then after execution of this ADDI instruction, memory location 500000₁₆ will contain 1062₁₆. ADDQ, on the other hand, adds a number from 0 to 7 to the register or memory location in the destination operand. This instruction occupies 16 bits and the immediate data 0 to 7 is specified by 3 bits in the instruction word. For example, consider ADDQ.B#4, D0. If [D0]_{low byte} = 60₁₆, then after execution of this ADDQ, low byte of register D0 will contain 64₁₆.
- For ADD or SUB, if the destination is Dn, then the source (EA) can use all modes; if the destination is a memory location (all modes except Dn, An, relative, and immediate), then the source must be Dn.
- All subtraction instructions subtract source from destination. For example, consider SUB.W D0, \$500000. If [D0]_{low word} = 3003₁₆ and [500000]₁₆ = 5005₁₆, then after execution of this SUB instruction, memory location 500000₁₆ will contain 2002₁₆.
- Consider SUBI.W #5004, D0. If [D0]_{low word} = 5004₁₆, then after execution of this SUBI instruction, D0 will contain 5000₁₆. Note that the same result can be obtained by using a SUBQ.W #4, D0. However, in this case the data 4 is inherent in the instruction word.

5.5.2.b Multiplication and Division Instructions

The 68000 instruction set includes both signed and unsigned multiplication of integer numbers.

TABLE 5.6 68000 Arithmetic Instructions

Instruction	Size	Operation
Addition and Subtraction Instructions		
ADD (EA), (EA)	B, W, L	$[EA] + [EA] \rightarrow EA$
ADDI #data, (EA)	B, W, L	$[EA] + \text{data} \rightarrow EA$
ADDQ #d8, (EA)	B, W, L	$[EA] + d8 \rightarrow EA$ d8 can be an integer from 0 to 7
ADDA (EA), An	W, L	$An + [EA] \rightarrow An$
SUB (EA), (EA)	B, W, L	$[En]_{\text{source}} - [EA]_{\text{dest}} \rightarrow EA_{\text{dest}}$
SUBI #data, (EA)	B, W, L	$[EA] - \text{data} \rightarrow EA$
SUBQ #d8, (EA)	B, W, L	$[EA] - d8 \rightarrow EA$ d8 can be an integer from 0 to 7
SUBA (EA), An	W, L	$An - [EA] \rightarrow An$
Multiplication and Division Instructions		
MULS (EA), Dn	W	$[Dn]_{16} * [EA]_{16} \rightarrow [Dn]_{32}$ (signed multiplication)
MULU (EA), Dn	W	$[Dn]_{16} * [EA]_{16} \rightarrow [Dn]_{32}$ (unsigned multiplication)
DIVS (EA), Dn	W	$[Dn]_{32} / [EA]_{16} \rightarrow [Dn]_{32}$ (signed division, high word of Dn contains remainder and low word of Dn contains the quotient)
DIVU (EA), Dn	W	$[Dn]_{32} / [EA]_{16} \rightarrow [Dn]_{32}$ (unsigned division, remainder is in high word of Dn and quotient is in low word of Dn)
Compare, Clear, and Negate Instructions		
CMP (EA), Dn	B, W, L	$Dn - [EA] \rightarrow$ No result; affects flags
CMPL (EA), An	W, L	$An - [EA] \rightarrow$ No result; affects flags
CMPI #data, (EA)	B, W, L	$[EA] - \text{data} \rightarrow$ No result; affects flags
CMPL (Ay) +, (Ax) +	B, W, L	$[Ax] + - [Ay] + \rightarrow$ No result; affects flags; Ax and Ay are incremented depending on operand size
CLR (EA)	B, W, L	$0 \rightarrow EA$
NEG (EA)	B, W, L	$0 - [EA] \rightarrow EA$
Extended Arithmetic Instructions		
ADDX Dy, Dx	B, W, L	$Dx + Dy + X \rightarrow Dx$
ADDX - (Ay), - (Ax)	B, W, L	$-[Ax] + -[Ay] + X \rightarrow [Ax]$
EXT Dn	W, L	If size is W, then sign-extended low byte of Dn to 16 bits; if size is L, then sign extend low 16 bits of Dn to 32 bits
NEGX (EA)	B, W, L	$0 - [EA] - X \rightarrow EA$
SUBX Dy, Dx	B, W, L	$Dx - Dy - X \rightarrow Dx$
SUBX - (Ay), - (Ax)	B, W, L	$-[Ax] - -[Ay] - X \rightarrow [Ax]$
Test Instruction		
TST (EA)	B, W, L	$[EA] - 0 \rightarrow$ flags affected
Test and Set Instruction		
TAS (EA)	B	If $[EA] = 0$, then set $Z = 1$; else $Z = 0$, $N = 1$ and then always set bit 7 of $[EA]$ to 1

Note: If [EA] in the ADD or SUB instruction is an address register, the operand length is WORD or LONG WORD. [EA] in any instruction is calculated using the addressing mode used. All instructions except ADDA and SUBA affect condition codes.

- Source [EA] in the above ADDA and SUBA can use all modes.
- Destination [EA] in ADDI and SUBI can use all modes except An relative, and immediate.
- Destination [EA] in ADDQ and SUBQ can use all modes except relative and immediate.
- [EA] in all multiplication and division instructions can use all modes except An.
- Source [EA] in CMP and CMPL instructions can use all modes.
- Destination [EA] in CMPI can use all modes except An, relative, and immediate.
- [EA] in CLR and NEG can use all modes except An, relative, and immediate.
- [EA] in NEGX can use all modes except An, relative, and immediate.
- [EA] in TST can use all modes except An, relative, and immediate.
- [EA] in TAS can use all modes except An relative, and immediate.

- MULS (EA), Dn multiplies two 16-bit signed numbers and provides a 32-bit result. For example, consider MULS #-2, D0. If $[D0] = 0004_{16}$, then after this MULS, D0 will contain the 32-bit result $FFFFFFF8_{16}$ which is -8 in decimal. MULU (EA), Dn, on the other hand, performs unsigned multiplication. Consider MULU (A1), D2. If $[A1] = 00002000_{16}$, $[002000] = 0400_{16}$, and $[D2] = 0300_{16}$, then after this MULU, D2 will contain 32-bit result $000C0000_{16}$.
- Consider DIVS #4, D0. If $[D0] = -9_{10} = FFFFFFF7_{16}$, then after this DIVS, register D0 will contain

D0	FFFF	FFFE
	16-bit remainder = -1_{10}	16-bit quotient = -2_{10}

Note that in 68000, after DIVS, the sign of the remainder is always the same as the dividend unless the remainder is equal to zero. Therefore, in the example above, since the dividend is negative (-9_{10}), the remainder is negative (-1_{10}). Also, division by zero causes an internal interrupt automatically. A service routine can be written by the user to indicate an error. N = 1 if the quotient is negative, and V = 1 if there is an overflow. The DIVU instruction is the same as the DIVS instruction except that the division is unsigned. For example, consider DIVU #2, D0. If $[D0] = 11_{10} = 0000000B_{16}$, then after execution of this DIVU, register D0 contains:

D	0001	0005
	16-bit remainder	16-bit quotient

As with the DIVS, division by zero using DIVU causes trap. Also, V = 1 if there is an overflow.

5.5.2.c Compare, Clear, and Negate Instructions

- The COMPARE instructions affect condition codes, but do not provide the subtraction result. Consider CMPM.W (A1)+, (A2)+. If $[A1] = 00400000_{16}$, $[A2] = 00500000_{16}$, $[400000] = 0008_{16}$, $[500000] = 2006_{16}$, then after this CMP instruction N = 0, C = 0, X = 0, V = 0, Z = 0, $[A1] = 00400002_{16}$, and $[A2] = 00500002_{16}$.
- CLR.L D3 clears all 32 bits of D3 to zero.
- Consider NEG.W (A1). If $[A1] = 00500000_{16}$, $[500000_{16}] = 0007_{16}$, then after this NEG instruction, the low 16 bits of location 500000_{16} will contain $FFF9_{16}$.

5.5.2.d Extended Arithmetic Instructions

- ADDX and SUBX instructions can be used in performing multiple precision arithmetic since there are no ADDC (add with carry) or SUBC (subtract with borrow) instructions. For example, in order to perform a 64-bit addition, the following two instructions can be used:

```
ADD.L D2, D3 ; Add low 32 bits of data and store in D3
ADDX.L D4, D5 ; Add high 32 bits of data along with any
               ; carry from the low 32-bit addition and
               ; store result in D5
```

Note that in the above D5 D3 contains one 32-bit data and D4 D2 contains the other 32-bit data. The 32-bit result is stored in D5 D3.

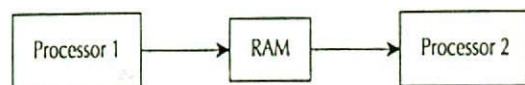


FIGURE 5.6 Two 68000s interfaced via shared RAM.

- Consider EXT.W D5. If [D5] low byte = $F8_{16}$, then after the EXT, [D5] = $FFF8_{16}$.

5.5.2.c Test Instruction

- Consider TST.W (A1). If [A1] = 00700000_{16} , $[700000] = F000_{16}$, then after the TST.W (A1), the operation $F000_{16}-0000_{16}$ is performed internally by the 68000, Z is cleared to zero, and N is set to 1. V and C flags are always cleared to zero.

5.5.2.f Test and Set Instruction

A multiprocessor system includes a mechanism to safely access resources such as memory shared by two or more processors. This mechanism is called mutual exclusion.

Mutual exclusion allows a processor to lock out other processors from accessing a shared resource. This shared resource is also called a Critical Program Section, since once a processor starts executing the program sequence in this section, it must complete it before another processor accesses it.

A binary flag called semaphore stored in the shared memory is used to indicate whether the shared memory is free (semaphore = 0) to be accessed or busy (semaphore = 1; the shared memory being used by another processor). Testing and setting the semaphore is a critical operation and must be executed in an indivisible cycle by a processor so that the other processors will not access the semaphore simultaneously.

The test and set instruction, along with a hardware lock mechanism, can initialize a semaphore. The test and set instruction tests and sets a semaphore. The processor, when executing this instruction, generates a signal to provide the lock mechanism during the execution time of the instruction. This prevents other processors from altering the semaphore between the processor's testing and setting the semaphore.

The 68000 provides the TAS instruction to test and set a semaphore. During execution of the TAS, the 68000 generates LOW on the AS (address strobe) pin which can be used to lock out other processors from accessing the semaphore.

Let us explain the application of the TAS instruction. TAS(EA) is usually used to synchronize two processors in multiprocessor data transfers. For example, consider two 68000-based microcomputers with shared RAM shown in Figure 5.6.

Suppose that it is desired to transfer the low byte of D0 from processor 1 to the low byte of D2 in processor 2. A memory location, namely, TRDATA, can be used to accomplish this. First, processor 1 can execute the TAS instruction to test the byte in the shared RAM with address TEST for zero value. If it is zero, the processor 1 can be programmed to move the low byte of D0 into location TRDATA in the shared RAM. The processor 2 can then execute an instruction sequence to move the contents of TRDATA from the shared RAM into the low byte of D2. The following instruction sequence will accomplish this:

Processor 1 Routine		Processor 2 Routine	
Proc 1	TAS TEST	Proc 2	TAS TEST
	BNE Proc 1		BNE Proc 2
	MOVE.B D0, TRDATA		MOVE.B TRDATA, D2
	CLR.B TEST		CLR.B TEST
	—		—
	—		—
	—		—

TABLE 5.7 68000 Logical Instructions

Instruction	Size	Operation
AND (EA), (EA)	B, W, L	$[EA] \wedge [EA] \rightarrow EA$
ANDI #data, (EA)	B, W, L	$[EA] \wedge \#data \rightarrow EA$; [EA] cannot be address register
ANDI #data8, CCR	B	$CCR \wedge \#data \rightarrow CCR$
ANDI #data16, SR	W	$SR \wedge \#data \rightarrow SR$
EOR Dn, (EA)	B, W, L	$Dn \oplus [EA] \rightarrow EA$; [EA] cannot be address register
EORI #data, (EA)	B, W, L	$[EA] \oplus \#data \rightarrow EA$; [EA] cannot be address register
EORI #data16, SR	W	$Data16 \oplus SR \rightarrow SR$ if $s = 1$; else trap
NOT (EA)	B, W, L	One's complement of [EA] $\rightarrow EA$
OR (EA), (EA)	B, W, L	$[EA] \vee [EA] \rightarrow EA$
ORI #data, (EA)	B, W, L	$[EA] \vee \#data \rightarrow EA$; [EA] cannot be address register
ORI #data8, CCR	B	$CCR \vee \#data8 \rightarrow CCR$
ORI #data16, SR	W	$SR \vee \#data \rightarrow SR$

- Source [EA] in AND and OR can use all modes except An.
- Destination [EA] in AND or OR or EOR can use all modes except Dn, An, relative, and immediate.
- Destination [EA] in ANDI, ORI, and EORI can use all modes except An, relative, and immediate.
- [EA] in NOT can use all modes except An, relative, and immediate.

Note that in the above, TAS TEST checks the byte addressed by TEST for zero. If [TEST] = 0, then Z is set to one; otherwise Z = 0 and N = 1. After this, bit 7 of [TEST] is set to 1. Note that a zero value of TEST indicates that the shared RAM is free for use and the Z bit indicates this after the TAS is executed. In each of the above instruction sequences, data is transferred using the MOVE instruction, TEST is cleared to zero so that the shared RAM is free for use by the other processor. Note that bit 7 of memory location TEST is called the semaphore.

In order to avoid testing of the TEST byte simultaneously by two processors, the TAS is executed in a Read-Modify-Write cycle. This means that once the operand is addressed by the 68000 executing the TAS, the system bus is not available to the other 68000 until the TAS is completed.

5.5.3 Logical Instructions

These instructions include logical OR, EOR, AND, and NOT as shown in Table 5.7.

- Consider AND.W D2, D6. If $[D2] = 0005_{16}$, $[D6] = 0FF1_{16}$, then after execution of this AND, the low 16 bits of D6 will contain 0001_{16} .
- Consider ANDI.B #\$80, CCR. If $[CCR] = 0F_{16}$, then after this ANDI, register CCR will contain 00_{16} .
- Consider EOR.W D2, D6. If $[D2] = 000F_{16}$ and $[D6] = 000F_{16}$, then after execution of this EOR, register D6 will contain 0000_{16} , and D2 will remain unchanged at $000F_{16}$.
- Consider NOT.B D0. If $[D0] = 04_{16}$, then after execution of this NOT instruction, the low byte of D0 will contain FB_{16} .
- Consider ORI #\$1008, SR. If $[SR] = A011_{16}$, then after execution of this ORI, register SR will contain $B019_{16}$. Note that this is a privileged instruction since the high byte of SR containing the control bits is changed and therefore can only be executed in the supervisor mode.

5.5.4 Shift and Rotate Instructions

The 68000 shift and rotate instructions are listed in Table 5.8.

TABLE 5.8 68000 Shift and Rotate Instructions

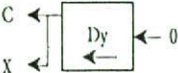
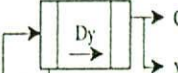
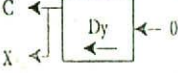
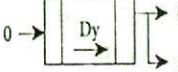
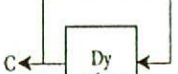
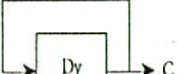
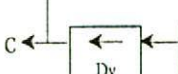
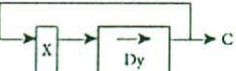
Instruction	Size	Operation
ASL, Dx, Dy	B, W, L	 <p>Shift [Dy] by the number of times to left specified in Dx; the low 6 bits of Dx specify the number of shifts from 0 to 63</p>
ASL #data, Dn	B, W, L	Same as ASL Dx, Dy except that the number of shifts is specified by immediate data from 0 to 7
ASL (EA)	B, W, L	[EA] is shifted one bit to left; the most significant of [EA] goes to x and c, and zero moves into the least significant bit
ASR Dx, Dy	B, W, L	 <p>Arithmetically shift [Dy] to the right by retaining the sign bit; the low 6 bits of Dx specify the number of shifts from 0 to 63</p>
ASR #data, Dn	B, W, L	Same as above except the number of shifts is from 0 to 7
ASR (EA)	B, W, L	Same as above except [EA] is shifted once to the right
LSL Dx, Dy	B, W, L	 <p>Low 6 bits of Dx specify the number of shifts from 0 to 63</p>
LSL #data, Dn	B, W, L	Same as above except the number of shift is specified by immediate data from 0 to 7
LSL (EA)	B, W, L	[EA] is shifted one bit to the left
LSR Dx, Dy	B, W, L	 <p>Same as LSL Dx, Dy except shift is to the right</p>
LSR #data, Dn	B, W, L	Same as LSL #data, Dn, except shift is to the right by immediate data from 0 to 7
LSR (EA)	B, W, L	Same as LSL [EA] except shift is to the right
ROL Dx, Dy	B, W, L	 <p>Low 6 bits of Dx specify the number of times [Dy] to be shifted</p>
ROL #data, Dn	B, W, L	Same as above except that the immediate data specify that [Dn] to be shifted from 0 to 7
ROL (EA)	B, W, L	[EA] is rotated once to the left
ROR Dx, Dy	B, W, L	 <p>Low 6 bits of Dx specify the number of shifts from 0 to 63</p>
ROR #data, Dn	B, W, L	Same as ROL #data, Dn except the shift is to the right by immediate data from 0 to 7
ROR (EA)	B, W, L	[EA] is rotated once to the right
ROXL Dx, Dy	B, W, L	 <p>Low 6 bits of Dx contain the number of rotates from 0 to 63</p>
ROXL #data, Dy	B, W, L	Same as above except immediate data specify number of rotates from 0 to 7
ROXL (EA)	B, W, L	[EA] is rotated one bit to right

TABLE 5.8 68000 Shift and Rotate Instructions (*continued*)

Instruction	Size	Operation
ROXR Dx, Dy	B, W, L	
ROXR #data, Dy	B, W, L	Same as ROXL Dx, Dy except the rotate is to the right Same as ROXL #data, Dy, except rotate is to the right by immediate data from 0 to 7
ROXR (EA)	B, W, L	Same as ROXL [EA] except rotate is to the right

Note: [EA] in ASL, ASR, LSL, LSR, ROL, ROR, ROXL, and ROXR can use all modes except Dn, An, relative, and immediate.

- All the instructions in Table 5.8 affect N and Z flags according to the result. V is reset to zero except for ASL.
- Note that in the 68000 there is no true arithmetic shift left instruction. In true arithmetic shifts, the sign bit of the number being shifted is retained. In the 68000, the instruction ASL does not retain the sign bit, whereas the instruction ASR retains the sign bit after performing the arithmetic shift operation. Consider ASL.W D3, D0. If $[D3]_{\text{low } 16 \text{ bits}} = 0003_{16}$, $[D0]_{\text{low } 16 \text{ bits}} = 87\text{EF}_{16}$, then after this ASL instruction $[D0]_{\text{low } 16 \text{ bits}} = 3\text{FF}8_{16}$, $C = 0$, and $X = 0$. Note that the sign of the contents of D0 is changed from 1 to 0, and therefore the overflow is set. ASL sets the overflow bit to indicate sign change during the shift. In the example, the sign bit of D0 is changed after shifting [D0] three times. ASR, on the other hand, retains the sign bit. For example, consider ASR.W #2, D5. If $[D5] = 8\text{FE}2_{16}$, then after this ASR, the low 16 bits of $[D5] = \text{E}3\text{F}8_{16}$, $C = 1$, and $X = 1$. Note that the sign bit is retained. ASL (EA) or ASR (EA) shifts (EA) one bit to the left or right, respectively. For example, consider ASL.W (A2). If $[A2] = 00004000_{16}$ and $[004000] = \text{F}000_{16}$, then after execution of the ASL, $[004000] = \text{E}000_{16}$, $X = 1$, and $C = 1$. On the other hand, after ASR.W (A2), memory location 004000_{16} will contain 7800_{16} , $C = 0$, and $X = 0$. Note that only memory-alterable modes are allowed for (EA). Also, only 16-bit operands are allowed for (EA) when the destination is memory location.
- LSL and ASL instructions are the same in the 68000 except that with the ASL, V is set to 1 if there is a sign change of the number during the shift.
- Consider LSR.W# 0002, D0. If $[D0] = \text{F}000_{16}$, then after the LSR, $[D0] = 3\text{C}00_{16}$, $X = 0$, and $C = 0$.
- Consider ROL.B# 02, D2. If $[D2] = \text{B}1_{16}$ and $C = 1$, then after execution of the ROL, the low byte of $[D2] = \text{C}7_{16}$ and $C = 0$. On the other hand, with $[D2] = \text{B}1_{16}$ and $C = 1$, consider ROR.B #02, D2. After execution of this ROR, register D2 will contain EC_{16} and $C = 0$.
- Consider ROXL.W D2, D1. If $[D2] = 0003_{16}$, $[D1] = \text{F}201_{16}$, $C = 0$, and $X = 1$, then the low 16 bits after execution of this ROXL are $[D1] = 900\text{F}_{16}$, $C = 1$, and $X = 1$.

5.5.5 Bit Manipulation Instructions

The 68000 has four bit manipulation instructions, and these are listed in Table 5.9.

- In all the above instructions, the 1's complement of the specified bit is reflected in the Z flag. The specified bit is then 1's complemented, cleared to zero, set to one, or unchanged by BCHG, BCLR, BSET, or BTST, respectively. In all the instructions in Table 5.10, if (EA) is Dn, then length of Dn is 32 bits; otherwise, the length of the destination is one byte.

TABLE 5.9 68000 Bit Manipulation Instructions

Instruction	Size	Operation
BCHG Dn, (EA)	B,L	A bit in [EA] specified by Dn or immediate data is tested; the 1's complement of the bit is reflected in both the Z flag and the specified bit position
BCHG #data, (EA)		
BCLR Dn, (EA)	B,L	A bit in [EA] specified by Dn or immediate data is tested and the 1's complement of the bit is reflected in the Z flag; the specified bit is cleared to zero
BCLR #data, (EA)		
BSET Dn, (EA)	B,L	A bit in [EA] specified by Dn or immediate data is tested and the 1's complement of the bit is reflected in the Z flag; the specified bit is then set to one
BSET #data, (EA)		
BTST Dn, (EA)	B,L	A bit in [EA] specified by Dn or immediate data is tested; the 1's complement of the specified bit is reflected in the Z flag
BTST #data, (EA)		

- [EA] in the above instructions can use all modes except An, relative, and immediate.
- If [EA] is memory location, then data size is byte; if [EA] is Dn then data size is long word.

- Consider BCHG.B #2, \$003000. If [003000] = 05₁₆, then after execution of this BCHG instruction, Z = 0 and [003000] = 01₁₆.
- Consider BCLR.L #3, D1. If [D1] = F210E128₁₆, then after execution of this BCLR, register D1 will contain F210E120₁₆ and Z = 0.
- Consider BSET.B #0, (A1). If [A1] = 00003000₁₆, [003000] = 00₁₆, then after execution of this BSET, memory location 003000 will contain 01₁₆ and Z = 1.
- Consider BTST.B #2, \$002000. If [002000] = 02₁₆, then after execution of this BTST, Z = 0 and [002000] = 02₁₆.

5.5.6 Binary-Coded Decimal Instructions

The 68000 instruction set contains three BCD instructions, namely, ABCD for adding, SBCD for subtracting, and NBCD for negating. These instructions operate on packed BCD operands and always include the extent (X) bit in the operation. The BCD instructions are listed in Table 5.10.

- Consider ABCD D1, D2. If [D1] = 25₁₆, [D2] = 15₁₆, X = 0, then after execution of the ABCD instruction, [D2] = 40₁₆, X = 0, and Z = 0.
- Consider SBCD - (A2), - (A3). If [A2] = 00002004₁₆, [A3] = 00003003₁₆, X = 1, [002003] = 05₁₆, [003002] = 06₁₆, then after execution of this SBCD, [003002] = 00₁₆, X = 0, and Z = 1.
- Consider NBCD (A1). If [A1] = [00003000₁₆], [003000] = 05, X = 1, then after execution of the NBCD, [003000] = FA₁₆.

5.5.7 Program Control Instructions

These instructions include branches, jumps, and subroutine calls as listed in Table 5.11.

TABLE 5.10 68000 Binary-Coded Decimal Instructions

Instruction	Operand size	Operation
ABCD Dy, Dx	B	[Dx]10 + [Dy]10 + X → Dx
ABCD - (Ay), - (Ax)	B	-[Ax]10 + -[Ay]10 + X → [Ax]
SBCD Dy, Dx	B	[Dx]10 - [Dy]10 - X → Dx
SBCD - (Ay), - (Ax)	B	-[Ax]10 - -[Ay]10 - X → [Ax]
NBCD (EA)	B	0 - [EA]10 - X → EA

Note: [EA] in NBCD can use all modes except An, relative, and immediate.

TABLE 5.11 68000 Program Control Instructions

Instruction	Size	Operation
Bcc d	B,W	If condition code cc is true, then $PC + d \rightarrow PC$; the PC value is current instruction location plus 2; d can be 8- or 16-bit signed displacement; if 8-bit displacement is used, then the instruction size is 16 bits with the 8-bit displacement as the low byte of the instruction word; if 16-bit displacement is used, then the instruction size is two words with 8-bit displacement field (low byte) in the instruction word as zero and the second word following the instruction word as the 16-bit displacement There are 14 conditions such as BCC (Branch if Carry Clear), BEQ (Branch if result equal to zero, i.e., $Z = 1$), and BNE (Branch if not equal, i.e., $Z = 0$) Note that the PC contents will always be even since the instruction length is either one word or two words depending on the displacement widths
BRA d	B,W	Branch always to $PC + d$ where PC value is current instruction location plus 2; as with Bcc, d can be signed 8 or 16 bits; this is an unconditional branching instruction with relative mode; note that the PC contents are even since the instruction is either one word or two words
BSR d	B,W	$PC \rightarrow [SP]$ $PC + d \rightarrow PC$ The address of the next instruction following PC is pushed onto the stack; PC is then loaded with $PC + d$; as before, d can be 8 or 16 bits; this is a subroutine call instruction using relative mode
DBcc Dn, d	W	If cc is false, then $Dn - 1 \rightarrow Dn$, and if $Dn = -1$, then $PC + 2 \rightarrow PC$; If $Dn \neq -1$, then $PC + d \rightarrow PC$; if cc is true, $PC + 2 \rightarrow PC$
JMP (EA)	Unsize	[EA] $\rightarrow PC$ This is an unconditional jump instruction which uses control addressing mode
JSR (EA)	Unsize	$PC \rightarrow [SP]$ [EA] $\rightarrow PC$ This is a subroutine call instruction which uses control addressing mode
RTR	Unsize	[SP] $\rightarrow CCR$ [SP] $\rightarrow PC$ Return and restore condition codes
RTS	Unsize	Return from subroutine [SP] $\rightarrow PC$
Sec (EA)	B	If cc is true, then the byte specified by [EA] is set to all ones, otherwise the byte is cleared to zero

- [EA] in JMP and JSR can use all modes except Dn, An, (An), $-(An)$, and immediate.
- [EA] in Sec can use all modes except An, relative, and immediate.

- Consider Bcc d. There are 14 branch conditions. This means that cc in Bcc can be replaced by 14 conditions providing 14 instructions. These are BCC, BCS, BEQ, BGE, BGT, BHI, BLE, BLS, BLT, BMI, BNE, BPL, BVC, and BVS. It should be mentioned that some of these instructions are applicable to both signed and unsigned numbers, some can be used with only signed numbers, and some instructions are applicable to only unsigned numbers as shown below:

For both signed and unsigned numbers	For signed numbers	For unsigned numbers
BCC d (Branch if C = 0)	BGE d (Branch if greater or equal)	BHI d (Branch if high)
BCS d (Branch if C = 1)	BGT d (Branch if greater than)	BLS d (Branch if low or same)
BEQ d (Branch if Z = 1)	BLE d (Branch if less than or equal)	
BNE d (Branch if Z = 0)	BLT d (Branch if less than)	
	BMI d (Branch if N = 1)	
	BPL d (Branch if N = 0)	
	BVC d (Branch if V = 0)	
	BVS d (Branch if V = 1)	

After signed arithmetic operations such as ADD or SUB the instructions such as BEQ, BNE, BVS, BVC, BMI, and BPL can be used. On the other hand, after unsigned arithmetic operations, the instructions such as BCC, BCS, BEQ, and BNE can be used.

If $V = 0$, BPL and BGE have the same meaning. Likewise, if $V = 0$, BMI and BLT perform the same function.

The conditional branch instructions can be used after typical arithmetic instructions such as subtraction to branch to a location if cc is true. For example, consider SUB.W D1, D2. Now if [D1] and [D2] are unsigned numbers, then

BCC d can be used if $[D2] > [D1]$
 BCS d can be used if $[D2] < [D1]$
 BEQ d can be used if $[D2] = [D1]$
 BNE d can be used if $[D2] \neq [D1]$
 BHI d can be used if $[D2] > [D1]$
 BLS d can be used if $[D2] \leq [D1]$

On the other hand, if [D1] and [D2] are signed numbers, then after SUB.W D1, D2, the following branch instructions can be used:

BEQ d can be used if $[D2] = [D1]$
 BNE d can be used if $[D2] \neq [D1]$
 BLT d can be used if $[D2] < [D1]$
 BLE d can be used if $[D2] \leq [D1]$
 BGT d can be used if $[D2] > [D1]$
 BGE d can be used if $[D2] \geq [D1]$

Now, as an example consider $\text{BEQ}^* + \$20$. If $[PC] = 000200_{16}$, then after execution of the BEQ instruction, program execution starts at 000220_{16} if $Z = 1$; if $Z = 0$, program execution continues at 000200 . Note that * is used by some assemblers to indicate displacement.

- The instruction BRA and JMP are unconditional JUMP instructions. The BRA (Branch Always) instruction uses the relative addressing mode, whereas the JMP uses only control addressing modes. For example, consider $\text{BRA}^* + \$20$. If $[PC] = 000200_{16}$, then after execution of the BRA, program execution starts at 000220_{16} .

Now consider $\text{JMP} (A1)$. If $[A1] = 00000220_{16}$, then after execution of the JMP, program execution starts at 000220_{16} .

- The instructions BSR and JSR are subroutine CALL instructions. BSR uses relative mode, whereas JSR uses absolute addressing mode. Consider the following program segment:

Main program		Subroutine	
	—	PROC	MOVEM.L D0-D7/A0-A6, -(SP)
	—	—	} main body of the subroutine
	—	—	
START	JSR PROC	—	
	—		MOVEM.L (SP)+, D0-D7/A0-A6
	—		RTS

In the above, JSR SUB instruction calls the subroutine called PROC. In response to JSR, the 68000 pushes the current PC contents called START onto the stack and loads the

starting address PROC of the subroutine into PC. The first MOVEM in the PROC pushes all registers onto the stack and after the subroutine is executed, the second MOVEM instruction pops all the registers back. Finally, RTS pops the address START from the stack into PC, and the program control is returned to the main program. Note that BSR PROC could have been used instead of JSR PROC in the main program. In that case, the 68000 assembler would have considered the PROC with BSR as a displacement rather than as an address with the JSR instruction.

The use of stack to save temporary variables during subroutine execution can be facilitated by utilizing a stack frame. The stack frame is a set of locations in stack used for saving return addresses, local variables, and I/O parameters. Local variables such as loop counters used in the subroutine are used by the subroutine and are not returned to the main program. The stack frame can be utilized by the subroutine to access a new set of parameters each time the main program calls it.

In 68000, the stack frame can be created by the main program and the subroutine using the LINK and UNLK instructions. The subroutine can access the variables on the stack by using displacements from a base register called a frame pointer. The 68000 LINK instruction can be used to create a stack frame and define the frame pointer.

As an example of LINK/UNLK, the instruction LINK A5, #-100 creates a stack frame of 100₁₀ with A5 as the frame pointer. The instruction such as MOVE.W d(A5), D1 and MOVE.W D1,d(A5) can then be used to access the stack frame.

A typical instruction sequence illustrating the use of LINK and UNLK is given below:

Main Program

```

MOVE.L #Const, -(USP)      ; Push 32-bit
                             ; constant to be passed
PEA.W ADDR                 ; Push starting address of
                             ; a table to be passed
JSR START                  ; Jump to the subroutine
                             START
-
-
-

```

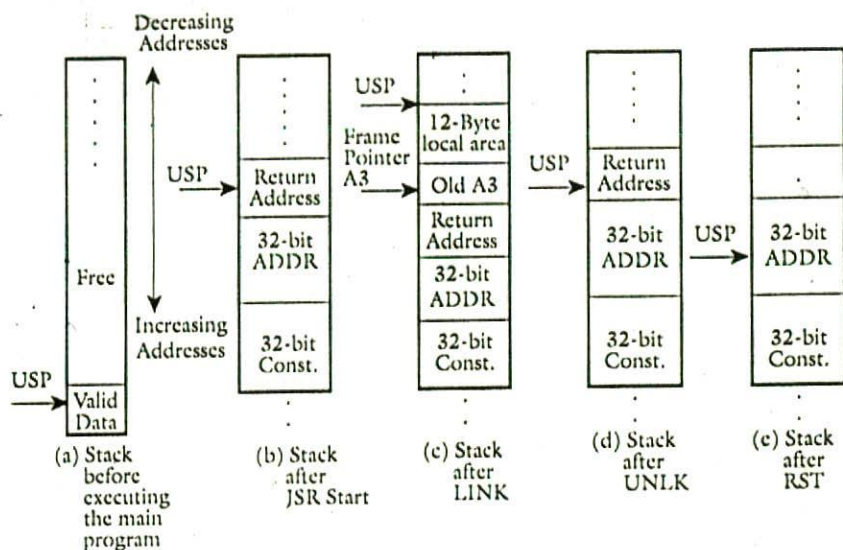
Subroutine

```

START      LINK A3, #-12      ; Allocate 12 bytes
-
-
-
MOVE.L DATA1, -4(A3),      ; Push local variable 1
MOVE.L DATA2, -8(A3),      ; Push local variable 2
MOVE.L DATA3, -12(A3)      ; Push local variable 3
Instructions { MOVEA.L 8(A3), A4      ; Obtain ADDR
arbitrarily  { MOVE.L (A4), D1        ; Read data from table
chosen      { SUBQ.L #5, -12(A3)      ; Subtract 5 from local
           {                       ; variable 3
           { MOVE.L 12(A3), D4        ; Read the 32-bit constant
-
-
-
UNLK A3      ; Restore original values
RTS          ; RETURN

```

The following illustrates the stack contents at various points:



In the main program, it is assumed that the 68000 is in user mode and, therefore, USP is used in the program. It is also assumed that the main program passes to the subroutine a 32-bit constant, and the starting address ADDR of a table to be accessed by the subroutine. The subroutine allocates 12 bytes to save three 32-bit local variables by using the LINK instruction. Several instructions are arbitrarily chosen (to illustrate the concept) to be executed by the subroutine to read data from the table, decrement the local variable 3 by 5, and read the 32-bit constant so that they can be used in the instruction sequence that follows (not shown in the subroutine). The UNLK instruction restores the original conditions and the RTS allows the program to return to the right place in the program.

- DBcc Dn, d tests both the condition codes and the value in a data register. DBcc first checks if cc (NE, EQ, GT, etc.) is satisfied. If satisfied, the next instruction is executed. If cc is not satisfied, the specified data register is decremented by 1. If [Dn] = -1, then next instruction is executed; if Dn \neq -1, then a branch is taken to PC + d. For example, consider DBNE D5, * -4 and [D5] = 00003002₁₆, [PC] = 002006₁₆. Now if Z = 1, then [D5] = 00003001₁₆; since [D5] \neq -1, program execution starts at 002002₁₆. There is a false condition in the DBcc instruction, and this instruction is the DBF (some assemblers use DBRA for this). In this case, the condition is always false. This means that after execution of this instruction, Dn is decremented by 1 and if [Dn] = -1, then the next instruction is executed; if [Dn] \neq -1, then branch to PC + d is taken.
- Consider SPL(A5). If [A5] = 00200020₁₆ and N = 0, then after execution of the SPL, memory location 200020₁₆ will contain 11111111₂.

5.5.8 System Control Instructions

The 68000 contains some system control instructions which include privileged instructions and trap instructions. Note that the privileged instructions can only be executed in the supervisor mode. The system control instructions are listed in Table 5.12.

- The RESET instruction, when executed in supervisor mode, outputs a signal on the RESET pin of the 68000 in order to initialize the external peripheral chips. The 68000 reset pin is bidirectional. The 68000 can be reset by asserting the reset pin using hardware, whereas the peripheral chips can be reset using the software RESET instruction.

TABLE 5.12 68000 System Control Instructions

Instruction	Size	Operation
Privileged Instructions		
RESET	Unsize	If supervisory state, then assert reset line; else TRAP
RTE	Unsize	If supervisory state, then restore SR and PC; else TRAP
STOP #data	Unsize	If supervisory state, then load immediate data to SR and then STOP; else TRAP
ORI to SR MOVE to/from USP ANDI to SR EORI to SR MOVE (EA) to/from SR	}	These privileged instructions were discussed earlier
Trap and Check Instructions		
TRAP # vector	Unsize	PC \rightarrow - [SSP] SR \rightarrow - [SSP] Vector address \rightarrow PC
TRAPV	Unsize	TRAP if V = 1
CHK (EA), Dn	W	If Dn < 0 or Dn > [EA], then TRAP
Condition Code Register		
ANDI to CCR EORI to CCR MOVE (EA) to/from CCR ORI to CCR	}	Already explained earlier

Note: (EA) in CHK can use all modes except An.

- MOVE.L USP, An or MOVE.L An, USP can be used to save, restore, or change the contents of USP in supervisor mode. The USP must be loaded in supervisor mode since MOVE USP is a privileged instruction.
- Consider TRAP # n. There are 16 TRAP instructions with n ranging from 0 to 15. The hexadecimal vector address is calculated using the following equation:

$$\text{Hexadecimal vector address} = 80 + 4 \cdot n$$

The TRAP instruction first pushes the contents of PC and then the SR onto the system (user or supervisor) stack. The hexadecimal vector address is then loaded into PC. The TRAP is basically a software interrupt.

One of the 16 trap instructions can be executed in the user mode to execute a supervisor program located at the specified trap routine. Using the TRAP instruction, control can be transferred to the supervisor mode from the user mode.

There are other traps which occur due to certain arithmetic errors. For example, division by zero automatically traps to location 14₁₆. On the other hand, an overflow condition, i.e., if V = 1, will trap to address 1C₁₆ if the instruction TRAPV is executed.

- The CHK (EA), Dn instruction compares [Dn] with (EA). If [Dn]_{low 16 bits} < 0 or if [Dn]_{low 16 bits} > (EA), then a trap to location 0018₁₆ is generated. Also, N is set to 1 if [Dn]_{low 16 bits} < 0 and N is reset to zero if [Dn]_{low 16 bits} > (EA). (EA) is treated as a 16-bit 2's complement integer. Note that program execution continues if [Dn]_{low 16 bits} lies between 0 and (EA).

Consider CHK (A5), D2. If [D2]_{low 16 bits} = 0200₁₆, [A5] = 00003000₁₆, [003000]₁₆ = 0100₁₆, then after execution of the CHK, the 68000 will trap since [D2] = 0200₁₆ is greater than [003000] = 0100₁₆.

The purpose of the CHK instruction is to provide boundary checking by testing if the content of a data register is in the range from zero to an upper limit. The upper limit used in the instruction can be set equal to the length of the array. Then every time the array is accessed, the CHK instruction can be executed to make sure that the array bounds have not been violated.

The CHK instruction is usually placed following the computation of an index value to ensure that the limits of this index value are not violated. This allows checking whether or not the address of an array being accessed is within the array boundaries when the address register indirect with index mode is used to access an element in the array. For example, the instruction sequence shown below will allow accessing of an array with base address in A3 and array length of 125₁₀ bytes:

```

—
—
—
CHK #124, D5
MOVE.B 0 (A3, D5.W), D4
—
—
—
—
—

```

In the above, if low 16 bits of D5 is greater than 124, the 68000 will trap to location 0018₁₆.

In the above, it is assumed that D5 is computed prior to execution of the CHK instruction. Also, the 68000 assembler requires that a displacement value of 0 be specified as in the instruction MOVE.B 0(A3, D5.W), D4.

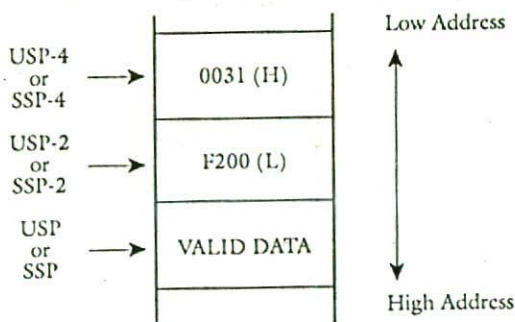
5.6 68000 Stacks

The 68000 supports stacks and queues with the address register indirect postincrement and predecrement addressing modes.

In addition to SPs, all seven address registers A0-A6 can be used as stack pointers by using appropriate addressing modes.

Subroutine calls, traps, and interrupts automatically use the system stack pointers: USP when S = 0 and SSP when S = 1. Subroutine calls push PC onto the stack, while RTS pops PC from the system stack. Traps and interrupt push both PC and SR onto the system, while RTE pops PC and SR from the system stack.

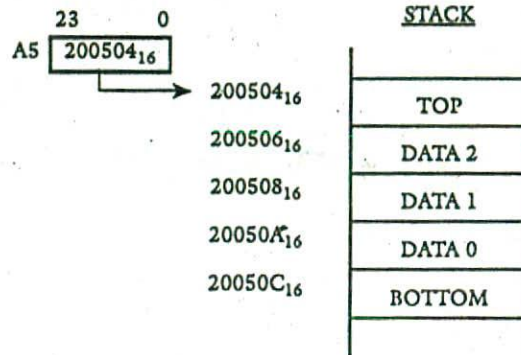
These stack operations fill data from high memory to low memory. This means that the system SP is predecremented by 2 (word) or 4 (long word) with push and post incremented by 2 (for word) or 4 (for long word) after pop. As an example, suppose that a 68000 call instruction (JSR or BSR) is executed when PC = \$0031 F200; then after execution of the subroutine call the stack will push the PC as follows:



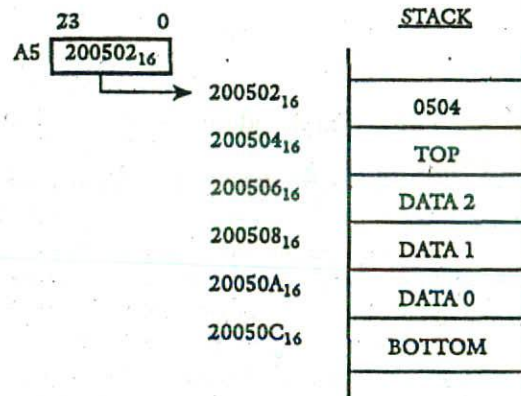
Note that the system stack pointer always points to valid data.

Stacks can be created by the user by using address register indirect with postincrement or predecrement modes. Using one of the seven (A0-A6) address registers, the user may create stacks which can be filled from either high memory to low memory or vice versa.

Stacks from high to low memory are implemented with predecrement mode for push and postincrement mode for pop. On the other hand, stacks from low to high memory is implemented with postincrement for push and predecrement for pop. For example, consider the following stack growing from high to low memory addresses in which A5 is used as the stack pointer:

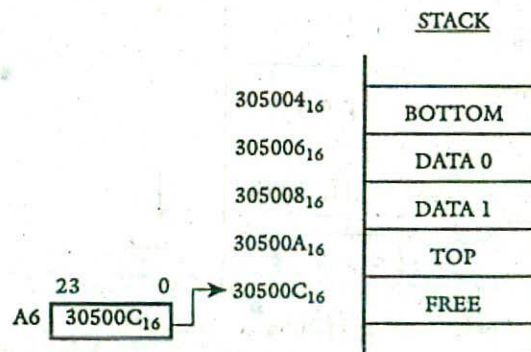


In order to push 16-bit content 0504₁₆ of memory location 305016₁₆, the instruction `MOVE.W $305016, -(A5)` can be used as follows:

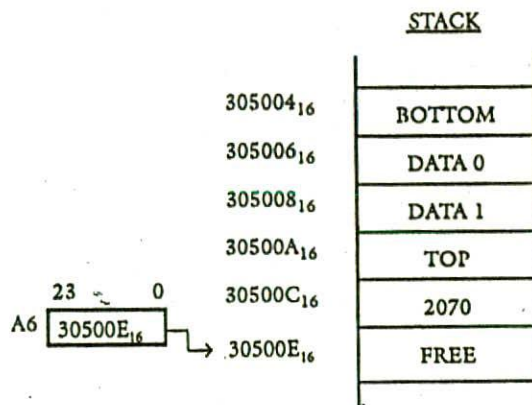


The 16-bit data 0504₁₆ can be popped from the stack into low 16 bits of D0 by using `MOVE.W (A5)+, D0`. A5 will contain 200504₁₆ after the POP. Note that in this case, the stack pointer A5 points to valid data.

Next, consider the stack growing from low to high memory addresses in which, say, A6 is used as the stack pointer:



Now, in order to PUSH 16-bit contents 2070₁₆ of the low 16 bits of D5, the instruction `MOVE.W D5, (A6)+` can be used as follows:



The 16-bit data 2070₁₆ can be popped from the stack into 16-bit memory location 417024₁₆ by using MOVE.W- (A6), \$417024.

Note that in this case, the stack pointer A6 points to the free location above the valid data.

Example 5.1

Determine the effect of each one of the following 68000 instructions:

- CLR D0
- MOVE.L D1, D0
- CLR.L (A0)+
- MOVE -(A0), D0
- MOVE 20(A0), D0
- MOVEQ.L #\$D7, D0
- MOVE 21(A1, A0.L), D0

Assume the following initial configuration before each instruction is executed. Also, assume all numbers in hex.

[D0] = 22224444, [D1] = 55556666
 [A0] = 00002224, [A1] = 00003333
 [002220] = 8888, [002222] = 7777
 [002224] = 6666, [002226] = 5555
 [002238] = AAAA, [00556C] = FFFF

Solution

Instruction	Effective address	Net effect (HEX)
CLR D0	Destination EA = D0	[D0] = 22220000
MOVE.L D1, D0	Destination EA = D0 Source EA = D1	[D0] ← 55556666
CLR.L (A0)+	Destination EA = [A0]	[002224] = 0000 [002226] = 0000 [A0] ← 00002228
MOVE -(A0), D0	Source EA = [A0] - 2 Destination EA = D0	[A0] ← 00002222 [D0] = 22227777
MOVE 20(A0), D0	Source EA = [A0] + 20 ₁₀ (or 14 ₁₆) = 002238 Destination EA = D0	[D0] = 2222AAAA
MOVEQ.L #\$D7, D0	Source data = D7 ₁₆ Destination EA = D0	[D0] ← FFFFFFFD7
MOVE 21(A0, A1.L), D0	Source EA = [A0] + [A1] + 21 ₁₀ = 556C Destination EA = D0	[D0] = 2222FFFF

Example 5.2

Write a 68000 assembly program segment that implements each one of the following Pascal segments. Assume the following information about the variables involved in this problem.

Variable	Comments
X	Address of a 16-bit signed integer of array 1 of 10 elements
Y	Address of a 16-bit signed integer of array 2 of 10 elements
SUM	Address of the sum
	(a) If $X \geq Y$, then $X := X + 10$; else $Y := Y - 12$
	(b) $\text{Sum} := 0$; for $i := 0$ to 9, do $\text{sum} := \text{sum} + A(i)$

Solution

```

(a)      LEA X, A0          ; Point A0 to X
        LEA Y, A1          ; Point A1 to Y
        MOVE (A0), D0      ; MOVE [X] into D0
        CMP (A1), D0       ; COMPARE [X] WITH [Y]
        BGE THRPT
        SUBI #12, (A1)      ; Execute else part
        BRA NEXT
THRPT    ADDI #10, (A0)      ; Execute then part
NEXT
        . . .

(b)      LEA SUM, A1        ; Point to SUM
        LEA Y, A0          ; Point A0 to Y [0]
        CLR D0             ; Clear the sum to zero
        MOVEQ.L #9, D1     ; Initialize D1 with loop
                           ; limit
LOOP     ADD (A0) +, D0     ; Perform the iterative
        DBF D1, LOOP       ; Summation
        MOVE D0, (A1)      ; Transfer the result
        . . .

```

Note that condition F in DBF is always false and thus we exit from the loop only when the content of the register D1 becomes -1. Therefore, we repeat the addition process for 10 times as desired.

Example 5.3

Write 68000 assembly program to clear 85_{16} consecutive bytes.

Solution

```

        ORG $2000
        MOVEA.L#$3000,A0   ; LOAD A0 WITH $3000
        MOVE #$84, D0      ; MOVE  $84_{16}$  INTO D0
LOOP     CLR.B(A0)+        ; CLEAR  $[3000_{16}]$  AND
                           ; POINT TO NEXT ADDRESS

```

```

        DBF D0, LOOP      ; DECREMENT AND BRANCH
FINISH  JMP FINISH        ; HALT

```

Note that the 68000 has no HALT instruction in the user mode. The 68000 has the STOP instruction in supervisor mode. Therefore, the unconditional JUMP to some location such as FINISH JMP FINISH in the above program must be used. Since DBF is a word instruction and considers D0's low 16-bit word as the loop count, D0 must be initialized by a word MOVE rather than byte MOVE, even though 84_{16} can be accommodated in a byte. Also, one should be careful about using MOVEQ, since MOVEQ sign extends a byte to a long word.

Example 5.4

Write 68000 assembly program to compute

$$\sum_{i=1}^N X_i Y_i$$

where X_i and Y_i are signed 16-bit numbers, $N = 100$. Assume no overflow and that the result is 32-bit wide.

Solution

```

        ORG $1000
        MOVEQ.L#99,D0      ; MOVE 9910 INTO D0
        LEA P, A0           ; LOAD ADDRESS P INTO A0
        LEA Q, A1           ; LOAD ADDRESS Q INTO A1
        CLR.L D1            ; Initialize D1 to zero
LOOP    MOVE (A0)+, D2      ; MOVE [X] TO D2
        MULS (A1)+, D2      ; D2 ← [X] * [Y]
        ADD.L D2, D1        ; D1 ←  $\sum X_i Y_i$ 
        DBF D0, LOOP        ; DECREMENT AND BRANCH
FINISH  JMP FINISH          ; HALT

```

Example 5.5

Write a 68000 subroutine to compute

$$Y = \sum_{i=1}^N \frac{X_i^2}{N}$$

Assume the X_i 's are 16-bit signed integers and $N = 100$. The numbers are stored in consecutive locations. Assume A0 points to the X_i 's and SP is already initialized. Also, assume no overflow.

Solution

```

SQRE   MOVEM.L D2/D3/A0, -(SP) ; Save registers
        CLR.L D1                ; Clear sum
        MOVEQ.L #99,D2          ; Initialize loop count
LOOP   MOVE.W (A0)+, D3          ; Move  $X_i$  into D3
        MULS D3,D3              ; Computer  $X_i^2$ 

```



```

ADD.L D3, D1
DBF D2, LOOP           ; Store sum of  $X_i^2$  into D1
#DIVU #100, D1         ; Compute  $\sum X_i^2/N$ 
MOVEM.L (SP)+, D2/D3/A0 ; Restore registers
RTS

```

Example 5.6

Write a 68000 assembly language program to move block of data length 100_{10} from the source block starting at location 002000_{16} to the destination block starting at location 003000_{16} .

Solution

```

MOVEA.L #$2000, A4      ; Load A4 with source address
MOVEA.L #$3000, A5      ; Load A5 with destination
                        ; address
MOVEQ.L #99, D0         ; Load D0 with count  $-1 = 99_{10}$ 
START MOVE.W (A4)+, (A5)+ ; MOVE source data to destination
DBF D0, START           ; Branch if  $D0 \neq -1$ 
END JMP END             ; HALT

```

Example 5.7

Write a 68000 assembly language program to add two words: each contains two ASCII digits. The first word is stored in two consecutive locations with the low byte pointed to by A0 at offset 0300_{16} , while the second word is stored in two consecutive locations with the low byte pointed to by A1 at 0700_{16} . Store the result in hex in memory pointed to by A1.

Solution

```

MOVEQ.B #1, D2          ; Initialize loop
MOVEA.L #$0300, A0      ; Initialize A0
MOVEA.L #$0700, A1      ; Initialize A1
MOVEE.W #0, CCR         ; Clear X-bit
START MOVE.B (A0)+, D0   ; Move data
ANDI.B #$0F, D0         ; Mask off upper nibble
ANDI.B #$0F, (A1)       ; Mask off upper nibble
ADDX.B D0, (A1)+        ; Add data
DBF D2, START
END JMP END             ; Halt

```

Example 5.8

Write 6800 assembly language program to compare a source string of 50_{10} words pointed to by A0 with a destination string pointed to by A1. The program should be halted as soon as a match is found or the end of string is reached.

Solution

```

MOVEQ.B #49, D0         ; Initialize loop count
BEGIN CMPM (A0)+, (A1)+ ; Compare string
DBEQ D0, BEGIN          ; Compare until match or end of
                        ; string
FINISH JMP FINISH       ; Halt

```

Example 5.9

Write a subroutine in 68000 assembly language which can be called by a main program. The subroutine will multiply a signed 16-bit number by a signed 8-bit number. The main program will call this subroutine, store the result in two memory words pointed to by A0 and stop. Assume A1 and A2 contain the signed 8-bit and 16-bit data respectively. Assume that the stack pointer is already initialized. Assume supervisor mode.

Solution

Main program

```

MOVE.B (A1),D0          ; Move 8-bit data
MOVE.W (A2),D1          ; Move 16-bit data
CALL MULTI              ; Call subroutine
MOVE.L D1,(A0)          ; Store result
FINISH JMP FINISH       ; Halt

```

Subroutine

```

MULTI  EXT.W D0          ; Sign extend D0.B
      MULS.W D0,D1       ; Signed multiply
      RTS

```

Example 5.10

Write 68000 assembly language program to subtract two 64-bit numbers. Assume D3D4 and D1D0 contain the two 64-bit numbers. Store the result in D3D4.

Solution

```

SUB.L D0,D4             ; Subtract
                        ; Low 32-bit
SUBX.L D1,D3            ; Subtract
                        ; High 32-bit
FINISH JMP FINISH       ; HALT

```

Example 5.11

Write 68000 assembly language program that will perform the following operation:

$$5 * X - 6 * Y + (X/8) \rightarrow D1.L$$

where X is unsigned 8-bit number stored in lowest byte of D0 and Y is 16-bit signed number stored in upper 16-bit number in D1 respectively. Discard remainder of X/8.

Solution

```

ANDI.W #$00FF,D0       ; Convert X to 16 bits
MOV.W D0,D2            ; Save D0
MULU.W #5,D0           ; D0.L ← 5 * X
SWAP.W D1              ; Data in low 16-bit
MULS.W #-6,D1          ; D1.L ← -6 * Y
ADD.L D0,D1            ; D1.L ← 5 * X - 6 * Y
LSR.W #3,D2            ; D2.W ← X/8

```



```

ANDI.L #$0000FFFF,D2 ; Mask off high 16 bits of D2
ADD.L D2,D1           ; Store result in D1.L
FINISH JMP FINISH

```

Example 5.12

Write 6800 assembly program to convert a number from Fahrenheit degrees in bytes to Celsius degrees. The source byte is assumed to reside in memory pointed to by A0 and the result to be stored in D0. Use the equation $C = [(F - 32)/9] * 5$.

Solution

```

MOVE.B (A0),D0 ; Get degrees F.
EXT.W D0       ; sign extend
SUBI.W #32,D0  ; subtract 32
MULS.W #5,D0   ; D0 ← (F - 32) * 5
DIVS #9,D0     ; D0 ← D0/9
FINISH JMP FINISH

```

5.7 68000 Pins and Signals

The 68000 is housed in one of the following packages:

- 64-pin dual in-line package (DIP)
- 68-Terminal Chip Carrier
- 68-pin Quad Pack
- 68-pin Grid Array

Figure 5.7 shows the pin diagrams of the 64-pin DIP package. Pin diagrams for the other three packages are shown in Appendix B.

The 68000 is provided with several Vcc and ground pins. Power is thus distributed in order to reduce noise problems at high frequencies.

In order to build a prototype to demonstrate that the paper design for the 68000-based microcomputer is correct, one must use either wire-wrap or solder for the actual construction. Prototype board must not be used. This is because at high frequencies above 4 MHz, there will be noise problems due to stray capacitances.

D0-D15 is the 16-bit data bus. Note that Dn indicates a data pin of the 68000 during hardware discussions while Dn refers to a data register of the 68000 during software discussions. All transfers to and from memory and I/O devices are conducted over the 16-bit bus. A1-A23 are the 23 address lines. A0 is obtained by encoding UDS (Upper Data Strobe) and LDS (Lower Data Strobe) lines. The 68000 operates on a single-phase TTL level clock at 4 MHz, 6 MHz, 8 MHz, 10 MHz, 12.5 MHz, 16.67 MHz, or 25 MHz. The 68000 also has a lower-power HCMOS version called the MC68HC000 which can run at 8 MHz, 10 MHz, 12.5 MHz, and 16.67 MHz.

There is no on-chip clock generator/driver circuitry and therefore the clock must be generated externally. This clock input is utilized internally by the 68000 to generate additional clock signals for synchronizing the 68000's internal operation.

Figure 5.8 shows the 68000 CLK waveform and clock timing specifications.

The clock is at TTL compatible voltage. The clock timing specification provides data for three different clock frequencies: 8 MHz, 10 MHz, and 12.5 MHz.

The 68000 CLK input can be provided by a crystal oscillator or by designing an external circuit. Figure 5.9 shows a simple oscillator to generate the 68000 CLK input. The clock circuit

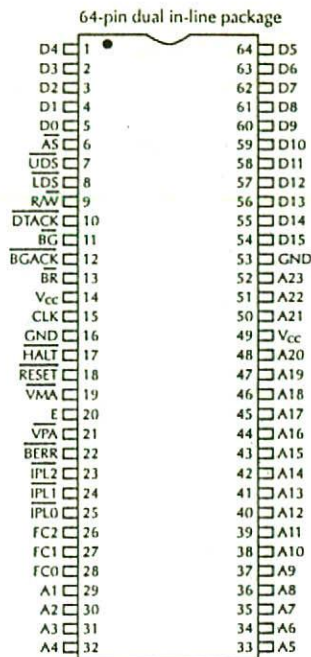


FIGURE 5.7 68000 pin diagram.

uses two inverters connected in series. Inverter 1 is biased in its transition region by the resistor R. Inverter 1 inputs the crystal output (sinusoidal) to produce logic pulse train at the output of inverter 1. Inverter 2 sharpens the wave and drives the crystal. For this circuit to work, HCMOS logic (74HC00, 74HC02, or 74HC04) must be used and a coupling capacitor should be connected across the supply terminals to reduce the ringing effect during high-frequency switching of the HCMOS devices. Additionally, the output of this oscillator is fed to the clock input of a D-flip-flop (74LS74) to further reduce the ringing. Hence, a clock signal of 50% duty cycle at a frequency of 1/2 the crystal frequency is generated.

The 68000 consumes about 1.5 watts of power.

The 68000 signals can be divided into five functional categories. These are

1. Synchronous and asynchronous control lines
2. System control lines
3. Interrupt control lines
4. DMA control lines
5. Status lines

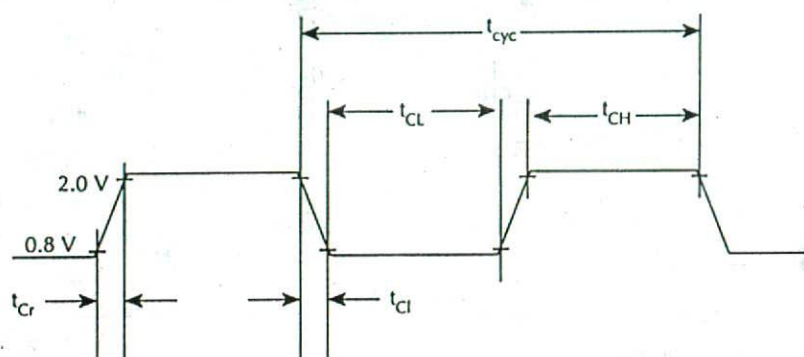
5.7.1 Synchronous and Asynchronous Control Lines

The 68000 bus control is asynchronous. This means that once a bus cycle is initiated, the external device must send a signal back in order to complete it. The 68000 also contains three synchronous control lines that facilitate interfacing to synchronous peripheral devices such as Motorola's inexpensive MC6800 family. Note that synchronous operation means that bus control is synchronized or clocked using a common system clock signal. In 6800 family peripherals, this common clock is a phase 2 (ϕ_2) or an E clock signal depending on the particular chip used.

With synchronous control, all READ and WRITE operations must be synchronized with the common clock. However, this may create problems when interfacing slow peripheral devices. This problem does not arise with asynchronous bus control.

AC Electrical Specifications — Clock Timing

Characteristic	Symbol	8 MHz		10 MHz		12.5 MHz		Unit
		Min	Max	Min	Max	Min	Max	
Frequency of Operation	1	4.0	8.0	4.0	10.0	4.0	12.5	MHz
Cycle Time	t_{cyc}	125	250	100	250	80	250	ns
Clock Pulse Width	t_{CL}	55	125	45	125	35	125	ns
	t_{CH}	55	125	45	125	35	125	
Rise and Fall Times	t_{CR}	—	10	—	10	—	5	ns
	t_{CI}	—	10	—	10	—	5	



Note: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted. The voltage swing through this range should start outside and pass through the range such that the rise or fall will be linear between 0.8 and 2.0 volts.

FIGURE 5.8 Clock input timing diagram.

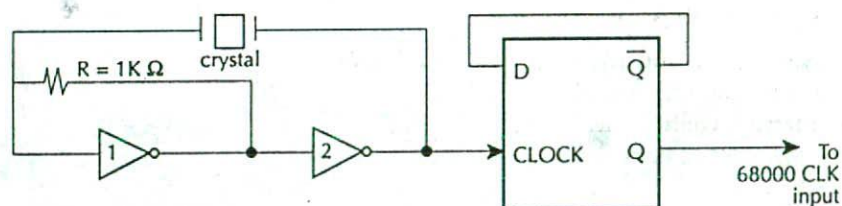


FIGURE 5.9 External clock circuitry.

The 68000 utilizes either synchronous or asynchronous bus cycles for data transfer. The purpose of using the synchronous bus cycles is to interface Motorola's inexpensive MC6800 peripherals to the 68000. On the other hand, the asynchronous bus protocol of the 68000 too is used to interface non-6800 type peripherals to the 68000. Once the 68000 starts a bus cycle, it expects one of three signals to tell it how to terminate the cycle. These signals are VPA (synchronous operation), DTACK (Asynchronous operation), or BERR (Bus Error; when no DTACK is received from the peripheral device). Synchronous and Asynchronous operations are described in the following.

The 68000 has three control lines to transfer data over its bus in a synchronous manner. These are E (enable), VPA (valid peripheral address), and VMA (valid memory address).

The E clock corresponds to phase 2 clock of the 6800. The E clock is output at a frequency that is 1/10th of the 68000 input clock. The VPA is an input which tells the 68000 that a 6800 device is being addressed and therefore data transfer must be synchronized with the E clock. The VMA is processor's response to VPA. VPA is asserted when the memory address is valid. This also tells the external device that the next data transfer over the data bus will be synchronized with the E clock. VPA can be generated by decoding the address pins and address strobe (AS). Note that the 68000 asserts AS low when the address on the address bus is valid. VMA is typically used as part of the chip select of the 6800 peripheral. This ensures that the 6800 peripherals are selected and deselected at the correct time. The 6800 peripheral interfacing sequence is provided in the following:

1. The 68000 initiates a cycle by starting a normal read or write cycle.
2. The 6800 peripheral defines the 6800 cycle by asserting the 68000 VPA input. If the VPA is asserted as soon as possible after the assertion of AS, then VPA will be recognized as being asserted on the falling edge of S4 (third cycle). If the VPA is not asserted at the falling edge of S4 (third cycle), the 68000 inserts wait states until VPA is recognized by the 68000 as asserted. DTACK should not be asserted while VPA is asserted. The 6800 peripheral must remove VPA within one clock after AS is negated.
3. The 68000 monitors enable (E) until it is low. The 68000 then synchronizes all read and write operations with the E clock. VMA output pin is asserted low by the 68000.
4. The 68000 peripheral waits until E is active (HIGH) and then transfers the data.
5. The 68000 waits until E goes low (on a read cycle the data is latched as E goes low internally). The 68000 then negates VMA, AS, UDS, and LDS. The 68000 thus terminates the cycle and starts the next cycle.

Asynchronous operation is not dependent on a common signal. The 68000 utilizes the asynchronous control lines to transfer data between the 68000 and peripheral devices via handshaking. Using asynchronous operation, the 68000 can be interfaced to any peripheral chip regardless of its speed.

The 68000 provides five lines to control address and data transfers asynchronously. These are AS (Address Strobe), R/W (Read/Write), DTACK (Data Acknowledge), UDS (Upper Data Strobe), and LDS (Lower Data Strobe).

The 68000 outputs \overline{AS} to notify the peripheral device when data are to be transferred. \overline{AS} is active LOW when the 68000 provides a valid address on the address bus. The R/W is HIGH for read and LOW for write. The DTACK is used to tell the 68000 that a transfer is to be performed.

When the 68000 wants to transfer data asynchronously, it first generates the required address on the address lines in order to select the peripheral device and then activates the \overline{AS} line.

Since the \overline{AS} line tells the peripheral chip when to transfer data, the \overline{AS} line should be part of the address decoding scheme. After enabling the \overline{AS} , the 68000 enters the wait state until it receives the DTACK from the selected peripheral device. On receipt of the DTACK, the 68000 knows that the peripheral device is ready for data transfer. The 68000 then utilizes the R/W, UDS, LDS and data lines to transfer data.

UDS and LDS are defined as follows:

\overline{UDS}	\overline{LDS}	Data Transfer occurs via	Address
1	0	D0-D7 pins for byte	Odd
0	1	D8-D15 pins for byte	Even
0	0	D0-D15 pins for word or Long word	Even

\overline{UDS} and \overline{LDS} are used to segment the memory into bytes instead of words. When the UDS is asserted, contents of even addresses are transferred on the high-order 8 lines of the data bus, D8-D15. The 68000 internally shifts these data to the low byte of the specified register. When LDS is asserted, contents of odd addresses are transferred on the low-order 8 lines of the data bus, D0-D7. During word and long word transfers, both UDS and LDS are asserted and information is transferred on all 16 data lines, D0-D15. However, an additional cycle is required for a long word 32-bit transfer. Note that during byte memory transfers, A0 corresponds to UDS for even addresses (A0 = 0) and to LDS for odd addresses (A0 = 1). The circuit in Figure 5.10 shows how even and odd memory chips are interfaced to the 68000.

5.7.2 System Control Lines

The 68000 has three control lines, namely, \overline{BERR} (Bus Error), \overline{HALT} , and \overline{RESET} , that are used to control system-related functions.

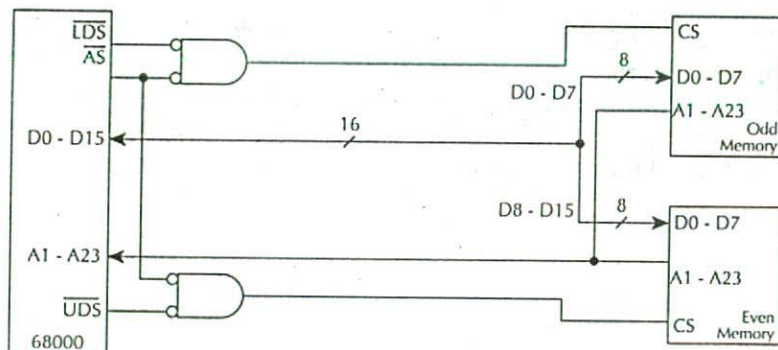


FIGURE 5.10 Interfacing of the 68000 to even and odd addresses.

The BERR is an input to the 68000 that is used to inform the processor that there is a problem with the instruction cycle currently being executed. With asynchronous operation, this problem may arise if the 68000 does not receive DTACK or VPA from a peripheral device. An external timer can be used to activate the BERR pin if the external device does not send DTACK within a certain period of time. On receipt of the BERR, the 68000 does one of the following:

- Automatically reruns the instruction cycle which caused the error
- Executes an error service routine

The troubled instruction cycle is rerun by the 68000 if it receives a HALT signal along with the BERR signal. On receipt of LOW on both HALT and BERR pins, the 68000 completes the current instruction cycle and then places its buses into the high impedance state. On removal of both HALT and BERR (that is, when both HALT and BERR are HIGH), the 68000 reruns the troubled instruction cycle. The cycle can be rerun repeatedly if both BERR and HALT are enabled/disabled continually.

On the other hand, an error service routine is executed only if the BERR is received without HALT. In this case, the 68000 will branch to a bus error vector address where the user can write a service routine. If two simultaneous bus errors are received due to rerun of the troubled instruction cycle via the BERR pin without HALT, the 68000 automatically goes into the HALT state until it is reset.

The HALT line can also be used by itself to perform single-stepping or to provide DMA. When HALT input is activated, the 68000 completes the current instruction and goes into a high impedance state until HALT is returned to HIGH. By enabling/disabling the HALT line continually, the single-stepping debugging can be accomplished. However, since most 68000 instructions consist of more than one instruction cycle, single-stepping using HALT is not normally used. Rather, the trace bit in the status register is used to single-step the complete instruction.

One can also use HALT to perform microprocessor-halt DMA. Since the 68000 has separate DMA control lines, DMA using the HALT line will not normally be used.

The HALT pin can also be used as an output signal. The 68000 will assert the HALT pin LOW when it goes into a HALT state as a result of a catastrophic failure. The double bus error (activation of BERR twice) is an example of this type of error. When this occurs, the 68000 places its bus into a high impedance state until it is reset. The HALT line informs the peripheral devices of the catastrophic failure.

The RESET line of the 68000 is also bidirectional. In order to reset the 68000, both the RESET and HALT pins must be asserted at the same time. The 68000 executes a reset service routine automatically for loading the PC with the starting address of the program. The 68000 RESET pin can also be used as an output line. A LOW can be sent to this output line by executing the RESET instruction in the supervisor mode in order to reset external devices connected to the 68000. The execution of the RESET instruction does not affect any data, address, or status register. Therefore, the RESET instruction can be placed anywhere in the program whenever the external devices need to be reset.

In order to reset the 68000, both the RESET and HALT pins must be asserted simultaneously by an external circuit. Figure 5.11 shows the timing diagram for the 68000 reset operation.

Upon hardware reset, the 68000 performs the following:

1. The 68000 reads four words from addresses \$000000, \$000002, \$000004, and \$000006. The 68000 loads the supervisor stack pointer high and low words with the contents of locations \$000000 and \$000002, respectively. Also, the program counter high and low words are loaded with the contents of locations \$000004 and \$000006, respectively.
2. The 68000 initializes the status register to an interrupt mask level of seven.
3. No other registers are affected by hardware reset.

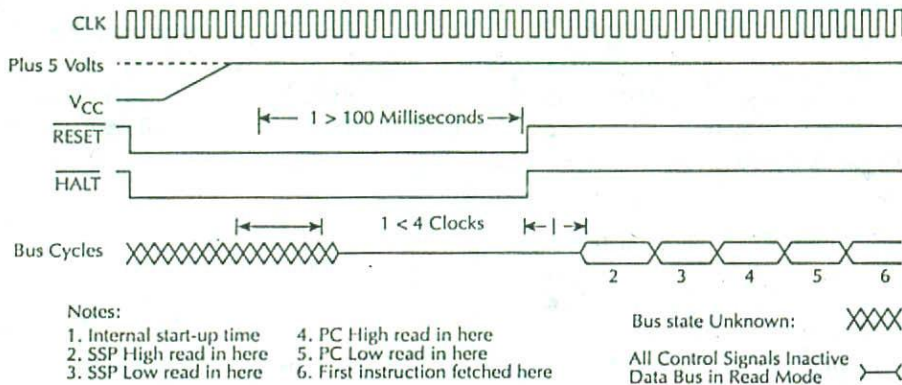


FIGURE 5.11 68000 reset operation timing diagram.

When a RESET instruction is executed, the 68000 drives the RESET pin low for 124 clock periods. In this case, the processor is trying to reset the rest of the system. Therefore, there is no effect on the internal state of the 68000. All of the 68000 internal registers and the status register are unaffected by the execution of a reset instruction. All external devices connected to the RESET line will be reset at the completion of the reset instruction.

Asserting the RESET and HALT lines for 10 clock cycles will cause a processor reset, except when Vcc is initially applied to the 68000. In this case, an external reset must be applied for at least 100 milliseconds.

The reset circuit (used for 8085) depicted in Figure 5.12a satisfies the 68000 reset requirements mentioned above.

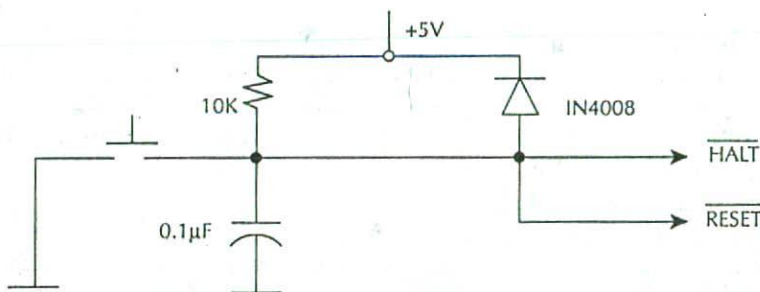


FIGURE 5.12a 68000 RESET circuit (simple).

The above circuit is similar to the 8085 reset circuit except that the output goes to both RESET and HALT lines of the 68000. A more accurate RESET circuit is shown in Figure 5.12b.

The Motorola MC1455 in Figure 5.12b is a timer chip that provides accurate time delays or oscillation. The timer is precisely controlled by external resistors and capacitors. The timer may be triggered by an external trigger input (falling waveform) and can reset by external reset input (falling waveform).

The reset circuit in Figure 5.12b will assert the 68000 RESET pin for at least 10 clock cycles. The internal block diagram of the MC1455 is shown in Figure 5.12c.

When the input voltage (Vcc) to the trigger comparator (COMPB) falls below $1/3 V_{cc}$, the comparator output (COMPA) triggers the flip-flop so that its output becomes low. This turns the capacitor discharge transistor OFF and drives the digital output to the HIGH state. This condition permits the capacitor to charge at an exponential rate set by the RC time constant.

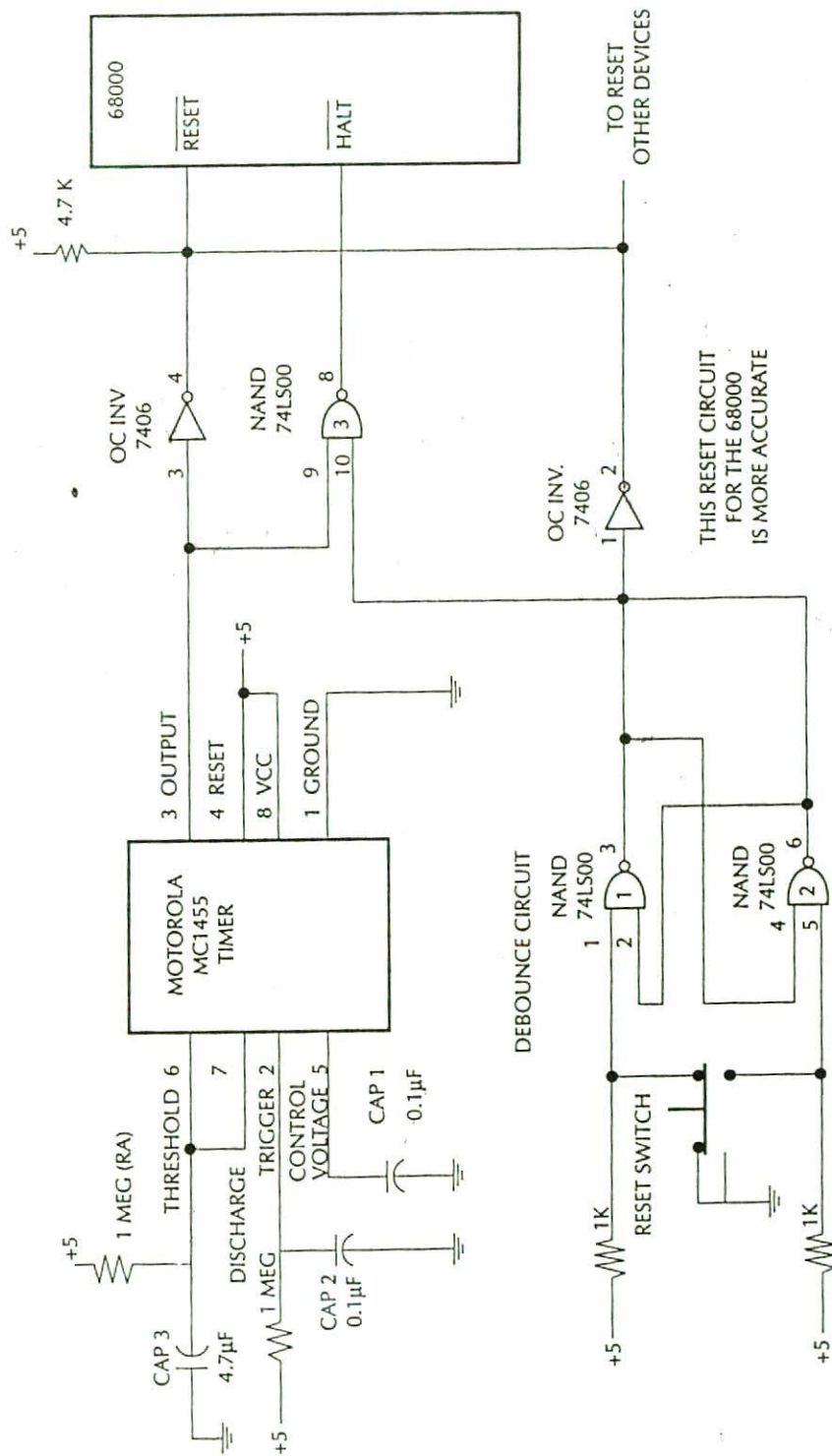


FIGURE 5.12b 68000 RESET circuit.

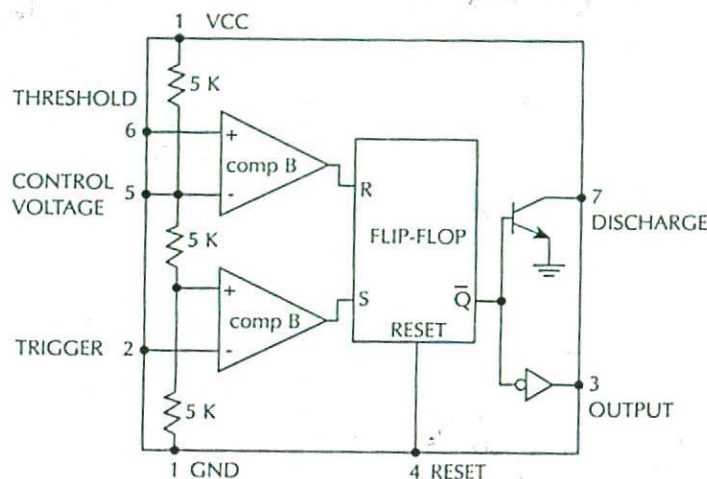


FIGURE 5.12c MC1455 internal block diagram.

When the capacitor voltage reaches $\frac{2}{3} V_{CC}$, the threshold comparator resets the flip-flop. The action discharges the timing capacitor and returns the digital output to the LOW state. The output will be HIGH for $t = 1.1 R_A C$ seconds, where $R_A = 1M$ and $C = 4.7 \mu F$.

The MC1455 can be connected so that it will trigger itself and the capacitor voltage will oscillate between $\frac{1}{3} V_{CC}$ and $\frac{2}{3} V_{CC}$. Once the flip-flop has been triggered by an input signal, it cannot be retriggered until the present timing period has been completed. A reset pin is provided to discharge the capacitor, thus interrupting the timing cycle. The reset pin should be tied to V_{CC} when not in use. With proper trigger input as in the figure, the MC1455 output will stay HIGH for $1.1 R_A C = 5.17 s$ ($1.1 * 10^6 * 4.7 * 10^{-6}$). The 68000 requires the RESET and HALT lines to be low for at least 10 cycles. If the 68000 clock cycle is $0.125 \mu s$ (8-MHz clock), then the 68000 RESET and HALT pins must be LOW for at least $0.125 \mu s * 10 = 1.25 \mu s$. Since the MC1455 output is connected to the 68000 RESET pin through an inverter, the RESET pin will be held LOW for 5.17 s (greater than $1.25 \mu s$). Hence, the timing requirement for the 68000 RESET pin is satisfied. The HALT pin is activated by NANDing the MC1455 true output and the debouncing circuit output. The HALT pin is LOW when both inputs to NAND gate #3 are HIGH. The MC1455 output is HIGH for 5.17 s and the output of the debounce circuit is HIGH when the push button is activated. This will generate a LOW at the HALT pin for 5.17 s (greater than $1.25 \mu s$). The timing requirements of the 68000 RESET and HALT pins will be satisfied by the reset circuit of Figure 5.12b.

Note that when the reset circuit is not activated, the bottom input of AND gate #2 is HIGH and the top input of AND gate #1 is LOW (grounded to LOW, see Figure 5.12b). Since a NAND gate always produces a HIGH output when one of the inputs is LOW, the output of NAND gate #1 will be HIGH. This will make the top input of NAND gate #2 HIGH and thus the output of NAND gate #2 will be LOW, which in turn will make the output of NAND gate #3 HIGH. Therefore, the 68000 will not be reset when the push button is not activated. Upon activation of the push button, the bottom input of NAND gate #2 is LOW (grounded to LOW); this will make the output of NAND gate #2 HIGH. Hence, both inputs of NAND gate #3 will be HIGH providing a LOW at the HALT pin for 5.17 s (greater than $1.25 \mu s$).

TABLE 5.13 Function Code Lines

FC2	FC1	FC0	Operation
0	0	0	Unassigned
0	0	1	User data
0	1	0	User program
0	1	1	Unassigned
1	0	0	Unassigned
1	0	1	Supervisor data
1	1	0	Supervisor program
1	1	1	Interrupt acknowledge

5.7.3 Interrupt Control Lines

$\overline{\text{IPL0}}$, $\overline{\text{IPL1}}$, and $\overline{\text{IPL2}}$ are interrupt control lines. These lines provide for seven interrupt priority levels ($\overline{\text{IPL2}}$, $\overline{\text{IPL1}}$, $\overline{\text{IPL0}} = 111$ means no interrupt and $\overline{\text{IPL2}}$, $\overline{\text{IPL1}}$, $\overline{\text{IPL0}} = 000$ means nonmaskable interrupt). $\overline{\text{IPL2}}$, $\overline{\text{IPL1}}$, $\overline{\text{IPL0}} = 001$ through 110 provides six maskable interrupts. The 68000 interrupts are discussed later in this chapter.

5.7.4 DMA Control Lines

$\overline{\text{BR}}$ (Bus Request), $\overline{\text{BG}}$ (Bus Grant), and $\overline{\text{BGACK}}$ (Bus Grant Acknowledge) lines are used for DMA purposes. The 68000 DMA will be discussed later in this chapter.

5.7.5 Status Lines

The 68000 has three output lines called the function code pins (FC2, FC1, and FC0). Table 5.14 shows how these lines tell external devices whether user data, user program, supervisor data, or supervisor program is being addressed. These lines can be decoded to provide user or supervisor programs and/or data, and interrupt acknowledge as shown in Table 5.13.

The FC2, FC1, and FC0 pins can be used to partition memory into four functional areas: user data memory, user program memory, supervisor data memory, and supervisor program memory. Each memory partition can directly access up to 16 megabytes, and thus the 68000 can be used to directly address up to 64 megabytes of memory. This is shown in Figure 5.13.

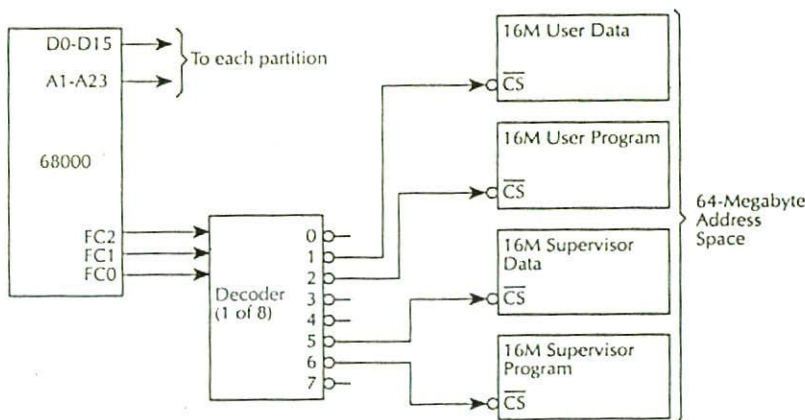


FIGURE 5.13 Partitioning 68000 address space using FC2, FC1, and FC0 pins.

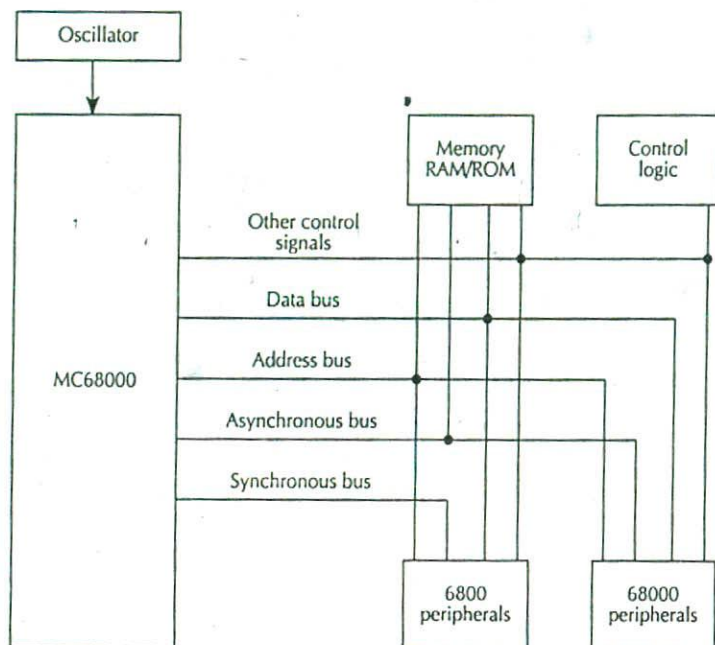


FIGURE 5.14 68000 basic system.

5.8 68000 System Diagram

Figure 5.14 shows a simplified version of the 68000 basic system diagram.

5.9 Timing Diagrams

The 68000 family of processors (68000, 68008, 68010, and 68012) uses a handshaking mechanism to transfer data between the processors and the peripheral devices. This means that all these processors can transfer data asynchronously to and from peripherals of varying speeds.

Figure 5.15 shows 68000 read and write cycle timing diagrams.

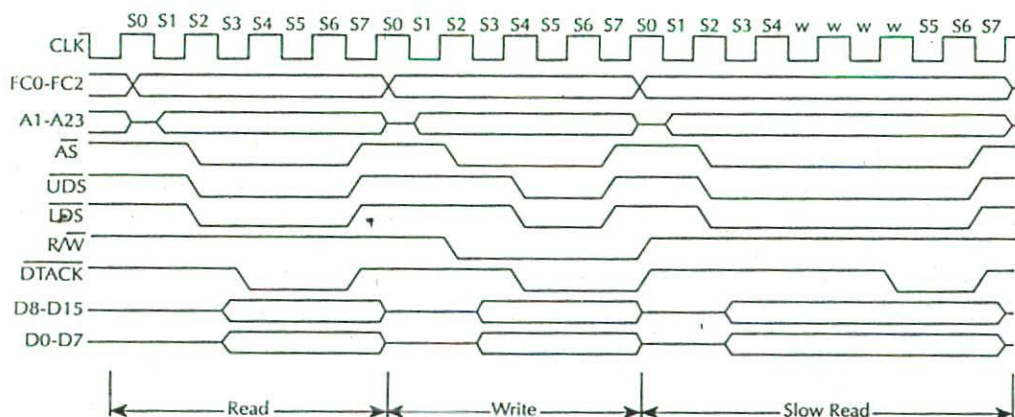


FIGURE 5.15 Read and write cycle timing diagrams.

During the read cycle, the 68000 obtains data from a memory location or an I/O port. If the instruction specifies a word such as `MOVE.W $020504, D1` or a long word such as `MOVE.L $030808, D0`, the 68000 reads both upper and lower bytes at the same time by asserting UDS and LDS pins. When the instruction is for a byte operation the 68000 utilizes an internal bit to find which byte to read and then outputs the data strobe required for that byte. For byte operations, when the address is even ($A0 = 0$), the 68000 asserts UDS and reads data via D8-D15 pins into low byte of the specified data register. On the other hand, for reading a data byte from an odd address ($A0 = 1$), the 68000 outputs LOW on LDS and reads data byte via D0-D7 pins into low byte of the specified data register. For example, consider `MOVE.B $507144, D5`. The 68000 outputs LOW on UDS (since $A0 = 0$) and high on LDS. The memory chip's 8 data lines must be connected to the 68000 D8-D15 pins. The 68000 reads the data byte via D8-D15 pins into the low byte of D5. Note that for reading a data byte from an odd location by executing an instruction such as `MOVE.B $507145, D5`, the 8 data lines of the memory chip must be connected to the 68000 D0-D7 pins. The 68000, in this case, outputs low on LDS ($A0 = 0$) and high on UDS, then reads the data byte into low byte of D5.

Now, let us discuss the read timing diagram of Figure 5.15. Consider Figure 5.15 for word read timing. During S0, address and data signals are in the high impedance state. At the start of S1, the 68000 outputs the address on its address pins (A1-A23). During S0, the 68000 outputs FC2-FC0 signals. AS is asserted at the start of S2 to indicate valid address on bus. AS can be used at this point to latch the signals on the address pins. The 68000 asserts UDS and LDS pins to indicate a word transfer. The 68000 also outputs high on the R/W pin to indicate a read operation.

The 68000 now waits for the peripheral device to assert DTACK. Upon placing data on the data bus, the peripheral device asserts DTACK. The 68000 samples the DTACK signal at the end of S4. If DTACK is not asserted by the peripheral device, the processor automatically inserts wait states (W).

However, upon assertion of DTACK, the 68000 negates AS, UDS, and LDS signals and then latches the data from data bus into an internal register at the end of the next cycle. Once the selected peripheral device senses that the 68000 has obtained data from the data bus (by recognizing the negation of AS, UDS, or LDS), the peripheral device must negate DTACK immediately, so that it does not interfere with the start of the next cycle.

If DTACK is not asserted by a peripheral at the end of state 4 (Figure 5.15), the 68000 inserts wait states. The 68000 outputs valid addresses on the address pins and keeps asserting AS, UDS, and LDS until the peripheral asserts DTACK. The 68000 always inserts an even number of wait states if DTACK is not asserted by the peripheral, since all 68000 operations are performed using two clock states per clock cycle. Note that in Figure 5.15, the 68000 inserts 4 wait states (2 cycles).

Consider Figure 5.15 for 68000 write word timing. The 68000 outputs the address of the location to be written into the address bus at the start of S1. During S0, the 68000 places the proper function code values at the FC2, FC1, and FC0 pins. If the 68000 used the data bus in the previous cycle, then it places all data pins in the high impedance state and then outputs LOW on AS and R/W pins. At the start of S3, the 68000 places data on D0-D15 pins. The 68000 then asserts UDS and LDS pins at the beginning of S4.

For the memory or I/O device, if DTACK is not asserted by memory or I/O device by the end of S4, the 68000 automatically inserts wait states into the write cycle.

The 68000 provides a special cycle called the read-modify-write cycle during execution of only the TAS instruction. This instruction reads a data byte, sets condition codes according to the byte value, sets bit 7 of the byte, and then writes the byte back into memory. The TAS instruction can be used in providing data transfer between two 68000 processors using shared RAM. The data byte mentioned above is held in the shared RAM. The read/modify/write cycle is indivisible. That is, it cannot be interrupted by any other bus request.

5.10 68000 Memory Interface

One of the advantages of the 68000 is that it can easily be interfaced to memory chips. This is because the 68000 goes into a wait state if DTACK is not asserted by the memory devices at the end of S4.

A simplified schematic showing an interface of a 68000 to two 2716s and two 6116s is shown in Figure 5.16. The 2716 is a 2 K \times 8 EPROM and the 6116 is a 2 K \times 8 static RAM. For a 4-MHz clock, each cycle is 250 ns. The 68000 samples DTACK at the falling edge of the S4 (third clock cycle) and latches data at the falling edge of S6 (fourth clock cycle). AS is used to assert DTACK. AS goes to LOW after 500 ns (two clock cycles). The time delay between AS going LOW and the falling edge of S6 is 500 ns.

Since the access times of the 2716 and 6116 are, respectively, 450 and 120 ns, delay circuits for DTACK are not required. As an example, the 68000-2716 timing parameters with various 68000 clock frequencies are as follows:

Case	68000 frequency	Clock cycle	Time before first DTACK is sampled	Comment
1	6 MHz	166.7 ns	3(166.7) = 500.1 ns	No timing problem
2	8 MHz	125 ns	3(125) = 375 ns	No timing problem since the 68000 latches data after 500 ns
3	10 MHz	100 ns	3(100) = 300 ns	Not enough time for the 2716 to place data on bus; needs delay circuit
4	12.5 MHz	80 ns	3(80) = 240 ns	Same as case 3
5	16.67 MHz	60 ns	3(60) = 180 ns	Same as case 3
6	25 MHz	40 ns	3(40) = 120 ns	Same as case 3

Note that LDS and UDS must be used as chip selects as in the figure. They must not be connected to A0 of the memory chips, since in that case half of the memory in each chip will be wasted.

Let us determine the memory map of Figure 5.16. Assume the don't care values of A23-A14 to be zeros.

Memory map for even 2716 (A12 must be zero to select even 2716 and A13 must be zero to deselect even 6116)

A23	A22	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12	A11	...	A1	A0
0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0

can be from all zeros to all ones

The memory map includes the addresses \$000000, \$000002, \$000004, ..., \$000FFE.

Memory map for odd 2716 (A12 must be 0 to select odd 2716 and A13 must be 0 to deselect odd 6116)

A23	A22	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12	A11	...	A1	A0
0	0	0	0	0	0	0	0	0	0	0	0	0	...	1	1

can be from all zeros to all ones

The memory map includes the addresses \$000001, \$000003, \$000005, ..., \$000FFF.

Memory map for even 6116 (A12 must be one to deselect even 2716 and A13 must be one to select even 6116)

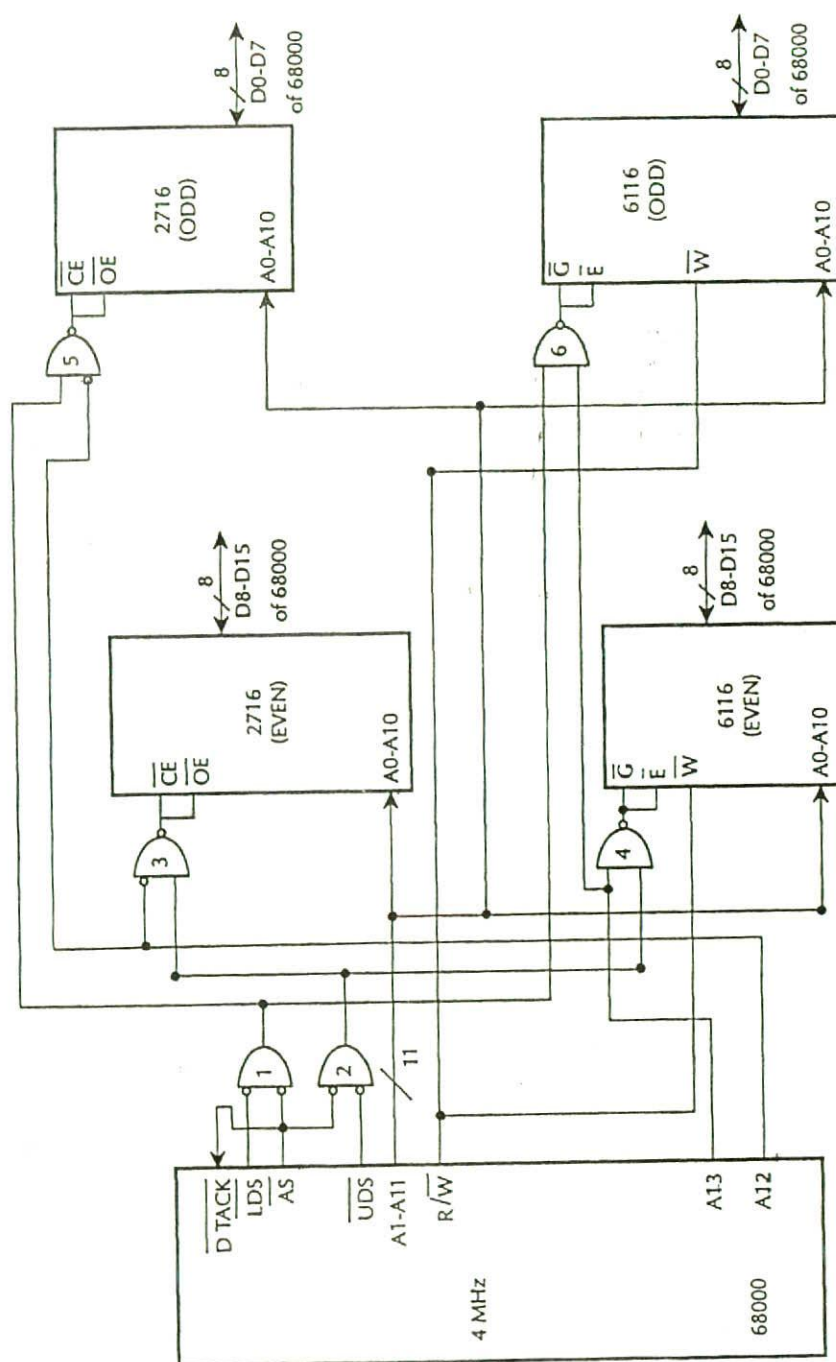


FIGURE 5.16 68000 interface to 2716/6116.

A23	A22	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12	A11	...	A1	A0
0	0	0	0	0	0	0	0	0	0	1	1	0			

can be from all zeros to all ones

The memory map includes the addresses \$003000, \$003002, ..., \$003FFE.

Memory map for odd 6116 (A12 must be one to deselect odd 2716 and A13 must be one to select to odd 6116)

A23	A22	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12	A11	...	A1	A0
0	0	0	0	0	0	0	0	0	0	1	1	1			

can be from all zeros to all ones

The memory map includes the addresses \$003001, \$003003, ..., \$003FFF.

In summary, the memory for the schematic in Figure 5.16 is shown in the following:

EVEN 2716 EPROM	\$000000, \$000002, \$000004,
	\$000FFE
ODD 2716 EPROM	\$000001, \$000003, \$000005,
	\$000FFF
EVEN 6116 RAM	\$003000, \$003002, \$003004,
	\$003FFE
ODD 6116 RAM	\$003001, \$003003, \$003005,
	\$003FFF

In the following, some examples will be considered to illustrate the use of memory map of the schematic of Figure 5.6 using the 68000 MOVE instruction.

Consider MOVE.B \$000004, D1. Upon execution of this MOVE instruction, the 68000 reads the [\$000004] from even 2716 to the low byte of D1. In order to execute the instruction, the processor places the upper 23 bits of \$000004 on its A23-A1 pins, and asserts \overline{UDS} and AS. Since $A_{12} = 0$, the output of AND gate 2 = 1, the output of AND gate 3 generates a LOW, making CE and OE of the even 2716 LOW. When the 68000 samples DTACK (asserted by AS in Figure 5.16), data placed on the 68000 D8-D15 lines by the even 2716 are read by the processor into low byte of D1. Similarly, a byte read operation from the odd 2716 can be explained.

Consider MOVE.W \$000004, D1. The 68000 asserts both \overline{UDS} and \overline{LDS} in this case. The outputs of AND gates 3 and 5 are LOW. Both the even 2716 and the odd 2716 are selected. Data placed on D0-D7 pins and D8-D15 pins of the 68000 from locations \$000004 and \$000005 of the even 2716 and odd 2716 are read by the processor, respectively, into bits 8-15 and bits 0-7 of D0.

Consider MOVE.B D2, \$003001. The 68000 asserts \overline{LDS} , AS and outputs HIGH on A12 and A13. The output of AND gate 6 is LOW and thus selects odd 6116. Since $R/W = 0$, the odd 6116 writes the low byte of D2 from 68000 D0-D7 pins into location \$003001. Similarly, other byte operations from the RAMs can be illustrated.

Consider MOVE.L D3, \$003002. The 68000 asserts AS, and both \overline{UDS} and \overline{LDS} . It also outputs HIGH on A12 and A13. Both RAMs are selected. The high 16 bits of the 32-bit data placed on D0-D15 pins by the 68000 are written into locations #003002 (byte 0) and \$003003 (byte 1), and the low 16 bits of the 32-bit data placed on D0-D15 pins by the 68000 are written into locations \$003004 (byte 2) and \$003005 (byte 3), respectively, of the even and odd 6116s. Similarly, the other long word operations can be illustrated. Note that a long-word write or read is done by the 68000 with 2 bus cycles.

5.11 68000 Programmed I/O

As mentioned before, the 68000 uses memory-mapped I/O. Programmed I/O can be achieved in the 68000 using one of the following ways:

1. By interfacing the 68000 asynchronously with its own family of peripheral devices such as the MC-68230, Parallel Interface/Timer chip.
2. By interfacing the 68000 synchronously with 6800 peripherals such as the MC6821 (note that synchronization means that every READ or WRITE operation is synchronized with the clock).

5.11.1 68000-68230 Interface

The MC68230 parallel interface/timer (PI/T) provides double buffered parallel interfaces and a timer for 68000 systems. Note that double buffering means that the ports have dual latches. Double buffering allows simultaneous reading of data from a port by the microprocessor and placing of data into the same port by an external device via handshaking. Double buffering is most useful in situations where a peripheral device and the processor are capable of transferring data at nearly the same speed. If there is a large difference in speed between the microprocessor and the peripheral, little or no benefit of double buffering is achieved. In these cases, however, there is no penalty for using double buffering. Double buffering permits the fetch operation of the data transmitter to be overlapped with the store operation of the data receiver.

The parallel interfaces provided by the 68230 can be 8 or 16 bits wide with unidirectional or bidirectional modes. In the unidirectional mode, a data direction register configures each port as an input or output. In the bidirectional mode, the data direction registers are ignored and the direction is determined by the state of four handshake pins.

The 68230 allows use of interrupts, and also provides a DMA request pin for connection to a DMA controller chip such as the MC68450. The timer contains a 24-bit-wide counter. This counter can be clocked by the output of a 5-bit (divide by 32) prescaler or by an external timer input pin (TIN).

Table 5.14 provides the signal summary and Figure 5.17 shows the 68230 pin diagram. The 68230 is a 48-pin device.

The purpose of D0-D7, R/W, and CS pins is obvious. RS1-RS5 are five register select input pins for selecting the 23 internal registers.

During read or interrupt acknowledge cycles, DTACK is asserted after data have been provided on the data bus and during write cycles it is asserted after data have been accepted at the data bus. A pullup resistor is required to maintain DTACK high between bus cycles.

Upon activation of the RESET input, all control and data direction registers are cleared and most internal operations are disabled by the assertion of RESET LOW.

The clock pin has the same specifications as the 68000.

PA0-PA7 and PB0-PB7 pins provide two 8-bit ports that may be concatenated to form a 16-bit port in certain modes. The ports may be controlled in conjunction with handshake pins H1-H4. A simple example of a handshake operation for input of data is for the I/O device to indicate to the port that new data are available at the port by activating H1 to HIGH. After input of data by the 68000, the H2 output of the 68230 is set to HIGH to indicate to the I/O device that data have been read and it may now provide another data byte to the port.

PC0-PC7 pins can be used as eight general purpose I/O pins or any combination of six special function pins and two general purpose I/O pins (PC0, PC1).

Port C can be configured as input or output by the port C data direction register.

The alternate function pins TIN, TOUT, and TIACK are timer I/O pins. For example, the PC2 pin can also be used as a timer input TIN. When the 68230 timer is used as an event

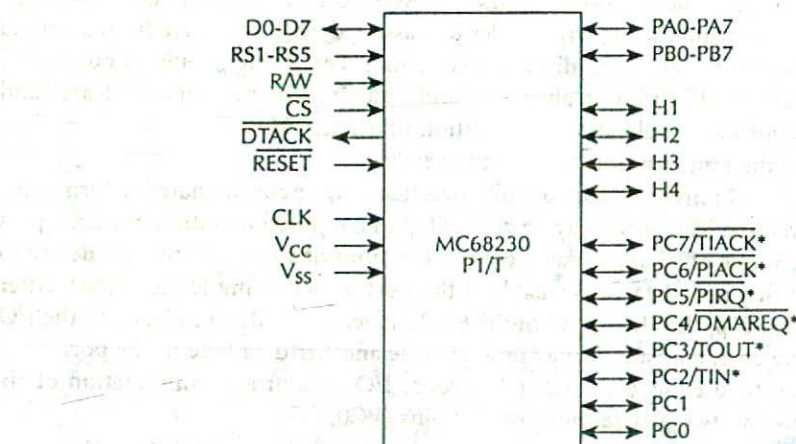
TABLE 5.14 Signal Summary of the MC68230

Signal name	Input/output	Active state	Edge/level sensitive	Output states
CLK	Input		Falling and rising edge	
\overline{CS}	Input	Low	Level	
D0-D7	Input/output	High = 1, low = 0	Level	High, low, high impedance
\overline{DMAREQ}	Output	Low		High, low
\overline{DTACK}	Output	Low		High, low, high impedance*
H1(H3) ^b	Input	Low or high	Asserted edge	
H2(H4) ^c	Input or output	Low or high	Asserted edge	High, low, high impedance
PA0-PA7, ^c	Input/output	High = 1, low = 0	Level	High, low, high impedance
PB0-PB7, ^c PC0-PC7	Input or output input or output			
\overline{PIACK}	Input	Low	Level	
\overline{PIRQ}	Output	Low		Low, high impedance*
RS1-RS5	Input	High = 1, low = 0	Level	
$\overline{R/W}$	Input	High read, low write	Level	
\overline{RESET}	Input	Low	Level	
\overline{TIACK}	Input	Low	Level	
TIN (external clock)	Input		Rising edge	
TIN (run/halt)	Input	High	Level	
TOUT (square wave)	Output	Low		High, low
TOUT (TIRQ)	Output	Low		Low, high impedance*

* Pullup resistors required.

^b H1 is level sensitive for output buffer control in modes 2 and 3.^c Note these pins have internal pullup resistors.

counter, the counter value can be decremented upon application of pulses at TIN by external circuitry. This means that TIN is the timer clock input. TIN can also be configured as the timer run/halt input. In this case, a HIGH at TIN enables the 68230 internal timer clock. Therefore, the 68230 timer runs when TIN = 1. On the other hand, a low on TIN disables the 68230 internal clock and stops the timer. TOUT may provide an active low timer interrupt request



* Individual Programmable Dual-Function Pin

FIGURE 5.17 Logical pin assignment.

TABLE 5.15 68230 Register Addressing Assignments

Register		Register select bit					Accessible	Affected by	
		5	4	3	2	1		Reset	Read cycle
Port General Control Register	(PGCR)	0	0	0	0	0	R W	Yes	No
Port Service Request Register	(PSRR)	0	0	0	0	1	R W	Yes	No
Port A Data Direction Register	(PADDDR)	0	0	0	1	0	R W	Yes	No
Port B Data Direction Register	(PBDDR)	0	0	0	1	1	R W	Yes	No
Port C Data Direction Register	(PCDDR)	0	0	1	0	0	R W	Yes	No
Port Interrupt Vector Register	(PIVR)	0	0	1	0	1	R W	Yes	No
Port A Control Register	(PACR)	0	0	1	1	0	R W	Yes	No
Port B Control Register	(PBCR)	0	0	1	1	1	R W	Yes	No
Port A Data Register	(PADR)	0	1	0	0	0	R W	No	*
Port B Data Register	(PBDR)	0	1	0	0	1	R W	No	*
Port A Alternate Register	(PAAR)	0	1	0	1	0	R	No	No
Port B Alternate Register	(PBAR)	0	1	0	1	1	R	No	No
Port C Data Register	(PCDR)	0	1	1	0	0	R W	No	No
Port Status Register	(PSR)	0	1	1	0	1	R W ^b	Yes	No
Timer Control Register	(TCR)	1	0	0	0	0	R W	Yes	No
Timer Interrupt Vector Register	(TIVR)	1	0	0	0	1	R W	Yes	No
Counter Preload Register High	(CPRH)	1	0	0	1	1	R W	No	No
Counter Preload Register Middle	(CPRM)	1	0	1	0	0	R W	No	No
Counter Preload Register Low	(CPRL)	1	0	1	0	1	R W	No	No
Count Register High	(CNTRH)	1	0	1	1	1	R	No	No
Count Register Middle	(CNTRM)	1	1	0	0	0	R	No	No
Count Register Low	(CNTRL)	1	1	0	0	1	R	No	No
Timer Status Register	(TSR)	1	1	0	1	0	R W ^b	Yes	No

Note: R = read; W = write.

* Mode dependent.

^b A write to this register may perform a special status resetting operation.

output or a general purpose square output, initially high. TIACK is an active low high-impedance input used for timer interrupt acknowledge.

Ports A and B have an independent pair of active low interrupt request (PIRQ) and interrupt acknowledge (PIACK) pins. PIRQ is an output pin and is used by the 68230 when it implements an interrupt-driven parallel I/O configuration. Therefore, PC2-PC7 pins may or may not be available for general purpose I/O. The 68230 PIRQ pin can be connected to one of the 68000 IPL pins and the 68230 PIACK pin can be connected to the NANDed output of 68000 FC2 FC1 FC0 pins. Since FC2 FC1 FC0 = 111₂ indicates interrupt acknowledge, PIACK is asserted when the 68000 is ready to service the interrupt. The DMAREQ pin provides an active low direct memory access controller request pulse for three clock cycles, compatible with MC68450 DMA controller chip.

Tables 5.15 provides the 68230 register addressing assignments.

The 68230 ports can be configured for various modes of operation. For example, consider ports A and B. Bits 6 and 7 of the port general control register, PGCR (R0) are used for configuring ports A and B in one of four modes as follows:

PGCR bits		
7	6	
0	0	Mode 0 (unidirectional 8-bit mode)
0	1	Mode 1 (unidirectional 16-bit mode)
1	0	Mode 2 (bidirectional 8-bit mode)
1	1	Mode 3 (bidirectional 16-bit mode)

The other pins of PGCR are defined as follows:

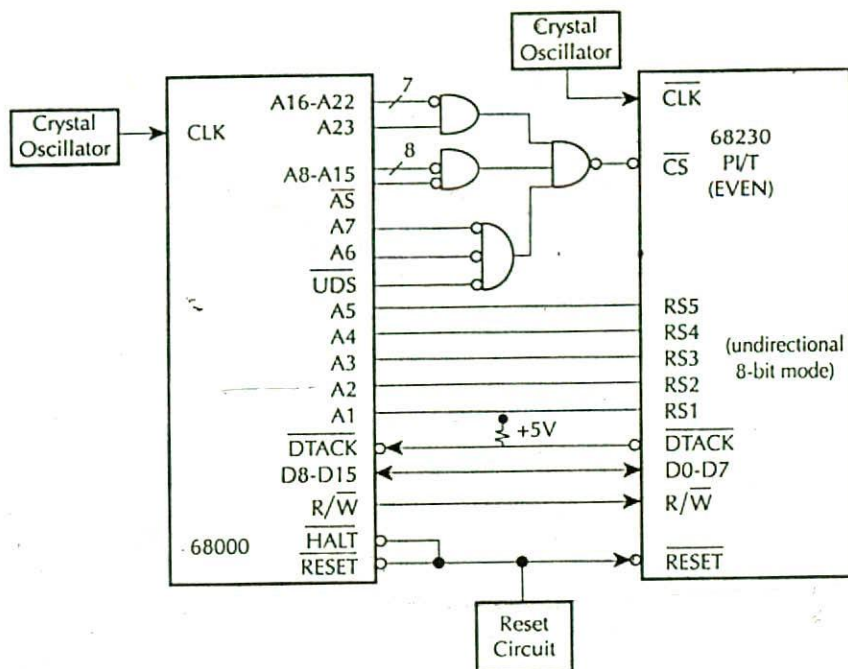


FIGURE 5.18 68000-68230 interface.

address for PADDR = \$800004
 address for PBDDR = \$800006
 address for PACR = \$80000C
 address for PBCR = \$80000E
 address for PADR = \$800010
 address for PBDR = \$800012

As an example, the following instruction sequence will select mode 00, submode 1X and configure bits 0-3 of port A as outputs, bits 4-7 as inputs, and port B as an output port:

```

PGCR    EQU $800000
PADDR   EQU $800004
PBDDR   EQU $800006
PACR    EQU $80000C
PBCR    EQU $80000E

MOVE.B  #$00, PGCR    ; Select mode 0
MOVE.B  #$0FF, PACR   ; Port A bit I/O submode
MOVE.B  #$0FF, PBCR   ; Port B bit I/O submode
MOVE.B  #$0F, PADDR   ; Configure Port A bits 0-3 as
                        ; outputs and bits 4-7 as inputs
MOVE.B  #$0FF, PBDDR  ; Configure Port B as an output
                        ; port.
  
```

5.11.2 Motorola 68000-6821 Interface

The Motorola 6821 is a 40-pin peripheral interface adapter (PIA) chip. It is provided with an 8-bit bidirectional data bus (D0-D7), two register select lines (RS0, RS1), read/write line

MOVE.B Dn, (EA) Transfer 8-bit data from the specified data register Dn to an output port.

Figure 5.19 shows a block diagram of how two 6821s are interfaced to the 68000 in order to generate four 8-bit I/O ports.

In Figure 5.19, I/O port addresses can be obtained as follows. When A23 is HIGH and AS is LOW, the OR gate output will be LOW. This OR gate output is used to provide \overline{PA} . The inverted OR gate output, in turn, makes CS1 HIGH on both the chips. Note that A23 is arbitrarily chosen. A23 is chosen to be HIGH to enable CS1 so that the addresses for the ports and the reset vector are not the same. Assuming the don't care address lines A22-A3 to be zeros, the addresses for the I/O ports, control registers, and the data direction registers for the even 6821 can be obtained as follows:

Port name	Memory address							
	A23	A22	...	A3	A2	A1	A0	
I/O port A/DDRA	1	0	...	0	0	0	0	= 800000 ₁₆
CRA	1	0	...	0	0	1	0	= 800002 ₁₆
I/O port B/DDRBB	1	0	...	0	1	0	0	= 800004 ₁₆
CRB	1	0	...	0	1	1	0	= 800006 ₁₆

Note that in the above, A0 = 0 for even addressing. Also, from Table 5.16, for accessing DDRA, bit 2 of CRA with memory address 800002₁₆ must be set to 1 and then port A can be configured with appropriate data in 800000₁₆. Note that port A and its data direction register, DDRA, have the same address, 800000₁₆. Similarly, port B and DDRB have the same address 800004₁₆. Bit 2 in CRA or CRB identifies whether address 800000₁₆ or 800004₁₆ is an I/O port or data direction register.

Similarly, the addresses for the ports, control registers, and the data direction register for the odd 6821 (A0 = 1) can be determined as follows: port A/DDRA (800001₁₆), CRA (800003₁₆), port B/DDRBB (800005₁₆), and CRB (800007₁₆).

5.12 68000/2716/6116/6821-Based Microcomputer

Figure 5.20a shows the schematic of a 68000-based microcomputer with 4K EPROM, 4K Static RAM, and four 8-bit I/O ports.

Let us explain the various sections of the hardware schematic. Two 2716-1 and two 6116 chips are required to obtain the 4K EPROM and 4K RAM. The LDS and UDS pins are ORed with the memory select signal to enable the chip selects for the EPROMs and the RAMs.

Address decoding is accomplished by using a 3 × 8 decoder. The decoder enables the memory or I/O chips depending on the status of A12-A14 address lines and AS line of the 68000. AS is used to enable the decoder. I0 selects the EPROMs, I1 selects the RAMs, and I2 selects the I/O ports.

When addressing memory chips, DTACK input of the 68000 must be asserted for data acknowledge. The 68000 clock in the hardware schematic is 8 MHz. Therefore, each clock cycle is 125 nanoseconds. In Figure 5.20a, AS is used to enable the 3 × 8 decoder. The outputs of the decoder are gated to assert 68000 DTACK. This means that AS is indirectly used to assert DTACK. From the 68000 read timing diagram of Figure 5.15, AS goes to LOW after approximately two cycles (250 ns for 8-MHz clock) from the beginning of the bus cycle. With no wait states, the 68000 samples DTACK at the falling edge of S4 (375 ns) and, if recognized, the 68000 latches data at the falling edge of S6 (500 ns). If the DTACK is not recognized at the falling edge of S4, the 68000 inserts one cycle (125 ns in this case) wait state, samples DTACK at the end of SW (wait state), and, if recognized, latches data at the end of S6 (625 ns),

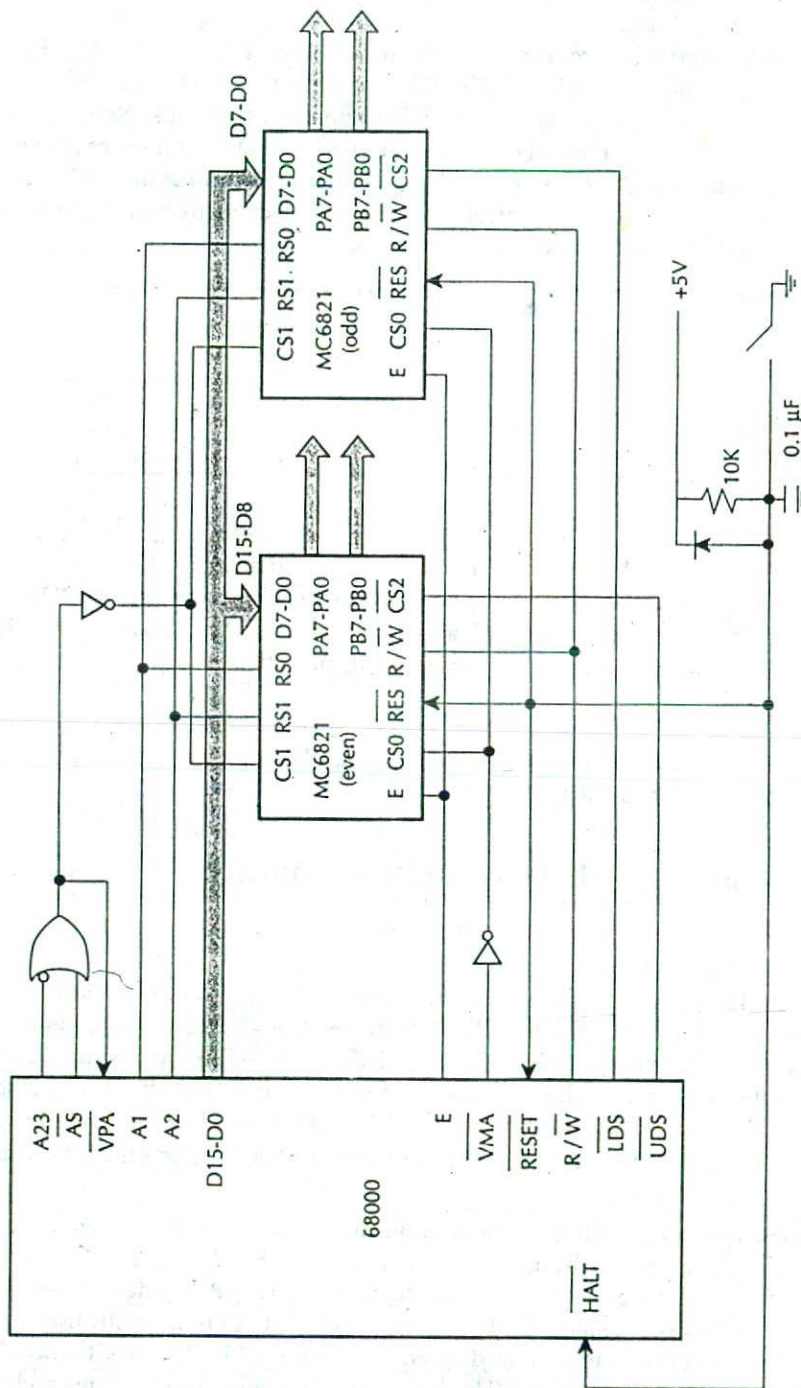


FIGURE 5.19 68000 I/O port block diagram.

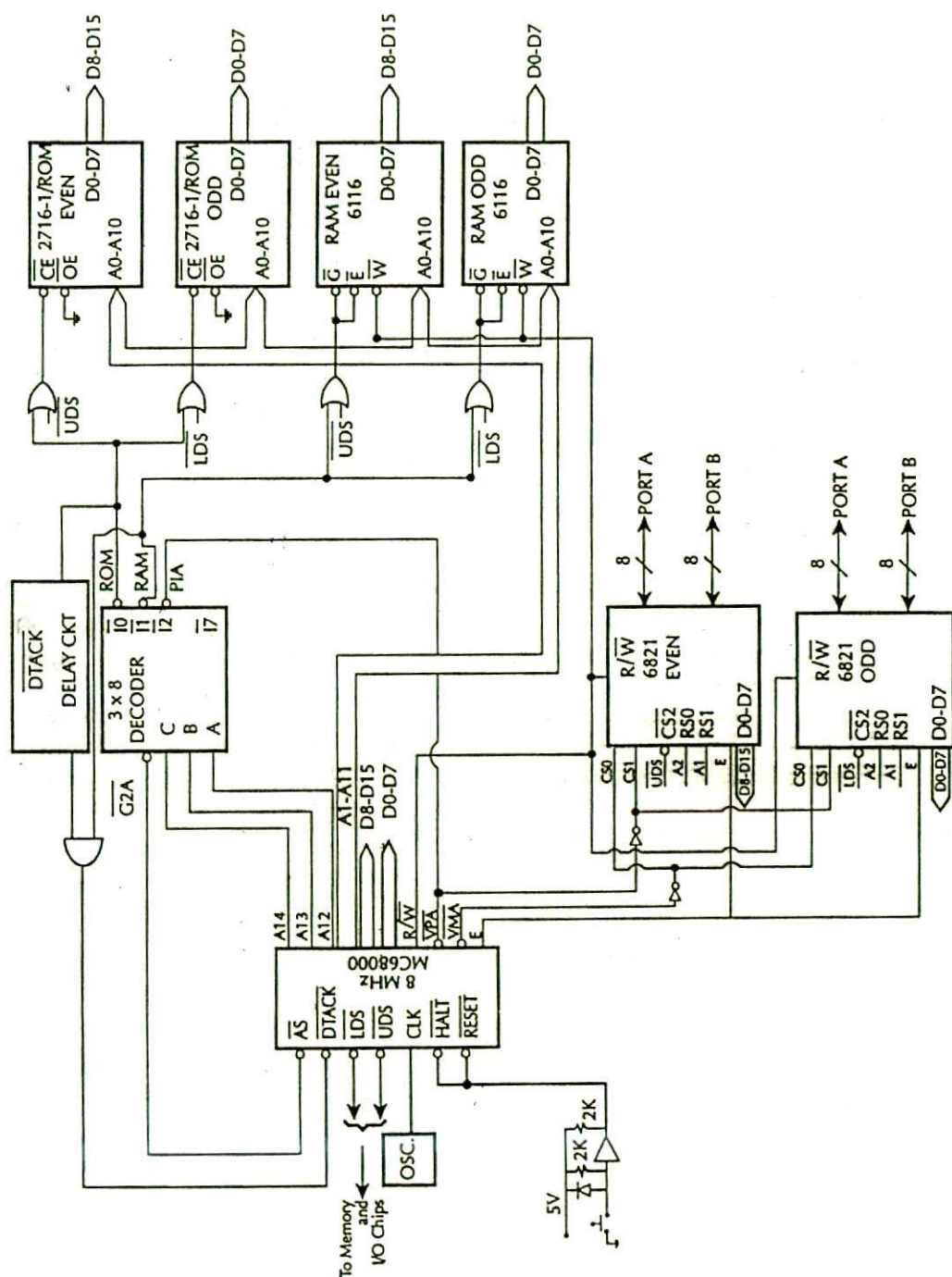
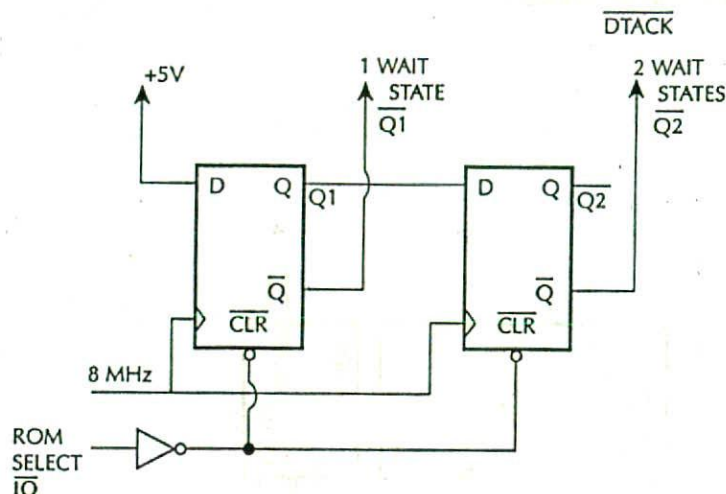


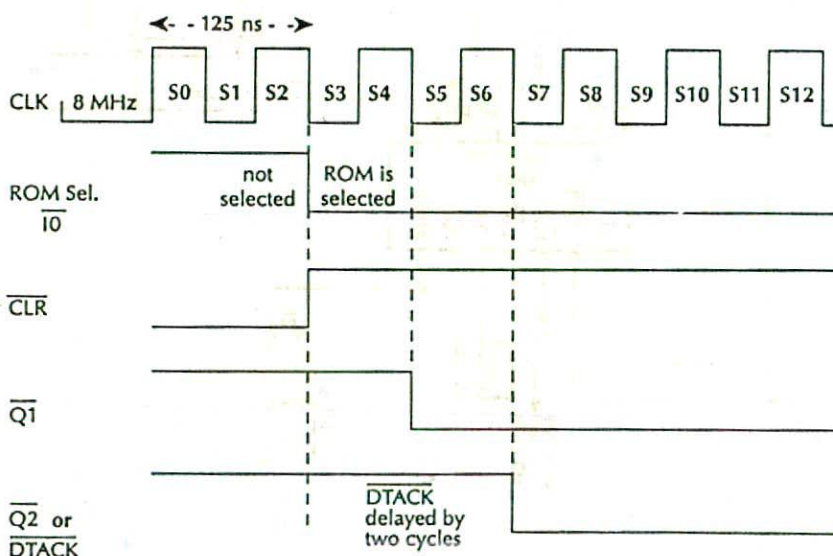
FIGURE 5.20a 68000-based microcomputer.

FIGURE 5.20b Delay circuit for \overline{DTACK} .

and the process continues. Since the access time of 2716-1 is 350 ns, \overline{DTACK} recognition by the 68000 at the falling edge of S6 (wait state) (500 ns) and, hence, latching of data at the falling edge of S8 (625 ns) will satisfy the timing requirement. This means that the decoder output $\overline{I0}$ for ROM select (delayed by approximately 250 ns by AS) must drive \overline{DTACK} Low at the end of S6. Therefore, $\overline{I0}$ must be delayed by 250 ns, i.e., two cycles.

A delay circuit (Figure 5.20b) is designed using a 74LS175-D-Flip-Flop. AS activates the delay circuit. The input is then shifted right two bits to obtain a two-cycle wait state to allow sufficient time for data transfer. \overline{DTACK} assertion and recognition are delayed by two cycles during data transfer with the EPROMs. A timing diagram for the \overline{DTACK} delay circuit is shown in Figure 5.20c.

When the ROM is not selected by the decoder, then clear pin is asserted (output of inverter). So, Q is forced LOW and Q is high. Therefore, \overline{DTACK} is not asserted. When the processor is addressing the ROMs, then the output of the inverter is HIGH so the clear pin is not asserted.

FIGURE 5.20c Timing diagram for the \overline{DTACK} delay circuit.

The D flip-flop will accept a high at the input, and Q2 will output high and Q2 will output low. Now that Q2 is low, it can assert DTACK. Q1 will provide one wait cycle and Q2 will provide two wait cycles. Since the 2716-1 EPROM has a 350-ns access time and the microprocessor is operating at 8 MHz (125 ns clock cycle), two wait cycles are inserted before asserting DTACK ($2 \times 125 = 250$ ns). Therefore, Q2 can be connected to the DTACK pin through an AND function.

No wait state is required for RAMs since the access time for the RAMs is only 120 nanoseconds.

Four 8-bit I/O ports are obtained by using two 6821 chips. When the I/O ports are selected, the VPA pin is asserted instead of DTACK. This will acknowledge to the 68000 that it is addressing a 6800-type peripheral. In response, the 68000 will synchronize all data transfer with the E clock.

The memory and I/O maps for the schematic are shown below:

Memory Mapping

						$\overline{\text{LDS}}$ or $\overline{\text{UDS}}$
A23—A15	A14	A13	A12	A11—A1	A0	
0—0	0	0	0	0—0	0	ROM (EVEN) = 2K
0—0	0	0	0	1—1	0	\$000000, \$000002, \$000004, ..., \$000FFE.
0—0	0	0	0	0—0	1	ROM (ODD) = 2K
0—0	0	0	0	1—1	1	\$000001, \$000003, \$000005, ..., \$000FFF
0—0	0	0	1	0—0	0	RAM (even) = 2K
0—0	0	0	1	1—1	0	\$001000, \$001002, ..., \$001FFE
0—0	0	0	1	0—0	1	RAM (odd) = 2K
0—0	0	0	1	1—1	1	\$001001, \$001003, ..., \$001FFF

Memory Mapped I/O

								$\overline{\text{LDS}}$ or $\overline{\text{UDS}}$
A23—A15	A14	A13	A12	A11—A3	A2	A1	A0	Register Selected (Address) EVEN
0—0	0	1	0	0—0	0	0	0	Port A or DDRA = \$002000
0—0	0	1	0	0—0	0	1	0	CRA = \$002002
0—0	0	1	0	0—0	1	0	0	Port B or DDRB = \$002004
0—0	0	1	0	0—0	1	1	0	CRB = \$002006
0—0	0	1	0	0—0	0	0	1	ODD
0—0	0	1	0	0—0	0	1	1	Port A or DDRA = \$002001
0—0	0	1	0	0—0	0	1	1	CRA = \$002003
0—0	0	1	0	0—0	1	0	1	Port B or DDRB = \$002005
0—0	0	1	0	0—0	1	1	1	CRB = \$002007

Note that upon hardware reset, the 68000 loads the supervisor SP high and low words, respectively, from addresses \$000000 and \$000002 and the PC high and low words, respectively, from locations \$000004 and \$000006. The memory map of Figure 5.20a contains these reset vector addresses in the even and odd 2716s.

Example 5.13

Assume that in the configuration of Figure 5.18, port A has three inputs and an LED connected to bits 0-3. Port B has an LED connected to bit 3. Write 68000 assembly program to

1. Turn the port A LED ON and port B LED OFF if port A has an even number of high switch inputs.
2. Turn the port A LED OFF and port B LED ON if port A has an odd number of high switch inputs.
3. Turn both LEDs off if there are no high switch inputs.

Assume EPROM/RAM in the system.

Solution

```
PGCR    EQU    $800000
PADDR    EQU    $800004
PBDDR    EQU    $800006
PACR    EQU    $80000C
PBCR    EQU    $80000E
PADR    EQU    $800010
PBDR    EQU    $800012

MOVE.B    #$0,    PGCR    ; Select mode 0
MOVE.B    #$0FF,  PACR    ; Port A bit I/O submode
MOVE.B    #$0FF,  PBCR    ; Port B bit I/O submode
MOVE.B    #$08,   PADDR   ; Configure port A
MOVE.B    #$08,   PBDDR   ; Configure port B
MOVE.B    PADR,    D1      ; Get port A switches
ANDI.B    #$07,    D1      ; Mask high five bits
                                ; Are all three inputs low
BEQ LOW                                ; If so, turn both LEDs off.
CMPI.B    #$03,    D1      ; Are high switch inputs even
BEQ EVEN                                ; If so, turn port A LED ON
                                ; and port B LED OFF.
CMPI.B    #$05,    D1      ; Are high switch inputs even
BEQ EVEN                                ; If so, turn port A LED ON
                                ; and port B LED OFF
CMPI.B    #$06,    D1      ; Are high switch inputs even
BEQ EVEN                                ; If so, turn port A LED ON
                                ; and port B LED OFF.
CMPI.B    #$07,    D1      ; Are high switch inputs ODD
BEQ ODD                                ; If so, turn port A LED OFF
                                ; and port B LED ON
CMPI.B    #$04,    D1      ; Are high switch inputs even
BEQ ODD                                ; If so, turn port A LED ON
                                ; and port B LED OFF
CMPI.B    $01,    D1      ; Are high switch inputs ODD
BEQ ODD                                ; If so, turn port A LED OFF
                                ; and port B LED ON
CMPI.B    $02,    D1      ; Are high switch inputs ODD
BEQ ODD                                ; If so, turn port A LED OFF
                                ; and port B LED ON
BRA LOW                                ; and port B LED ON
ODD      MOVE.B    #$00,  PADR    ; Turn port A LED OFF
          MOVE.B    #$08,  PBDR    ; Turn port B LED ON
```

```

        JMP FINISH                ; Halt
EVEN    MOVE.B #$08, PADR        ; Turn port A LED ON
        MOVE.B #$00, PBDR        ; Turn port B LED OFF
        JMP FINISH                ; Halt
LOW     MOVE.B #$00, PADR        ; Turn port A LED OFF
        MOVE.B #$00, PBDR        ; Turn port B LED OFF
FINISH  JMP FINISH                ; Halt

```

5.13 68000 Interrupt I/O

The 68000 services interrupts in the supervisor mode. The 68000 interrupt I/O can be divided into two types: external interrupts and internal interrupts.

5.13.1 External Interrupts

The 68000 provides seven levels of external interrupts, 1 through 7. The external hardware provides an interrupt level using the pins IPL0, IPL1, IPL2. Like other processors, the 68000 checks for and accepts interrupts only between instructions. It compares the value of inverted IPL0-IPL2 with the current interrupt mask contained in the bits 10, 9, and 8 of the status register.

If the value of the inverted IPL0-IPL2 is greater than the value of the current interrupt mask, then the processor acknowledges the interrupt and initiates interrupt processing. Otherwise, the 68000 continues with the only current interrupt level if pending. Interrupt request level zero (IPL0-IPL2 all HIGH) indicates that no interrupt service is requested. An inverted IPL2, IPL1, IPL0 of 7 is always acknowledged and has the highest priority. Therefore, interrupt level 7 is "nonmaskable". Note that the interrupt level is indicated by the interrupt input pins (inverted IPL2, IPL1, IPL0).

To ensure that an interrupt will be recognized, the following interrupt rules should be considered:

1. The incoming interrupt request level must be at a higher priority level than the mask level set in the interrupt mask bits (except for level 7, which is always recognized).
2. The IPL0-IPL2 pins must be held at the interrupt request level until the 68000 acknowledges the interrupt by initiating an interrupt acknowledge (IACK) bus cycle.

Interrupt level 7 is edge-triggered. On the other hand, the interrupt levels 1 to 6 are level sensitive. But as soon as the status register is saved the processor updates its interrupt mask to the same level.

The 68000 does not have any EI (Enable Interrupt) or DI (Disable Interrupt) instructions. Instead, the level indicated by I2 I1 I0 in the SR disables all interrupts below or equal to this value and enables all interrupts above this. For example, in the supervisor mode, I2, I1, and I0 can be modified by using instructions such as AND with SR. If I2, I1, and I0 are modified to contain 100₂, then interrupt levels 1 to 4 are disabled and levels 5 to 7 are enabled. Note that I2 I1 I0 = 111 disables all interrupts except level 7.

Upon hardware reset, the 68000 operates in supervisor mode and sets I2 I1 I0 to 111₂ and disables the interrupt levels 1 through 6. Note that if I2 I1 I0 is modified to 110₂, the 68000 also disables levels 1 through 6 and level 7 is, of course, always enabled.

Once the 68000 has decided to acknowledge an interrupt request, it pushes PC and SR onto the stack, enters supervisor state by setting S-bit to 1, clears TF to inhibit tracing, and updates the priority mask bits and also the address lines A3-A1 with the interrupt level. The 68000 then asserts AS to inform the external devices that A3-A1 has the interrupt level. The processor sets FC2 FC1 FC0 to 111 to run an IACK cycle for 8-bit vector number acquisition. The 68000

multiplies the 8-bit vector by 4 to determine the pointer to locations containing the starting address of the service routine. The 68000 then branches to the service routine. The last instruction of the service routine should be RTE which pops PC and SR back from the stack. In order to explain how the 68000 interrupt priorities work, assume that I2 I1 I0 in SR has the value 011₂. This means that levels 1, 2, and 3 are disabled and levels 4, 5, 6, and 7 are enabled. Now, if the 68000 is interrupted with level 5, the 68000 pushes PC and SR, updates I2 I1 I0 in SR with 101₂, and loads PC with the starting address of the service routine.

Now, while in the service routine of level 5, if the 68000 is interrupted by level 6 interrupt, the 68000 pushes PC and SR onto the stack. The 68000 then completes execution of the level 6 interrupt. The RTE instruction at the end of the level 6 service routine pops old PC and old SR and returns control to the level 5 interrupt service routine at the right place and continues with the level 5 service routine.

External logic can respond to the interrupt acknowledge in one of the following ways: by requesting automatic vectoring or by placing a vector number on the data bus (nonautovector), or by indicating that no device is responding (Spurious Interrupt). If the hardware asserts VPA to terminate the IACK bus cycle, the 68000 directs itself automatically to the proper interrupt vector corresponding to the current interrupt level. No external hardware is required for providing interrupt address vector. This is known as autovectoring. The vectors for the seven autovector levels are given below:

	I2	I1	I0
Level 1 ← Interrupt vector \$19 for	0	0	1
Level 2 ← Interrupt vector \$1A for	0	1	0
Level 3 ← Interrupt vector \$1B for	0	1	1
Level 4 ← Interrupt vector \$1C for	1	0	0
Level 5 ← Interrupt vector \$1D for	1	0	1
Level 6 ← Interrupt vector \$1E for	1	1	0
Level 7 ← Interrupt vector \$1F for	1	1	1

During autovectoring, the 68000 asserts $\overline{\text{VMA}}$ after assertion of $\overline{\text{VPA}}$ and then completes a normal 68000 read cycle.

In a nonautovector situation, the interrupting device uses external hardware to place a vector number on data lines D0-D7, and then performs a DTACK handshake to terminate the IACK bus cycle. The vector numbers allowed are \$40 to \$FF, but Motorola has not implemented a protection on the first 64 entries so that user-interrupt vectors may overlap at the discretion of the system designer. The 68000 multiplies this vector by 4 and determines the pointers to an interrupt address vector.

During the IACK cycle, the 68000 always checks the $\overline{\text{VPA}}$ line for LOW, and if $\overline{\text{VPA}}$ is asserted, the 68000 obtains the interrupt vector address using autovectoring. If $\overline{\text{VPA}}$ is not asserted, the 68000 checks DTACK for LOW. If DTACK is asserted, the 68000 obtains the interrupt address vector using nonautovectoring.

Another way to terminate an interrupt acknowledge bus cycle is with the BERR (Bus Error) signal. Even though the interrupt control pins are synchronized to enhance noise immunity, it is possible that external system interrupt circuitry may initiate an IACK bus cycle as a result of noise. Since no device is requesting interrupt service, neither DTACK nor $\overline{\text{VPA}}$ will be asserted to signal the end of the nonexistent IACK bus cycle. When there is no response to an IACK bus cycle after a specified period of time (monitored by the user by an external timer), the BERR can be asserted. This indicates to the processor that it has recognized a spurious interrupt. The 68000 provides 18H as the vector to fetch for the starting address of this exception handling routine.

The 68000 determines the interrupt address vector for each of the above cases as follows. After obtaining the 8-bit vector n , the 68000 reads the long word located at memory address

Vector Number(s)	Address			Assignment
	Dec	Hex	Space	
0	0	000	SP	Reset Initial SSP
-	4	004	SP	Reset Initial PC
2	8	008	SD	Bus Error
3	12	00C	SD	Address Error
4	16	010	SD	Illegal Instruction
5	20	014	SD	Zero Divide
6	24	018	SD	CHK Instruction
7	28	01C	SD	TRAPV Instruction
8	32	020	SD	Privilege Violation
9	36	024	SD	Trace
10	40	028	SD	Line 1010 Emulator
11	44	02C	SD	Line 1111 Emulator
12*	48	030	SD	(Unassigned, Reserved)
13*	52	034	SD	(Unassigned, Reserved)
14*	56	038	SD	(Unassigned, Reserved)
15	60	03C	SD	Uninitialized Interrupt Vector
16-23*	64	04C	SD	(Unassigned, Reserved)
	95	05F		-
24	96	060	SD	Spurious Interrupt
25	100	064	SD	Level 1 Interrupt Autovector
26	104	068	SD	Level 2 Interrupt Autovector
27	108	06C	SD	Level 3 Interrupt Autovector
28	112	070	SD	Level 4 Interrupt Autovector
29	116	074	SD	Level 5 Interrupt Autovector
30	120	078	SD	Level 6 Interrupt Autovector
31	124	07C	SD	Level 7 Interrupt Autovector
32-47	128	080	SD	TRAP Instruction Vectors
	191	08F		-
48-63*	192	0C0	SD	(Unassigned, Reserved)
	255	0FF		-
64-255	256	100	SD	User Interrupt Vectors
	1023	3FF		-

*Vector numbers 12, 13, 14, 16 through 23, and 48 through 63 are reserved for future enhancements by Motorola. No user peripheral devices should be assigned to these numbers.

FIGURE 5.21 68000 exception map. 'SP' means supervisor program space; 'SD' means supervisor data space.

4*n. This long word contains the address of the service routine. Therefore, the address is found using indirect addressing. Note that the spurious interrupt and bus error interrupt due to troubled instruction (when no DTACK is received by the 68000) have two different vectors. Spurious interrupt occurs when the BUSERROR pin is asserted during interrupt processing.

During IACK cycle, FC2 FC1 FC0 = 111. A1-A3 has the interrupt level. The vector number is provided on the D0-D7 pins by external hardware. Note that during nonautovectoring, if VPA goes to LOW, the 68000 ignores it.

5.13.2 Internal Interrupts

The internal interrupt is a software interrupt. This interrupt is generated when the 68000 executes a software interrupt instruction called TRAP or by some undesirable event such as division by zero or execution of an illegal instruction.

5.13.3 68000 Exception Map

Figure 5.21 shows an interrupt map of the 68000. Vector addresses \$00 through \$2C include vector addresses for reset, bus error, trace, divide by 0, etc., and addresses \$30 through \$4C are

unassigned. The RESET vector requires four words (addresses 0, 2, 4, and 6) and other vectors require only two words. As an example of how the 68000 determines the interrupt address vector, consider autovector 1. After VPA is asserted, if $IPL2-0 = 110$ (level = 001), the 68000 automatically obtains the 8-bit vector 25_{10} (19_{16}) and multiplies 19_{16} by 4 to obtain the 24-bit address 000064_{16} . The 68000 then loads the 16-bit contents of location 000064_{16} and the next location 000066_{16} into PC. For example, if the user wants to write the service routine for autovector 1 at address 271452_{16} , then $XX27_{16}$ and 1452_{16} , must, respectively, be stored at 000064_{16} and 000066_{16} . Note that XX in $XX27_{16}$ are two don't care nibbles (4 bits).

After hardware reset, the 68000 loads the supervisor SP high and low words, respectively, from addresses 000000_{16} and 000002_{16} , and the PC high and low words, respectively, from 000004_{16} and 000006_{16} . The assembler directive, Define Constant (DC) can be used to load PC and SSP. For example, the following instruction sequence loads SSP with $\$004100$ and PC with $\$001000$:

```
ORG    $000000
DC.L   $00004100
DC.L   $00001000
```

5.13.4 68000 Interrupt Address Vector

Suppose that the user decides to write a service routine starting at address $\$123456$ using autovector 1. Since the autovector 1 uses addresses $\$000064$ and $\$000066$, the numbers $\$0012$ and $\$3456$ must be stored in locations $\$000064$ and $\$000066$, respectively. The DC.L assembler directive can be used to load $\$123456$ into location $\$000064$ as follows:

```
ORG    $000064
DC.L   $00123456
```

5.13.5 An Example of Autovector and Nonautovector Interrupts

As an example to illustrate the concept of autovector and nonautovector interrupts, consider Figure 5.22. In this figure, I/O device 1 uses nonautovector and I/O device 2 uses autovector interrupts. The system is capable of handling interrupts from eight devices, since an 8-to-3 priority encoder such as the 74LS148 is used. Suppose that I/O device 2 drives the I/O2 LOW in order to activate line 3 of this encoder. This, in turn, interrupts the processor. When the 68000 decides to acknowledge the interrupt, it drives $FC0-FC2$ HIGH. The interrupt level is reflected on A1-A3 when AS is activated by the 68000. IACK3 and I/O2 signals are used to generate VPA. Once the VPA is asserted, the 68000 obtains the interrupt vector address using autovectoring.

In case of I/O1, line 5 of the priority encoder is activated to initiate the interrupt. By using appropriate logic, DTACK is asserted using IACK5 and I/O1. The vector number is placed on D0-D7 by enabling an octal buffer such as the 74LS244 using IACK5. The 68000 inputs this vector number and multiplies it by 4 to obtain the interrupt address vector.

5.14 68000 DMA

Three DMA control lines are provided with the 68000. These are BR (Bus Request), BG (Bus Grant), and BGACK (Bus Grant Acknowledge).

The BR line is an input to the 68000. The external device activates this line to tell the 68000 to release the system bus.

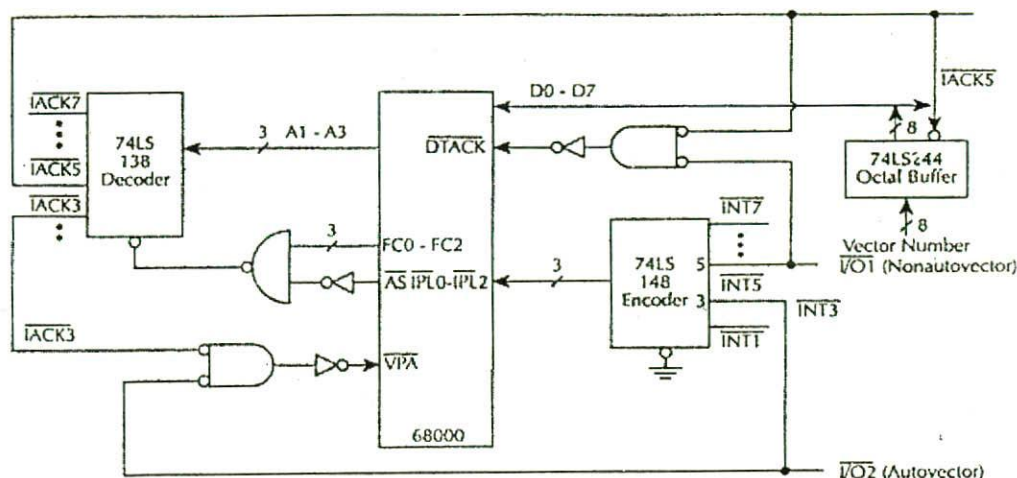


FIGURE 5.22 Autovector and nonautovector interrupts.

At least one clock period after receiving the \overline{BR} , the 68000 will enable its \overline{BG} output line to acknowledge the DMA request. However, the 68000 will not relinquish the bus until it has completed the current instruction cycle. The external device must check \overline{AS} (Address Strobe) line to determine the completion of instruction cycle by the 68000. When \overline{AS} becomes HIGH, the 68000 will tristate its address and data lines and will give up the bus to the external device.

After taking over the bus, the external device must enable \overline{BGACK} line. The \overline{BGACK} line tells the 68000 and other devices connected to the bus that the bus is being used. The 68000 buses stay in a tristate condition until \overline{BGACK} becomes HIGH.

5.15 68000 Exception Handling

A 16-bit microcomputer is usually capable of handling unusual or exceptional conditions. These conditions include situations such as execution of illegal instruction or division by zero and all interrupts. In this section, the exception handling capabilities of the MC68000 are described.

The 68000 exceptions can be divided into three groups, namely, groups 0, 1, and 2. Group 0 has the highest priority and group 2 has the lowest priority. Within the first two groups, there are additional priority levels. A list of 68000 exceptions along with individual priorities is shown below:

- Group 0 Reset (highest level in this group), Address Error (next level), and Bus Error (lowest level)
- Group 1 Trace (highest level), Interrupt (next level), Illegal op code (next level), and Privilege Violation (lowest level)
- Group 2 TRAP, TRAPV, CHK, and ZERO, DIVIDE (no individual priorities assigned in group 2)

Exceptions from group 0 always override an active exception from group 1 or group 2. Group 0 exception processing begins at the completion of the current bus cycle (two clock cycles). Note that the number of cycles required for a READ or WRITE operation is called a

bus cycle. This means that during an instruction fetch if there is a group 0 interrupt, the 68000 will complete the instruction fetch and then service the interrupt.

Group 1 exception processing begins at the completion of the current instruction.

Group 2 exceptions are initiated through execution of an instruction. Therefore, there are no individual priority levels within group 2. Exception processing occurs when a group 2 interrupt is encountered, provided there are no group 0 or group 1 interrupts.

When an exception occurs, the 68000 saves the contents of the program counter and status register onto the stack and then executes a new program whose address is provided by the exception vectors. Once this program is executed, the 68000 returns to the main program using the stored values of program counter and status register.

Exceptions can be of two types: internal or external.

The internal exceptions are generated by situations such as division by zero, execution of illegal or unimplemented instructions, and address error. As mentioned before, internal interrupts are called traps.

The external exceptions are generated by bus error, reset, or interrupts. The basic concepts associated with interrupts, relating them to the 68000, have already been described. In this section we will discuss the other exceptions.

In response to an exception condition, the processor executes a user-written program. In some microcomputers, one common program is provided for all exceptions. The beginning section of the program determines the cause of the exception and then branches to the appropriate routine. The 68000 utilizes a more general approach. Each exception can be handled by a separate program.

As mentioned before, the 68000 has two modes of operation: user mode and supervisor mode. The operating system runs in supervisor mode and all other programs are executed in user mode. The supervisor mode is, therefore, privileged. Several privileged instructions such as MOVE to SR can only be executed in supervisor mode. Any attempt to execute them in user mode causes a trap.

We will now discuss how the 68000 handles exceptions which are caused by external reset, instructions causing traps, bus and address errors, tracing, execution of privileged instructions in user mode, and execution of illegal/unimplemented instructions.

The reset exception is generated externally. In response to this exception, the 68000 automatically loads the initial starting address into the processor.

The 68000 has a TRAP instruction which always causes an exception. The operand for this instruction varies from 0 to 15. This means that there are 16 TRAP instructions. Each TRAP instruction is normally used to call subroutines in an operating system. Note that this automatically places the 68000 in supervisor state. TRAPs can also be used for inserting breakpoints in a program. Two other 68000 instructions cause traps if a particular condition is true. These are TRAPV and CHK. TRAPV generates an exception if the overflow flag is set. The TRAPV instruction can be inserted after every arithmetic operation in a program for causing a trap whenever there is the possibility of an overflow. A routine can be written at the vector address for the TRAPV to indicate to the user that an overflow has occurred. The CHK instruction is designed to ensure that access to an array in memory is within the range specified by the user. If there is a violation of this range, the 68000 generates an exception.

A bus error occurs when the 68000 tries to access an address which does not belong to the devices connected to the bus. This error can be detected by asserting the BERR pin on the 68000 chip by an external timer when no DTACK is received from the device after a certain period of time. In response to this, the 68000 executes a user-written routine located at an address obtained from the exception vectors. An address error, on the other hand, occurs when the 68000 tries to READ or WRITE a word (16-bit) or long word (32-bit) at an odd address. The address error has a different exception vector from the bus error.

The trace exception in the 68000 can be generated by setting the trace bit in the status register, in response to the trace exception after execution of every instruction. The user can write a routine at the exception vectors for the trace instruction to display registers and memory. The trace exception provides the 68000 with the single-stepping debugging feature.

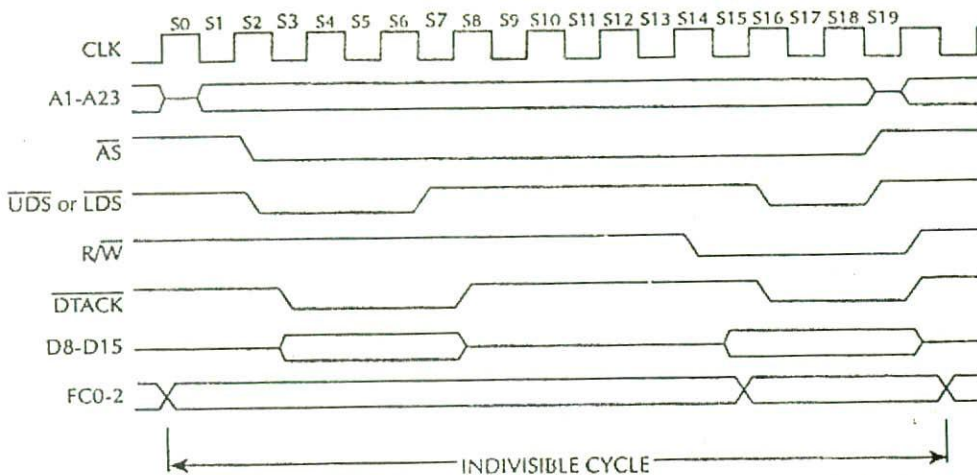


FIGURE 5.23 MC68000 read-modify-write cycle for TAS.

As mentioned before, the 68000 has some privileged instructions which must be executed in supervisor mode. An attempt to execute these instructions causes privilege violations. Finally, the 68000 causes an exception when it tries to execute an illegal or unimplemented instruction. It is common practice to use the illegal instruction \$4AFA as a breakpoint.

5.16 Multiprocessing with 68000 Using the TAS Instruction and AS (Address Strobe) Signal

Earlier, the 68000 TAS instruction was discussed. The TAS instruction supports the software aspects of interfacing two or more 68000s via shared RAM. When TAS is executed, an indivisible read-modify-write cycle is performed. The timing diagram for this specialized cycle is shown in Figure 5.23. During both the read and the write portions of the cycle, the AS remains LOW, and the cycle starts as the normal read cycle.

However, in the normal read, the AS going inactive indicates the end of the read. During execution of the TAS, the AS stays LOW throughout the cycle, and therefore AS can be used in the design of a bus locking circuit.

Due to bus locking, only one processor at a time can perform a TAS operation in a multiprocessor system. The TAS instruction supports semaphore operations (globally shared resources) by checking a resource for availability and reserving or locking it for use by a single processor. The TAS instruction can, therefore, be used to allocate memory space reservations.

The TAS instruction execution flow for allocating memory is shown in Figure 5.24a and b. The shared RAM of Figure 5.24b is divided into M sections. The first byte of each section will be pointed to by (EA) of TAS (EA) instruction. In the flowcharts, (ea) first points to the first byte of section 1. The instruction TAS (ea) is then executed.

The TAS instruction checks the most significant bit (N bit) in (EA). $N = 0$ indicates that the section 1 is free; $N = 1$ means section 1 is busy. If $N = 0$, then section 1 will be allocated for use. On the other hand, if $N = 1$, section 1 is busy; a program will be written to subtract one section length from (EA) to check the next section for availability. Also, (EA) must be checked with the value TASLOCM. If $(EA) < TASLOCM$, then no space is available for allocation. However, if $(EA) > TASLOCM$, TAS is executed and the availability of that section is determined.

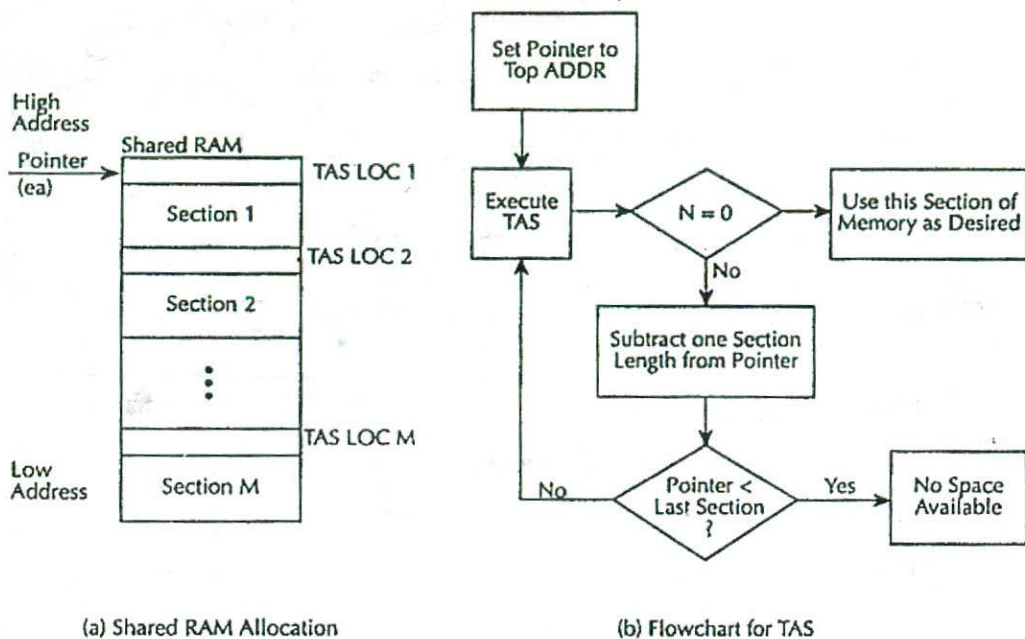
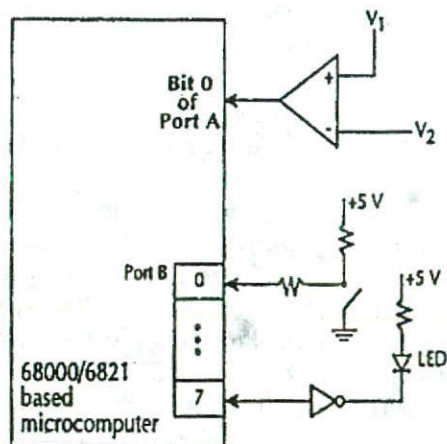


FIGURE 5.24 Memory allocation using TAS.

In a multiprocessor environment, the TAS instruction provides software support for interfacing two or more 68000 via shared RAM. The AS signal can be used to provide the bus locking mechanism.

Examples of 68000 programmed and interrupt I/O are provided below.

Example 5.14



i) In the above figure, the 68000/6821 based microcomputer is required to perform the following:

If $V_1 > V_2$, turn the LED ON if the switch is open. Write 68000 assembly language program to accomplish the above by inputting comparator output via bit 0 of Port A.

ii) Repeat part i) using autovector level 1 and nonautovector (vector \$40). Use port B for LED and switch as above. Assume supervisor mode.

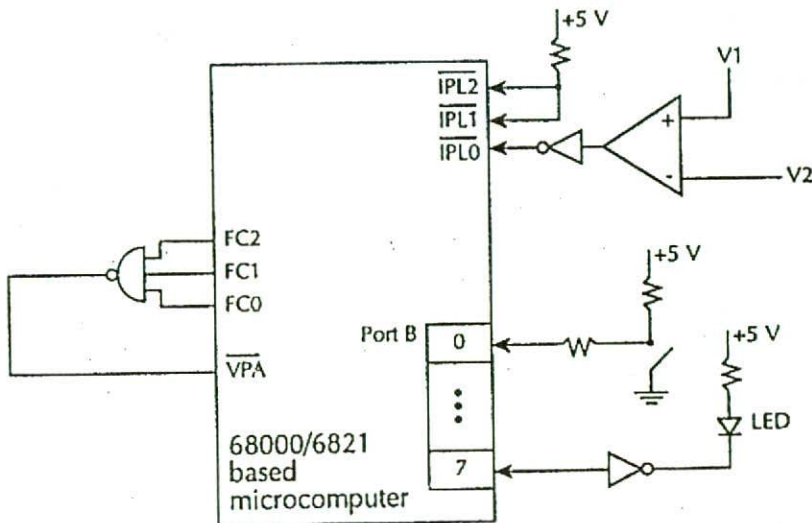
Write the main program and service routine in 68000 assembly language program respectively at addresses \$642F14 and \$251F20.

Also, initialize supervisor stack pointer at \$941760.

Solution

```
i)      BCLR.B #2,CRA      ; Address DDRA
        MOVE.B #0,DDRA    ; Configure Port A as input
        BSET.B #2,CRA      ; Address Port A
        BCLR.B #2,CRB      ; Address DDRB
        MOVE.B #$80,DDRB   ; Configure Port B
        BSET.B #2,CRB      ; Address Port B
START    MOVE.B PORTA,D0    ; Input Comparator
        ANDI.B #$01,D0     ; Check if high
        BEQ START
        MOVE.B PORTB,D1    ; Input switch
        ROXR.B #2,D1       ; Align switch status
        MOVE.B D1,PORTB    ; Output to LED
FINISH   JMP FINISH
```

ii) Using Autovectoring Level 1



Main Program

```
ORG $642F14
BCLR.B #2,CRB      ; Address DDRB
MOVE.B #$80,DDRB   ; Configure Port B
BSET.B #2,CRB      ; Address Port B
ANDI.W #$0F8FF,SR  ; Enable interrupts
WAIT    JMP WAIT    ; Wait for interrupts
FINISH  JMP FINISH  ; HALT
```



```

        BSET.B #2,CRB      ; Address Port B
        ANDI.W #0F8FF,SR   ; Enable interrupts
WAIT    JMP WAIT           ; Wait for interrupts
FINISH  JMP FINISH         ; Halt

```

Service Routine

```

        ORG $251F20
        MOVE.B PORTB,D0    ; Input switch
        ROXR.B #2,D0       ; Align switch
        MOVE.B D0,PORTB    ; Output to LED
        RTE

```

Reset Vector

```

        ORG 0
        DC.L $00941760
        DC.L $00642F14

```

Service Routine Vector

```

        ORG $000100
        DC.L $00251F20

```

Questions and Problems

5.1 Assume that $[D0] = 25774411_{16}$. What will be the contents of D0 after execution of each of the following instructions:

- i) CLR.B D0
- ii) CLR D0
- iii) CLR.L D0

5.2 Determine the contents of registers and the locations affected by each of the following instructions:

- i) MOVE.L $-(A2), (A3)+$

Assume the following data prior to execution of the MOVE:

```

        [A2] = $300504, [A3] = $510718,
        [$300500] = $01, [$3004FF] = $F1,
        [$3004FE] = $72, [$3004FD] = $A1,
        [$510718] = $53, [$510719] = $20,
        [$51071A] = $31, [51071B] = $27

```

- ii) MOVEA.W D1, A4

Assume the following data prior to execution of the MOVEA:

```

        [D1] = $37158470
        [A4] = $F1218234

```

- iii) MOVEA.L A2, A3

Assume the following data prior to execution of the MOVEA:

```

        [A2] = $1234F144
        [A3] = $20718714

```

5.3 Identify the following 68000 instructions as privileged or nonprivileged:

- i) MOVE SR, (A2)
- ii) MOVE CCR, (A0)

iii) **LEA.L (A2), A5**

iv) **MOVE.L A2, USP**

5.4 What are the contents of register D1 after execution of the following two instructions? Assume [D1] = \$34788480 prior to the execution of the instructions:

```
EXT.L D1
MOVEQ.L #$2F, D1
```

5.5 Find the contents of D1 after execution of the following DIVS instruction:

```
DIVS (A1), D1
```

Assume [A1] = \$205014, [\$205014] = \$FF, [\$205015] = \$FE, [D1] = \$00000005 prior to execution of the instruction. Identify the quotient and remainder of the result in D1. Comment on the sign of the remainder.

5.6 Write a 68000 assembly program to divide an 8-bit signed number in low byte of D1 by an 8-bit signed number in low byte of D2. Store quotient and remainder in D1.

5.7 Write a 68000 assembly language program to add two 128-bit numbers. Assume that the first number is stored in consecutive memory locations starting at \$605014. The second number is stored in consecutive memory locations starting at \$708020. Store the result in memory locations beginning at \$708020. Assume that all data storage to follow the conventional manner; that is highest byte to be stored as the lower address.

5.8 Write a 68000 assembly program to add top-two 32 bits of the stack. Store the 32-bit result onto the stack. Assume user mode.

5.9 Write a 68000 assembly program to multiply an 8-bit signed number in low byte of D1 by a 16-bit signed number in the high word of D5. Store the result in D3.

5.10 Write a 68000 assembly program to add twenty 32-bit numbers stored in consecutive memory locations starting at address \$502040. Store the 32-bit result onto the stack. Assume that for each 32-bit number, the lowest address stores the highest byte of the number.

5.11 Write 68000 assembly language to find the minimum value of a string of ten signed 16-bit numbers using indexed addressing.

5.12 Write a 68000 assembly program to compare two strings of twenty ASCII characters. The first string is stored starting at \$003000. The second string is stored starting at \$004000. The ASCII character in location \$003000 of string 1 will be compared with the ASCII character in location \$004000 of string 2, [\$003001] to be compared with [\$004001], and so on. Each time there is a match, store \$EEEE onto the stack; otherwise store \$0000.

5.13 Write a 68000 assembly program to divide a 27-bit unsigned number in high 27 bits of D0 by 16_{10} . Do not use any divide instruction. Neglect the remainder. Store quotient in the low 27-bits of D0.

5.14 Write a subroutine in 68000 assembly language to compute

$$Z = \sum_{i=1}^{100} (X_i - Y_i)$$

Assume that Xi's and Yi's are signed 16-bit and stored in consecutive locations starting at \$020054 and \$305116, respectively. Assume A0 and A1 point to Xi's and Yi's, respectively, and SP is already initialized. Also write the main program in 68000 assembly language to perform all initializations, call the subroutine, and then compute $Z/100$.

5.15 Write a subroutine in 68000 assembly language to subtract two unsigned eight-digit BCD numbers. The BCD number 1 is stored at location starting from \$300000 thru \$300003, with the least significant digit at \$300003 and the most significant digit at \$300000. Similarly, the BCD number 2 is stored at location starting from \$400000 through \$400003, with the least significant digit at \$400003 and the most significant digit at \$400000. The BCD number 2 is to be subtracted from BCD number 1. Store result in D1.

5.16 Write a subroutine in 68000 assembly to convert a 3-digit unsigned BCD number to binary. The most significant digit is stored in memory location starting at \$003000, the next digit is stored at \$003001, and so on. Store the binary result in D3.

Use the value of the 3-digit BCD number in $V = D2 \times 10^2 + D1 \times 10^1 + D0 = (D2 \times 10 + D1) \times 10 + D0$.

5.17 Write a recursive subroutine (A subroutine calling itself) in 68000 assembly language to find the factorial of an 8-bit number n by using $n! = n(n-1)(n-2) \dots 1$. Store the result in D0.

5.18 Determine the status of $\overline{\text{LDS}}$, $\overline{\text{UDS}}$, $\overline{\text{AS}}$, $\overline{\text{FC2-FC0}}$, and address lines immediately after execution of the following instruction sequence (before the 68000 tristates these lines to fetch the next instruction):

```
MOVE #$2000, SR
MOVE.B D2, $030001
—
—
```

Assume the 68000 is in supervisor mode prior to execution of the above instructions.

5.19 Write a 68000 assembly program to output the contents of memory locations \$003000 and \$003001 to two seven-segment displays connected to two 8-bit ports A and B of a 68000/6821 system. Assume that displays are connected to ports A and B the same as shown in the figure of problem 5-39. Note that only the connections between Port A and one seven-segment display is shown in the figure.

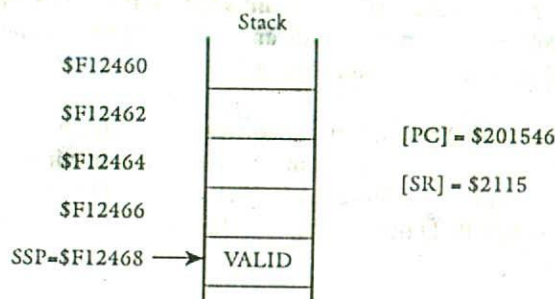
5.20 Assume that in the configuration of Figure 5.19, port A and port B each has three switches and an LED connected to bits 0 through 3. Write 68000 assembly program to

- Turn the port A LED ON and port B LED OFF if port A has an even and port B has an odd number of high switch inputs.
- Turn the port A LED OFF and port B LED ON if port A has an odd and port B has an even number of high switch inputs.
- Turn both LEDs ON if both ports A and B have even number of high switch inputs.
- Turn both LEDs OFF if both ports A and B have odd number of high switch inputs.

5.21 Interface a 68000 to 2716s, 6116s, and a 68230 to provide 4K EPROM, 4K RAM, and two 8-bit I/O ports. Draw a neat schematic and determine memory and I/O maps. Assume 16.67 MHz internal clock for the 68000.

5.22 If the $\overline{\text{IPL2}}$, $\overline{\text{IPL1}}$, $\overline{\text{IPL0}}$ pins are interrupted by an external device with the code 001₂, when the interrupt mask value 121110 is 3₁₀, will the interrupt be serviced immediately or ignored by the 68000?

- 5.23 Discuss briefly the various 68000 exceptions.
- 5.24 Write a service routine for reset in 68000 assembly language that will initialize all data and address registers to zero, supervisor SP to \$3F0728, user SP to \$1F0524, and then jump to \$001000.
- 5.25 Assume the following stack and register values before occurrence of an interrupt:



If an external device requests an interrupt by asserting the $\overline{\text{IPL2}}$ $\overline{\text{IPL1}}$ $\overline{\text{IPL0}}$ pins with the value 000, determine the contents of SSP and SR during interrupt and after execution of RTE at the end of the service routine of the interrupt. Draw the memory layouts showing where SSP points and the stack contents during and after the interrupt. Assume that the stack is not used by the service routine.

5.26 Suppose that two pumps (P1,P2) and two LEDs (L1,L2) are to be connected to a 68000-based microcomputer. Each pump has a 'pump running' output to indicate the ON/OFF status. The microcomputer runs the pumps via bits 2 and 3 of 8-bit port A. Two LEDs L1 (for P1) and L2 (for P2) are connected to bits 0 and 1 of 8-bit port B to indicate whether each pump is running. Assume that the pump can be turned ON by HIGH and turned OFF by LOW.

- Using programmed I/O, draw a block diagram and write a 68000 assembly program to accomplish the above.
- Using interrupt I/O, draw a block diagram. Write the main program and service routine in 68000 assembly language to accomplish the above. The main program will perform all initializations and then start the pumps.

5.27 Compare the basic features of the 68000 with those of 68008, 68010, and 68012.

5.28 Write a 68000 assembly language program to add a 32-bit number stored in D0 (bits 0 through 15 containing the high-order 16 bits of the number and bits 16 through 31 containing the low-order 16 bits) with another 32-bit number stored in D1 (bits 0 through 15 containing the low-order 16 bits of the number and bits 16 through 31 containing the high-order 16 bits). Store the result in D0.

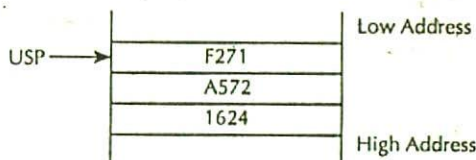
5.29 Write a subroutine in 68000 assembly language using the TAS instruction to find, reserve, and lock a memory segment for the main program. The memory is divided into four segments (0, 1, 2, 3) of 8 bytes each. The first byte of each segment includes a flagbyte to be used by the TAS instruction. In the subroutine, a maximum of four 8-byte memory segments must be checked for a free segment. Once a free segment is found, the TAS instruction is used to set the flagbyte. The starting address of the free segment must be stored in A5, and D5 must

be cleared to zero to indicate a free segment. If no free block is found, a nonzero value of all ones must be stored in D5.

5.30 What are the remainder and quotient, and which registers contain them after execution of the following instruction sequence?

```
CLR D0
MOVEQ #L, D1
MOVE #2, D2
DIVS D2, D1
```

5.31 Write a 68000 assembly language program to divide \$A5721624 by \$F271. Store the remainder and quotient onto the user stack. Assume that the numbers are signed and stored in the stack as follows:

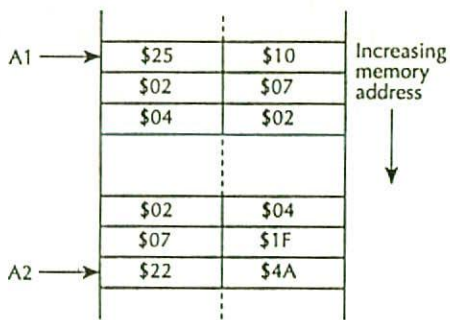


5.32 Write 68000 assembly language program to compute $X = Y + Z - \$30FE$ where X, Y, Z are 64-bit variables. The upper 32 bits of Y and Z are stored respectively in locations \$005000 and \$005008 followed by the lower 32 bits. Store the upper 32 bits of the 64-bit result at address \$006000 followed by the lower 32 bits.

5.33 Assume that registers D0, D1, and D2 contain a signed byte, a signed word, and a signed 32-bit number respectively. Write 68000 assembly language program that will compute the signed 32-bit result: $D0.B + D1.W - D2.L \rightarrow D2.L$.

5.34 Write 68000 assembly language program to compute $X = 5 * Y + (Z/W)$ where address \$005000, \$005002, and \$005004 store the 16-bit signed integers Y, Z and W. Store the 32-bit result in memory starting at address \$005006. Discard the remainder of Z/W.

5.35 Write a 68000 assembly language program to add two 48-bit data values in memory as shown in figure below:

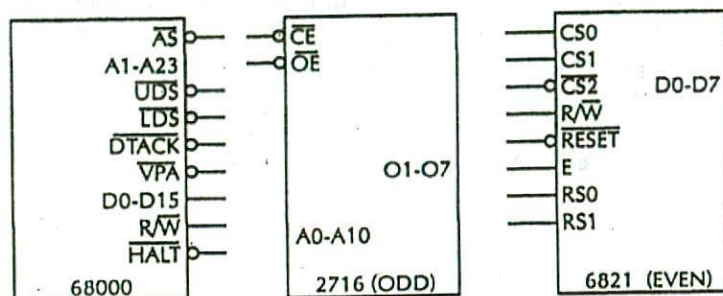


Store the result at the address pointed to by A2. The operation is given by:

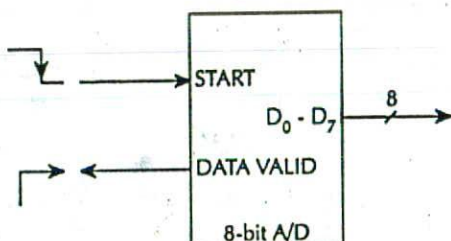
\$25	10	02	07	04	02
\$02	04	07	1F	22	4A
\$27	14	09	26	26	4C

Assume the data pointer and data are already initialized.

5.36 Assume the pins and signals for a 68000, 2716 (odd) and 6821 (even) shown in the figure below. Connect the chips and draw a neat schematic. Determine the memory and I/O maps. Assume a 16.67 MHz internal clock for the 68000.



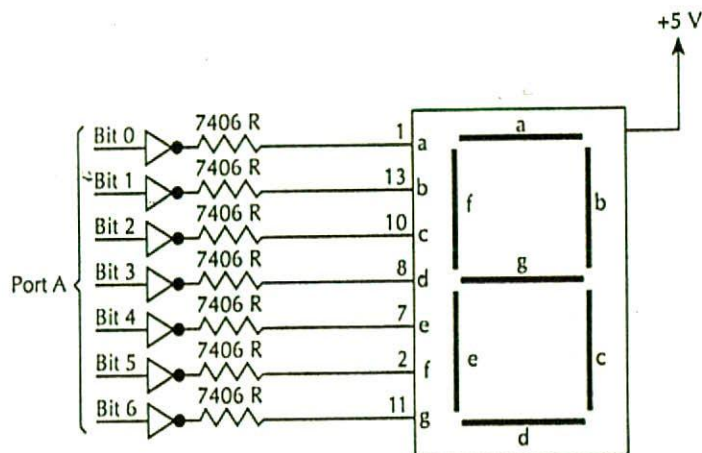
5.37 Assume the memory and I/O maps of Figure 5.20. Interface the following A/D to the 68000/2716-1/6116/6821 based microcomputer:



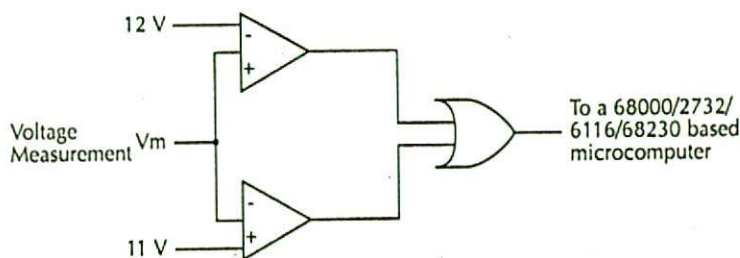
Write a 68000 assembly language program to input the A/D converter and turn ON an LED connected to bit 5 of Port A of even 6821 if the number read from A/D is ODD; otherwise turn the LED OFF. Assume that the LED is turned ON by a HIGH and turned OFF by a LOW.

5.38 Assume a 68000/6821 based system. Write a 68000 assembly program to input 16-bit data via Port A and Port B, and then divide this by the 8-bit data in the highest byte (bits 31-24) of D0. Assume all numbers to be signed.

5.39 A 68000/6821 based microcomputer is required to drive a common anode seven-segment display connected to Port A as follows:



Write 68000 assembly language program to display a single hexadecimal digit (0 to F) from address \$003000. Use a look up table.



5.40

In the above figure, if $V_m > 12V$, turn an LED ON connected at bit 2 of Port A. On the other hand, if $V_m < 11V$, turn the LED OFF. Use registers and memory locations of your choice. Draw a block diagram showing the microcomputer and the connection of the figure to its ports. Also, write 68000 assembly language programs to accomplish the above using:

- Polled I/O
- Autovector level 7
- Non-autovector (vector \$45)

1. The first part of the report is a general introduction to the subject of the study. It discusses the importance of the study and the objectives of the research.



The second part of the report is a detailed description of the methodology used in the study. It includes a description of the sample, the data collection methods, and the statistical analysis techniques used.

MOTOROLA MC68020

This chapter first describes in detail the hardware, software, and interfacing features associated with the MC68020.

Topics include MC68020 architecture, addressing modes, instruction set, I/O, coprocessors, and system design.

6.1 Introduction

The MC 68020 is Motorola's first 32-bit microprocessor. The MC68020 is designed to execute all user object code written for previous members of the MC 68000 family.

The MC68020 is manufactured using HCMOS (combining HMOS and CMOS on the same device). The MC68020 consumes a maximum of 1.75 watts. It contains 200,000 transistors on a $\frac{3}{8}$ " piece of silicon. The chip is packaged in a square ($1.345" \times 1.345"$) pin grid array (PGA) and contains 169 pins (114 pins used) arranged in a 13×13 matrix.

The processor speed of the MC68020 can be 12.5 MHz, 16.67 MHz, 20 MHz, 25 MHz, or 33 MHz. The chip must be operated from a minimum frequency of 8 MHz. Like the MC68000, it does not have any on-chip clock generation circuitry. The MC68020 contains 18 addressing modes and 101 instructions. All addressing modes and instructions of the MC68000 are included in the MC68020. The MC68020 supports coprocessors such as the MC68881/MC68882 floating-point and MC68851 memory management unit (MMU) coprocessors.

The following features of MC68020 are compared with those of MC68000:

Characteristic	68000	68020
Technology	HMOS	HCMOS
Size	$3\frac{1}{8}" \times 7\frac{7}{8}"$	$1.345" \times 1.345"$ (square size)
Number of pins	64, 68	169 (13×13 matrix; pins come out at the bottom of the chip; 114 pins currently used)
Control unit	Nanomemory (two-level control memory)	Nanomemory (two-level control memory)
Clock	6 MHz, 8 MHz, 10 MHz, 12.5 MHz, 16.67 MHz, 20 MHz, 25 MHz, or 33 MHz (no minimum requirements)	12.5 MHz, 16.67 MHz, 20 MHz, 25 MHz, or 33 MHz (must be 8 MHz minimum)
ALU	One 16-bit ALU	Three 32-bit ALUs
Address bus size	24 bits with A0 encoded from \overline{UDS} and \overline{LDS}	32 bits; no encoding of A0 required
Data bus size	Uses D0-D7 for odd addresses and D8-D15 for even addresses during byte transfers; for word and long word uses D0-D15	8, 16, and 32 bits (byte, word, long word transfers occur, respectively, via D24-D31 lines, D16-D31 lines, and D0-D31 lines)
Instruction and data access	All word and long word accesses must be at even addresses for both instructions and data; for byte, instruction must be at even	Instructions must be accessed at even addresses; data accesses can be at any address for byte, word, and long word

(continued)

Characteristic	68000	68020
Instruction and data access (continued)	addresses, and data can be at either odd or even addresses	
Instruction cache	None	128-entry 16-bit word cache; at the start of an instruction fetch, the 68020 always outputs LOW on the ECS (external cycle start) pin and accesses the cache; if the instruction is found in the cache, the 68020 inhibits outputting LOW on the AS pin; otherwise the 68020 sends LOW on the AS pin and reads the instruction from the main memory
Directly addressable memory	16 megabytes	4 gigabytes (4, 294, 964, 296 bytes)
Registers	8 32-bit data registers 7 32-bit address registers 2 32-bit SPs 1 32-bit PC (24 bits used) 1 16-bit SR	8 32-bit data registers 7 32-bit address registers 3 32-bit SPs 1 32-bit PC (all bits used) 1 16-bit SR 1 32-bit VBR (vector base register) 2 3-bit function code registers (SFC and DFC) 1 32-bit CAAR (cache address register) 1 32-bit CACR (cache control register)
Addressing modes	14	18
Instruction set	56 instructions	101 instructions
Stack pointers	USP, SSP	USP, MSP (master SP), ISP (interrupt SP)
Status register	T, S, I0, I1, I2, X, N, Z, V, C	T0, T1, S, M, I0, I1, I2, X, N, Z, V, C T1 T0 0 0 No tracing 0 1 Trace on jumps 1 0 Trace on instruction execution 1 1 Undefined S M 0 X USP (X is don't care; can be 0 or 1) 1 0 ISP 1 1 MSP
Coprocessor interface	Emulated in software; that is, by writing subroutines, coprocessor functions such as floating-point arithmetic can be obtained	Can directly be interfaced to coprocessor chips. Coprocessor functions, such as floating-point arithmetic can be obtained via 68020 instructions
FC2, FC0, FC1 pins	FC2, FC0, FC1 = 111 means interrupt acknowledge	FC2, FC0, FC1 = 111 means CPU space cycle and then by decoding A16-A19, one can obtain breakpoints, coprocessor functions, and interrupt acknowledge

Some of the 68020 characteristics tabulated above will now be explained:

- The three independent ALUs are provided for data manipulation and address calculations.
- A 32-bit barrel shift register (occupies 7% of silicon) is included in the 68020 for very fast shift operations regardless of the shift count.
- The 68020 has three SPs. In the supervisor mode (when S = 1), two SPs can be accessed. These are MSP (when M = 1) and ISP (when M = 0). The ISP can be used to simplify and speed up task switching for operating systems. The 68020 user SP (USP) is used for the same purpose as the 68000 USP in the user mode.
- The vector base register (VBR) is used in interrupt vector computation. For example, in the 68000 the interrupt address vector is obtained by multiplying an 8-bit vector number by 4. In the 68020, on the other hand, the interrupt address vector is obtained by using $VBR + 4 \times 8\text{-bit vector number}$.

- The SFC (source function code) and DFC (destination function code) registers are 3 bits wide. These registers allow the supervisor to move data between address spaces. In supervisor mode, 3-bit addresses can be written into SFC or DFC using instructions such as MOVEC A1, SFC. The MOVES.W(A0), D0 can then be used to move a word from a location within the address space specified by SFC and (A0) to D0. The 68020 outputs [SFC] to the FC2, FC1, and FC0 pins. By decoding these pins via external decoder, the desired source memory location addressed by (A0) can be moved to D0. Now, if this data in D0 is to be moved to another space, then the following instructions will accomplish this:

```
MOVEC A3, DFC
MOVES.W D0, (A5)
```

Note that there is no MOVES mem, mem instruction. SFC and DFC allow one to move data from one space to another. Since in the above, MOVES.W D0, (A5) outputs [DFC] to FC2, FC1, and FC0 pins which can be used to enable the chip containing the memory location addressed by (A5). [D0] is then moved to this location.

The new addressing modes in the 68020 include scaled indexing, 32-bit displacements, and memory indirection. In order to illustrate the concept of scaling, consider moving the contents of memory location 50₁₆ to A1. Using the 68000, the following instruction sequence will accomplish this:

```
MOVEA.W #10, A0          ; Load starting address of a
                           ; table to A0
MOVE.W #10, D0            ; Load index value to D0
ASL #2, D0                ; Scale index
MOVEA.L 0(A0, D0.W), A1   ; Access data
```

The scaled indexing can be used with the 68020 to perform the same as follows:

```
MOVEA.W #10, A0          ; Load starting address of
                           ; a table to D0
MOVE.W #10, D0            ; Load index value to D0
MOVE.L (0, A0, D0.W*4), A1 ; Access data
```

Note that [D0] in the above is scaled by 4. Scaling 1, 2, 4, or 8 can be obtained.

- The new 68020 instructions include bit field instructions to better support compilers and certain hardware applications such as graphics, 32-bit multiply and divide instructions, pack and unpack instructions for BCD, and coprocessor instructions. Bit field instructions can be used to input data from A/D converters and eliminate wasting main memory space when the A/D converter is not 32-bits.
- FC2, FC1, FC0 = 111 means CPU space cycle. The 68020 makes CPU space access for breakpoints, coprocessor operations, or interrupt acknowledge cycles. The CPU space classification is generated by the 68020 based upon execution of breakpoint instructions, coprocessor instructions, or during the interrupt acknowledge cycle. The 68020 then decodes A19-A16 to determine the type of CPU space. For example, FC2, FC1, FC0 = 111 and A19, A18, A17, A16 = 0010 mean coprocessor instruction.
- For performing floating-point operations, the 68000 user must write subroutines using the 68000 instruction set. The floating-point capability in the 68020 can be obtained by connecting the Motorola 68881 floating-point coprocessor chip. The 68020's two coprocessor chips include the 68881 (floating-point) and 68851 (memory management). The MC68020 can have up to eight coprocessor chips. When a coprocessor is

connected to the 68020, the coprocessor instructions are added to the 68020 instruction set automatically, and this is transparent to the user. For example, when the 68881 floating-point coprocessor is added to the 68020, instructions such as FADD (floating-point ADD) are available to the user. The programmer can then execute the instruction:

FADD FD0, FD1

Note that registers FD0 and FD1 are in the 68881. When the 68020 encounters the FADD instruction, it writes a command in the command register in the 68881, indicating that the 68881 has to perform this operation. The 68881 then responds to this by writing in the 68881 response register. Note that all coprocessor registers are memory-mapped. The 68020 thus can read the response register and obtain the result of the floating-point ADD from the appropriate location.

6.2 Programming Model

Figure 6.1 shows the MC68020 programming model. The user model has sixteen 32-bit general-purpose registers (D0-D7 and A0-A7), a 32-bit program counter (PC), and a condition code register (CCR) contained within the supervisor status register (SR). The supervisor model has two 32-bit supervisor stack pointers (ISP and MSP), a 16-bit status register (SR), a 32-bit vector base register (VBR), two 3-bit alternate function code registers (SFC and DFC), and two 32-bit cache handling (address and control) registers (CAAR and CACR). General-purpose registers D0-D7 are used as data registers for operation on all data types. General-purpose registers A0-A6, user stack pointer (USP) A7, interrupt stack pointer (ISP) A7', and master stack pointer (MSP) A7'' are address registers that may be used as software stack pointers or base address registers.

The status register (Figure 6.2) consists of a user byte (condition code register CCR) and a system byte. The system byte contains control bits to indicate that the processor is in the trace mode (T1, T0), supervisor/user state (S), and master/interrupt state (M). The user byte consists of the following condition codes: carry (C), overflow (V), zero (Z), negative (N), and extend (X).

The bits in 68020 user byte are set at reset in the same way as the 68000 user byte. The bits I2, I1, I0, and S have the same meaning as the 68000. In the 68020, two trace bits (T1, T0) are included as opposed to one trace bit (T) in the 68000. These two bits allow the 68020 to trace on both normal instruction execution and jumps. The 68020 M-bit is not included in the 68000 status register.

The vector base register (VBR) is used to locate the exception processing vector table in memory.

The MC68020 distinguishes address spaces as supervisor/user and program/data. To support full access privileges in the supervisor mode, the alternate function code registers (SFC and DFC) allow the supervisor to access any address space by preloading the SFC/DFC registers appropriately.

The cache registers (CACR and CAAR) allow software manipulation of the instruction cache. The CACR provides control and status accesses to the instruction cache, while the CAAR holds the address for those cache control functions that require an address.

6.3 Data Types, Organization, and CPU Space Cycle

The MC68000 family supports data types of bits, byte integers (8 bits), word integers (16 bits), long word integers (32 bits), and binary coded decimal (BCD) digits. In addition to these, four

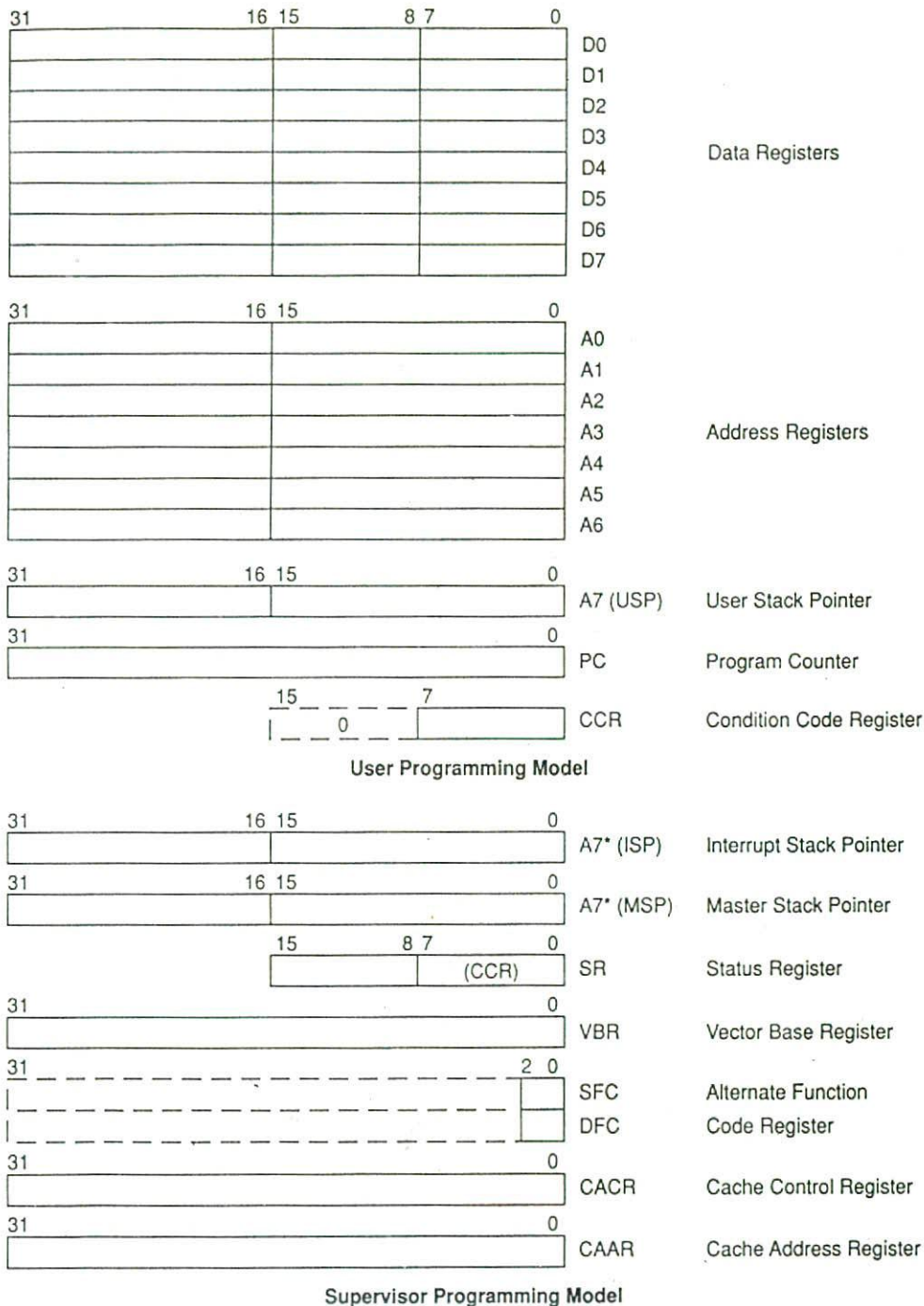


FIGURE 6.1 MC68020 programming model.

new data types are supported by the MC68020: variable-width bit field, packed BCD digits, quad words (64 bits), and variable-length operands.

Data stored in memory are organized on a byte-addressable basis, where the lower addresses correspond to higher-order bytes. The MC68020 does not require data to be aligned on even byte boundaries, but data that are not aligned are transferred less efficiently.

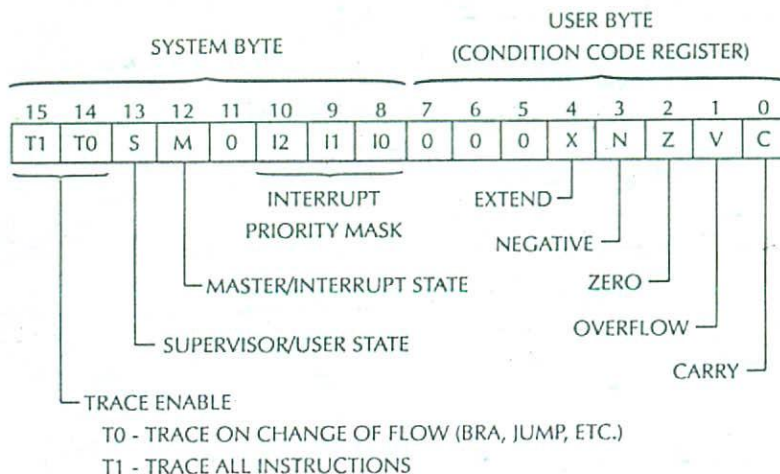


FIGURE 6.2 MC68020 status register.

Instruction words must be aligned on even byte boundaries. Figure 6.3 shows how data are organized in memory.

Table 6.1 shows decoding of the function code pins. The function code pins define the user/supervisor program and data spaces in the same way as the MC68000, except that FC2 FC1 FC0 = 111 for the MC68020 defines a new cycle called the CPU space cycle. Note that for the MC68000, FC2, FC1, FC0 = 111 provides the interrupt acknowledge cycle. CPU space is not intended for general instruction execution, but is reserved for processor functions. The CPU space has been subdivided into 16 types of access. The type of CPU access is indicated by address bits (A19-A16) in combination with the CPU space function code (FC2 FC1 FC0 = 111).

Table 6.2 defines the four different types of CPU accesses. The definition of regions in the CPU space makes it possible to acknowledge break points and interrupts and to communicate with coprocessors and other special devices (such as the MMU) without dictating memory organization for user- and supervisor-related activity.

The MC68020 has three stack pointers: the user stack pointer (USP) register A7, the interrupt stack pointer (ISP) register A7', and the master stack pointer (MSP) register A7". During normal operation most codes will be executed in user space and programs will use the A7 stack for temporary storage and parameter passing between software routines (modules). The ISP register is only used when an exception occurs, such as an external interrupt when control is passed to supervisor mode and the relevant exception process is performed. The MSP holds process-related information for the various tasks and allows for the separation of task-related and non-task-related exception process stacking.

When the master stack is enabled through bit (M) in the SR, all noninterrupting exceptions, such as divide by zero, software traps, and privilege violation, are placed in the master stack.

6.4 MC68020 Addressing Modes

Figure 6.4 lists the MC68020's 18 addressing modes. Table 6.3 compares the addressing modes of the MC68000 with those of the MC68020.

Since MC68000 addressing modes are covered in detail with examples in Chapter 5, the MC68020 modes which are not available in the MC68000 are covered in the following discussion.

6.4.1 Address Register Indirect (ARI) with Index and 8-Bit Displacement

Assembler syntax: (d8, An, Xn. size * SCALE)

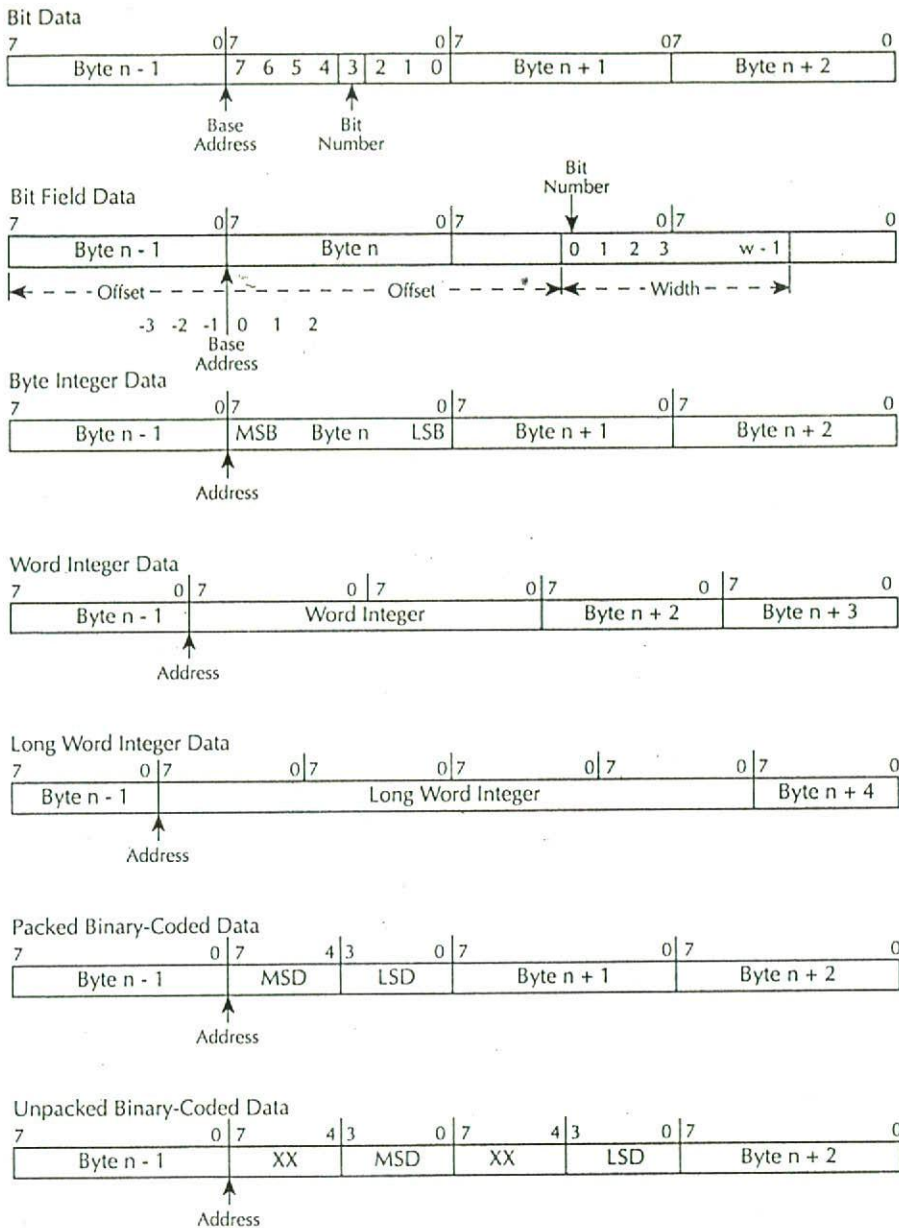


FIGURE 6.3 Memory data organization.

$$EA = (An) + (Xn.size * scale\ value) + d8$$

Xn can be W or L.

If index register (An or Dn) is 16 bits, then it is sign-extended to 32 bits and then multiplied by 1, 2, 4, or 8 prior to being used in EA calculation. d8 is also sign-extended to 32 bits prior to EA calculation. An example is

```
MOVE.W (0, A2, D2.W*2), D1
```


TABLE 6.1 Processing State Address Space

Function code			Address space
FC2	FC1	FC0	
0	0	0	(Undefined, reserved)*
0	0	1	User data space
0	1	0	User program space
0	1	1	(Undefined, reserved)*
1	0	0	(Undefined, reserved)*
1	0	1	Supervisor data space
1	1	0	Supervisor program space
1	1	1	CPU space

* Address space 3 is reserved for user definition, while 0 and 4 are reserved for future use by Motorola.

TABLE 6.2 CPU Space Address Access Encodings

CPU Space Access Types	Function Code	Address Bus																																				
	2 0	31																23	19	16																4	2	0
Breakpoint Acknowledge	1 1 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	BKPT#	0	0			
Access Level Control	1 1 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	0				
Coprocessor Comm	1 1 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0				
Interrupt Acknowledge	1 1 1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3	1	0			

CPU Space Type Field

TABLE 6.3 Addressing Modes, MC68000 Vs. MC68020

Addressing modes available		68000	68020
Data Register Direct	Dn	Yes	Yes
Address Register Direct	An	Yes	Yes
Address Register Indirect (ARI)	(An)	Yes	Yes
ARI with Postincrement	(An)+	Yes	Yes
ARI with Predecrement	-(An)	Yes	Yes
ARI with Displacement (16-bit displ)	(d, An)	Yes	Yes
ARI with Index (8-bit displ)	(d, An, Xn)	Yes*	Yes
ARI with Index (Base Displ: 0, 16, 32)	(bd, An, Xn)	No	Yes
Memory Indirect (Postindexed)	([bd, An], Xn, od)	No	Yes
Memory Indirect (Preindexed)	([bd, An, Xn], od)	No	Yes
PC Indirect with Displ. (16-Bit)	(d, PC)	Yes	Yes
PC Indirect with Index (8-Bit Displ)	(d, PC, Xn)	Yes*	Yes
PC Indirect with Index (Base Displ)	(bd, PC, Xn)	No	Yes
PC Memory Indirect (Postindexed)	([bd, PC], Xn, od)	No	Yes
PC Memory Indirect (Preindexed)	([bd, PC, Xn], od)	No	Yes
Absolute Short	(xxx).W	Yes	Yes
Absolute Long	(xxx).L	Yes	Yes
Immediate	#<data>	Yes	Yes

* 68000 has no scaling capability; 68020 can scale Xn by 1, 2, 4 or 8

Addressing Modes	Syntax
Register Direct Data Register Direct Address Register Direct	Dn An
Register Indirect Address Register Indirect Address Register Indirect with Post Increment Address Register Indirect with Predecrement Address Register Indirect with Displacement	(An) (An)+ -(An) (d16, An)
Register Indirect with Index Address Register Indirect with Index (8-Bit Displacement) Address Register Indirect with Index (Base Displacement)	(d8, An, Xn) (bd, An, Xn)
Memory Indirect Memory Indirect Post-Indexed Memory Indirect Pre-Indexed	([bd, An], Xn, od) ([bd, An, Xn], od)
Program Counter Indirect with Displacement	(d16, PC)
Program Counter Indirect with Index PC Indirect with Index (8-Bit Displacement) PC Indirect with Index (Base Displacement)	(d8, PC, Xn) (bd, PC, Xn)
Program Counter Memory Indirect PC Memory Indirect Post-Indexed PC Memory Indirect Pre-Indexed	([bd, PC,], Xn, od) ([bd, PC, Xn], od)
Absolute Absolute Short Absolute Long	xxx.W xxx.L
Immediate	#(data)

NOTES:

Dn = Data Register, D0-D7

An = Address Register, A0-A7

d8, d16 = A two's-complement, or sign-extended displacement; added as part of the effective address calculation; size is 8 (d8) or 16 (d16) bits; when omitted, assemblers use a value of zero.

Xn = Address or data register used as an index register; form is Xn.SIZE*SCALE, where SIZE is .W or .L (indicates index register size) and SCALE is 1, 2, 4, or 8 (index register is multiplied by SCALE); use of SIZE and/or SCALE is optional.

bd = A two's-complement base displacement; when present, size can be 16 or 32 bits.

od = Outer displacement, added as part of effective address calculation after any memory indirection; use is optional with a size of 16 or 32 bits.

PC = Program Counter

(data) = Immediate value of 8, 16, or 32 bits.

() = Effective Address.

[] = Use as indirect address to long word address.

FIGURE 6.4 MC68020 addressing modes.

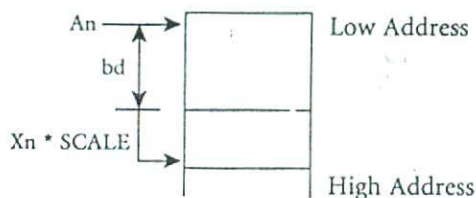
Suppose that $[A2] = \$5000\ 0000$, $[D2.W] = \$1000$, and $[\$5000\ 2000] = \1571 ; then after execution of the **MOVE**, $[D1]_{\text{low 16-bit}} = \1571 , since $EA = \$5000\ 0000 + \$1000 * 2 + 0 = \$5000\ 2000$.

6.4.2 ARI with Index (Base Displacement, bd: Value 0 or 16 Bits or 32 Bits)

Assembler syntax: (bd, An, Xn.size*SCALE)

EA = (An) + (Xn.size * SCALE) + bd

The figure below shows the use of ARI with index, Xn and base displacement, bd for accessing tables or arrays:

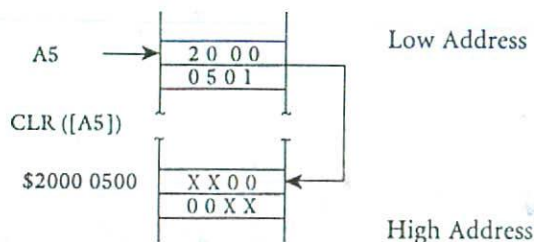


An example is **MOVE.W (\$5000, A2, D1.W * 4), D5**. If $[A2] = \$3000\ 0000$, $[D1.W] = \$0200$, and $[\$3000\ 5800] = \0174 , then after this **MOVE**, $[D5]_{\text{low 16 bits}} = \0174 , since $EA = \$5000 + \$3000\ 0000 + \$0200 * 4 = \$3000\ 5800$.

6.4.3 Memory Indirect

Memory indirect is distinguished from address register indirect by use of square brackets ([]) in the assembler notation.

The concept of memory indirect mode is depicted below:



In the above, register A5 points to the effective address \$2000 0501. Since **CLR ([A5])** is a 16-bit clear instruction, two bytes in location \$2000 0501 and \$2000 0502 are cleared to zero.

Memory indirect mode can be indexed with scaling and displacements. There are two types of memory indirect with scaled index and displacements: postindexed memory indirect mode and preindexed memory indirect mode.

For postindexed memory indirect mode, an indirect memory address is first calculated using the base register (An) and base displacement (bd). This address is used for an indirect memory access of a long word followed by adding a scaled indexed operand and an optional outer displacement (od) to generate the effective address. bd and od can be zero, 16 bits, or 32 bits.

In this memory indirect mode, indexing occurs after memory indirection.

Assembler syntax:

([bd, An], Xn.size Size * Scale, od)

EA = ([bd + An]) + Xn.size * Scale + od

An example is **MOVE.W** (**[\$0004, A1]**, **D1.W * 2, 2**), **D2**. If **[A1] = \$2000 0000**, **[\$2000 0004] = \$0000 3000**, **[D1.W] = \$0002**, **[\$0000 3006] = \$1A40**, then after execution of the above **MOVE**, intermediate pointer = $(4 + \$2000\ 0000) = \$2000\ 0004$, **[\$2000 0004]**, which is **\$0000 3000** used as a pointer. Therefore, $EA = \$0000\ 3000 + \$0000\ 0004 + 2 = \$00003006$; hence, $[D2]_{\text{low 16 bits}} = \$1A40$.

For memory indirect preindexed, the scaled indexed operand is added to the base register (**An**) and base displacement (**bd**). This result is then used as an indirect address into the data space. The 32-bit value at this address is fetched and an optional outer displacement (**od**) is added to generate the effective address. The indexing, therefore, occurs before indirection.

Assembler syntax:

```
([bd, An, Xn.size * Scale], od)
EA = (bd + An + Xn.size * Scale) + od
```

As an application of memory indirect preindexed mode, consider several I/O devices in a system. The addresses of these devices can be held in a table pointed to by **An**, **bd**, and **Xn**. The actual programs for the devices can be stored in memory pointed to by the respective device addresses and **od**.

As an example of memory indirect preindexed mode, consider **MOVE.W** (**[\$0004, A2, D1.W * 4]**, **2**), **D5**. If **[A2] = \$3000 0000**, **[D1.W] = \$0002**, **[\$3000 000C] = \$0024 1782**, **[\$0024 1784] = \$F270**, then after execution of the above **MOVE**, intermediate pointer = $\$3000\ 0000 + 4 + \$0002 * 4 = \$3000\ 000C$. Therefore, **[\$3000 000C]** which is **\$0024 1782** is used as a pointer to memory. $EA = \$0024\ 1782 + 2 = \$0024\ 1784$. Hence, $[D5]_{\text{low 16 bits}} = \$F270$. Note that in the above, **bd**, **Xn**, and **od** are sign-extended to 32 bits if one (or more) of them is 16 bits wide before the calculation.

6.4.4 Memory Indirect with PC

In this mode, **PC** (program counter) is used to form the address rather than an address register. The effective address calculation is similar to address register indirect.

6.4.4.a PC Indirect with Index (8-Bit Displacement)

The effective address is obtained by adding the **PC** contents, the sign-extended displacement, and the scaled indexed (sign-extended to 32 bits if it is 16 bits before calculation) register.

Assembler syntax:

```
(d8, PC, Xn.size * Scale)
EA = (PC) + (Xn.size * SCALE) + D8
```

For example, consider **MOVE.W** **D2, (2, PC, D1.W * 2)**. If **[PC] = \$4000 0020**, **[D1.W] = \$0020**, **[D2.W] = \$20A2**, then after this **MOVE**, $EA = 2 + \$4000\ 0020 + \$0020 * 2 = \$4000\ 0062$. Hence, **[\$4000 0062] = \$20A2**.

6.4.4.b PC Indirect with Index (Base Displacement)

This address of the operand is obtained by adding the **PC** contents, the scaled index register contents, and the base displacement.

Assembler syntax:

```
(bd, PC, Xn.size * Scale)
EA = (PC) + (Xn.size * SCALE) + bd
```

Xn and **bd** are sign-extended to 32 bits if either or both are 16 bits.

As an example, consider **MOVE.W (4, PC, D1.W * 2), D2**. If $[PC] = \$2000\ 0004$, $[D1.W] = \$0020$, $[2000\ 0048] = \$2560$, then after this **MOVE**, $[D2.W] = \$2560$.

6.4.4.c PC Indirect (Postindexed)

An intermediate memory pointer in program space is calculated by adding PC (used as a base register) and bd. The 32-bit content of this address is used in the EA calculation. EA is obtained by adding the 32-bit contents with a scaled index register and od. Note that bd, od, and index register are sign-extended to 32 bits before using in calculation if one (or more) is 16 bits.

Assembler syntax:

```
([bd, PC], Xn.size * Scale, od)
EA = ([bd + PC] + Xn.size * Scale + od)
```

Consider another example: **MOVE.W ([2, PC], D1.W*4, 0), D1**. If $[PC] = \$3000\ 0000$, $[D1.W] = \$0010$, $[\$3000\ 0002] = \$2040\ 0050$, $[\$2040\ 0090] = \$A240$, then after this **MOVE**, $[D1.W] = \$A240$.

6.4.4.d PC Indirect (Preindexed)

The scaled index register is added to the PC and bd. This sum is then used as an indirect address into the program space. The 32-bit value at this address is added to od to find EA.

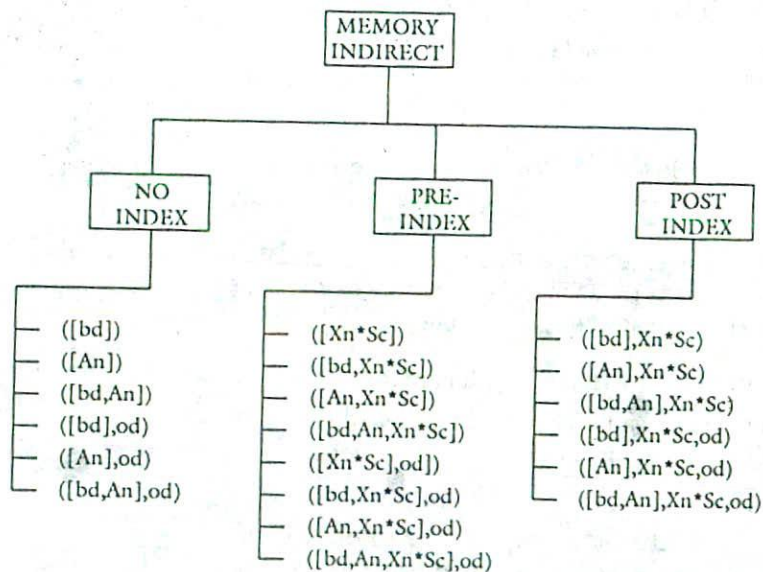
Assembler syntax:

```
([bd, PC, Xn.size * Scale], od)
EA = (bd + PC + Xn.size * Scale) + od
```

od, bd, and the index register are sign-extended to 32 bits if one (or more) of them is 16 bits before the EA calculation.

As an example, consider **MOVE.W ([4, PC, D1.W * 2], 4), D5**. If $[PC] = \$5000\ 0000$, $[D1.W] = \$0010$, $[\$5000\ 0024] = \$2050\ 7000$, $[\$2050\ 7004] = \0708 , then after this **MOVE**, $[D5.W] = \$0708$.

A summary of PC modes is provided below:



Example 6.1

Show the contents of registers A2, D4, A5, and the affected memory location(s), after execution of the following instruction:

MOVEA.W (0, A2, D4.L), A5

Assume prior to execution of the MOVEA instruction:

[A2]	=	\$0571	6660
[D4]	=	\$3072	8400
[A5]	=	\$7271	5554
[\$35E3 EA58]	=	\$05	
[\$35E3 EA59]	=	\$07	
[\$35E3 EA60]	=	\$F7	
[\$35E3 EA61]	=	\$F1	
[\$35E3 EA62]	=	\$40	

Solution

Effective address:

D4.L	=	\$3072	8400
A2.L	=	\$0571	6660
d	=	\$0000	0000
EA	=	\$35E3	EA 60

Therefore, after execution of the MOVEA instruction:

[A2]	=	\$0571	6660
[D4]	=	\$3072	8400
[A5]	=	\$FFFFFF7F1	

Example 6.2

The following MC68000 instruction sequence:

MOVEA.L 6 (USP), A1
MOVE.W (A1), D5

is used by a subroutine to access a parameter whose address has been passed into A1 and then moves the parameter to D5.

Find the equivalent MC68020 instruction.

Solution

MOVE.W ([6, USP]), D5

Example 6.3

Find a MC68020 compare instruction with the appropriate addressing mode to replace the following MC68000 instruction sequence:


```
ASL.L    #3, D6
CMP.L    0 (A1, D6.L), D3
```

Solution

The equivalent MC 68020 instruction is

```
CMP.L (0, A1, D6.L * 8), D3
```

Example 6.4

Find the contents of register A1, D0, D5 and the affected memory locations after execution of:

```
MOVE.B (0, A1, D0.W * 2), D5
```

Assume the following data prior to execution of the MOVE instruction:

[A1]	=	\$0000 2000
[D0]	=	\$0000 0004
[D5]	=	\$7124 8002
[\$0000 2006]	=	\$51
[\$0000 2007]	=	\$74
[\$0000 2008]	=	\$82
[\$0000 2009]	=	\$FO

Solution

Effective address:

$$\begin{aligned}
 &= d8 + A1.L + D0.W * 2 \\
 &= 0 + \$0000\ 2000 + \$0000\ 0004 * 2 \\
 &= \$0000\ 2008
 \end{aligned}$$

Therefore,

[A1]	=	\$0000 2000
[D0]	=	\$0000 0004
[D5]	=	\$7124 8082

6.5 68020 Instructions

The MC 68020 instruction set includes all 68000 instructions, plus some new ones. Some of the 68000 instructions are enhanced.

The 68020 new instructions beyond those of the 68000 can be grouped as follows:

1. 68020 new privileged MOVE instructions
2. RTD instruction
3. CHK/CHK2 and CMP/CMP2 instructions
4. TRAPcc instructions
5. Bit field instructions
6. PACK and UNPK instructions
7. Multiplication and division instructions
8. 68000 enhanced instructions
9. 68020 Advanced Instructions (to be discussed in Section 6.6)

- BKPT instructions
- CALLM/RTM instructions
- CAS/CAS2 instructions
- Coprocessor instructions

TABLE 6.4 MC68020 Privileged Move Instructions

Instruction	Operand size	Operation	Notation
MOVEC	32	Rc → Rn Rn → Rc	MOVEC.L Rc, Rn MOVEC.L Rn, Rc
MOVES	8, 16, 32	Rn → destination using DFC Source using SFC → Rn	MOVES.S Rn, (EA) MOVES.S(EA), Rn

6.5.1 New Privileged Move Instruction

The new privileged move instructions are executed by the 68020 in the supervisor mode.

The MOVE instructions are summarized in Table 6.4. The MOVEC instruction was added to allow the new supervisor registers (Rc) to be accessed. Since these registers are used for system control, they are generally referred to as control registers and include the vector base register (VBR), the source function code and destination function code registers (SFC, DFC), the master, interrupt, and user stack pointers (MSP, ISP, USP), and the cache control and address registers (CACR, CAAR). Register (Rn) can be either an address register or data register.

The operand size indicates that these MOVEC operations are always long-word. Notice only register-to-register operations are allowed.

A control register (Rc) can be copied to an address or data register (Rn), or vice versa. When copying the 3-bit, SFC, or DFC register into Rn, all 32 bits of the register are overwritten and the upper 29 bits are "0".

The MOVE to alternate space instruction (MOVES) allows the operating system to access any addressed space defined by the function codes.

It is typically used when an operating system running in the supervisor mode must pass a pointer or value to a previously defined user program or data space.

The MOVES instruction allows register-to-memory or memory-to-register operations. When a memory-to-register occurs, this instruction causes the contents of the source function code register to be placed on the external function hardware pins.

For a register-to-memory move, the processor places the destination function code register on the external function code pins.

The MOVES instruction can be used to MOVE information from one space to another. For example, in order to move the 16-bit content of a memory location addressed by A0 in supervisor data space (FC2 FC1 FC0 = 101) to a memory location addressed by A1 in user data space (FC2 FC1 FC0 = 001), the following instruction on sequence can be used:

```

MOVEQ.L #5, D0      ; Move source space 5 to D0
MOVEQ.L #1, D1      ; Move dest space 1 to D1
MOVEC.L D0, SFC     ; Initialize SFC
MOVEC.L D1, DFC     ; Initialize DFC
MOVES.W (A0), D2    ; Move memory location addressed by (A0)
                    ; and SFC to D2
MOVES.W D2, (A1)    ; Move D2 to a memory location addressed
                    ; by (A1) and DFC

```


In the above, the first four instructions initialize SFC to 101_2 and DFC to 001_2 . Since there is no MOVES mem, mem instruction, the register D2 is used as a buffer for the memory-to-memory transfer. MOVES.W (A0), D2 transfers [SFC] to FC2 FC1 FC0 and also reads the content of a memory location addressed by SFC and (A0) to D2. The 68020 FC2 FC1 FC0 pins can be decoded to enable the appropriate memory bank containing the memory location addressed by (A0). Next, MOVES.W D2, (A1) outputs [DFC] to FC2 FC1 FC0 and then moves [D2] to a memory location addressed by (A1) contained in a memory bank which can be enabled by decoding FC2, FC1, and FC0 pins.

Example 6.5

Find the content of memory location \$5000 2000 after execution by MOVE.W SR, [A6]. Assume the following data prior to execution of the MOVE instruction:

$$\begin{aligned} [A6] &= \$5000\ 2000 \quad [SR] = \$26A1 \\ [\$5000\ 2000] &= \$02, \quad [\$5000\ 2001] = \$F1 \end{aligned}$$

Also, assume supervisor mode.

Solution

SR is moved to a memory location pointed to by \$5000 2000. After execution:

$$\begin{aligned} [\$5000\ 2000] &= \$26 \\ [\$5000\ 2001] &= \$A1 \end{aligned}$$

Example 6.6

Find the content of DFC after execution of MOVEC.L A5, DFC. Assume the following data prior to execution of the instruction:

$$[DFC] = 100_2, \quad [A5] = \$2000\ 0105$$

Solution

After execution of the MOVEC, [DFC] = 101.

Example 6.7

Find the contents of D5 and the function code pins FC2, FC1, and FC0 after execution of MOVES.B D5, (A5). Assume the following data prior to execution of the MOVES:

$$\begin{aligned} [SFC] &= 101_2, \quad [DFC] = 100_2 \\ [A5] &= \$7000\ 0023 \\ [D5] &= \$718F\ 2A05 \\ [\$7000\ 0020] &= \$01, \quad [\$7000\ 0021] = \$F1 \\ [\$7000\ 0022] &= A2 \\ [\$7000\ 0023] &= \$2A \end{aligned}$$

Solution

After execution of the above MOVES:

$$\begin{aligned} FC2\ FC1\ FC0 &= 100_2 \\ [\$7000\ 0023] &= \$05 \end{aligned}$$

TABLE 6.5 RTD Instruction

Instruction	Operand size	Operation	Notation
RTD	Unsize	(SP) \rightarrow PC, SP + 4 + d \rightarrow SP	RTD # <displacement>

6.5.2 Return and Delocate Instruction

Return and delocate instruction RTD is useful when a subroutine has the responsibility to remove parameters off the stack that were pushed onto the stack by the calling routine. Note that the calling routine's JSR (jump to subroutine) or BSR (branch to subroutine) instructions do not automatically push parameters onto the stack prior to the call, as do the CALLM instructions. Rather, the pushed parameters must be placed there using the MOVE instruction. Table 6.5 shows the format of the RTD instruction.

The RTD instruction operates as follows:

1. Read the long word from memory pointed to by the stack pointer.
2. Copy it into the program counter.
3. Increment the stack pointer by 4.
4. Sign-extend the 16-bit immediate data displacement to 32 bits.
5. Add it to the stack pointer.

Since parameters are pushed onto the stack to lower memory locations, only a positive displacement should be added to the SP when removing parameters from the stack. The displacement value (16 bits) is sign-extended to 32 bits.

Example 6.8

Write a 68020 instruction sequence to illustrate the use of RTD instruction by using BSR instruction and pushing three 32-bit parameters onto the stack.

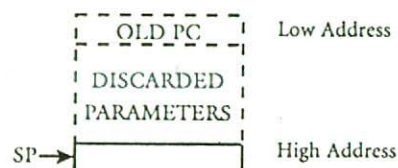
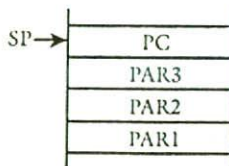
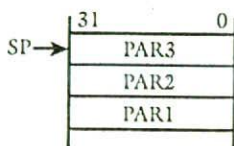
Solution

```
MOVE.L PAR1, -(SP)      BSR SUBR          RTD#12
MOVE.L PAR2, -(SP)
MOVE.L PAR3, -(SP)
```

Calling routine pushes parameters on stack; this causes the stack pointer to be decremented by 12

Calling routine calls subroutine and the PC stacked; the subroutine then accesses the parameters to perform the task

Last instruction of the subroutine returns and delocates the parameters off stack by adding (12) to the stack pointer



6.5.3 CHK/CHK2 and CMP/CMP2 Instructions

The enhanced MC68020 check instruction (CHK) now compares a 32-bit two's complement integer value residing in a data register (Dn) against a lower bound (L.B.) value of zero and

against an upper bound (U.B.) value of the programmer's choice. These bounds are located at the effective address (EA) specified in the instruction format.

The CHK instruction has the following format:

CHK.S (EA), D_n

where (.S) is the operand size designator which is either word (.W) or long word (.L).

If the data register value is less than zero ($D_n < 0$) or if the data register value is greater than the upper bound ($D_n > UB$), then the processor traps through exception vector 6 (offset \$18) in the exception vector table. Of course, the operating system or the programmer must define a check service handler routine at this vector address. The condition codes after execution of the CHK are affected as follows:

If $D_n < 0$ then $N = 1$.

If $D_n > UB$ (Upper Bound) then $N = 0$.

If $0 \leq D_n \leq UB$ then N is undefined. X is unaffected and all other flags are undefined and program execution continues with the next instruction.

This instruction can be used for maintaining array subscripts since all subscripts can be checked against an upper bound (i.e., $UB = \text{array size minus one}$). If the compared subscript is within the array bounds ($0 \leq \text{subscript value} \leq UB \text{ value}$), then the subscript is valid, and the program continues normal instruction execution. If the subscript value is out of array limits (i.e., $0 > \text{subscript value}$, or the subscript value $> UB \text{ value}$), then the processor traps through the CHK exception.

Example 6.9

Find the effects of execution of the MC68020 CHK instruction: CHK.L (A5), D3, where A5 contains a memory pointer to the array's upper bound value. Register D3 contains the subscript value to be checked against the array bounds. Assume the following data prior to execution of the CHK instruction:

[D3] = \$0150 7126, [A5] = \$00710004,
[\$0071 0004] = \$0150 0000

Solution

	Before	Operation	After
CHK.L (A5), D3	<p style="text-align: center;">D3</p> <div style="border: 1px solid black; padding: 2px; text-align: center;">0 1 5 0 7 1 2 6</div>	<p>$0 \leq D3.L < \\$01500000$ $\therefore N=0$, TRAP</p>	<p>Enter check exception service routine</p>
A5=\$0071004	<p style="text-align: center;">MEMORY</p> <div style="border: 1px solid black; padding: 2px; text-align: center;"> 31 0 0 1 5 0 0 0 0 0 </div>		<p style="text-align: center;">CCR</p> <div style="border: 1px solid black; padding: 2px; text-align: center;"> X N Z V C X 0 U U U </div>

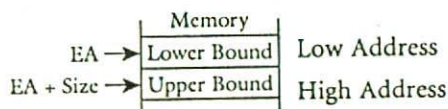
The long-word array subscript value \$01507126 contained in data register D3 is compared against the long-word upper-bound value \$01500000 pointed to by address register A5. Since the value \$01507126 contained in D3 exceeds the upper-bound value \$01500000 pointed to by A5, the N-bit is cleared. (The remaining CCR bits are either undefined or not affected.) This out-of-bound condition causes the program to trap to a check exception service routine. The operation of the CHK is summarized in Table 6.6.

TABLE 6.6 CHK Instruction Operation

Instruction	Operand size	Operation	Notation
CHK	16, 32	If $Dn < 0$ or $Dn > \text{source}$ then TRAP	CHK (Ea), Dn

The 68020 CMP.S (EA), Dn ($S = B$ or W or L) subtracts (EA) from Dn and affects the condition codes without any result being saved.

Both the CHK2 and the CMP2 instructions have similar formats (CHK2.S (EA), Rn) and (CMP2.S (EA), Rn). They compare a value contained in a data or address register designated by (Rn) against two (2) bounds chosen by the programmer. The size of the data to be compared (.S) may be specified as either byte (.B), word (.W), or long word (.L). As shown in the figure below, the lower-bound value (LB) must be located in memory at the effective address (EA) specified in the instruction, and the upper-bound value (UB) must follow immediately at the next higher memory address [i.e., UB addr. = LB. addr. + SIZE where SIZE = $B (+1)$, $W (+2)$, or $L (+4)$] as follows:



If the compared register is a data register (i.e., $Rn = Dn$) and the operand size (.S) is a byte or word, then only the appropriate low-order part of the data register is checked. If the compared register is an address register (i.e., $Rn = An$) and the operand size (.S) is a byte or word, then the bound operands are sign extended to 32 bits, and the extended operands are compared against the full 32 bits of the address register. After execution of CHK2 and CMP2, the condition code flags are affected as follows:

Carry = 1 if the contents of Dn are out of bounds
 = 0 otherwise
 Z = 1 if the contents of Dn are equal to either bound
 = 0 otherwise.

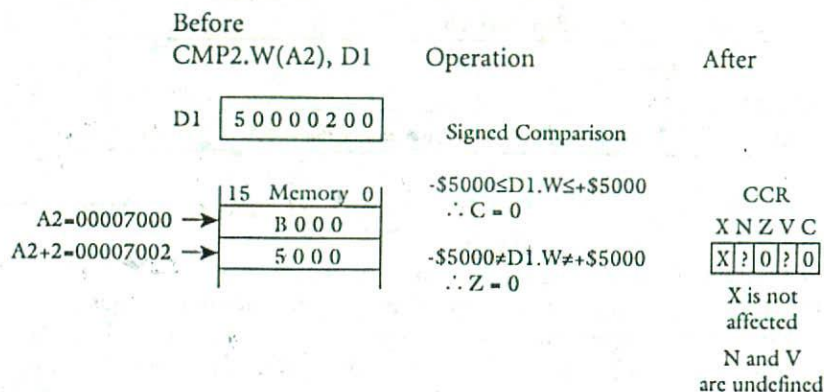
In the case where an upper bound equals the lower bound, the valid range for comparison becomes a single value. The only difference between the CHK2 and CMP2 instructions is that for comparisons determined to be out of bounds, CHK2 causes exception processing utilizing the same exception vector as the CHK instructions, whereas the CMP2 instruction execution only affects the condition codes.

In both instructions, the compare is performed for either signed or unsigned bounds. The MC68020 automatically evaluates the relationship between the two bounds to determine which kind of comparison to employ. If the programmer wishes to have the bounds evaluated as signed values, the arithmetically smaller value should be the lower bound. If the bounds are to be evaluated as unsigned values, the programmer should make the logically smaller value the lower bound.

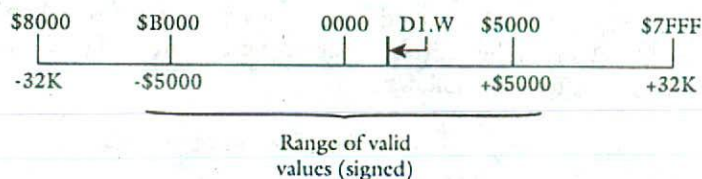
The following CHK2 and CMP2 instruction examples are identical in that they both utilize the same registers, comparison data, and bound values. The difference is how the upper and lower bounds are arranged.

Example 6.10

Determine the effects of CMP2.W (A2), D1. Assume the following data prior to execution of CMP2: [D1] = \$5000 0200, [A2] = \$0000 7000, [\$0000 7000] = \$B000, [\$0000 7002] = \$5000.



In this example, the word value \$B000 contained in memory (as pointed to by address register A2) is the lower bound and the word value immediately following \$5000 is the upper bound. Since the lower bound is the arithmetically smaller value, the programmer is indicating to the 68020 to interpret the bounds as signed numbers. The 2's complement value \$B000 is equivalent to an actual value of $-\$5000$. Therefore, the instruction evaluates the word contained in data register D1 (\$0200) to determine if it is greater than or equal to the upper bound, $+\$5000$, or less than or equal to the lower bound, $-\$5000$. Since the compared value \$0200 is within bounds, the carry bit (C) is cleared to zero. Also, since \$0200 is not equal to either bound, the zero bit (Z) is cleared. The figure below shows the range of valid value that D1 could contain:



A typical application for the CMP2 instruction would be to read in a number of user entries and verify that each entry is valid by comparing it against the valid range bounds. In the above CMP2 example, the user-entered value would be in register D1, and register A2 would point to a range for that value. The CMP2 instruction verifies if the entry is in range by clearing the CCR carry bit if it is in bounds and setting the carry bit if it is out of bounds.

Example 6.11

Find the effects of execution of CHK2.W (A2), D1. Assume the following data prior to execution of CHK2:

[D1] = \$5000 0200, [A2] = \$0000 7000, [\$0000 7000] = \$5000
 [\$0000 7002] = \$B000

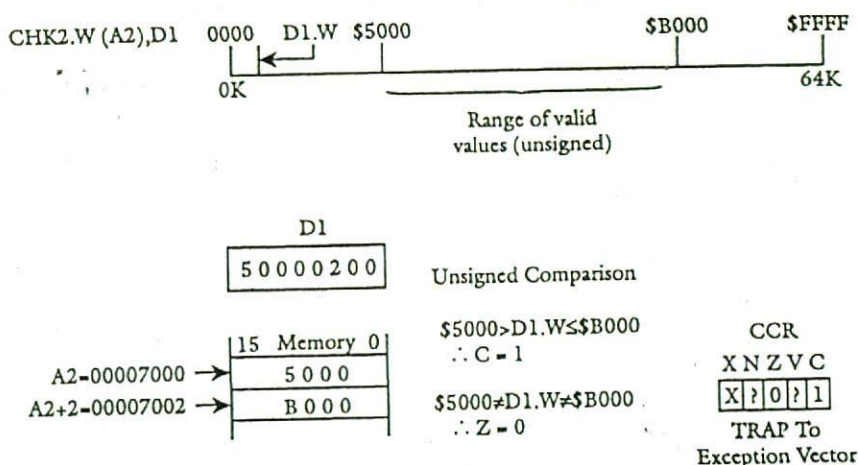
Solution

This time, the value \$5000 is located in memory as the lower bound, and the value \$B000 as the upper bound. Now, since the lower bound contains the logically smaller value, the programmer is indicating to the 68020 to interpret the bound values as unsigned number, representing only a magnitude. Therefore, the instruction evaluates the word value contained in register D1 (\$0200) to determine if it is greater than or equal to lower bound \$5000 or less

Instruction	Operand Size	Operation	Notation
CMP2	8, 16, 32	Compare $R_n < \text{source} - \text{lower bound}$ or $R_n > \text{source} - \text{upper bound}$ and set ccr	CMP2 (EA), R_n
CHK2	8, 16, 32	If $R_n < \text{source} - \text{lower bound}$ or $R_n > \text{source} - \text{upper bound}$ then TRAP	CHK2 (EA), R_n

FIGURE 6.4 Operation of CMP2 and CHK2.

than or equal to the upper bound \$B000. Since the compared value \$0200 is less than \$5000, the carry bit is set to indicate an out-of-bounds condition and the program traps to the CHK/CHK2 exception vector service routine. Also, since \$0200 is not equal to either bound, the zero bit (Z) is cleared. The range of valid values that D1 would contain is shown below:



A typical application for the CHK2 instruction would be to cause a trap exception to occur if a certain subscript value is not within the bounds of some defined array. Using the CHK2 example format just given, if we define an array of 100 elements with subscripts ranging from 50_{10} – 40_{10} , and if the two words located at (A2) and (A2 + 2) contain 40 and 49, respectively, and register D1 contains 100_{10} , then execution of the CHK2 instruction would cause a trap through the CHK/CHK2 exception vector. The operation of the CMP2 and the CHK2 instructions is summarized in Figure 6.4.

6.5.4 Trap On Condition Instructions

The new trap on condition (Trap cc) instruction has been added to allow a conditional trap exception on any of the following conditional conditions, as shown in Table 6.7.

These are the same conditions that are allowed for the set on condition (Scc) and the branch on condition (Bcc) instructions. The TRAPcc instruction evaluates the selected test condition based on the state of the condition code flags, and if the test is true, the MC68020 initiates exception processing by trapping through the same exception vector as the TRAPV instruction (vector 7, offset \$1C, Vector address = VBR + offset). The trap on cc instruction format is TRAPcc (or) TRAPcc (.S) # <data>, where (.S) is the operand size designator, which is either word (.W) or long word (.L).

A summary of the TRAPcc instruction operation is shown in Figure 6.5.

TABLE 6.7 Conditions for TRAPcc

CC	Carry clear	\bar{C}
CS	Carry set	C
EQ	Equal	Z
F	Never true	0
GE	Greater or equal	$N \cdot V + \bar{N} \cdot \bar{V}$
GT	Greater than	$N \cdot V \cdot Z + \bar{N} \cdot \bar{V} \cdot \bar{Z}$
HI	High	$C \cdot Z$
LE	Less or equal	$Z + N \cdot \bar{V} + \bar{N} \cdot V$
LS	Low or same	$C + \bar{Z}$
LT	Less than	$N \cdot \bar{V} + \bar{N} \cdot V$
MI	Minus	N
NE	Not equal	\bar{Z}
PL	Plus	N
T	Always true	1
VC	Overflow clear	\bar{V}
VS	Overflow set	V

Instruction	Operand Size	Operation	Notation
TRAPcc	None	If cc then TRAP	TRAPcc
	16		TRAPcc.W #<data>
	32		TRAPcc.L #<data>

FIGURE 6.5 TRAPcc operation.

6.5.5 Bit Field Instructions

The bit field instructions allow an operation such as clear, set, one's complement, input, insert, and test one or more bits in a string of bits (bit field).

Table 6.8 lists all the bit field instructions. Note that the condition codes are affected according to the value in field before execution of the instruction. All bit field instructions affect the N and Z bits as shown for BFTST. C and V are always cleared. X is always unaffected.

For all instructions: Z = 1, if all bits in a field prior to execution of the instruction are zero; Z = 0 otherwise. N = 1 if the most significant bit of the field prior to execution of the instruction is one; N = 0 otherwise.

(EA) address of the byte that contains bit 0 of the array
 offset # (0 to 31) or Dn (-2^{31} to $2^{31} - 1$)
 width # (1 to 32) or Dn (1 to 31, mod 32)

TABLE 6.8 Bit Field Instructions

Instruction	Operand size	Operation	Notation
BFTST	1-32	Field MSB \rightarrow N, Z = 1 if all bits in field are zero; Z = 0 otherwise	BFTST (EA){offset:width}
BFCLR	1-32	0's \rightarrow field	BFCLR (EA){offset:width}
BFSET	1-32	1's \rightarrow field	BFSET (EA){offset:width}
BFCHG	1-32	Field \rightarrow field	BFCHG (EA){offset:width}
BFEXTS	1-32	Field \rightarrow Dn; sign extended	BFEXTS (EA){offset:width}, Dn
BFEXTU	1-32	Field \rightarrow Dn; zero extended	BFEXTU (EA){offset:width}, Dn
BFINS	1-32	Dn \rightarrow field	BFINS Dn, (EA) {offset:width}
BFFFO*	1-32	Scan for first bit set in field	BFFFO (EA) {offset:width}, Dn

* The offset of the first bit set in bit field is placed in Dn; if no set bit is found, Dn contains the offset plus field width.

where # and Dn respectively indicate immediate and data register modes.

Immediate offset is from 0 to 31, while offset in Dn can be specified from -2^{31} to $2^{31} - 1$. All instructions are unsized. They are useful for memory conservation, graphics, and communications.

As an example, consider BFCLR \$5002 {4:12}. Assume the following memory contents:

	7	6	5	4	3	2	1	0	← bit number
\$5001	1	0	1	0	0	0	0	1	
\$5002 (Base-Address) →	1	0	0	1	1	1	0	0	
\$5003	0	1	1	1	0	0	0	1	
\$5004	0	0	0	1	0	0	1	0	

Bit 7 of the base address \$5002 has the offset 0. Therefore, bit 3 of \$5002 has offset value of 4. Bit 0 of location \$5001 has offset value -1 , bit 1 of \$5001 has the offset value -2 , and so on. The above BFCLR instruction clears 12 bits starting with bit 3 of \$5002. Therefore, bits 0 to 3 of location \$5002 and bits 0 to 7 of location \$5003 are cleared to zero. Therefore, the above memory contents are as follows:

	7	6	5	4	3	2	1	0	
\$5001	1	0	1	0	0	0	0	1	
\$5002	1	0	0	1	0	0	0	0	← offset 4
\$5003	0	0	0	0	0	0	0	0	← width 12
\$5004	0	0	0	1	0	0	1	0	← offset 16

The use of bit field instructions may result in memory savings. For example, assume that an input device such as a 12-bit A/D converter interfaced via a 16-bit port of an MC68020-based microcomputer. Now, suppose that one million pieces of data are to be collected from this port. Each 12 bits can be transferred to a 16-bit memory location or bit field instructions can be used.

Using 16-bit location for each 12-bit:

Memory bytes required:

$$= 2 * 1 \text{ million} \\ = 2 \text{ million bytes}$$

Using bit fields:

12 bits = 1.5 bytes

$$\text{Memory requirements} = 1.5 * 1 \text{ million} \\ = 1.5 \text{ million bytes}$$

$$\text{Savings} = 2 \text{ million bytes} - 1.5 \text{ million bytes} \\ = 500,000 \text{ bytes}$$

Example 6.12

Find the effects of:

BFCHG	\$5004	{D5 : D6}
BFEXTU	\$5004	{2 : 4}, D5
BFINS	D4, (A0)	{D5 : D6}
BFFFO	\$5004	{D6 : 4}, D5

Assume the following data prior to execution of each of the above instructions:

		MEMORY									
A0	0000 5004		7	-	-	-	-	-	-	-	0
		-16	1	0	0	0	0	1	0	1	
D5	FFFF FFFF	-8	0	0	0	0	0	1	0	0	
		\$5004H → 0	0	0	0	0	1	0	0	1	
D6	0000 0004	+8	0	1	0	1	0	0	0	1	
		+16	0	0	0	0	1	0	1	0	
CCR	01001	+24	0	1	0	0	1	0	1	0	
		+32	0	1	0	1	0	1	1	0	
D4	7125 F214	+40	1	0	0	1	0	0	0	1	

Register contents are given in hex, CCR and memory contents in binary, and offset to the left of memory in decimal.

Solutions

BFCHG \$5004 (D5 : D6)

Offset = -1, width = 4

	X	N	Z	V	C
CCR	0	0	1	0	0

Memory

					1
\$5004	1	1	1		

BFEXTU \$5004 [2 : 4], D5

Offset = 2, Width = 4

	X	N	Z	V	C
CCR	0	0	0	0	0

D5	0	0	0	0	0	0	0	2
----	---	---	---	---	---	---	---	---

BFINS D4, (A0) (D5 : D6)

Offset = -1, Width = 4

Memory

					0
\$5004	1	0	0		

	X	N	Z	V	C
CCR	0	0	1	0	0

BFFFO \$5004 (D6 : 4), D5

Offset = 4, Width = 4

	X	N	Z	V	C
CCR	0	1	0	0	0

D5	0	0	0	0	0	0	0	4
----	---	---	---	---	---	---	---	---

(Hex)

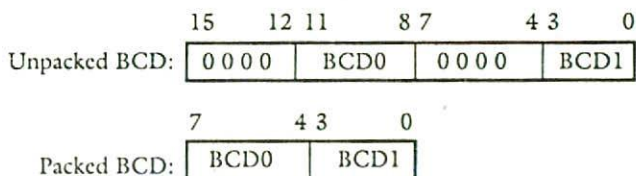
TABLE 6.9 Pack and Unpack Instructions

Instruction	Operand size	Operation	Notation
PACK	16 → 8	Unpacked source + # data → packed destination	PACK -(An), -(An), #<data> PACK Dn, Dn, #<data>
UNPK	8 → 16	Packed source → unpacked source, unpacked source + # data → unpacked destination	UNPK -(An), -(An), #<data> UNPK Dn, Dn, #<data>

Note: Condition codes are not affected.

6.5.6 Pack and Unpack Instructions

Table 6.9 lists the details of PACK and UNPK instructions. Both instructions have three operands and are unsized. They do not affect the condition codes. The PACK instruction converts two unpacked BCD digits to two packed BCD digits. The UNPK instruction reverses the process and converts two packed BCD digits to unpacked BCD digits. Immediate data can be added to convert numbers from one code to another. That is, these instructions can be used to translate codes such as ASCII or EBCDIC to BCD and vice versa.



#data - Appropriate constants can be used to translate from ASCII or EBCDIC to BCD or from BCD to ASCII or EBCDIC.

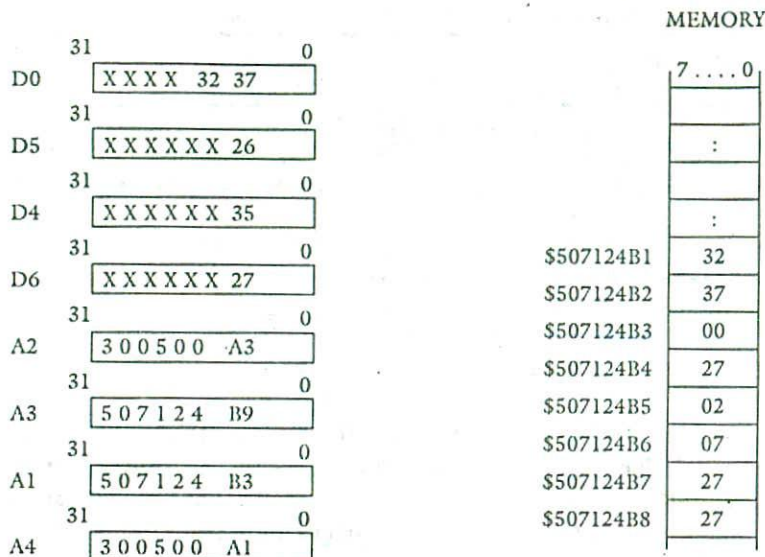
The PACK and UNPK instructions are useful when an I/O device such as an ASCII keyboard is interfaced to an MC68020-based microcomputer. Data can be entered into the microcomputer via the keyboard in ASCII codes. The PACK instruction can be used with appropriate adjustments to convert these ASCII codes into BCD. Arithmetic operations can be performed inside the microcomputer and the result will be in PACKED BCD. The UNPK instruction can similarly be used with appropriate adjustment to convert packed BCD to ASCII codes.

Example 6.13

Find the effects of execution of the following PACK and UNPK instructions:

1. PACK D0, D5, # \$0000
2. PACK - (A1), - (A4), # \$0000
3. UNPK D4, D6, # \$3030
4. UNPK - (A3), - (A2), # \$3030

Assume the following data:

**Solution**

1. `PACK D0, D5 # $000`

$$\begin{array}{r}
 [D0] = 32 \ 37 \\
 \text{low} \\
 \text{word} \\
 + 00 \ 00 \\
 \hline
 32 \ 37 \\
 \swarrow \searrow \\
 [D5] = 27
 \end{array}$$

Note that ASCII code for 2 is 32 and for 7 is 37. Hence the above PACK instruction converts ASCII code to packed BCD.

2. `PACK - (A1), - (A4), $0000`
- $$\begin{array}{r}
 [\$5071 \ 24B2] = 37 \quad + 3237 \\
 [\$5071 \ 24B1] = 32 \quad + 0000 \\
 \hline
 3237
 \end{array}$$
- Therefore, `[3005 00A0] = 27 Packed BCD`

Hence, the above instruction with the specified data converts two ASCII digits to their equivalent packed BCD.

3. `UNPK D4, D6 # $3030`
- $$\begin{array}{r}
 [D4] = \text{XXXXXX} \ 35 \\
 \quad \quad \quad 03 \ 05 \\
 + \quad \quad 30 \ 30 \\
 \hline
 \quad \quad 33 \ 35
 \end{array}$$
- Therefore, after this UNPK
- $$\begin{array}{l}
 [D6] = \text{XXXX} \ 33 \ 35 \\
 [D4] = \text{XXXXXX} \ 35
 \end{array}$$

Therefore, this instruction with the assumed data converts from packed BCD 35 to ASCII 33 35.

4.

UNPK - (A3), - (A2), # \$3030

[\$5071 24B8] = 27

02	07
30	30
32	37

Hence,

[\$300 500 A2] = 37

[\$300 500 A1] = 32

This instruction with the assumed data converts two packed BCD digits to their equivalent ASCII digits.

6.5.7 Multiplication and Division Instructions

The MC68020 includes the following signed and unsigned multiplication instructions:

MULS.W (EA), Dn 16 × 16 → 32, (EA)16 * Dn16 → Dn32

or
MULU

MULS.L (EA), Dn 32 × 32 → 32, (EA) * Dn → Dn

or
MULU

Holds low 32 bits of the result
after multiplication; upper 32 bits
of the result are discarded

MULS.L (EA), Dm:Dn 32 × 32 → 64, (EA) * Dn → Dm:Dn

or
MULU

Holds 32-bit multiplicand before
multiplication and low 32 bits of
the product after multiplication

Holds high 32 bits
of the product after
multiplication

Holds 32-bit multiplier before
multiplication

(EA) in the above can be all modes except An. The condition codes N, Z, V are affected, C is always cleared to zero, and X is unaffected for both MULS and MULU. For signed multiplication, overflow (V = 1) can only occur for 32 × 32 multiplication producing a 32-bit result if the high-order 32 bits of the 64-bit product are not the sign extension of the low-order 32 bits. In the case of unsigned multiplication, overflow (V = 1) can occur for 32 × 32 multiplication producing a 32-bit result if the high-order 32 bits of the 64-bit product are not zero.

Both MULS and MULU have a word form and a long-word form. For the word form (16 × 16) the multiplier and multiplicand are both 16 bits and the result is 32 bits. The result is saved in the destination data register. For 32 bit × 32 bit, the multiplier and multiplicand are both 32 bits and the result is either 32 bits or 64 bits. When the result is 32 bits for a 32-bit × 32-bit operation, the low-order 32 bits of the 64-bit product is provided.

The signed and unsigned division instructions of the MC68020 include the following: Source is the divisor and destination is the dividend. The result (remainder and quotient) is stored in the destination.

DIVS.W (ea), Dn 32/16 → 16r:16q

or
DIVU

DIVS.L (ea), Dn 32/32 → 32q (no remainder is

or
DIVU

provided)

DIVS.L (ea), Dr:Dq 64/32 → 32r:32q

or

DIVU

DIVSL.L (ea), Dr:Dq 32/32 → 32r:32q

or

DIVUL

Contains the 32-bit dividend

The destination contains the dividend and (EA) contains the divisor. (EA) in the above instructions can use all modes except An.

The condition codes for either signed or unsigned division are affected as follows: N = 1 if the quotient is negative; N = 0 otherwise. N is undefined if overflow or divide by zero. Z = 1 if the quotient is zero; Z = 0 otherwise. Z is undefined for overflow or divide by zero. V = 1 for division overflow; V = 0 otherwise. X is unaffected, and C = 0 (always).

Division by zero causes a trap. If overflow is detected before completion of the instruction, V is set to one, but the operands are unaffected.

Both signed and unsigned division instructions have a word form and three long-word forms.

For the word form, the destination operand is 32 bits and the source operand is 16 bits. The 32-bit result in Dn contains the 16-bit quotient in the low word and the 16-bit remainder in the high word. The sign of the remainder is the same as the sign of the dividend.

For DIVS.L (EA), Dn

or

DIVU,

both destination and source operands are 32 bits. The result in Dq contains the 32-bit quotient and the remainder is discarded.

For DIVS.L (EA), Dr:Dq

or

DIVU,

the destination is 64 bits contained in any two data registers and the source is 32 bits. The 32-bit register Dr (D0-D7) contains the 32-bit remainder and the 32-bit Dq (D0-D7) contains the 32-bit quotient.

Example 6.14

Find the effects of the following multiplication and division instructions:

- MULU.L # \$2, D5 if [D5] = \$FFFFFFF**
- MULS.L # \$2, D5 if [D5] = \$FFFFFFF**
- MULU.L # \$2, D5:D2**
if [D5] = \$2ABC 1800
and [D2] = \$FFFFFFF
- DIVS.L # \$2, D5 if [D5] = \$FFFFFFFC**
- DIVS.L # \$2, D2:D0**
if [D2] = \$FFFFFFF
and [D0] = \$FFFFFFFC
- DIVSL.L # \$2, D6:D1 if [D1] = \$0004 1224**
and [D6] = \$FFFFFFFD

Solution

- 1.
- MULU.L # \$2, D5 if [D5] = \$FFFFFFFF**

$$\begin{array}{r}
 \text{\$FFFFFFFF} \\
 * \text{\$00000002} \\
 \hline
 00000001 \quad \text{FFFFFFFFE} \\
 \downarrow \qquad \downarrow \\
 \text{V} = 1 \qquad \text{Low 32-bit} \\
 \text{since} \qquad \text{result in D5} \\
 \text{this is} \\
 \text{nonzero}
 \end{array}$$

Therefore, [D5] = \$FFFFFFFE, N = 0 since the most significant bit of result is 0, Z = 0 since the result is not zero, V = 1 since the high 32 bits of the 64-bit product are not zero, C = 0 (always), and X = not affected.

- 2.
- MULS.L # \$2, D5 if [D5] = \$FFFFFFFF**

$$\begin{array}{r}
 \text{\$FFFFFFFF} \quad (-1) \\
 * \text{\$00000002} \quad (+2) \\
 \hline
 \text{\$FFFFFFFF} \quad \text{\$FFFFFFFE} \quad (-2) \\
 \downarrow \\
 \text{Result in D5}
 \end{array}$$

Therefore, [D5] = \$FFFFFFFE, X = unaffected, C = 0, N = 1, V = 0, and Z = 0.

- 3.
- MULU.L # \$2, D5:D2 if [D5] = \$2ABC 1800 and [D2] = \$FFFFFFFF**

$$\begin{array}{r}
 \text{\$FFFFFFFF} \\
 * \text{\$00000002} \\
 \hline
 00000001 \quad \text{FFFFFFFFE} \\
 \text{D5} \qquad \qquad \text{D2}
 \end{array}$$

N = 0, Z = 0, V = 1 since high 32 bits of the 64-bit product are not zero, C = 0, and X = not affected.

- 4.
- DIVS.L # \$2, D5 if [D5] = \$FFFFFFFFC**

$$\begin{array}{r}
 \text{\$FFFFFFFFC} \\
 \text{\$00000002} \quad \text{\$FFFFFFFE} \quad \text{\$FFFFFFFC} \\
 \hline
 \text{\$00000002} \quad \text{\$FFFFFFFE} \quad \text{\$FFFFFFFC} \\
 \text{+2} \qquad \qquad \text{-4}
 \end{array}$$

[D5] = \$FFFF FFFE, X = unaffected, N = 1, Z = 0, V = 0, and C = 0 (always).

- 5.
- DIVS.L # \$2, D2:D0 if [D2] = \$FFFF FFFF and [D0] = \$FFFF FFFC**

$$\begin{array}{r}
 \text{\$0000 0000} \quad \text{\$FFFF FFFF} \quad \text{\$FFFF FFFC} \\
 \hline
 \text{\$0000 0000} \quad \text{\$FFFF FFFF} \quad \text{\$FFFF FFFC} \\
 \text{2} \qquad \qquad \text{-4}
 \end{array}$$

D2 = \$0000 0000 = remainder, D0 = \$FFFF FFFE = quotient, X = unaffected, Z = 0, N = 1, V = 0, and C = 0 (always).

TABLE 6.10 Enhanced Instructions

Instruction	Operand size	Operation
BRA label	8, 16, 32	PC + d → PC
Bcc label	8, 16, 32	If cc is true, then PC + d → PC; else next instruction
BSR label	8, 16, 32	PC → - (SP); PC + d → PC
CMPI.S # data, (EA)	8, 16, 32	Destination - # data → CCR is affected
TST.S (EA)	8, 16, 32	Destination - 0 → CCR is affected
LINK.S An, # - d	16, 32	An → - (SP); SP → An, SP + d → SP
EXTB.L Dn	32	Sign extend byte to long word

Note: S can be B, W, L. In addition to 8- and 16-bit signed displacements for BRA, Bcc, and BSR like the 68000, the 68020 also allows signed 32-bit displacements. Link is unsized in the 68000. (EA) in CMPI and TST support all MC68000 modes plus PC relative. Examples are CMPI.W#\$2000, (START, PC). In addition to EXT.W Dn and EXT.L Dn as with the 68000, the 68020 also provides the EXTB.L instruction.

6. DIVSL.L # \$2, D6:D1 if [D1] = \$0004 1224 and [D6] = \$FFFF FFFD

$$\begin{array}{r}
 \begin{array}{c} -1 \\ \hline Q = \text{FFFFFFF} \\ \hline 0000\ 0002 \left| \begin{array}{c} \text{FFFFFFD} \\ \hline -3 \end{array} \right. \\ \hline \end{array}
 \end{array}
 \quad
 \begin{array}{c}
 \begin{array}{c} -1 \\ \hline R = \text{FFFFFFF} \\ \hline \end{array}
 \end{array}$$

[D6] = \$FFFFFFFF = remainder, [D1] = \$FFFFFFFF = quotient, X = unaffected, N = 1, Z = 0, V = 0, and C = 0 (always).

6.5.8 MC68000 Enhanced Instructions

The MC68020 includes the enhanced version of the 68000 instructions listed in Table 6.10.

Example 6.15

Write a program in MC68020 assembly language to find the first one in a bit field which is greater than or equal to 16 bits and less than or equal to 512 bits. Assume the number of bits to be checked is divisible by 16. If no ones are found, store \$0000 0000 in D3; otherwise store the offset of the first set bit in D3 and stop. Assume A2 points to start of the array and D2 contains the number of bits in the array.

Solution

```

CLR.L D3                ; D3 is offset in bits
                        ; Contains the first bit
                        ; number set
DIVU #16, D2            ; [D2] = number of searches
SUBQ.W #1, D2           ; Decrement D2 by 1 for use
                        ; in DBNE later
MOVEQ.L #16, D5         ; Load field width into D5
START BFFFO (A2) {D3:D5}, D3 ; Search for one in 16 bits
DBNE D2, START          ; Decrement branch if not
                        ; equal

```

```

        BNE FINISH          ; If no ones found, stop
        CLR.L D3            ; Store 0 in D3
FINISH JMP FINISH          ; HALT

```

Example 6.16

Write a program in MC68020 assembly language to convert 20 packed BCD digits to their ASCII equivalent and store the result to memory location \$F1002004. The data bytes start at \$5000.

Solution

```

        MOVEA.L # $5000, A5      ; Load starting address of
                                   ; the BCD array into A5
        MOVEA.L # $F1002004, A6  ; Load starting address
                                   ; of the ASCII array
                                   ; into A6
        MOVEQ.L #19, D2          ; Load data length to D2
START  MOVE.B (A5) +, D3         ; Load BCD value
        UNPK D3, D3, # $3030    ; Convert to ASCII
        MOVE.W D3, (A6) +       ; Store ASCII data at
                                   ; address pointed to by
                                   ; A6
        DBF D2, START           ; Decrement and branch if
                                   ; false
FINISH JMP FINISH               ; Otherwise STOP

```

Example 6.17

Write a program in MC68020 assembly language to divide a signed 32-bit number in D0 by a signed 8-bit number in D1 by storing the division result in the following manner:

1. Store 32-bit quotient in D0 and neglect remainder.
2. Store 32-bit remainder in D1 and 32-bit quotient in D0.

Assume dividend and divisor are already in D0 and D1, respectively.

Solution

```

1.  EXTBL D1                ; Sign extend divisor to 32-bit
    DIVSL D1, D0            ; 32-bit quotient in D0 and
FINISH JMP FINISH           ; remainder is discarded and
                                ; halt
2.  EXTBL D1                ; Sign extend divisor to 32 bits
    DIVSL D1, D1:D0         ; 32-bit remainder in D1 and
                                ; 32-bit quotient in D0
FINISH JMP FINISH           ; Halt

```

Example 6.18

Write a 68020 assembly language program to compute the following:

$$D2.L:D1.L = (INTEGER + D1.L) * (D1.L - D0.B)$$

Assume that the location INTEGER and register D0 contain signed 32-bit numbers while the low byte of D0 holds an unsigned number less than 127₁₀. Store the 64-bit result in D2:L:D1.L.

Solution

```

MOVE.L    INTEGER, D2    ;
ADD.L     D1,D2          ; compute sum
EXTB.L    D0              ; compute sum
SUB.L     D0,D1           ; compute (D1.L-D0.B)
MULS.L    D2,D2:D1       ; perform multiplication
FINISH    JMP            FINISH

```

Example 6.19

Write a program in 68020 assembly language to multiply a signed byte by a 32-bit signed number to obtain a 64-bit result. The numbers are respectively pointed to by the addresses that are passed on to the user stack by a subroutine pointed to by (USP+4) and (USP+6). Store the 64-bit result in D2:D1.

Solution

```

MOVE.B    ([4,USP]),D0    ; Move first data
EXTB.L    D0              ; sign extend to 32 bits
MOVE.L    ([6,USP]),D1    ; Move second data
MULS.L    D0,D2:D1        ; Store result in D2:D1
FINISH    JMP FINISH      ; Halt

```

6.6 68020 ADVANCED INSTRUCTIONS

This section provides a detailed description of the 68020 advanced instructions including BKPT, CAS/CAS2, TAS, CALLM/RTM, and coprocessor instructions.

6.6.1 Breakpoint Instruction

A breakpoint is a debugging tool that allows the programmer to check or pass over an entire section of a program. Execution of a breakpoint usually results in exception processing. Hence, the programmer can use any of the TRAP vectors as breakpoints. Also, any of the interrupt levels can be used by external hardware to cause a breakpoint.

The BKPT instruction is included with the 68010, 68012 and 68020 microprocessors.

In order to place a breakpoint in a program loop located in RAM, the operating system can temporarily remove an instruction word from the program and insert the BKPT instruction in its place. In the case of the 68010/68012, the operating system services the breakpoint exception routine since trap to illegal instruction exception is taken upon execution of the BKPT instruction. Such interruption by the operating system due to BKPT instruction is undesirable since execution of the program being debugged is slowed down. This is why the 68020 BKPT instruction includes more real-time debug support via external hardware such as the Motorola MC68851 Paged Memory Management Unit chip in which the program loop count and the saved operation word are stored for up to eight BKPT instructions (BKPT#0 thru BKPT#7). The operating system exchanges the operation word with one of the eight breakpoint instruction as follows:

Breakpoint number	Instruction	op code
0	BKPT#0	\$4848
1	BKPT#1	\$4849
2	BKPT#2	\$484A
3	BKPT#3	\$484B
4	BKPT#4	\$484C
5	BKPT#5	\$484D
6	BKPT#6	\$484E
7	BKPT#7	\$484F

The operating system then stores the operation word in a register of external hardware and initializes that loop counter. The 68020 generates a special cycle called breakpoint cycle in the CPU space (FC2 FC1 FC0 = 111) each time it executes the BKPT instruction in the program being debugged. Figure 6.6 shows how CPU space 0 is encoded during execution of a BKPT instruction. 68020 then outputs the breakpoint number on the address bus. The external hardware, if present, decrements the corresponding loop counter and places the operation word on the data bus. The 68020 inputs this operation word and continues with the execution of the program. As soon as the 68020 executes the same BKPT instruction for the last count (last loop counter value), external breakpoint hardware asserts the BERR signal generating an illegal instruction trap. This is shown in the flowchart of Figure 6.7

Example 6.20

Explain the breakpoint operation shown in Figure 6.8.

Solution

In Step 1 the illegal instruction BKPT is inserted into the program flow as requested by a user. In this example, the 16-bit op code for ADD.L D2, D3 is replaced with the 16-bit op code for BKPT #4.

In Step 2, the op code ADD.L D2, D3 is stored by the MC68020 in the CPU space long-word memory location corresponding to breakpoint number four, address \$10.

Step 3 shows how long-word memory locations \$00 to \$1C in CPU space 0 are used to temporarily store the op codes replaced by breakpoint instructions 0 through 7. The displaced op code is stored in the upper 16 bits, and an optional count value can be loaded into the lower 16 bits through external hardware control.

In Step 4, each time BKPT #4 is executed, the MC68020 accesses CPU space 0 location \$10. If the count value (bits 0–15) is not zero, the replaced op code is placed on the data bus (bits 16–31), the counter is decremented by one, and the appropriate DSACKX lines are asserted by the external hardware. The MC68020 reads this op word and executes it. If the count value is zero, the 68020 BERR pin can be asserted by external hardware to cause exception processing.

The breakpoint function is summarized in Table 6.11. A hardware implementation to take advantage of the BKPT instruction's looping feature is shown in Figure 6.9. A debug monitor maintains a small amount of breakpoint memory. It is used to hold up to eight replacement

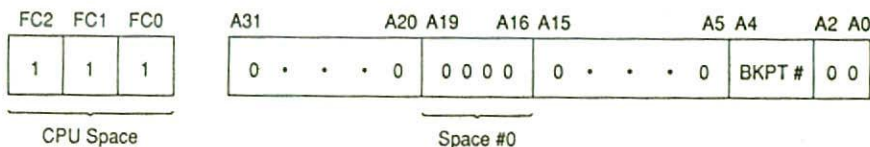


FIGURE 6.6 BKPT instruction format.

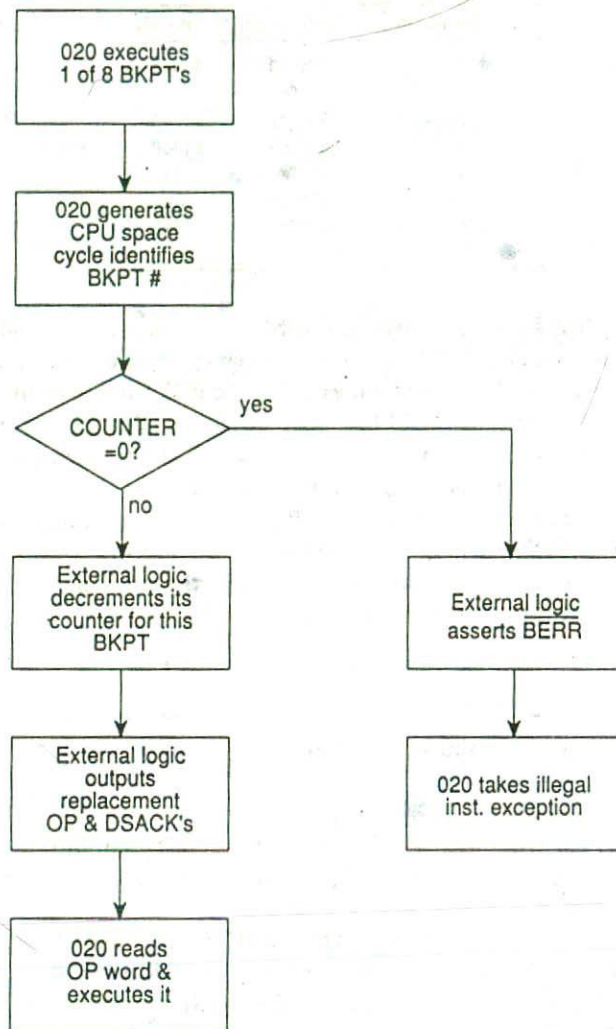


FIGURE 6.7 Breakpoint instruction flowchart.

op codes and eight counters. When a user wants a breakpoint to be encountered after say “10” passes of the op code, the debugger initiates a breakpoint counter to 10, replaces the user op code with a breakpoint instruction, saves the user op code in the breakpoint data memory, and executes the user’s program. When the breakpoint op code is encountered, the MC68020 generates the breakpoint acknowledge bus cycle. The 74LS138 3-to-8 decoder shown on the right decodes the function codes as all ones (validated by the AS signal) and asserts its bottom output indicating a CPU bus space cycle. This output enables one of the inputs of the 74LS138 shown on the left. The top enable output is enabled when address lines A 16 through A 19 are all zero, indicating a type 0 CPU space cycle. This decoder enables one of eight breakpoint counters.

If the counter $\neq 0$, then it is decremented by the external hardware and the replacement op code is returned on the data bus to the MC68020 with DSACK asserted. In this case, we started with a count of 10, so it is decremented to 9. When the breakpoint acknowledge cycle occurs on the 11th pass, the counter = 0 and this time bus error BERR signal is asserted by the external hardware. This forces illegal instruction processing and subsequent servicing of the breakpoint. With this type of hardware, operating system support is not required until the

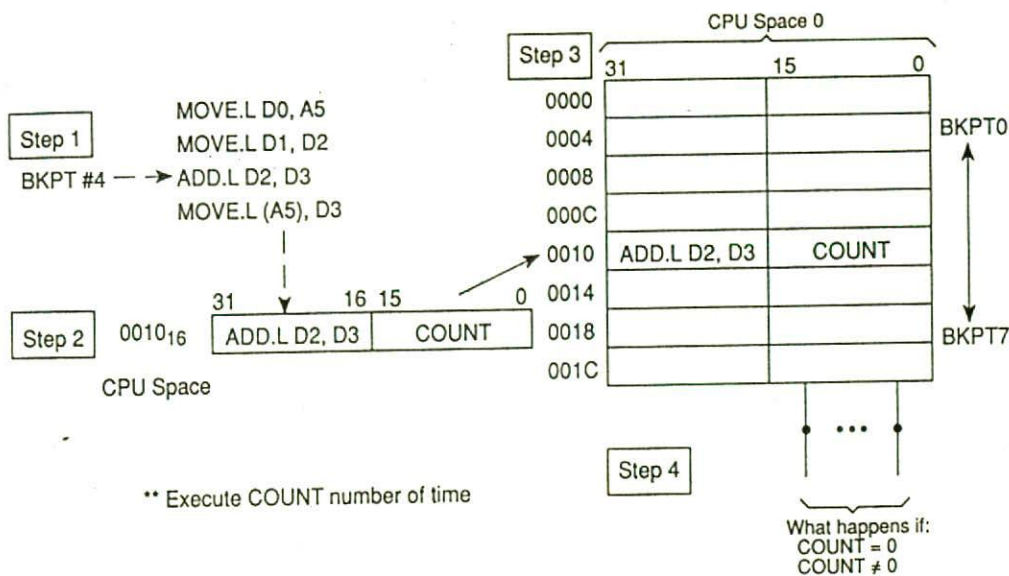


FIGURE 6.8 Executing a BKPT instruction.

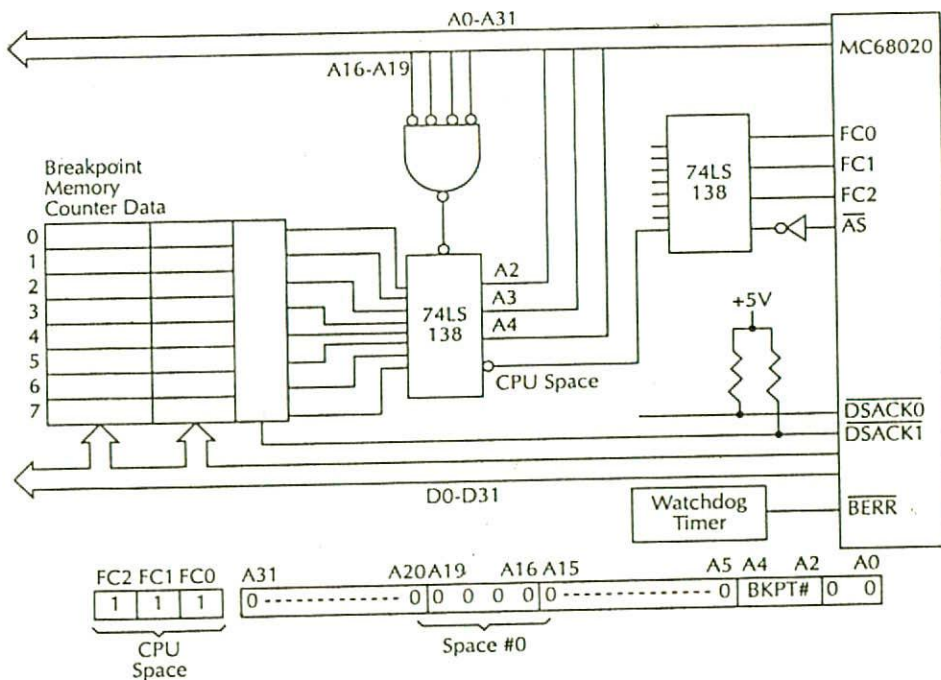


FIGURE 6.9 Breakpoint block diagram.

TABLE 6.11 Function of the Breakpoint

Instruction	Operand size	Operation	Notation
BKPT	None	If breakpoint cycle acknowledged then execute returned op word, else trap as illegal instruction	BKPT # data

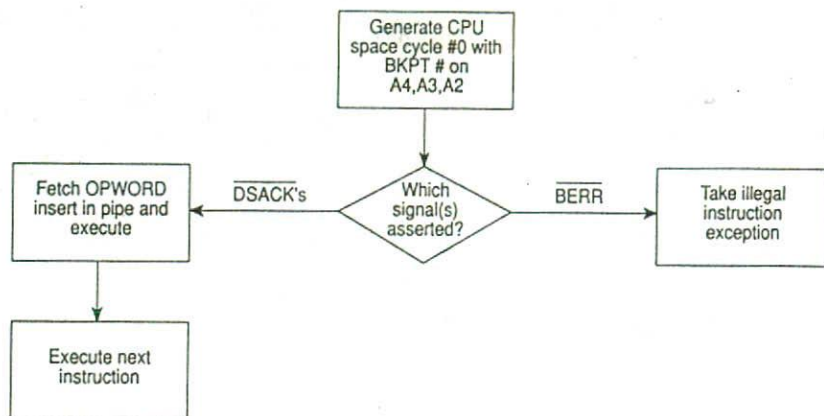


FIGURE 6.10 Breakpoint instruction flowchart.

count has been exhausted. This reduces the operating system overhead significantly. The MC68020 breakpoint instruction flow diagram is shown in Figure 6.10.

6.6.2 Call Module/Return from Module Instructions

For large programs, flowcharting does not provide an efficient software design tool. The flowcharts, however, can assist the programmer in dividing a large program into subprograms called modules (typically 20 to 50 lines). The task of dividing a large program into modules is called modular programming. Typical problems of modular programming include how to modularize a large program and then how to combine the individual modules.

One of the advantages of modular programming includes ease in writing, debugging, and testing a module rather than a large program.

By using modular programming, a program performing a specific function is divided into several smaller subfunctions. These subfunctions can be utilized once or several times during program execution. A main module is written and tested. The main module is used to call the subfunction modules in the sequence required to accomplish the overall function of the program.

Modular programming is supported by the MC68020 call module (CALLM) and the return from module (RTM) instructions.

The CALLM instruction creates a module stack frame on the stack (similar to an interrupt), stores the module state in that frame, and points to the address of a module descriptor (in the effective address) which contains control information for the entry into the called module.

The CALLM instruction loads the processor with the data provided by the module descriptor. Thus the CALLM instruction does not directly access the program module. Rather, it indirectly calls the routine via the module descriptor whose contents are maintained by the operating system. The module descriptor can be thought of as a gateway through which the calling program must gain access. The module being called can be thought of as a subroutine that is to act upon the arguments passed to it.

The CALLM instruction syntax is

CALLM # data, (EA)

To use this instruction, an immediate data value (data 0–255) must be specified to indicate the number of bytes of argument parameters to be passed to the called module. The effective address (EA) that points to the external module descriptor must also be included. The program containing the CALLM instruction must define a RAM storage area for the module

TABLE 6.12 CALLM and RTM Instructions

Instruction	Operand size	Operation	Notation
CALLM	None	Save current module state on stack; load new module state from destination	CALLM data, (EA)
RTM	None	Reload saved module state in stack frame; place module data area pointer in Rn	RTM Rn

descriptor to reside in. The address of the module descriptor is known to the operating system at "run time". It is the operating system that loads the starting address of the library routine to be called into the module descriptor.

The return from module (RTM) instruction is the complement of the CALLM instruction. The RTM syntax is RTM Rn.

Register Rn (Rn can be data or address register) represents the module data area pointer. The RTM instruction recovers the previous module state from the stack frame and returns program execution in the calling module. The processor state (program counter, status word) comes from the previously stacked data. The operation of the module instructions is shown in Table 6.12.

Figure 6.11 shows an overview of the CALLM and RTM operation. When the CALLM instruction executes, it creates a module call stack frame and saves the current module state in that frame. It then loads the new module state from the module descriptor. The start address fetched from the module descriptor points to the module entry word of the library routine. The second word of the routine is its first op word. The last instruction of the routine is the RTM instruction.

The RTM instruction reloads the saved module state in the created stack frame, and user program execution continues with the next instruction.

6.6.3 CAS Instructions

The MC68020 compare and swap (CAS and CAS2) instructions provide support for multitasking and multiprocessing. The compare and swap instructions are used when several processors must communicate through a common block of memory, when globally shared data structures (such

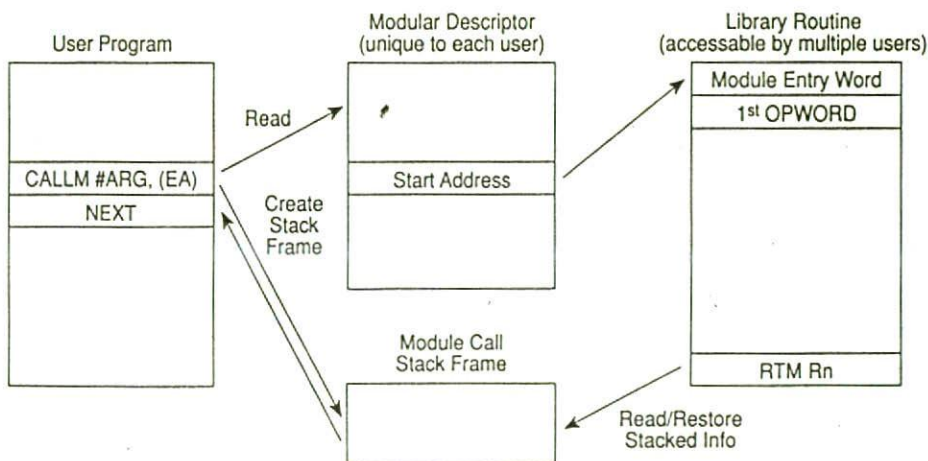


FIGURE 6.11 Overview of the CALLM and RTM operation.

as counters, stack pointers, or queue pointers) must be securely updated, or when multiple bus cycles may be required. A typical application is a counting semaphore (shared incrementer). Another application of the compare and swap instructions would be to manipulate pointers for system stacks and queues that use linked lists when a new item (element) is inserted or an existing element is deleted.

The CAS instruction uses two registers (Dc and Du) and the address (EA) of the globally shared operand variable (or pointer) to be protected. CAS operates on byte, word, and long word operands. The assembly level programming notation for the CAS instruction is

CAS.S (.B or .W or .L)Dc, Du, (EA)

The CAS instruction first compares the old "fetched" starting pointer in register Dc with the present starting pointer in the variable's original location (EA) to see if another task accessed the variable pointer and changed it while the first task was using it; if the two compared pointer values Dc and (EA) are equal (that is, pointer contents (EA) remain unchanged by interrupting tasks), then CAS passes the updated pointer value (located in Du) to the destination operand (EA) and sets the equal condition code flag in the CCR to Z = 1.

If the two compared values are not equal (the EA pointer has been changed by an interrupting task), then CAS copies the new changed pointer value contents of (EA) to register Dc and clears the equal condition code flag in the CCR to Z = 0.

A condensed version of the CAS operation is shown below:

```

CAS Dc, Du, (EA)
1.  (EA) - Dc → cc
2.  IF (EA) = Dc
    THEN Du → (EA)
    ELSE (EA) → Dc
  
```

Next, application of the CAS for queue insertion will be discussed.

Figure 6.12a shows how to insert a new entry in a queue using the MOVE instruction.

In a single user/single task environment the above can be accomplished by the following instruction sequence:

```

MOVE.L HEAD, (NEXT, A1)
MOVE.L A1, HEAD
  
```

But in a multiuser/multitask environment, the above instruction sequence may not accomplish the task. For example, more than one user may attempt to insert a new entry in an

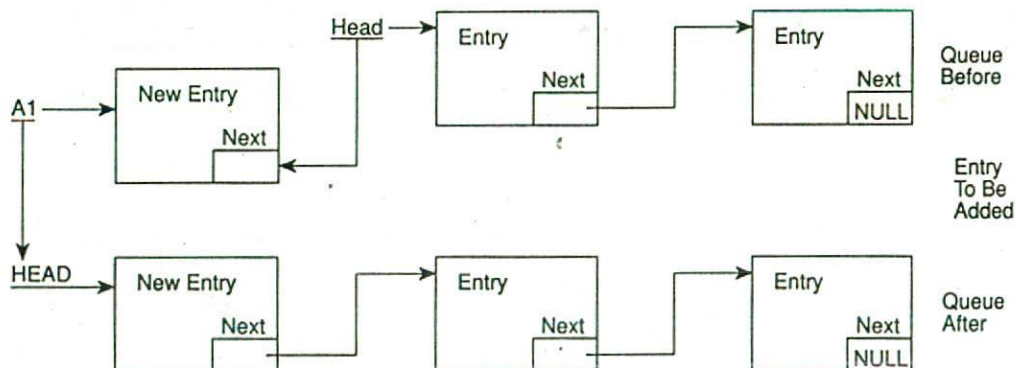


FIGURE 6.12a Inserting a new entry in queue (single user/single task).

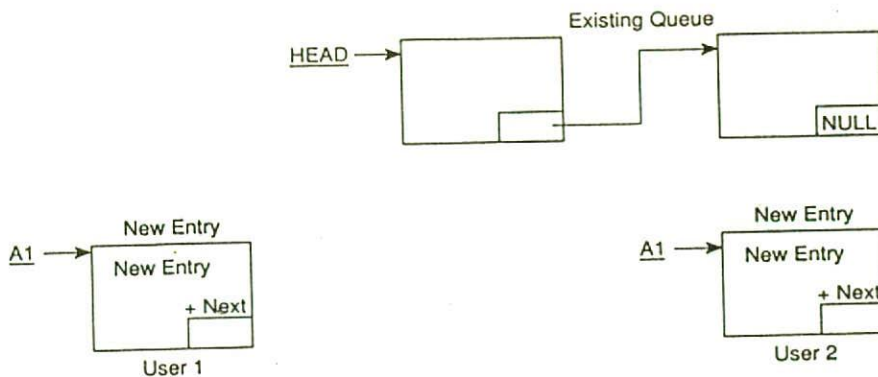


FIGURE 6.12b Two users attempting to insert a new entry into an existing space.

existing queue. Suppose that after user 1 executes `MOVE.L HEAD, (NEXT, A1)`, user 2 executes the instruction sequence

```
MOVE.L HEAD, (NEXT, A1)
MOVE.L A1, HEAD
```

and inserts a user 2 new entry in the existing queue before user 1 gets to `MOVE.L A1, HEAD`. This situation is depicted in Figures 6.12b and b.12c. This situation can be avoided by using the CAS instruction. The user 2 entry gets lost if the MOVE instruction is used for insertion. In Figure 6.12c, the HEAD (known to user 1 at the start) gets changed between the time user 1 established the forward link by the time user 1 updated HEAD.

The following instruction sequence uses CAS to insert a new entry in a queue in a multiuser/multitask environment:

```
MOVE.L HEAD, D0      ; Capture current HEAD
MOVE.L A1, D1        ; Need value in D0 for CAS
LOOP MOVE.L D0, (NEXT, A1) ; Establish forward link
```

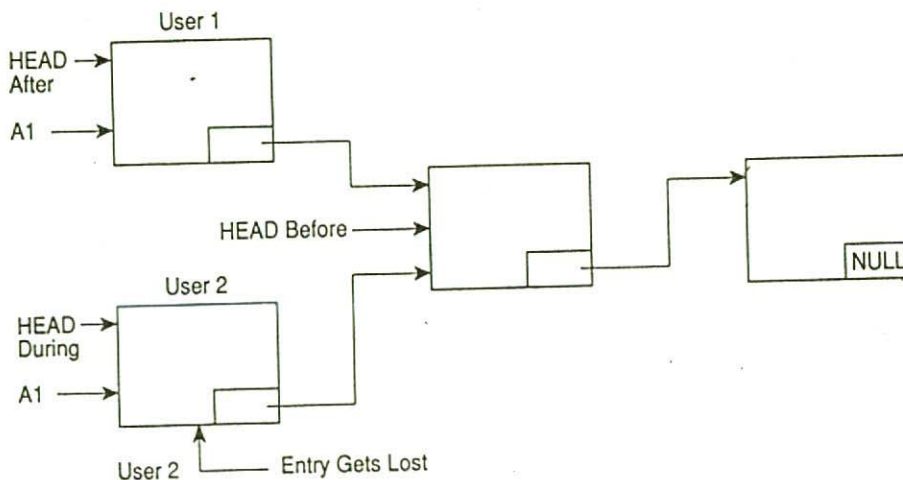


FIGURE 6.12c Insertion of entries into an existing queue in a multiuser environment using MOVE (situation to avoid).

Case 1: No Intervention

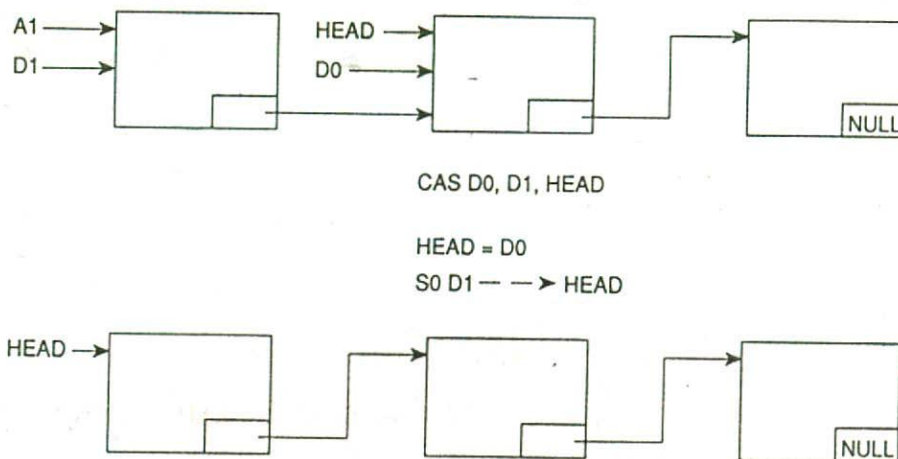


FIGURE 6.12d Inserting an element in a queue using CAS without intervention.

```

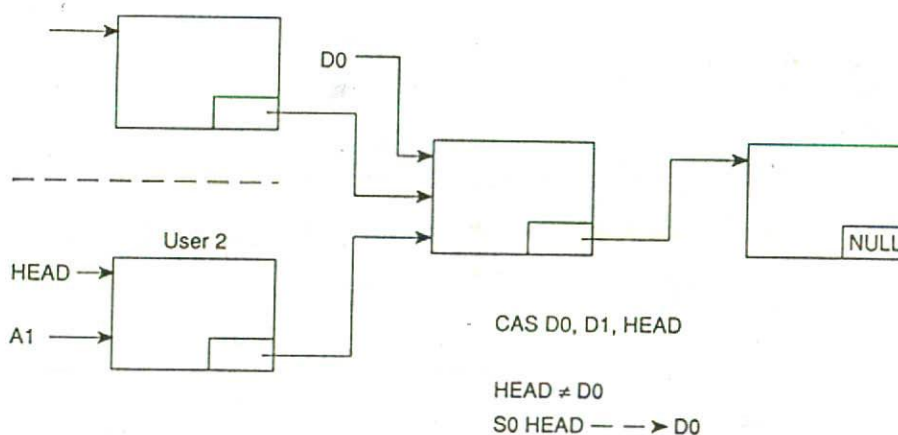
CAS.L D0, D1, HEAD    ; If HEAD unchanged, update
BNE LOOP              ; You have new HEAD in D0, try
                        ; again

```

Figures 6.12d and 6.12e illustrate the use of CAS in the above instruction sequence, respectively, in a system with no intervention and simultaneous insertion by multiusers.

The CAS instruction cannot be interrupted. This ensures secure updates of variables in a multiprocessing system. Executing CAS causes the read-write-modify (RMC) pin to be asserted, which locks the bus. Other bus masters in the system must wait for RMC to be deasserted before they can take control of the bus. The CAS instructions read, compare, and

Case 2: "Simultaneous Insert"



The program tries again, from loop. First instruction not necessary as CAS did it.

FIGURE 6.12e Simultaneous insertion of elements.

store operations are performed while the bus is locked. This prevents other processors from interfering with the instruction while it performs its compare and swap operation.

Instruction CAS2 is identical to CAS except that it can be used to compare and update dual operands within the same indivisible cycle. The CAS2 instruction operates on word or long word operands. The notation for the CAS2 instruction is CAS2 .W Dc1:Dc2, Du1:Du2, (Rn1):(Rn2). With the CAS2 instruction, both comparisons must show a match for the contents of the update registers Du1 and Du2 to be stored at the operands' destination addresses in memory pointed to by the registers Rn1 and Rn2. If either comparison fails to match, both destination operands obtained from memory are copied to the compare registers Dc1 and Dc2. The CAS2 instruction memory references to operand destination addresses must be specified using register indirect addressing with either a data or address register used as a pointer to memory.

The condensed version of CAS2 Dc1:Dc2, Du1:Du2, (EA1):(EA2) operation is given below:

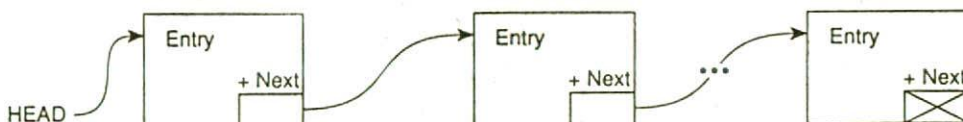
```
If (EA1) = Dc1 and (EA2) = Dc2, then
    Du1 → (EA1) and Du2 → (EA2);
else (EA1) → Du1 and (EA2) → Du2
```

Example 6.21

Write a 68020 instruction sequence to delete an element from a linked list using the CAS2 instruction.

Solution

```
LEA HEAD, A0          ; Load address of head pointer
                       ; into A0
MOVE.L (A0), D0        ; Move value of head pointer
                       ; into D0
LOOP  TST.L D0          ; Check for null pointer
      BEQ EMPTY        ; If empty, no deletion required
      LEA (NEXT, D0.L), A1 ; Load address forward of link
                       ; into A1
      MOVE.L (A1), D1    ; Put value of forward link
                       ; in D1
      CAS2.L D0:D1, D1:D1, ; If no change, update head
            (A0):(A1)    ; and forward pointer
      BNE LOOP          ; D0 has new head, try again.
EMPTY - - -
```



After deleting an element:

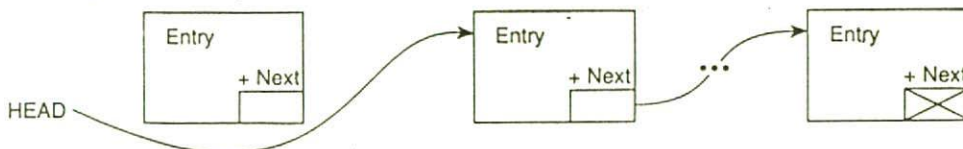


TABLE 6.13 Summary of the TAS, CAS, and CAS2 Instructions

Instruction	Operand size	Operation	Notation
TAS	8	Destination $\rightarrow 0 \rightarrow$ CCR 1 \rightarrow bit 7 of destination	TAS (EA)
CAS	8,16,32	Destination $\rightarrow Dc \rightarrow$ CCR if Z, then $Du \rightarrow$ destination; else destination $\rightarrow Dc$	CAS.S Dc, Du, (EA)
CAS2	16,32	dest1 $\rightarrow Dc1 \rightarrow$ CCR; if Z, dest2 $\rightarrow Dc2 \rightarrow$ CCR; if Z, $Du1 \rightarrow$ dest 1; $Du2 \rightarrow$ dest2; else Dest 1 $\rightarrow Dc1$, dest 2 $\rightarrow Dc2$	CAS2.S Dc1:Dc2,Du1: Du2,(Rn1):(Rn2)

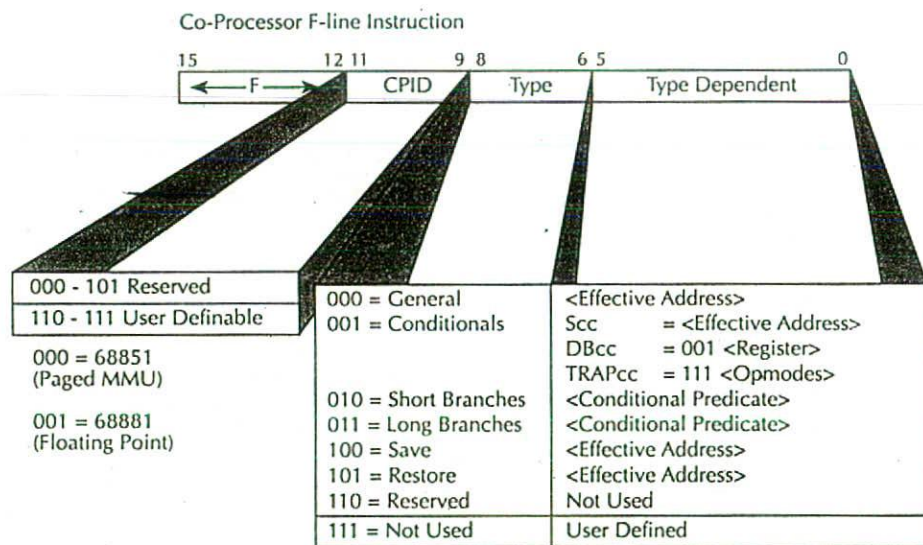
Note: S = B, W, or L.

Table 6.13 summarizes the operation of the CAS, CAS2, and TAS instructions.

6.6.4 Coprocessor Instructions

Table 6.14 lists the MC68020 coprocessor instructions. These instructions are available on the MC68020 system when a coprocessor such as the MC68881 (floating point) or MC68851 (paged memory management unit) is interfaced to the system. Note that cp in these instructions is replaced by F for floating point (MC68881) or P for paged (MC68851), depending on the coprocessor.

When MAIN obtains an F-line op word, it cooperates with a coprocessor to complete execution of the instruction



The F or P provides the 3-bit cp-id in the instruction. The GEN is replaced by the specific operation. For example, rMOVE is the MOVE instruction for the floating-point coprocessor. Upon execution of the cpGEN instruction, the MC68020 passes a command word to the coprocessor. The coprocessor then specifies the general data processing and movement instructions. The coprocessor finds the specific operation from the command word which follows the F-line instruction word. If the instruction requires an effective address for an operand to be fetched or stored, the third word follows the command word containing this effective address.

Note that F-line instruction word means that the high four bits of the instruction word are 1111. Normally, the coprocessor defines the specific instances of this instruction to provide

TABLE 6.14 Coprocessor Instructions

Instruction	Operand size	Operation	Notation
cpGEN	User defined	Pass command word to coprocessor and respond to coprocessor primitives	cpGEN (parameters defined by coprocessors)
cpBcc	16,32	If cpec true, then $PC + d \rightarrow PC$	cpBcc (label)
cpDBcc	16	If cpec false, then ($Dn - 1 \rightarrow Dn$; if $Dn \neq 1$, then $PC + d \rightarrow PC$)	cpDBccDn, (label)
cpScc	8	If cpec true, then 1's \rightarrow destination, else 0's \rightarrow destination	cpScc (EA)
cpTRAPcc	None	If cpec true, then TRAP	cpTRAPcc
	16,32		cpTRAPcc # (data)
cpSAVE*	None	Save internal state of coprocessor	cpSAVE (EA)
cpRESTORE*	None	Restore internal state of coprocessor	cpRESTORE (EA)

* Privileged instructions for operating system context switching (task switching) support.

• The 68020 (MAIN) and the coprocessor (CO) each do what they know how to do best

MAIN: Tracks instruction stream

Takes exceptions

Takes branches

CO: Does graphics manipulations

Calculates transcendentals, floating point

Does matrix manipulations

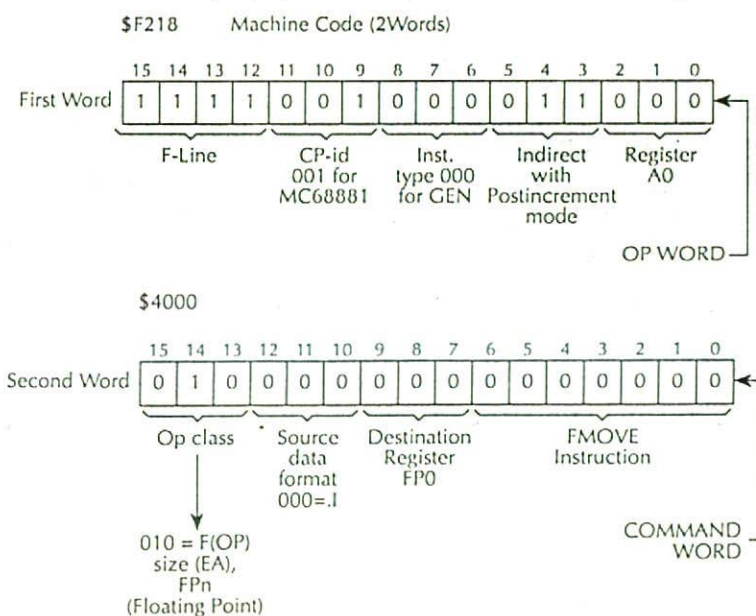
its instruction set. The condition codes may be modified by the coprocessors. An example of a cpGEN instruction for the MC68881 is

FMOVE.L (A0) +, FP0

In this instruction F replaces cp and MOVE replaces GEN in the cpGEN format. This instruction moves 4 bytes from the MC68020-based microcomputer memory, starting with a location addressed by the contents of A0 to low 32 bits of the 80-bit register (FP0) in the MC68881; A0 is then incremented by 4.

In the above for each type (bits 6–8), type dependent (bits 0–5) specifies effective address, conditional predicate, etc. For example, for general instructions (bits 8, 7, 6 = 000), type dependent (bits 0–5) specifies (ea).

The instruction FMOVE.L (A0) +, FP0 contains two words in memory as follows:



Therefore, the **FMOVE.L (A0) +, FP0** contains two words in memory with the first word \$F218 as the op word and the second word \$4000 as the command word for the coprocessor.

The "cpBcc displacement" is the coprocessor conditional branch instruction. If the specified coprocessor condition is satisfied, program execution continues at location (PC) + displacement; otherwise the next instruction is executed. The displacement is a two's complement integer which may be either 16 or 32 bits. The coprocessor determines the specific condition from the condition field in the operation word. A typical example of this instruction is

FBEQ.L START

The cpDBcc (label) instruction works as follows. If cpcc is false, then $D_n - 1 \rightarrow D_n$ and if $D_n \neq -1$, then $PC + d \rightarrow PC$, or if $D_n = -1$, then next instruction is executed. On the other hand, if cpcc is true, then the next instruction is executed. The coprocessor determines the specific condition from the condition word which follows the operation word. A typical example of this instruction is

FDBNE.W START

The operand size is 16 bits.

The cpScc.B (EA) instruction tests a specific condition. If the condition is true, the byte specified by (EA) is set to true (all ones); otherwise that byte is set to false (all zeros). The coprocessor determines the specific condition from the condition word which follows the operation word. (EA) in the instruction can use all modes except An, immediate, (d16, PC), (d8, PC, Xn), (bd, PC, Xn), and (bd, PC, Xn, od). An example of this instruction is

FSEQ.B \$8000 1F20.

The cpTRAPcc or cpTRAPcc # data checks the specific condition on a coprocessor. If the selected coprocessor condition is true, the MC68020 initiates exception processing. The vector number is generated to reference the cpTRAPcc exception vector; the stacked PC is the address of the next instruction. If the selected condition is false, no operation is performed and the next instruction is executed. The coprocessor determines the specific condition from the word which follows the operation word. The user-defined optional immediate data (third word of the machine code for cpTRAPcc # data) is used by the trap handler routine. A typical example of this instruction is

FTRAPEQ

cpTRAPcc is unsized and cpTRAPcc # data has word and long word operands. cpSAVE and cpRESTORE are privileged instructions. Both instructions are unsized. cpSAVE (EA) saves the internal state of a coprocessor. (EA) can be predecrement on all alterable control addressing modes. This instruction is used by an operating system to save the context (internal state) of a coprocessor. The 68020 initiates a cpSAVE instruction by reading an internal register.

The cpRESTORE (EA) instruction restores the internal state of a coprocessor. (EA) can use postincrement or control addressing modes. This instruction is used by an operating system to restore the context of a coprocessor for both the user-visible and the user-invisible state.

6.7 MC68020 Cache/Pipelined Architecture and Operation

The MC68020 has a 256-byte direct-mapped instruction cache organized as 64 long word entries. Each cache entry consists of a tag field made up of the upper 24 address bits, the FC2 (user/supervisor) value, one valid bit (V), and 32 bits (two words) of instruction. Figure 6.13 shows the MC68020 on-chip cache organization. The 68020 cache only stores instructions; data or operands are fetched directly from main memory as needed.

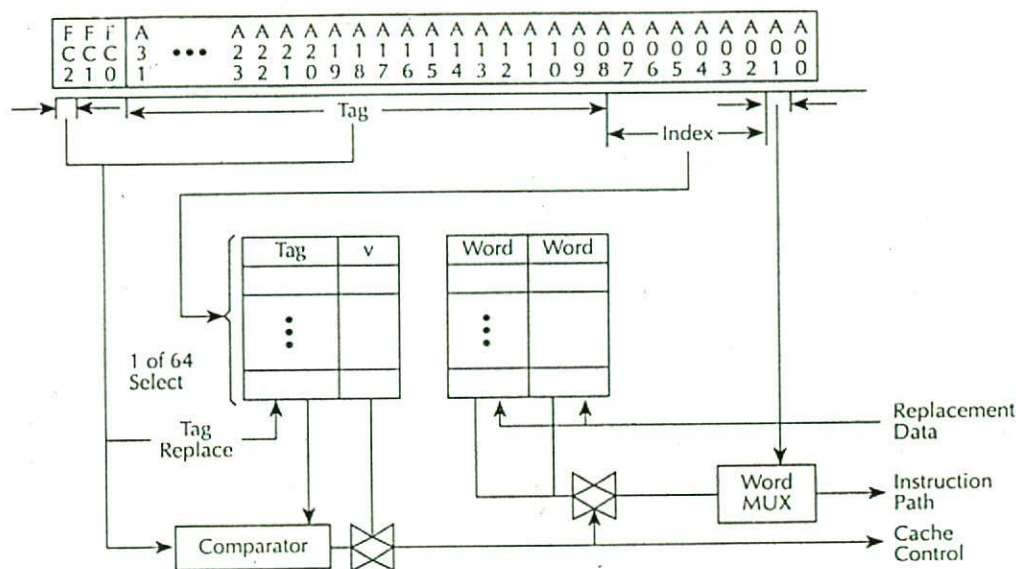


FIGURE 6.13 MC68020 on-chip cache organization.

When the 68020 fetches an instruction, it checks the cache first to determine if the word required is in the cache. First, one of 64 entries is selected by using the index field (A2–A7) of the access address as an index into the cache. Next, the 68020 compares address bits A31–A8 and function code bit FC2 with the 24-bit tag of the selected entry. If the function code and the address bit match, the 68020 sets the valid bit for the cache entry. This is called a cache hit. Finally, the 68020 uses address bit A1 to select the proper instruction word from the cache memory. If there is no match or valid bit is clear, a cache miss occurs and the instruction is fetched from external memory and put into the cache.

The MC68020 uses a 32-bit data bus and fetches instructions on long word address boundaries. Hence, each 32-bit fetch brings in two 16-bit instruction words which are then written into the on-chip cache. Subsequent prefetches will find the next 16-bit instruction word already present in the cache, and the related bus cycle is saved. Even when the cache is disabled, the subsequent prefetch will find the bus controller still holds the two instruction words and can satisfy the prefetch, again saving the related bus cycle. The bus controller provides an instruction hit rate of up to 50% even with the on-chip cache disabled.

Only the CPU uses the internal cache, so users have no direct access to the entries. However, several instructions allow the user to control the cache or dynamically disable it through the external hardware cache disable pin (CDIS). Typically, it is used by an emulator or bus state analyzer to force all bus cycles to be external cycles. The processor's two cache registers (CACR, CARR) can be programmed while in the supervisor mode by using the MOVEC instruction. Enabling, disabling, freezing, or clearing the cache is carried out by the cache control register (CACR). The CACR also allows the operating system to maintain and optimize the cache. The cache control (CACR) is shown in Figure 6.14.



FIGURE 6.14 Cache control register format. C = clear cache, CE = clear entry, F = freeze cache, and E = enable cache.

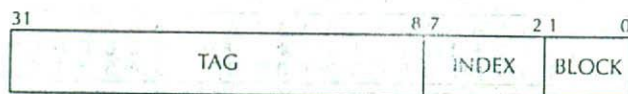


FIGURE 6.15 Cache address register format.

The clear cache (C) bit of the CACR is used to invalidate all entries in the cache. This is termed as “flushing the cache”. Setting the (C) bit causes all valid bits (V) in the cache to be cleared, thus invalidating all entries.

The Clear Entry (CE) of the CACR is used in conjunction with the address specified in the cache address register (CAAR). When writing to and setting the (CE) bit, the processor uses the CAAR index field to locate the selected address in the CAAR and invalidate the associated entry by clearing the valid bit.

The freeze cache (F) bit of the CACR keeps the cache enabled, but cache misses are not allowed to update the cache entries. This bit can be used by emulators to freeze the cache during emulation execution. It could be used to lock a critical region of the code in the cache after it has been executed, providing the cache is enabled and the freeze bit is cleared. The enables cache (E) bit is used for system debug and emulation. This bit allows the designer to operate the processor with the cache disabled as long as the (E) bit remains cleared.

The cache address register (CAAR) format is shown in Figure 6.15.

The CAAR is used by the MC68020 to provide an address for the Clear Entry (CE) function as implemented in the CACR. The index portion of this register is used to specify which one of 64 entries to invalidate by clearing the associated cache entry valid bit (V).

Although the micromachine of the MC68020 is highly pipelined, the predominant pipeline mechanism is a three-stage instruction pipeline. Figure 6.16 shows the MC68020 pipeline organization. The pipeline is completely internal to the processor and is used as part of the

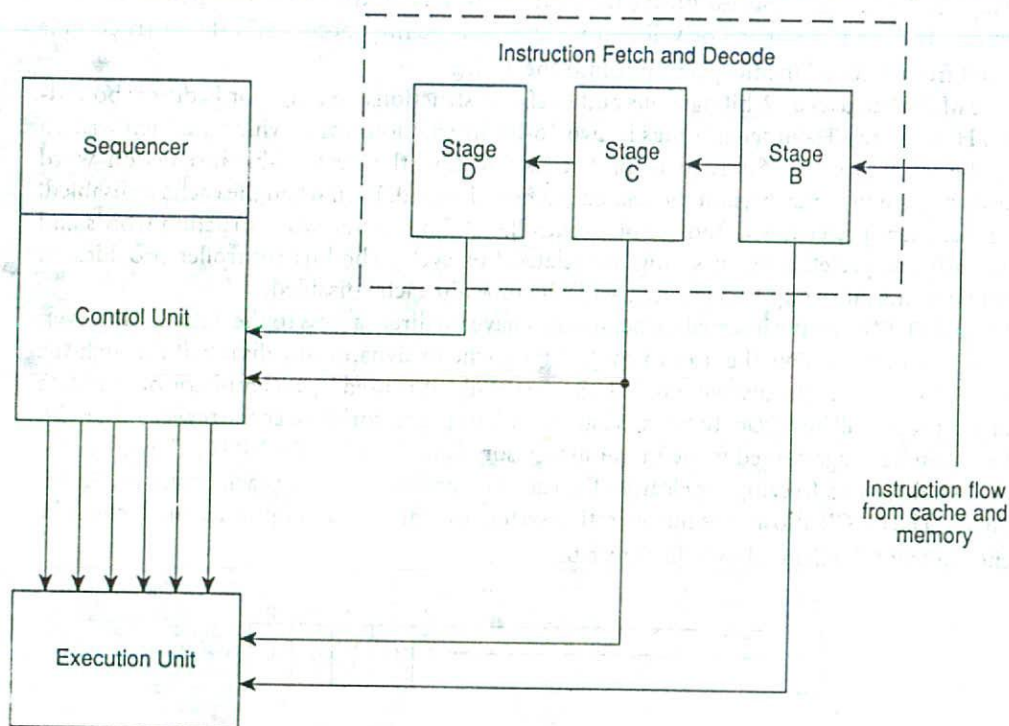


FIGURE 6.16 MC68020 pipeline.

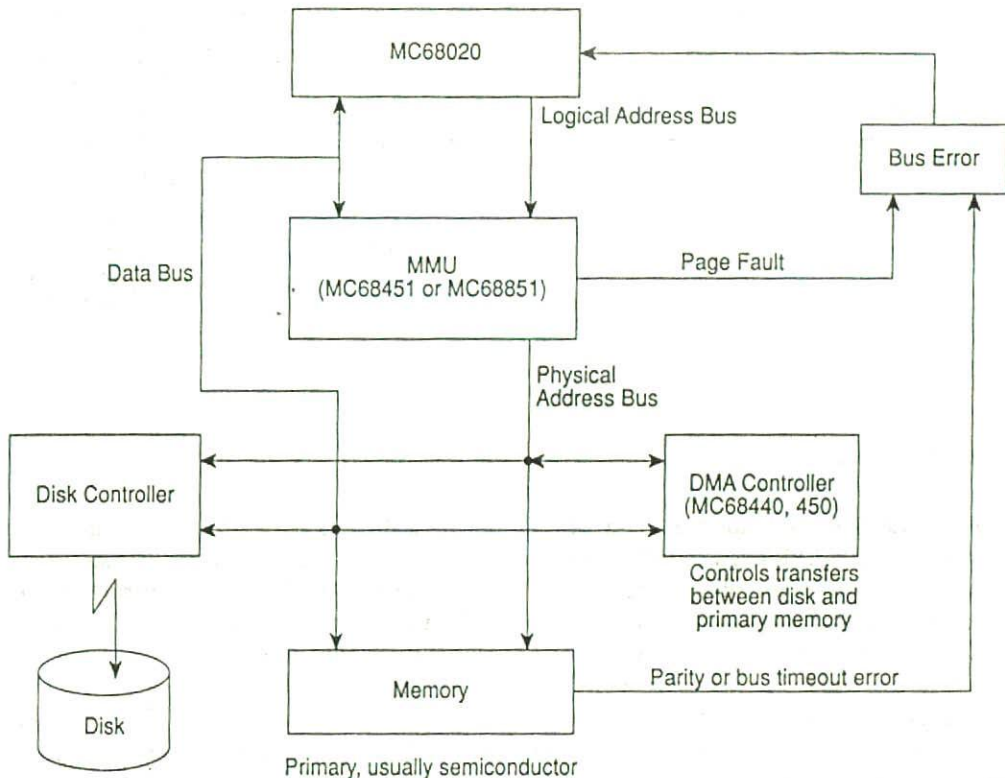


FIGURE 6.17 Typical virtual memory system, minimum configuration.

instruction fetching and decoding circuitry. Instructions from the on-chip cache, or from external memory (if the cache is disabled), go into the first stage of the pipeline and synchronously pass through the following two stages.

The pipeline output gives the 68020's control and execution unit a completely decoded instruction. The 68020 loads data and other operands into the pipeline so they are ready for immediate use. The pipeline speeds 68020 operation by making information available immediately. The benefit of the pipeline is to allow concurrent operations (parallelism) for up to three words of a single instruction or for up to three consecutive one-word instructions. Therefore, the performance benefits of a pipeline are maximized during the execution of in-line code.

6.8 MC68020 Virtual Memory

Virtual memory is a technique that allows all user programs executing on a processor to behave as if each had the entire 4-GB addressing range of the MC68020 at its disposal, regardless of the amount of physical memory actually present in the system. Virtual memory can be supported by providing a limited amount of high-speed physical memory that can be accessed directly by the processor while maintaining an image of a much larger "virtual" memory on a secondary storage device, such as high-capacity disk drives. Figure 6.17 shows a minimal system configuration for a typical virtual memory system. Also, any given instruction must be able to be aborted and restarted. When a processor attempts to access a location in the virtual memory map that is not resident in physical memory (this is called a "page fault"), the access to that location is temporarily suspended while data are fetched from secondary storage and placed into physical memory. Page faults force a trap to the bus error exception vector.

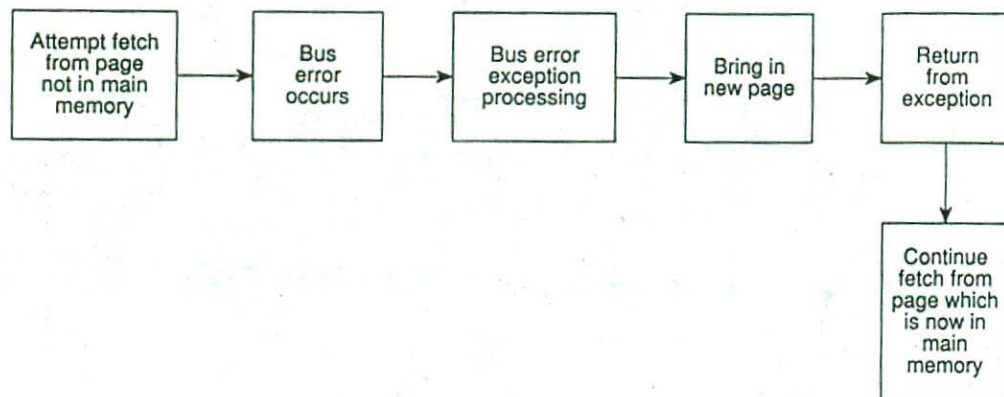


FIGURE 6.18 Bus error sequence.

The MC68020 processor has the abort capability via the bus error (BERR) input to the processor. When BERR is asserted, exception processing causes the processor to save sufficient information to allow complete restoration of the faulted instruction. The faulted instruction will be recovered by the instruction continuation method, in which the faulted instruction is allowed to complete execution from the point of the fault. Instruction continuation is crucial for supporting virtual I/O devices in memory-mapped I/O systems. To handle instruction continuation properly, the processor must save certain internal processor information on the stack prior to running the exception code and return this information to the processor after executing the return-from-exception (RTE) instruction. Figure 6.18 shows the BUS error processing sequence that occurs when attempting to fetch an instruction from a page not in the main memory.

6.9 MC68020 Coprocessor Interface

The MC 68020's coprocessor interface is capable of extending the MC68020 instruction set and supporting new data types.

The interface between the main processor and a coprocessor is transparent to the user. The programmer does not have to be aware that a separate piece of hardware is executing some of the program code sequence. Hardware-implemented microcode within the MC68020 handles coprocessor interfacing so that a coprocessor can provide its capabilities to the programmer without appearing as external hardware, but rather as a natural extension to the main processor architecture.

When communicating with a coprocessor, the MC68020 executes bus cycles in CPU memory space to access a set of Coprocessor Interface Registers (CIRs). Table 6.15 shows the separate

TABLE 6.15 Coprocessor Interface Register

Register	Function	R / W
Response	Requests action from CPU	R
Control	CPU directed control	W
Save	Initiate save of internal state	R
Restore	Initiate restore of internal state	R / W
Operation word	Current coprocessor instruction	W
Command word	Coprocessor specific command	W
Condition word	Condition to be evaluated	W
Operand	32-bit operand	R / W
Register select	Specifies CPU register or mask	R
Instruction address	Pointer to coprocessor instruction	R / W
Operand address	Pointer to coprocessor operand	R / W

31	15	0
00	Response	Control
04	Save	Restore
08	Operation Word	Command
0C	(Reserved)	Condition
10	Operand	
14	Register Select	(Reserved)
1B	Instruction Address	
1C	Operand Address	

FIGURE 6.19 Coprocessor interface register set map.

coprocessor interface registers along with their functions. As shown within this interface register set, the various registers are allocated to specific functions required for operating the coprocessor interface. There are registers specifically for passing information such as commands, operands, and EAs. Other registers are allocated for use during a context switch operation.

Figure 6.19 shows the coprocessor interface register set map and communication protocol between the main processor and the coprocessor necessary to execute a coprocessor instruction. The MC68020 implements the CIR communication protocol automatically, so the programmer is only concerned with the coprocessor's instruction and data type extensions to the MC68020 programmer's model.

Figure 6.20 shows the CIR set(s) (that is, more than one coprocessor) address map in CPU space.

The MC68020 indicates that it is accessing CPU memory space by encoding the function code lines high (FC0–FC2 = 111_2). Thus, the CIR set is mapped into CPU space the same way that a peripheral interface register set is generally mapped into data space. The address bus then selects the desired coprocessor chip.

Encoding of the address bus during coprocessor communication is shown in Figure 6.21. By using the cp-ID field on the address bus, up to eight separate coprocessors can be interfaced concurrently to the MC68020. Figure 6.21 also shows how this can be done. Interfacing to these separate coprocessors is a matter of decoding the relevant cp-ID field (address lines A13–A15) and the corresponding CPU space type field (address lines A16–A19) encoded within the coprocessor instruction, so that the MC68020 communicates with the relevant register set in CPU space.

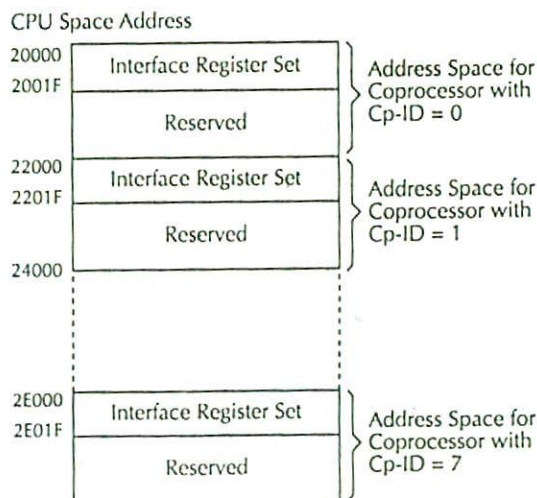


FIGURE 6.20 Coprocessor address map in CPU space.

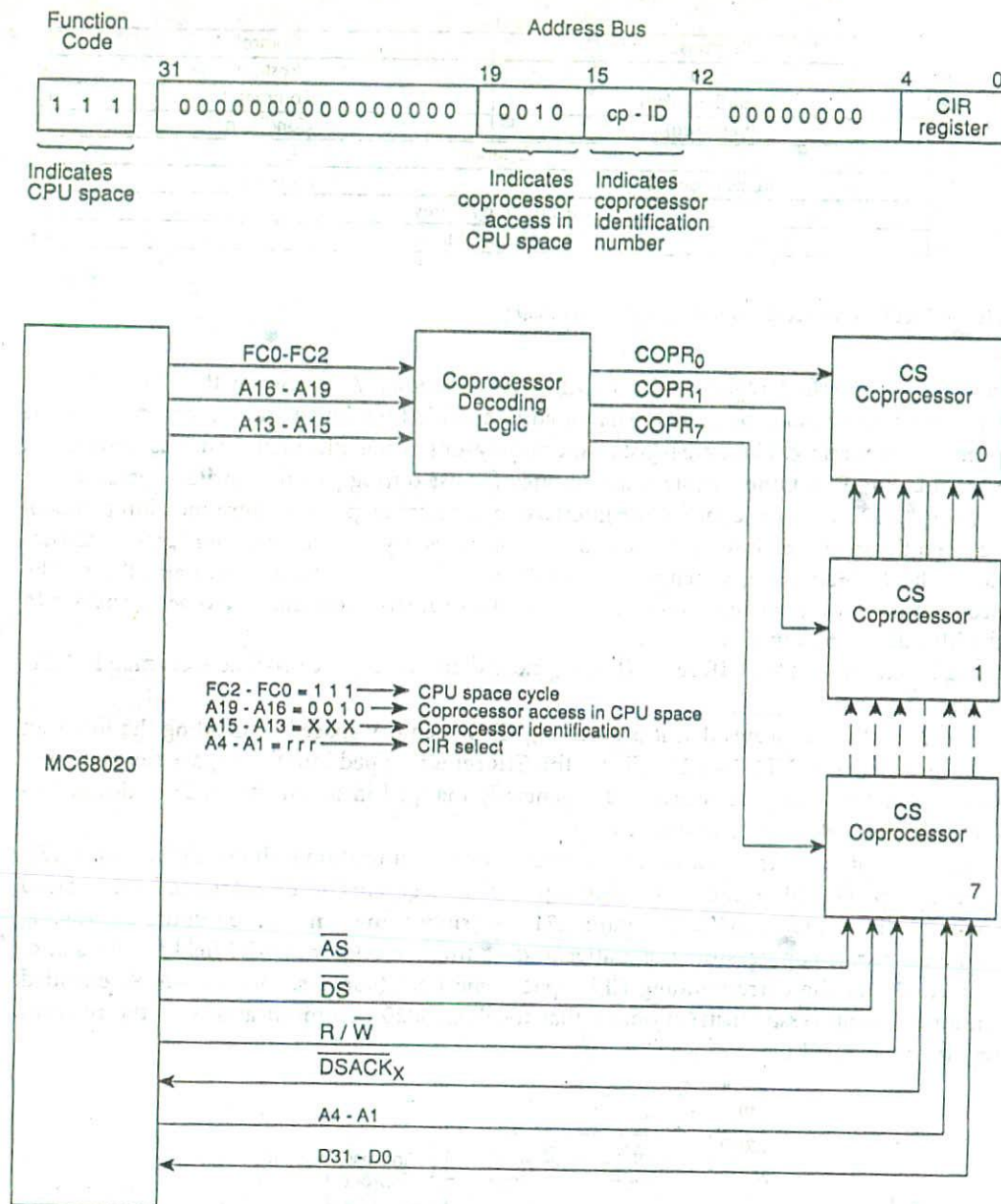


FIGURE 6.21 68020 interface to coprocessors. Coprocessor communication address bus encoding and multicoprocessor address decoding example.

6.9.1 MC68881 Floating-Point Coprocessor

The MC68881 HCMOS floating-point coprocessor implements IEEE standards for binary floating-point arithmetic. When interfaced to the MC68020, the MC68881 provides a logical extension to the MC68020 for performing floating-point operations. The MC68882 is an upgrade of the MCC68881 and provides in excess of 15 times the performance of the MC68881. It is a pin and software compatible upgrade of the MC68881.

The 68882 provides higher execution speed than the 68881. The 68882 contains a special on-chip hardware that converts between binary and extended floating-point numbers at a much higher speed than the 68881. Both chips are packaged in 68-pin PGA (Pin Grid Array).

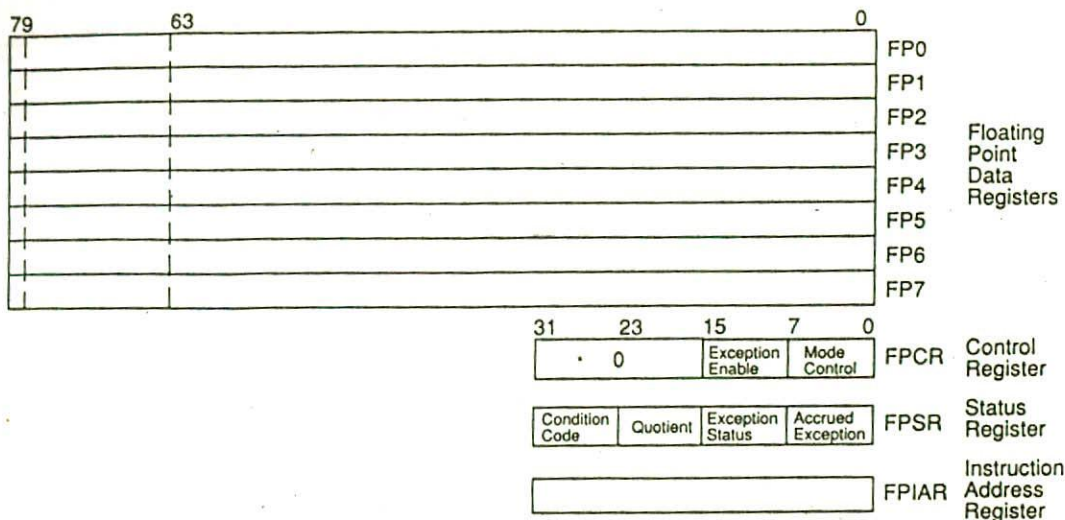


FIGURE 6.22a MC68881 programming model.

A summary of the MC68881 features is listed below:

- Eight general-purpose floating-point data registers, each supporting an 80-bit extended precision real data format (a 64-bit mantissa, plus a sign bit and a 15-bit signed exponent)
- A 67-bit arithmetic unit
- A 67-bit barrel shifter for fast shift operations
- instructions with 35 arithmetic operations
- Supports trigonometric and transcendental functions
- Supports seven data types: bytes, word, long word integers; single, double, and extended precision real numbers; and packed binary coded decimal string real numbers
- 22 constants including π , e , and powers of 10

The MC68881 is a non-DMA-type coprocessor which uses a subset of the general-purpose coprocessor interface supported by the MC68020. The MC68881 programming model is shown in Figure 6.22a.

The MC68881 programming model includes the following:

- Eight 80-bit floating-point registers (FP0–FP7). (These general-purpose registers are analogous to the MC68020 D0–D7 registers.)
- A 32-bit control register containing enable bits for each class of exception trap and mode bits, to set the user-selectable rounding and precision modes
- A 32-bit status register containing floating-point condition codes, quotient bits, and exception status information
- A 32-bit Floating-Point Instruction Address Register (FPIAR) containing the MC68020 memory address of the last floating-point instruction that was executed. (This address is used in exception handling to locate the instruction that caused the exception.)

The MC68881 can be interfaced as a coprocessor to the MC68020. Figure 6.22b provides the MC68020/68881 block diagram.

The MC68020 implements the coprocessor interface protocol in hardware and microcode. When the MC68020 encounters a typical MC68881 instruction, the MC68020 writes the

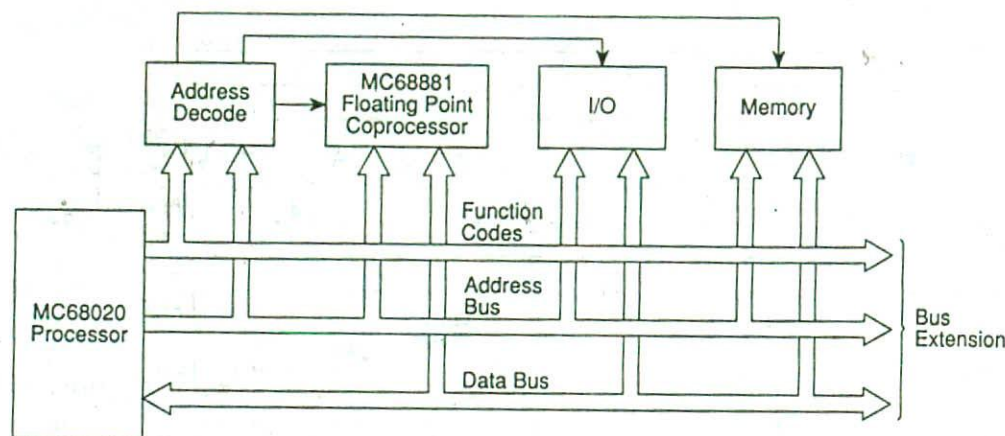


FIGURE 6.22b Typical coprocessor configuration.

instruction to the memory-mapped command CIR and reads the response CIR. In this response, the BIU (Bus Interface Unit) translates any additional action required of the MC68020 by the MC68881. Upon satisfying the coprocessor requests, the MC68020 can fetch and execute subsequent instructions.

The MC68881 supports the following data formats:

- Byte Integer (B)
- Word Integer (W)
- Long Word Integer (L)
- Single Precision Real (S)
- Double Precision Real (D)
- Extended Precision Real (X)
- Packed Decimal String Real (P)

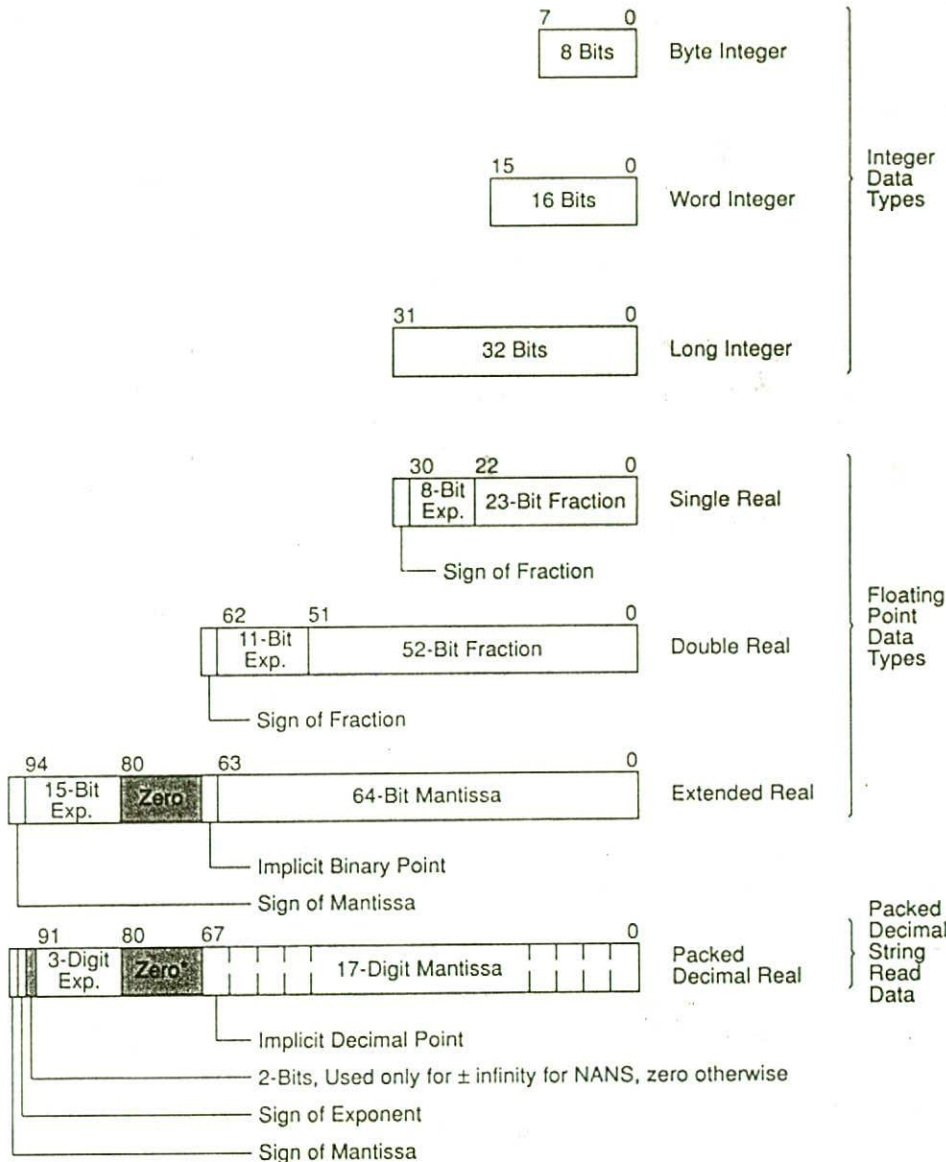
The capital letters included in parentheses denote the suffixes added to instructions in the assembly language source to specify the data format to be used.

Figure 6.23 shows the MC68881 data formats.

These data formats are defined by IEEE standards. Integer data types do not include any fractional part of the number. The real formats contain an exponent part and a mantissa part. The single-real and double-real formats provide the sign of the mantissa, 8- and 11-bit exponents, and 23- and 52-bit fractional parts, respectively.

The extended real format is 96 bits wide with a 15-bit exponent, 64-bit mantissa, and a sign bit for the mantissa. The packed decimal real is 96 bits wide which provides sign for both mantissa and exponent parts. It includes a 3-digit (12 bits for BCD) base-10 exponent and a 17-digit (68 bits) base-10 mantissa. This format contains two bits to indicate infinity or not-a-number (NAN) representations. Note that NAN is a symbolic representation of a special number or situation in floating-point format. NANs include all numbers with nonzero fractions with the format's maximum exponent. The infinity data types include the zero fraction and maximum exponent.

Whenever an integer is used in a floating-point operation, the integer is automatically converted to an extended-precision floating-point number before being used. For example, the instruction `FADD.W #2, FP0` converts the constant 2 to the floating-point number format in `FP0`, adds the two numbers, and then stores the result in `FP0`. This allows integer in floating-



* Unless a binary-to-decimal conversion overflow occurs

FIGURE 6.23 MC68881 data format summary.

point operations and also saves user memory, since an integer representation of a number is normally smaller than the equivalent floating-point representation.

The floating-point representation contains single-precision (32 bits), double-precision (64 bits), and extended-precision numbers (96 bits) as specified by the IEEE format. The single-precision and double-precision data types should be used in most calculations involving real numbers. The exponent is biased and the mantissa is in sign and magnitude form. Single and double precision require normalized numbers. Note that a normalized number has the most significant bit of the mantissa positioned such that the one lies to the left of the binary point.

Therefore, only the fractional part of the mantissa is stored in memory, which means that the most significant bit is implied and equal to one.

Extended-precision numbers are 96 bits wide but only 80 bits are used, and the unused 16 bits are for future expansion. Extended-precision numbers are for use as temporary variables, intermediate variables, or in situations where extra precision is required. Note that in extended precision, the 1 in $1 \cdot f$ is explicit.

For example, a compiler might select extended-precision arithmetic for determining the value of the right side of an equation with mixed sized data, and then convert the answer to the data type on the left side of the equation. Extended-precision data should not be stored in large arrays due to the amount of memory required by each number. As with other data types, the packed BCD strings are automatically converted to extended-precision real values when they are input to the MC68881. This permits packed BCD number to be used as input to any operation such as $FADD.P \# -2.012E + 18, FP1$.

The MC68881 does not include any addressing modes. If the 68881 requests the 68020 to transfer an operand via the coprocessor interface, the 68020 provides the addressing mode calculations requested in the instruction.

Floating-point data registers FP0-FP7 always contain extended-precision values. Also, all data used in an operation are converted to extended precision by the MC68881 before the operation is performed. The MC68881 provides all results in extended precision. The MC68881 instructions can be grouped into six types:

1. MOVE instructions between the MC68881 and memory on the MC68020
2. Monadic operations
3. Dyadic operations
4. Branch, set, or trap conditionally
5. Miscellaneous

6.9.1.a 68881 Data Movement Instructions

The 68881 floating-point data movement instructions include FMOVE, FMOVEM, and FMOVECR.

Some examples of FMOVE instruction include:

- FMOVE.B LOC, FP3 transfers an 8-bit integer from memory address LOC to FP3.
- FMOVE.X FP4, FP0 transfers the extended-precision floating-point number from FP4 to FP0.
- FMOVE.S FP3, D1 moves a single precision floating-point number from FP3 to D1.
- FMOVE.P FP5, (A0) transfers a BCD floating-point number from FP5 into 12 consecutive bytes of memory starting at the address pointed to by A0.
- FMOVE.X FP0, -(USP) subtracts 12 from USP and transfers the extended precision floating-point number from FP6 to the user stack.

When data are moved from memory or from a 68020 data register to a floating-point register, the 68020 converts the data to an extended floating-point number.

Also, the FMOVEM instruction from the 68881 floating-point register to memory or a 68020 data register convert data from the extended precision format to the form defined by the extension included with the instruction. Note that data are always represented in extended-precision form in a floating-point register.

The FMOVEM instruction transfers data between multiple 68881 floating-point registers and memory. Typical examples include:

```
FMOVEM (EA), FP1-FP4/FP6
FMOVEM FP0/FP1/FP5, (EA)
```

The registers FP0-FP7 are always moved as 96-bit extended data with no conversion.

Any combination of FP0-FP7 can be moved. (EA) can be control modes, predecrement or postincrement mode. For control or postincrement mode, the order of transfer is from FP7-FP0. For predecrement mode, the order of transfer is from FP0-FP7. Any combination of FPCR, FPSR, and FPIAR can also be moved by the MOVEM instruction. These registers are always moved in the order FPCR, FPSR, and FPIAR.

The 68881 FMOVECR #*\$mn*, FP*n* instruction reads an extended precision constant from a ROM within the 68881/68882 into a floating-point register. The numeric values of some of the constants are selected from the following:

<i>\$mn</i>	Constant
\$00	π
\$0B	$\text{Log}_{10} 2$
\$0C	e
\$0D	$\text{Log}_2 e$
\$0E	$\text{Log}_{10} e$
\$30	$\text{Log}_2 2$
\$31	$\text{Log}_2 10$
\$32	10^0
\$33	10^1
\$34	10^2
\$35	10^4
\$36	10^8

and so on.

For example the instruction FMOVECR .X #*\$0C*, FP6 moves the extended precision value of e into FP6.

6.9.1.b Monadic

Monadic instructions have a single input operand. This operand may be in a floating-point data register, memory, or in an MC68020 data register. The result is always stored in a floating-point data register. Typical examples include:

```

FTAN . (fmt) (EA), FPn
or
FTAN . X FPm, FPn
or
FTAN . X FPn

```

The FTAN instruction converts the source operand to extended precision (if necessary), computes the tangent of that number, and then stores the result in the destination floating-point data register.

Table 6.16a flowcharts the monadic function. Tables 6.16b and c list all MC68881 monadic instructions.

TABLE 6.16a Monadic Functions

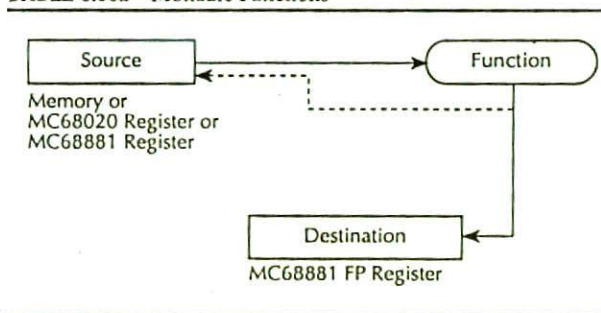


TABLE 6.16b MC68881 — Monadic Instructions

Transcendental functions	
FACOS	ARC COSINE
FASIN	ARC SINE
FATAN	ARC TANGENT
FATANH	HYPERBOLIC ARC TANGENT
FCOS	COSINE
FCOSH	HYPERBOLIC COSINE
FETOX	E TO THE X POWER
FETOXM1	E TO THE (X - 1) power
FLOG10	LOG TO THE BASE 10
FLOG2	LOG TO THE BASE 2
FLOGN	LOG BASE e OF X
FLOGNP1	LOG BASE e OF (X + 1)
FSIN	SINE
FSINCOS	SIMULTANEOUS SIN/COS
FSINH	HYPERBOLIC SINE
FTAN	TANGENT
FTANH	HYPERBOLIC TANGENT
FTENTOX	TEN TO THE X POWER
FTWOTOX	TWO TO THE X POWER

TABLE 6.16c MC68881 — Monadic Instructions

Nontranscendental functions	
FABS	ABSOLUTE VALUE
FINT	INTEGER PART
FNeg	NEGATE
FSQRT	SQUARE ROOT
FNOP	NO OPERATION (SYNCHRONIZE)
FGETEXP	GET EXPONENT
FGETMAN	GET MANTISSA
FTST	TEST

6.9.1.c Dyadic Instructions

Dyadic instructions have two input operands. The first input operand comes from a floating-point data register, memory, or an MC68020 data register. The second input operand comes from a floating-point data register. The second input is also the destination floating-point data register. Typical examples include:

```
FCMP.L (fmt) (EA), FFn
or
FCMP.X FFn, FFn
```

Table 6.17a flowcharts the dyadic function and Table 6.17b lists dyadic instructions.

TABLE 6.17a MC68881 Dyadic Functions

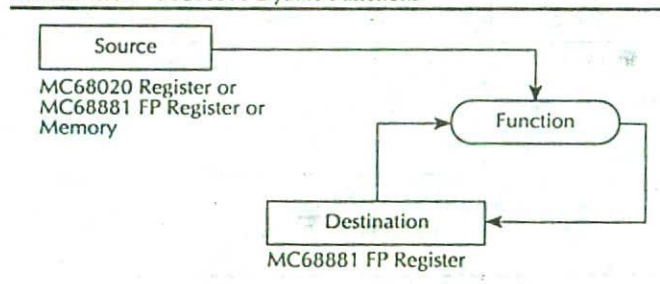


TABLE 6.17b MC 68881 Dyadic Instructions

FADD	ADD
FCMP	COMPARE
FDIV	DIVIDE
FMOD	MOD
FMUL	MULTIPLY
FREM	IEEE REMAINDER
FSCALE	SCALE EXPONENT
FSUB	SUBTRACT

6.9.1.d BRANCH, Set, or Trap-On Condition

These instructions are similar to those of the MC68020 except that move conditions exist due to the special values in IEEE floating-point arithmetic. When the MC68020 encounters a floating-point conditional instruction, it passes the instruction to the MC68881 for performing the necessary condition checkup. The MC68881 then checks the condition and tells the MC68020 whether the condition is true or false. The MC68020 then takes the appropriate action.

The MC68881 conditional instructions are

FBcc.W	displ	Branch
or .L		
FDBcc.W	displ	Decrement and branch
or .L		
FSc.W	displ	Set byte according to condition
or .L		
FTRAPcc.W	displ	Trap-on condition
or .L		

All the above instructions can have 16- or 32-bit displacement.

cc is one of the 32 floating-point conditional test specifiers. Table 6.18 lists a few of them.

TABLE 6.18 Floating-Point Conditional Test Specifiers

Mnemonic	Definition
F	False
EQ	Equal
NE	Not equal
T	True
SF	Signaling false
SEQ	Signaling equal
GT	Greater than
GE	Greater than or equal
LT	Less than
LE	Less than or equal
GL	Greater or less than
GLE	Greater, less, or equal
NGLE	Not (greater, less, or equal)
NGL	Not (greater or less)
NLE	Not (less or equal)
NLT	Not (less than)
NGE	Not (greater or equal)
NGT	Not (greater than)
SNE	Signaling not equal
ST	Signaling true

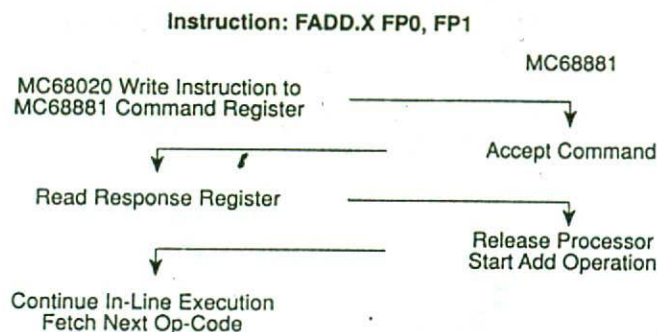


FIGURE 6.24a MC68020/MC68881 concurrence example.

6.9.1.e Miscellaneous Instructions

These instructions include moves to and from the status, control, and instruction address registers. The virtual memory instructions FSAVE and FRESTORE that save and restore the internal state of the MC68881 also fall into this category. These instructions include:

```

FMOVE (EA), FPcr
FMOVE FPcr, (EA)
FRESTORE (EA)
  
```

FPcr means floating-point control register. The MC68881 does not perform addressing mode calculations. The MC68020 carries out this calculation as specified in the instruction.

Typical addressing modes include immediate, postincrement, predecrement, direct, and the indexed/indirect modes of the MC68020. Some addressing modes are restricted for some instructions. For example, PC relative mode is not permitted for a destination operand.

The MC68881 can execute an instruction concurrently or nonconcurrently with the MC68020 depending on the instruction being executed. Figures 6.24a and b show examples of concurrence and nonconcurrency.

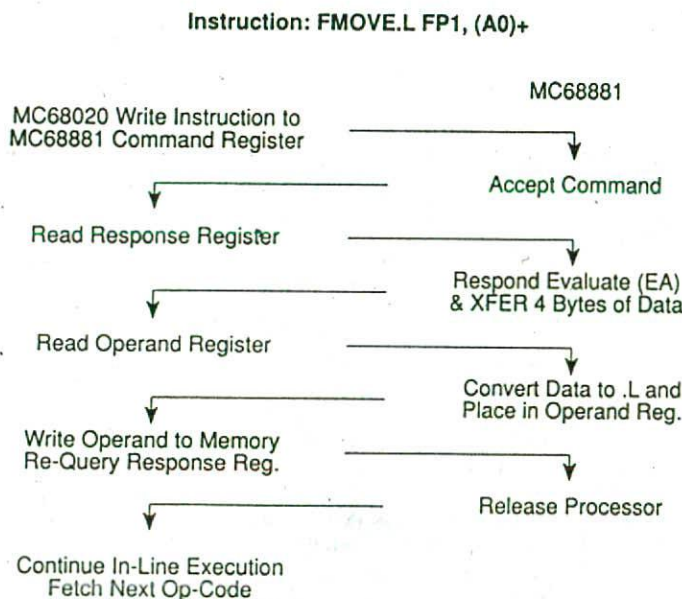


FIGURE 6.24b MC68020/MC68881 nonconcurrency example.

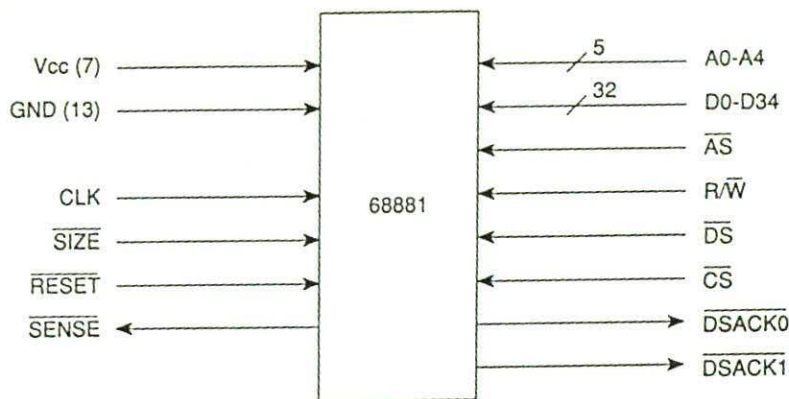


FIGURE 6.25 MC68881 pins and signals.

Figure 6.25 shows the MC68881 pins and signals.

The 68881 is included either in a 64-pin DIP or in a 68-PGA (Pin Grid Array) package. There are 7 Vcc (+5V) and 13 grounds for power distribution to reduce noise.

The five address lines A0-A4 are used by the MC68020 to select the coprocessor interface registers mapped in the MC68020 address space. These address pins select the registers as listed in Table 6.19.

When the MC68881 is configured to operate over an 8-bit data bus for processors such as MC68008, the A0 pin is used as an address signal for byte accesses of the MC68881 interface register. When the MC68881 is configured to operate over a 16-bit data bus (68000) or 32-bit data bus (68020), both A0 and SIZE pins are used, according to Table 6.20. The SIZE and A0 pins are used to configure the MC68881 for operation over 8-, 16-, or 32-bit data buses.

The address strobe \overline{AS} goes LOW to indicate that there is a valid address on the address bus, and both \overline{CS} and R/\overline{W} are valid.

A low on \overline{DS} indicates that there is valid data on the data bus during a write cycle.

If the bus cycle is a MC68020 read from MC68881, the MC68881 asserts $\overline{DSACK1}$ and $\overline{DSACK0}$ to indicate that the information on the data bus is valid. If the bus cycle is a MC68020 write to the MC68881, $\overline{DSACK1}$ and $\overline{DSACK0}$ are used to acknowledge acceptance of the data by the MC68881.

The MC68881 also uses $\overline{DSACK0}$ and $\overline{DSACK1}$ to dynamically indicate the device size on a cycle-by-cycle basis as discussed in Section 6.10.

TABLE 6.19 Coprocessor Interface Register Selection

A4-A0	Offset	Width	Type	Register
0000x	\$00	16	Read	Response
0001x	\$02	16	Write	Control
0010x	\$04	16	Read	Save
0011x	\$06	16	R / W	Restore
0100x	\$08	16	—	(reserved)
0101x	\$0A	16	Write	Command
0110x	\$0C	16	—	(reserved)
0111x	\$0E	16	Write	Condition
100xx	\$10	32	R / W	Operand
1010x	\$14	16	Read	Register select
1011x	\$16	16	—	(reserved)
110xx	\$18	32	Read	Instruction address
111xx	\$1C	32	R / W	Operand address

TABLE 6.20 Data Bus Configuration

A0	Size	Data bus
—	Low	8-bit
Low	High	16-bit
High	High	32-bit

A low on MC68881 RESET pin clears the floating-point control, status, and instruction address registers. When performing power-up reset, external circuitry should keep the RESET pin asserted for a minimum of four clock cycles after V_{cc} is within tolerance. After V_{cc} is within tolerance for more than the initial power-up time, the RESET pin must be asserted for at least two clock cycles.

The MC68881 clock input is a TTL-compatible signal that is internally buffered for generation of the internal clock signals. The clock input should be a constant frequency square wave with no stretching or shaping techniques required. The MC68881 can be operated from a 12-, 16.67-, or 20-MHz clock.

The SENSE pin may be used as an additional ground pin for more noise immunity or as an indicator to external hardware that the MC68881 is present in the system. This signal is internally connected to the GND of the die, but it is not necessary to connect it to the external ground for correct device operation. If a pullup resistor (larger than 10 K ohm) is connected to this pin, external hardware may sense the presence of the MC68881 in a system.

Figure 6.26 shows the MC68020/MC68881 interface.

The A0 and SIZE pins are connected to V_{cc} for 32-bit operation. Note that A19, A18, A17, A16 = 0010₂ (indicating coprocessor function), FC2 FC1 FC0 = 111₂ (meaning CPU space cycle), and A15 A14 A13 = 001₂ (indicating 68881 floating-point coprocessor) are used to enable 68881 CS. The 68020 A19 and A18 pins are not used in the chip select decode since their values are 00₂.

The BERR pin is asserted for a low CS and a high SENSE signal. A trap routine can be executed to perform the coprocessor operations.

Example 6.22

Write 68020 assembly language program using 68881 floating-point instructions to compute the volume of a sphere = $4/3 \cdot \pi \cdot r^3$ where r is the radius of the sphere stored in the 68020 register D0. Assume r is a single precision number.

Solution

```

FMOVE.S D0, FP0      ; MOVE r to FP0
FMOVE.X FP0, FP1      ; Make another copy of r
FSGLMUL.X FP1, FP1    ; Compute r2 by single precision
                     ; multiply
FSGLMUL.X FP0, FP1    ; Compute r3 by single precision
                     ; multiply
FMOVE.S #4E0, FP2     ; Load constant 4
FDIV.S #3E0, FP2      ; Compute 4/3
FMOVECR.X #0, FP4     ; Obtain  $\pi$ 
FSGLMUL.X FP2, FP4    ; Compute (4/3) $\pi$ 
ESGLMUL.X FP1, FP4    ; Compute volume
FMOVE.S FP4, D0       ; Store volume in D0.

```

FINISH JMP FINISH

Example 6.23

For the following figure, write an MC68020 assembly program using floating-point coprocessor instructions; determine X and Θ .

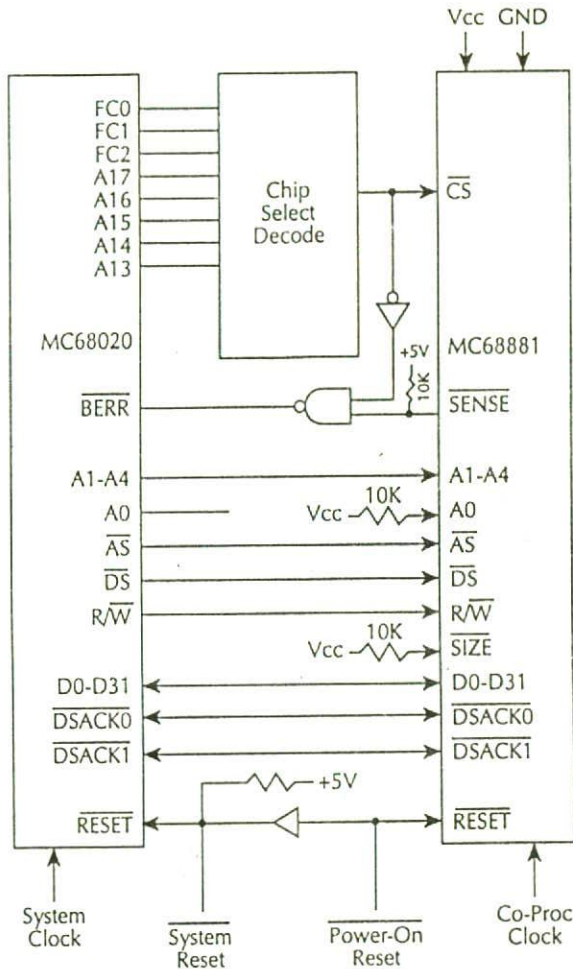
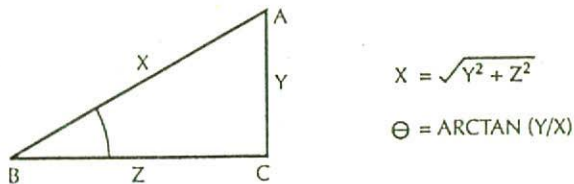


FIGURE 6.26 MC68020/MC68881 32-bit interface.



Assume Y and Z are 32 bits wide.

Solution

```

MOVE.L Y, FP2      ; MOVE Y TO FP2
MOVE.L Z, FP3      ; MOVE Z TO FP3
FMOVE.X FP2, FP0   ; MOVE Y TO FP0
FMOVE.X FP3, FP1   ; MOVE Z TO FP1
FMUL.X FP0, FP0    ; Y2
FMUL.X FP1, FP1    ; Z2
FADD.X FP0, FP1    ; Z2 + Y2
FSQRT.X FP1        ; X = √(Y2 + Z2)
FDIV.X FP3, FP2    ; Θ is

```

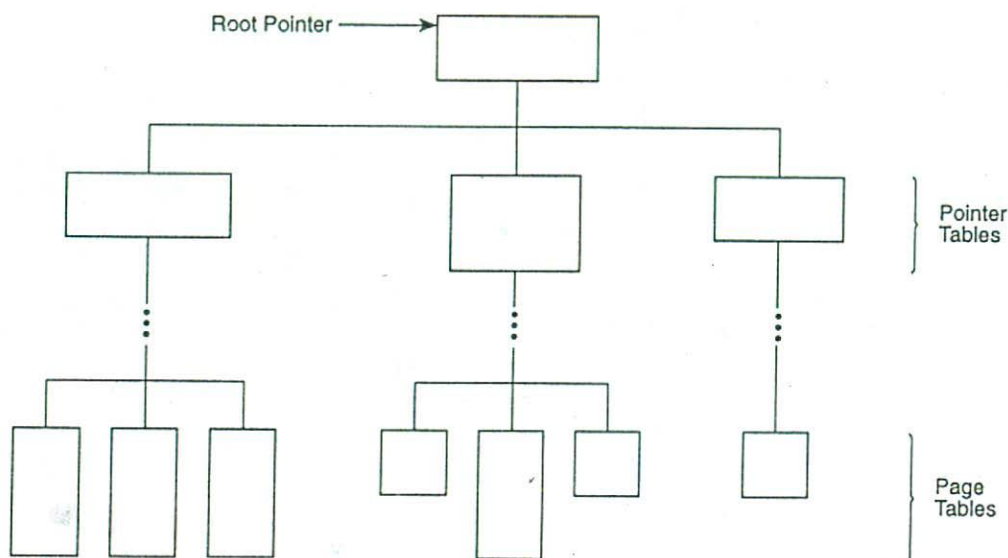



FIGURE 6.27 MC68851 translation table tree structure.

```

FATAN•X FP2      ; ARCTAN (Y/Z)
FINISH JMP FINISH

```

6.9.2 MC68851 MMU

The MC68851 coprocessor is a demand Paged Memory Management Unit (PMMU) designed to support the MC68020-based virtual memory system. The 68851 is included in a 132-PGA package and can be operated at a frequency of either 12.5 or 16.67 MHz.

The main functions of the 68851 are to provide logical-to-physical address translation, protection mechanism, and to support the 68020 breakpoint operations.

The 68851 translates a logical address comprised of a 32-bit address and a 4-bit function code. The 68851 has four function code pins (discussed later), FC0-FC3, issued by the 68020 into a corresponding 32-bit physical address in main memory. The 68851 initiates address translation by searching for the page descriptor corresponding to the logical-to-physical mapping in the on-chip 64-entry full-associative Address Translation Cache (ATC). This cache stores recently used page descriptors. If the descriptor is not found in the ATC, then the 68851 aborts the logical bus cycle, signals the 68020 to retry the operation, and requests mastership of the logical bus. Upon receiving indication that the logical bus is free, the 68851 takes over the logical bus and executes bus cycles to search the translation tables in physical (main) memory pointed to by the relevant root pointer to locate the required translation descriptor that defines the page accessed by this logical address.

After obtaining the needed translation descriptor, the 68851 returns control of the logical bus to the 68020 to retry the previous bus cycle which can now be verified for access rights and be properly translated by the 68851.

The 68851 automatically searches the translation tables in case of descriptor misses in the ATC using hardware without any assistance from the operating system. The 68851 translation tables have a tree structure, as depicted in Figure 6.27.

As shown in the figure, the root of a translation table is pointed to by one of the three 64-bit root pointer registers: CPU root pointer, supervisor root pointer, or DMA root pointer.

The CPU root pointer points at the translation table tree for the currently executing task; the supervisor root pointer points to the operating systems translation table; and the DMA root pointer points to a DMA controller's (if present in the system) translation table.

All addresses contained in the translation tables are physical addresses. In Figure 6.29, table entries at the higher levels of the tree (pointer tables) contain pointers to other tables. Entries at the leaf level (page tables) contain page descriptors. The pointer table lookup normally uses the function codes as the index, but they may be suppressed. The 68851 includes the 4-bit bidirectional function code pins, FC0-FC3, which indicate the address space of the current bus cycle. The 4-bit function code consists of the three function code outputs (FC0-FC2) of the 68020 and a fourth bit that indicates that a DMA access is in progress. When the 68851 is bus master, it drives the function code pins as outputs with a constant value of FC3-FC0 = \$5, indicating the supervisor data space.

The 68851 hierarchical protection mechanism monitors and enforces the protection/privilege mechanism. These may be up to 8 levels with privilege hierarchy, and the upper 3 bits of the incoming logical address define these levels, with level 0 as the highest privilege in the hierarchy and level 7 as the lowest level. Privilege levels of 0, 2, or 4 can also be implemented with the 68851, in which case the access level encoding is included in the upper zero, one, or two logical address lines, respectively. The 68851 access level mechanism, when enabled, compares the access level of the memory logical address with the current access level as defined in the Current Access Level (CAL) register. The current access level defines the highest privilege level that a task may assume at that time.

If the privilege level provided by the bus cycle is more privileged than allowed, then the 68851 terminates this access as a fault.

In the 68851 protection mechanism, the privilege level of a task is defined by its access level. Smaller values for access levels specify higher privilege levels. In order to access program and/or data requiring a higher privilege level than the level of the current task, the 68851 provides CALLM (call module) and RTM (return from module) instructions. These instructions allow a program to call a module operating at the same or higher privilege level and to return from that module after completing the module function.

The 68851 provides a breakpoint acknowledge facility to support the 68020. When the 68020 executes a breakpoint instruction, it executes a breakpoint acknowledge cycle and reads a predefined address in the CPU space cycle. The 68851 decodes this address and responds by either providing a replacement op code for the breakpoint op code and asserting DSACKx inputs or by asserting 68020 BERR input to execute illegal instruction exception processing. The 68851 can be programmed to provide (1) the replacement op code n times ($1 \leq n \leq 255$) in a loop and then assert BERR or (2) assert BERR on every breakpoint.

The 68851 instructions provide an extension to the 68020 instruction set via the coprocessor interface. These instructions provide:

1. Loading and storing of MMU registers
2. MMU control functions
3. Access rights and conditionals checking

For example, the PMOVE instruction moves data to or from the 68851 registers using all 68020 addressing modes. PVALID compares the access level bits of an incoming logical address with those of the Valid-Access Level (VAL) register and traps if the address bits are less. PTEST searches the ATC and translation tables for an entry corresponding to a specific address and function code. The results of the test are placed in the 68851 status register which can be tested by various conditional branch and set instructions.

Optionally, the PTEST instruction can obtain the address of the page descriptor. A companion instruction, PLOAD, takes a logical address and function code, searches the translation table, and loads the ATC with an entry to translate the logical address.

PLOAD can be used to load the ATC before starting the memory transfer. This can speed up a DMA operation. PFLUSH and its variations clear the ATC of either all entries, entries with a specified function code, or those limited to a particular function code and logical address.

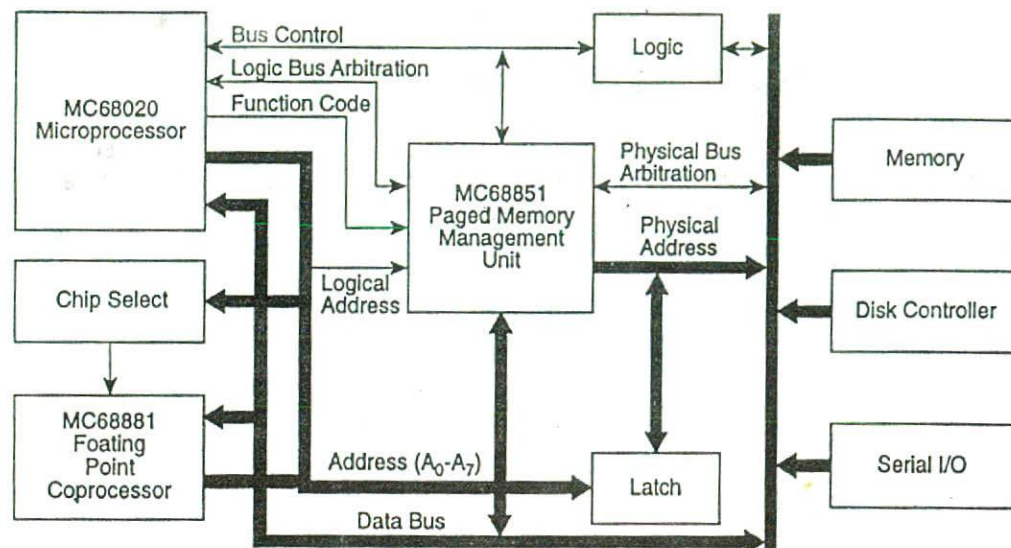


FIGURE 6.28 A 68020-based microcomputer with coprocessors.

PSAVE saves the contents of any register that reflects the current task's state and the internal state of the 68851 dealing with coprocessor and module call operations. PRESTORE restores the information saved by PSAVE. PSAVE and PRESTORE permit the context of the 68851 to be switched.

The PBcc, PDBcc, PSc, and PTRAPcc instructions have the same meaning as those of the 68020 except that the conditions are based on the 68851 condition codes.

Figure 6.28 shows a 68020-based microcomputer system which interfaces 68881 and 68851 chips to the 68020. The 68851 MMU is placed between the logical and physical address buses. The 68851 allows interfacing memory, disk controller, and serial I/O devices to the 68020.

6.10 MC68020 Pins and Signals

Figure 6.29 shows the MC68020 functional signal groups. Tables 6.21 lists these signals along with a description of each.

There are 10 VCC (+5V) and 13 ground pins to distribute the power in order to reduce noise.

Both the 32-bit address (A0-A31) and data (D0-D31) buses are nonmultiplexed. Like the MC68000, the three function code signals FC2, FC1, and FC0 identify the processor state (supervisor or user) and the address space of the bus cycle currently being executed as follows:

FC2	FC1	FC0	Cycle type
0	0	0	(Undefined, reserved)*
0	0	1	User data space
0	1	0	User program space
0	1	1	(Undefined, reserved)*
1	0	0	(Undefined, reserved)*
1	0	1	Supervisor data space
1	1	0	Supervisor program space
1	1	1	CPU space

* Address space 3 is reserved for user definition, while 0 and 4 are reserved for future use by Motorola.

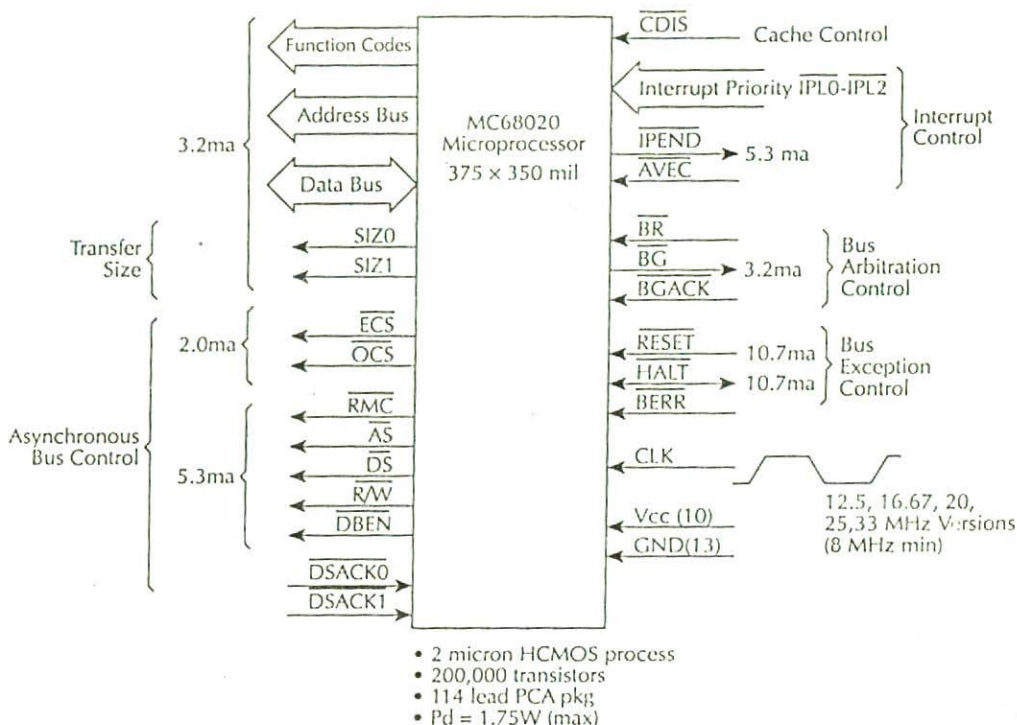


FIGURE 6.29 MC68020 functional signal groups.

TABLE 6.21 Hardware Signal Index

Signal name	Mnemonic	Function
Address bus	A0-A31	32-bit address bus used to address any of 4,294,967,296 bytes
Data bus	D0-D31	32-bit data bus used to transfer 8, 16, 24, or 32 bits of data per bus cycle
Function codes	FC0-FC2	3-bit function code used to identify the address space of each bus cycle
Size	SIZ0/SIZ1	Indicates the number of bytes remaining to be transferred for this cycle; these signals, together with A0 and A1, define the active sections of the data bus
Read-modify-write cycle	RMC	Provides an indicator that the current bus cycle is part of an indivisible read-modify-write operation
External cycle start	ECS	Provides an indication that a bus cycle is beginning
Operand cycle start	OCS	Identical operation to that of ECS except that OCS is asserted only during the first bus cycle of an operand transfer
Address strobe	AS	Indicates that a valid address is on the bus
Data strobe	DS	Indicates that valid data is to be placed on the data bus by an external device or has been placed on the data bus by the MC68020
Read/write	R / W	Defines the bus transfer as an MPU read or write
Data buffer enable	DBEN	Provides an enable signal for external data buffers
Data transfer and size acknowledge	DSACK0 / DSACK1	Bus response signals that indicate the requested data transfer operation are completed; in addition, these two lines indicate the size of the external bus port on a cycle-by-cycle basis

TABLE 6.21 Hardware Signal Index (continued)

Signal name	Mnemonic	Function
Cache disable	CDIS	Dynamically disables the on-chip cache to assist emulator support
Interrupt priority level	$\overline{\text{IPL0}}\text{--}\overline{\text{IPL2}}$	Provides an encoded interrupt level to the processor
Autovector	AVEC	Requests an autovector during an interrupt acknowledge cycle
Interrupt pending	$\overline{\text{IPEND}}$	Indicates that an interrupt is pending
Bus request	BR	Indicates that an external device requires bus mastership
Bus grant	BG	Indicates that an external device may assume bus mastership
Bus grant acknowledge	$\overline{\text{BGACK}}$	Indicates that an external device has assumed bus control
Reset	RESET	System reset
Halt	HALT	Indicates that the processor should suspend bus activity
Bus error	BERR	Indicates an invalid or illegal bus operation is being attempted
Clock	CLK	Clock input to the processor
Power supply	VCC	+5 volt \pm 5% power supply
Ground	GND	Ground connection

Note that in MC68000, FC2, FC1, FC0 = 111 indicates interrupt acknowledge cycle. In the MC68020, this means CPU space cycle. In this cycle, by decoding the address lines A19-A16, the MC68020 can perform various types of functions such as coprocessor communication, breakpoint acknowledge, interrupt acknowledge, and module operations as follows:

A19	A18	A17	A16	Function performed
0	0	0	0	Breakpoint acknowledge
0	0	0	1	Module operations
0	0	1	0	Coprocessor communication
1	1	1	1	Interrupt acknowledge

Note that A19, A18, A17, A16 = 0011₂ to 1110₂ is reserved by Motorola. In the coprocessor communication CPU space cycle, the MC68020 determines the coprocessor type by decoding A15-A13 as follows:

A15	A14	A13	Coprocessor type
0	0	0	MC68851 paged memory management unit
0	0	1	MC68881 floating-point coprocessor

The MC68020 offers a feature called dynamic bus sizing which enables designers to use 8- and 16-bit memory and I/O devices without sacrificing system performance. The key elements used to implement dynamic bus sizing are the internal data multiplexer, the SIZE output (SIZ0 and SIZ1 pins), and the DSACKX inputs (DSACK0 and DSACK1). The MC68020 uses these signals dynamically to interface to the various-sized devices (8-, 16-, or 32-bit) on a cycle-by-cycle basis. For example, if the MC68020 executes an instruction that reads a long-word operand, it will attempt to read all 32 bits during the first bus cycle. The MC68020 always assumes the memory or I/O size to be 32 bits when starting the bus cycle. Hence, it always transfers the maximum amount of data on all bus cycles. If the device responds that it is 32 bits, the processor latches all 32 bits of data and continues to the next operand. If the device responds that it is 16 bits wide, the MC68020 generates two bus cycles, obtaining 16 bits of data each time; an 8-bit transfer is handled similarly, but four bus cycles are required, obtaining 8 bits of data each time. Each device (8-, 16-, or 32-bit) assignment is fixed to particular sections of the data bus to minimize the number of bus cycles needed to transfer devices. For example,

TABLE 6.22 Dynamic Sizing Control Signals

DSACK1	DSACK Codes and Results	
	DSACK0	Result
H	H	Insert wait states in current bus cycle
H	L	Complete cycle — port size is 8 bits
L	H	Complete cycle — port size is 16 bits
L	L	Complete cycle — port size is 32 bits

SIZE Output Encodings		
SIZ1	SIZ0	Size
0	1	Byte
1	0	Word
1	1	3 bytes
0	0	Long word

Note: To adjust the size of the physical bus interface, external circuits must issue strobe signals to gate data bus buffers. By using data strobe control, external logic can enable the proper section of the data bus.

the 8-bit devices transfer data via D31-D24 pins, the 16-bit devices via D31-D16 pins, and the 32-bit devices via D31-D0 pins.

A routing and duplication multiplexer takes the four byte of 32 bits and routes them to their required positions; depending on its bus size, the positioning of bytes is determined by SIZ1, SIZ0, and A1 and A0 address output pins.

The four signals added to support dynamic bus sizing are DSACK0, DSACK1, SIZ0, and SIZ1. Data transfer and device size acknowledge signals (DSACK0 and DSACK1) are used to terminate the bus cycle and to indicate the external size of the data bus (see Table 6.22). As the MC68020 steps through memory during the data operand transfer process, the two size line outputs (SIZ0 and SIZ1) indicate how many bytes are still to be transferred during a given bus cycle (Table 6.22).

The multiplexer routes and/or duplicates one or more bytes in the 32-bit data to permit any combination of aligned or misaligned transfers. The remaining number of bytes to be transferred during the second and subsequent cycles, if required, is defined by the SIZ0 and SIZ1 outputs. The address lines A0 and A1 define the byte position in Figure 6.30. For example, A1A0 = 00₂, A1A0 = 01₂, A1A0 = 10₂, and A1A0 = 11₂ indicate byte 0 (OP0), byte 1 (OP1), byte 2 (OP2), and byte 3 (OP3), respectively, of the 32-bit operand. A2-A31 indicate the long-word base address of that portion of the operand to be accessed.

Table 6.23 defines the data pattern along with SIZ1, SIZ0, A1, and A0 of the MC68020's internal multiplexor to the external data bus D31-D0.

In each cycle, the MC68020 outputs to the SIZ1 SIZ0 pins to indicate to the external device the number of bytes remaining to be transferred. The DSACK1 and DSACK0 inputs to the MC68020 from the device terminate the bus cycle and for the subsequent cycles (if required) indicate the device size. The A1 and A0 output address pins to the device from the MC68020 indicate which data pins are to be used in the data transfer. For example, an 8-bit device always transfers data to MC68020 via D31-D24 pins for all combinations of A1A0 (00₂, 01₂, 10₂, 11₂). On the other hand, the 16-bit device always transfers data via D31-D16 pins. However, in the first cycle, if the address is even (A1A0 = 00₂ or 10₂), 16-bit data transfer takes place using D31-D16 pins with the data byte addressed by the odd address via D23-D16 pins. On the other hand, if the starting address is odd (A1A0 = 01₂ or 11₂) for a 16-bit device, only a byte is transferred in the first cycle via D23-D16 pins. For a 32-bit device, in the first cycle, if A1A0 = 00₂, all 32-bit data are transferred via D0-D31 pins; if A1A0 = 01₂, three bytes are transferred via D23-D0 pins; if A1A0 = 10₂, two bytes are transferred via D15-D0 pins; if A1A0 = 11₂, only one byte is transferred via D7-D0 pins. Note that the MC68020's

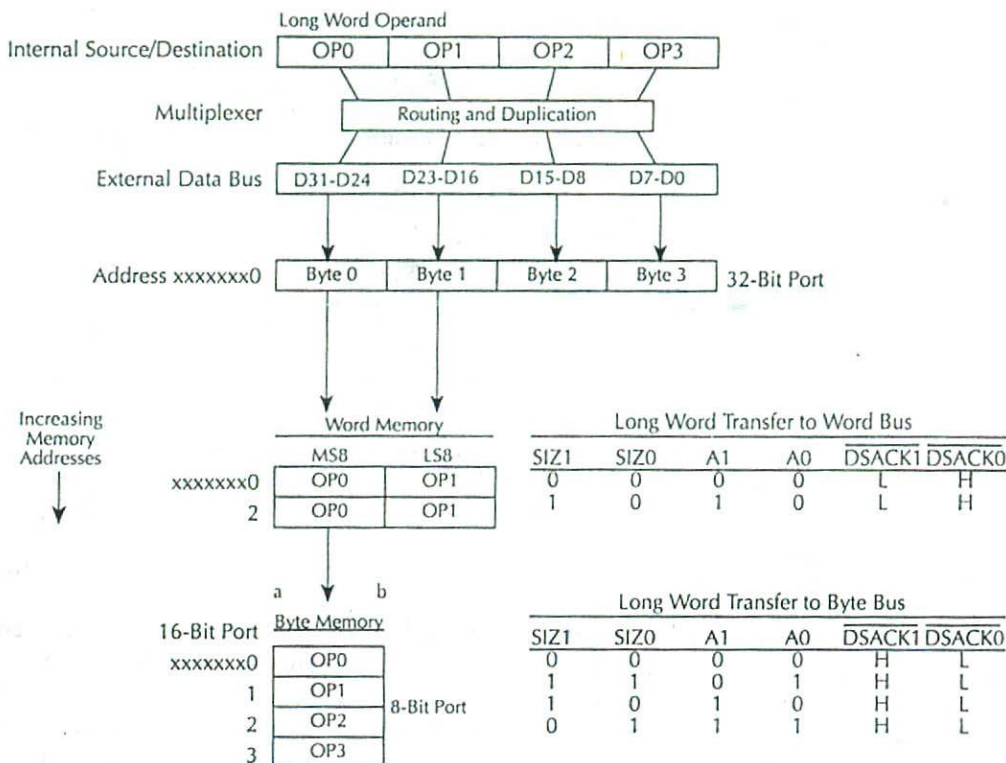


FIGURE 6.30 Dynamic bus sizing interface to port sizes.

TABLE 6.23 Internal to External Data Bus Multiplexor

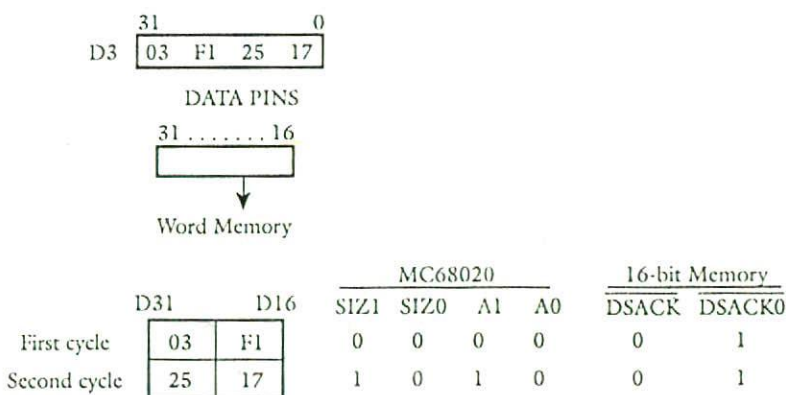
MC68020 Register				
Byte 0 Byte 0 Byte 0 Byte 0				
Multiplexor				
External Data Pins				
D31-D24 D23-D16 D15-D8 D7-D0				
Byte 3 Byte 3 Byte 3 Byte 3				
Byte 2 Byte 3 Byte 2 Byte 3				
Byte 2 Byte 2 Byte 3 Byte 2				
Byte 1 Byte 2 Byte 3 Byte 0*				
Byte 1 Byte 1 Byte 2 Byte 3				
Byte 1 Byte 2 Byte 1 Byte 2				
Byte 1 Byte 1 Byte 2* Byte 1				
Byte 0 Byte 1 Byte 2 Byte 3				
Byte 0 Byte 0 Byte 1 Byte 2				
Byte 0 Byte 1 Byte 0 Byte 1				
Byte 0 Byte 0 Byte 1* Byte 0				
Transfer size	SIZ1	SIZ0	A1	A0
Byte	0	1	X	X
Word	1	0	X	0
	1	0	X	1
3 bytes	1	1	0	0
	1	1	0	1
	1	1	1	0
	1	1	1	1
Long word	0	0	0	0
	0	0	0	1
	0	0	1	0
	0	0	1	1

Note: X = don't care; * = byte ignored on read, this byte output on write.

data transfers with even and odd starting addresses are known as aligned and misaligned transfers, respectively.

The MC68020 always starts transferring data with the most significant byte first. As an example, consider `MOVE.L D3, $50005170`. Since the address is even, this is an aligned transfer from the 32-bit data register D3 to an even memory address. In the first bus cycle, the MC68020 does not know the size of the external device and hence outputs all the data on D31-D0 pins, taking into consideration that the device size may be byte, word, or long word. The MC68020 outputs OP0, OP1, OP2, and OP3, respectively, on D31-D24, D23-D16, D15-D8, and D7-D0 pins. If the device is 8-bit, it will take the data OP0 from the D31-D24 pins and write to locations \$50005170 in the first cycle. However, by the second cycle, the device asserts DSACK1 and DSACK0 as 10_2 indicating an 8-bit device; the MC68020 then transfers the remaining 24 bits via D31-D24 in three consecutive cycles. If the device is 32-bit, it obtains data bytes OP0-OP3 in one cycle. Now, let us consider a 16-bit device. During the first cycle, the MC68020 outputs A1A0 = 00_2 indicating an aligned transfer, and SIZ1 SIZ0 = 00_2 indicating 32-bit transfer. Therefore, in the first cycle, the device obtains OP0 and OP1 from D31-D24 and D23-D16, respectively. The device then asserts DSACK1 and DSACK0 as 01_2 to terminate the cycle and to indicate to the MC68020 that it is a 16-bit device. In the second cycle, the MC68020 outputs the A1A0 as 10_2 , indicating that 16-bit data to be obtained by the device via D31-D16 pins and SIZ1 and SIZ0 as 10_2 , indicating that two more bytes remain to be transferred. The MC68020 places the low two bytes (OP2 and OP3) from register D3 via the multiplexer on D31-D16 pins. The device takes these data and places them into locations \$5000 5172 and \$5000 5173, respectively, by activating DSACK1 and DSACK0 as 01_2 , indicating completion of the cycle.

If [D3] = \$03F1 2517 (OP0 = \$03, OP1 = \$F1, OP2 = \$25, and OP3 = \$17), then data transfer for `MOVE.L D3, $5000 5170` takes place as shown in the following:



Now let us consider a misaligned transfer to a 16-bit device. For example, consider `MOVE.L D4, $6017 2421`. Assume [D4] = \$7126E214, that is, OP0 = \$71, OP1 = \$26, OP2 = \$E2, OP3 = \$14. Now, suppose that the device is 16-bit. In the first cycle, the MC68020 outputs \$717126E2 via the multiplexor; the multiplexor places these data on D31-D0 pins considering that the device may be 8-, 16-, or 32-bit as follows:

D31 : D24	D23 : D16	D15 : D8	D7 : D0
71	71	26	E2

This is because the device accepts \$71 if it is 8-bit via D31-D24 pins, \$71 via D24-D16 pins if it is 16-bit, and 24-bit data \$7126E2 via D23-D0 pins if it is 32-bit.

For 8-bit and 32-bit devices, four and two cycles are required, respectively, to complete the long-word transfer. Now, let us consider the 16-bit device for this example in detail.

In the first cycle, the MC68020 outputs SIZ_1, SIZ_0 as 00_2 , indicating a 32-bit transfer, and $A1A_0$ as 01_2 , indicating that a byte transfer is to take place via D23-D16 pins in the first cycle. The memory device obtains \$71 from D23-D16 and writes these data to \$60172421 and then activates $DSACK_1, DSACK_0$ as 01_2 , indicating to the MC68020 that it is a 16-bit device. In the second cycle, the MC68020 outputs SIZ_1, SIZ_0 as 11_2 , indicating that three more bytes remain to be transferred, and $A1A_0$ as 10_2 , indicating a 16-bit transfer is to take place via D31-D16.

The memory device activates $DSACK_1, DSACK_0$ as 01_2 to write \$26E2 to locations \$60172422 and \$60172423. The MC68020 then terminates the cycle.

In the third cycle, the MC68020 outputs SIZ_1, SIZ_0 as 01_2 , indicating a byte remaining to be transferred, and $A1A_0$ as 00_2 , indicating that the remaining byte transfer is to take place via D31-D24. The MC68020 then outputs \$14 to D31-D24 pins. The device activates $DSACK_1, DSACK_0$ as 01_2 and writes \$14 to location \$60172424. The MC68020 then terminates the cycle.

Note that the MC68020 outputs the 32-bit address via its A31-A0 pins and the device uses this address to write data to the selected memory location. A1 and A0 are used by the device to determine which data lines are to be used for data transfer. For example, the 16-bit device transfers data via D16-D23 for an odd memory address and it transfers data via D31-D24 for an even memory location. Therefore, SIZ_1, SIZ_0, A_1 , and A_0 must be used as inputs to the address decoding logic. This is discussed later.

The data transfer for the above example takes place as follows:

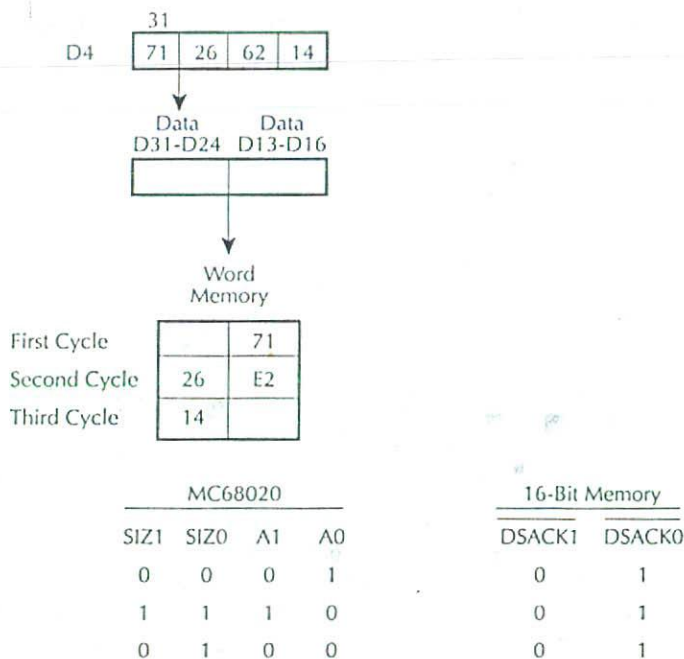


Figure 6.31 shows a functional block diagram for the MC68020 interfaces to 8-, 16-, and 32-bit memory or I/O devices.

Aligned long-word transfers to 8-, 16-, and 32-bit devices are shown in Figure 6.32.

MC68020 byte addressing is summarized in Figure 6.33.

Figure 6.34 shows misaligned long-word transfers to 8-, 16-, and 32-bit devices.

Now, let us explain the other MC68020 pins.

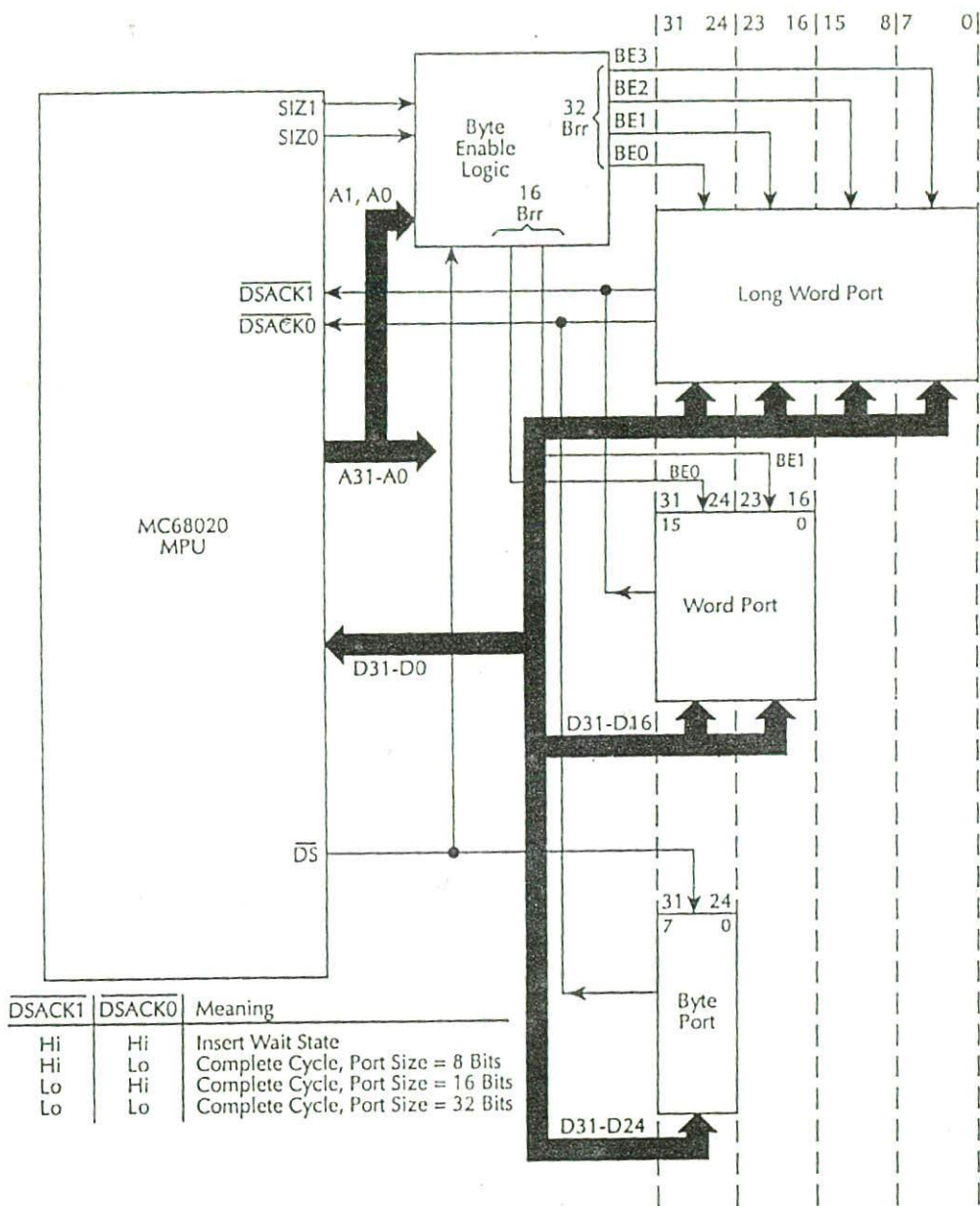
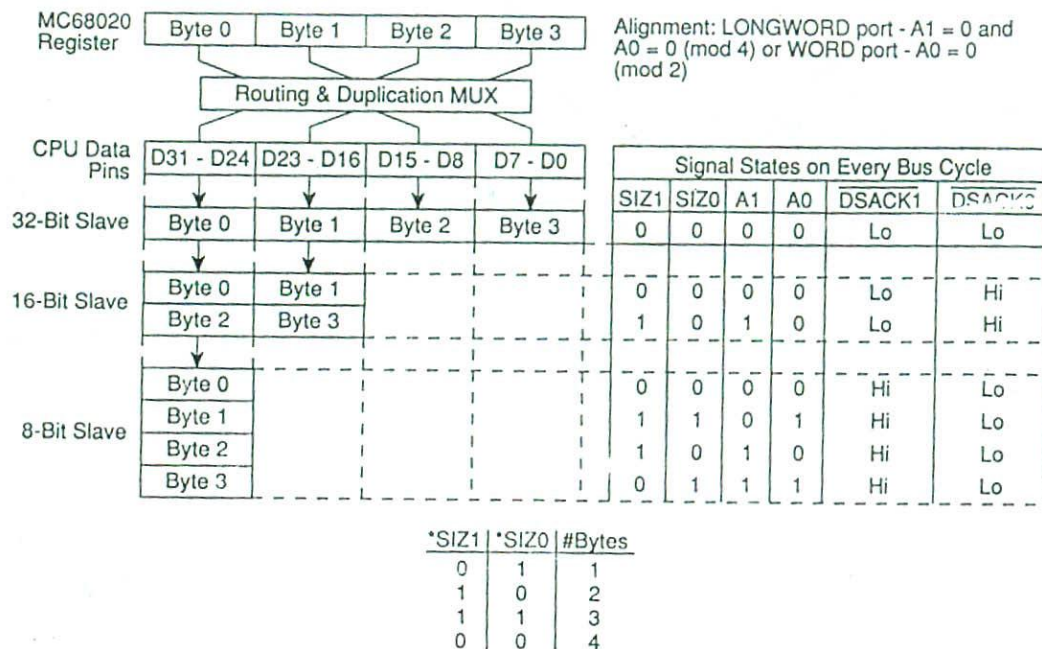


FIGURE 6.31 MC68020 dynamic bus sizing block diagram.

The ECS (external cycle start) pin is a MC68020 output pin. The MC68020 asserts this pin during the first one half clock of every bus cycle to provide the earliest indication of the start of a bus cycle. The use of ECS must be validated later with AS, since the MC68020 may start an instruction fetch cycle and then abort if the instruction is found in the cache. In the case of a cache hit, the MC68020 does not assert AS, but provides A31-A0, SIZ1, SIZ0, and FC2-FC0 outputs.



* Size pins indicates number of bytes remaining to complete the operand transfer.

FIGURE 6.32 Aligned long-word transfer.

The MC68020 asserts the OCS (operand cycle start) pin only during the first bus cycle of an operand transfer or instruction prefetch.

The MC68020 asserts the RMC (read-modify-write) pin to indicate that the current bus operation is an indivisible read-modify-write cycle. RMC should be used as a bus lock to ensure integrity of instructions which use read-modify-write operations such as TAS and CAS.

In a read cycle, the MC68020 asserts the DS (data strobe) pin to indicate that the slave device should drive the bus. During a write cycle, it indicates that the MC68020 has placed valid data on the data bus.

DEN (data buffer enable) is output by the MC68020 which may be used to enable external data buffers.

The CDIS (cache disable) input pin to the MC68020 dynamically disables the cache when asserted.

The interrupt pending (IPEND) input pin indicates that the value of the inverted IPL2 - IPL0 pins is higher than the current I2I1I0 in SR or that a nonmaskable interrupt has been recognized.

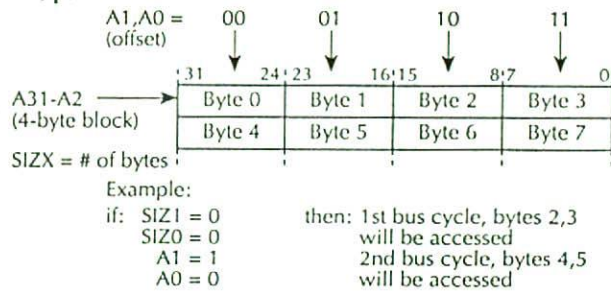
The MC68020 AVEC input is activated by an external device to service an autovector interrupt. The AVEC has the same function as the MC68000 VPA.

The functions of the other signals such as AS, R/W, IPL2 - IPL0, BR, BG, and BGACK are similar to those of the MC68000.

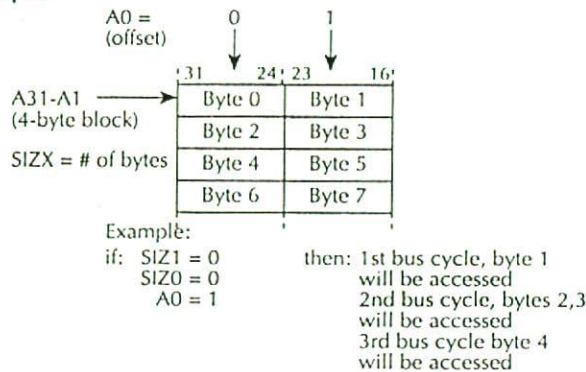
The MC68020 system control pins are functionally similar to those of the MC68000. However, there are some minor differences. For example, for hardware reset, RESET and HALT pins need not be asserted simultaneously. Therefore, unlike the MC68000, RESET and HALT pins are not tied together in the MC68020 system.

RESET and HALT pins are bi-directional, open-drain (external pull-up resistances are required), and their functions are independent.

Longword (32-Bit) port



Word (16-Bit) port



Byte (8-Bit) Port

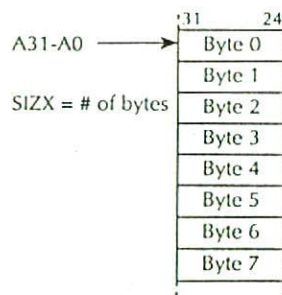


FIGURE 6.33 MC68020 byte addressing.

When HALT input is asserted by an external device, the following activities take place:

- All control signals become inactive.
- Address lines, R/W line, and function code lines remain driven with last bus cycle information.
- All bus activities stop after current bus cycle completion.

Assertion of HALT stops only external bus activities and the processor execution continues. That is, the MC68020 can continue with instruction execution internally if cache hits occur and if the external bus is not required.

The MC68020 asserts the HALT output for double bus fault. The BERR input pin, when asserted by an external device, causes the bus cycle to be aborted and strobes negated. If the BERR is asserted during operand read or write (not prefetch), exception processing occurs immediately. The BERR pin can typically be used to indicate a nonresponding device (no DSACKX received from the device), vector acquisition failure, illegal access determined by

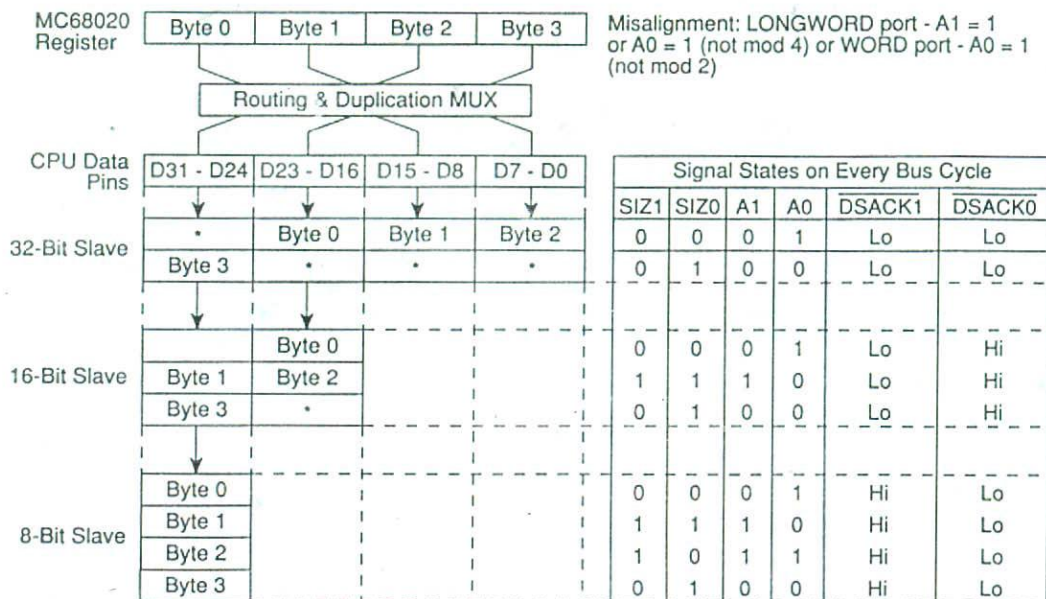


FIGURE 6.34 Misaligned long-word transfer.

memory management unit hardware such as access fault (protected memory scheme), and page fault (virtual memory system).

Figure 6.35 shows the MC68020 reset characteristics.

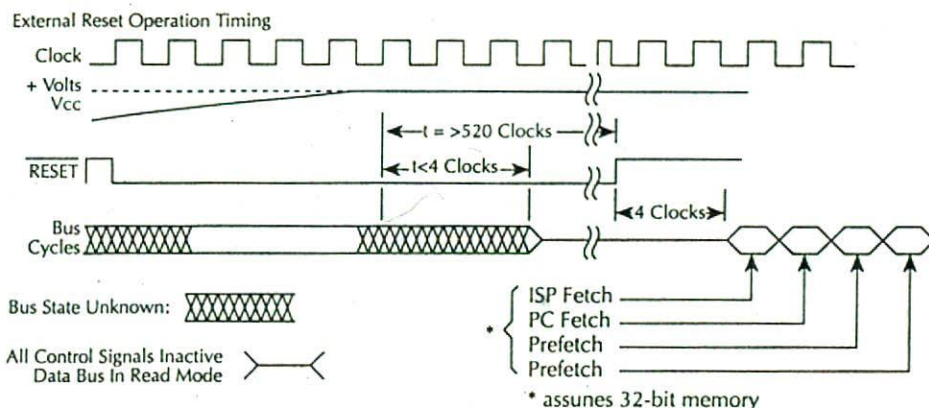
The RESET signal is a bi-directional signal. The RESET pin, when asserted by an external circuit for a minimum of 520 clock periods, resets the entire system including the MC68020. Upon hardware reset, the MC68020 completes any active bus cycle in an orderly manner and then performs the following:

- Reads the 32-bit contents of address \$00000000 and loads it into the ISP (contents of \$00000000 are loaded to the most significant byte of the ISP and so on)
- Reads the 32-bit contents of address \$00000004 into the PC (contents of \$00000004 to most significant byte of the PC and so on)
- Sets I2I1I0 bits of the SR to 111, sets S-bit in the SR to 1, and clears T1, T0, M bits in the SR
- Clears the VBR to \$00000000
- Clears the cache enable bit in the CACR
- All other registers are unaffected by hardware reset

When the RESET instruction is executed, the MC68020 asserts the RESET pin for 512 clock cycles and the processor resets all the external devices connected to the RESET pin. Software reset does not affect any internal register.

Figure 6.36 shows a MC68020 reset circuit.

The Motorola MC3456 contains a dual timing circuit. The MC3456 uses an external resistor-capacitor network as its timing elements. Like the MC1455 timer used in the 68000 reset circuit, the MC3456 includes comparators and an R-S flip-flop. From the MC3456 data sheet, the RC values connected at the TRG input of the MC3456 will make output OP HIGH for $T = 1.1 R_A C_A$ seconds, where R_A = resistor connected at the TSH = 1 Mohm, and C_A =



Reset Flow Diagram

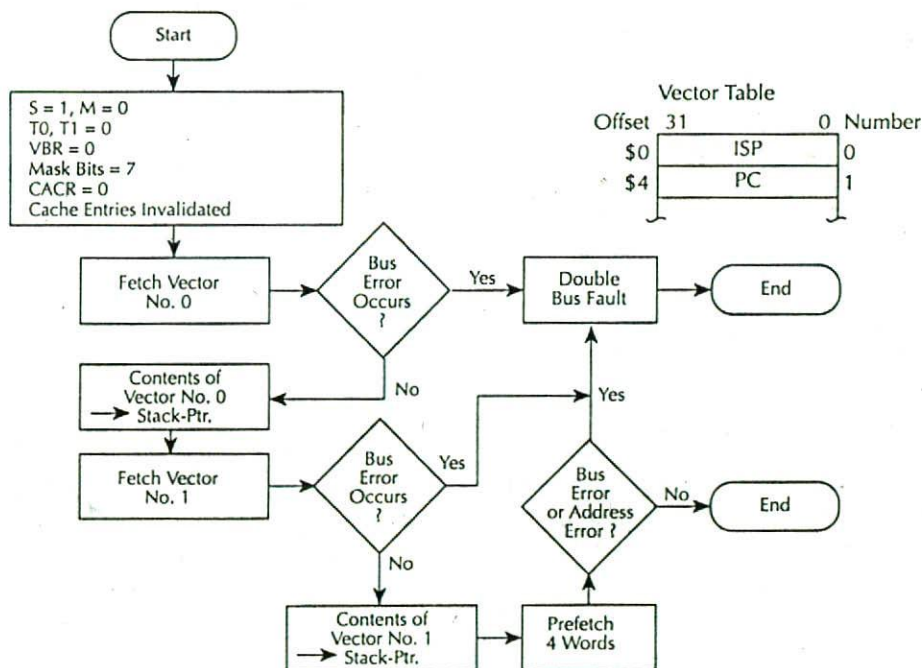


FIGURE 6.35 MC68020 reset characteristics (higher level exception).

capacitor connected at $\text{DIS} = 0.47 \mu\text{F}$. This means that OP will be HIGH for 517 msec ($1.1 \text{ M} * 0.47 \mu\text{F}$); the OP will then go back to LOW state. Therefore, the output of the inverter (connected at OP) will be LOW for 517 ms.

The push button connected to the input of the debouncing circuit, when not activated, will generate HIGH at the bottom input of AND gate #2 and LOW at the top of input of AND gate #1.

Since a NAND gate generates a HIGH with one of the inputs as LOW, the output of NAND gate #1 will be HIGH. This means that both inputs of NAND gate #2 will be HIGH. Therefore, the output of NAND gate #2 will be LOW. This LOW level is inverted and connected to a WIRED.OR circuit along with the inverted OP of the MC3456 at the MC68020 RESET pin.

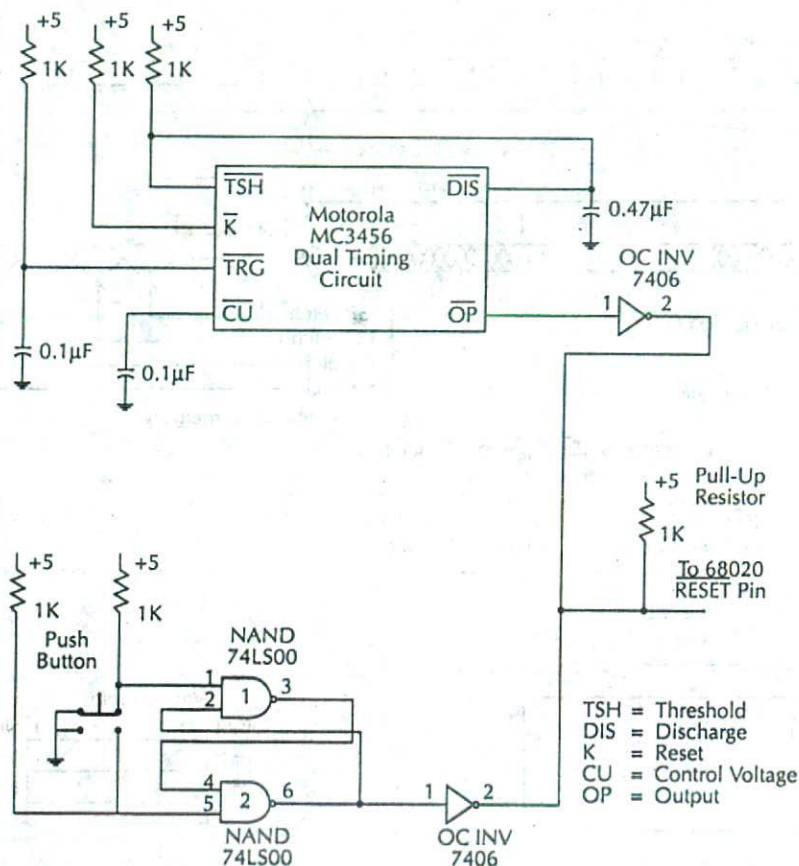


FIGURE 6.36 MC68020 reset circuit.

The output of the debouncing circuit will be HIGH when the push button is activated. For example, activation of the push button will generate a HIGH at the top input of NAND gate #1 and a LOW at the bottom input of NAND gate #2. The AND gate #2 will generate a HIGH at its output. Therefore, the inverted output will be LOW which is presented at the WIRED.-OR circuit of the 68020 RESET pin. A LOW will be provided at the 68020 RESET pin only when both inputs are LOW, that is, when the push button is activated and inverted OP of MC3456 is LOW.

The MC3456 timer will keep this RESET pin signal LOW for 517 msec. As mentioned before, the 68020 requires the RESET pin to stay LOW for at least 520 clock cycles. For a 60-ns (16-MHz) clock, the 68020 must then be LOW for at least $31.2 \mu\text{s}$ ($520 * 60 \text{ ns}$). Since the reset circuit of Figure 6.39 outputs a LOW for 517 msec ($>31.2 \mu\text{s}$) upon activation of the push button, the 68020 RESET pin will be asserted properly.

Example 6.24

Determine the number of bus cycles, the bytes written to memory (in Hex), and signal levels of A1, A0, SIZ1, and SIZ0 pins that would occur when the instruction `MOVE.L D1, (A0)` with $[D1] = \$5012\ 6124$ and $[A0] = \$2000\ 2053$ is executed by the MC68020.

- Assume
1. 32-bit memory
 2. 16-bit memory
 3. 8-bit memory

Indicate the bus cycles in which $\overline{\text{OCS}}$ is asserted.

Solution

1. 32-bit memory; misaligned transfer since starting address is odd

					MC68020 D31-D0 pins			
					D31-D24	D21-D16	D15-D8	D7-D0
First bus cycle	A1	A0	SIZ1	SIZ0	1	1	0	0
Second bus cycle	0	0	1	1	12	61	24	

$\overline{\text{OCS}}$ is asserted in the first bus cycle.

2. 16-bit memory

					MC68020 D31-D16 pins	
					D31-D24	D23-D16
First bus cycle	A1	A0	SIZ1	SIZ0	1	1
Second bus cycle	0	0	1	1	12	61
Third bus cycle	1	0	0	1	24	

$\overline{\text{OCS}}$ is asserted in the first bus cycle.

3. 8-bit memory

					MC68020	
					D31-D24 PINS	
First bus cycle	A1	A0	SIZ1	SIZ0	1	1
Second bus cycle	0	0	1	1	12	61
Third bus cycle	0	1	1	0	24	
Fourth bus cycle	1	0	0	1		

$\overline{\text{OCS}}$ is asserted in the first bus cycle.

Example 6.25

Determine the contents of the PC, SR, MSP, and the ISP after a MC68020 hardware reset. Assume a 32-bit memory with the following data prior to the reset:

MEMORY	
\$00000000	\$50001234
\$00000004	\$72152614

REGISTERS	
MSP	\$27140124
ISP	\$61711420
PC	\$35261271
SR	\$0301

Solution

After hardware reset, the following are the memory and register contents:

MEMORY	
\$00000000	\$50001234
\$00000004	\$72152614

REGISTERS	
MSP	\$27140124
ISP	\$50001234
PC	\$72152614
SR	\$2701

Note that $[SR] = \$2701 =$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	1	1	0	0	0	0	0	0	0	1
T1 T0 S M				I2 I1 I0				X N Z V C							

Therefore, $T1T0 = 00$, $S = 1$, $M = 0$, and $I2I1I0 = 111$. Other bits are unaffected.

6.11 MC68020 Timing Diagrams

The MC68020 always activates all data lines. The MC68020 can perform either synchronous or asynchronous operation. Synchronous operation permits interfacing the devices which use the MC68020 clock to generate DSACKX and other asynchronous inputs. The asynchronous input setup and hold times must be satisfied for the assertion or negation of these inputs. The MC68020 then guarantees recognition of these signals on the current falling edge of the clock.

On the other hand, asynchronous operation provides clock frequency independence for generating DSACKX and other asynchronous inputs. This operation requires utilization of only the bus handshake signals (AS, DS, DSACKX, BERR, and HALT). In asynchronous operation, AS indicates the beginning of a bus cycle and DS validates data in a write cycle.

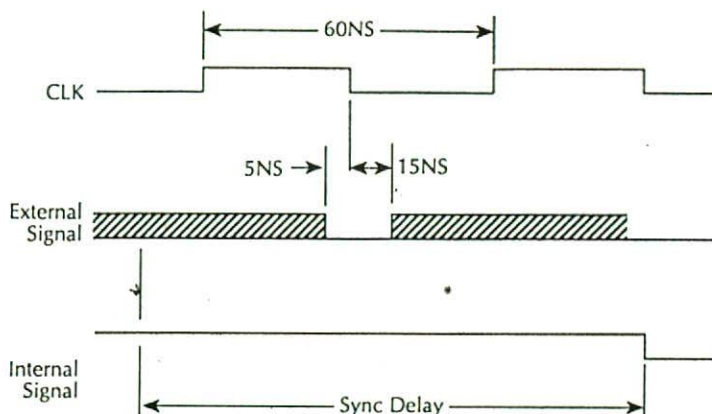
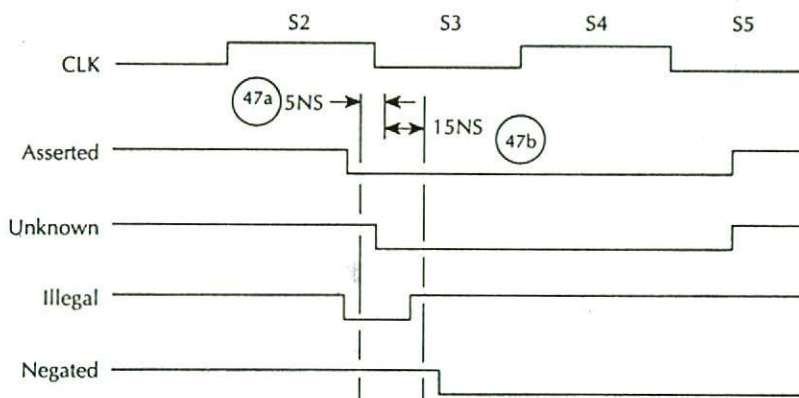


FIGURE 6.37a Synchronizing asynchronous inputs.



In general, signal level must be maintained from S2 into S5 or results will be unpredictable. An exception to this rule is bus error.

FIGURE 6.37b Asynchronous input recognition.

The SIZ1, SIZ0, A1, and A0 signals are decoded to generate strobe signals. These strobes indicate which data bytes are to be used in the transfer. The memory or I/O devices then place data on the right portion of the data bus for a read cycle or latch data in a write cycle. The selected device finally activates the DSACKX lines according to the device size to terminate the cycle. If no DSACKX is received by the MC68020, or the access is invalid, the external device can assert BERR to abort or BERR and HALT to retry the bus cycle. There is no limit on the time from assertion of AS to the assertion of DSACKX, since the MC68020 keeps inserting wait states in increments of one cycle until DSACKX is recognized by the processor.

For synchronization, the MC68020 uses a time delay to sample an external asynchronous input for high or low and then synchronizes this input to the clock.

Figures 6.37a and b show an example of synchronization and recognition of asynchronous inputs.

Note that for all inputs, there is a sample window of 20 ns during which the MC68020 latches the input level. In order to guarantee the recognition of a certain level on a particular falling edge of the clock, the input level must be held stable throughout this sample window of 20 ns. If an input changes during the sample window, the level recognized by the MC68020 is unknown or illegal. One exception to this rule is the delayed assertion of BERR where the signal must be stable through the window or the MC68020 may exhibit erratic behavior.

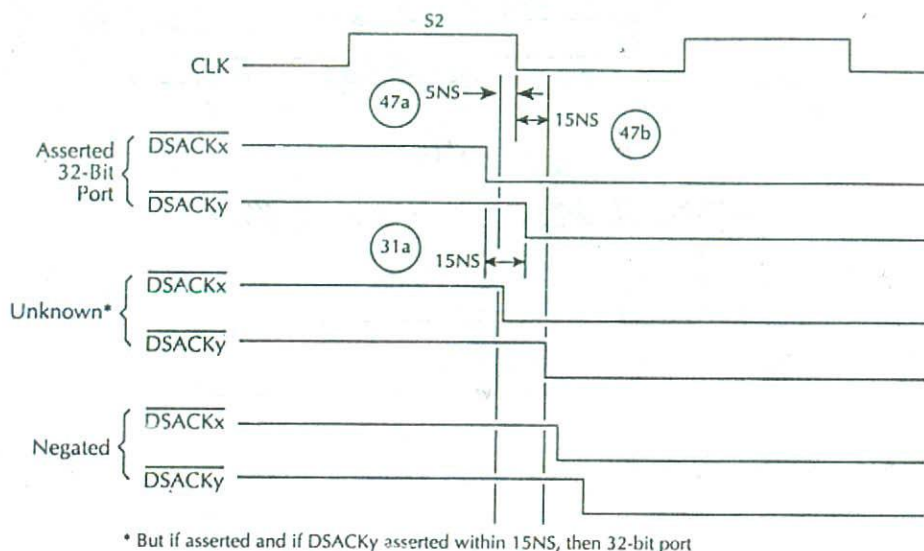
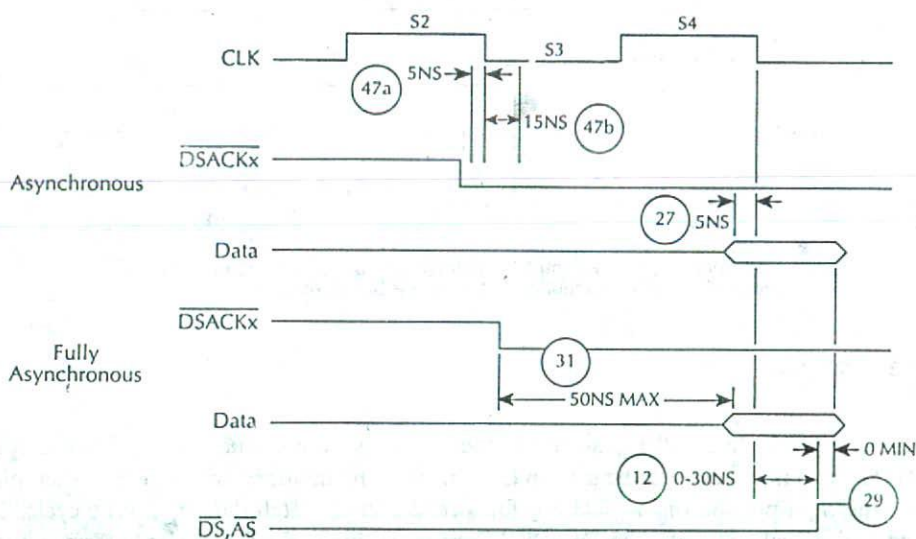
FIGURE 6.38a \overline{DSACKX} input recognition.

FIGURE 6.38b Protocol for reading data.

Note that if the \overline{BERR} is asserted during an instruction prefetch, the MC68020 delays bus error exception processing until the faulted data are required for execution. Bus error processing will take place for faulty access if change in program flow such as branching occurs, since the faulty data are not required. Also, after satisfying the setup and hold times, all input signals must meet certain protocols. For example, when \overline{DSACKx} is asserted it must remain asserted until \overline{AS} is negated. Figures 6.38a and b show timing of \overline{DSACKX} input recognition and the MC68020's reading of data satisfying the required protocol.

In the timing diagrams of Motorola's 68020 manuals, parameter #47 (47a and 47b) provides the asynchronous input setup time of 20 ns. All numbers circled in the timing diagrams are the timing parameters provided in Motorola manuals.

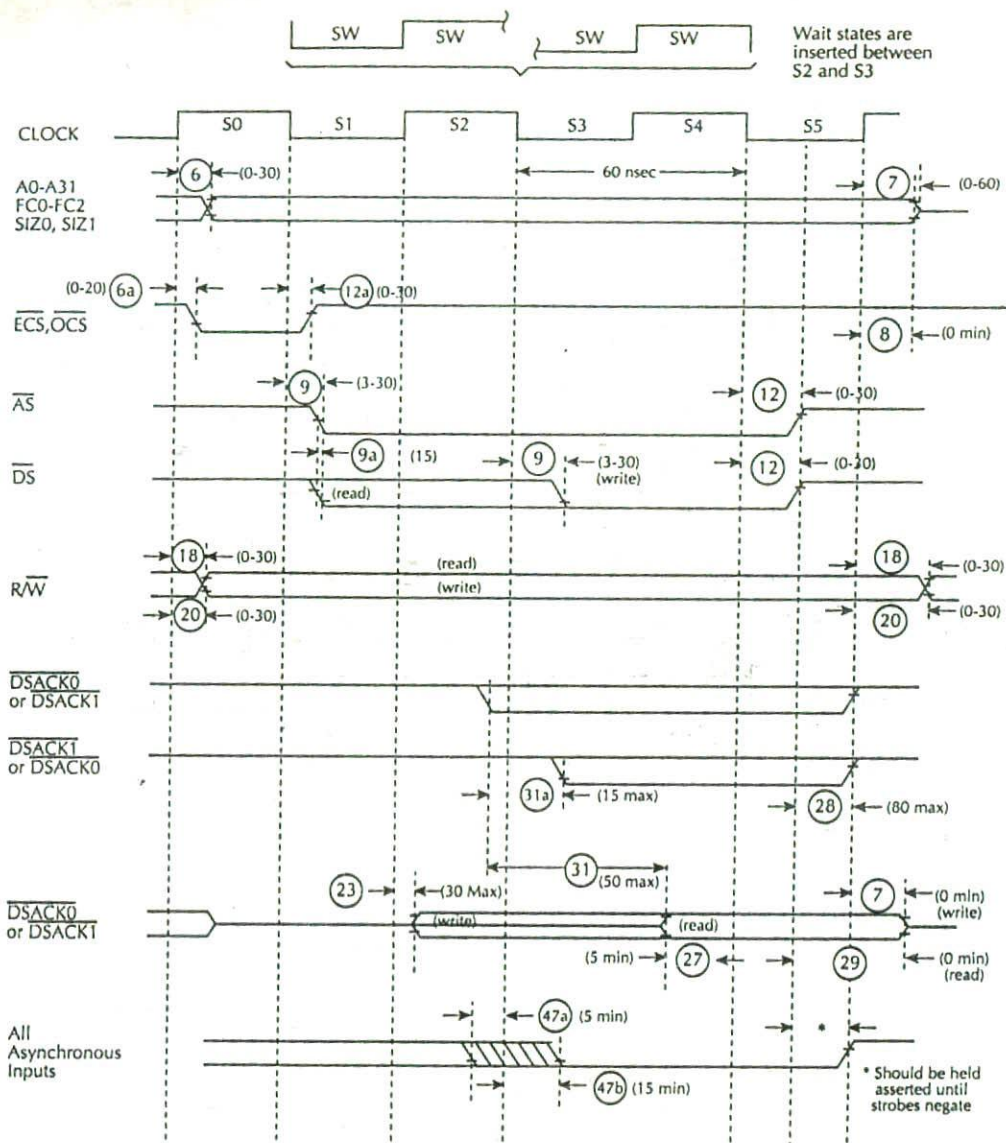


FIGURE 6.39a Asynchronous bus cycle timing. All time is in nanoseconds.

In Figure 6.38b (synchronous operation), assertion of $\overline{\text{DSACKX}}$ is recognized on the falling edge of S2; the MC68020 latches valid data on the falling edge of S4. For asynchronous operation, data are latched 50 ns (parameter 31) after assertion of $\overline{\text{DSACKX}}$. If $\overline{\text{DSACKX}}$ or $\overline{\text{BERR}}$ is not asserted by the external device during the 20 ns window of the falling edge of S2, the 68020 inserts wait states until one of these input signals is asserted. A minimum of three clock cycles is required for a read operation. $\overline{\text{DSACKX}}$ remains asserted until $\overline{\text{AS}}$ negation is satisfied in Figure 6.39a.

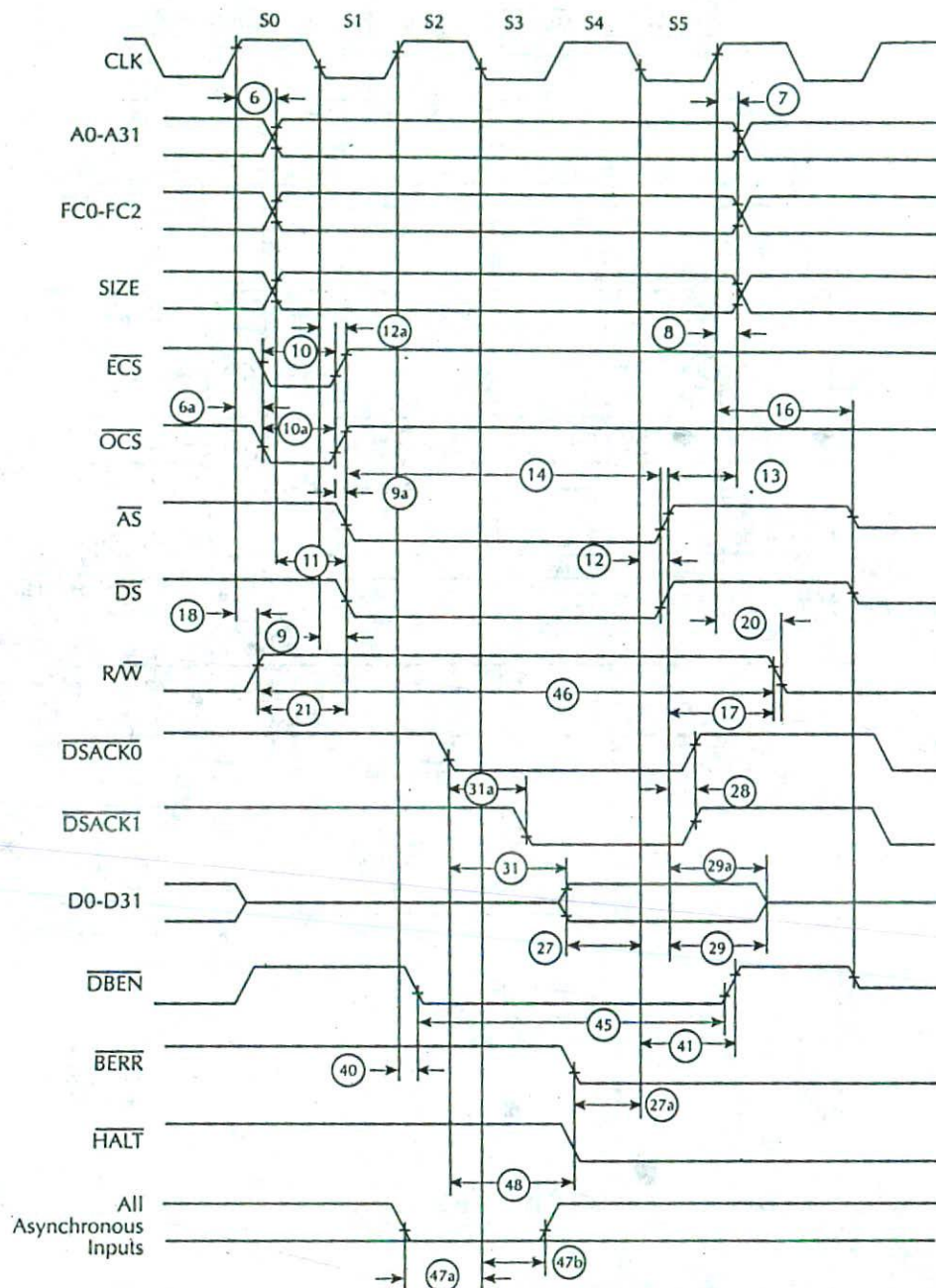


FIGURE 6.39b Read cycle timing diagram. Timing measurements are referenced to and from a low voltage of 0.8 volts and high voltage of 2.0 volts, unless otherwise noted. The voltage swing through this range should start outside and pass through the range such that the rise or fall will be linear between 0.8 and 2.0 volts.

Figure 6.39a shows asynchronous bus cycle timing along with various parameters. Figures 6.39b, c, and d show typical MC68020 read and write timing diagrams (general form) along with their AC specifications. Note that in Figures 6.39b and c signals such as SIZ_0 , SIZ_1 , $DSACK_X$, D_0 - D_{31} , A_1 , and A_0 , which precisely distinguish data transfers between 8-, 16-, and 32-bit devices, are kept in general form.

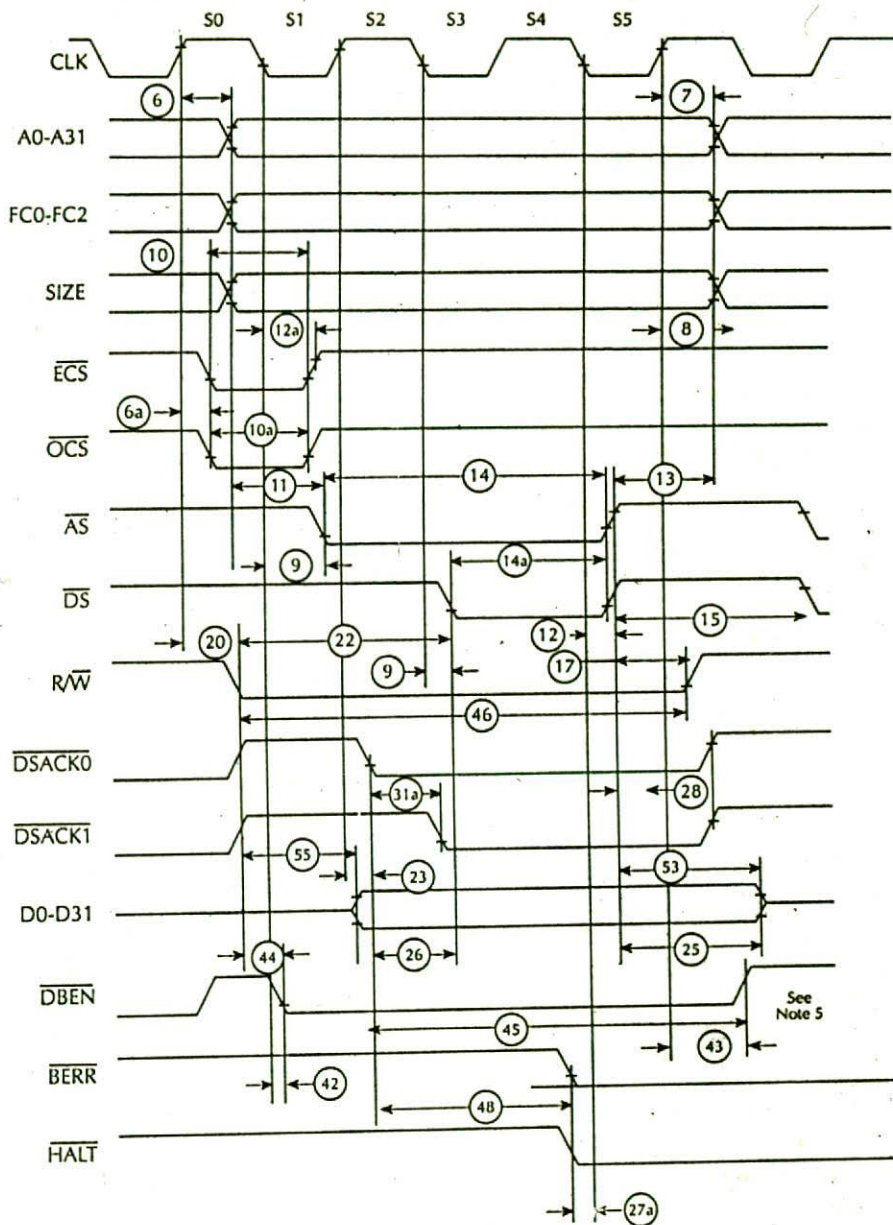


FIGURE 6.39c Write cycle timing diagram. Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted. The voltage swing through this range should start outside and pass through the range such that the rise or fall will be linear between 0.8 and 2.0 volts. "Note 5" refers to Figure 6.39d.

6.12 Exception Processing

The MC68020 exceptions are functionally similar to those of the MC68000 with some minor variations. The MC68020 exceptions can be generated by external or internal causes. Externally generated exceptions include interrupts, bus errors, reset, and coprocessor-detected

Num.	Characteristic	12.5 MHz		16.67 MHz		20 MHz		25 MHz		Unit
		Min	Max	Min	Max	Min	Max	Min	Max	
6	Clock High to Address/FC/Size/RMC Valid	0	40	0	30	0	25	0	25	ns
6A	Clock High to ECS, OCS asserted	0	30	0	20	0	15	0	15	ns
7	Clock High to Address/Data/FC/RMC/Size High Impedance	0	80	0	60	0	50	0	40	ns
8	Clock High to Address/FC/Size/RMC Invalid	0	—	0	—	0	—	0	—	ns
9	Clock Low to AS, DS Asserted	3	40	3	30	3	25	3	20	ns
9A ¹	AS to DS Assertion (Read) (Skew)	-20	20	-15	15	-10	10	-10	10	ns
10	ECS Width Asserted	25	—	20	—	15	—	10	—	ns
10A	OCS Width Asserted	25	—	20	—	15	—	10	—	ns
10B ⁷	ECS, OCS Width Negated	20	—	15	—	10	—	5	—	ns
11 ⁶	Address/FC/Size/RMC Valid to AS Asserted (and DS Asserted, Read)	20	—	15	—	10	—	5	—	ns
12	Clock Low to AS, DS Negated	0	40	0	30	0	25	0	20	ns
12A	Clock Low to ECS, OCS Negated	0	40	0	30	0	25	0	20	ns
13	AS, DS Negated to Address/FC/Size/RMC Invalid	20	—	15	—	10	—	5	—	ns
14	AS (and DS Read) Width Asserted	120	—	100	—	85	—	65	—	ns
14A	DS Width Asserted, Write	50	—	40	—	38	—	30	—	ns
15	AS, DS Width Negated	50	—	40	—	38	—	30	—	ns
15A ⁸	DS Negated to AS Asserted	45	—	35	—	30	—	25	—	ns
16	Clock High to AS/DS/R/W/DBEN High Impedance	—	80	—	60	—	50	—	40	ns
17 ⁶	AS, DS Negated to R/W High	20	—	15	—	10	—	5	—	ns
18	Clock High to R/W High	0	40	0	30	0	25	0	20	ns
20	Clock High to R/W Low	0	40	0	30	0	25	0	20	ns
21 ⁶	R/W High to AS Asserted	20	—	—	—	—	—	—	—	ns
22 ⁶	R/W Low to DS Asserted (Write)	90	—	—	—	—	—	—	—	ns
23	Clock High to Data Out Valid	—	—	—	—	—	—	—	—	ns
25 ⁶	AS, DS Negated to Data Out Invalid	20	—	15	—	10	—	5	—	ns
25A ⁹	DS Negated to DBEN Negated (Write)	20	—	15	—	10	—	5	—	ns
26 ⁶	Data out Valid to DS Asserted (Write)	20	—	15	—	10	—	5	—	ns
27	Data-In Invalid to Clock Low (Data Setup)	10	—	5	—	5	—	5	—	ns
27A	Late BERR/HALT Asserted to Clock Low Setup Time	25	—	20	—	15	—	10	—	ns
28	AS, DS Negated to DSACKx/BERR/HALT/AVEC Negated	0	110	0	80	0	65	0	50	ns
29	DS Negated to Data-In Invalid (Data-In Hold Time)	0	—	0	—	0	—	0	—	ns
29A	DS Negated to Data-In (High Impedance)	—	80	—	60	—	50	—	40	ns
31 ²	DSACKx Asserted to Data-In Invalid	—	60	—	50	—	43	—	32	ns
31A ³	DSACKx Asserted to DSACKx Valid (DSACK Asserted Skew)	—	20	—	15	—	10	—	10	ns

FIGURE 6.39d Read and write cycle specifications.

errors. Internally generated exceptions are caused by certain instructions, address errors, tracing, and breakpoints. Instructions that may cause internal exceptions as part of their instruction execution are CHK, CHK2, CALLM, RTM, RTE, DIV, and all variations of the TRAP instruction. In addition, illegal instructions, privilege violations, and coprocessor violations cause exceptions. Table 6.24 lists the priority and characteristics of all MC68020 exceptions.

Num.	Characteristic	12.5 MHz		16.67 MHz		20 MHz		25 MHz		Unit
		Min	Max	Min	Max	Min	Max	Min	Max	
32	RESET Input Transition Time	—	1.5	—	1.5	—	1.5	—	1.5	Chks
33	Clock Low to \overline{BG} Asserted	0	40	0	30	0	25	0	20	ns
34	Clock Low to \overline{BG} Negated	0	40	0	30	0	25	0	20	ns
35	\overline{BR} Asserted to \overline{BG} Asserted (\overline{RMC} Not Asserted)	1.5	3.5	1.5	3.5	1.5	3.5	1.5	3.5	Chks
37	\overline{BGACK} Asserted to \overline{BG} Negated	1.5	3.5	1.5	3.5	1.5	3.5	1.5	3.5	Chks
37A	\overline{BGACK} Asserted to \overline{BR} Negated	0	1.5	0	1.5	0	1.5	0	1.5	Chks
39	\overline{BG} Width Negated	120	—	90	—	75	—	60	—	ns
39A	\overline{BG} Width Asserted	120	—	90	—	75	—	60	—	ns
40	Clock High to \overline{DBEN} Asserted (Read)	0	40	0	30	0	25	0	20	ns
41	Clock High to \overline{DBEN} Negated (Read)	0	40	0	30	0	25	0	20	ns
42	Clock Low to \overline{DBEN} Asserted (Write)	0	40	0	30	0	25	0	20	ns
43	Clock Low to \overline{DBEN} Negated (Write)	0	40	0	30	0	25	0	20	ns
44 ⁶	R/W Low to \overline{DBEN} Asserted (Write)	20	—	15	—	10	—	5	—	ns
45 ⁶	\overline{DBEN} Width Asserted	Read	80	60	50	40	—	—	—	ns
		Write	160	120	100	80	—	—	—	ns
46	R/W Width Asserted (Write and Read)	—	—	—	—	—	—	—	—	ns
47A	Asynchronous Input Setup Time	180	—	150	—	125	—	100	—	ns
47B	Asynchronous Input Hold Time	10	—	5	—	5	—	5	—	ns
48 ⁴	\overline{DSACKs} Asserted to $\overline{BERR}/\overline{HALT}$ Asserted	—	35	—	30	—	25	—	20	ns
53	Data Out Hold from Clock High	0	—	0	—	0	—	0	—	ns
55	R/W Asserted to Data Bus Impedance Changes	40	—	30	—	25	—	20	—	ns
56	RESET Pulse Width (Reset Instruction)	512	—	512	—	512	—	512	—	Chks
57	\overline{BERR} Negated to \overline{HALT} Negated (Rerun)	0	—	0	—	0	—	0	—	ns
58 ¹⁰	\overline{BGACK} Negated to Bus Driven	1	—	1	—	1	—	1	—	Chks
59 ¹⁰	\overline{BG} Negated to Bus Driven	1	—	1	—	1	—	1	—	Chks

Notes:

1. This number can be reduced to 5 nanoseconds if strobes have equal loads.
2. If the asynchronous setup time (#47) requirements are satisfied, the \overline{DSACKx} low to data setup time (#31) and \overline{DSACKx} low to \overline{BERR} low setup time (#48) can be ignored. The data must only satisfy the data-in to clock low setup time (#27) for the following clock cycle. \overline{BERR} must only satisfy the late \overline{BERR} low to clock low setup time (#27A) for the following clock cycle.
3. This parameter specifies the maximum allowable skew between $\overline{DSACK0}$ to $\overline{DSACK1}$ asserted or $\overline{DSACK1}$ to $\overline{DSACK0}$ asserted, specification #47 must be met by $\overline{DSACK0}$ to $\overline{DSACK1}$.
4. In absence of \overline{DSACKx} , \overline{BERR} is an asynchronous input using the asynchronous input setup time (#47).
5. \overline{DBEN} may stay asserted on consecutive write cycles.
6. Actual value depends on the clock input waveform.
7. This is a new specification that indicates the minimum high time for \overline{ECS} and \overline{OCS} in the event of an internal cache hit followed immediately by a cache miss or operand cycle.
8. This is a new specification that guarantees operation with the MC68881, which specifies a minimum time for \overline{DS} negated to \overline{AS} asserted (specification #13A). Without this specification, incorrect interpretation of specifications #9A and #15 would indicate that the MC68020 does not meet the MC68881 requirements.
9. This is a new specification that allows a system designer to guarantee data hold times on the output side of data buffers that have output enable signals generated with \overline{DBEN} .
10. These are new specifications that allow system designers to guarantee that an alternate bus master has stopped driving the bus when the MC68020 regains control of the bus after an arbitration sequence.

FIGURE 6.39d (continued)

MC68020 exception processing is similar in concept to the MC68000 with some minor variations. In the MC68020 exception processing occurs in four steps and varies according to the cause of the exception. The four steps are summarized below:

1. During the first step, an internal copy is made of the SR, and the SR is set for exception processing. This means that the status register enters the supervisor state and tracing is disabled.

TABLE 6.24 Exception Priorities and Recognition Times

Exception priorities			Time of recognition
Group 0	.0	Reset	End of clock cycle
Group 1	.0	Address error	
	.1	Bus error	Within an instruction cycle
Group 2	.0	BKPT #N, CALLM, CHK, CHK2, cp TRAPcc	
		cp mid-instruction	
		cp protocol violation, divide-by-zero, RTE, RTM, TRAP #N, TRAPV	
Group 3	.0	Illegal instruction, unimplemented LINE F, LINE A, privilege violation, cp preinstruction	Before instruction cycle begins
Group 4	.0	cp post-instruction	End of instruction cycle
	.1	Trace	
	.2	Interrupt	

Note: Halt and bus arbitration are recognized at end of bus cycle. 0.0 is highest priority; 4.2 is lowest.

- In the second step, the vector number of the exception vector is determined from either the exception requesting peripheral (nonautovector) or internally upon assertion of the AVEC (autovector) input. Note that in the MC68000, VPA is asserted for autovectoring. The vector base register points to the base of the 1-KB exception vector table which contains 256 exception vectors. The processor uses exception vectors as memory pointers to fetch the address of routines that handle the various exceptions.
- In the third step, the processor saves PC and SR on the supervisor stack. For coprocessor exceptions, additional internal state information is saved on the stack as well.
- The fourth step of the execution process is the same for all exceptions. The exception vector is determined by multiplying the vector number by four and adding it to the contents of the vector base register (VBR) to determine the memory address of the exception vector. The PC (and ISP for the reset exception) is loaded with the exception vector. The instruction located at the address given in the exception vector is fetched and the exception handling routine is thus executed.

Exception processing saves certain information on the top of the supervisor stack. This information is called the exception stack frame.

The MC68020 provides six different stack frames. The sizes of these frames vary from four words to forty six words depending on the exception. For example, the normal four word stack frame is generated by interrupts, privilege violations etc. A six-word stack frame is generated by instruction-related exceptions such as CHK/CHK2 and zero divide.

The MC68020 utilizes the concept of two supervisor stacks pointed to by MSP and ISP. The M-bit (when $S = 1$) determines the active supervisor stack pointer. The MC68020 accesses MSP when $S = 1$, $M = 0$. The MSP can be used for program traps and other exceptions, while the ISP can be used for interrupts. The use of two supervisor stacks allows isolation of user processes or tasks and asynchronous supervisor I/O tasks.

IPL2, IPL1, IPL0, AVEC, and IPEND pins are used as the MC68020 hardware interrupt control signals (Figure 6.40). The MC68020 supports seven levels of prioritized interrupts encoded by using IPL2, IPL1, IPL0 pins like the 68000.

In Figure 6.40, when an interrupting priority level 1 through 6 is requested, the MC68020 compares the interrupt level to the interrupt mask to determine whether the interrupt should be processed. An interrupt recognized as valid does not force immediate exception processing; a valid interrupt causes IPEND to be asserted, signaling to external devices that the MC68020 has an interrupt pending. Exception processing for a pending interrupt that begins at the next instruction boundary of a higher priority exception is also not currently valid. The DESKEW logic in Figure 6.40 continuously samples the IPL2-IPL0 pins on every falling edge of the

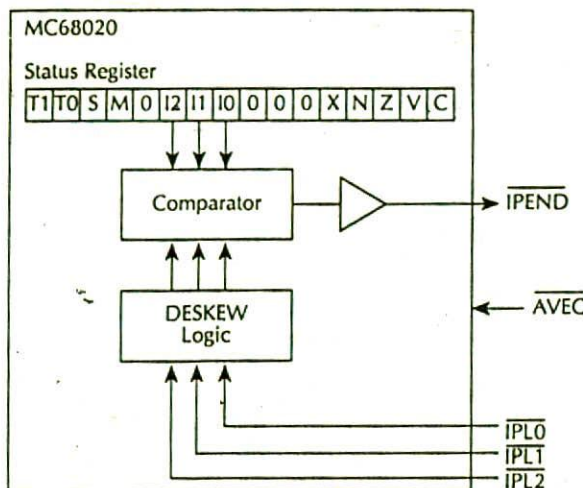


FIGURE 6.40 Interrupt control signals.

clock, but deskews or latches an interrupt request when it remains at the same level for two consecutive falling edges of the input clock. Figure 6.41 gives an example of the MC68020 interrupt deskewing logic.

Whenever the processor reaches an instruction execution boundary, it checks for a pending interrupt. If it finds one, the MC68020 begins an exception processing and executes an interrupt acknowledge cycle (IACK) with $FC2\ FC1\ FC0 = 111_2$ and $A19\ A18\ A17\ A16 = 1111_2$. The MC68020 basic hardware interrupt sequence is shown in Figure 6.42a. Figure 6.42b shows the interrupt acknowledge flowchart. Before the interrupt acknowledge cycle is completed, the MC68020 must receive either AVEC, DSACKX, or BERR; otherwise it will execute wait states until one of these input pins is activated externally.

If AVEC is asserted, the MC68020 automatically obtains the vector address internally (autovectored). If the MC68020 DSACKX pins are asserted, the MC68020 takes an 8-bit vector from the appropriate data lines (D0-D7 for 32-bit device, D16-D23 for 16-bit device, and D24-D31 for 8-bit device). This is called nonautovectored interrupts and the MC68020 obtains the interrupt vector address by adding VBR with $4 * (8\text{-bit vector})$.

Figure 6.43 shows an example of autovectored and nonautovectored interrupt logic. Finally, if the BERR pin is asserted, the interrupt is considered to be spurious and the MC68020 assigns the appropriate vector number for handling this.

Example: level 5 followed by level 7 request

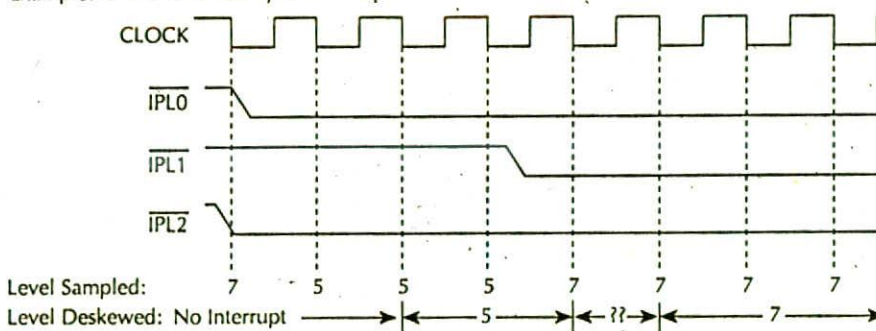


FIGURE 6.41 MC68020 interrupt deskewing logic.

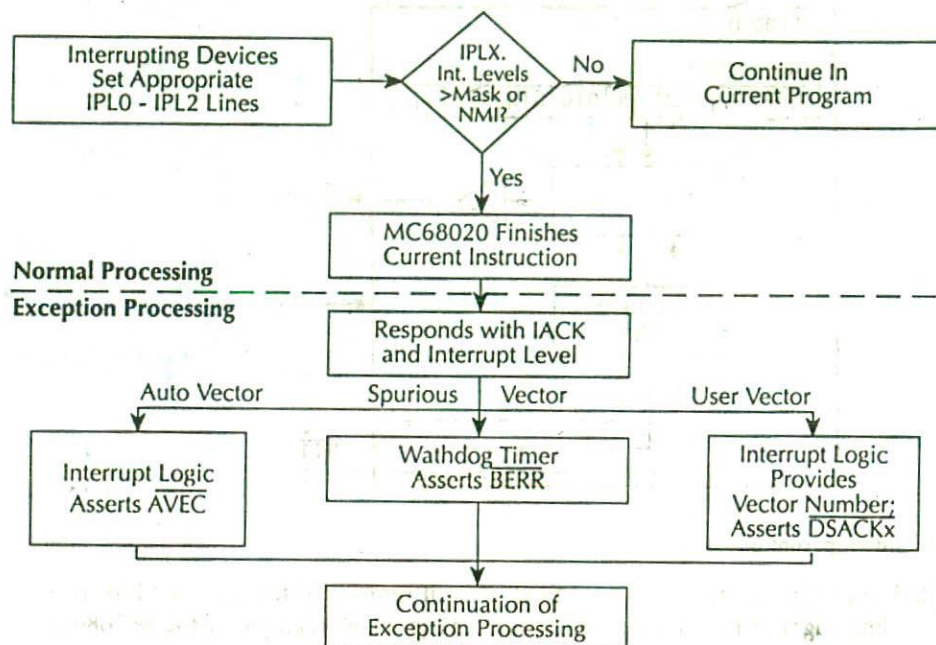


FIGURE 6.42a MC68020 basic hardware interrupt sequence.

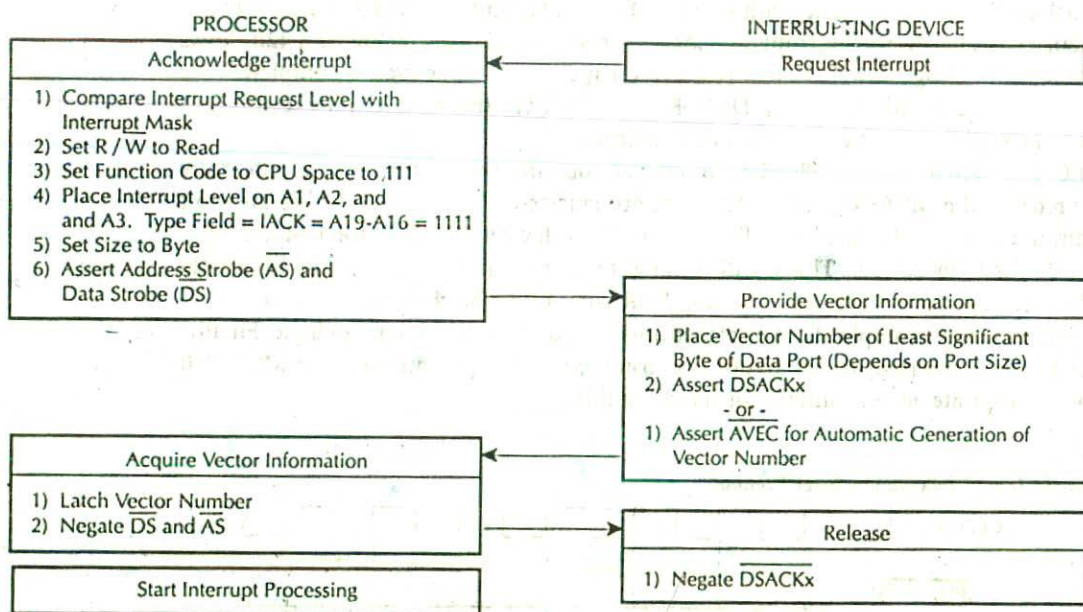


FIGURE 6.42b Interrupt acknowledge sequence flowchart.

6.13 MC68020 System Design

The following MC68020 system design will use a 128-KB, 32-bit wide memory and 8-bit parallel I/O port. The memory system is partitioned into four unique address space encodings: user data, user program, supervisor data, and supervisor program. This design uses RAMs for memory accesses and EPROMs for program memory accesses. Each address space has 32 KB

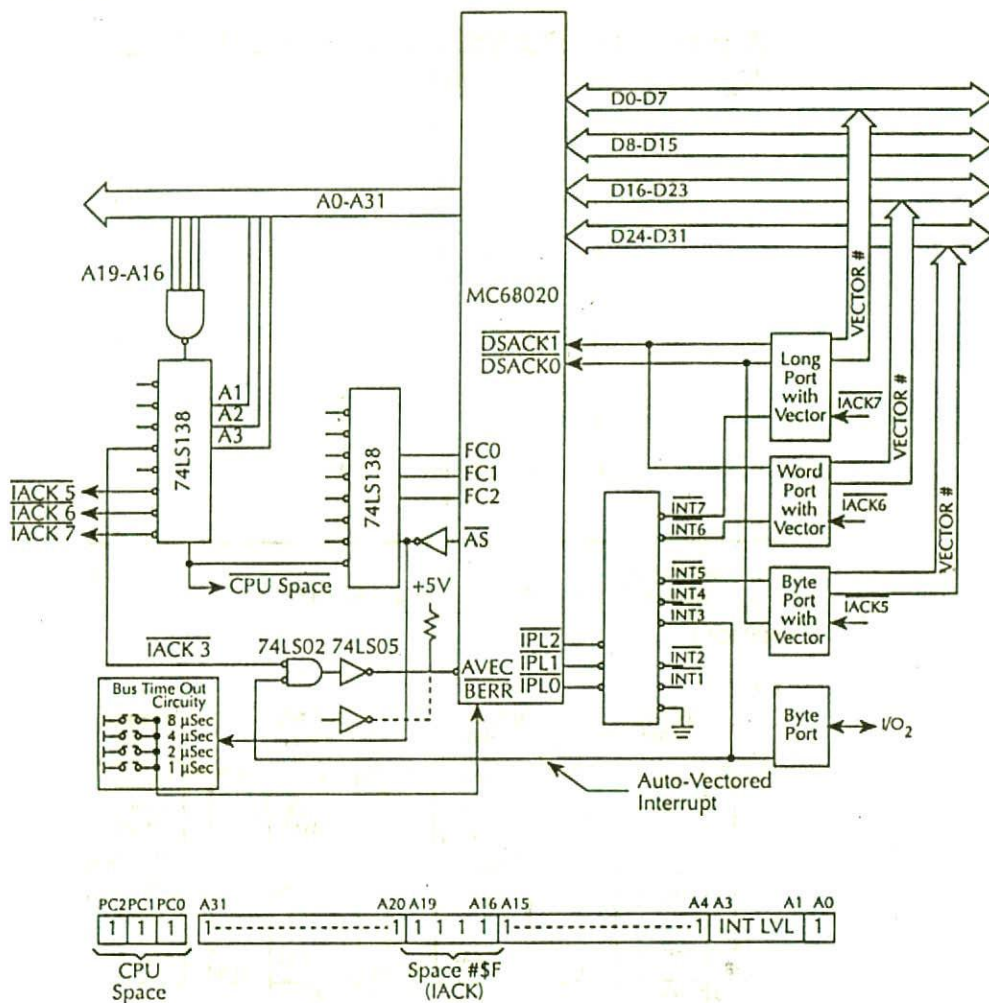


FIGURE 6.43 Autovectored and nonautovectored interrupt logic.

of memory available for use. Data I/O port space is appended to either user or supervisor data spaces (see MEMORY MAP); this is done by decoding the user/supervisor data space and address line A17.

The 32-bit-wide system memory consists of 4-byte-wide memories, each connected to its associated portion of the system data bus (D24-D31, D16-D23, D8-D15, and D0-D7). To manipulate this memory configuration, 32-bit data bus control byte enable logic is incorporated to generate byte strobes (DBBEE44, DBBEE33, DBBEE22, and DBBEE11) (Table 6.25).

The control byte enable state table shows the necessary individual byte strobe states as dictated by the MC68020's size (SIZ1, SIZ0) and address offset (A1, A0) encodings.

Karnaugh Maps (Table 6.26) for each data strobe signal have been created to identify the logic required to implement its state table requirement. The logic created for each data strobe is then combined into a complete 32-bit control logic schematic and connected to the memory structure as shown in the system hardware schematic diagram (Figure 6.44).

The system hardware design also identifies the required interconnections between the MC68020 MPU, the 74LS138 address space decoder, the EPSON SRM 20256LC (32KX8

TABLE 6.25 Control Byte Enable State Table for 32-Bit Device

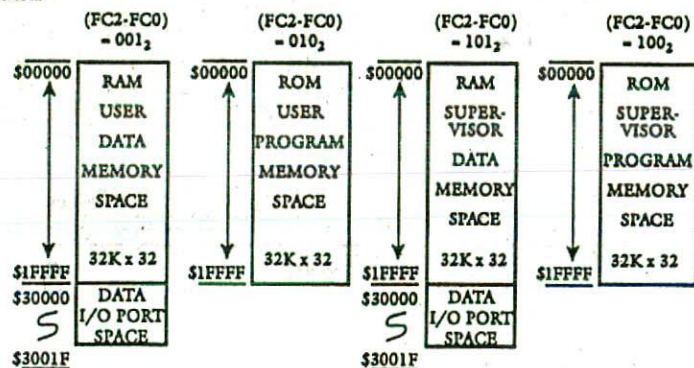
SIZE1	SIZE0	A1	A0	DBBE11	DBBE22	DBBE33	DBBE44
0	1	0	0	1	0	0	0
		0	1	0	1	0	0
		1	0	0	0	1	0
		1	1	0	0	0	1
1	0	0	0	1	1	0	0
		0	1	0	1	1	0
		1	0	0	0	1	1
		1	1	0	0	0	1
1	1	0	0	1	1	1	0
		0	1	0	1	1	1
		1	0	0	0	1	1
		1	1	0	0	0	1
0	0	0	0	1	1	1	1
		0	1	0	1	1	1
		1	0	0	0	1	1
		1	1	0	0	0	1

Hardware Design

68020 System with 128K × 32-Bit Memory and 8-Bit I/O Port

MEMORY

MAP



Assume all don't cares to be zeros.
A17=0 for memory, 1 for I/O.

TABLE 6.26 K Maps for Strobe Signals for 32-Bit Devices

SIZE1	SIZE0	A1 A0			
		00	01	11	10
00	00	1			
	01	1			
	11	1			
	10	1			

K-MAP₁

$$DBBE11 = \overline{A1} \cdot \overline{A0}$$

TABLE 6.26 K Maps for Strobe Signals for 32-Bit Devices (continued)

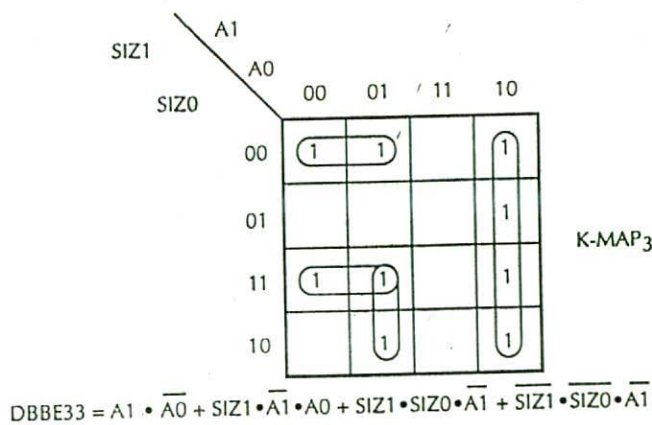
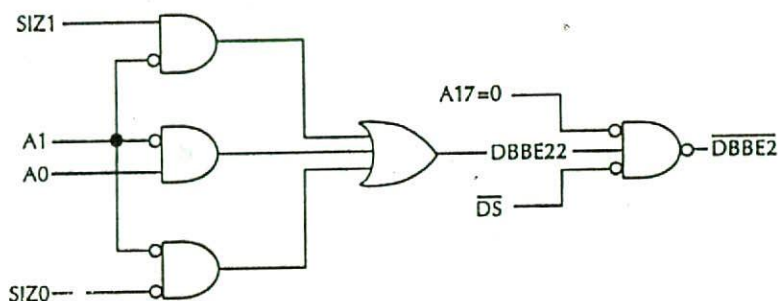
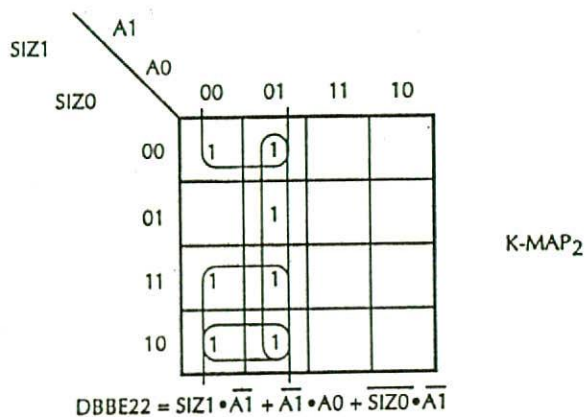
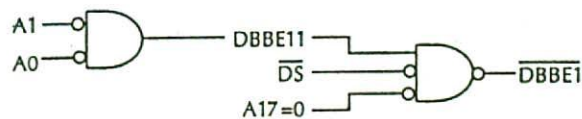
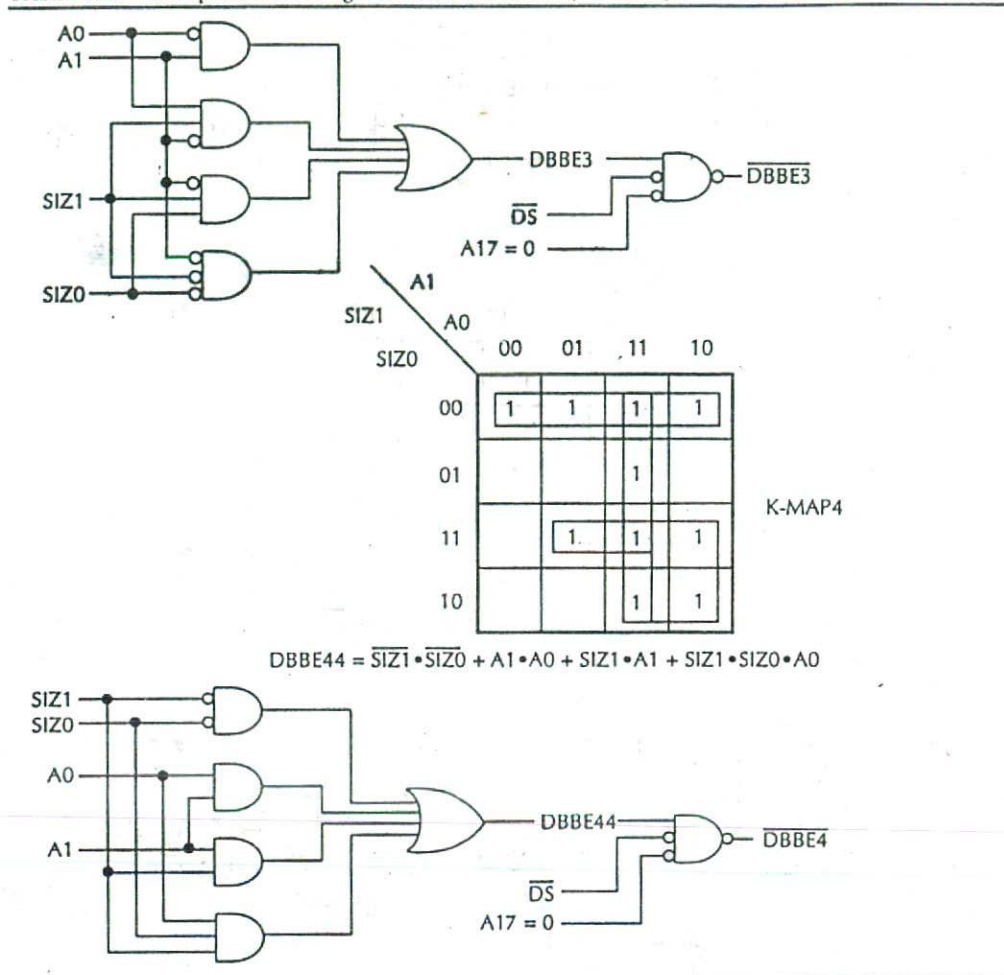


TABLE 6.26 K Maps for Strobe Signals for 32-Bit Devices (continued)



CMOS SRAM with 120 ns access time) user/supervisor data RAMs, the 27C256 (32KX8 CHMOS EPROM with 120 ns access time) user/supervisor program EPROMs, the 32-bit port control logic, and the MC68230 parallel I/O interface.

Since each memory is 32 KB \times 8, only MPU address lines A2-A16 are connected. The 74LS138 selects memory banks to enable, as dictated by the decoder FC2-FC0 signals. Control logic-generated data strobes (DBBE4-DBBE1) select which byte-wide portion of the data bus to activate. The 8-bit parallel I/O interface (MC68230) provides three bidirectional 8-bit ports as well as asynchronous handshake signals necessary for communication protocols.

The control byte enable logic diagram for generating DBBE1-DBBE4 is shown in Figure 6.45.

The enable logic uses A0, A1, SIZ0, and SIZ1 to decode DBBE1 — DBBE4. The memory is separated into four address space spaces: user data, user program, supervisor data, and supervisor program. The 68020 address pin A17 is used to select memory or I/O (A17 = 0 for memory, A17 = 1 for I/O). The 68020 DSACK1 and DSACK0 pins for 32-bit memory (DSACK1 = 0, DSACK0 = 0) are asserted by DBBE1 — DBBE4 via appropriate logic shown in Figure 6.44. For the 68230 8-bit I/O chip, the 68020 DSACK1 and DSACK0 pins must be asserted as DSACK1 = 0 and DSACK0 = 1 for I/O transfer. Note that DSACK1 = 1 and DSACK0 = 1 mean insert wait states.

The data enable signals ($\overline{\text{DBBE1}}$ — $\overline{\text{DBBE4}}$) are connected to the $\overline{\text{OE}}$ pin of each memory chip to ensure selection of correct memory segment. A sample window of at least 20 ns is needed for $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$; otherwise these signals will be unknown or illegal.

Each cycle of the 68020 with an 8-MHz clock is 125 ns. The 68020 samples $\overline{\text{DSACKX}}$ at the end of two clock cycles (250 ns for 8 MHz). If the 68020 $\overline{\text{DSACKX}}$ pins are asserted with at least 20 ns window, the 68020 will latch data at the end of three cycles (375 ns at the falling edge of S4).

Let us discuss the timing requirements of Figure 6.44.

The 27C256 EPROM and SRM 20256LC SRAM have access times of 120 ns each. The byte enable signals ($\overline{\text{DBBE1}}$ — $\overline{\text{DBBE4}}$) are derived from the 68020 $\overline{\text{DS}}$ and some other signals as shown in Figure 6.45. Among these signals, $\overline{\text{DS}}$ takes the longest to go to a LOW (approximately 1.2 cycles for a read and 2 cycles for a write). This means that each memory chip is enabled by the appropriate $\overline{\text{DBBE1}}$ — $\overline{\text{DBBE4}}$ signals after 150 ns for read (EPROM and SRAM) and 250 ns for write (SRAM) for an 8 MHz 68020. With 120 ns access time, the write operation for the SRAMs will take the longest (370 ns). Since the 68020 latches data at the end of three cycles (375 ns for 8 MHz) without any wait states, appropriate delays for assertion of $\overline{\text{DSACKX}}$ along with at least 20 ns window must be provided. A delay of 500 ns (arbitrarily chosen) with associated logic is included in Figure 6.44 to accomplish this. A ring counter can be used for the delay circuit.

The 68020-based microcomputer in Figure 6.44 includes 32-bit memory and 8-bit I/O. Note that $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$ signals are asserted as 00 for 32-bit, 01 for 8-bit, and 11 to insert wait states.

Consider timing requirements for 32-bit memory in Figure 6.44. When a memory chip is selected for read or write, one or more of the enable signals ($\overline{\text{DBBE1}}$ — $\overline{\text{DBBE4}}$) will be LOW. This will provide a LOW at the output of AND gate 1. This, in turn, will drive the $\overline{\text{DSACK0}}$ pin of the 68020 to a LOW. The 68020 AS pin along with the output of AND gate 1 at inputs of AND gate 3 will enable the 500 ns delay circuit. This will drive the 68020 $\overline{\text{DSACK1}}$ pin to a LOW after 500 ns providing at least 20 ns window for the 68020 $\overline{\text{DSACKX}}$ pins and appropriate timing for both EPROMs and SRAMs.

Next, consider timing requirements for the 8-bit I/O. After I/O is selected, the 68230 DTACK pin goes to a LOW and the memory enable lines ($\overline{\text{DBBE1}}$ — $\overline{\text{DBBE4}}$) are all HIGH. The output of AND gate 1 and inverted DTACK will drive $\overline{\text{DSACK0}}$ to a HIGH. Also, the output of AND gate 1 along with the 68020 AS pin will provide a LOW at the 68020 $\overline{\text{DSACK1}}$ pin after 500 ns indicating an 8-bit device. Thus, timing requirements for 8-bit I/O are satisfied.

QUESTIONS AND PROBLEMS

6.1 Find the contents of 68020 registers that are affected and the condition codes after execution of

- i) **ADD.B D2, D3**
- ii) **ADD.W D5, D6**
- iii) **ADDA.L A2, A4**

Assume the following data prior to execution of each of the above instructions:

$$[\text{D2}] = \$01\text{F462F1}$$


```

[D2] = $01F462F1
[D3] = $01001110
[D5] = $00008210
[D6] = $00001010
[A2] = $71240010
[A4] = $21040100

```

- 6.2 i) How many ALUs does the 68020 have? Comment on the purpose of each.
 ii) What is the purpose of the 68020 32-bit barrel shifter?
- 6.3 a) Summarize the basic differences between the 68000 and 68020.
 b) Discuss the differences between 68000 and 68020 debug capabilities implemented in their status registers. Will the instructions listed below cause trace or change of flow:
- i) **MOVE SR, D5**
 ii) **TRAPEQ START**
 when Z = 1?

6.4 Determine the number of bus cycles, the bytes written to memory (in Hex), and signal levels of A1, A0, SIZ1, and SIZ0 pins that would occur when the following 68020 instruction

```

MOVE.W D2, (A5) with
[D2] = $20161462 and
[A5] = $10057012

```

is executed. Assume:

- i) 16-bit memory
 ii) 8-bit memory

6.5 Show the contents of the affected 68020 registers and memory locations after execution of the instruction:

```
MOVE ($100, A5, D3.W *4), D1
```

Assume the following register and memory contents prior to execution of the above instruction:

```

[A5] = $0000F210
[D3] = $00001002
[D1] = $F125012A
[$00013318] = $4567
[$0001331A] = $2345

```

6.6 Show the contents of the affected 68020 registers and memory locations after execution of the instruction:

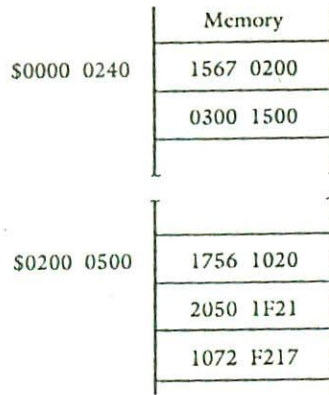
```
MOVE.B ([ $102, A2, D0.W * 2], $206), D1
```

Assume the following register and memory contents prior to execution:

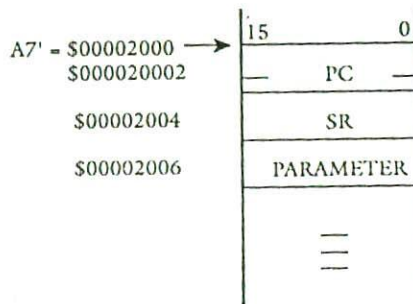
```

[A2] = $0000 0100
[D0] = $0000 0020
[D1] = $0000 0300

```



6.7 A subroutine in the supervisor mode is required to read a parameter from the stack configuration given below:



Write a 68000 instruction sequence to read the parameter into D5 and then write the equivalent 68020 instruction. Assume that the (A7') and the offset of the parameter (6) are known.

6.8 The 68000 instruction sequence below searches a table of 10_{10} 32-bit elements for a match. The address register A5 points to element 0, D3 contains the length (10_{10}) to be searched, and D2 contains the number to be matched. Find the 68020 single instruction which can replace lines 3, 4, and 5.

```

Line
1      MOVE.B #10, D3
2      SUBQ.B #1, D3
3  START  MOVE D3, D5
4          ASL.L #2, D5
5          CMP.L 0(A5, D5.L), D2
6          DBEQ D3, START
          -
          -
          -

```

6.9 Find the contents of D1, D2, A4, CCR and the memory locations after execution of the following 68020 instructions:

- i) `BFEXTS $5000 {8:16}, D4`
- ii) `BFINS D2, (A4) {D1:4}`
- iii) `BFSET $5000 {D1:10}`

Assume the following data prior to execution of each of the above instructions:

[D1] = \$0000 0004 [D4] = \$0000 3000
 [D2] = \$1234 5678 [A4] = \$0000 5000

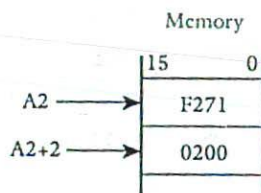
	7						0
-16	0	1	1	0	1	1	1
-8	1	1	1	0	1	1	1
\$5000 →	0	0	1	0	1	0	1
+8	0	1	0	1	1	1	0
+16	1	0	1	0	1	0	1

6.10 Find the 68020 condition codes after execution of CMP2.W (A2), D7. Determine the range of valid values. Indicate whether the comparison is signed or unsigned. Also, indicate the register values along with upper and lower bounds on the following:



Assume the following data prior to execution:

[D7] = \$F271 1020

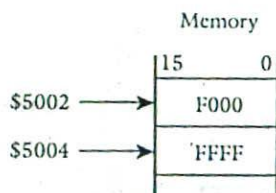


6.11 Find the 68020 condition codes and also determine if an exception occurs due to execution of

CHK2.W \$5002, A1

Assume the following data prior to execution:

[A1] = \$0000 F200



6.12 Fill in the missing hex values for the following 68020 instructions:

Address	Instruction	Label	Mnemonic
\$0200 0200	\$60 — —	START 0	BRA.B START2
.	.	.	.
\$0200 0206	\$60 — —	START.1	BRA.W START0
.	— —	.	.
.	.	.	.
\$0200 020F	.	START 2	.
.	.	.	.
.	.	.	.

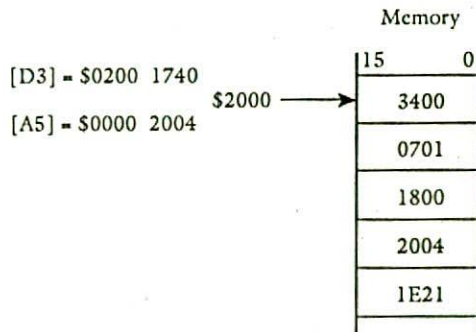
6.13 Identify the following 68020 instructions as valid or invalid. Comment if an instruction is invalid?

- i) **BFSET (A0) {-2:5}, D7**
- ii) **DIVS D5,D5**
- iii) **CHK.B D2, (A1)**

6.14 Determine the values of Z and C flags after execution of each of the following 68020 instructions:

- i) **CHK2.W (A5), D3**
- ii) **CMP2.L \$2001, A5**

Assume the following data prior to execution of each of the instructions:



6.15 Write a 68020 assembly language program to compute

$$Y = \sum_{i=1}^{20} X_i^2 / N$$

Assume X_i 's to be unsigned 32-bit numbers and the array starts at \$0000 2000.

6.16 Write a 68020 assembly language program to translate each of 10 packed BCD digits to its ASCII equivalent. Assume that the BCD digits are stored at an address starting at \$5000 and above. Store the ASCII bytes starting at \$6000.

- 6.17 What are the minimum times for a read bus cycle and a write bus cycle for a 16.67-MHz 68020?
- 6.18 What are the functions of the 68020 VBR, CACR, and CAAR?
- 6.19 Determine the values of FC2, FC1, FC0, SR, MSP, ISP, PC, A31-A0, D31-D0, SIZ1, SIZ0, and R/W of the 68020 upon hardware reset for the first 3 bus cycles. Assume the following memory contents prior to reset:

Memory	
	31 0
\$0000 0000	2000 0100
\$0000 0004	5000 9002
\$0000 0008	7001 2000
\$5000 9000	2100 3600
\$5000 9004	0100 F000

6.20 For a 25-MHz 68020

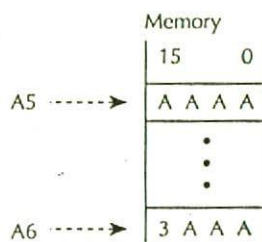
- Determine the length of time an address is valid during assertion of \overline{AS} for a read bus cycle. Assume no wait states.
 - Determine the length of time the data is valid when \overline{DS} is asserted during a write bus cycle. Assume no states.
- 6.21
- What happens to the 68020 when \overline{BERR} and \overline{HALT} pins are asserted together?
 - Identify which of the following 68020 instructions cause the RMC signal to be asserted: CAS2, TAS, CHK2, CALLM.
 - Which signals cause RMC to be asserted?
 - What happens when 68020 IPEND is asserted?
- 6.22
- Which exceptions of the 68020 are not available on the 68000?
 - What is a throwaway stack frame? When is it created?
 - List two 68020 instructions which may cause a format error exception.
 - How would the 68020 get out of a double bus fault?
- 6.23 Draw a neat schematic to interface a 4K EPROM, 4K RAM, and 2 I/O ports. Use 2716, 6116, 68230, and a 68020. Determine memory and I/O maps.
- 6.24 Assume a 68020/68881 system. Write a program in 68020 assembly language to find the area of a circle, $A = \pi r^2$, where r is the 32-bit radius of the circle, in single precision.
- 6.25 What are the purposes of the 68020 CALLM and RTM instructions?

6.26 Determine the contents of the 68020 registers, memory locations and condition code register after execution of CAS.B D2, D4, (A0). Assume the following data prior to execution of the instruction:

```
[D0] = $0000 0000
[D5] = $AAAA AAAA
[D6] = $0000 3AAB
[D1] = $0000 0001
```

6.27 Determine the contents of the 68020 registers and memory locations after execution of CAS2.W D5:D6, D0:D0, (A5):(A6). Assume following data prior to execution:

```
[D0] = $0000 0000
[D5] = $AAAA AAAA
[D6] = $0000 3AAB
[D1] = $0000 0001
```



- 6.28 i) Name two exception vectors for the MC68851 and the MC68881.
 ii) What is the size of the 68020 on-chip cache?
 iii) What is the 68020 cache access time?

6.29 What are the values of SIZ1, SIZ0, FC2, FC1, FC0, $\overline{R/W}$, and A31-A0 pins after execution of the 68020 BKPT #3 instruction?

6.30 Determine the values of FC2, FC1, FC0, and A31-A0 pins for a 68020 CPU space cycle with a floating-point coprocessor command register being accessed.

6.31 Assume a 68020/68881 system. Write a program in 68020 assembly language to find the magnitude of the complex number,

$$Z = X + iY$$

$$\text{where } i = \sqrt{-1}$$

Assume X and Y are 32-bit integers.

6.32 Determine the effects of executing CAS.B D3,D5,(A0). Assume the following data prior to execution:

```
[D3] = $2512 2551
[D5] = $7015 2652
[A1] = $5000 0004
[$5000 0004] = $5312 and
[CCR] = $0000 00102.
```


6.33 Write a 68020 assembly language instruction sequence using CAS for counting a semaphore. That is, the instruction sequence will increment a count in a shared location.

6.34 Determine the effects of executing CAS2.LD4:D5,D6:D4,(A5):(A6). Assume the following data prior to execution:

[D4] = \$0000 7000

[A5] = \$1234 5000

[D5] = \$0000 8000

[A6] = \$5000 0200

[D6] = \$0000 9000

6.35 Write a 68020 instruction sequence using CAS2 to insert an element in a double-linked list.

- 6.36 i) What are the main functions of the 68851 MMU?
ii) Summarize the address translation and protection mechanism of the 68851.
iii) How does the 68851 support the 68020 breakpoint function?

6.37 For a 16-bit device, use K-maps to express the memory byte enable lines, DBBE1 and DBBE2 in terms of 68020 A1, A0, SIZ1, SIZ0 and DS signals in minimized form. Note that for each expression, all 68020 signals mentioned above may not be necessary.

MOTOROLA MC68030/MC68040, INTEL 80486 AND PENTIUM MICROPROCESSORS

This chapter describes the basic features of the Motorola 68030/68040, the Intel 80486, and the Pentium microprocessors.

7.1 Motorola MC68030

The MC68030 is a virtual memory microprocessor based on an MC68020 core with additional features. The MC68030 is designed by using HCMOS technology and can be operated at 16.67-, 20-, and 33-MHz clocks.

The MC68030 contains all features of the MC68020, plus some additional features. The basic differences between the MC68020 and MC68030 are listed below:

Characteristics	MC68020	MC68030
On-chip cache	256-byte instruction cache	256-byte instruction cache and 256-byte data cache
On-chip Memory Management Unit (MMU)	None	Paged data memory management (demand page of the MC68851)
Instruction set	101	103 (four new instructions are for on-chip MMU; CALLM and RTM instructions are not supported)

Like the MC68020, the MC68030 also supports 7 data types and 18 addressing modes. The MC68030 I/O is identical to the MC68020. The enhancements to the MC68020 such as instruction cache and MMU, along with the basic MC68030 features, are described in the following section.

7.1.1 MC68030 Block Diagram

Figure 7.1 shows the MC68030 block diagram and includes the major sections of the processor.

The bus controller includes all the logic for performing bus control functions and also contains the multiplexer for dynamic bus sizing. It controls data transfer between the MC68030 and memory or I/O devices at the physical address.

The control section contains the execution unit and associated logic. Programmable Logic Arrays (PLAs) are utilized for instruction decode and sequencing.

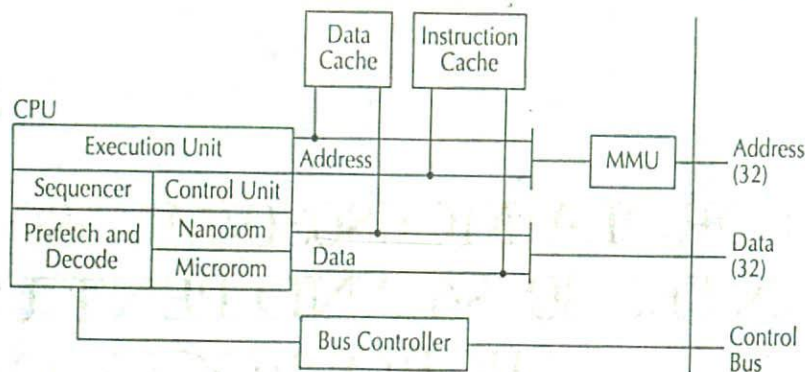


FIGURE 7.1 MC68030 block diagram.

The instruction and data cache units operate independently. They obtain information from the bus controller for future use. Each cache has its own address and data buses and thus permits simultaneous access. Both the caches are organized as 64 long word entries (256 bytes) with a block size of four long words. The data cache uses a write-through policy with no write allocation on cache misses.

The memory management unit maps address for page sizes from 256 bytes to 32K bytes. Mapping information stored in descriptors resides in translation tables in memory that are automatically searched by the MC68030 on demand. Most recently used descriptors are maintained in a 22-entry fully associative cache called the Address Translation Cache (ATC) in the MMU, permitting address translation and other MC68020 functions to occur simultaneously. The MMU utilizes the ATC to translate the logical address generated by the MC68030 into a physical address.

7.1.2 MC68030 Programming Model

Figure 7.2 shows the MC68030 programming model. The additional registers implemented in the 68030 beyond those of the 68020 are for supporting the MMU features. All common registers in the 68030 are the same as the 68020, except the cache control register which has additional control bits for the data cache and other new cache functions.

The 68030 additional registers are

- 32-bit translation control register (TC)
- 64-bit CPU root pointer (CRP)
- 64-bit supervisor root pointer (SRP)
- 32-bit transparent translation registers TT0 and TT1
- 16-bit MMU status register, MMUSR

The TC includes several fields that control address translation. These fields enable and disable address translation, enable and disable the use of SRP for the supervisor address space, and select or ignore the function codes in translating addresses. Other TC fields specify memory page sizes, the number of address bits used in translation, and the translation table structure.

The CRP holds a pointer to the root of the translation tree for the currently executing 68030 task. This tree includes the mapping information for the task's address space.

The SRP holds a pointer to the root of the translation tree for the supervisor's address space when the 68030 is configured to provide a separate address space for the supervisor programs.

Registers TT0 and TT1 can each specify separate memory blocks as directly addressable without address translation. Logical addresses in these areas are the same as physical addresses

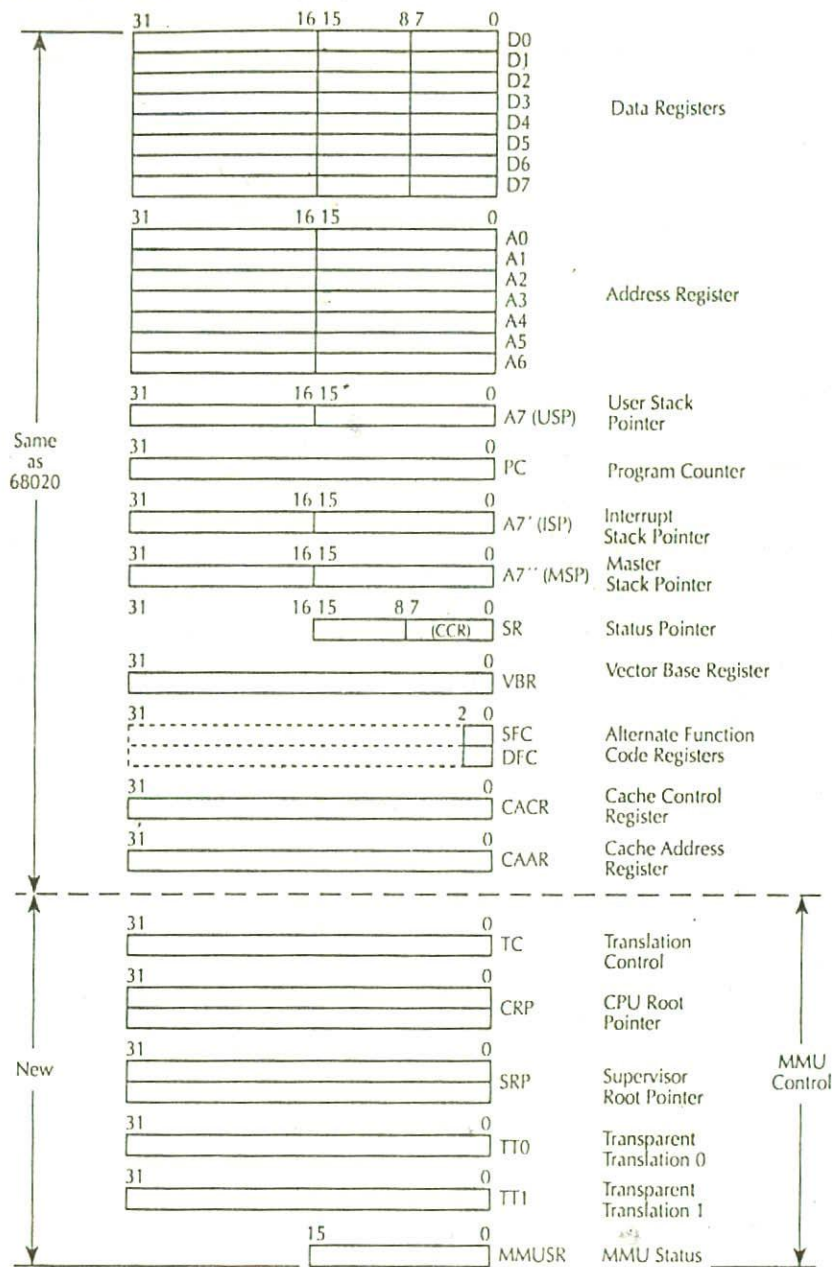


FIGURE 7.2 MC68030 programming model.

for memory access. Therefore, registers TT0 and TT1 provide fast movement of larger blocks of data in memory or I/O space, since delays associated with the translation scheme are not encountered. This feature is useful in graphics and real-time applications.

The MMUSR register includes memory management status information resulting from a search of the address translation cache or the translation tree for a particular logical address.

7.1.3 MC68030 Data Types, Addressing Modes, and Instructions

Like the MC68020, seven basic data types are supported on the MC68030:

TABLE 7.1 MC68030 New Instructions

Instruction	Operand syntax	Operand size	Operation
PFLUSH	(FC),#mask [(EA)]	None	Invalidate ATC entries at effective address
PLOAD	(FC),(EA),[R/W]	None	Create ATC entry for effective address
PMOVE	Rn,(EA)	16, 32	Register n → destination
	(EA),Rn	16, 32	Source → register n
PTEST	(FC), (EA),#level, [R/W]An	None	Information about logical address → PMMU status

- Bits
- Bit fields
- BCD digits
- Byte integers (8 bits)
- Word integers (16 bits)
- Long word integers (32 bits)
- Quad word integers (64 bits)

The 18 addressing modes of the MC68020 are also supported by the MC68030.

The MC68030 includes all MC68020 instructions (except CALLM and RTM), plus a subset of the MC68851 (PMMU) instructions. These instructions (Table 7.1) include PMOVE, PTEST, PLOAD, and PFLUSH, and they are compatible with the corresponding instructions in the MC68851 PMMU. The MC68020 requires the MC68851 coprocessor interface to execute these instructions. These instructions are executed by the MC68030 just like all other instructions.

All the MMU instructions are privileged and do not affect the condition codes. These new instructions are explained in the following sections.

7.1.3.a PMOVE Rn, (EA) or (EA), Rn

Rn can be any MMU register. (EA) uses control alterable addressing mode. The operand size depends on the MMU registers used as follows:

CRP, SRP	Quad word (64-bit)
TC, TT	Long word (32-bit)
MMUSR	Word (16-bit)

The PMOVE instruction moves data to and from the MMU registers. This instruction is normally used to initialize the MMU registers and to read the contents of the MMUSR for determining a fault.

As an example, consider PMOVE (A5), SRP. This instruction moves a 64-bit word pointed to by A5 to the supervisor root pointer.

7.1.3.b PTEST

The PTEST has four forms:

```

PTESTR (function code), (EA), #level
PTESTR (function code), (EA), #level, An
PTESTW (function code), (EA), #level
PTESTW (function code), (EA), #level, An

```

The PTEST instruction interrogates the MMU about a logical address and is normally used to determine the cause of a fault. The PTEST instruction executes a table search of the ATC or

the translation tables to a specified level for the translation descriptor corresponding to the (EA) and indicates the results of the search in the MMU status register.

The PTESTR or PTESTW means that the search is to be done as if the bus cycle is a read or a write. The details of the operand are given below:

- The function code is specified in one of the following ways
 - Immediate (three bits in the command word)
 - Data register (three least-significant bits of the data register specified in the instruction)
 - Source function code register
 - Destination function code register
- The (EA) operand provides the address to be tested.
- The level operand defines the maximum depth of table or number of descriptors to be searched. Level 0 means searching ATC only while levels 1 to 7 indicate searching the translation tables only.

When the address register, An, is specified for a translation table search, the physical address of the last table entry (last descriptor) fetched is loaded into the address register.

The MMUSR includes the results of the search.

As an example of PTEST, consider

PTESTR SFC, (A3), #4, A5

The function code for the page is in the Source Function Code register, SFC. The content of A3 is the logical address and the search is to be extended to 4 levels. Search is to be done as if for a read bus cycle and the physical address of the last entry checked is to go to A5.

The PTEST instruction is unsized and the condition codes are unaffected, but the MMUSR contents are changed.

7.1.3.c PLOAD

The PLOAD instruction loads an entry into the ATC. This is normally used in demand paging systems to load the descriptors into the ATC before returning to execute the instruction that caused the page fault.

The two forms for the instruction are

PLOADR (function code), (EA)
PLOADW (function code), (EA)

The function code is specified in one of the following ways:

- Immediate
- Data register
- Source function code register
- Destination function code register

(EA) can be control alterable addressing modes only. The PLOAD instruction searches the ATC for (EA) and also searches the translation table for the descriptor with respect to (EA). It is used to load a descriptor from the translation tables to the address translation cache.

The condition codes and MMUSR contents are unaffected.

As an example, consider PLOADR SFC, (A5). The function code for the desired page is in SFC. The logical address for which the descriptor is desired is in A5 and the descriptor is to be loaded as for a read bus cycle.

7.1.3.d PFLUSH

The PFLUSH instruction invalidates cache entries. The forms of PFLUSH are

PFLUSHA

PFLUSH (function code), mask

PFLUSH (function code), mask, (EA)

The operands can be specified as follows:

- (Function code) can be immediate (3 bits), data register (least significant 3 bits), SFC, or DFC.
- mask when set to one uses corresponding bit in function code for matching.
- (EA) can be any one of the control alterable addressing modes.

The instruction is unsized and condition codes and MMUSR contents are unaffected.

The PFLUSHA instruction invalidates all ATC entries. When (function code) and mask are specified in the PFLUSH instruction, the instruction invalidates all entries for the specified function code or codes. When the PFLUSH instruction also specifies (EA), the instruction invalidates the page descriptor for that effective address entry in each specified function code.

The mask operand includes three bits corresponding to the function code bits. Each bit in the mask that is set to one means that the corresponding bit of the function code operand applies to the operation. Each bit in the mask with zero value means that a bit of function code operand and the function code that is ignored. As an example, consider PFLUSH #1,4. The mask operand of 100_2 causes the instruction to consider only the most significant bit of the function code operand. Since the function code is 001_2 , function codes 000, 001, 010, and 011 are selected.

7.1.4 MC68030 Cache

The instruction cache in the 68030 is a 256-byte direct-mapped cache with 16 blocks. Each block contains four long words. Each long word can be accessed independently and thus 64 entries are possible with A1 selecting the correct word during an access. Figure 7.3 shows the 68030 instruction cache.

The index or a block (or line) is addressed by address lines A4 through A7 and each long word entry is selected by A2 and A3. The tag includes address lines A8 through A31 along with FC2.

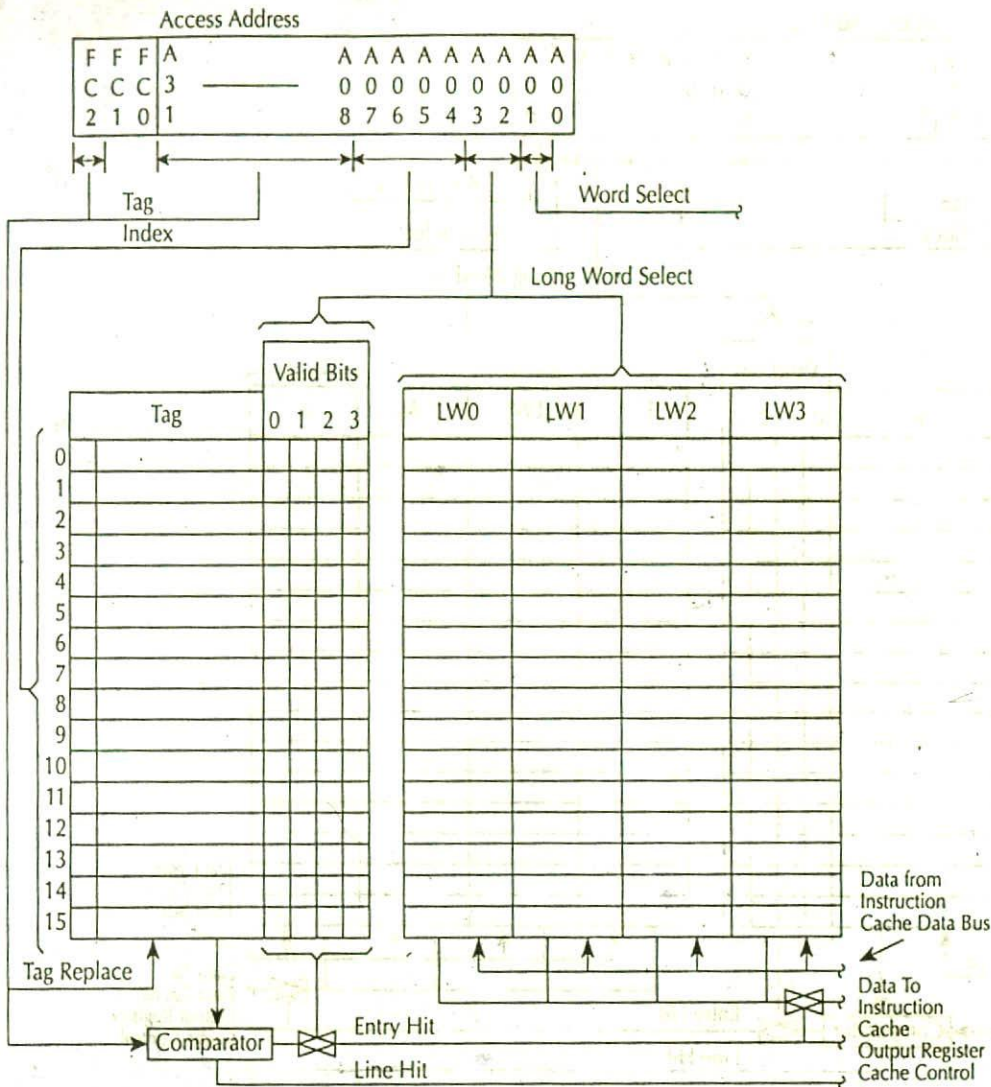
A0 is not used since instructions must be at even addresses (A0 is always zero).

An entry means that a line hit has occurred and the valid bit (four valid bits, one for each long word) for the selected entry is set. If an entry miss occurs, the cache entry can be updated with the instructions read from memory. If the cache is disabled, no cache hits or updates will take place, and if the cache is enabled but frozen, hits can occur without updates.

CACR can be used to clear cache entries and enable or disable the caches. The system hardware can disable the on-chip caches at any time by asserting the CDIS input pin.

Figure 7.4 shows the organization of the 68030 data cache. The data cache is organized in the same way as the instruction cache except that all three function code bits are used to determine a line hit. The data cache can be updated for both read and write. If the data cache is disabled, no hits or updates will occur. However, when the data cache is enabled and frozen, hits can occur and updates for write but not read can take place. The data cache can be updated for both read and write if enabled and not frozen.

The data cache utilizes a write-through policy with no write allocation of data writes. This means that if a cache hit takes place on a write cycle, both the data cache and the external device



- Cache size = 64 longwords
- Line size = 4 longwords
- Set size = 1 (direct mapped)
- For an entry hit to occur

- The access address tag field (A8-A31 and FC2) must match the tag field selected by the index field (A4-A7)
- The selected longword entry (A2-A3) must be valid
- The cache must be enabled in the Cache Control Register

FIGURE 7.3 MC68030 instruction cache.

are updated with the new data. If a write cycle generates a miss in the data cache, only the external device is updated and no data cache entry is replaced or allocated for that address.

Let us now consider some examples of the 68030 cache. Consider Figure 7.5 showing an instruction cache entry hit. The figure shows the CLR.W(A1) instruction with op code 4251_{16} at $PC = 11CCA29E_{16}$ to be accessed in user space ($FC2 = 0$). The most significant column in the tag field is the FC2 bit which is 1 for supervisor space and 0 for user space. The 68030 outputs 0 on FC2 and $11CCA29E_{16}$ on its address pins. Assume that the instruction cache is enabled. Since $A3-A0$ is 1110_2 , the address bits $A3-A2$ are 11_2 . This means LW3 in cache is

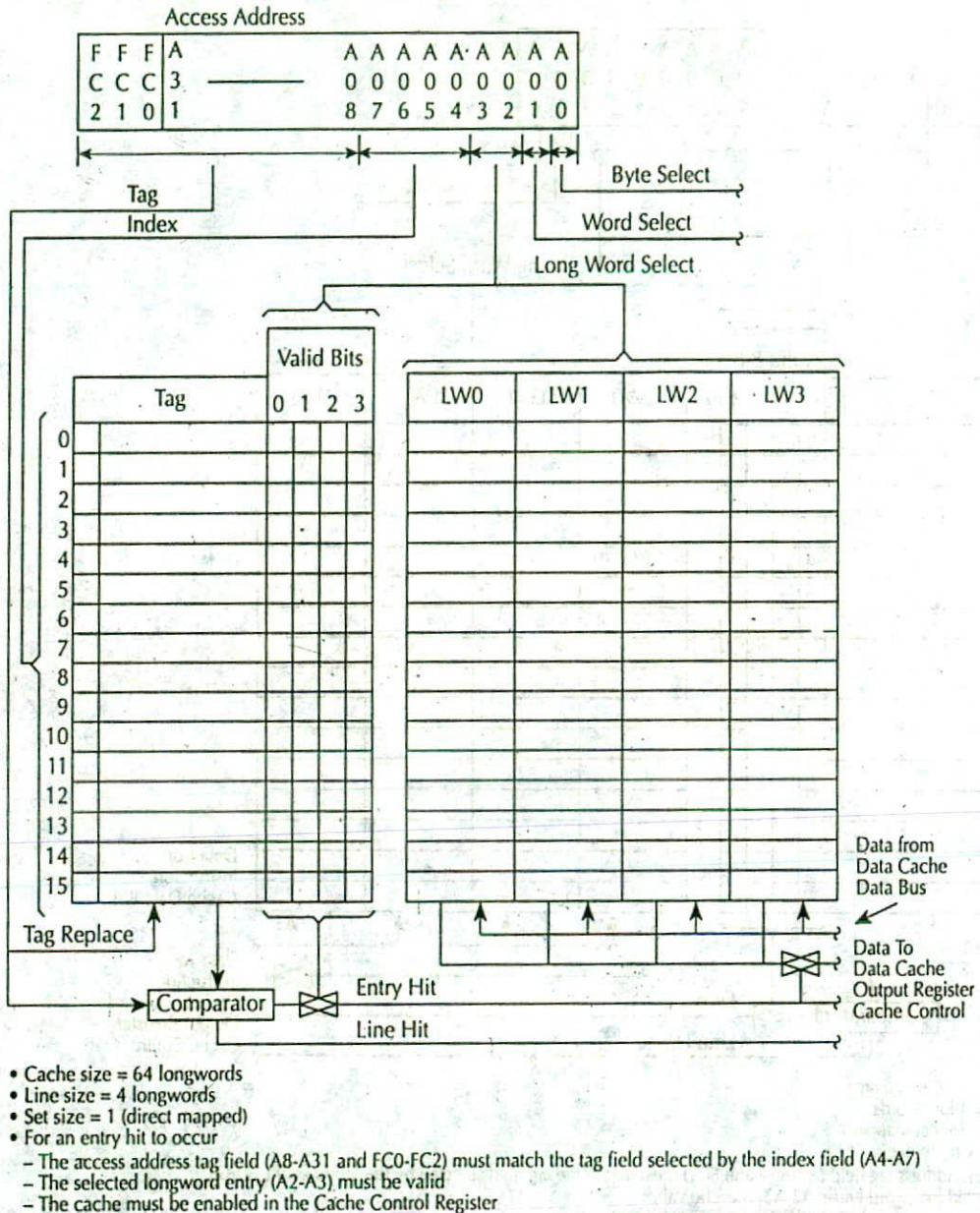


FIGURE 7.4 MC68030 data cache.

accessed. A31-A8 is $11CCA2_{16}$, which is used as the tag field with A7 through A4 (9_{10}) as the index field. A line hit occurs since the tag at index 9 in the cache matches A31 through A8 and $FC2 = 0$. Therefore, the valid bit 3 is set to one. The op code 4251_{16} is thus read into the cache output register.

Figure 7.6 illustrates an instruction cache miss. In this case, the op code 4251_{16} for $CLR.W(A1)$ stored at the PC value of $276F1A64_{16}$ in supervisor space ($FC2 = 1$) is to be accessed. The 68030 outputs one on the $FC2$ pin and $276F1A64_{16}$ on the A31-A0 pins. A miss occurs since the tag in line 6 ($057CD2_{16}$) does not match the address bits A31-A8 ($276F1A_{16}$) and $FC2 = 1$ does not match the function code bit (0), the most significant bit in the tag field. Assume the cache is enabled and not frozen. The 68030 reads 4251_{16} into LW1 from external memory and updates

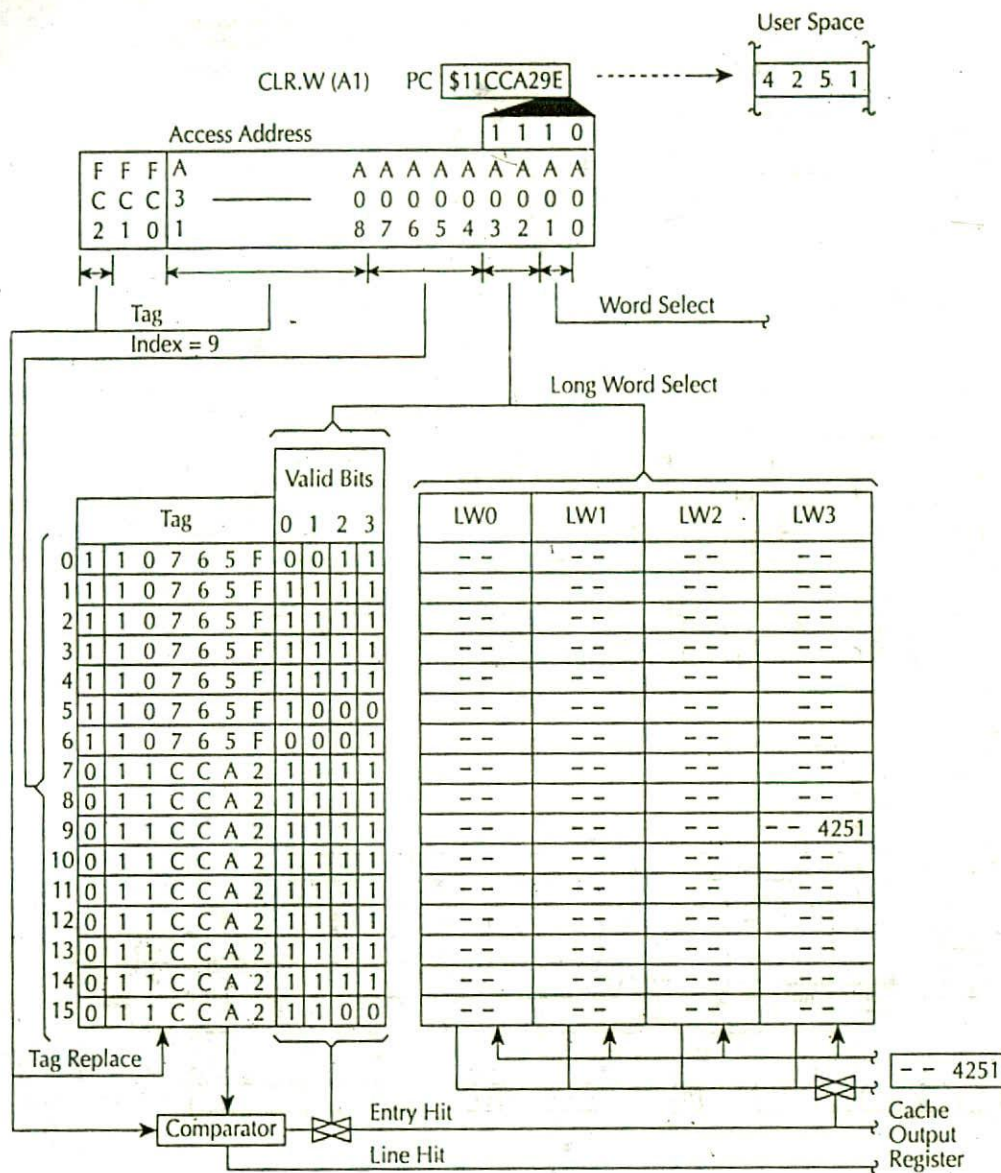


FIGURE 7.5 Instruction cache hit entry example.

the FC2 field and tag field with 1 and 276F1A₁₆, respectively. The valid bits are updated as 0100₂.

Figure 7.7 shows an example of a data cache hit. The 68030 executes MOVE.W(A1), D0 to read data C28F₁₆ at address 75B4A176₁₆ pointed to by A1 in user data space FC2 FC1 FC0 = 001. Note that the most significant column shows the value on the function code pins. Since A3A2 = 01, LW1 is accessed with tag value of 75B4A1₁₆ and index value of 7. Assume that the cache is enabled. Index = 7 since the tag value in the cache matches A31-A8 and the function code values (FC2 FC1 FC0 = 001) match the function code field value of 1 in the cache, a cache hit occurs. The valid bit 1 is set to one and datum C28F₁₆ is placed in the cache output register. Figure 7.8 shows an example of data cache miss. The 68030 executes the instruction MOVE.W(A1), D0 to read the contents of 01F376B8₁₆ pointed to by A1 into D0 in the supervisor data space (FC2 FC1 FC0 = 101₂). Since A3A2 = 10₂, LW2 in the cache is accessed. Assume the cache is enabled but

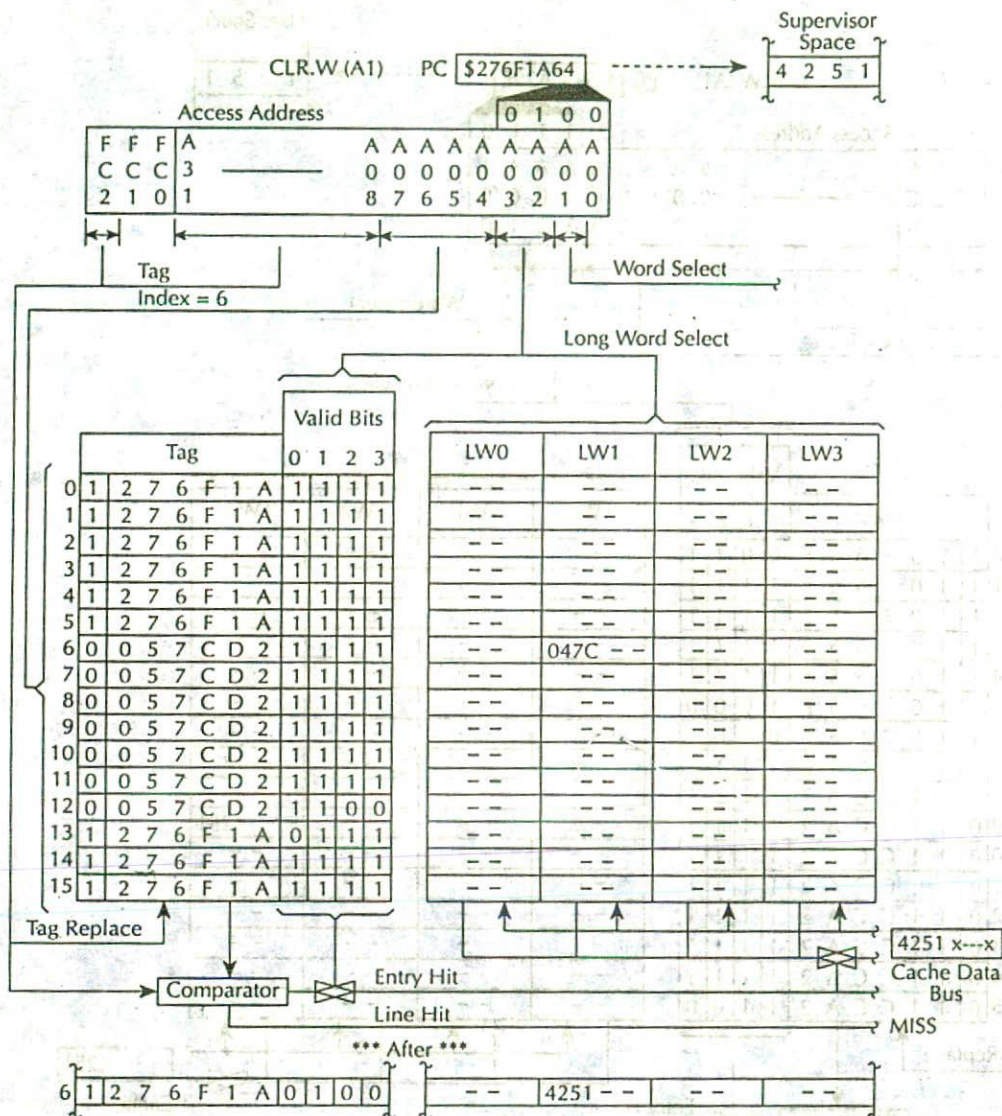


FIGURE 7.6 Instruction cache miss.

not frozen. Since the function code pins in cache do not match the 68030 function codes, cache miss occurs. Valid bit 2 is set to one. The 68030 obtains datum 1576₁₆ from external memory into low word of D0 and then updates the function code and tag fields of the cache. The 68030 then invalidates LW0, LW1, LW3 and validates LW2 by writing 0010₂ in the valid bits.

7.1.5 68030 Pins and Signals

The 68030 is housed in a 13×13 PGA package for 16.67 MHz and RC Suffix Package for 20 MHz.

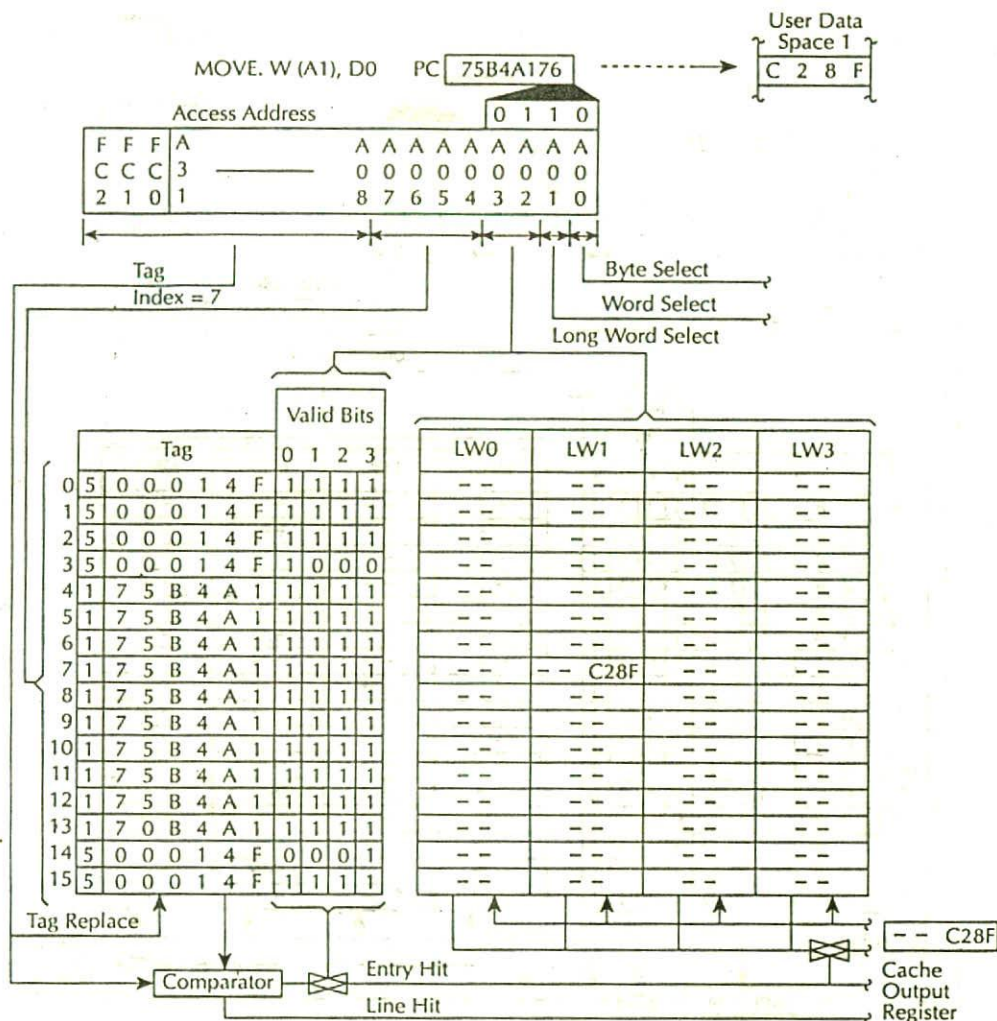


FIGURE 7.7 Data cache entry hit example.

Figure 7.9 shows the 68030 pin diagram. Figure 7.10 shows the 68030 functional signal groups. Table 7.2 summarizes the signal descriptions.

7.1.6 MC68030 Read and Write Timing Diagrams

The MC68030 provides three ways of data transfer between itself and peripherals. These are

- Asynchronous transfer
- Synchronous transfer
- BURST mode transfer

In asynchronous operation, the external devices connected to the system bus can operate at clock frequencies different from the MC68030 clock. Asynchronous operation requires only

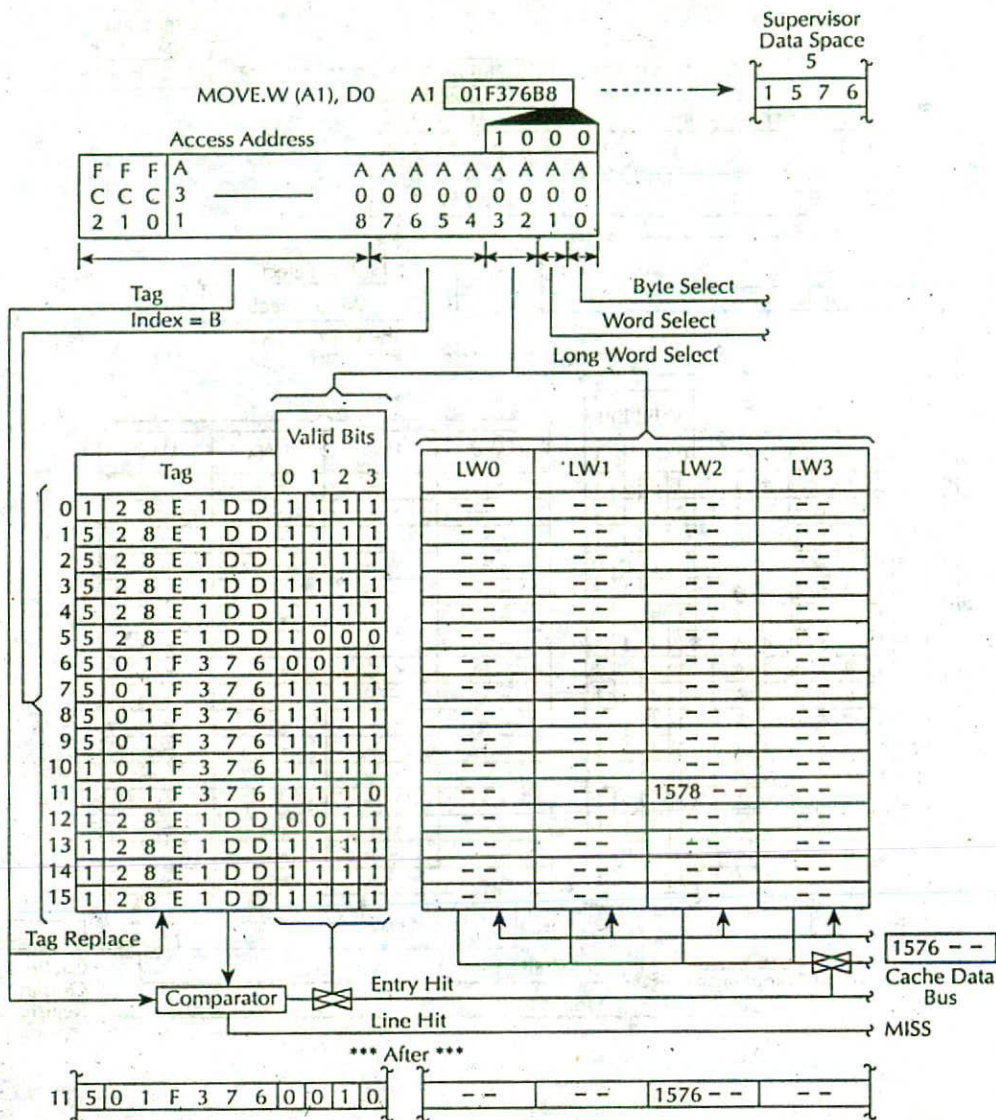


FIGURE 7.8 Data cache miss.

the handshake lines AS, DS, DSACKx, BERR and HALT. The asynchronous bus cycles of the MC68030 are similar to those of the MC68020. The MC68030 can transfer data in a minimum of three clock cycles. The dynamic bus sizing using the DSACKx signals can determine the amount of data transferred on a cycle-by-cycle basis.

Synchronous bus cycles are terminated with the STERM (synchronous termination) signal instead of DSACKx and always transfer 32-bit data in a minimum of two clock cycles instead of three clock cycles.

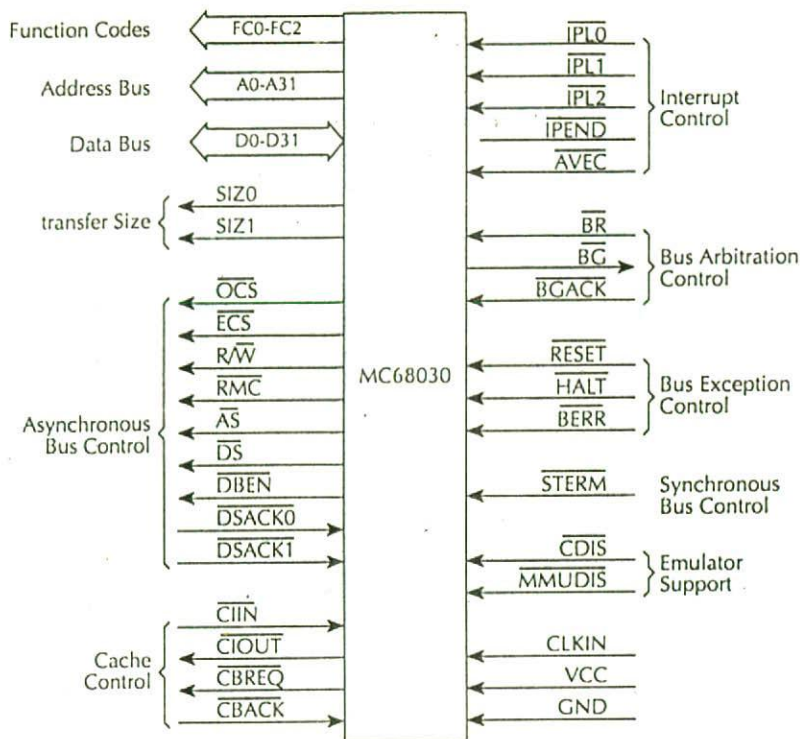


FIGURE 7.9 MC68030 functional signal groups.

The synchronous cycles terminated with STERM are for 32-bit devices only, while the synchronous cycles terminated by DSACKx can be for 8-, 16-, or 32-bit ports. The main difference between the use of STERM and DSACKx is that STERM can be asserted and data can be transferred earlier than for a synchronous cycle terminated with DSACKx. Wait cycles can be inserted by delaying the assertion of STERM if required.

BURST mode transfer is the most efficient technique of transferring data between the 68030 and external devices. This can be used to fill blocks of the instruction and data caches when the MC68030 asserts CBREQ (cache burst request).

BURST mode transfer takes place in synchronous operation and requires assertion of STERM to terminate each of its cycles. BURST mode is enabled by bits in the cache control register (CACR).

As an illustration of 68030 read/write timing diagrams, 68030 longword read and write cycles for asynchronous operation will be considered.

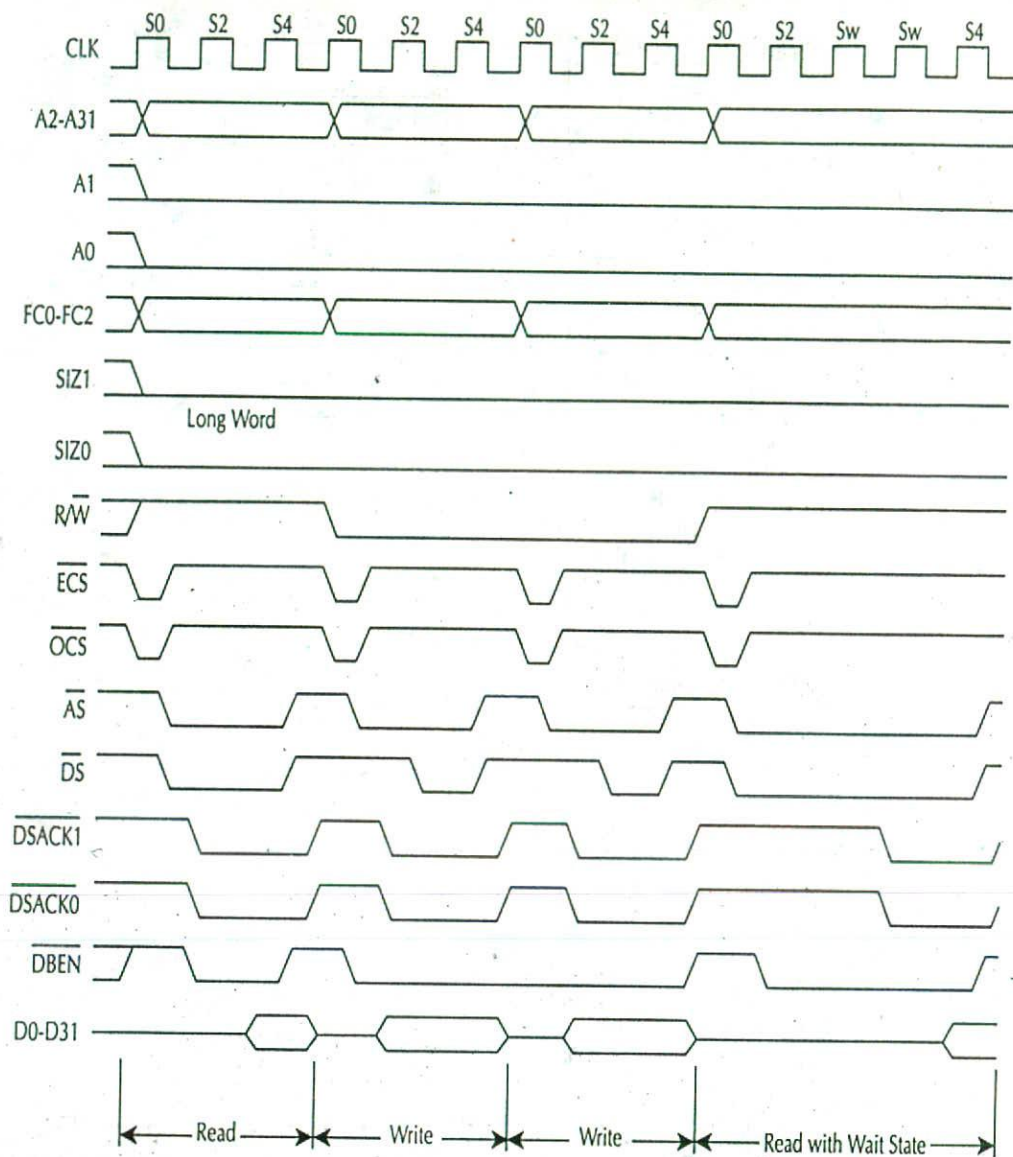


FIGURE 7.10 Asynchronous read-write cycles — 32-bit device.

Figure 7.10 depicts two write cycles (between two read cycles, with no idle time in-between) for a 32-bit device. The timing diagram in Figure 7.10 for read is explained as follows:

1. The read cycle for an instruction such as `MOVE.L (A0), D1` starts in state 0 (S0). The MC68030 drives the external cycle start (ECS) pin LOW indicating the start of an external cycle. When the cycle is the first external cycle of a read operand, the MC68030 outputs low on the operand cycle start (OCS) pin. The MC68030 then places a valid address (content of register A0 in this case) on A0-A31 pins and valid function codes on FC0-FC2 pins. The MC68030 drives R/W HIGH for read and drives data buffer enable (DBEN) inactive to disable data buffers. SIZ0 and SIZ1 signals become valid, indicating the number of bytes to be transferred. Cache inhibit out (CIOUT) becomes valid, indicating the state of the MMUCI bit in the address translation descriptor or in the selected TTx register.

TABLE 7.2 Signal Description

Signal name	Mnemonic	Function
Function codes	FC0-FC2	3-bit function code used to identify the address space of each bus cycle
Address bus	A0-A31	32-bit address bus used to address any of 4,294,967,296 bytes
Data bus	D0-D31	32-bit data bus used to transfer 8, 16, 24, or 32 bits of data per bus cycle
Size	SIZ0/SIZ1	Indicates the number of bytes remaining to be transferred for this cycle; these signals, together with A0 and A1, define the active sections of the data bus
Operand cycle start	$\overline{\text{OCS}}$	Identical operation to that of $\overline{\text{ECS}}$ except that $\overline{\text{OCS}}$ is asserted only during the first bus cycle of an operand transfer
External cycle start	$\overline{\text{ECS}}$	Provides an indication that a bus cycle is beginning
Read/write	R / W	Defines the bus transfer as an MPU read or write
Read-modify-write cycle	RMC	Provides an indicator that the current bus cycle is part of an indivisible read-modify-write operation
Address strobe	$\overline{\text{AS}}$	Indicates that a valid address is on the bus
Data strobe	$\overline{\text{DS}}$	Indicates that valid data is to be placed on the data bus by an external device or has been placed on the data bus by the MC68020
Data buffer enable	$\overline{\text{DBEN}}$	Provides an enable signal for external data buffers
Data transfer and size acknowledge	$\overline{\text{DSACK0}} / \overline{\text{DSACK1}}$	Bus response signals that indicate the requested data transfer operation is completed; in addition, these two lines indicate the size of the external bus port on a cycle-by-cycle basis and are used for asynchronous transfers
Cache inhibit in	$\overline{\text{CIIN}}$	Prevents data from being loaded into the MC68030 instruction and data caches
Cache inhibit out	$\overline{\text{CIOUT}}$	Reflects the CI bit in ATC entries; indicates that external caches should ignore these accesses
Cache burst request	$\overline{\text{CBREQ}}$	Indicates a burst request for the instruction or data cache
Cache burst acknowledge	$\overline{\text{CBACK}}$	Indicates that accessed device can operate in burst mode
Interrupt priority level	IPL0 - IPL2	Provides an encoded interrupt level to the processor
Interrupt pending	$\overline{\text{IPEND}}$	Indicates that an interrupt is pending
Autovector	$\overline{\text{AVEC}}$	Requests an autovector during an interrupt acknowledge cycle
Bus request	$\overline{\text{BR}}$	Indicates that an external device requires bus mastership
Bus grant	$\overline{\text{BG}}$	Indicates that an external device may assume bus mastership
Bus grant acknowledge	$\overline{\text{BGACK}}$	Indicates that an external device has assumed bus mastership
Reset	$\overline{\text{RESET}}$	System reset; same as 68020
Halt	$\overline{\text{HALT}}$	Indicates that the processor should suspend bus activity
Bus error	$\overline{\text{BERR}}$	Indicates an invalid or illegal bus operation is being attempted
Synchronous termination	$\overline{\text{STERM}}$	Bus response signal that indicates a port size of 32 bits and that data may be latched on the next falling clock edge
Cache disable	$\overline{\text{CDIS}}$	Dynamically disables the on-chip cache to assist emulator support
MMU disable	$\overline{\text{MMUDIS}}$	Dynamically disables the translation mechanism of the MMU
Clock	CLK	Clock input to the processor
Power supply	Vcc	+5 volt \pm 5% power supply
Ground	GND	Ground connection

- During S1, the MC68030 activates $\overline{\text{AS}}$ LOW, indicating a valid address on A0-A31. The MC68030 then activates $\overline{\text{DS}}$ LOW and negates $\overline{\text{ECS}}$ and $\overline{\text{OCS}}$ (if asserted).
- During S2, the MC68030 activates the $\overline{\text{DBEN}}$ pin to enable external data buffers. The selected device such as a memory chip utilizes $\overline{\text{DS}}$, SIZ0, SIZ1, R / W, and $\overline{\text{CIOUT}}$ to place data on the data bus. The MC68030 outputs LOW on $\overline{\text{CIIN}}$ if appropriate. Any or all bytes on D0-D31 are selected by SIZ0, SIZ1, A0, and A1. At the same time, the selected device asserts the $\overline{\text{DSACKx}}$ signals. The MC68030 samples $\overline{\text{DSACKx}}$ at the falling edge of the S2 and if $\overline{\text{DSACKx}}$ is recognized, the MC68030 inserts no wait states.

As long as at least one of the $\overline{\text{DSACKx}}$ inputs is asserted by the end of S2 (satisfying the asynchronous setup time requirement), the MC68030 latches data on the next falling edge of the clock and ends the cycle.

If $\overline{\text{DSACKx}}$ is not recognized by the MC68030 by the beginning of S3, the MC68030 inserts wait states and $\overline{\text{DSACKx}}$ must remain HIGH throughout the asynchronous input setup and hold times around the end of the S2. During the MC68030's wait states, the MC68030 continually samples $\overline{\text{DSACKx}}$ on the falling edge of each of the subsequent cycles until one $\overline{\text{DSACKx}}$ is asserted.

4. With no wait states, at the end of S4, the MC68030 latches data.
5. During S5, the MC68030 negates AS, DS, and DBEN. The MC68030 keeps the address valid during S5 to provide address hold time for memory systems. R/W, SIZ0, and SIZ1, and FC0-FC2 also remain valid during S5.

The timing diagram in Figure 7.10 for write is explained as follows:

1. The MC68030 outputs LOW on both $\overline{\text{ECS}}$ and $\overline{\text{OCS}}$ and places a valid address and function codes on A0-A31 and FC2-FC0, respectively. The MC68030 also places LOW on R/W and validates SIZ0, SIZ1, and CIOOUT.
 2. In S1, the MC68030 asserts AS and DBEN and negates $\overline{\text{ECS}}$ and $\overline{\text{OCS}}$ (if asserted).
 3. During S2, the MC68030 outputs data to be written on D0-D31 and samples $\overline{\text{DSACKx}}$ at the end of S2.
 4. During S3, the MC68030 outputs LOW on $\overline{\text{DS}}$, indicating that the data are stable on the data bus. As long as at least one of the $\overline{\text{DSACKx}}$ inputs is recognized by the end of S2, the cycle terminates after one cycle. If $\overline{\text{DSACKx}}$ is not recognized by the start of S3, the MC68030 inserts wait states. If wait states are inserted, the MC68030 continues to sample the $\overline{\text{DSACKx}}$ signals on the subsequent falling edges of the clock, until one of the $\overline{\text{DSACKx}}$ inputs is recognized. The selected device such as memory utilizes R/W, DS, SIZ0, SIZ1, and A0 and A1 to latch data from the appropriate portion of D0-D31 pins.
- The MC68030 generates no new control signals during S4.
5. During S5, the MC68030 negates AS and DS. The MC68030 holds the address and data valid to provide address hold time for memory systems. The processor also keeps R/W, SIZ0, SIZ1, FC0-FC2, and DBEN valid during S5.

7.1.7 MC68030 On-Chip Memory Management Unit

7.1.7.a MMU Basics

A Memory Management Unit (MMU) translates addresses from the microprocessor (logical) to physical addresses. Logical addresses are assigned to the task when it is linked, while physical addresses are assigned at the time the task is loaded into memory based on free physical addresses. The MMU keeps a task in its own address space. If a task attempts to go out of its own address space, the MMU asserts a bus error. Besides protection, this feature is valuable in demand paging systems.

An operating system must know the free physical memory addresses available so that it can load the next task to these spaces. One way of accomplishing this is by dividing the memory into contiguous blocks of equal size. These are called pages on the logical side of the MMU and page frames on the physical side. The system memory will contain page descriptions which point to the page frames and status of the page frames.

The page size determines the number of address bits to be translated by the MMU and the number which address memory directly. The lower bits of the logical address related to the page size are not translated by the MMU and go directly to the physical address bus as shown in Figure 7.11. Note that A8-A3 bits provide the page number and A0-A7 bits define the 256-byte page offset in this case.

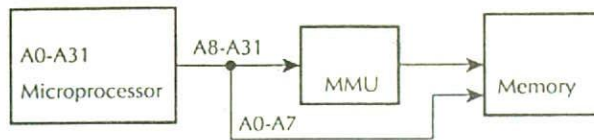


FIGURE 7.11 Logical to physical translation.

A translation table can be used to translate pages to page frames. Figure 7.12 shows an example of table translation of pages. The task control block contains a pointer to the starting address of the translation table. A logical address is translated by accessing the location in the translation table determined by starting address of translation + (page number * entry size). The entry accessed contains the page frame number. Note that entry size is 4 for 32 bits (4 bytes).

Each task has a translation table. Single or multiple translation tables can be used. Figure 7.13 shows an example of a single pointer level translation. The TCB contents \$00010000 contain the starting address of the translation table. The 12-bit page offset (\$C5E for P2) replaces the low 12 bits of the physical address directly. The logical address for P2 is translated by accessing the translation table at the location

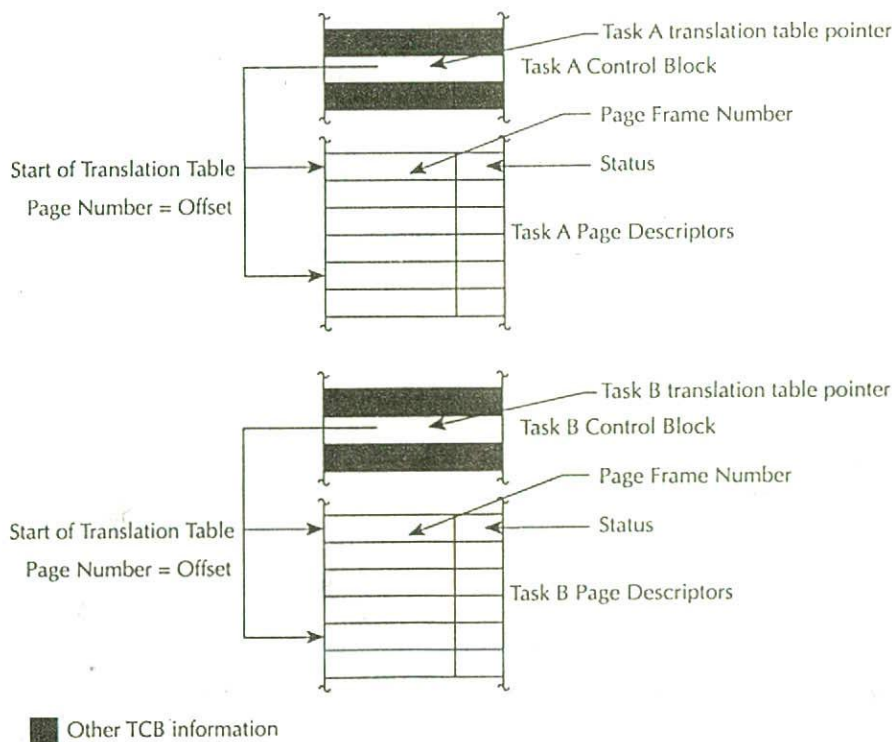
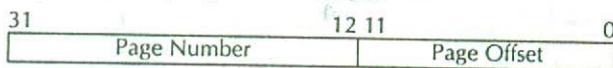
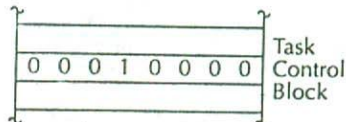


FIGURE 7.12 Table translation of pages.

Logical Address, Page Size = 4K Bytes



Example

☐ Other TCB information

	Page		
P1	0	\$10000	7 4 5 A 7
P2	1	4	3 6 2 7 1
	2	8	F 0 1 E 2
	3	C	A 7 6 5 3
	4	10	C C 2 6 0
P3	5	14	2 B E 4 9

Page Descriptor Table

Translation Examples

	Logical	Physical
P1	\$0123	\$745A7123
P2	\$1C5E	\$36271C5E
P3	\$5678	\$2BE49678

• If a task has only 2 pages, 0 and \$FFFF, the page descriptor table must be the same size as if the task used all pages. Size $2^{20} \times 4$ bytes = 1048576 \times 4 = 4194304 bytes.

• Each task has a translation table.

FIGURE 7.13 Single-pointer level translation.

$$\begin{aligned}
 &= \$10000 + (\text{Page number} * \text{entry size}) \\
 &= \$10000 + (\$00001 * 4) \\
 &= \$100004
 \end{aligned}$$

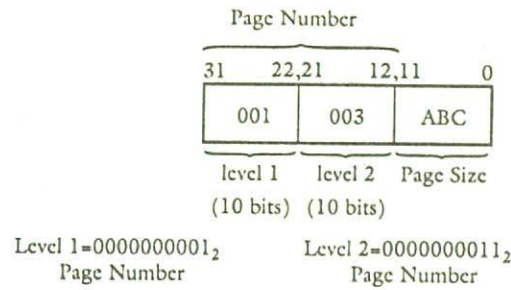
Therefore, location \$100004 in the translation table is accessed and its content \$36271 is obtained as the page frame number. The 12-bit page offset \$C5E is concatenated with the page frame number to obtain the physical address in page P2 as \$36271C5E.

One of the disadvantages of the single-level translation is that for most systems, the required table sizes would be too large. Therefore, multilevel translation tables are used. For example, consider the double pointer level translation table of Figure 7.14. In this case, the TCB contains a pointer to the starting address of the level 1 translation table. Level 1 descriptors contain pointers to level 2 page descriptors.

A logical address is translated in two steps:

1. A pointer is obtained from level 1 translation table as follows: pointer from TCB = starting address of level 1 translation table + (level 1 * entry size of level 1 table).
2. This pointer is used to access a location in level 2 table by adding the pointer obtained from level 1 table with (level 2 * entry size of level 2 table).

The location in table 2 thus obtained contains the page frame number. Consider the logical address:



If TCB contents (\$10000) point to the starting address of table 1, the level 1 location

$$\begin{aligned}
 &= \$10000 + \text{level 1} * \text{entry size of level 1} \\
 &= \$10000 + 1 * 4 \\
 &= \$10004
 \end{aligned}$$

Therefore, if the content of \$10004 is \$21000 pointing to the starting address of the level 2 table, then the accessed location in table 2

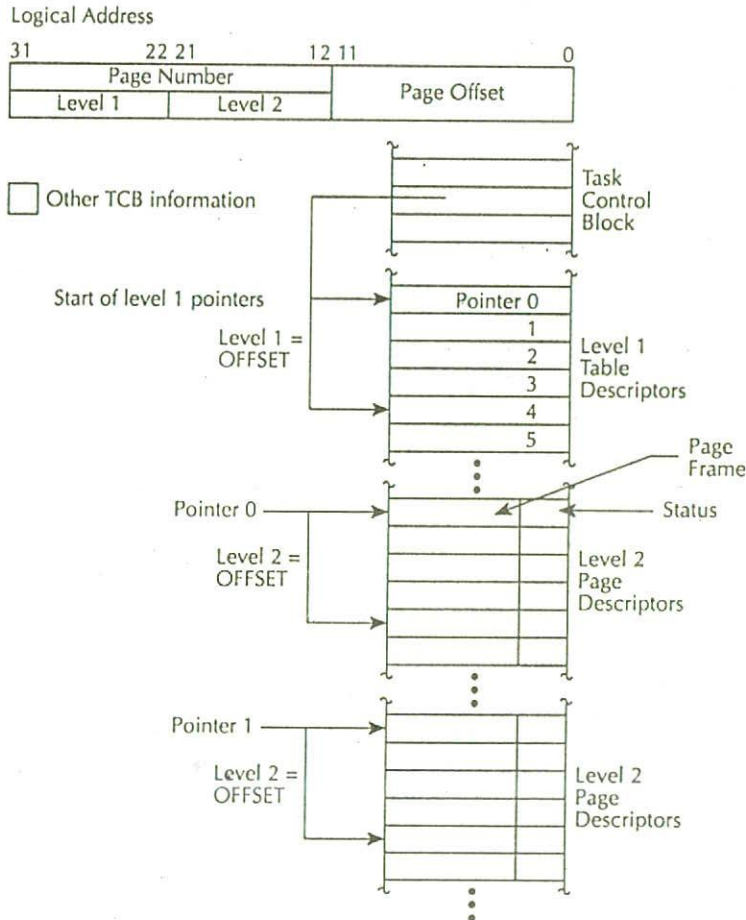


FIGURE 7.14 Double pointer level translation.

$$\begin{aligned}
 &= \$21000 + (3_{10} * 4) \\
 &= \$2100C
 \end{aligned}$$

Assume the content of \$2100C is \$25719. Then \$25719 is concatenated with the page offset \$ABC to obtain the 32-bit physical address \$25719ABC.

An example of the double-pointer level translation is shown at the top of the next page. Assume a page size of 4K bytes. Let us translate the following logical addresses to physical addresses for the task shown above:

\$0000072A and \$00401A59

Consider the logical address \$0000072A. This address has level 1 and level 2 page numbers of 0. This means that level 1 location

$$\begin{aligned}
 &= \$00780000 + 0 * 4 \\
 &= \$00780000
 \end{aligned}$$

Therefore, [\$00780000] points to the starting address of the level 2 table; then the accessed location in table 2

$$\begin{aligned}
 &= \$00800000 + 0 * 4 \\
 &= \$00800000
 \end{aligned}$$

The content of location \$00800000 (\$176A5) is concatenated with the page offset \$72A so that the physical address is \$176A572A.

Next, consider the logical address \$00401A59. The uppermost 10 bits (0000 0000₁₂) define the level 1 page number as one. The next 10 bits (0000 0000₁₂) define the level 2 page number as one. The page size is \$A59. Level 1 location

$$\begin{aligned}
 &= \$7800\ 0000 + (1 * 4) \\
 &= \$7800\ 0004
 \end{aligned}$$

The content of location \$7800 0004 (\$00801000) points to the starting address of level 2 table. Level 2 location

$$\begin{aligned}
 &= \$00801000 + 1 * 4 \\
 &= \$0080\ 1004
 \end{aligned}$$

The content of location \$0080 1004 (\$91024) is concatenated with the page offset \$A59 to obtain the physical address \$91024 A59.

The main advantage of the multiple translation table is that it reduces the required translation table size significantly. However, multiple memory accesses (two in the example of two-level translation) are required to obtain a descriptor. However, an address translation cache can be used to speed up memory access.

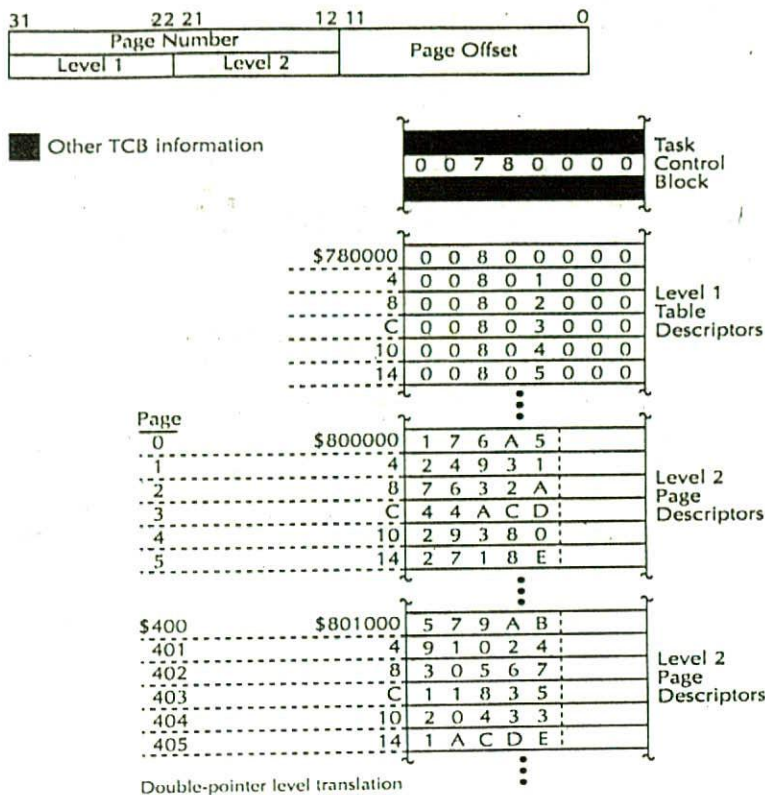
An MMU provides three basic functions:

- Translates page number to page frame number
- Restricts access, i.e., a task is restricted to its own address space.
- Provides write protection by not allowing write access to write-protected pages

Status information is included in the low bits of the translation table entry data to provide protection information.

7.1.7.b 68030 On-chip MMU

Figure 7.15 provides a block diagram of the MC68030 and identifies the on-chip MMU.



The 68030 on-chip MMU translates logical addresses to physical addresses. If the desired information is in the address translation cache, no time delay occurs due to MMU address translation. The page descriptor is obtained by the MMU by searching the translation tables if needed.

The pins used by the 68030 on-chip MMU are A31-A0, CIOUT, and MMUDIS.

The MMU outputs the translated physical addresses (from logical addresses) or the addresses for fetching translation data (when a table is searched) on the A31-A0 pins. The MMU asserts the CIOUT pin for pages which are defined as noncacheable. The MMUDIS pin, when asserted by an external emulator, disables the MMU.

The 68030 executes a normal bus cycle when the address translation information is in the ATC. However, if the address translation information is not in the ATC, the 68030 executes additional bus cycles to obtain the desired address translation information.

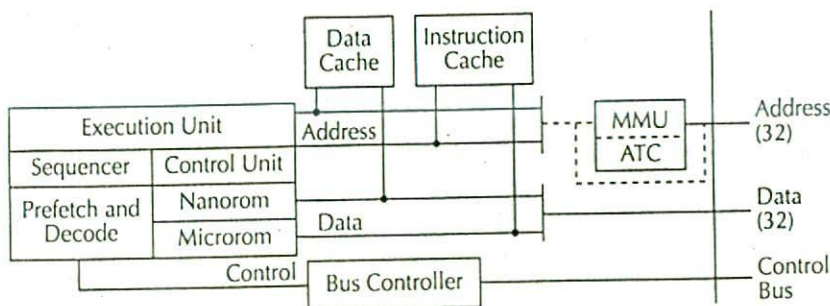


FIGURE 7.15 MC68030 on-chip MMU block diagram.

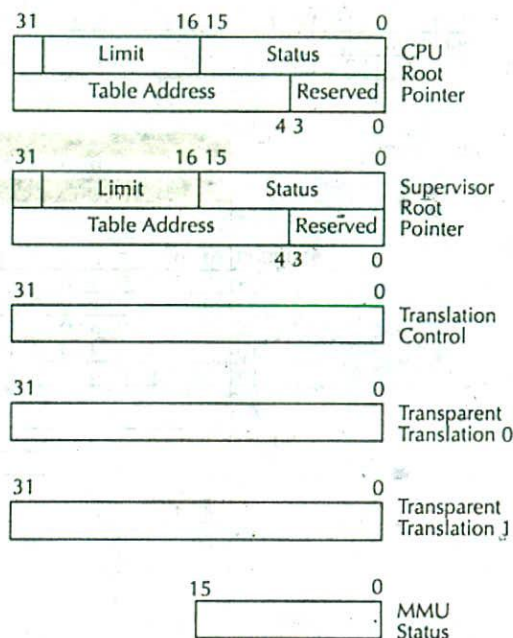


FIGURE 7.16 MC68030 MMU registers.

The 68030 includes three main elements in its MMU: a set of registers, an ATC, and table search logic.

Figure 7.16 shows the MMU registers.

The CRP or SRP points to the beginning of the task translation table and supervisor translation table (if enabled), respectively.

The translation control register controls the MMU functions such as MMU enable.

The transparent translation registers output specified logical addresses on the A31-A0 pins.

The MMUSR provides the results of execution of the MMU instructions PMOVE (EA), MMUSR, and PTEST. The MMUSR stores memory management status information resulting from a search of the address translation cache, or the translation tree, for a particular logical address.

As mentioned before, four MMU instructions — PMOVE, PTEST, PLOAD, and PFLUSH — are included in the 68030 instruction set. The MMU provides up to 5 levels of address translation tables. Address translation starts with the contents of the root pointers CRP or SRP.

Figure 7.17 provides a typical 5-level table translation scheme.

Figure 7.18 provides the ATC block diagram along with a simplified flowchart for the physical address translation.

The ATC is a content-addressable, fully associative cache with up to 22 descriptors. The ATC stores recently used descriptors so that a table search is not required if future accesses are required.

There are six types of descriptors used in MMU operation. These are ATC page descriptors, table descriptors, early termination page descriptors, invalid descriptors, and indirect descriptors. Some descriptors have a long and a short form.

Figure 7.19 shows the page descriptor summary.

Each ATC entry consists of a logical address and information from a corresponding page descriptor. The 28-bit logical or tag portion of each entry consists of three fields. These are the

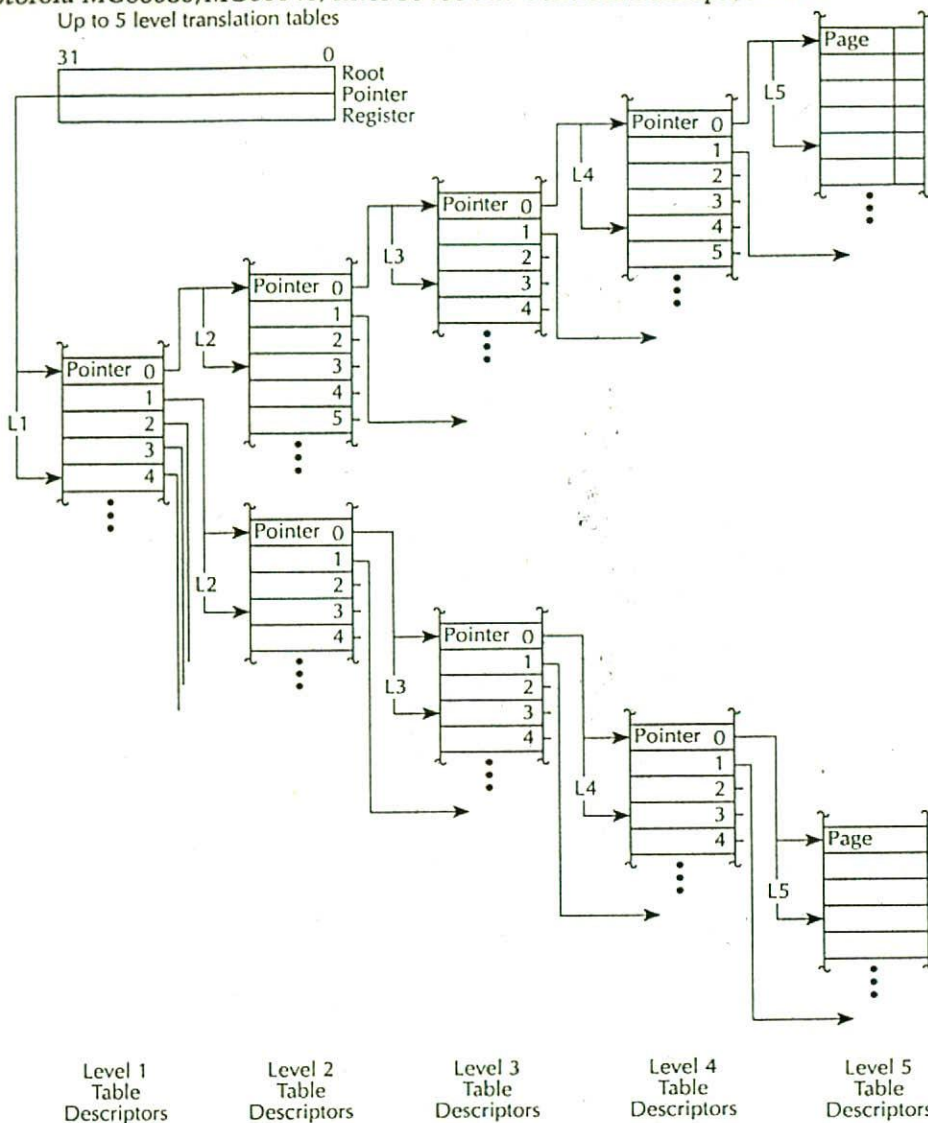


FIGURE 7.17 MC68030 MMU five-level translation scheme.

valid bit field (bit 27), function code field (bits 26-24), and 24-bit logical address field (bits 0-23). The V-bit indicates that the entry is valid if $V = 1$. The function code field includes the function code bits (FC0-FC2) corresponding to the logical address in this entry. The 24-bit logical address includes the most significant logical address bits for this entry. All 24 bits are used in comparing this entry to an incoming logical address when the page size is 256 bytes. For larger page sizes, some least significant bits of this field are ignored.

Table descriptors have short (32-bit) and long (64-bit) forms. In the 32-bit short form, the table descriptor includes four status bits (bits 0-3) and a 28-bit table address. The status bit provides information such as write protection and descriptor type.

The table address contains the 28-bit physical base address of a table of descriptors. The long table descriptor includes a 28-bit table address, a 16-bit status, and a 16-bit limit field. The status bits in the long form include additional information such as whether the table is a

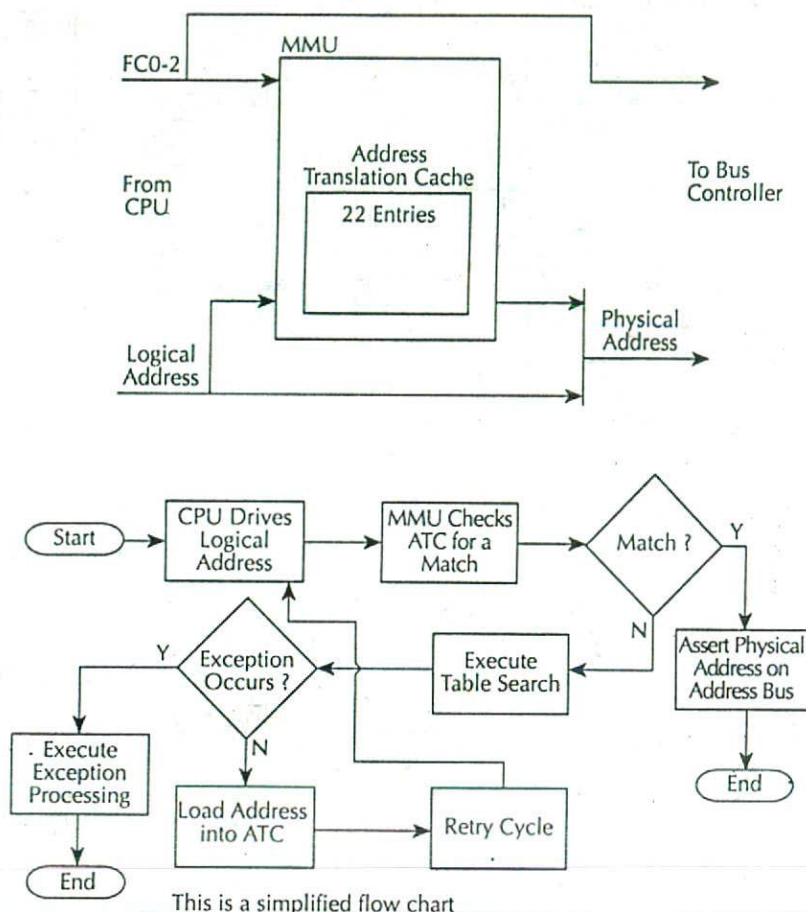


FIGURE 7.18 Address translation cache (ATC) and flowchart for translation.

supervisor-only table. The limit field includes a limit to which the index portion of an address is compared to detect an out-of-bounds index.

The page descriptors also have short and long formats. The short page format is identical to the short table format except that for page tables, page table and status information related to pages are used. The long page descriptor contains a 24-bit page address and a 16-bit status. The status bits provide information such as write protect, descriptor type, and identification of a modified page.

The early termination page descriptor (both short and long form) includes the page descriptor identification bits in the descriptor-type field of the status bits, but the descriptor resides in a pointer table. This means that the table in which an early termination page descriptor is located is not at the bottom level of the address translation tree. The invalid descriptors can also be short and long forms. These descriptors only contain the two descriptor-type status bits and are used with long or short page and table descriptors.

The indirect descriptors also include short and long forms. They contain the physical address of a page descriptor and two descriptor-type bits which identify an indirect descriptor that points to a short or long format page descriptor.

Table 7.3 summarizes differences between 68030 on-chip MMU and 68851.

Now let us discuss the details of the 68030 MMU registers. The CRP points to the first level translation table. It is normally loaded during a context (task) switch. The ATC is automatically flushed when the CRP is loaded. Figure 7.20 shows the details of the CRP.

Address Translation Cache Page Descriptor

27	23	0	27	23	0
V	FC	Logical Address	Status	Physical Address	

Table Descriptors

31																4 3				0				Short							
Table Address																Status				Long											
31																16 15				0 31				4 3				0			
Limit																Status				Table Address								Not Used			

Page Descriptors

31	8	7	0				
Page Address			Status				
				Short			
				Long			
31	16	15	0	31	8	7	0
Unused		Status		Page Address		Unused	

- Early Termination Page Descriptors
- Invalid Descriptors
- Indirect Descriptors

FIGURE 7.19 Page descriptor summary.

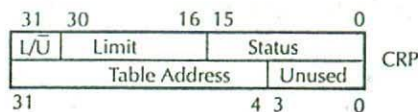
The limit field in the CRP limits the size of the next table. That is, it limits the size of the table index field. If the limit is exceeded during a table search, an ATC entry is created with bus error set. Two fields are assigned to the limit. These are L/U and limit. L/U = 1 means a lower range limit, while L/U = 0 indicates an upper range limit. If L/U = 0 and limit = \$7FFF or L/U = 1 and limit = 0, the limit function is suppressed. The translation control register (TC) permits the user to control the translation of the access. TC is also used to enable and disable the main function of the MMU. Figure 7.21 shows the details of TC.

The SRP is similar to the CRP except that it is used only for supervisor access. This register must be enabled in the TC before use. The format of SRP is shown in Figure 7.22.

TABLE 7.3 68030 On-Chip MMU vs. 68851

Features of the 68851 not included on the 68030 MMU:

- No access level (no CALLM or RTM instructions)
- No breakpoint registers
- No DMA root pointer
- No task aliasing
- 22 entry ATC instead of 64
- No lockable content-addressable memory (CAM) entries
- No shared globally entries
- Instructions supported
 - PFLUSHA, PFLUSH, PMOVE, PTEST, PLOAD
- Instructions not supported (F-line trap)
 - PVALID, PFLUSHR, PFLUSHES, PBcc, PDBcc, PSec,
 - PTRAPcc, PSAVE, PRESTORE
- Control alterable effective addresses only for the MMU instructions



• Status Field



• DT Values

- 00 Not allowed. If loaded with this value, an MMU Configuration Error exception occurs.
- 01 Table address field is a page descriptor.
- 10 Table address field points to descriptors which are 4 bytes long.
- 11 Table address field points to descriptors which are 8 bytes long.

FIGURE 7.20 CRP details.

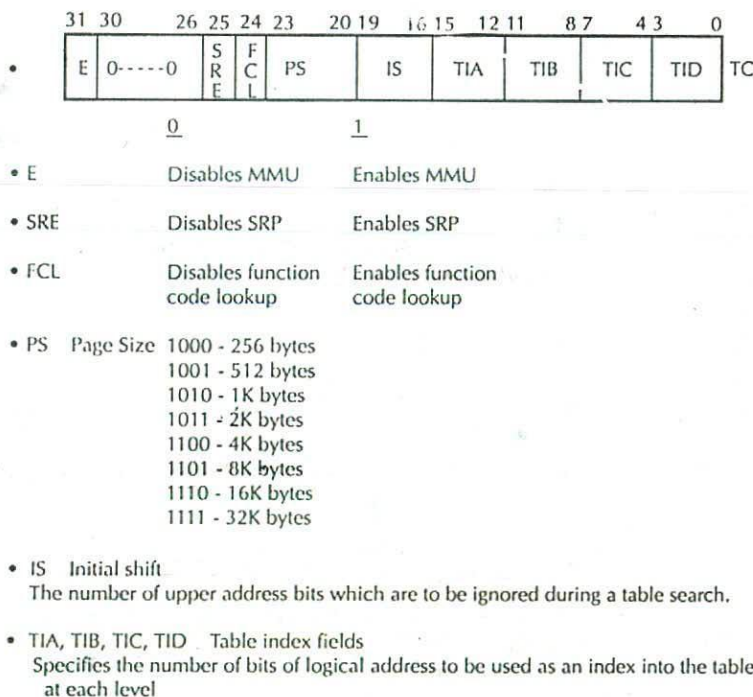


FIGURE 7.21 TC details.

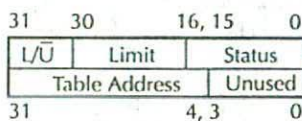


FIGURE 7.22 SRP form.

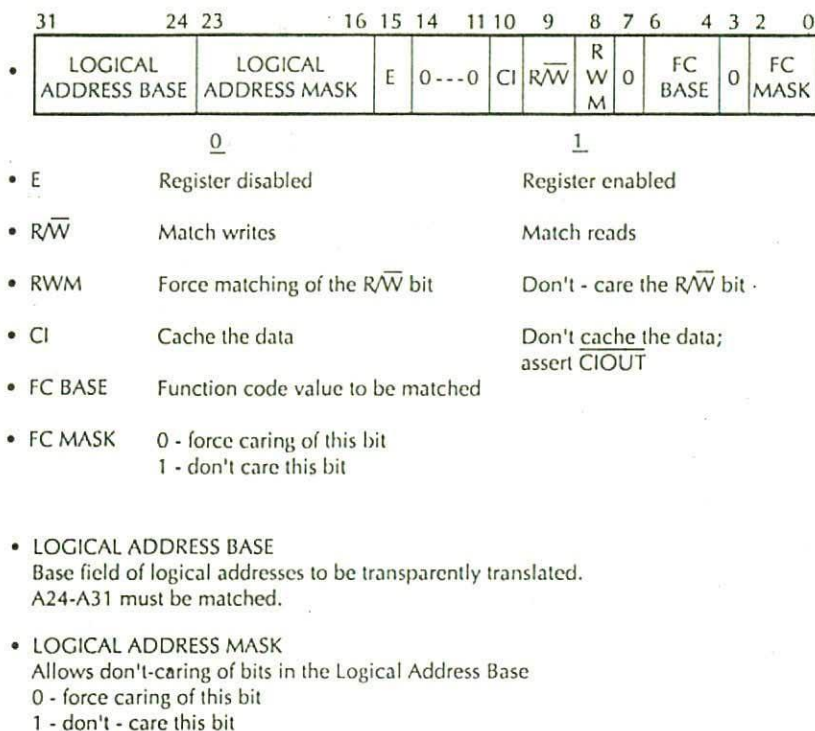


FIGURE 7.23 TT0 and TT1 details.

If a task is required to access physical memory directly, such as in graphics applications TT0 or TT1 can be used to specify the physical areas of memory to be accessed.

Figure 7.23 provides formats for TT0 and TT1.

The MMUSR includes the results of execution of the PTEST instruction for level = 0 or level $\neq 0$.

Figure 7.24 provides the MMUSR details.

The ATC entries include two (logical and physical) 28-bit fields. The logical field includes a valid bit, function codes, and page number. The physical field contains four control/status bits and page frame number. Figure 7.25 includes the ATC entry structure.

The translation tables supported by the 68030 contain a tree structure. The root of a translation table tree is pointed to by one or two root pointer registers.

Table entries at higher levels of the tree contain pointers to other tables, while entries at the leaf level (page tables) contain page descriptors. The technique used for table searches utilizes portions of the logical address as an index for each level of the lookup. All addresses contained in the translation table entries are physical addresses.

Figure 7.26 shows the 68030 MMU translation table tree structure.

The function codes are usually used as an index in the first level of lookup in the table. However, this may be suppressed. In table searching, up to 15 of the logical address lines can be ignored. The number of levels in the table indexed by the logical address can be set from one to four and up to 15 logical address bits can be used as an index at each level. One main advantage of this tree structure is to deallocate large portions of the logical address space with a single entry at the higher levels of the tree.

The entries in the translation tables include status information with respect to the pointer for the next level of lookup or the pages themselves. These bits can be used to designate certain pages or blocks of pages as supervisor-only, write-protected, or noncacheable. The 68030 MMU exceptions include the following:

- The information in the MMU status register is the result of execution of the PTEST instruction.

- Level = 0 Search the ATC only
- Level \neq 0 Search translation tables only

15	14	13	12	11	10	9	8	7	6	5	4	3	0
B	L	S	0	W	I	M	0	0	T	0	0	0	N

Bit	Meaning	Level = 0	Level \neq 0
B	Bus error	Bus error is set in Matching ATC entry	Bus error occurred during table walk
L	Limit bit	Always cleared	An index exceeded the limit
S	Supervisor violation	Always cleared	Supervisor bit is set in descriptor
W	Write protect	Address is write protected	Address is write protected
I	Invalid bit	No entry in ATC or B is set	No translation in table, or B or L is set
M	Modified bit	ATC entry has M bit set	Page descriptor has M Bit set
T	Transparent	Address is within range of TT0 or TT1	Always cleared
N	Number of tables	0	Number of tables used in translation

FIGURE 7.24 MMUSR details.

27	26	24	23	0	27	26	25	24	23	0
V	FC	LOGICAL ADDRESS			B	CI	WP	M	PHYSICAL ADDRESS	

Bit	Meaning	0	1
V	Valid	Invalid	Valid
B	Bus error	No bus error	Error occurred when entry was formed. If this logical address occurs, a bus error exception will occur.
CI	Cache inhibit	$\overline{\text{CIOUT}}$ is negated	$\overline{\text{CIOUT}}$ is asserted disabling internal and external cache.
WP	Write protect	Page is not write protected	Page is write protected. If a write is attempted, a bus error exception occurs.
M	Modified	Page has not been modified	Page has been modified.

- FC Function Code

FIGURE 7.25 ATC entry structure.

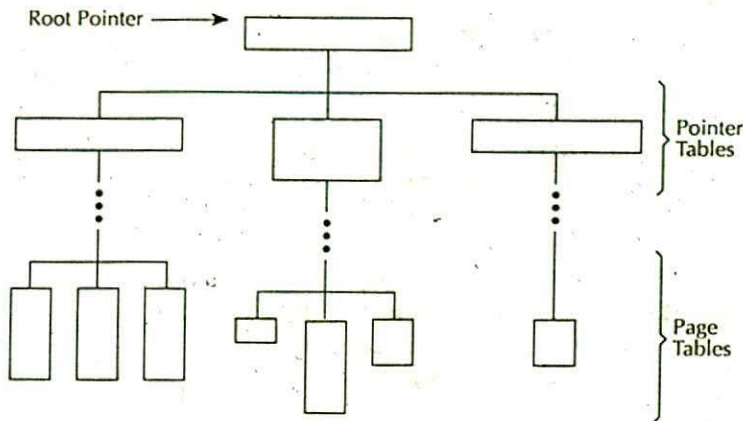


FIGURE 7.26 MC68030 MMU translation table tree structure.

- Bus error
- F-line
- Privilege violation
- Configuration error

Example 7.1

Assume user program space. The MC68030 executes CLR.W(A1) with PC contents and instruction cache contents as follows.

Given the op code for CLR.W(A1) is 4251_{16} , $[PC] = \$02513080$.

<u>Instruction Cache Contents</u>										
Function Codes = 010 ₂			valid bits							
Index			0	1	2	3	LW0	LW1	LW2	LW3
7	2	025130	1	1	1	1	4251xxxx	—	—	—
8	2	025130	0	1	1	1	4251xxxx	—	—	—
9	2	025130	1	1	1	1	—	—	—	—

When the instruction CLR.W(A1) is fetched by the 68030, will a cache hit or miss occur? Why?

Solution

A miss will occur since the valid bit is zero.

Example 7.2

Write an instruction sequence to freeze the data cache and disable the instruction cache.

Solution

```

MOVEC CACR,D1    ; Get current
                  ; CACR
BCLR.L #0,D1     ; Disable
                  ; instruction
                  ; cache
BSET.L #9,D1     ; Freeze data cache
MOVEC D1, CACR   ; Write to
                  ; CACR

```

Example 7.3

Assume a page size of 4K bytes. Determine the physical address from the logical addresses of the task shown below:

Page			
0	\$002000	12345	Page
1	\$002004	7AB12	Descriptor
2	\$002008	1BCA7	Table

Logical addresses to be translated are \$000001A5, \$00002370.

Solution

From logical addresses \$000001A5, the page number is upper 20 bits, i.e., page number is zero. This is because the page size is given as 4K bytes and hence the lower 12 bits ($2^{12} = 4K$) are used as the page size. Since TCB is \$002000, \$12345 is concatenated with the lower 12 bits (\$1A5) of the logical address \$000001A5 to obtain the physical address \$123451A5.

Similarly, the physical address for the logical address \$00002370 is \$1BCA7370.

Example 7.4

Determine the contents of SRP when enabled in TC to describe a page descriptor table located at \$05721050 and limited to logical address 0 – \$7000.

Solution

The format for SRP is

31	30	16, 15	2, 1	0
L / U	Limit	0----	DT	
Table Address		Unused		
31	3	0		

L / U is 0 for upper limit range. LIMIT is \$7000. DT must be 01 for page descriptor. Table address is \$05721050. Hence, SRP is

31	0	
0	7000	0001
05721050		

Example 7.5

What happens upon execution of the PTEST instruction with level 0 (i.e., search translation tables only)? Assume that the MMUSR contains \$4005.

Solution

From Figure 7.24, level 0 means search translation tables only. Bit 14 defines the limit bit and bits 0–2 define the number of tables used in translation. [MMSUR] = \$4005 indicates limit bit = 1 and number of tables used in translation is 5. Therefore, the index limit is exceeded when searching the five translation tables.

7.2 MC68040

This section presents an overview of the Motorola 68040 32-bit microprocessor. Emphasis is given to the coverage of its on-chip floating-point hardware.

7.2.1 Introduction

The MC68040 is Motorola's enhanced 68030, 32-bit microprocessor. The MC68040 is implemented in Motorola's latest HCMOS technology. Providing an ideal balance between speed, power, and physical device size, the MC68040 integrates an MC68030-compatible integer unit, an MC68881/MC68882-compatible floating-point unit (FPU), dual independent demand-paged memory management units (MMUs) for instruction and data stream accesses, and independent 4K-byte instruction and data cache. A high degree of instruction execution parallelism is achieved through the use of multiple independent execution pipelines, multiple internal buses, and separate physical caches for both instruction and data accesses.

The main on-chip features of the MC68040 include:

- MC68030-Compatible Integer Execution Unit
- MC68881/MC68882-Compatible Floating-Point Execution Unit
- Independent Instruction and Data Memory Management Units (MMUs)
- 4K-Byte Physical Instruction Cache and 4K-Byte Physical Data Cache Accessible Simultaneously
- Concurrent integer unit, FPU, MMU, and Bus Controller Operation which maximize throughput
- 32-Bit, nonmultiplexed external address and data buses with synchronous interface

The Intel equivalent of the 68040 is the 80486. Table 7.4 depicts a comparison of the main features and capabilities of the two processors. The most significant difference between the two devices is throughput. The 68040 executes more integer and floating point operations per second (at a clock rate of 25Mhz) than the 486. The primary reasons for the 040's outstanding performance are its pipelined integer and floating point units, as well as its advanced memory management implementation. Both these features are discussed in detail later.

7.2.2 Register Architecture/Addressing Modes

Figure 7.27 shows the MC68040 programming model. The registers are partitioned into two levels of privilege: user and supervisor. The user model has sixteen 32-bit general-purpose registers (D0-D7 and A0-A7), a 32-bit program counter (PC), and a condition code register (CCR) contained within the supervisor status register (SR). The MC68040 user programming model also incorporates the MC68882 programming model consisting of eight 80-bit floating-point data registers, a floating-point control register, a floating-point status register, and a

TABLE 7.4 Main features of Motorola MC68040 vs. Intel 80486

Chip	ALU/Data Bus	Clock Rate	Cache	FPU	Integer Throughput
68040	32/32 Bit	25-50 MHz	On-Chip	On-Chip	22 MIPS
80486	32/32 Bit	25-66 MHz	On-Chip	On-chip	15 MIPS

floating-point instruction address register. The supervisor model has two 32-bit stack pointers (ISP and MSP), a 16-bit status register (SR), a 32-bit vector base register (VBR), two 3-bit alternate function code registers (SFC and DFC), a 32-bit cache control register, a supervisor root pointer (SRP) and user root pointer (URP), a translation control register (TCR), four transparent translation registers: two for instruction accesses (ITT1-ITT0), and two for data access (DTT1-DTT0), and an MMU status register (MMUSR).

The key feature distinguishing the 68040 from its predecessors is its on-chip Floating Point Unit (FPU). The 68040 has 11 on-chip registers dedicated to support floating point operations: eight 80 bit data registers, and one each 32 bit floating point control (FPCR), status (FPSR), and instruction address (FPIAR) register.

The eight floating point registers always contain extended precision numbers. All external operands are converted to extended precision prior to use in any calculation, or storage in any FP register. The FPCR consists of one exception enable byte for traps during exception operations, and one mode byte for setting user-defined rounding modes. The FPCR may be modified and read by the user and is cleared by a hardware reset or a restore null state operation. The remaining 16-bits in the FPCR are reserved by Motorola for future definition.

The FPSR contains several status byte indicating status for the last calculation. The FPSR Condition Code byte indicates the results of an operation were negative, zero, infinity, or not a number (NaN). The Exception Status byte indicates floating point operational exceptions such as Branch/Set on Unordered (arithmetic operation attempted using NaN), Signaling NaN, and Operand error. The FPSR Accrued Exception byte contains a history of exceptions

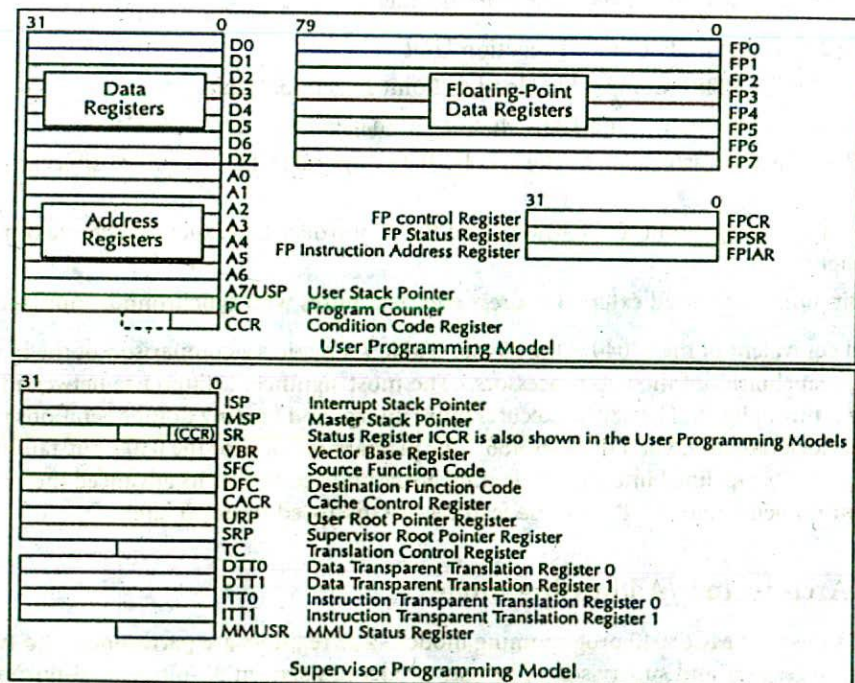


FIGURE 7.27 68040 programming model.

generated by previous operations. The FPSR Quotient byte is included for compatibility with the MC68881/2 floating point units, and contains the least significant 7 bits and the sign bit of the quotient from the previous operation.

The 68040 FPU operates simultaneously with the Integer Unit. In addition to this, the pipelined FPU can concurrently execute two instructions. This pipelined superscalar architecture increases the complexity of discovering which of several instructions executing concurrently may have generated an exception. The FPIAR holds the logical address of the instruction prior to its execution, to allow a floating point exception handler to locate the possible cause of an exception.

The FPU supports the same data as the 68881/68882. The MC68040 supports the same addressing mode as the 68020/68030.

7.2.3 Instruction Set/Data Types

The instructions provided by the MC68040 are listed in Table 7.5.

TABLE 7.5 68040 Instruction Set Summary

Mnemonic	Description
ABCD	Add Decimal with Extend
ADD	Add
ADDA	Add Address
ADDI	Add Immediate
ADDQ	Add Quick
ADDX	Add with Extend
AND	Logical AND
ANDI	Logical AND Immediate
ASL, ASR	Arithmetic Shift Left and Right
Bcc	Branch Conditionally
BCHG	Test Bit and Change
BCLR	Test Bit and Clear
BFCHG	Test Bit Field and Change
BFCLR	Test Bit Field and Clear
BFEXTS	Signed Bit Field Extract
BFEXTU	Unsigned Bit Field Extract
BFFFO	Bit Field Fine First One
BFINS	Bit Field Insert
BFSET	Test Bit Field and Set
BFTST	Test Bit Field
BRA	Branch
BSET	Test Bit and Set
BSR	Branch to Subroutine
BTST	Test Bit
CAS	Compare and Swap Operands
CAS2	Compare and Swap Dual Operands
CHK	Check Register Against Bounds
CHK2	Check Register Against Upper and Lower Bounds
CINV*	Invalidate Cache Entries
CLR	Clear
CMP	Compare
CMPA	Compare Address
CMPI	Compare Immediate
CMPM	Compare Memory to Memory
CMP2	Compare Register Against Upper and Lower Bounds
CPUSH*	Push then Invalidate Cache Entries
DBcc	Test Condition, Decrement and Branch
DIVS, DIVSL	Signed Divide

TABLE 7.5 68040 Instruction Set Summary (continued)

Mnemonic	Description
DIVU, DIVUL	Unsigned Divide
EOR	Logical Exclusive OR
EORI	Logical Exclusive OR Immediate
EXG	Exchange Registers
EXT, EXTB	Sign Extend
FABS*	Floating-Point Absolute Value
FADD*	Floating-Point Add
FBcc	Floating-Point Branch
FCMP	Floating-Point Compare
FDBcc	Floating-Point Decrement and Branch
FDIV*	Floating-Point Divide
FMOVE*	Move Floating-Point Register
FMOVM	Move Multiple Floating-Point Registers
FMUL*	Floating-Point Multiply
FNEG*	Floating-Point Negate
FRESTORE	Restore Floating-Point Internal State
FSAVE	Save Floating-Point Internal State
FScc	Floating-Point Set According to Condition
FSORT*	Floating-Point Square Root
FSUB*	Floating-Point Subtract
FTRAPcc	Floating-Point Trap-On Condition
FTST	Floating-Point Test
ILLEGAL	Take Illegal Instruction Trap
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link and Allocate
LSL, LSR	Logical Shift Left and Right
MOVE	Move
MOVE16*	16-Byte Block Move
MOVEA	Move Address
MOVE CCR	Move Condition Code Register
MOVE SR	Move Status Register
MOVE USP	Move User Stack Pointer
MOVEC*	Move Control Register
MOVEM	Move Multiple Registers
MOVEP	Move Peripheral
MOVEQ	Move Quick
MOVES*	Move Alternate Address Space
MULS	Signed Multiply
MULU	Unsigned Multiply
NBCD	Negate Decimal with Extend
NEG	Negate
NEGX	Negate with Extend
NOP	No Operation
NOT	Logical Complement
OR	Logical Inclusive OR
ORI	Logical Inclusive OR Immediate
PACK	Pack BCD
PEA	Push Effective Address
PFLUSH*	Flush Entry(ies) in the ATCs
PTEST*	Test a Logical Address
RESET	Reset External Devices
ROL, ROR	Rotate Left and Right
ROXL, RORX	Rotate with Extend Left and Right
RTD	Return and Deallocate
RTE	Return from Exception
RTR	Return and Restore Codes

TABLE 7.5 68040 Instruction Set Summary (continued)

Mnemonic	Description
RTS	Return from Subroutine
SBCD	Subtract Decimal with Extend
Sec	Set Conditionally
STOP	Stop
SUB	Subtract
SUBA	Subtract Address
SUBI	Subtract Immediate
SUBQ	Subtract Quick
SUBX	Subtract with Extend
SWAP	Swap Register Words
TAS	Test Operand and Set
TRAP	Trap
TRAPcc	Trap Conditionally
TRAPV	Trap on Overflow
TST	Test Operand
UNLK	Unlink
UNPK	Unpack BCD

* MC68040 additions or alterations to the MC68030 and MC68881/MC68882 instruction set.

Instructions in Table 7.5 which have an asterisk are unique to the 68040. The ptest and pflush MMU control instructions have new formats, but are similar to those of the 68030. The movel6 instruction allows for faster block transfers. Since this instruction operates only between 16 byte memory locations, and most compiler data types are 8 bytes or less in size, this instruction is not useful for most operations. Larger data structures such as arrays may benefit from use of the movel6 instruction, but only if they are aligned on a 16-byte boundary. The 68040 Translation Control Register is accessed with a movec instruction as opposed to pmove instruction on the 68030. On the 68040, the bsr and bra instructions are both one clock cycle faster than jsr and jmp respectively. Since each pair is usually interchangeable, the branch instructions are preferred.

Though the 68040 possesses an on-chip FPU, only a subset of the 68882 floating point coprocessor instructions are directly supported. The remaining functions such as transcendental (trigonometric and exponential) are emulated in software via a trap using Motorola's Floating-Point software package (FPSP) for the 68040. The FPSP emulates the floating-point instructions of the 68881/68882 which are not provided by the 68040. This is an illegal instruction mechanism used to indicate instructions not recognized by the hardware. The software emulator is also invoked by the underflow exception and a new unsupported-data type exception that handles denormalized and packed-decimal data representations in 68040. Motorola claims even these instructions will execute 25–100% faster on a 25 MHz 68040 than on a 33MHz 68882 FPU. One way to avoid using emulated floating point instructions is to call a library routine linked to the user program to do the FP operations. This approach is faster than emulation since the trap and decode times associated with emulation (which are comparable to the actual computing time) are eliminated. Also, emulation routines carry calculations to extended (80 bit) precision. This always adds extra calculation iterations, and thus extra time, but is not always needed by the application. Table 7.6 depicts the indirectly supported FP instructions. Note that loading of constant π is not directly supported by the 68040. Constant π can be indirectly loaded by using the MOVECR instruction (MOVECR·X#0, FpN). See table on page 413.

The MC68040, with its integer unit and floating-point unit (FPU), support the operand data types shown in Table 7.7. The operand types supported by the integer unit include the data types supported by the MC68030 plus a new data type (16-byte block) for the MOVE16 instruction. The user instruction MOVE16 has been added to the instruction set to support 16 byte memory to memory data transfers. Integer unit operands can reside in registers, in

TABLE 7.6 Indirectly Supported Floating-Point Instructions by the 68040

Mnemonic	Description
FACOS	Floating-Point Arc Cosine
FASIN	Floating-Point Arc Sine
FATAN	Floating-Point Arc Tangent
FATANH	Floating-Point Hyperbolic Arc Tangent
FCOS	Floating-Point Cosine
FCOSH	Floating-Point Hyperbolic Cosine
FETOX	Floating-Point e^x
FETOXL	Floating-Point $e^x - 1$
FGETEXP	Floating-Point Get Exponent
FGETMAN	Floating-Point Get Mantissa
FINT	Floating-Point Integer Part
FINTRZ	Floating-Point Integer Part, Round-to-Zero
FLOG10	Floating-Point \log_{10}
FLOG10	Floating-Point \log_2
FLOGN	Floating-Point \log_e
FLOGNP1	Floating-Point $\log_e (x + 1)$
FSQRT	Floating-Point Square Root
FMOD	Floating-Point Modulo Remainder
FMOVECR	Floating-Point Move Constant ROM
FREM	Floating-Point IEEE Remainder
FSCALE	Floating-Point Scale Exponent
FSGLDIV	Floating-Point Single Precision Divide
FSFLMUL	Floating-Point Single Precision Multiply
FSIN	Floating-Point Sine
FSINCOS	Floating-Point Simultaneous Sine and Cosine
FSINH	Floating-Point Hyperbolic Sine
FTAN	Floating-Point Tangent
FTANH	Floating-Point Hyperbolic Tangent
FTENTOX	Floating-Point 10^x
FTWOTOX	Floating Point 2^x

memory, or within the instructions themselves, and may be a single bit, a bit field, a byte, a word, a long word, a quad word, or a 16-byte block. Each integer data register is 32 bits wide. Byte operands occupy the lower 8 bits, word operands the lower order 16 bits, and long-word operands the entire 32 bits. The LSB of a long-word integer is addressed as bit zero and the MSB is addressed as bit 31. For bit fields, the MSB is addressed as bit zero, and the LSB is addressed as the width of the field minus one.

TABLE 7.7 68040 Data Types

Operand Data Type	Size	Supported By:	Notes
Bit	1 Bit	IU	—
Bit Field	1-32 Bits	IU	Field of Consecutive Bit
BCD	8 Bits	IU	Packed: 2 Digits Byte Unpacked: 1 Digit Byte
Byte Integer	8 Bits	IU, FPU	—
Word Integer	16 Bits	IU, FPU	—
Long-Word Integer	32 Bits	IU, FPU	—
Quad-Word Integer	64 Bits	IU	Any Two Data Registers
16-Byte	128 Bits	IU	Memory-Only, Aligned to 16-Byte Boundary
Single-Precision Real	32 Bits	FPU	1-Bit Sign, 8-Bit Exponent, 23-Bit Mantissa
Double-Precision Real	64 bits	FPU	1-Bit Sign, 11-Bit Exponent, 52-Bit Mantissa
Extended-Precision Real	80 Bits	FPU	1-Bit Sign, 15-Bit Exponent, 64-Bit Mantissa

Example 7.6

Write a 68040 assembly language program to compute the volume of a sphere to extended precision by using $V = 4/3 * \pi r^3$ where r is the radius of the sphere.

Solution

```

ORG $1000      ; initialize program at location hex
                1000
FMOVE (A7),fp0 ; move radius value from user stack
                to fp register 0
FMOVE (A7),fp1 ; and into fp register 1
FMUL fp1,fp1   ; obtain r squared, store in fp1
FMUL fp0,fp1   ; obtain r cubed, store in fp1
FMOVE #4,fp2   ; load decimal constant 4 into fp
                register 2
FDIV #3,fp2    ; obtain 4/3, store in fp2
FMOVE #22,fp4  ; load decimal constant 22 into fp
                register 4
FDIV #7,fp4    ; obtain  $\pi$  store in fp4
FMUL fp2,fp4   ; obtain  $(4 * \pi)/3$ , store in fp4
FMUL fp4,fp1   ; obtain  $((4 * \pi)/3) * r$  cubed, store
                in fp1
FINISH JMP FINISH ; halt

```

This program will leave the extended precision results of the routine in floating point register #1.

Example 7.7

Write a 68040 assembly language program to compute the hypotenuse of a right angle triangle by using

$$Z = \sqrt{x^2 + y^2}$$

Solution

The following program requires software emulation of the indirectly supported fsqrt floating point instruction to perform a Pythagorean theorem calculation of the length of the hypotenuse of a right angle triangle:

```

ORG $1000      ; initialize at hex location 1000
FMOVE (A7)+,fp0 ; load adjacent side length into fp
                reg 0
FMUL fp0,fp0   ; square side 1, store in fp0
FMUL fp2,fp2   ; square side 2, store in fp2
FADD fp0,fp2   ; sum the squared side lengths and
                store in fp reg 2
FSQRT fp2      ; calculate length of hypotenuse
                and store in fp 2
FINISH JMP FINISH ; halt

```

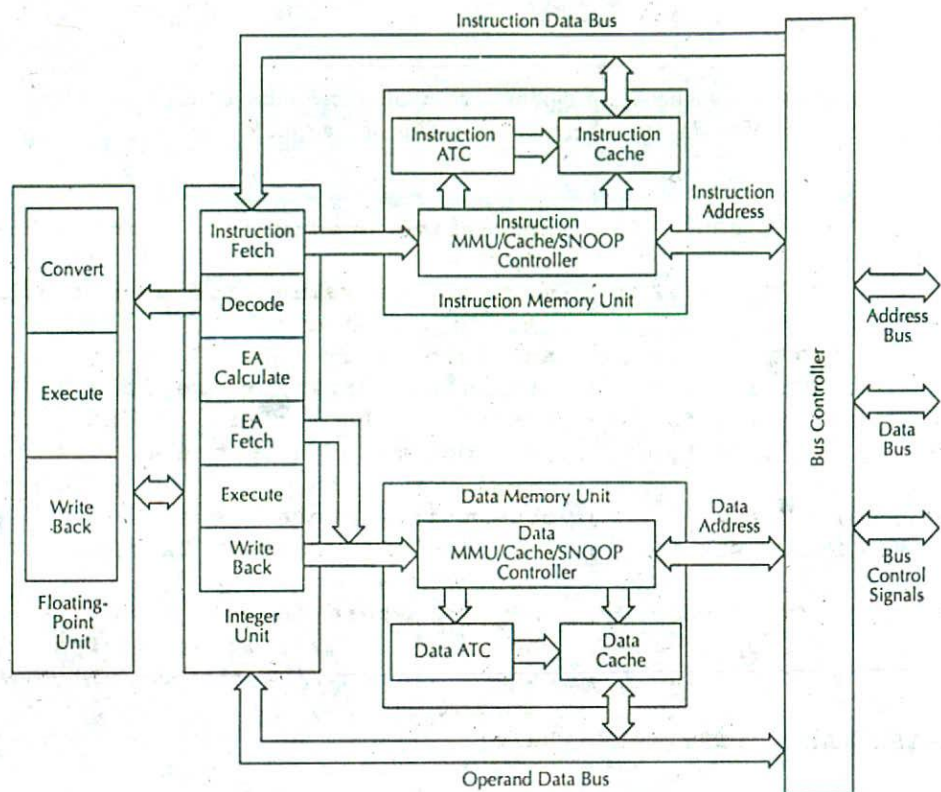



FIGURE 7.28 MC68040 block diagram.

7.2.4 68040 Processor Block Diagram

Figure 7.28 shows a block diagram of the MC68040. Instruction execution is pipelined in both the Integer Unit (IU) and FPU, which interface to fully independent data and instruction memory units. In the MC68040, high-usage instructions (such as branch instructions) execute in a minimum number of clock cycles. In the development of the IU, Motorola ran thousands of real world code traces to determine which instructions were used most often. The integer pipeline consists of six stages: Instruction prefetch, Instruction decode, Effective address calculate, Effective address fetch, Execute and Writeback. The six-stage IU pipeline executes instructions at an average of 1.3 instructions/clock cycle: about 3 times faster than the 68030 IU. This approaches the one instruction/clock cycle execution rate of a RISC architecture; an outstanding accomplishment for a CISC architecture processor. Figure 7.29 depicts the instruction pipeline of the Integer Unit.

The Floating Point Unit (FPU) pipeline consists of three stages; Operand Conversion, Execute, and Result Normalization. A floating point instruction will flow-through the IU until it reaches the execute stage. Here, the instruction and its operands are transferred to the operand conversion stage of the FPU. If the instruction requires data to be transferred back to the integer unit, then the IU execute stage must wait for the data. This may stall the IU pipeline by 10's of clock cycles. If not, the IU continues instruction execution in parallel with the FPU. The compiler may minimize these pipeline stalls by reordering integer and floating point instructions to minimize register and memory location dependencies between concurrently executing instructions. While the FPU pipeline is slower than the IU pipe, the on-chip FPU operations in the 68040 take about one half the number of clock cycles of the 68882 coprocessor which is required for floating point operations with the 680x0 series prior to the 68040.

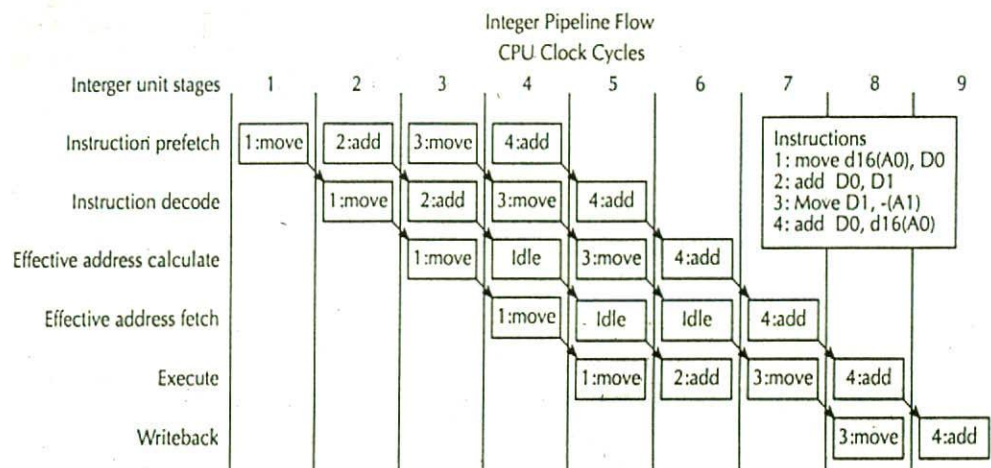


FIGURE 7.29 Integer unit instruction pipeline.

The instruction and data memory units operate independently. They consist of an MMU which utilizes the ATC to translate the logic address generated by the MC68040 into a physical address, while the bus snoop circuit ensures cache coherency in multimaster applications. This split cache or “harvard architecture” uses a subset of the 68030’s MMU instruction set.

7.2.5 68040 Memory Management

The memory management function performed by the MMU is called demand paged memory management. Since a task specifies the areas of memory it requires as it executes, memory allocation is supported on a demand basis. If a requested area of memory is not currently mapped by the system, then the access causes a demand for the operating system to load to allocate the required memory image. The technique used by the MC68040 is paged memory management because physical memory is managed in blocks of a specified number of bytes, called page frames. The logical address space is divided into fixed-size pages that contain the same number of bytes as the page frames. The memory management software assigns a physical base address to a logical page. The system software then transfers data between secondary storage and memory, one or more pages at a time. Because of the phenomenon of locality of reference, instruction and data that are used in a program have a high probability of being reused within a short time. Additionally, instruction and data operands residing near the instruction and data currently in use also have a high probability of being utilized within a short period.

Memory management in the MC68040 has been improved by including separate independent paged MMUs for instruction and data accesses. The data and instruction MMUs support virtual memory systems by translating logical addresses to physical addresses using translation tables stored in memory. Each MMU contains an address translation cache (ATC) in which recently used logical-to-physical address translations are stored. For normal accesses, the translation process proceeds as follows for the accessed instruction or data memory unit:

1. Compare the logical address and privilege mode to the parameters in the transparent translation registers and use the logical address as a physical address for the access if one of the transparent translation registers match. Each transparent translation register can define a block of logical addresses that are used as physical addresses without translation.

2. Compare the logical address and privilege mode to the tag portions of the entries in the ATC. When the access address and privilege mode matches a tag in the ATC and no access violation is detected, the ATC outputs the corresponding physical address to the cache controller, which access the data within the cache.
3. When no transparent translation register nor valid ATC entry matches, the MMU aborts the current access and searches the translation tables in memory for the correct translation. If the table search completes without errors, the MMU stores the translation in the ATC and provides the physical address for the access, allowing the memory unit to retry the original access by repeating step 2.

Separate 4K-byte on chip instruction and data cache operate independently, and are accessed in parallel with address translation. Each cache and corresponding MMU reside on a separate internal address bus and data bus, allowing simultaneous access to both. Both four-way set associative caches have 64 sets of four, 16-byte lines. Each cache line contains an address tag, status information, and four long words of data (D0-D3). The address tag contains the upper 22-bits of the physical address. The processor fills the cache lines using burst mode access which transfers the entire line as four long words. This mode of operation not only fills the cache efficiently, but also captures adjacent instruction or data items that are likely to be required in the near future due to locality characteristics of the executing task.

Each memory unit access by the integer unit is translated from a logical address to a physical address to access the data in the cache. To minimize latency of the requested data, the lower untranslated bits of the logical address (which map directly to physical address bits) are used to access a set of cache lines in parallel with the translation of the upper logical address bits. Bits PA9-PA4 are used to index into the cache and select one of the 64 sets of four lines. The four tags from the selected cache set are compared to the translated physical address bits PA31-PA12 from the MMU and bits PA11 and PA10 of the untranslated page offset. The cache has a hit, if any one of the four tags match.

The caches improve the overall performance of the system by reducing the number of bus transfers required by the processor to fetch information from memory and by increasing the bus bandwidth available for other bus masters in the system. To further improve system performance the data cache in the MC68040 supports both write back and write through caching modes for storing write accesses.

Write back — The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.

Write through — The information is written to both the block in the cache and to the block in the lower-level memory.

The MC68040 implements a bus snoop that maintains coherency of the caches by monitoring the accesses by an alternate bus master and performing cache maintenance operations as requested by the alternate master. Matching cache lines can be invalidated during the alternate master access to memory, or memory can be inhibited to allow the MC68040 to respond to the access as a slave. By intervening in the access, the processor can update its internal caches for an external write (destination data) or supply cache data to the alternate bus master for an external read (source data). In this manner, the caches in the MC68040 are prevented from accumulating old or invalid copies of data (stale data), and external masters are allowed access to locally modified data within the caches that is no longer consistent with external memory (dirty data). Cache coherency is also supported by allowing memory pages to be specified as write through instead of write back; processor writes to write through pages always update external memory via an external bus access after updating the cache, keeping memory and cache data consistent.

7.2.6 Discussion and Conclusion

The 68040 represents a significant advance in the Motorola 680x0 processor family. The most common commercial applications for these devices are high-end Macintosh computers such as the Quadra series and aftermarket Macintosh accelerator boards such as the Radius Rocket. The FPU pipeline and interface overhead reduction have resulted in an on-chip floating-point unit which outperforms the 68030/68882 floating point capabilities, even when floating point instructions are emulated in software, the 68040 is clocked at a slower speed than the 68030. Memory management technologies such as the Address Translation Cache, write-back main memory block updates, and split instruction/data caches have reduced clock cycles/instruction to a rate (average = 1.3 clock cycles/instr) which rivals RISC architecture processors. This is very unusual, since a primary benefit of reduced instruction set computers is increased throughput due to a smaller number of instruction being fetched from memory. The combination of multistage pipelining, instruction sequencing, and memory management have produced a processor which outperforms its primary competition in both fixed and floating point throughput.

7.3 Intel 80486 Microprocessor

Intel 80486 is an enhanced 32-bit microprocessor with an on-chip floating-point hardware.

7.3.1 Intel 80486/80386 Comparison

Table 7.8 compares the basic features of the 80486 with those of 80386.

TABLE 7.8 80386 vs. 80486

Characteristic	80386	80486
Introduced in	1985; 386SX in 1988	1989
Principle features	Adds paging 32-bit extension, on-chip address translation, and greater speed to 80286 functions. 32-bit microprocessor	Adds on-chip cache, floating-point unit, and greater speed to 386 functions. 32-bit microprocessor
Data bus size accommodated	16-, 32-bit	8-, 16-, 32-bit
Parity	None	Generation and checking
Address bus size	32-bit	32-bit
Bus utilization with zero wait states	70%	50%
On-chip transistors	275,000	1.2 million
Directly addressable memory	4 Gigabytes	4 Gigabytes
Virtual memory size	64 Tetrabytes	64 Tetrabytes
Internal clock	Divides the external clock input by 2	Same as external clock input
Clock	25 MHz to 50 MHz	25 MHz to 66 MHz
Pins	100 for 80386SX and 168 for other 80386's	168
Address and data buses	non-multiplexed	non-multiplexed
Instruction Execution speed	4.5 clocks	1.8 clocks
Registers	8 32-bit general purpose registers 32-bit EIP and Flag register 6 16-bit segment registers 6 64-bit segment descriptor registers 4 32-bit system control registers (CR0-CR3)	All registers listed under the 80386 plus the following registers: 8 80-bit 8 2-bit 8 16-bit 3 16-bit 2 48-bit

TABLE 7.8 80386 vs. 80486 (continued)

Characteristic	80386	80486
Tested debug registers for built-in self test	2 Test registers 8 Debug registers for testing control ROMs, TLB, etc.	5 Test registers 8 Debug registers for testing control ROMs, TLB, cache, etc.
Start of a bus cycle	Defined by activation of ADS output pin by the 80386	Defined by activation of ADS output pin by the 80486
Bus cycle definition	Bus cycle is defined by the encoding of M/IO#, D/C#, and W/R# pins	Bus cycle is defined in the same way except that the encoding of M/IO#, D/C#, and W/R# are different on the 80386 and 80486
Address	Defined by A2-A31 BE0#-BE3#	Same as the 80386
End of bus cycle	Indicated by the RDY# pin RDY# input indicates that an external device has presented valid data on the data bus or that the external device has accepted data	RDY# and BRDY# pins The RDY# pin has the same meaning as the 80386 except that the BRDY# input is used during a burst transfer. A maximum of 16 bytes can be transferred during the burst. The 80486 asserts the BLAST# output pin to end the burst
Direct Memory Access (DMA)	Two pins are used: HOLD input pin HLDA output pin	Three pins are used: HOLD input pin HLDA output pin BREQ output
Bus lock	LOCK# output pin allows the processor to complete multiple bus cycles without interruption via the HOLD pin	Two pins are used: LOCK# and PLOCK#. The 80486 LOCK# output pin has the same meaning as the 80386. The PLOCK# (bus pseudo-lock) output pin allows the processor to complete multiple bus cycles of aligned 64-bit data including floating-point
Bus backoff	Not available	The BOFF# input pin indicates that another bus master needs to complete a bus cycle in order for the 80486's current cycle to complete
Address HOLD	Not available	The AHOLD input pin causes the 80486 to float its address bus in the next clock cycle. This allows an external device to drive an address into the 80486 for internal cache line invalidation
On-chip memory management hardware	Yes	Yes
Operating modes: Real, Protected, and Virtual 8086 modes	Yes. Does not support maximum or minimum mode like the 8086	Same as the 80386
Addressing modes	11	11
On-chip floating-point hardware	No	Yes
Instructions	Listed in Chapter 4 including the floating-point instructions where the 80386 is interfaced to the 80386	All 80386 instructions including the floating-point instructions for the on-chip floating-point hardware plus six new instructions

7.3.2 Special Features of the 80486

Intel 80486 is a 32-bit microprocessor like the Intel 80386. It executes the complete instruction set of the 80386 and the 80387DX floating-point coprocessor. Unlike the 80386, the 80486 on-chip floating-point hardware eliminates the need for an external floating-point coprocessor chip and the on-chip cache minimizes the need for external cache and associated control logic.

The 80486 is object-code-compatible with the 8086, 8088, 80186, 80286, and 80386 processor. It can perform a complete set of arithmetic and logical operations on 8-, 16-, and 32-bit data types using a full-width ALU and eight general-purpose registers. Four-gigabytes of physical memory can be addressed directly via its separate 32-bit addresses and data paths. An on-chip memory management unit is added which maintains the integrity of memory in the multitasking and virtual-memory environments. Both memory segmentation and paging are supported.

As mentioned above, the 80486 has an internal 8K byte cache memory, which provides fast access to recently used instructions and data. The internal write through cache can hold 8K bytes of data or instructions. The internal cache can be invalidated or flushed, so that an external cache controller can maintain cache consistency in multiprocessor environments. Write-back and flush controls over an external cache are also provided. The internal cache and instruction prefetch buffer can be filled very rapidly from memory each clock cycle.

The on-chip floating-point unit performs floating-point operations on the 32-, 64-, and 80-bit arithmetic formats specified in IEEE standard, and is object-code-compatible with the 8087, 80287, 80386 coprocessors.

An instruction restart feature allows programs to continue execution following an interrupt generated by an unsuccessful memory access attempt; an important feature for supporting demand-paged virtual memory.

The fetching, decoding, execution, and address translation of instructions is overlapped within the 80486 processor using instruction pipe-lining. This allows a continuous execution rate of one clock cycle per instruction for most instructions.

The 80486 processor can operate in three modes (set in software): real, protected, and virtual 8086 mode.

After reset or power up, the 80486 is initialized in Real Mode. This mode has the same base architecture as the 8086, but allows access to the 32-bit register set of the 80486 processor. Nearly all of the 80486 processor instructions are available, but default operand-size is 16 bits. The main purpose of Real Mode is to set up the processor for Protected Mode.

The Protected Mode or the Protected Virtual Address Mode is where the complete capabilities of the 80486 become available. Segmentation and paging can both be used in Protected Mode. All existing 8086, 80286, and 386 processor software can be run under the 80486 processor's hardware-assisted protection mechanism.

The Virtual 8086 Mode is a sub-mode for the Protect Mode. It allows 8086 programs to be run but adds the segmentation and paging protection mechanisms of Protected Mode. It is more flexible to run 8086 in this mode than in the Real Mode since it can simultaneously execute the 80486 operating system and both 80286 and 80486 processor applications.

The 80486 is provided with a bus backoff feature. Using this, the 80486 will float its bus signals if another bus master needs control of the bus during a 80486 bus cycle and then restarts its cycle when the bus again becomes available.

The 80486 includes dynamic bus sizing. Using this feature, external controllers can dynamically alter the effective width of the data bus with 8-, 16-, or 32-bit bus widths.

In terms of programming models, the Intel 386 DX or SX has very few differences with the 80486 processor. The 80486 processor defines new bits in the EFLAGS, CR0, and CR3 registers, and in entries in the first and second-level page tables. In the 386 processor, these bits were reserved, so the new architectural features should not be a compatibility issue.

The AC (alignment check) flag (bit number 18) is a new flag added to the 80486 EFLAGS register. All other flags in the 80486 EFLAGS are same as the 80386. The 80486 AC flag can be used in conjunction with an AM (Alignment Mask) bit in the 80486 CR0 control register to control alignment checking. The 80486 performs alignment checking when both the AC flag and AM bit are set to one. This checking is disabled when AM bit is cleared to zero. An alignment check exception is generated when reference is made to an unaligned operand such as a word at an odd byte address or a doubleword at an address which is not an integral multiple of four. This is the 80486 new exception vector 17.

On the 80386, loading a segment descriptor would always cause a locked read and write to set the accessed bit of the descriptor. On the 80486, the locked read and write occur only if the bits are not already set.

The following control register bits in CR0 and CR3 have been added beyond those of the 80386:

1. Five new bits have been defined in the CR0 register. These are NE (Numeric error), WP (Write Protect), AM (Alignment Mask), NW (Not Write-through) and CD (Cache disable). The NE bit enables the standard mechanism for reporting floating-point numeric errors when set. The WP bit, when set to one, write-protects user-level pages against supervisor-mode access. The purpose of the AM bit is already explained. The NW bit enables write-through and cache invalidation cycle when cleared to zero.
2. Two new bits have been defined in the 80486 CR3 control register. These are the PCD (Page-level Cache Disable) and the PWT (Page-level Writes Transparent) bit. The state of the PCD bit is driven on the PCD pin during bus cycles which are not paged, such as interrupt acknowledge cycles, when paging is enabled. The state of the PWT bit, on the other hand, is driven on the PWT pin during bus cycles which are not paged such as interrupt acknowledge cycles when paging is enabled.

7.3.3 80486 New Instructions Beyond Those of the 80386

There are six instructions added to the 80486 instruction set beyond those of the 80386 instruction set as follows:

1. Three New Application Instructions
 - BSWAP instruction
 - XADD instruction
 - CMPXCHG Instruction
2. Three New System Instructions
 - INVD Instruction
 - WBINVD Instruction
 - INVLPG Instruction

The 80386 can execute all its floating-point instructions when the 80387 is present in the system. The 80486, on the other hand, can directly execute all its floating-point instructions (same as the 80386 floating-point instructions) since it has the on-chip floating-point hardware.

The 80486's six new instructions are described in the following. Since the 80386 floating-point instructions are discussed in Chapter 4 and are the same as the 80486, they are not covered in this chapter.

The three new application instructions included with the 80486 are listed below:

1. BSWAP reg_{32} instruction reverses the byte order of a 32-bit register, converting a value in little/big endian form to big/little endian form. That is, the BSWAP instruction exchanges bits 7...0 with bits 32...24 and bits 15...8 with bits 23...16 of a 32-bit register.

Executing this instruction twice in a row leaves the register with the original value. When BSWAP is used with a 16-bit operand size, the result left in the destination operand is undefined. Consider an example of 32-bit operand, if [EAX] = 12345678H, then after BSWAP EAX, the contents of EAX are 78563412H.

Note that little endian is a byte-oriented method where the bytes are ordered (left to right) as 3,2,1, and 0 with byte 3 being the most significant byte. Big endian on the other hand, is also a byte-oriented method where the bytes are ordered (left to right) as 0,1,2, and 3 with byte 3 being the most significant byte.

The BSWAP instruction speeds up execution of decimal arithmetic by operating on four digits at a time.

2. XADD dest, source

```
reg8/mem8, reg8
reg16/mem16, reg16
reg32/mem32, reg32
```

The XADD dest, source loads the destination into the source and then loads the sum of the destination and the original value of the source into the destination. For example, if [AX] = 0123H, [BX] = 9876H, then after XADD AX, BX, the contents of AX and BX are respectively 9999H and 0123H.

3. CMPXCHG dest, source

```
reg8/mem8, reg8
reg16/mem16, reg 16
reg32/mem32, reg32
```

The CMPXCHG instruction compares the accumulator (AL, AX or EAX register) with the destination. If they are equal, the source is loaded into the destination; Otherwise, the destination is loaded into the accumulator. For example, if [DX] = 4324H, [AX] = 4532H, and [BX] = 4532H, then after CMPXCHG BX, DX, the ZF flag is set to one and [BX] = 4324H.

There are three new system instructions used for managing the cache and TLB. These are described below:

1. INVD instruction flushes the internal cache and a special function bus cycle is issued which indicates that any external caches should also be flushed. Data held in write-back external caches is discarded.
2. WBINVD instruction flushes the internal cache and a special function bus cycle is used which indicates that any external cache should write-back its contents to main memory. Another special function bus cycle follows, directing the external cache to flush itself.
3. INVLPG instruction is used to invalidate a single entry in the TLB.

The form of the 80486 MOV instruction used to access the test register has changed beyond that of the 80386. Also, new test registers have been defined for the cache in the 80486 and the model of the 80486 TLB (Translation Lookaside buffer) accessed through the test register has changed. Note that the TLB in 80486 is a cache for page table entries.

7.4 Intel Pentium Microprocessor

Table 7.9 summarizes the basic differences between the basic features of 486 and Pentium processor families:

TABLE 7.9 Basic Differences Between 80486 and Pentium Processor Families

Feature	486 Processor	Pentium Processor
Clock	25 to 66 MHz	60 to 100 MHz
Address and data buses	32-bit data bus 32-bit address bus	32-bit address bus 64 bit data bus
Pipeline model	Single	Dual
Internal cache	8K for both data and instruction	8K data 8K instruction
Cache type	Write-through	Write back
Number of transistors	1.2 million	3.2 million
Performance at 66 MHz in MIPS (Millions of Instructions Per Second)	54 MIPS	112 MIPS
Number of pins	168	273

Microprocessors have served largely separate markets and purposes: business PCs and engineering workstations. The PCs have used Microsoft's DOS and Windows operating systems while the workstations have used various features of UNIX. The PCs have not been utilized in the workstation market because of their relatively modest performance, especially with regard to complicated graphics display and their floating-point performance. Workstations have been kept out of the PC market partially because of their high prices and hard-to-use system software.

The Pentium will bring the PCs up to workstation-class computational performance with sophisticated graphics. The Intel Pentium is a 32-bit microprocessor with a 64-bit data bus. The Intel Pentium, like its predecessor, the Intel 80486 is 100% object code compatible with other X86 systems.

The Pentium processor has three modes of operation. The mode determines which instructions and architecture features are accessible. These modes are: real-address mode (also called "real mode"), protected mode, and system management mode.

In the real-address mode, the Pentium processor runs programs written for 8086/8088, 80186/80188, or for the real-address mode of an 80286, 80386, or 80486. The architecture of the Pentium processor in this mode is identical to that of the 8086/8088, 80186, and 80188 processors.

In the protected mode, all instruction and architectural features of the Pentium are available to the programmer. Some of the architectural features of the Pentium processor include memory management, protection, multitasking, and multiprocessing. While in protected mode, the virtual-8086 mode can be enabled for any task. For the v86 mode, the Pentium can directly execute 'real address mode' 8086 software in a protected, multitasking environment.

The Pentium processor is provided with a System Management Mode (SMM) similar to the one used in the 80486SL line, which allows engineers to design for low power usage. SMM is entered through activation of an external interrupt pin (System Management Interrupt, SMI#). All registers are saved for later restoration. The SMM interrupt service routine is then executed from a separate address space. SMM is exited by executing a new instruction (RSM) executable from SMM. The Pentium then returns to the interrupted program.

In December 1994, Intel detected a flaw in the Pentium chip while performing certain division calculations. The Pentium is not the first chip that Intel had problems with. The first version of the Intel 80386 had a math flaw which Intel quickly fixed before any complaints. Some experts feel that Intel should have acknowledged the math problem in the Pentium when it was first discovered and then offered to replace the chips. In that case, the problem with the Pentium most likely would have been ignored by the users. However, Intel was heavily criticized by computer magazines when the division flaw was first detected in the Pentium.

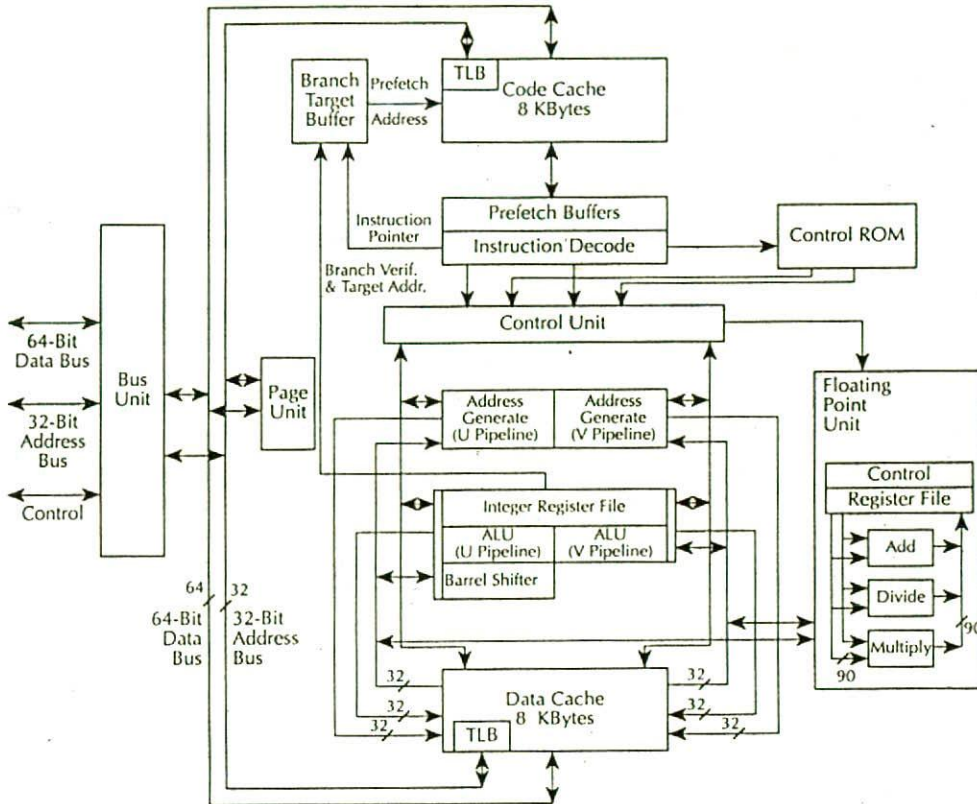


FIGURE 7.30 Pentium™ processor block diagram.

The flaw in the division algorithm in the Pentium was caused by a problem with a look-up table used in the division. Errors occur in the fourth through fifteenth significant decimal digits. This means that in a result such as 5.758346, the last three digits could be incorrect. For example, the correct answer for the operation $4,195,835 - (4,195,835 \div 3,145,727) \times (3,145,727)$ is zero. The Pentium provided a wrong answer of 256. IBM claimed this problem can occur once every 24 days.

It is the author's opinion that the circuitry inside the 32-bit microprocessors are so complex that the math flaw in the Pentium is not unusual. Intel already regained its reputation by fixing the division flaw in the Pentium and has already been shipping replacement chips.

Intel shipped approximately 4 million original Pentium chips by the end of 1994. The manufacturing cost to Intel for replacement chip amounts to approximately \$150 per chip and results in \$100 per chip for shipping and upgrade services. Based on these numbers, the cost to Intel for a total recall of Pentium chips will be \$1 billion which is almost 50% of Intel's 1994 earnings.

The author strongly believes in Intel products. Intel lived up to the expectations of the author and many other users of Intel products by correcting the division flaw in the Pentium chip.

7.4.1 Pentium Processor Block Diagram

Figure 7.30 shows a block diagram of the Pentium processor.

The Pentium microprocessor is based on a superscalar design. This means that the processor includes dual pipelining and executes more than one instruction per clock cycle. Note that

scalar microprocessors such as the 80486 family have only one pipeline and execute one instruction per clock cycle and superscalar processors allow more than one instruction to be executed per clock cycle.

The Pentium microprocessor contains the complete 80486 instruction set along with some new ones which are discussed later. Pentium's on-chip memory management unit is completely compatible with that of the 80486.

The Pentium includes faster floating-point on-chip hardware than the 80486. Pentium's on-chip floating-point has been completely redesigned over the 80486. Faster algorithms provide up to ten times speed-up for common operations such as add, multiply, and load. The two instruction pipelines and on-chip floating-point unit are capable of independent operations. Each pipeline issues frequently-used instructions in a single clock cycle. The dual pipelines can jointly issue two integer instructions in one clock cycle or one floating-point instruction (under certain circumstances, two floating-point instructions) in one clock cycle.

Branch prediction is implemented in the Pentium by using two prefetch buffers: one to prefetch code in a linear fashion and one to prefetch code according to the contents of the Branch Target Buffer (BTB) so the required code is almost always prefetched before it is needed for execution. Note that the branch addresses are stored in the Branch Target Buffer (BTB).

The Pentium includes 8Kbytes of on-chip code cache and 8Kbytes of data cache. Each cache is two-way set associative. Also, each cache has a dedicated Translation Lookaside Buffer (TLB) to translate linear addresses to physical address used by each cache.

The block diagram shows the two instruction pipelines, the "U" pipe and the "V" pipe which are not equivalent and interchangeable. The U-pipe can execute all integer and floating-point instructions, while the V-pipe can only execute simple integer instructions and the floating-point exchange register contents (FXCH) instructions.

The instruction decode unit decodes the prefetched instructions so that the Pentium can execute them. The control ROM includes the microcode for the Pentium processor and has direct control over both pipelines.

A barrel shifter is included in the chip for fast shift operations.

7.4.2 Pentium Registers

The Pentium processor includes the same registers as the 80486. Three new system flags are added to the 32-bit EFLAGS register. These are the ID (ID flag, bit 21 of EFLAGS), the VIP (Virtual Interrupt Pending, bit 20 of EFLAGS) and the VIF (Virtual Interrupt Flag, bit 19 of EFLAGS).

The ability of a program to set or clear the ID flag indicates that the processor supports the CUID instruction. The CUID instruction provides information about the family member and model of the microprocessor on which it is executing.

7.4.3 Pentium Addressing Modes and Instructions

The Pentium includes the same addressing modes as the 80386/80486.

The Pentium microprocessor includes three new application instructions and four new system instructions beyond those of the 80486.

The new application instructions are CMPXCHG8B, CUID and RDTSC. The new system instructions are

- i) **MOV CR4, r32 and MOV r32, CR4**
- ii) **RDMSR**
- iii) **WRMSR**
- iv) **RSM**

The details of RDTSC instruction is considered proprietary and confidential by Intel and is not provided in Intel manuals. Detailed information on this may be obtained from Intel upon signing a non-disclosure agreement.

CMPIXCHG8B reg64 compares the 64-bit value in EDX:EAX with the 64-bit contents

or
mem64

of reg 64 or mem64. If they are equal, the 64-bit value in ECX:EBX is stored in reg64 or mem64; otherwise the content of reg64 or mem64 is loaded into EDX:EAX.

Certain features of the Pentium processor described in the 'Pentium Processor Data Book' are considered proprietary by Intel and may be obtained upon signing a non-disclosure agreement with Intel. These features are specific to the Pentium processor and may not be continued in the same way in future processors. Examples include functions for testability, performance monitoring, and machine check errors. These features are accessed through Model Specific Registers. The new instructions RDMSR and WRMSR are used to read from and write into these registers. In order to use such model specific features, software should check "Family" member by using the CPUID instruction. Software which uses these registers and functions may not be compatible with Intel's future processors. The RSM instruction resumes operation of a program interrupted by a System Management Mode interrupt.

Pentium floating-point instructions execute much faster than those of the 80486 instructions. For example, a 66MHz Pentium microprocessor provides about three times the floating-point performance of a 66MHz Intel 80486 DX2 microprocessor.

7.4.4 Pentium Vs. 80486 Basic Differences in Registers, Paging, Stack Operations, and Exceptions

7.4.4.a Registers of the Pentium processor vs. those of the 80486

This section discusses the basic differences between the Pentium and 80486 control, debug, and test registers. Two bits, namely CD (Cache Disable) and NW (Not Write-through) in the control register, CR0 are redefined in the Pentium. The values of zero in both CD and NW in CR0 implement a write-back strategy for the data cache in the Pentium. On the 80486, these values implement a write-through strategy.

One new control register, CR4 is included in the Pentium. CR4 contains bits that enable certain extensions to the 80486 provided in the Pentium processor. These extensions include functions such as debugging extensions that support I/O breakpoints and machine check exceptions for handling certain hardware error conditions.

The Pentium processor defines the type of breakpoint access by two bits in DR7 to perform breakpoint functions such as breakpoint on instruction execution only, break on data writes only and break on data reads or writes but not instruction fetches.

The implementation of test registers on the 80486 used for testing the cache and TLB has been redesigned using model specific registers in the Pentium processor.

7.4.4.b Paging

The Pentium processor provides an extension to the memory management/paging functions of the 80486 to support larger page sizes.

7.4.4.c Stack Operations

The Pentium microprocessor, 80486 and 80386 push a different value on the stack for a PUSH SP instruction than the 8086. The 32-bit processors push the value of the SP before it is decremented while the 8086 pushes the value of the SP after it is decremented.

7.4.4.d Exceptions

The Pentium processor implements new exceptions beyond those of the 80486. For example, a machine check exception is newly defined for reporting parity errors and other hardware errors.

External hardware interrupts on the Pentium may be recognized on different instruction boundaries due to the pipelined execution of the Pentium processor and possibly, an extra instruction passing through the v-pipe concurrently with an instruction in the u-pipe. When the two instructions complete execution, the interrupt is then serviced. Therefore, the EIP pushed onto the stack when servicing the interrupt on the Pentium processor may be different than that for the 80486 (i.e. it is serviced later).

The priority of exceptions is the same on both the Pentium and 80486.

7.4.5 Input/Output

The Pentium processor handles I/O in the same way as the 80486. The Pentium can use either standard I/O or memory mapped I/O. Standard I/O is accomplished by using IN/OUT instructions and a hardware protection mechanism.

When memory-mapped I/O is used, memory-reference instructions are used for input/output and the protection mechanism is provided via segmentation or paging.

The Pentium can transfer 8-, 16- or 32-bits to a device. Like memory-mapped I/O, 16-bit ports using standard I/O should be aligned to even addresses so that all 16 bits can be transferred in a single bus cycle. Like doublewords in memory-mapped I/O, 32-bit ports in Standard I/O should be aligned to address which are multiples of four. The Pentium supports I/O transfer to misaligned ports, but there is a performance penalty because an extra bus cycle must be used.

The INS and OUTS instructions move blocks of data between I/O ports and memory. The INS and OUTS instructions, when used with repeat prefixes, perform block input or output operations. The string I/O instructions can operate on byte (8-bit) strings, word (16-bit) strings, or double word (32-bit) strings.

When the Pentium is running in Protected mode, I/O operates as in real address mode with additional protection features as follows:

1. References to memory-mapped I/O ports, like any other memory reference, are subject to access protection and control by both segmentation and paging.
2. Using standard I/O, execution of an I/O instruction is subject to two protection mechanisms:
 - a. The IOPL field in the EFLAGS register controls access to the I/O instructions.
 - b. The I/O permission bit map of a TSS (Task State Segment) controls access to individual port in the I/O address space.

7.4.6 Applications With the Pentium

The performance of the Pentium's Floating-Point Unit (FPU) makes it appropriate for wide areas of numeric applications:

1. Business Data Processing

Pentium's FPU can accept decimal operands and produce extra decimal results of up to 18 digits. This greatly simplifies accounting programming. Financial calculations that use power functions can take advantage of exponential and logarithmic functions.

2. Many mini and mainframe large simulation problems can be executed by the Pentium. These applications include complex electronic circuit simulations using SPICE and simulation of mechanical systems using finite element analysis.
3. Since the FPU of the Pentium can execute integer instructions concurrently, the Pentium can be used in graphics applications such as computer-aided design (CAD).
4. The Pentium's FPU can move and position machine control heads with accuracy in real time. Axis positioning can efficiently be performed by the hardware trigonometric support provided by the FPU. The Pentium can, therefore, be used for computer numerical control (CNC) machines.
5. The pipelined instruction feature of the Pentium processor makes it an ideal candidate for DSP (Digital Signal Processing) and related applications for computing matrix multiplications and convolutions.

Other possible application areas for the Pentium includes robotics, navigation, data acquisition, and Process Control.

QUESTIONS AND PROBLEMS

- 7.1
 - i) Identify the the 68030 registers that are not included in the 68020.
 - ii) Name a 68030 register which is included in the 68020 but formatted in a different way from the 68020. Why is it structured differently?
- 7.2 What are the functions of the 68030 REFILL, STATUS, and STERM pins?
- 7.3
 - i) What is the difference between the 68030 DSACK and STERM?
 - ii) What are the minimum bus access times in clock cycles for 68030 synchronous and asynchronous operations?
- 7.4 What conditions must be satisfied before a cache hit occurs for either instruction or data cache in the 68030?
- 7.5 Assume user data space. The 68030 executes the instruction CLR.W(A1) with [A1] = \$20507002 and [\$20507002] = \$1234 with the following information in the data cache:

		Valid bits								
		Tag	0	1	2	3	LW0	LW1	LW2	LW3
0	1	205070	1	1	1	1	12345124	00121234	70001111	01112222
1	1	205070	0	1	1	1	77777777	12121212	FF0011EE	AAAA1111
2	1	5F1236	1	0	1	0	72101234	25252020	FFFFFFFF	00110011
3	1	021472	0	0	0	1	11223344	BBBBAAAA	71171625	00200000

Assume all numbers in hex.

Will a cache hit occur? Why or why not?

- 7.6 What are the functions of 68030 EBREQ and CBACK?
- 7.7 Write a 68030 instruction sequence to clear the instruction cache entry for address \$00005040.

7.8 For a page size of 4K bytes, translate the 68030 logical address \$00000240 to a physical address. Assume the following data:

		000010000	TCB
\$001000		0020 0000	Level 1 Descriptors
\$001004		0020 0010	
\$001008		0020 0020	
Page 0	20000	7A237	Level 2 Page descriptor
1	20004	8F1C2	
2	20008	20315	

Assume all numbers in hex.

7.9 Identify the inputs and outputs that affect the 68030 MMU.

- 7.10 i) How many levels of translation tables does the 68030 MMU provide?
 ii) What is the maximum number of page descriptor entries in the ATC?

7.11 Assume the following data contents in the 68030 CRP:

31												0
0	0	1	0	0	0	0	0	3				
	2	A	6	7	1	1	2	4				
31												0

Assume all numbers in hex.

- i) Is this a page or table descriptor?
 ii) What is the address of the next table?
 iii) What is the allowed range of logical addresses for the next table?
- 7.12 Determine the contents of the 68030 TC which will enable MMU, SRP, disable function code look-up, and will permit 2 levels of equal size look-up with a page size of 1K byte for a 30-bit address.
- 7.13 Show the contents of the 68030 TT0 for translating addresses \$20000000–\$21FFFFFF in supervisor data space. These addresses should be read-only and cacheable.
- 7.14 Find the following 68030 MMU instructions:
 i) for accessing an MMU register
 ii) for placing descriptors in the ATC
 iii) for invalidating one or more entries in the cache
 iv) for testing a descriptor either in the memory or in the cache
- 7.15 Write a 68030 instruction sequence for initializing the TC, CRP, and SRP registers from memory pointed to by A2. Can you initialize all MMU registers by this instruction? If not, list the register or registers.

- 7.16 Compare 68030 on-chip MMU features with those of the 68851 MMU.
- 7.17 What is meant by configuration error exception in the 68030?
- 7.18 In the 68030 MMU (demand paging system), what happens when a task attempts to access a page not in memory? What action should be taken by the demand paging operating system?
- 7.19 Discuss the main features of the 68040 as compared with those of the 68030.
- 7.20 What is the basic difference between the 68881/68882 floating-point coprocessor and the 68040 on-chip floating-point hardware?
- 7.21 Write a 68040 assembly language program to compute the area of a circle, $A = \pi r^2$ where r is the 32-bit radius of the circle in D0.
- 7.22 Write a 68040 assembly language program to compute:

$$20\text{Log}_{10}(V_1/V_2)$$

where V_1 and V_2 are 32-bit single precision numbers stored in D0 and D1 respectively. Store the result in D2.

- 7.23 Write a 68040 assembly language program to compute the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where a , b , and c are 32-bit integers.

- 7.24 Compare the on-chip hardware features of the 80486 and Pentium microprocessors.
- 7.25 What are the sizes of the address and data buses of the 80486 and Pentium?
- 7.26 Identify the main differences between the 80486 and Pentium.
- 7.27 What are the clock speed, pipeline model, number of on-chip transistors and number of pins on the 80486 and Pentium processors?
- 7.28 Discuss the application areas in which the Pentium-based PC's will be used.
- 7.29 Identify the main differences between the Intel 80386 and 80486.
- 7.30 Identify the pins on the 80386 and 80486 by which the bus cycle is defined.
- 7.31 What are the functions of the 80486 i) RDY# and BRDY# pins? ii) LOCK# and PLOCK#?
- 7.32 What is meant by the 80486 BUS BACKOFF feature?
- 7.33 What is meant by the 8086 real, protected, and virtual 8086 modes?

- 7.34 What is the meaning of 80486 alignment checking? How is it handled in the 80486?
- 7.35 How many new control bits have been added to control registers CR0 and CR3 by the 80486 beyond those of the 80386?
- 7.37 How many new instructions are added to the 80486 beyond those of the 80386? List them.
- 7.38 Given the following registers contents,

[EBX] = 7F27108AH
[ECX] = 2A157241H,

What is the content of ECX after execution of the following 80486 instruction sequence:

MOV EBX, ECX
BSWAP ECX
BSWAP ECX
BSWAP ECX
BSWAP ECX

- 7.39 If [EBX] = 0123A212H, and [EDX] = 46B12310H, then what are the contents of EBX and EDX after execution of the 80486 instruction XADD EBX, EDX?
- 7.40 If [BX] = 271AH, [AX] = 712EH and [CX] = 1234H, what are the contents of CX after execution of the 80486 instruction CMPXCHG CX, BX?
- 7.41 What is the purpose of each of the 80486 INVD, WBINVD and INVLPG instructions?
- 7.42 What are three modes of the Pentium Processor? Discuss them briefly.
- 7.43 What is meant by the statement "The Pentium processor is based on a superscalar design."?
- 7.44 What are the purposes of the U-pipe and V-pipe of the Pentium processor?
- 7.45 Discuss how the branch prediction feature is implemented in the Pentium processor.
- 7.46 What are the sizes of the data and instruction caches in the Pentium?
- 7.47 How many new bits are added in the Pentium's EFLAGS register beyond those of the 80486? Discuss them.
- 7.48 How many new application and system instructions are added to the Pentium beyond those of the 80486? List them.
- 7.49 Discuss the main differences between the Pentium and 80486 control, debug, and test registers.
- 7.50 Discuss the new exceptions implemented in the Pentium beyond those of the 80486.
- 7.51 How is the I/O protection mechanism implemented in the standard I/O and Memory-mapped I/O of the Pentium? Discuss each.
- 7.52 Discuss two applications of the Pentium processor. Why is the Pentium an ideal candidate for these applications?