Fundamentals of Digital Logic and Microcomputer Design. M. Rafiquzzaman

11

INTEL AND MOTOROLA 32- & 64-BIT MICROPROCESSORS

This chapter provides a summary of the basic features of 32- and 64-bit microprocessors manufactured by Intel and Motorola. Intel 80386 and Motorola 68020 are covered in detail while an overview of the other 32-bit microprocessors is also included. Finally, a brief coverage of the 64-bit microprocessors is provided.

11.1 Typical Features of 32-bit and 64-bit Microprocessors

This section describes the basic aspects of typical 32- and 64-bit microprocessors. Topics include on-chip features such as pipelining, memory management, floating-point, and cache memory implemented in typical 32- and 64-bit microprocessors.

The first 32-bit microprocessor was Intel's problematic iAPX432, and was introduced in 1980. Soon afterwards, the concept of "mainframe on a chip" or "micromainframe" was used to indicate the capabilities of these microprocessors and to distinguish them from previous 8- and 16-bit microprocessors.

The introduction of several 32-bit microprocessors revolutionized the microprocessor world. The performance of these 32-bit microprocessors is actually more comparable to that of superminicomputers such as Digital Equipment Corporation's VAX11/750 and VAX11/780. Designers of 32-bit microprocessors have implemented many powerful features of these mainframe computers to increase the capabilities of the microprocessor chip sets. These include pipelining, on-chip cache memory, memory management, and floating-point arithmetic.

As mentioned in Chapter 8, pipelining is the technique in which instruction fetch and execute cycles are overlapped. This method allows simultaneous preparation for execution of one or more instructions while another instruction is being executed. Pipelining was used for many years in mainframe and minicomputer CPUs to speed up the instruction execution time of these machines. The 32-bit microprocessors implement the pipelining concept and simultaneously operate on several 32-bit words, which may represent different instructions or part of a single instruction.

Although pipelining greatly increases the rate of execution of nonbranching code, pipelines must be emptied and refilled each time a branch or jump instruction is in the code. This may slow down the processing rate for code with many branches or jumps. Thus, there is an optimum pipeline depth, which is strongly related to the instruction set, architecture, and gate density attainable on the processor chip. For many of the applications run on the 32-bit microprocessors, the three-stage pipeline is considered a reasonably optimal depth.

With memory management, virtual memory techniques, traditionally a feature of mainframes, are also implemented as on-chip hardware on typical 32-bit microprocessors.

This allows programmers to write programs much larger than those that could fit in the main memory space available to the microprocessors; the programs are simply stored on a secondary device, such as a disk drive, and portions of the program are swapped into main memory as needed.

Segmentation circuitry has been included in many 32-bit microprocessor chips. With this technique, blocks of code called "segments," which correspond to modules of the program and have varying sizes set by the programmer or compiler, are swapped. For many applications, however, an alternative method borrowed from mainframes and superminis called "paging" is used. Basically, paging differs from segmentation in that pages are of equal sizes. Demand paging, in which the operating system automatically swaps pages as needed, can be used with all 32-bit microprocessors.

Floating-point arithmetic is yet another area in which the new chips are mimicking mainframes. With early microprocessors, floating-point arithmetic was implemented in software, largely as a subroutine. When required, execution would jump to a piece of code that would handle the tasks. This method, however, slows the execution rate considerably, so floating-point hardware, such as fast bit-slice (registers and ALU on a chip) processors and, in some cases, special-purpose chips, was developed. Other than the Intel 8087, these chips behaved more or less like peripherals. When floating-point arithmetic was required, the problems were sent to the floating-point processor and the CPU was freed to move on to other instructions while it waited for the results. The floating-point processor is implemented as on-chip hardware in typical 32-bit microprocessors, as in mainframe and minicomputer CPUs. Caching or memory-management schemes are utilized with all 32-bit microprocessors in order to minimize access time for most instructions.

A cache, used for years in minis and mainframes, is a relatively small, high-speed memory installed between a processor and its main memory. The theory behind a cache is that a significant portion of the CPU time spent running typical programs is tied up in executing loops; thus, the chances are good that if an instruction to be executed is not the next sequential instruction, it will be one of some relatively small number of instructions back, a concept known as locality of reference. Therefore, a high-speed memory large enough to contain most loops should greatly increase processing rates. Cache memory is included as on-chip hardware in typical 32-bit microprocessors.

Typical 32-bit microprocessors such as Pentium and PowerPC chips are superscalar processors. This means that they can execute more than one instruction in one clock cycle. Also, some 32-bit microprocessors such as the PowerPC contain an on-chip real-time clock. This allows these processors to use modern multitasking operating systems that require time keeping for task switching and for keeping the calendar date.

A few 32-bit microprocessors implement a multiple branch prediction feature. This allows these microprocessors to anticipate jumps of the instruction flow ahead of time. Also, some 32-bit microprocessors determine an optimal sequence of instruction execution by looking at decoded instructions and then determining whether to execute or hold the instructions. Typical 32-bit microprocessors use a "look ahead" approach to execute instructions. Typical 32-bit microprocessors instruction pool for a sequence of instructions and perform a useful task rather than execute the present instruction and then go to the next.

The 64-bit microprocessors include all the features of 32-bit microprocessors. In addition, they also contain multiple on-chip integer and floating-point units, a larger address and data bus. The 64-bit microprocessors can typically execute 4 instructions per clock cycle and can run at a clock speed of more than 300 MHz.

The Pentium microprocessor is designed using a combination of mostly microprogramming (CISC--Complex Instruction Set Computer) and some hardwired control (RISC --Reduced Instruction Set Computer) whereas the PowerPC is designed using hardwired control with almost no microcode. The PowerPC is a RISC microprocessor. This means that a simple instruction set is included with PowerPC. The PowerPC instruction set includes register to register, load, and store instructions. All instructions involving arithmetic operations use registers; load and store instructions are utilized to access memory. Almost all computations can be obtained from these simple instructions. Finally, the 64-bit microprocessors are ideal candidates for data-crunching machines and high-performance desktop systems/workstations.

11.2 Intel 32-Bit and 64-Bit Microprocessors

This section provides a summary of Intel 32-bit and 64-bit microprocessors. The Intel line of microprocessors has gone through many changes. The 8080/8085 (8-bit) was the first major chip by Intel but did not see major use. In 1978, Intel introduced a more powerful processor called the 8086. The 8086 is covered in detail in earlier sections of this chapter. This chip had many improved features over the 8080/85. As mentioned before, the 8086 is a 16-bit processor and utilizes pipelining. Pipelining allows the processor to execute and fetch instructions at the same time. The Intel line has progressed through the years to the 80286, 80386, 80486, and Pentium. The general trend has been an expansion of the bit width of the processors both internally and externally. The Pentium processor was introduced in 1993, and the name was changed from 80586 to Pentium because of copyright laws. The processor uses more than 3 million transistors and had an initial speed

TABLE 11.1 Intel 80386/80486/Pentium Microprocessors

	80386DX	80386SX	80486DX	80486SX	80486DX2	Pentium
• Introduced	October 1985	June 1988	April 1989	April 1991	March 1992	March 1993
• Maximum Clock Speed (MHz)	40	33	50	25	100	233
• MIPS*	6	2.5	20	16.5	54	112
• Transistors	275,000	275,000	1.2 million	1.185 million	1.2 million	3.1 million
• On-chip cache memory	Support chips available	Support chips available	Yes	Yes	Yes	Yes
• Data bus	32-bit	16-bit	32-bit	32-bit	32-bit	64-bit
• Address bus	32-bit	24-bit	32-bit	32-bit	32-bit	32-bit
• Directly addr. memory	4 GB	16MB	4 GB	4 GB	4 GB	4 GB
• Pins	132	100	168	168	168	273
• Virtual memory	Yes	Yes	Yes	Yes	Yes	Yes
• On-chip memory management and protection	Yes	Yes	Yes	Yes	Yes	Yes
• Floating point unit	387DX	387SX	on chip	487SX	on chip	on chip

* MIPS means million of instructions per second that the microprocessor can execute. MIPS is typically used as a measure of performance of a microprocessor. Faster microprocessors have a higher MIPS value.

of 60 MHz. The speed has increased over the years to the latest speed of 233 MHz. Table 11.1 compares the basic features of the Intel 80386DX, 80386SX, 80486DX, 80486SX, 80486DX2, and Pentium. These are all 32-bit microprocessors. Note that the 80386SL (not listed in the table) is also a 32-bit microprocessor with a 16-but data bus like the 80386SX. The 80386SL can run at a speed of up to 25 MHz and has a direct addressing capability of 32 MB. The 80386SL provides virtual memory support along with on-chip memory management and protection. It can be interfaced to the 80387SX to provide floating-point support. The 80386SL includes an on-chip disk controller hardware.

The Pentium microprocessor uses *superscalar* technology to allow multiple instructions to be executed at the same time. The Pentium uses BICMOS technology, which combines the speed of bipolar transistors and the power efficiency of CMOS technology. The internal registers are only 32 bits even though externally it has a 64-bit data bus. It has a 32-bit address bus, which allows 4 gigabytes of addressable memory space. The math coprocessor is on-chip and is up to ten times faster than the 486 in performing certain instructions. There are two execution units in the Pentium that allow the multiple execution. The multiple execution only works for instructions that are data independent, meaning that an instruction executed immediately after another using the previous result cannot be done. The Pentium uses two execution units called the "U and V pipes." Each has five pipeline stages. The U pipe can execute any of the instructions in the 80x86 set, but the V pipe executes only simple instructions. Another new feature of the Pentium is branch prediction. This feature allows the Pentium to predict and prefetch codes and advance them though the pipeline without waiting for the outcome of the zero flag.

The implementation of virtual memory is an important feature of the Pentium. It allows a total of 64 terabytes of virtual memory. The 386/486 allowed only a 4K page size for virtual memory, but the Pentium allows either 4K or 4M page sizes. The 4K page option makes it backward compatible with the 386/486 processors. The 4M page size option allows mapping of a large program without fragmentation. It reduces the amount of page misses in virtual memory mode.

In the next section, the Intel 80386 is covered in detail.

Table 11.1 compares the basic features of 80386, 80486, and Pentium.

11.3 Intel 80386

The Intel 80386 is Intel's first 32-bit microprogrammed microprocessor. Its introduction in 1985 facilitated the introduction of Microsoft's Windows operating systems. The high-speed computer requirement of the graphical interface of Windows operating systems was supplied by the 80386. Also, the on-chip memory management of the 80386 allowed memory to be allocated and managed by the operating system. In the past, memory management was performed by software.

The Intel 80386 is a 32-bit microprocessor and is based on the 8086. A variation of the 80386 (32-bit data bus) is the 80386SX microprocessor, which contains a 16-bit data bus along with all other features of the 80386. The 80386 is software compatible at the object code level with the Intel 8086. The 80386 includes separate 32-bit internal and external data paths along with 8 general-purpose 32-bit registers. The processor can handle 8-, 16-, and 32-bit data types. It has separate 32-bit data and address pins, and generates a 32-bit physical address. The 80386 can directly address up to 4 gigabytes (2^{32}) of physical memory and 64 tetrabytes (2^{46}) of virtual memory. The 80386 can be operated from a

12.5-, 16-, 20-, 25-, 33-, or 40-MHz clock. The chip has 132 pins and is typically housed in a pin grid array (PGA) package. The 80386 is designed using high-speed HCMOS 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 protection rules required in a multitasking environment. 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.

The main differences between the 8086 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 8086.

11.3.1 Internal 80386 Architecture

The internal architecture of the 80386 includes several functional units 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. Typical functional units of the 80386 are these:

- Bus interface unit (BIU)
- Execution unit (EU)
- Segmentation unit
- Paging unit

The 80386 BIU performs similar function as the 8086 BIU. The execution unit processes the instructions from the instruction queue. It contains mainly a control unit and a data unit. The control unit contains microcode and parallel hardware for fast multiplication, division, and effective address calculation. The data unit includes an ALU, 8 general-purpose registers, and a 64-bit barrel shifter for performing multiple bit shifts in one clock cycle. The data unit carries out data operations requested by the control unit. 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 the linear addresses into physical addresses. If paging is not enabled, the physical address is identical to the linear address and no translation is necessary. The 80386 segmentation and paging units support memory management functions. The 80386 does not contain any on-chip cache. However, external cache memory can be interfaced to the 80386 using a cache controller chip.

11.3.2 Processing Modes

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 "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. When entering into V86 mode, the 80386 can execute an 8086 program. The processor can then leave V86 mode and enter protected mode to

execute an 80386 program.

As mentioned, the 80386 enters real-address mode upon hardware reset. In this mode, the protection enable (PE) bit in a control register—the control register 0 (CR0)—is cleared to zero. Setting the PE bit in CR0 places the 80386 in protected mode. When the 80386 is in protected mode, setting the VM (virtual mode) bit in the flag register (the EFLAGS register) places the 80386 in V86 mode.

11.3.3 Basic 80386 Programming Model

The 80386 basic programming model includes the following aspects:

- Memory organization and segmentation
- Data types
- Registers
- Addressing modes
- Instruction set

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

Memory Organization and Segmentation

The 4-gigabyte physical memory of the 80386 is structured as 8-bit bytes. Each byte can be uniquely accessed as a 32-bit address. The programmer can write assembly language programs without 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:

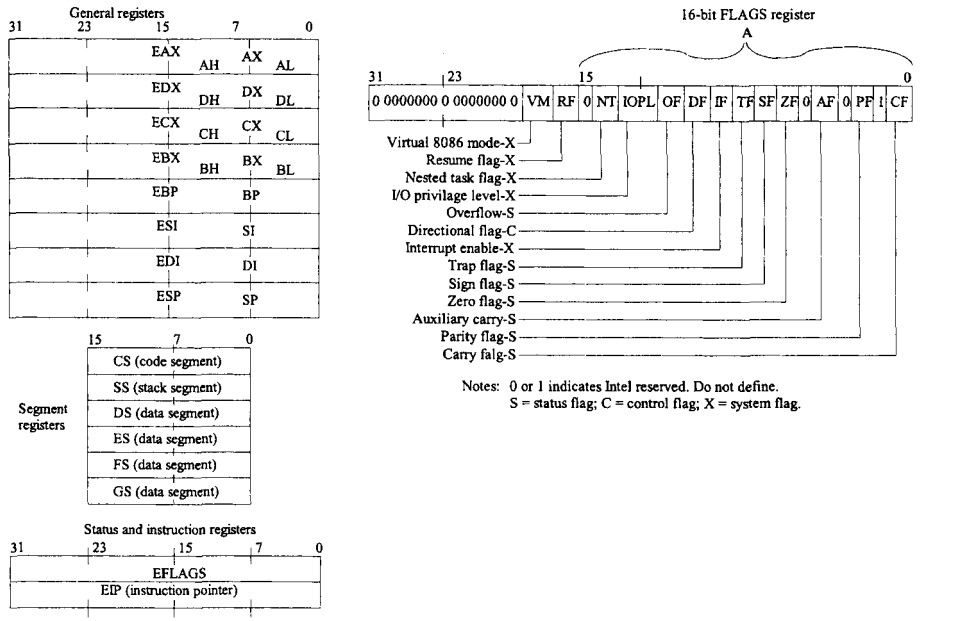
An address space includes a single array of up to 4 gigabytes. The 80386 maps the 4-gigabyte space into the physical address space automatically by using an address-translation scheme transparent to the applications programmers.

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 64 terabytes. The processor maps this address space onto the physical address space (up to 4 gigabytes by an address-translation technique).

Data Types

Data types can be byte (8-bit), word (16-bit with the low byte addressed by n and the high byte addressed by $n + 1$), and double word (32-bit with byte 0 addressed by n and byte 3 addressed by $n + 3$). 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 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. That is, for 32-bit words, addresses should start at 0, 4, 8, ... for the highest speed.

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 that 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 from 0 bytes to $2^{32} - 1$ bytes), bit field (a contiguous sequence of bits starting at any bit position of any byte and containing up to 32 bits), bit string (a contiguous sequence



(a) Applications register set (b) EFLAGS register

FIGURE 11.1 80386 registers

of bits starting at any position of any byte and containing up to 2^{32} - 1 bits), and packed/ unpacked BCD. When the 80386 is interfaced to a coprocessor such as the 80287 or 80387, then floating-point numbers are supported.

Registers

Figure 11.1 shows the 80386 registers. As shown in the figure, the 80386 has 16 registers classified as general, segment, status, and instruction pointer. The 8 general registers are the 32-bit registers EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. The low-order word of each of these 8 registers has the 8086 register name 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 processor.

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 same as that of the corresponding 8086 registers. The two additional data segment registers FS and GS are included in the 80386 so that the four data segment registers (DS, ES, FS, and GS) can access four separate data areas and allow programs to access different types of data structures.

The flag register is a 32-bit register, named EFLAGS in Figure 11.1, that shows the meaning of each bit in this register. The low-order 16 bits of EFLAGS is named FLAGS and can be treated as a unit. This is useful when executing 8086 code because this part of EFLAGS is similar to the FLAGS register of the 8086. The 80386 flags are grouped into three types: status flags, control flags, and system flags.

The status flags include CF, PF, AF, ZF, SF, and OF, like the 8086. The control flag DF is used by strings like the 8086. 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. Let us explain some of the system flags:

- **IOPL** (I/O privilege level). This 2-bit field supports the 80386 protection feature.
- **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). is used during debugging.
- **VM** (virtual 8086 mode). When the VM bit is set to 1, the 80386 executes 8086 programs. When the VM bit is 0, the 80386 operates in protected mode.
- 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 low-order 16 bits of EIP is named IP and is useful when the 80386 executes 8086 instructions.

11.3.4 80386 Addressing Modes

The 80386 has 11 addressing modes, classified into register/immediate and memory addressing modes. The register/immediate type includes 2 addressing modes, and the memory addressing type contains 9 modes.

Register/Immediate Modes

Instructions using the register or immediate modes operate on either register or immediate operands. In 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 1. In immediate mode, the operand is included as part of the instruction. An example is `MOV EDX,5167812FH`, which moves the 32-bit data $5167812F_{16}$ to the EDX register. Note that the source operand in this case is in immediate mode.

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 elements:

1. **Displacement.** The 8- or 32-bit immediate data following the instruction is the displacement; 16-bit displacements can be used by inserting an address prefix before the instruction
2. **Base.** The contents of any general-purpose register can be used as a base.
3. **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.
4. **Scale.** The index register's contents can be multiplied (scaled) by a factor of 1, 2, 4, or 8. A scaled index mode is efficient for accessing arrays or structures.

Effective Address, EA = base register + (index register × scale) + displacement

The 9 memory addressing modes are a combination of these four elements. Of the 9 modes, 8 of them are executed with the same number of clock cycles because the effective address calculation is pipelined with the execution of other instructions; the mode containing base, index, and displacement elements requires one additional clock cycle.

1. **Direct mode.**The operand's effective addresses is included as part of the instruction as an 8-, 16-, or 32-bit displacement. An example is `DEC WORD PTR`

`[4000H]`.

2. **Register indirect mode.** A base or index register contains the operand's effective address. An example is `MOV EBX, [ECX]`.
3. **Base mode.** The contents of a base register is 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 is 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 is multiplied by a scaling factor (1, 2, 4, or 8), and the result 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 is 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 is multiplied by a scaling factor (1, 2, 4, 8), and the result is added to the contents of a base register to obtain the operand's effective address. An example is `MOV [ECX*4] [EDX], EAX`.
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 is multiplied by a scaling factor, and the result is added to the contents of base register and displacement to obtain the operand's effective address. An example is `MOV [ESI * 8] [EBP + 60H], ECX`.

11.3.5 80386 Instruction Set

The 80386 can execute all 16-bit instructions in real and protected modes. This is provided in order to make the 80386 software compatible with the 8086. The 80386 uses either 8- or 32-bit displacements and any register as the base or index register while executing 32-bit code. However, the 80386 uses either 8- or 16-bit displacements with the base and index registers while executing 16-bit code. The base and index registers utilized by the 80386 for 16- and 32-bit addresses are as follows:

	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

In the following, the symbol ( ) will indicate the contents of a register or a memory location. A description of some of the new 80386 instructions is given next.

1. Arithmetic Instructions

There are two new sign extension instructions beyond those of the 8086.

`CWDE` Sign-extend 16 bit contents of AX to a 32-bit double word in EAX.

`CDQ` Sign-extend a double word (32 bits) in EAX to a quadword (64 bits) in EDX:EAX

The 80386 includes all of the 8086 arithmetic instructions plus some new ones. Two

of the instructions are as follows:

Instruction	*Operation*
ADC reg32/mem32, imm32	[reg32 or mem32] ← [reg32 or mem32] + 32-bit immediate data + CF
ADC reg32/mem32, imm8	[reg32 or mem32] ← [reg32 or mem32] + 8-bit immediate data sign-extended to 32 bits + CF

Similarly, the other add instructions include the following:

```
ADC     reg32/mem32,    reg32/mem32
ADD     reg32/mem32,    imm32
ADD     reg32/mem32,    imm8
ADD     reg32/mem32,    reg32/mem32
```

The 80386 SUB/SBB instructions have the same operands as the ADD/ADC instructions.

The 80386 multiply instructions include all of the 8086 instructions plus some new ones. Some of them are listed next:

Instruction	*Operation*
IMUL EAX, reg32/mem32	EDX:EAX ← EAX * reg32 or mem32 (signed multiplication). CF and OF flags are cleared to 0 if the EDX value is 0; otherwise, they are set.
IMUL AX, reg16/mem16	DX:AX ← AX * reg16/mem16 (signed multiplication)
IMUL AL, reg8/mem8	(signed multiplication) AX ← AL * reg8/mem8
IMUL reg16, reg16/mem16,imm8	reg16 ← reg16/mem16 * (imm8 sign-extended to 16-bits) (signed multiplication). The result is the low 16 bits of product.
IMUL reg32, reg32/mem32, imm8	reg32 ← reg32/mem32 * (imm8 sign-extended to 32 bits) (signed multiplication). The result is the low 32 bits of product.

The unsigned multiplication MUL instruction has the same operands as IMUL.
The 80386 divide instructions include all of the 8086 instructions plus some new ones. Some of them are listed next:

Instruction	*Operation*
IDIV EAX, reg32/mem32	EDX:EAX ÷ reg32 or mem32 (signed division). EAX = quotient and EDX = remainder.
IDIV AL, reg8/mem8	AX ÷ reg8 or mem8 (signed division) AL = quotient and AH = remainder.
IDIV AX, reg16/mem16	DX:AX ÷ reg16 or mem16 (signed division) AX = quotient and DX = remainder.

The DIV instruction performs unsigned division, and the operation is the same as IDIV.

2. *Bit Instructions*

The six 80386 bit instructions are as follows:

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

These instructions are discussed separately next.

- **BSF** (bit scan forward) takes the form

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

BSF scans (checks) the 16-bit (word) or 32-bit (double word) number defined by *s* from right to left (bit 0 to bit 15 or bit 31). The bit number of the first 1 found is stored in *d*. If the whole 16-bit or 32-bit number is 0, the ZF flag is set to 1; Otherwise, ZF = 0. For example, consider `BSF EBX, EDX`. If (EDX) = 01241240_{16}, then after `BSF EBX, EDX`, (EBX) = 00000006_{16} and ZF = 0. The bit number 6 in EDX (contained in the second nibble of EDX) is the first 1 found when (EDX) is scanned from the right.

- **BSR** (bit scan reverse) takes the form

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

BSR scans (checks) the 16-bit or 32-bit number defined 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 0's, ZF is set to 1 and operand d is undefined; ZF is reset to 0 if a 1 is found.

- **BT** (bit test) takes the form

```
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* (bit offset) to the carry flag. Only CF is affected. If operand *s* is an immediate data, only 8 bits are allowed in the instruction. This operand is taken modulo 32 so that 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 bits or 32 bits can be determined by adding *s* (bit index) divided by the operand size (16 or 32) to the memory address of *d*. The bit within this 16- or 32-bit word is defined by *d* taken modulo the operand size (16 or 32). If d is a memory operand, the 80386 may access 4 bytes in memory starting at effective address plus 4 × [bit offset divided by 32]. As an example, consider

BT CX, DX. If (CX) = 081F and (DX) = 0021_{16}, then after BT CX, DX, because the contents of DX is 33_{10}, the bit number 1 [remainder of 33/16 = 1 of CX (value 1)] is reflected in CF and therefore, CF = 1.

- **BTC** (bit test and complement) takes the form

  BTC *d*, *s*

  where *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 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 for the BT instruction.
- **BTR** (bit test and reset) takes the form

  BTR *d*, *s*

  Where *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 0. Everything else applicable to the BT instruction also applies to BTR.
- **BTS** (bit test and set) takes the form

  BTS *d*, *s*

  BTS is the same as BTR except that the specified bit in *d* is set to 1 after the bit value of *d* defined by *s* is reflected in CF. Everything else applicable to the BT instruction also applies to BTS.

3. *Set Byte on Condition Instructions*

These instructions set a byte to 1 or reset a byte to 0 depending on any of the 16 conditions defined by the status flags. The byte may be located in memory or in a 1-byte general register. These instructions are very useful in implementing Boolean expressions in high-level languages. The general structure of these instructions is SET*cc* (set byte on condition *cc*), which sets a byte to 1 if condition *cc* is true or else resets the byte to 0.

As an example, consider SETB BL (set byte if below; CF = 1). If (BL) = 52_{16} and CF = 1, then, after this instruction is executed, (BL) = 01_{16} 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 this instruction, (BL) = 00_{16}, CF = 0, and ZF = 1; all other flags are undefined. The other SET*cc* instructions can similarly be explained.

4. *Conditional Jumps and Loops*

JECXZ disp8 jumps if [ECX] = 0; disp8 means a relative address. JECXZ tests the contents of the ECX register for zero and not the flags. If [ECX] = 0, then, after execution of the JECXZ instruction, the program branches with a signed 8-bit relative offset ($+127_{10}$ to -128_{10} with 0 being positive) defined by disp8. The JECXZ instruction is useful at the beginning of a conditional loop that terminates with a conditional loop instruction such as LOOPNE *label*. JECXZ prevents entering the loop with [ECX] = 0, which would cause the loop to execute up to 2^{32} times instead of zero times.

The loop instructions are listed next:

LOOP disp8	Decrement CX/ECX by 1 and jump if CX/ECX $\neq$ 0
LOOP/LOOPZ disp8	Decrement CX/ECX by 1 and jump if CX/ECX $\neq$ 0 or ZF = 1

LOOPNE/LOOPNZ disp8	Decrement CX/ECX by 1 and jump if CX/ECX ≠ 0 or ZF = 0

The 80386 loop instructions are similar to those of the 8086 except that if the counter is more than 16 bits, the ECX register is used as the counter.

5. *Data Transfer Instructions*

a. *Move Instructions*

The move instructions are described as follows:

MOVSX	*d*,	*s*	Move and sign-extend
MOVZX	*d*,	*s*	Move and zero-extend
	reg16,	reg8	
	reg16,	mem8	
	reg32,	reg8	
	reg32,	mem8	
	reg32,	reg16	
	reg32,	mem16	

`MOVSX` reads the contents of the effective address or register as a byte or a word from the source, 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, 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 `MOVSX BX, CL`. If (CL) = 81_{16} and (BX) = $21AF_{16}$, then, after execution of this `MOVSX`, register BX contains $FF81_{16}$ and the contents of CL do not change. Now, consider `MOVZX CX, DH`. If (CX) = $F237_{16}$ and (DH) = 85_{16}, then, after execution of this `MOVZX`, register CX contains 0085_{16} and DH contents do not change.

b. *Push and Pop Instructions*

There are new push and pop instructions in the 80386 beyond those of the 8086: `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, ECX, 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 32_{10}.

c. *Load Pointer Instructions*

There are five instructions in the load pointer instruction category: `LDS`, `LES`, `LFS`, `LGS`, and `LSS`. The 80386 can have four versions for each one 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 the 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 the reg16 or reg32 register specified.

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

6. *Flag Control Instructions*

There are two new flag control instructions in the 80386 beyond those of the 8086: `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` pops the 32 bits (double word) from the top of the stack and stores the value in EFLAGS. All flags except VM and RF are affected.

7. *Logical Instructions*

There are new logical instructions in the 80386 beyond those of the 8086:

`SHLD`	*d*,	*s*,	count	Shift left double
`SHRD`	*d*,	*s*,	count	Shift right double
	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 5 bits, so shifts from 0 to 31 can be obtained.

`SHLD` shifts the contents of *d*:*s* by the specified shift count with the result stored back into *d*; *d* is shifted to the left by the shift count with the low-order bits of *d* 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 bit shifted out of the most significant bit of *d*. If the shift count is zero, this instruction works as an NOP. For the 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` shifts the contents of *d*:*s* by the specified shift count to the right with the result stored back into *d*. The bits in *d* are shifted right by the shift count, with the high-order bits 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 an 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) = $183F_{16}$ and (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.

8. String Instructions

a. Compare String Instructions

A new 80386 instruction, `CMPS` mem32, mem32 (or `CMPSD`) beyond the compare string instructions available with the 8086 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), DS is used as the segment register unless a segment override byte is present; for the second operand (EDI), ES must be used as the segment register and cannot be overridden. ESI and EDI are incremented by 4 if DF = 0 and are decremented by 4 if DF = 1. `CMPSD` can be preceded by the `REPE` or `REPNE` prefix for block comparison. All flags are affected.

b. Load and Move String Instructions

There are new load and move instructions in the 80386 beyond those of 8086. These are `LODS` mem32 (or `LODSD`) and `MOVS` mem32, mem32 (or `MOVSD`). `LODSD` loads the (32-bit) double word from a memory location specified by DS: ESI into EAX. After the load, ESI is automatically incremented by 4 if DF = 0 and decremented by 4 if DF = 1. No flags are affected. `LODS` can be preceded by the `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 (32-bit) double word at the 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. After the move, ESI and EDI are incremented by 4 if DF = 0 and are decremented by 4 if DF = 1. `MOVS` can be preceded by the `REP` prefix for block movement of ECX double words. No flags are affected.

c. String I/O Instructions

There are new string I/O instructions in the 80386 beyond those of the 8086: `INS` mem32, `DX` (or `INSD`) and `OUTS DX`, mem32 (or `OUTSD`). `INSD` inputs 32-bit data from a port addressed by the contents 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 and decremented by 4 if DF = 1. `INSD` can be preceded by the `REP` prefix for block input of ECX double words. No flags are affected. `OUTSD` outputs 32-bit data from a memory location addressed by DS: ESI to a port addressed by the contents 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 double words.

d. Store and Scan String Instructions

There is a new 80386 `STOS` mem32 (or `STOSD`) instruction. `STOS` stores the contents of the EAX register to a double word addressed by ES and EDI. ES cannot be overridden. After the storage, 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 double words. There is also a new scan instruction, 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 double words. All flags are affected.

*e. **Table Look-Up Translation Instruction***

A modified version of the 8086 `XLAT` instruction is available in the 80386. `XLAT` mem8 (`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 a 16-bit address and by DS:EBX for the 32-bit address. DS can be overridden. No flags are affected.

*9. **High-Level Language Instructions***

Three instructions, `ENTER`, `LEAVE`, and `BOUND`, are included in the 80386. The `ENTER` imm16,imm8 instruction creates a stack frame. The data imm8 defines the nesting depth of the subroutine and can be from 0 to 31. The value 0 specifies the first subroutine only. The data imm8 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 ESP as the current stack pointer. The data imm16 specifies the number of bytes of local variables for which the stack space is to be allocated. 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. The 80386 uses BP (low 16 bits of EBP) and SP (low 16 bits of ESP) for 16-bit operands and uses EBP and ESP for 32-bit operands.

The `BOUND` instruction ensures that a signed array index is within the limits specified by a block of memory containing an upper and lower bound. The 80386 provides two forms of the `BOUND` instruction:

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

The first form is for 16-bit operands. 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) $<d_l$ or $>d_u$ 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 permits a check to determine whether or not an address of an array being accessed is within the array boundaries when the register indirect with index mode is used to access an array element. For example, the following instruction sequence will allow accessing an array with base address in ESI, the index value in EDI, and an array lenght 50 bytes; assuming the 32-bit contents of memory location, 20000100_{16} and 20000104_{16} are 0 and 49, respectively:

```
        :
BOUND   EDI, 20000100H
MOV     EAX, [EDI][ESI]
        :
```

Example 11.1

Determine the effect of each of the following 80386 instructions:

```
(a) CDQ
(b) BTC CX, BX
(c) MOVSX ECX, E7H
```

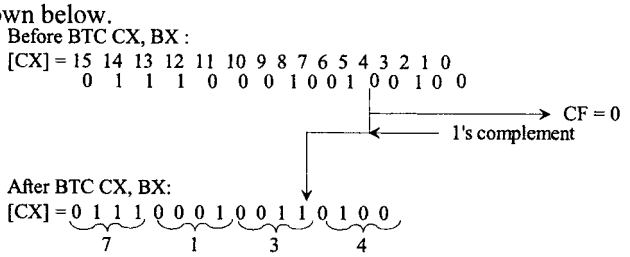
Assume (EAX) = FFFFFFFFH, (ECX) = F1257124H, (EDX) = EEEEEEEEH, and (BX) = 0004H prior to execution of each of these given instructions.

Solution

(a) After `CDQ`,

(EAX) = FFFFFFFFH
(EDX) = FFFFFFFFH

(b) After `BTC CX, BX`, bit 4 of register CX is reflected in CF and then ones complemented in CX, as is shown below.



Hence,

(CX) = 7134H
(BX) = 0004H

(c) `MOVSX ECX, E7H` copies the 8-bit data E7H into the low byte of ECX and then sign-extends to 32 bits. Therefore, after `MOVSX ECX, E7H`,

(ECX) = FFFFFFE7H

Example 11.2

Write an 80386 assembly language program to multiply a signed 8-bit number in AL by a signed 32-bit number in ECX. Assume that the segment registers are already initialized.

Solution

```
        CBW                     ;       Sign-extend byte to word
        CWDE                    ;       Sign-extend word to 32-bit
        IMUL EAX, ECX   ;       Perform singed multiplication
        HLT                     ;       Stop
```

Example 11.3

Write an 80386 assembly language program to move two columns of ten thousand 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.

Solution

```
MOV     ECX, 10000          ;       Initialize counter
MOV     BX, SOURCE_SEG      ;       Initialize DS
MOV     DS, BX              ;       register
MOV     BX, DEST_SEG        ;       Initialize ES
```

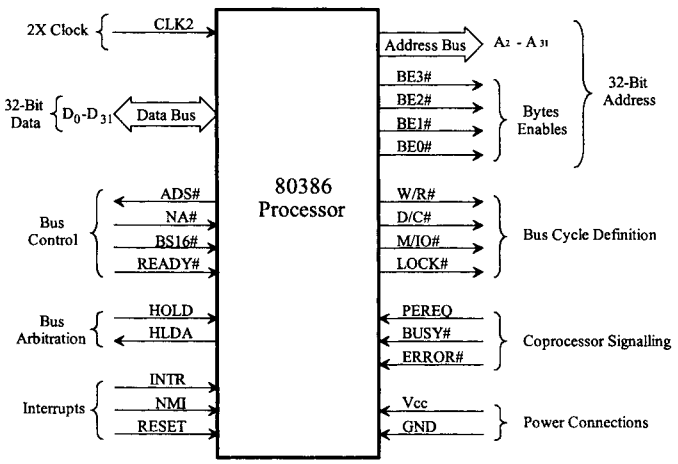


FIGURE 11.2 80386 Functional signal groups

```
MOV     ES, BX                  ;       register
MOV     ESI, SOURCE_INDX        ;       Initialize ESI
MOV     EDI, DEST_INDX          ;       Initialize EDI
CLD                             ;       Clear DF to auto-increment
REP MOVSD                       ;       MOV A (i) to
HLT                             ;       B (i) until ECX = 0
```

11.3.6 80386 Pins and Signals

The 80386 contains 132 pins in Pin Grid Array (PGA) or other packages.

Figure 11.2 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 then divided by 2 by the 80386 internally to provide the clock used for instruction execution. 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 will start executing instructions at address FFFF FFF0H. The 82384 clock generator provides system clock and reset signals.

D_0-D_{31} provides the 32-bit data bus. The 80386 can transfer 16- or 32-bit data via the data bus.

The address pins A_2-A_{31} along with the byte enable signals BE0# through BE3# are used to generate physical memory or I/O port addresses. Using the pins, the 80386 can directly address 4 gigabytes by physical memory (00000000H through FFFFFFFFH).

The byte enable outputs, BE0# through BE3# of the 80386, define which bytes of D_0-D_{31} are utilized in the current data transfer. These definations are given below:

BE0# is low when data is transferred via D_0-D_7

BE1# is low when data is transferred via D_8-D_{15}

BE2# is low when data is transferred via D_{16}-D_{23}
BE3# is low when data is transferred via D_{24}-D_{31}

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

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 , 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. Some of these bus cycles are listed below.

M/IO#	D/C#	W/R#	Bus cycle type
Low	Low	Low	INTERRUPT ACKNOWLEDGE
Low	High	Low	I/O DATA READ
Low	High	High	I/O DATA WRITE
High	Low	Low	MEMORY CODE READ
High	High	Low	MEMORY DATA READ
High	High	High	MEMORY DATA WRITE

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 indicate a valid bus cycle (W/R#, D/C#, M/IO#) and bus enable / address (BE0#-BE3#, A_2-A_{31}) signals.

When READY# input is LOW 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. BS16# input pin permits the 80386 to interface to 32- and 16-bit memory or I/O. For 16-bit memory or I/O, BS16# input pin is asserted low by an external device, the 80386 uses the low-order half (D_0-D_{15}) of the data bus corresponding to BE0# and BE1# for data transfer.

BS16# is asserted high for 32-bit memory or I/O. 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.

There are two interrupt pins or the 80386. These are INTR (maskable) and NMI (nonmaskable) pins. NMI is leading-edge sensitive, whereas 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 D_0-D_7 to identify the source of interrupt. Interrupts are serviced in a similar manner as the 8086.

11.3.7 80386 Modes

As mentioned before, the 80386 can be operated in real, protected, or virtual 8086 mode. These modes can be selected by some of the bits in the status register. 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 register. In real mode, the 80386 can directly address up to one megabyte of memory. The address lines A_2-A_{19}, BE0#-BE3# are used

by the 80386 in this mode.

The protected mode provides more memory space than is provided by the real mode. Furthermore, this mode supports on-chip memory management and protection features along with a multitasking operating system. Finally, the virtual 8086 mode permits the execution of 8086 programs, taking full advantage of the 80386 protection mechanism. In particular, the virtual the 8086 mode allows execution of 8086 operating system and application programs concurrently with the 80386 operating system and application programs.

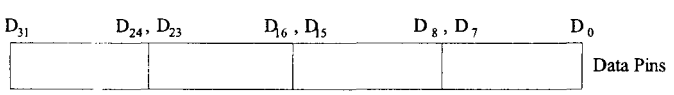
11.3.8 80386 System Design

In this section, the 80386 is interfaced to typical EPROM chips. As mentioned in the last section the 80386 address and data lines are not multiplexed. There is a total of thirty address pins (A_2-A_{31}) on the chip. A_0 and A_1 are decoded internally to generate four byte enable outputs, BE0#, BE1#, BE2#, and BE3#. In real mode, the 80386 utilizes 20-bit addresses and A_2 through A_{19} address pins are active and the address pins A_{20} through A_{31} are used in real mode at reset, high for code segment (CS)-based accesses, low for others, and always low after CS changes. In the protected mode, on the other hand, all address pins A_2 through A_{31} are active. In both modes, A_0 and A_1 are obtained internally. In all modes, the 80386 outputs on the byte enable pins to activate appropriate portions of the data to transfer byte (8-bit), word (16-bit), and double-word (32-bit) data as follows:

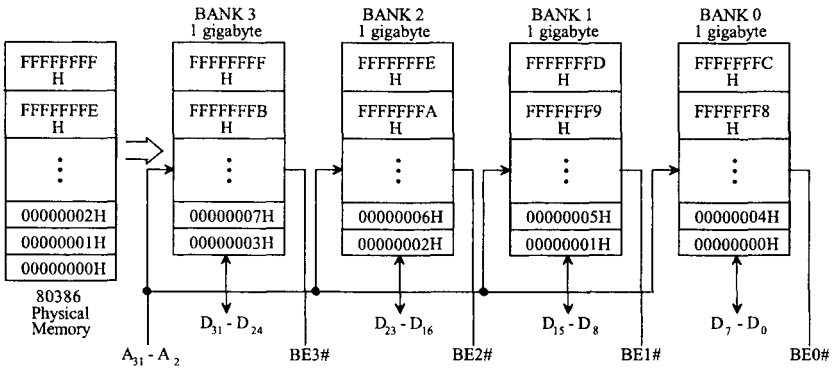
Byte Enable Pins	Data Bus
BE0#	D_0-D_7
BE1#	D_8-D_{15}
BE2#	D_{16}-D_{23}
BE3#	D_{24}-D_{31}

The 80386 supports dynamic bus sizing. This feature connects the 80386 with 32-bit or 16-bit data busses 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 D_0-D_{15} pins. 32-bit transfers take place as two consecutive 16-bit transfers over data pins D_0 through D_{15}. On the other hand, the 32-bit memory or I/O device can activate the BS16# pin HIGH to transfer data over D_0-D_{31} pins.

The 80386 address pins A_1 and A_0 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 $A_1A_0 = 00_2$ are transferred over D_0-D_7. Similarly, the contents of addresses which include 1,5,9, ..., with $A_1A_0 = 01_2$ are transferred over D_{15}-D_8. On the other hand, the contents of memory addresses 2, 6, 10, ... with $A_1A_0 = 10_2$ are transferred over D_{16}-D_{23} while contents of addresses 3, 7, 11, ... with $A_1A_0 = 11_2$ are transferred over D_{24}-D_{31}. Note that A_1A_0 is encoded from BE3# -BE0#. 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 locations 00000004H and 00000005H are written with the contents of AL and AH via D_0-D_7 and D_8-D_{15} respectively. For 32-bit transfer, the 80386 executing a MOVE instruction from an aligned address such as `MOV [00000004H], EAX`, drives all bus enable pins (BE0# -BE3#) to low and writes four bytes to memory locations 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 multiple 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 into location 00000003H (bank 3) from AL in the first bus cycle. The 80386 then drives BE0# to low in the second bus cycle and writes into location 00000004H (bank 0) from AH. 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 low 16-bits of EAX into addresses 00000002H and 00000003H from banks 2 and 3 respectively. In the second cycle, the 80386 enables BE0# and BE1# to low and then writes the contents of upper 16-bits of EAX into addresses 00000004H and 00000005H.

In the following, design concepts associated with the 80386's interface to memory will be discussed. The 80386 device will use 128 Kbyte, 32-bit wide memory. Four 27C256's (32 K x 8 HCMOS EPROMs) are used.

Since the 27C256 chip is 32K x 8 chip, the 80386 address lines A_2-A_{16} are used for addressing the 27C256's. The 80386 M/IO#, D/C#, W/R#, and BE0#-BE3# are also used. Figure 11.3 shows a simplified 80386 - 27C256 interface.

In figure 11.3, A_1 A_0, BE3#-BE0#, D/C#, and ADS# pins of the 80386 are used to generate four byte enable signals, $\overline{E0}$, $\overline{E1}$, $\overline{E2}$, and $\overline{E3}$.

The 80386 outputs low on ADS# (Address status) pin to indicate valid bus cycle (W/R#, D/C#, M/IO#) and address (BE0# -BE3#) signals.

The 80386 A_1 and A_0 bits (obtained internally) indicate which portion of the data bus will be used to transfer data. For example, A_1 A_0 = 11 means that contents of addresses such as 00000003H, 00000007H, ... will be used by the 80386 to transfer data via its D_{31}-D_{24} pins. BE3#-BE0# and D/C# are used to produce the byte enable signals which

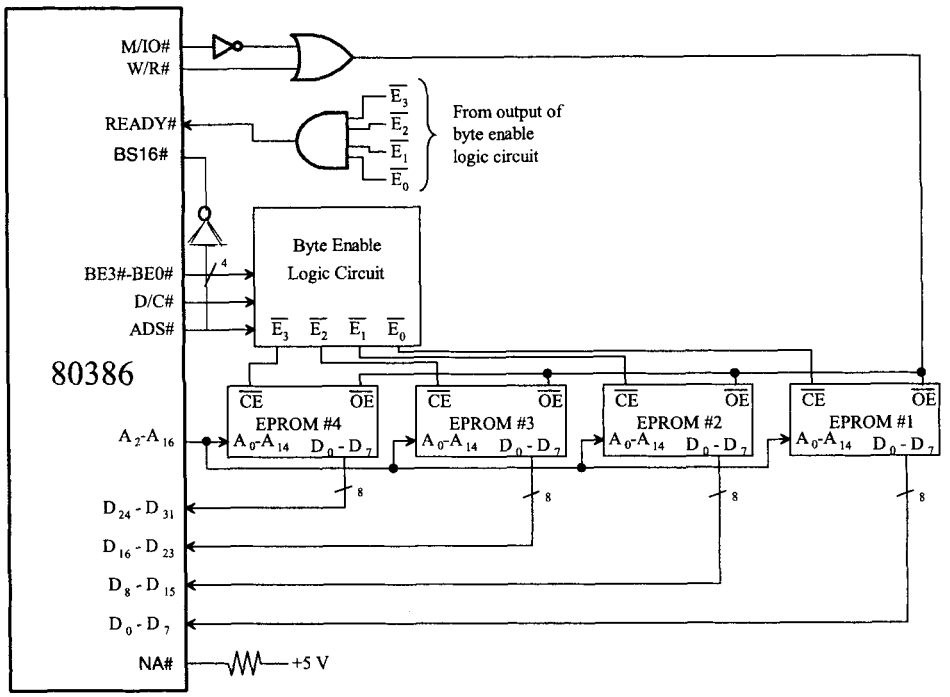


FIGURE 11.3 80386/27C256 Interface.

are connected to the $\overline{CE}$ pin of the appropriate EPROM. The inverted M/IO# is logically ORed with the W/R# pin. The output of this OR gate is connected to the $\overline{OE}$ pin of all four EPROM's.

$\overline{E0}$, $\overline{E1}$, $\overline{E2}$, and $\overline{E3}$ are ANDed and connected to the READY# pin. When the READY# pin is asserted LOW, the 80386 latches or reads data. Until READY# pin is asserted LOW by the external device, the 80386 inserts wait states. One must ensure that the data is ready before READY# is asserted. The BS16# is asserted HIGH by connecting it to inverted ADS# to indicate 32-bit memory. NA# is connected to +5 V to disable pipelining.

The memory map can be determined as follows:

EPROM#1 :

$$\underbrace{A_{31}\,A_{30}\,\cdots\,A_{17}}_{\text{Don't cares}\atop\text{Assume zeros}}\;\underbrace{A_{16}\,\cdots\,A_{2}}_{\text{all zeros}\atop\text{to ones}}\;\overset{\uparrow}{A_1}\;\overset{\uparrow}{A_0}\;\begin{matrix}0\;0\end{matrix}$$

= 00000000H, 00000004H, ... , 0001FFFCH

Similarly, the memory maps for other EPROMs are :
EPROM#2: 00000001H, 00000005H, ... , 0001FFFDH
EPROM#3: 00000002H, 00000006H, ... , 0001FFFEH
EPROM#4: 00000003H, 00000007H, ... , 0001FFFFH

11.3.9 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. Eight-bit I/O devices can be connected to any of the four 8-bit sections of the data bus. For efficient operation, 32-bit I/O devices should be assigned to addresses that are even multiples of four. For standard I/O, the 80386 includes there 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
```

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) ← SI ± 2
For 32-bit:     INSD,           (ES:EDI) ← ((DX))
                                EDI ← EDI ± 4
                OUTSD,          ((DX)) ← (ES:ESI)
                                ESI ← ESI ± 4
```

11.4 Intel 80486 Microprocessor

The Intel 80486 is an enhanced 80386 microprocessor with on-chip floating-point hardware.

11.4.1 Intel 80486/80386 Comparison

Table 11.2 compares the basic features of the 80486 with those of the 80386.

11.4.2 Special Features of the 80486

The 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 an external cache and associated control logic.

TABLE 11.2 80386 vs. 80486

Characteristic	80386	80486
Introduced in	1985; 386SX in 1988	1989
Main features	Adds paging 32-bit extension, on chip address translation, and greater speed than 8086. 32-bit microprocessor	Adds on-chip cache, floating-point unit, and greater speed than 386. 32-bit microprocessor.
Data bus size accommodated	16-, 32-bit	8-, 16-, 32-bit
On-chip Cache	No; Can be interfaced externally	Yes
Address bus size	32-bit	32-bit
On-chip transistors	275,000	1.2 million
Directly addressable memory	4 Gigabytes	4 Gigabytes
Virtual memory size	64 Terabytes	64 Terabytes
Clock	25 MHz to 50 MHz	25 MHz to 100 MHz
Pins	100 for 80386SX; 168 for other 80386's	168
Address and data buses	non-multiplexed	non-multiplexed
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
Address	Defined by A_2-A_{31}; BE0#-BE3#	Same as the 80386
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.
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 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.
On-chip memory management hardware	Yes	Yes
Operating modes: Real, Protected, and Virtual 8086 modes	Yes. Does not support maximum or minimum modes like the 8086	Same as the 80386
On-chip floating-point hardware	No	Yes
Instructions	129 including the floating-point instrucions where the 80386 is interfaced to the 80387	All 80386 instructions including the floating-point instructions for the on-chip floating-point hardware plus six new instructions

The 80486 is object code compatible with the 8086, 8088, 80186, 80286, and 80386 processors. 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.

The 80486 has an internal 8 Kbyte cache memory. This provides fast access to recently used instructions and data. The internal write-through cache can hold 8 Kbytes of data or instructions. The on-chip floating-point unit performs floating-point operations on the 32-, 64-, and 80- bit arithmetic formats specified in the IEEE standard and is object code compatible with the 8087, 80287, and 80387 coprocessors. The fetching, decoding, execution, and address translation of instructions is overlapped within the 80486 processor using instruction pipelining. This allows a continuous execution rate of one clock cycle per instruction for most instructions.

Like the 80386, 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 the default operand size is 16 bits. The main purpose of real mode is to set up the processor for protected mode.

Protected mode, or 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 8086, 80286, and 386 processor software can be run under the 80486 processor's hardware-assisted protection mechanism.

Virtual 8086 mode is a submode for protected 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 real mode because virtual 8086 mode can simultaneously execute the 80486 operating system and both 8086 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 restart 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 80386 has very few differences with the 80486 processor. The 80486 processor defines new bits in the EFLAGS, CR0, and CR3 registers. In the 80386 processor, these bits were reserved, so the new architectural features should be a compatibility issue.

11.4.3 80486 New Instructions Beyond Those of the 80386

There are six basic instructions plus floating-point instructions added to the 80486 instruction set beyond those of the 80386 instruction set as follows:

1. Three New Application Instructions
   - `BSWAP`
   - `XADD`
   - `CMPXCHG`
2. Three New System Instructions

- INVD
- WBINVD
- INVLPG

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) because it has the on-chip floating-point hardware.

The three new application instructions included with the 80486 are BSWAP reg32; XADD dest, source; and CMPXCHG dest, source. BSWAP reg32 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 31–24 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 a 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 in which 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 0 being the most significant byte. The BSWAP instruction speeds up execution of decimal arithmetic by operating on four digits at a time.

XADD dest, source has the form

XADD	dest,	source
	reg8/mem8,	reg8
	reg16/mem16,	reg16
	reg32/mem32,	reg32

The XADD dest, source instruction 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.

CMPXCHG dest, source has the form:

CMPXCHG	dest,	source
	reg8/mem8,	reg8
	reg16/mem16,	reg16
	reg32/mem32,	reg32

The CMPXCHG instruction compares the (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 AL,AX or EAX. 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.

11.5 Intel Pentium Microprocessor

Table 11.3 summarizes the fundamental differences between the basic features of 486 and Pentium families. Microprocessors have served largely separate markets and purposes: business PCs and engineering workstations. The PCs have used Microsoft's DOS and Windows operating systems whereas the workstations have used various features of UNIX.

TABLE 11.3 Basic Differences Between 80486 and Pentium Processor

Feature	486 Processor	Pentium Processor
Clock	25 to 100 MHz	60 to 233 MHz
Address and data buses	32-bit data bus 32-bit address bus	64-bit data bus 32-bit address bus
Pipeline model	Single	Dual
Internal cache	8K for both data and instruction	8k for data and 8k for instruction
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

The PCs have not been utilized in the workstation market because of their relatively modest performance, especially with regard to complicated graphics display and floating-point calculations. Workstations have been kept out of the PC market partially because of their high prices and hard-to-use system software.

The Pentium has brought 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 8086/80386 systems. BICMOS(Bipolar and CMOS) technology is used for the Pentium.

The Pentium processor has three modes of operation; real-address mode (also called "real mode"), protected mode, and system management mode. The mode determines which instructions and architecture features are accessible. In *real-address mode,* the Pentium processor runs programs written for 8086 or for the real-address mode of an 80386 or 80486.

The architecture of the Pentium processor in this mode is identical to that of the 8086 microprocessor. In *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 (v86) 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 also provided with a *system management mode* (SMM) similar to the one used in the 80486SL, which allows to design for low power usage. SMM is entered through activation of an external interrupt pin (system management interrupt, SMI#). 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 has had problems with. The first version of the Intel 80386 had a math flaw that Intel quickly fixed before there were any complaints. Some experts feel that Intel should have acknowledged the math problem in the Pentium when it was first discovered and then have 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 in the Pentium chip was first detected.

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 the fifteenth significant

decimal digits. This means that in a result such as 5.78346, the last three digits could be incorrect. For example, the correct answer for the operation 4,195,835 - (4,195,835 + 3,145,727) + (3,145,727) is zero. The Pentium provided a wrong answer of 256. IBM claimed this problem can occur once every 24 days. Intel eventually fixed the division flaw problem in the Pentium.

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 that 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 hardware 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).

There are 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, whereas 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.

11.5.1 Pentium Registers

The Pentium processor includes the same registers as the 80486. Three new system flags are added to the 32-bit EFLAGS register.

11.5.2 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. One of the new application instruction is the `CMPXCHG8B`. As an example, `CMPXCHG8B` reg64 or mem64 compares the 64-bit value in EDX:EAX with the 64 bit contents of reg64 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.

Pentium floating-point instructions execute much faster than those of the 80486 instructions.

For example, a 66-MHz Pentium microprocessor provides about three times the floating-point performance of a 66-MHz Intel 80486 DX2 microprocessor.

11.5.3 Pentium versus 80486: Basic Differences in Registers, Paging, Stack Operations, and Exceptions

Registers of the Pentium Processor versus Those of the 80486

This section discusses the basic differences between the Pentium and 80486 control, debug, and test registers.

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 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 break 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 has been redesigned in the Pentium processor.

Paging

The Pentium processor provides an extension to the memory management/paging functions of the 80486 to support larger page sizes.

Stack Operations

The Pentium, 80486, and 80386 microprocessors push a different value of SP on the stack for a PUSH instruction than does the 8086. The 32-bit processors push the value of the SP before it is decremented whereas the 8086 pushes the value of the SP after it is decremented.

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.

11.5.4 Pentium 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 double words in memory-mapped I/O, 32-bit ports in standard I/O should be aligned to addresses that 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.

11.5.5 Applications with the Pentium

The performance of the Pentium's floating-point unit (FPU) makes it appropriate for wide areas of numeric applications:

- 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.
- Many minicomputer 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.
- 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.
- 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 include robotics, navigation, data acquisition, and process control.

11.5.6 Pentium versus Pentium Pro

The Pentium was first introduced by Intel in March 1993, and the Pentium Pro was introduced in November 1995. The Pentium processor provides pipelined superscalar architecture. The Pentium processor's pipelined implementation uses five stages to extract high throughput and the Pentium Pro utilizes 12-stage, superpipelined implementation, trading less work per pipestage for more stages. The Pentium Pro processor reduced its pipestage time by 33% compared with a Pentium processor, which means the Pentium Pro processor can have a 33% higher clock speed than a Pentium processor and still be equally easy to produce from a semiconductor manufacturing process. A 200-MHz Pentium Pro is always faster than a 200-MHz Pentium for 32-bit applications such as computer-aided design (CAD), 3-D graphics, and multimedia applications.

The Pentium processor's superscalar architecture, with its ability to execute two instructions per clock, was difficult to exceed without a new approach. The new approach used by the Pentium Pro processor removes the constraint of linear instruction sequencing between the traditional "fetch" and "execute" phases, and opens up a wide instruction pool. This approach allows the "execute" phase of the Pentium Pro processor to have much more visibility into the program's instruction stream so that better scheduling may take place. This allows instructions to be started in any order but always be completed in the original program order.

Microprocessor speeds have increased tremendously over the past 10 years, but the speed of the main memory devices has only increased by 60 percent. This increasing

TABLE 11.4 Pentium vs. Pentium Pro

Pentium	Pentium Pro
First introduced March 1993	Introduced November 1995
2 instructions per clock cycle	3 instructions per clock cycle
Primary cache of 16K	Primary cache of 16K
Current clock speeds of 100, 120, 133, 150, 166, 200, and 233 MHz	Current clock speeds 166, 180, 200 MHz
More silicon is needed to produce the chip	Tighter design reduces silicon needed and makes chip faster (shorter distances between transistors)
Designed for operating systems written in 16-bit code	Designed for operating systems written in 32-bit code.

memory latency, relative to the microprocessor speed, is a fundamental problem that the Pentium Pro is designed to solve. The Pentium Pro processor "looks ahead" into its instruction pool at subsequent instructions and will do useful work rather than be stalled. The Pentium Pro executes instructions depending on their readiness to execute and not on their original program order. In summary, it is the unique combination of improved branch prediction, choosing the best order, and executing the instructions in the preferred order that enables the Pentium Pro processor to improve program execution over the Pentium processor. This unique combination is called "dynamic execution."

The Pentium Pro does a great job running some operating systems such as Windows NT or Unix. The first release of Windows 95 contains a significant amount of 16-bit code in the graphics subsystem. This causes operations on the Pentium Pro to be serialized instead of taking advantage of the dynamic execution architecture. Nevertheless, the Pentium Pro is up to 30% faster than the fastest Pentium in 32-bit applications. Table 11.4 compares the basic features the Pentium with those of the Pentium Pro.

11.5.7 Pentium II / Celeron / Pentium II Xeon™ / Pentium III / Pentium 4

The 32-bit Pentium II processor is Intel's latest addition to the Pentium line of microprocessors, which originated form the widely cloned 80x86 line. It basically takes attributes of the Pentium Pro processor plus the capabilities of MMX technology to yield processor speeds of 333, 300, 266, and 233 MHz. The Pentium II processor uses 0.25 micron technology (this refers to the width of the circuit lines on the silicon) to allow increased core frequencies and reduce power consumption. The Pentium II processor took advantage of four new technologies to achieve its performance ratings:

- Dual Independent Bus Architecture (DIB)
- Dynamic Execution
- Intel MMX Technology
- Single-Edge-Contact Cartridge

DIB was first implemented in the Pentium Pro processor to address bandwidth limitations. The DIB architecture consists of two independent buses, an L2 cache bus and a system bus, to offer three times the bandwidth performance of single bus architecture processors. The Pentium II processor can access data from both buses simultaneously to accelerate the flow of information within the system.

Dynamic execution was also first implemented in the Pentium Pro processor. It consists of three processing techniques to improve the efficiency of executing instructions.

These techniques include multiple branch prediction, data flow analysis, and speculative

execution. Multiple branch prediction uses an algorithm to determine the next instruction to be executed following a jump in the instruction flow. With data flow analysis, the processor determines the optimum sequence for processing a program after looking at software instructions to see if they are dependent on other instructions. Speculative execution increases the rate of execution by executing instructions ahead of the program counter that are likely to be needed.

MMX (**m**atrix **m**ath **e**xtensions) technology is Intel's greatest enhancement to its microprocessor architecture. MMX technology is intended for efficient multimedia and communications operations. To achieve this, 57 new instructions have been added to manipulate and process video, audio, and graphical data more efficiently. These instructions support single-instruction multiple-data (SIMD) techniques, which enable one instruction to perform the same function on multiple pieces of data. Programs written using the new instructions significantly enhance the capabilities of Pentium II.

The final feature in Intel's Pentium II processor is single-edge-contact (SEC) packaging. In this packaging arrangement, the core and L2 cache are fully enclosed in a plastic and metal cartridge. The components are surface mounted directly to a substrate inside the cartridge to enable high-frequency operation.

Intel Celeron processor utilizes Pentium II as core .The Celeron processor family includes: 333 MHz, 300A MHz, 300 MHz, and 266 MHz processors.The Celeron 266 MHz and 300 MHz processors do not contain any level 2 cache. But the Celeron 300A MHz and 333 MHz processors incorporate an integrated L2 cache. All Celeron processors are based on Intel's 0.25 micron CMOS technology. The Celeron processor is designed for inexpensive or "Basic PC" desktop systems and can run Windows 98. The Celeron processor offers good floating-point (3D geometry calculations) and multimedia (both video and audio) performance.

The Pentium II Xeon processor contains large, fast caches to transfer data at super high speed through the processor core. The processor can run at either 400 MHz or 450 MHz. The Pentium II Xeon is designed for any mid-range or higher Intel-based server or workstation.The 450 MHz Pentium II Xeon can be used in dual-processor (two-way) workstations and servers. The 450 MHz Pentium II Xeon processor with four-way servers is expected to be available in the future.

The Pentium III operates at 450 MHz and 500 MHz. It is designed for desktop PCs. The Pentium III enhances the multimedia capabilities of the PC, including full screen video and graphics. Pentium III Xeon processors run at 500 MHz and 550 MHz. They are designed for mid-range and higher Internet-based servers and workstations. It is compatible with Pentium II Xeon processor-based platforms. Pentium III Xeon is also designed for demanding workstation applications such as 3-D visualization, digital content creation, and dynamic Internet content development. Pentium III-based systems can run applications on Microsoft Windows NT or UNIX-based environments. The Pentium III Xeon is available in a number of L2 cache versions such as 512-Kbytes, 1-Mbyte, or 2-Mbytes (500 MHz); 512 Kbytes (550 MHz) to satisfy a variety of Internet application requirements.

The Intel Pentium 4 is an enhanced Pentium III processor. It is currently available at 1.30, 1.40, 1.50, and 1.70 GHz. The chip's all-new internal design contains Intel NetBurst™ micro-architecture. This provides the Pentium 4 with hyper pipelined technology (which doubles the pipeline depth to 20 stages), a rapid execution engine (which pushes the processor's ALUs to twice the core frequency), and 400 MHz system bus. The Pentium 4 contains 144 new instructions. Furthermore, inclusion of an improved Advanced Dynamic Execution and an improved floating point pushes data efficiently through the pipeline.

This enhances digital audio, digital video and 3D graphics. Along with other features such as streaming SIMD Extensions 2 (SSE2) that extends MMX™ technology, the Pentium 4 gives the advanced technology to get the most out of the Internet. Finally, the Pentium 4 offers high performance when networking multiple PCs, or when attaching Pentium 4 based PC to home consumer electronic systems and new peripherals.

11.6 Merced/IA-64

Intel and Hewlett-Packard recently announced a 64-bit microprocessor called "Merced" and also known as "Intel Architecture–64" (IA-64) or Itanium. The microprocessor is not an extension of Intel's 32-bit 80x86 or Pentium series processors, nor is it an evolution of HP's 64-bit RISC architecture. IA-64 is a new design that will implement innovative forward-looking features to help improve parallel instruction processing: that is, long instruction words, instruction prediction, branch elimination, and speculative loading. These techniques are not necessarily new concepts, but they are implemented in ways that are much more efficient.

An 80x86 instruction varies in length from 8 to 108 bits, and the microprocessor spends time and work decoding each instruction while scanning for the instruction boundaries during execution. In addition, Pentium processors frantically try to reorder instructions and group them so that two instructions can be fed into two processing pipelines simultaneously. Although improving performance, this approach is still rather ineffective and has a high cost of logic circuitry in the chip.

The IA-64 packs three instructions into a single 128-bit bundle—something Intel calls "explicitly parallel instruction computing" (EPIC). During compilation of a program, the compiler explicitly tells the microprocessor inside the 128-bit packet which of the instructions can be executed in parallel. Hence, the microprocessor does not need to scramble at run-time to discover and reorder instructions for parallel execution because all of this has already been done at compilation. While trying to keep the instruction pipeline full, 80x86 or Pentium family processors try to predict which way branches will take place and speculatively execute instructions along the predicted path. In case of wrong guesses, the microprocessor must discard the speculative results, flush the pipelines, and reload the correct instructions into the pipe. This results in a large loss of microprocessor cycles.

In dealing with branch prediction, the IA-64 puts the burden on the compiler. Wherever practical, the compiler inserts flags into the instruction packets to mark separate paths from a branch instruction. These flags, known as "predicates," allow the microprocessor to funnel instructions for a specific branch into a pipe and execute each branch separately and simultaneously. This effectively lets the microprocessor process different paths of a branch at the same time, then discard the results of the path it does not need.

One drawback of the 80x86 processor series is the fact that data is not fetched from memory until the microprocessor needs it and calls for it. The IA-64 implements speculative loading, which allows the memory and I/O devices to be delivering data to the microprocessor before the processor actually needs it, eliminating some of the delays the 80x86 processor incurs while waiting for data to appear on the bus.

During compilation of a program, the compiler scans the source code and when it sees an upcoming load instruction, removes it and inserts a speculative load instruction a few cycles ahead of it. In this manner, the IA-64 is able to continue executing code while minimizing delay time that the memory or I/O devices inherently incur.

11.7 Overview of Motorola 32- and 64-bit Microprocessors

This section provides an overview of the state-of-the-art in Motorola's microprocessors. Motorola's 32-bit microprocessors based on 68HC000 architecture include the MC68020, MC68030, MC68040, and MC68060. Table 11.5 compares the basic features of some of these microprocessors with the 68HC000.

The PowerPC family of microprocessors were jointly developed by Motorola, IBM, and Apple. The PowerPC family contains both 32- and 64-bit microprocessors. One of the noteworthy feature of the PowerPC is that it is the first top-of-the-line microprocessor to include an on-chip real-time clock (RTC). The RTC is common in single-chip microcomputers rather than microprocessors. The PowerPC is the first microprocessor to implement this on-chip feature, which makes it easier to satisfy the requirements of time-keeping for task switching and calendar date of modern multitasking operating systems. The PowerPC microprocessor supports both the Power Mac and standard PCs. The PowerPC family is designed using RISC architecture

11.7.1 Motorola MC68020

The MC68020 is Motorola's first 32-bit microprocessor. The design of the 68020 is based on the 68HC000. The 68020 can perform a normal read or write cycle in 3 clock cycles without wait states as compared to the 68HC000, which completes a read or write operation in 4 clock cycles without wait states. As far as the addressing modes are concerned, the 68020 includes new modes beyond those of the 68HC000. Some of these modes are scaled indexing, larger displacements, and memory indirection. Furthermore, several new instructions are added to the 68020 instruction set, including the following:

- Bit field instructions are provided for manipulating a string of consecutive bits with a variable length from 1 to 32 bits.

TABLE 11.5 Motorola MC68HC000 vs. MC68020/68030/68040

	MC68HC000	MC68020	MC68030	MC68040
Comparable Clock Speed	33MHz (4MHz min)*	33 MHz (8 MHz min.)*	33 MHz (8 MHz min.)*	33 MHz (8 MHz min.)*
Pins	64, 68	114	118	118
Address Bus	24-bit	32-bit	32-bit	32-bit
Addressing Modes	14	18	18	18
Maximum Memory	16 Megabytes	4 Gigabytes	4 Gigabytes	4 Gigabytes
Memory Management	NO	By interfacing the 68851 MMU chip	On-chip MMU	On-chip MMU
Cache (on chip)	NO	Instruction cache	Instruction and data cache	Instruction and data cache
Floating Point	NO	By interfacing 68881/68882 floating-point coprocessor chip	By interfacing 68881/68882 floating-point coprocessor chip	On-chip floating point hardware
Total Instructions	56	101	103	103 plus floating- point instructions
ALU size	One 16-bit ALU	Three 32-bit ALU's	Three 32-bit ALU's	Three 32-bit ALU's

*Higher clock speeds available

- Two new instructions are used to perform conversions between packed BCD and ASCII or EBCDIC digits. Note that a packed BCD is a byte containing two BCD digits.
- Enhanced 68000 array-range checking (`CHK2`) and compare (`CMP2`) instructions are included. `CHK2` includes lower and upper bound checking; `CMP2` compares a number with lower and upper values and affects flags accordingly.
- Two advanced instructions, namely, `CALLM` and `RTM`, are included to support modular programming.
- Two compare and swap instructions (`CAS` and `CAS2`) are provided to support multiprocessor systems.

A comparison of the differences between the 68020 and 68HC000 will be provided later in this section.

The 68030 and 68040 are two enhanced versions of the 68020. The 68030 retains most of the 68020 features. It is a virtual memory microprocessor containing an on-chip MMU (memory management unit). The 68040 expands the 68030 on-chip memory management logic to two units: one for instruction fetch and one for data access. This speeds up the 68040's execution time by performing logical-to-physical-address translation in parallel. The on-chip floating-point capability of the 68040 provides it with both integer and floating-point arithmetic operations at a high speed. All 68HC000 programs written in assembly language in user mode will run on the 68020/68030 or 68040. The 68030 and 68040 support all 68020 instructions except `CALLM` and `RTM`. Let us now focus on the 68020 microprocessor in more detail.

MC68020 Functional Characteristics

The MC68020 is designed to execute all user object code written for the 68HC000. Like the 68HC000, it is manufactured using HCMOS technology. The 68020 consumes a maximum of 1.75 W. It contains 200,000 transistors on a 3/8" piece of silicon. The chip is packaged in a square (1.345" × 1.345") pin grid array (PGA) and other packages. It contains 169 pins (114 pins used) arranged in a 13 × 13 matrix.

The processor speed of the 68020 can be 12.5, 16.67, 20, 25, or 33 MHz. The chip must be operated from a minimum frequency of 8 MHz. Like the 68HC000, it does not have any on-chip clock generation circuitry. The 68020 contains 18 addressing modes and 101 instructions. All addressing modes and instructions of the 68HC000 are included in the 68020. The 68020 supports coprocessors such as the MC68881/MC68882 floating-point and MC68851 MMU coprocessors.

These and other functional characteristics of the 68020 are compared with the 68HC000 in Table 11.6. Some of the 68020 characteristics in Table 11.6 will now be explained.

- 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). ISP can be used to simplify and speed up task switching for operating systems.
- The vector base register (VBR) is used in interrupt vector computation. For example, in the 68HC000, the interrupt vector address is obtained by using VBR + 4 × 8-bit vector.

TABLE 11.6 Functional Characteristics, MC68HC000 vs. MC68HC020

Characteristic	*68HC000*	*68020*
Technology	HCMOS	HCMOS
Number of pins	64, 68	169 (13 × 13 matrix; pins come out at bottom of chip; 114 pins currently used.)
Control unit	Nanomemory (two-level memory)	Nanomemory (two-level memory)
Clock	6 MHz, 10 MHz, 12.5 MHz, 16.67 MHz, 20 MHz, 25 MHz, 33 MHz (4 MHz minimum requirement).	12.5 MHz, 16.67 MHz, 20 MHz, 25 MHz, 33 MHz (8 MHz minimum requirement).
ALU	One 16-bit ALU	Three 32-bit ALUs
Address bus size	24 bits with A_0 encoded from $\overline{UDS}$ and $\overline{LDS}$.	32 bits with no encoding of A_0 is required.
Data bus size	The 68HC000 can only be configured as 16-bit memory (two 8-bit chips) via D_0-D_7 for odd addresses and D_8-D_{15} for even addresses during byte transfers; for word and long word, uses D_0-D_{15}. The I/O can be configured as byte (one 8-bit word) or 16-bit (two 8-bit words).	The 68020 can be configured as 8-bit memory (a single 8-bit chip) via D_{31}-D_{24} pins or 16-bit memory (two 8-bit chips) via D_{31} - D_{16} pins or 32-bit memory (four 8-bit chips) via D_{31}-D_0 pins. I/O can be configured as 8-bit or 16-bit or 32-bit.
Instructions and data access	Instructions must be at even addresses for .B, .W, and .L. Byte data can be accessed at either even or odd addresses while word and long word data must be at even addresses.	Instructions must be accessed at even addresses for .B, .W, and .L; data accesses can be at either even or odd addresses for .B, .W, .L.
Instruction cache	None	128K 16-bit word cache. At start of an instruction fetch, the 68020 always outputs LOW on $\overline{ECS}$ (early cycle start) pin and accesses the cache. If instruction is found in the cache, the 68020 inhibits outputting LOW on $\overline{AS}$ pin; otherwise, the 68020 sends LOW on $\overline{AS}$ pin and reads instruction from 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 CACR (cache control register)

Addressing modes	14	18
Instruction set	56 instructions	101 instructions
Barrel shifter	No	Yes. For fast-shift operations.
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
Coprocessor interface	Emulated in software; that is, by writing subroutines, coprocessor functions such as floating-point arithmetic can be obtained.	Can be directly interfaced to coprocessor chips, and coprocessor functions such as floating-point arithmetic can be obtained via 68020 instructions.
FC0, FC1, FC2 pins	FC0, FC1, FC2 = 111 means interrupt acknowledge.	FC0, FC1, FC2 = 111 means CPU space cycle; then by decoding A16-A19, one can obtain breakpoints, coprocessor functions, and interrupt acknowledge.

- 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 such instructions such as `MOVEC A2,SFC`. The upper 29 bits of SFC are assumed to be zero. 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 an external decoder, the desired source memory location addressed by [A0] can be accessed.
- The new addressing modes in the 68020 include scaled indexing, 32-bit displacements, and memory indirection. To illustrate the concept of scaling, consider moving the contents of memory location 50_{10} to A1. Using the 68000, the following instruction sequence will accomplish this

```
MOVEA.W #10, A0
MOVE.W #10, D0
ASL #2, D0
MOVEA.L 0 (A0, D0.W), A1
```

The scaled indexing mode can be used with the 68020 to perform the same as follows:

```
MOVEA.W #10, A0
MOVE.W #10, D0
MOVEA.L (0, A0, D0.W * 4), A1
```

Note that [D0] here is scaled by 4. Scaling by 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 A/D converters and eliminate wasting main memory space when the A/D converter is not 32 bits wide. For example, if the A/D is 12 bits wide, then the instruction `BFEEXTU $22320000 {2:13},D0` will input bits 2-13 of memory location $22320000 into D0. Note that $22320000 is the memory-mapped port, where the 12-bit A/D is connected at bits 2–13. The next A/D can be connected at bits 14–25, and so on.

- 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 or coprocessor instructions, or during an interrupt acknowledge cycle. The 68020 then decodes A_{16}–A_{19} to determine the type of CPU space. For example, FC2, FC1, FC0 = 111 and A_{19}, A_{18}, A_{17}, A_{16} = 0010 mean coprocessor instruction.
- For performing floating-point operation, the 68HC000 user must write subroutines using the 68HC000 instruction set. The floating-point capability in the 68020 can be obtained by connecting a floating-point coprocessor chip such as the Motorola 68881. The 68020 has two coprocessor chips: the 68881 (floating point) and the 68851 (memory management). The 68020 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. Hence, the 68020 can read the response register and obtain the result of the floating-point add from the appropriate locations .
- The 68HC000 $\overline{\text{DTACK}}$ pin is replaced by two pins on the 68020: $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$. These pins are defined as follows:

$\overline{\text{DSACK0}}$	$\overline{\text{DSACK0}}$	Device Size
0	0	32-bit device
0	1	16-bit device
1	0	8-bit device
1	1	Data not ready; insert wait states

The 68020 can be configured as a byte, 16-bit, or 32-bit memory system. As a byte memory system, the data pins of a single 8-bit memory containing all addresses in increments of one can be connected to the 68020 D_{31}–D_{24} pins. All data transfers occur via pins D_{31}–D_{24}. The byte memory chip informs the 68020 of its size by activating $\overline{\text{DSACK1}}$ = 1 and $\overline{\text{DSACK0}}$ = 0 so that the 68020 transfers data via its D_{31}–D_{24} pins. For byte instructions, one byte is transferred via these pins; for word (16-bit) instructions, two consecutive bytes are transferred via these pins; for long word (32-bit) instructions, four consecutive bytes are transferred via these pins.

When the 68020 is configured as a word (16-bit) memory system, two byte memory chips are interfaced to the 68020 via its D_{31}– D_{16} pins. The data pins of the byte memory chips containing even and odd addresses are connected to the 68020 pins D_{31}–D_{24} and D_{23}-D_{16}, respectively. The memory chips inform the 68020 of the 16-bit memory configuration by activating $\overline{\text{DSACK1}}$ = 0 and $\overline{\text{DSACK0}}$ = 1. The 68020 then uses D_{31}–D_{16} to transfer data for byte, word, or long word instructions. For byte instructions, one byte is transferred via pins D_{31}–D_{24} or D_{23}–D_{16} depending on whether the address is even or odd. For word instructions, the contents of both even and odd addresses are transferred via pins D_{31}–D_{16} with even-address byte via D_{31}-D_{24} pins and odd-addressed byte via D_{23}-D_{16} pins;

for long word instructions, four consecutive bytes are transferred via pins D_{31}–D_{16} with the contents of even addresses via pins D_{31}–D_{16} using additional cycles. Data transfer can be aligned or misaligned. For 16-bit memory systems, a word or long word instruction with data transfer starting at an even address is called an "aligned transfer." For example, the instruction `MOVE.W D1,$30000000` will store one data byte at the even address $30000000 via pins D_{31}–D_{24} and one data byte at the odd address $30000001 via pins D_{23}–D_{16} in one cycle. On the other hand, `MOVE.W D0,$30000001` is a misaligned transfer. The 68020 transfers one byte to $30000001 via pins D_{23}–D_{16} in the first cycle and another byte to $30000002 via pins D_{31}–D_{24} in the second cycle. Thus, the misaligned transfer for word instruction takes two cycles in a 16-bit memory configuration. For 32-bit transfers, `MOVE.L D1,$30000000` is an aligned transfer. During the first cycle, the 68020 transfers 8-bit contents of the highest byte of D0 to $30000000 via pins D_{31}–D_{24}, and the next 8-bit contents of D0 to $30000001 via pins D_{23}–D_{16}. During the second cycle, the 68020 transfers next byte of D0 to $30000002 via pins D_{31}–D_{24} and the lowest byte of register D0 to $30000003 via pins D_{23}–D_{16}. Thus, for aligned transfer with 16-bit memory configuration, the 68020 transfers data in two cycles for 32-bit transfers. Next, consider the instruction, `MOVE.L D0,$30000001`. This is a misaligned transfer. The 68020 transfers the most significant byte of D0 to $30000001 via pins D_{23}–D_{16} in the first cycle, the next byte of register D0 to $30000002 via pins D_{31}–D_{24}, and the next byte of D0 to $30000003 via pins D_{23}-D_{16} in the second cycle and finally, the lowest byte of D0 to address $30000004 via pins D_{31}–D_{24} in the third cycle. Thus, for misaligned transfers in a 16-bit memory configuration, the 68020 requires 3 cycles to transfer data for long word instructions.

When the 68020 is configured as a 32-bit memory system, four byte memory chips are connected to D_{31}–D_0. The memory chip with data pins connected to D_{31}–D_{24} contains addresses 0, 4, 8, ...; the memory chip with data pins connected to D_{23}–D_{16} contains addresses 1, 5, 9, ...; the memory chip with data pins connected to D_{15}–D_8 includes addresses 2, 6, 10, ...; and the memory chip with data pins connected to D_7–D_0 contains addresses 3, 7, 11, ... The memory chips inform the 68020 of the 32-bit memory configuration by activating $\overline{\text{DSACK1}}$ = 0 and $\overline{\text{DSACK0}}$ = 0. The 68020 then uses pins D_{31}–D_0 to transfer data for byte, word, or long word instructions. For byte instructions, data is transferred via the appropriate 8 data pins of the 68020 depending on the address in one cycle. For word instructions starting at addresses 0, 4, 8, ..., addresses 1, 5, 9, ..., and addresses 2, 6, 10, ..., data are aligned, and will be transferred in one cycle. For example, consider `MOVE.W D1,$20000005`. The 68020 transfers the contents of D1 (bits 15-8) to address $20000005 via pins D_{23}–D_{16} and contents of register D1 (bits 7-0) to address $20000006 via pins D_{15}–D_8 in one cycle. On the other hand, `MOVE.W D1,$20000007` is a misaligned transfer. In this case, the 68020 transfers the contents of register D1 (bits 15-8) to address $20000007 via pins D_7–D_0 in the first cycle and the contents of D1 (bits 7-0) to address $20000008 via pins D_{31}–D_{24} in the second cycle.

For long word instructions, data transfers with addresses starting at 0, 4, 8, ... are aligned transfers. They will be performed in one cycle. Data with addresses in all other three chips are misaligned and will require additional cycles. For I/O configuration, one to four chips can be connected to the appropriate D_{31}–D_0 pins as required by an application. The addresses in the I/O chips will be memory mapped and connected to the appropriate portions of pins D_{31}–D_0 in the same way as the memory chips.

MC68020 Programmer's Model

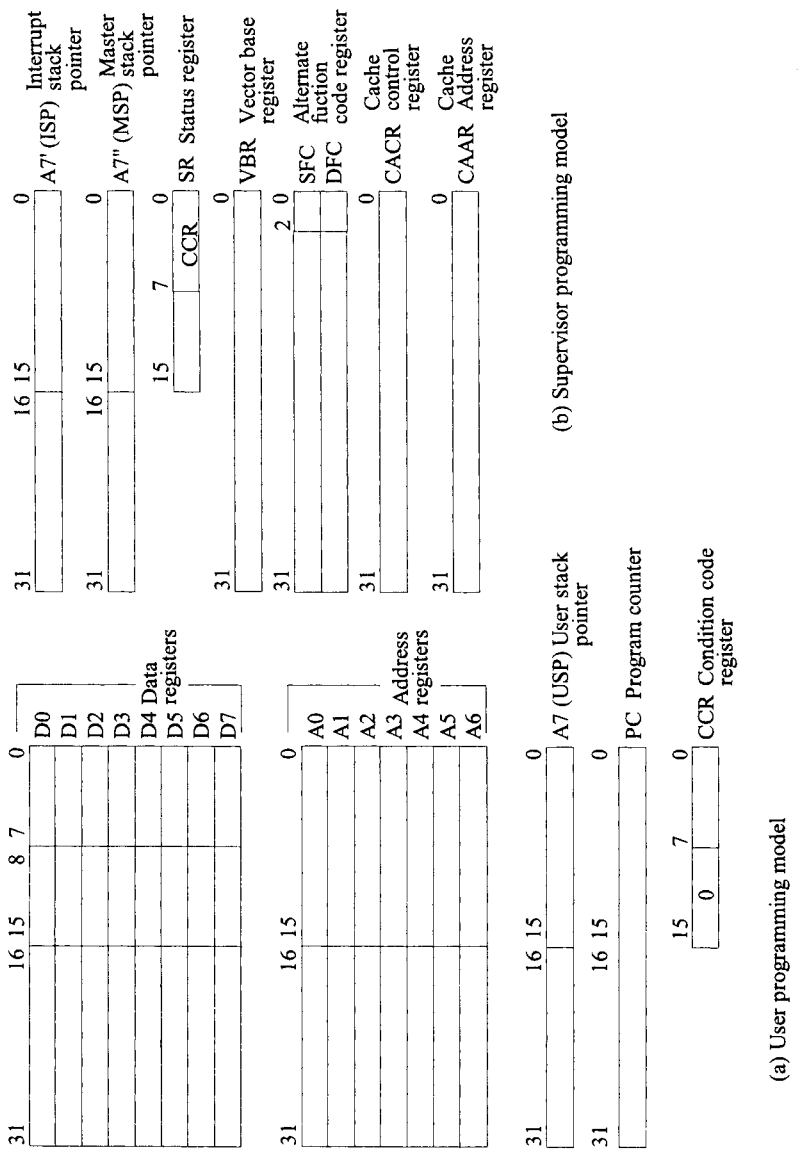
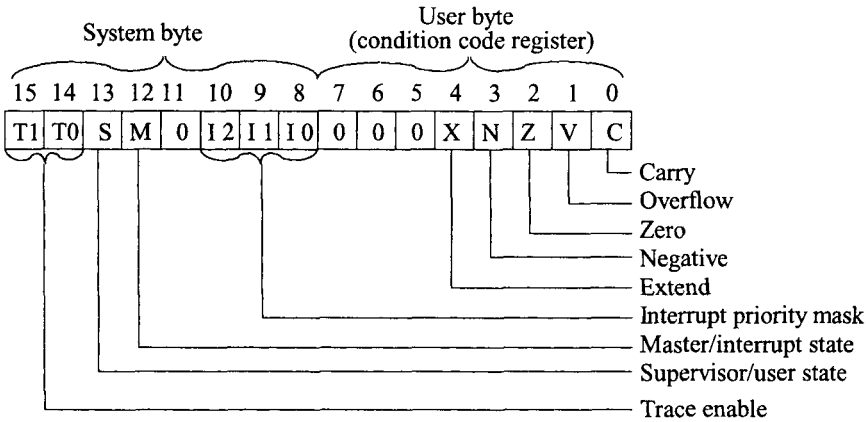


FIGURE 11.4 MC68020 programming model

The MC68020 programmer's model is based on sequential, nonconcurrent instruction execution. This implies that each instruction is completely executed before the next instruction is executed. Although instructions might operate concurrently in actual hardware, they do not operate concurrently in the programmer's model.

Figure 11.4 shows the MC68020 user and supervisor programming models. The user model has fifteen 32-bit general-purpose registers (D0–D7 and A0–A6), 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



T0 - Trace on change of flow (BRA, JUMP,etc.)
T1 - Trace all instructions

T1	T0		S	M	
0	0	No tracing	0	X	USP
0	1	Trace on change of flow	1	0	ISP
1	0	Trace on any instuction execution	1	1	MSP
1	1	Undefined; reserved			

FIGURE 11.5 MC68020 status register

alternate function code registers (SFC and DFC), and two 32-bit cache-handling (address and control) registers (CAAR and CACR). The user stack pointer (USP) A7, interrupt stack pointer (ISP) A7', and master stack pointer (MSP) A7" are system stack pointers.

The status register, as shown in Figure 11.5, 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 the 68020 user byte are set or reset in the same way as those of the 68HC000 user byte. Bits I2, I1, I0, and S have the same meaning as those of the 68HC000. In the 68020, two trace bits (T1, T0) are included as opposed to one trace bit (T) in the 68HC000. These two bits allow the 68020 to trace on both normal instruction execution and jumps. The 68020 M bit is not included in the 68HC000 status register.

The vector base register (VBR) is used to allocate the exception processing vector table in memory. VBR supports multiple vector tables so that each process can properly manage independent exceptions. The 68020 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 code. The CACR provides control and status accesses to the instruction cache; the CAAR holds the address for those cache control functions that require an address.

MC68020 Addressing Modes

Table 11.7 lists the MC68020's 18 addressing modes. Table 11.8 compares the addressing

TABLE 11.7 68020 Addressing Modes

Mode	*Syntax*
• Register direct	
Data register direct	Dn
Address register direct	An
• Register indirect	
Address register indirect (ARI)	(An)
Address register indirect with postincrement	(An)+
Address register indirect with predecrement	–(An)
Address register indirect with displacement	(d16, An)
• Register indirect with index	
Address register indirect with index (8-bit displacement)	(d8, An, Xn)
Address register indirect with index (base displacement)	(bd, An, Xn)
• Memory indirect	
Memory indirect, postindexed	([bd, An], Xn, od)
Memory indirect, preindexed	([bd, An, Xn], od)
• Program counter indirect with displacement	(d16,PC)
• Program counter indirect with index	
PC indirect with index (8-bit displacement)	(d8, PC, Xn)
PC indirect with index (base displacement)	(bd, PC, Xn)
• Program counter memory indirect	
PC memory indirect, postindexed	([bd, PC], Xn, od)
PC memory indirect, preindexed	([bd, PC, Xn], od)
• Absolute	
Absolute short	(xxx).W
Absolute long	(xxx).L
• Immediate	#data

Notes:

Dn	=	data register, D0 -D7
An	=	address register, A0-A6
d8, d16	=	2's complement or sign-extended displacement; added as part of effective address calculation; size is 8 (d8) or 16 (d16) bits; when omitted, assemblers use a value of 0
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	=	2'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
ARI	=	Address Register Indirect

modes of the 68HC000 with those of the MC68020. Because 68HC000 addressing modes were covered earlier in this chapter in detail with examples, the 68020 modes not available in the 68HC000 will be covered in the following discussion.

ARI (Address Register Indirect) with Index (Scaled) and 8-Bit Displacement

- Assembler syntax: (d8, A*n*, X*n*.size * scale)
- EA = (A*n*) + (X*n*.size * scale) + d8
- Xn can be W or L.

If the index register (A*n* or D*n*) is 16 bits, then it is sign-extended to 32 bits and multiplied by 1, 2, 4 or 8 to be used in EA calculations. An example is `MOVE.W (0, A2, D2.W * 2),D1`. Suppose that [A2] = $50000000, [D2.W] = $1000, and [$50002000] = $1571; then, after the execution of this `MOVE`, $[D1]_{\text{low 16 bits}}$ = $1571 because EA = $5000000 + $1000 * 2 + 0 = $50002000.

ARI (Address Register Indirect) with Index and Base Displacement

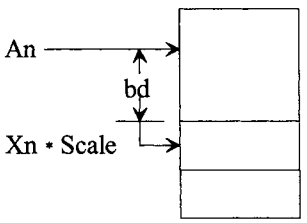
- Assembler syntax: (bd, A*n*, X*n*.size * scale)
- EA = (A*n*) + (X*n*.size * scale) + bd
- Base displacement, bd, has value 0 when present or can be 16 or 32 bits.

The following figure (next page) shows the use of ARI with index, X*n*, and base displacement, bd, for accessing tables or arrays:

TABLE 11.8 Addressing Modes, MC68HC000 vs. MC68020

Addressing Modes Available	*Syntax*	*68HC000*	*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 disp)	(d, An)	Yes	Yes
ARI with index (8-bit disp)	(d, An, Xn)	Yes*	Yes*
ARI with index (base disp; 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 disp. (16-bit)	(d, PC)	Yes	Yes
PC indirect with index (8-bit disp)	(d, PC, Xn)	Yes*	Yes*
PC indirect with index (base disp)	(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	(xxxx).W	Yes	Yes
Absolute long	(xxxxxxxx).L	Yes	Yes
Immediate	#<data>	Yes	Yes

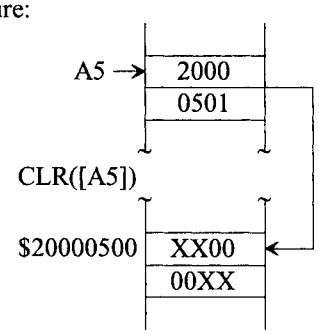
*68HC000 has no scaling capability; 68020 can scale Xn by 1,2,4,or 8.



An example is MOVE.W ($5000, A2, D1.W * 4), D5. If [A2] = $30000000, [D1.W] = $0200, and [$30005800] = $0174, then, after execution of this MOVE, $[D5]_{\text{low 16 bits}} = \$0174$ because EA = $5000 + $30000000 + $0200 * 4 = $30005800.

Memory Indirect

Memory indirect mode is distinguished from address register indirect mode by the use of square brackets in the assembler notation. The concept of memory indirect mode is depicted in the following figure:



Here, register A5 points to the effective address $20000501. Because `CLR ([A5])` is a 16-bit clear instruction, 2 bytes in location $20000501 and $20000502 are cleared to 0.

Memory indirect mode can be indexed with scaling and displacements. There are two types of memory indirect mode with scaled indexing 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 (A*n*) 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. Note that bd and od can be zero, 16 bits, or 32 bits. In postindexed memory indirect mode, indexing occurs after memory indirection.

- Assembler syntax: ([bd, A*n*], X*n*.size * scale, od)
- EA = ([bd + A*n*]) + (X*n*.size * scale + od)

An example is `MOVE.W ([$0004, A1], D1.W * 2, 2), D2`. If [A1] = $20000000, [$2000004] = $00003000, [D1.W] = $0002, and [$00003006] = $1A40, then, after execution of this `MOVE`, intermediate pointer = (4 + $20000000) = $20000004, [$2000004], which is $00003000 used as a pointer. Therefore, EA = $00003000 + $00000004 + 2 = $00003006. Hence, $[D2]_{\text{low 16 bits}} = \$1A40$.

For memory indirect preindexed mode, the scaled index operand is added to the base register (A*n*) 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 read and an optional outer displacement (od) is added to generate the effective address. The indexing, therefore, occurs before indirection.

- Assembler syntax: ([bd, A*n*, X*n*.size * scale], od)

- EA = (bd, A*n* + X*n*.size * scale) + od

As an example of the 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 these devices can be stored in memory pointed to by the respective device addresses plus od.

The memory indirect preindexed mode will now be illustrated by a numerical example. Consider

```
MOVE.W ([$0002, A1,D0.W*2], 2), D1
```

If [A1] = $20000000, [D0.W] = $0004, [$2000000A] = $00121502, [$00121504] = $F124, then after execution of this MOVE, intermediate pointer = $20000000 + $0002 + $0004*2 = $2000000A. Therefore, [$2000000A], which is $00121502, is used as a memory pointer. Hence, [D1] low 16 bits = $F124.

MC68020 Instruction Set

The MC68020 instruction set includes all 68HC000 instructions plus some new ones. Some of the 68HC000 instructions are enhanced. Over 20 new instructions are added to provide new functionality. A list of these instructions is given in Table 11.9.

Succeeding sections will discuss the 68020 instructions listed next:

- 68020 new privileged move instructions
- RTD instruction
- CHK/CHK2 and CMP/CMP2 instructions
- TRAP*cc* instructions
- Bit field instructions

TABLE 11.9 68020 New Instructions

Instruction	*Description*
BFCHG	Bit field change
BFCLR	Bit field clear
BFEXTS	Bit field signed extract
BFEXTU	Bit field unsigned extract
BFFFO	Bit field find first one set
BFINS	Bit field insert
BFSET	Bit field set
BFTST	Bit field test
CALLM	Call module
CAS	Compare and swap
CAS2	Compare and swap (two operands)
CHK2	Check register against upper and lower bounds
CMP2	Compare register against upper and lower bounds
cpBcc	Coprocessor branch on coprocessor condition
cpDBcc	Coprocessor test condition, decrement, and branch
cpGEN	Coprocessor general function
cpRESTORE	Coprocessor restore internal state
cpSAVE	Coprocessor save internal state
cpSETcc	Coprocessor set according to coprocessor condition
cpTRAPcc	Coprocessor trap on coprocessor condition
PACK	Pack BCD
RTM	Return from module
UNPK	Unpack BCD

- PACK and UNPK instructions
- Multiplication and division instructions
- 68HC000 enhanced instructions

68020 New Privileged Move Instructions

The 68020 new privileged move instructions can be executed by the 68020 in the supervisor mode. They are listed below:

Instruction	*Operand Size*	*Operation*	*Notation*
MOVE	16	SR → destination	MOVE SR, (EA)
MOVEC	32	Rc → Rn	MOVEC.L Rc, Rn
		Rn → Rc	MOVEC.L Rn, Rc
MOVES	8, 16, 32	Rn → destination using DFC	MOVES.S Rn, (EA)
		Source using SFC → Rn	MOVES.S (EA),Rn

Note that Rc includes VBR, SFC, DFC, MSP, ISP, USP, CACR, and CAAR. Rn can be either an address or a data register.

The operand size (.L) indicates that the MOVEC operations are always long word. Notice that only register to register operations are allowed. A control register (Rc) can be copied to an address or a data register (Rn) or vice versa. When the 3 bit SFC or DFC register is copied into Rn, all 32 bits of the register are overwritten and the upper 29 bits are "0."

The MOVES (move to alternate space) instruction 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 operand size (.S) indicates that the MOVES instruction can be byte (.B), word (.W), or long word (.L). The MOVES instruction allows register to memory or memory to register operations. When a memory to register move 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 function code pins. The MOVES instruction can be used to move information from one space to another.

Example 11.3

(a) Find the contents of address $70000023 and the function code pins FC2, FC1, and FC0 after execution of `MOVES.B D5, (A5)`. Assume the following data prior to execution of this MOVES instruction: [SFC] = 001_2, [DFC] = 101_2 , [A5] = $70000023, [D5] = $718F2A05, [$70000020] = $01, [$70000021] = $F1, [$70000022] = $A2, [$70000023] = $2A

Solution

After execution of this MOVES instruction,

FC2 FC1 FC0 = 101_2 , [$70000023] = $05

(b) The following 68000 instruction sequence:

```
MOVEA.L  8(A7),A0
MOVE.W   (A0),D3
```

is used by a subroutine to access a parameter whose address has been passed into A0 and then moves the parameter to D3. Find the equivalent 68020 instruction.

Solution `MOVE.W ([8,A7]),D3`

Return and Delocate Instruction

The return and delocate (RTD) instruction 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. The format of the RTD instruction is shown next:

Instruction	*Operand Size*	*Operation*	*Notation*
RTD	Unsized	(SP) → PC, SP + 4 + d → SP	RTD # <disp>

As an example, consider RTD #8, which, at the end of a subroutine, deallocates 8 bytes of unwanted parameters off the stack by adding 8 to the stack pointer and returns to the main program. The size of the displacement is 16-bit.

CHK/CHK2 and CMP/CMP2 Instructions

The 68020 check instruction (CHK) compares a 32-bit twos complement integer value residing in a data register (D*n*) against a lower bound (LB) value of zero and against an upper bound (UB) value of the programmer's choice. The upper bound value is located at the effective address (EA) specified in the instruction format. The CHK instruction has the following format: `CHK.S` (EA), D*n* where the operand size (.S) designates word (.W) or long word (.L).

If the data register value is less than zero (D*n* < 0) or if the data register 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 ≤ D*n* ≤ UB then N is undefined. X is unaffected and all other flags are undefined and program execution continues with the next instruction.

The CHK instruction can be used for maintaining array subscripts because all subscripts can be checked against an upper bound (i.e., UB = array size - 1). If the compared subscript is within the array bounds (i.e., 0 ≤ subscript value ≤ UB 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 > subscript value or subscript value > UB value), then the processor traps through the CHK exception.

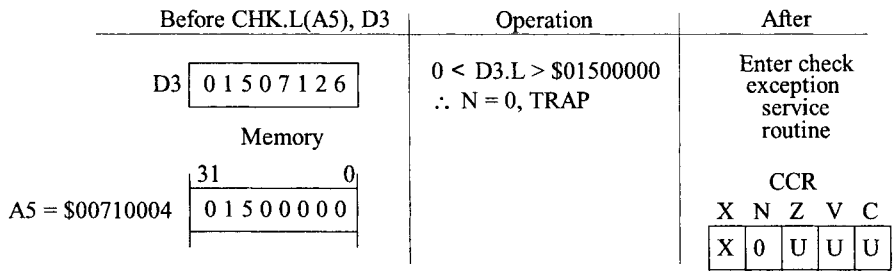
Example 11.4

Determine the effects of execution of CHK.L (A5), D3, where A5 represents 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 this CHK instruction:

[D3] = $01507126
[A5] = $00710004
[$00710004] = $01500000

Solution

The long word array subscript value $01507126 contained in data register D3 is compared against the long word UB value $01500000 pointed to by address register A5. Because the value $01507126 contained in D3 exceeds the UB value $01500000 pointed to by A5, the N bit is cleared. (X is unaffected and the remaining CCR bits are undefined.) This out-of-bounds condition causes the program to trap to a check exception service routine.



The operation of the CHK instruction can be summarized as follows:

Instruction	*Operand Size*	*Operation*	*Notation*
CHK	16, 32	If D*n* < 0 or D*n* > source, then TRAP	CHK (EA), D*n*

The 68020 CMP.S (EA), D*n* instruction subtracts (EA) from D*n* and affects the condition codes without any result. The operand size designator (.S) is either byte (.B) or word (.W) or long word (.L).

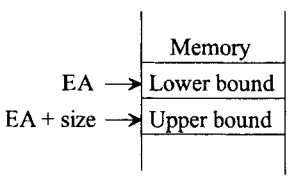
Both the CHK2 and the CMP2 instructions have similar formats:

CHK2 . S (EA), R*n*

and

CMP2 . S (EA), R*n*

They compare a value contained in a data or address register (designated by R*n*) against two (2) bounds chosen by the programmer. The size of the data to be compared (.S) may be specified as byte (.B), word (.W), or long word (.L). As shown in the following figure, the lower bound (LB) value must be located in memory at the effective address (EA) specified in the instruction, and the upper bound (UB) value must follow immediately at the next higher memory address. That is, UB addr = LB addr + size, where size = B (+1), W (+2), or L (+4).



If the compared register is a data register (i.e., R*n* = D*n*) 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., R*n* = A*n*) 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 codes 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 affects only the condition codes.

In both instructions, the compare is performed for either signed or unsigned

bounds. The 68020 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.

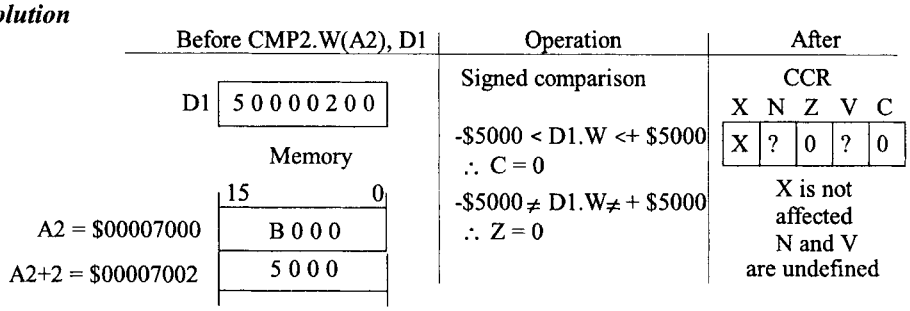
The following CMP2 and CHK2 instruction examples are identical in that they both utilize the same registers, comparison data, and bound values. The difference is how the upper and lower bounds are arranged.

Example 11.5

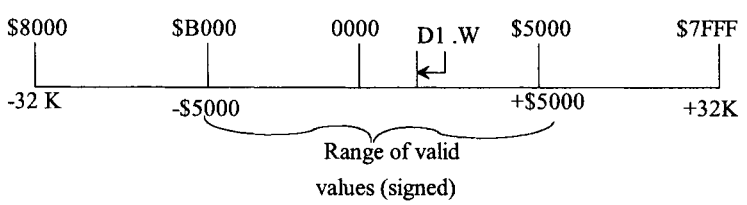
Determine the effects of execution of `CMP2.W (A2),D1`. Assume the following data prior to execution of this `CMP2` instruction:

[D1] = $50000200, [A2] = $00007000
[$00007000] = $B000, [$00007002] = $5000

Solution



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 $5000 immediately following $B000 is the upper bound. Because the lower bound is the arithmetically smaller value, the programmer is indicating to the 68020 to interpret the bounds as signed numbers. The twos 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 whether it is greater than or equal to the upper bound, +$5000, or less than or equal to the lower bound, -$5000. Because the compared value $0200 is within bounds, the carry bit (C) is cleared to 0. Also, because $0200 is not equal to either bound, the zero bit (Z) is cleared. The following figure shows the range of valid values that D1 could contain:



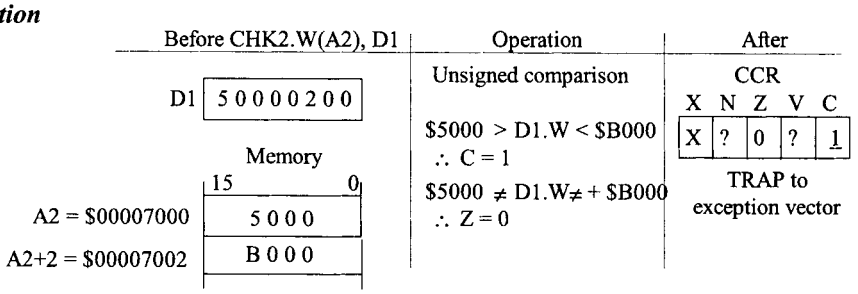
A typical application for the `CMP2` instruction would be to read in a number of user entries and verify that each entry is valid by comparing it against the valid range bounds. In the preceding `CMP2` example, the user-entered value would be in register D1 and register A2 would point to a range for that value. The `CMP2` instruction would verify whether 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 11.6

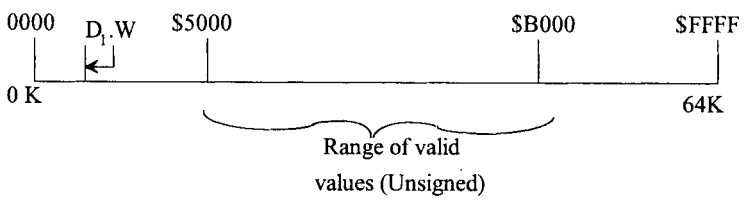
Determine the effects of execution of `CHK2.W (A2), D1`. Assume the following data prior to execution of this `CHK2` instruction:

[D1] = $50000200, [A2] = $00007000
[$00007000] = $5000, [$00007002] = $B000

Solution



This time, the value $5000 located in memory is the lower bound and the value $B000 is the upper bound.



Now, because the lower bound contains the logically smaller value, the programmer is indicating to the 68020 to interpret the bounds as unsigned numbers, representing only a magnitude. Therefore, the instruction evaluates the word contained in register D1 ($0200) to determine whether it is greater than or equal so the lower bound, $5000, or less than or equal to the upper bound, $B000. Because 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, because $0200 is not equal to either bound, the zero bit (Z) is cleared. The figure above shows the range of valid values that D1 could contain.

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 0- 99_{10}, and if the two words located at (A2) and (A2 + 2) contain 50 and 99, 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 `CHK2` instructions are summarized as follows:

Instruction	*Operand Size*	*Operation*	*Notation*
`CMP2`	8,16, 32	Compare R*n* < source – lower bound or R*n* > source – upper bound and set CCR	`CMP2 (EA), Rn`
`CHK2`	8, 16, 32	If R*n* < source – lower bound or R*n* > source – upper bound, then TRAP	`CHK2 (EA), Rn`

Trap-on-Condition Instructions

The new trap condition (`TRAPcc`) instruction allows a conditional trap exception on any of the condition codes shown in Table 11.10. These are the same conditions that are

TABLE 11.10 Conditions for TRAP*cc*

Code	*Description*	*Result*
CC	Carry clear	$\overline{C}$
CS	Carry set	C
EQ	Equal	Z
F	Never true	0
GE	Greater or equal	$N \cdot V + \overline{N} \cdot \overline{V}$
GT	Greater than	$N \cdot V \cdot Z + \overline{N} \cdot \overline{V} \cdot \overline{Z}$
HI	High	$\overline{C} \cdot \overline{Z}$
LE	Less or equal	$Z + N \cdot \overline{V} + \overline{N} \cdot V$
LS	Low or same	C + Z
LT	Less than	$N \cdot \overline{V} + \overline{N} \cdot V$
MI	Minus	N
NE	Not equal	$\overline{Z}$
PL	Plus	N
T	Always true	1
VC	Overflow clear	$\overline{V}$
VS	Overflow set	V

allowed for the set-on-condition (S*cc*) and the branch-on-condition (B*cc*) instructions. The TRAP*cc* instruction evaluates the selected test condition based on the state of the condition code flags, and if the test is true, the 68020 initiates exception processing by trapping through the same exception vector as the TRAPV instruction (vector 7, offset $1C, VBR = VBR + offset). The trap-on-condition instruction format is

TRAP*cc* or TRAP*cc*.S #<data>

where the operand size (.S) designates word (.W) or long word (.L).

If either a word or long word operand is specified, a 1- or 2-word immediate operand is placed following the instruction word. The immediate operand(s) consists of argument parameters that are passed to the trap handler to further define requests or services it should perform. If *cc* is false, the 68020 does not interpret the immediate operand(s) but instead adjusts the program counter to the beginning of the following instruction. The exception handler can access this immediate data as an offset to the stacked PC. The stacked PC is the next instruction to be executed.

A summary of the TRAP*cc* instruction operation is shown next:

Instruction	*Operand Size*	*Operation*	*Notation*
TRAP*cc*	None	If *cc*, then TRAP	TRAP*cc*
	16	Same	TRAPcc.W #<data>
	32	Same	TRAPcc.L #<data>

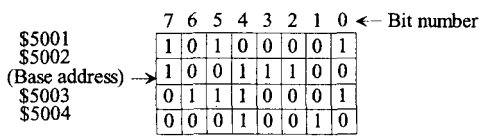
Bit Field Instructions

The bit field instructions, which allow operations to clear, set, ones complement, input, insert, and test one or more bits in a string of bits (bit field), are listed on the next page. Note that the condition codes are affected according to the value in the field before execution of the instruction. All bit field instructions affect the N and Z bits as shown for BFTST. That is, 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. C and V are always cleared. X is

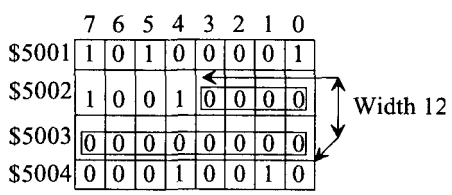
always unaffected. Next, consider BFFFO. The offset of the first bit set 1 in a bit field is placed in D*n*; if no set bit is found, D*n* contains the offset plus the field width. Immediate offset is from 0 to 31, whereas offset in D*n* can be specified from -2^{31} to $2^{31} - 1$. All instructions are unsized. They are useful for memory conservation, graphics, and communications. The bit field instructions are listed below:

Instruction	*Operand Size*	*Operation*	*Notation*
BFTST	1-32	Field MSB → N, Z = 1 if all bits in field are zero; Z = 0 otherwise	BFTST (EA) {offset:width}
BFCLR	1-32	0's → Field	BFCLR (EA) {offset:width}
BFSET	1-32	1's → Field	BFSET (EA) {offset:width}
BFCHG	1-32	$\overline{\text{Field}}$ → Field	BFCHG (EA) {offset:width}
BFEXTS	1-32	Field → D*n*; sign-extended	BFEXTS (EA) {offset:width}, D*n*
BFEXTU	1-32	Field → D*n*; Zero-extended	BFEXTU (EA) {offset:width}, D*n*
BFINS	1-32	D*n* → field	BFINS Dn, (EA) {offset:width}
BFFFO	1-32	Scan for first bit-set in field	BFFFO (EA) {offset:width}, D*n*

As an example, consider BFCLR $5002{4:12}. Assume the following memory contents:



Bit 7 of the base address $5002 has the offset 0. Therefore, bit 3 of $5002 has the offset value of 4. Bit 0 of location $5001 has offset value -1, bit 1 of $5001 has offset value -2, and so on. The example BFCLR instruction just given clears 12 bits starting with bit 3 of $5002. Therefore, bits 0–3 of location $5002 and bits 0–7 of location $5003 are cleared to 0. Therefore, the memory contents change as follows:



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 is interfaced via a 16-bit port of a MC68020 based microcomputer. Now, suppose that 1 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 a 16-bit location for each 12 bits:

  Memory requirements = 2 × 1 million
  = 2 million bytes

- Using bit fields:

  12 bits = 1.5 bytes

  Memory requirements = 1.5 × 1 million
  = 1.5 million bytes

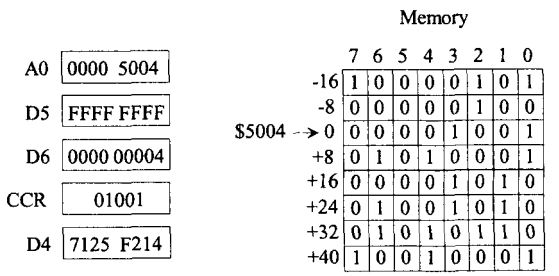
  Savings = 2 million bytes – 1.5 million bytes
  = 500,000 bytes

Example 11.7

Determine the effect of each of the following bit field instructions:

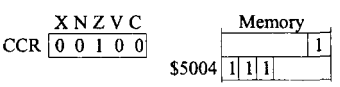
```
BFCHG $5004{D5:D6}
BFEXIU $5004{2:4},D5
BFINS D4,(A0){D5:D6}
BFFFO $5004{D6:4},D5
```

Assume the following data prior to execution of each of the given instructions. Register contents are given in hex, CCR and memory contents in binary, and offset to the left of memory in decimal.

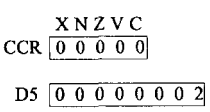


Solution

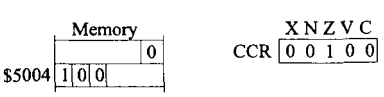
- `BFCHG $5004 {D5:D6}`
  Offset = - 1, Width = 4



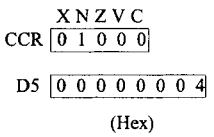
- `BFEXTU $5004 {2:4},D5`
  Offset = 2, Width = 4



- `BFINS D4,(A0) {D5:D6}`
  Offset = - 1, Width = 4



- `BFFFO $5004 {D6:4},D5`
  Offset = 4, Width = 4

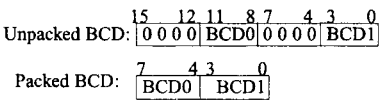


Pack and Unpack Instructions

The details of the PACK and UNPK instructions are listed next:

Instruction	*Operand Size*	*Operation*	*Notation*
PACK	16 → 8	Unpacked source + #data → packed destination	PACK –(A*n*), –(A*n*), #<data> PACK D*n*, D*n*,#<data>
UNPK	8 → 16	Packed source → unpacked source unpacked source + #data → unpacked destination	UNPK –(A*n*), –(A*n*), #<data> UNPK D*n*, D*n*,#<data>

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 two 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 a BCD and vice versa.

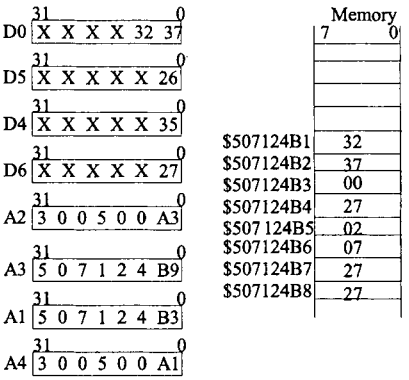
The PACK and UNPK instructions are useful when I/O devices such as an ASCII keyboard and an ASCII printer are interfaced to an MC68020-based microcomputer. Data can be entered into the microcomputer via the keyboard in ASCII codes. The PACK instruction can be used with appropriate adjustments to convert these ASCII codes into packed 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 adjustments to convert packed BCD to ASCII codes for outputting to the ASCII printer.

Example 11.8

Determine the effect of execution of each of the following PACK and UNPK instructions:

- `PACK D0,D5,#$0000`
- `PACK-(A1),-(A4),#$0000`
- `UNPK D4,D6,#$3030`
- `UNPK-(A3),-(A2),#$3030`

Assume the following data prior to execution of each of the above instructions:



Solution

- `PACK D0,D5,#$0000`

[D0] = 32 37
low word
+ 00 00
32 37
[D5] = 27

Note that ASCII code for 2 is $32 and for 7 is $37. Hence, this pack instruction converts ASCII code to packed BCD.

- `PACK -(A1),-(A4),$0000`

71 24B2] = 37 3237
71 24B1] = 32 0000
3237
∴ [3005 00A0] = 27 packed BCD

Hence, this pack instruction with the specified data converts two ASCII digits to their equivalent packed BCD form.

- `UNPK D4,D6,#$3030`

[D4] = XXXXXX 35
03 05
+ 30 30
33 35
∴ [D6] = XXXX 33 35
[D4] = XXXXXX 35

Therefore, this UNPK instruction with the assumed data converts from packed BCD in D4 to ASCII code in D6; the contents of D4 are not changed.

- `UNPK -(A3),-(A2),#$3030`

[$5071 24B8] = 27
02 07
30 30
32 37
∴ [$300500A2] = 37
[$300500A1] = 32

This UNPK instruction with the assumed data converts two packed BCD digits to their equivalent ASCII digits.

Multiplication and Division Instructions

The 68020 includes the following signed and unsigned multiplication instructions:

Instruction	*Operand Size*	*Operation*
`MULS.W` (EA), D*n* *or* `MULU`	16 × 16 → 32	(EA)16 * (D*n*)16 → (D*n*)32
`MULS.L` (EA), D*n* *or* `MULU`	32 × 32 → 32	(EA) * D*n* → D*n* D*n* holds 32 bits of the result after multiplication. Upper 32 bits of the result are discarded.
`MULS.L` (EA),D*h*:D*n* *or* `MULU`	32 × 32 → 64	(EA) * D*n* → D*h*:D*n* (EA) holds 32-bit multiplier before multiplication D*h* holds high 32 bits of product after multiplication. Dn holds 32-bit multiplicand before multiplication and low 32 bits of product after multiplication.

(EA) can use all modes except A*n*. The condition codes N, Z, and V are affected; C is always cleared to 0, 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 the long word form (32 × 32), 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 are provided.

The signed and unsigned division instructions of the 68020 include the following, in which the source is the divisor, the destination is the dividend.

Instruction	*Operation*
`DIVS.W` (EA), D*n* *or* `DIVU`	32/16 → 16*r*:16*q*
`DIVS.L` (EA), D*q* *or* `DIVU`	32/32 → 32*q* No remainder is provided.
`DIVS.L` (EA),D*r*:D*q* *or* `DIVU`	64/32 → 32*r*:32*q*
`DIVSL.L` (EA),D*r*:D*q* *or* `DIVUL`	Dr/(EA) → 32*r*:32*q* D*r* contains 32-bit dividend

(EA) can use all modes except A*n*. 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 for 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. Division by zero causes a trap. If overflow is detected before completion of the instruction, V is set to 1, 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 D*n* 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 the instruction

`DIVS.L` (EA), D*q*
or
`DIVU`

both destination and source operands are 32 bits. The result in D*q* contains the 32-bit quotient and the remainder is discarded.

For the instruction

`DIVS.L` (EA), D*r*:D*q*
or
`DIVU`

the destination is 64 bits contained in any two data registers and the source is 32 bits. The 32-bit register D*r* (D0–D7) contains the 32-bit remainder and the 32-bit register D*q* (D0–D7) contains the 32-bit quotient.

For the instruction

`DIVSL.L` (EA), D*r*:D*q*
or
`DIVUL`

the 32-bit register D*r* (D0–D7) contains the 32-bit dividend and the source is also 32 bits. After division, D*r* contains the 32-bit remainder and D*q* contains the 32-bit quotient.

Example 11.9

Determine the effect of execution of each of the following multiplication and division instructions:

- `MULU.L #$2,D5` if [D5] = $FFFFFFFF
- `MULS.L #$2,D5` if [D5] = $FFFFFFFF
- `MULU.L #$2,D5:D2` if [D5] = $2ABC1800 and [D2] = $FFFFFFFF
- `DIVS.L #$2,D5` if [D5] = $FFFFFFFC
- `DIVS.L #$2,D2:D0` if [D2] = $FFFFFFFF and [D0] = $FFFFFFFC
- `DIVSL.L #$2,D6:D1` if [D1] = $00041234 and [D6] = $FFFFFFFD

Solution

- `MULU.L #$2,D5` if [D5] = $FFFFFFFF

```
          $FFFFFFFF
        * $00000002
  00000001  FFFFFFFE
```

V = 1 since this is nonzero; Low 32-bit result in D5

Therefore, [D5] = $FFFFFFFE, N = 0 since the most significant bit of the result is

0, Z = 0 because the result is nonzero, V = 1 because the high 32 bits of the 64-bit product are not zero, C = 0 (always), and X is not affected.

- `MULS.L #$2,D5` if [D5] = $FFFFFFFF

$$\begin{array}{rr} & \$FFFFFFFF\ (-1) \\ * & \$00000002\ (+2) \\ \hline \$FFFFFFFF & \underbrace{\$FFFFFFFE}\ (-2) \end{array}$$

Result in D5

Therefore, [D5] = $FFFFFFFE, X is unaffected, C = 0, N = 1, V = 0, and Z = 0.

- `MULU.L #$2,D5:D2` if [D5] = $2ABC1800 and D2 = $FFFFFFFF

$$\begin{array}{r} \$FFFFFFFF \\ * \ \$00000002 \\ \hline \underbrace{00000001}_{D5}\ \underbrace{FFFFFFFE}_{D2} \end{array}$$

Here N = 0, Z = 0, V = 0, C = 0, and X is not affected.

- `DIVS.L #$2,D5` if [D5] = $FFFFFFFC

$$\begin{array}{r} \overbrace{FFFF\ FFFE}^{-2} \\ \underbrace{00000002}_{+2}\ \overline{)\ \underbrace{FFFF\ FFFC}_{-4}} \end{array}$$

[D5] = $FFFFFFFE, X is unaffected, N = 1, Z = 0, V = 0, and C = 0 (always).

- `DIVS.L #$2,D2:D0` if [D2] = $FFFFFFFF and [D0] = $FFFFFFFC

$$\begin{array}{r} \overbrace{q = FFFF\ FFFE}^{-2},\ \overbrace{r = 0000\ 0000}^{0} \\ \underbrace{0000\ 0002}_{2}\ \overline{)\ \underbrace{FFFF\ FFFF\ FFFF\ FFFC}_{-4}} \end{array}$$

[D2] = $00000000 = remainder, [D0] = $FFFFFFFE = quotient, X is unaffected, Z = 0, N = 1, V = 0, and C = 0 (always).

- `DIVSL.L #$2,D6:D1` if [D1] = $00041234 and [D6] = $FFFFFFFD

$$\begin{array}{r} \overbrace{q = FFFFFFFF}^{-1},\ \overbrace{r = FFFFFFFF}^{-1} \\ 0000\ 0002\ \overline{)\ \underbrace{FFFFFFFD}_{-3}} \end{array}$$

[D6] = $FFFFFFFF = remainder, [D1] = $FFFFFFFF = quotient, X is unaffected, N = 1, Z = 0, V = 0, and C = 0 (always).

MC68HC000 Enhanced Instructions

The MC68020 includes the enhanced version of the instructions as listed next:

Instruction	*Operand Size*	*Operation*
`BRA` *label*	8, 16, 32	PC + d → PC
`B`*cc 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 that S can be B, W, or L. In addition to 8- and 16-bit signed displacements for BRA, B*cc*, and BSR like the 68HC000, the 68020 also allows signed 32-bit displacements. LINK is unsized in the 68HC000. (EA) in CMPI and TST supports all 68HC000 modes plus PC relative. An example is CMPI.W #$2000, (START, PC). In addition to EXT.W D*n* and EXT.L D*n* like the 68HC000, the 68020 also provides an EXTB.L instruction.

Example 11.10

Write a program in 68020 assembly language to multiply a 32-bit signed number in D2 by a 32-bit signed number in D3 by storing the multiplication result in the following manner:
(a) Store the 32-bit result in D2.
(b) Store the high 32 bits of the result in D3 and the low 32 bits of the result in D2.

Solution

```
(a)         MULS.L  D3,D2
   FINISH   JMP     FINISH
(b)         MULS.L  D3,D3:D2
   FINISH   JMP     FINISH
```

Example 11.11

Write a program in 68020 assembly language to convert 10 packed BCD bytes (20 BCD digits) stored in memory starting at address $00002000 and above, to their ASCII equivalents and, store the result in memory locations starting at $FFFF8000.

Solution

```
       MOVEA.W  #$2000,A0    ;   Load starting addr. of BCD array into A0
       MOVEA.W  #$8000,A1    ;   Load starting addr. of ASCII array into A1
       MOVEQ.L  #9,D0        ;   Load data length into D0
START  MOVE.B   (A0)+,D1     ;   Load a packed BCD byte
       UNPK     D1,D2,#$3030;   Convert to ASCII
       MOVE.W   D2,(A1)+     ;   Store ASCII data to addr. pointed to by A1
       DBF.W    D0,START     ;   Decrement and branch if false
FINISH JMP      FINISH       ;   otherwise stop
```

M68020 Pins and Signals

The 68020 is arranged in a 13 x 13 matrix array (114 pins defined) and fabricated in a pin grid array (PGA) or other packages such as RC suffix package. Both the 32-bit address (A_0–A_{31}) and data (D_0–D_{31}) pins of the 68020 are nonmultiplexed. The 68020 transfers data

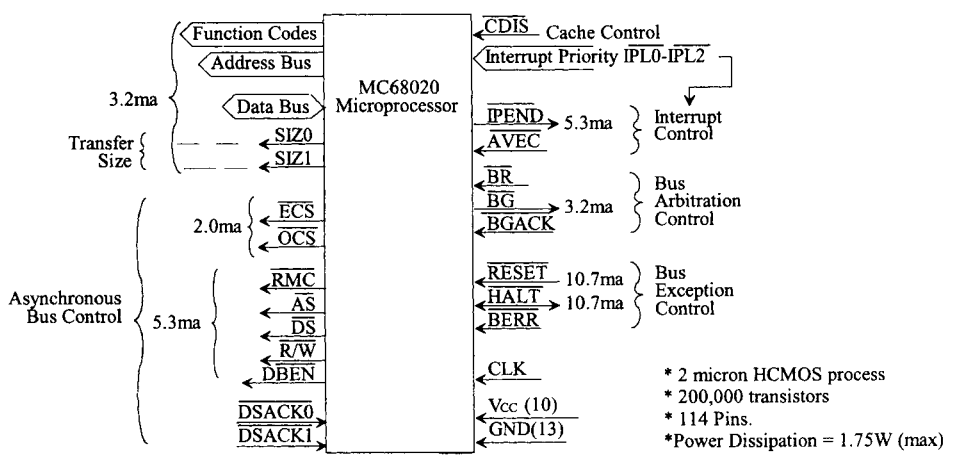


FIGURE 11.6 MC68020 functional signal groups

with an 8-bit device via D_{31}–D_{24}, with a 16-bit device via D_{16}–D_{31}, and with a 32-bit device via D_{31}–D_0. Figure 11.6 shows the MC68020 functional signal group. Table 11.11 lists these signals along with a description of each. There are 10 Vcc (+5 V) and 13 ground pins to distribute power in order to reduce noise.

Like the MC68HC000, 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 except that the 68020 defines the CPU space cycle 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

Note that in the 68HC000, FC2, FC1, FC0 = 111 indicates the interrupt acknowledge cycle. In the MC68020, it indicates the CPU space cycle. In this cycle, by decoding the address lines A_{19}–A_{16}, the MC68020 can perform various types of functions such as coprocessor communication, breakpoint acknowledge, interrupt acknowledge, and module operations as follows:

A_{19}	A_{18}	A_{17}	A_{16}	*Function performed*
0	0	0	0	Breakpoint acknowledge
1	0	0	1	Module operations
0	0	1	0	Coprocessor communication
1	1	1	1	Interrupt acknowledge

Note that A_{19}, A_{18}, A_{17}, A_{16} = 0011_2 to 1110_2 is reserved by Motorola. In the coprocessor communication CPU space cycle, the MC68020 determines the coprocessor type by decoding A_{15}–A_{13} as follows:

A_{15}	A_{14}	A_{13}	*Coprocessor Type*
0	0	0	MC68851 paged memory management unit
0	0	1	MC68881 floating-point coprocessor

The 68020 offers a feature called "dynamic bus sizing," which enables designers to use 8-bit and 16- and 32-bit memory and I/O devices without sacrificing system performance. The SIZ0, SIZ1, $\overline{\text{DSACK0}}$ and $\overline{\text{DSACK1}}$ pins are used to implement this. These pins are defined as follows:

SIZ1	*SIZ0*	*Number of Bytes Remaining to be Transferred*
0	1	Byte
1	0	Word
1	1	3 bytes
0	0	Long words

$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$	*Device Size*
0	0	32-bit device
0	1	16-bit device
1	0	8-bit device
1	1	Data not ready; insert wait states

During each bus cycle, the external device indicates its width via $\overline{\text{DSACK0}}$ and $\overline{\text{DSACK1}}$. The $\overline{\text{DSACK0}}$ and $\overline{\text{DSACK1}}$ pins are used to indicate completion of bus cycle.

TABLE 11.11 Hardware Signal Index

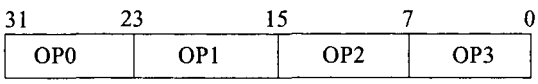
Signal Name	*Mnemonic*	*Function*
Address bus	A_0-A_{31}	32-bit address bus used to address any of 4,294,967,296 bytes
Data bus	D_0-D_{31}	32-bit data bus used to transfer 8,16,24, or 32 bits of data per bus cycle
Function codes	FC0-FC2	3-bit function code used to identify the address space of each bus 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	$\overline{RMC}$	Provides an indicator that the current bus cycle is part of an indivisible read-modify-write operation
External cycle start	$\overline{ECS}$	Provides an indication that a bus cycle is beginning
Operand cycle start	$\overline{OCS}$	Identical operation to that of $\overline{ECS}$ except that $\overline{OCS}$ is asserted only during the first bus cycle of an operand transfer
Address strobe	$\overline{AS}$	Indicates that a valid address is on the bus
Data strobe	$\overline{DS}$	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/$\overline{W}$	Defines the bus transfer as a 68020 read or write
Data buffer enable	$\overline{DBEN}$	Provides an enable signal for external data buffers
Data transfer and size acknowledge	$\overline{DSACK0}$/ $\overline{DSACK1}$	Bus response signals that indicate the requested data transfer operation are completed; in addition, these two lines indicate the use of the external bus port on a cycle-by-cycle basis
Cache disable	$\overline{CDIS}$	Dynamically disables the on-chip cache
Interrupt priority level	$\overline{IPL0}$-$\overline{IPL2}$	Provides an encoded interrupt level to the processor
Autovector	$\overline{AVEC}$	Requests an autovector during an interrupt acknowledge cycle
Interrupt pending	$\overline{IPEND}$	Indicates that an interrupt is pending
Bus request	$\overline{BR}$	Indicates that an external device requires bus mastership
Bus grant	$\overline{BG}$	Indicates that an external device may assume bus mastership
Bus grant acknowledge	$\overline{BGACK}$	Indicates that an external device has assumed bus control
Reset	$\overline{RESET}$	System reset
Halt	$\overline{HALT}$	Indicates that the processor should suspend bus activity
Bus error	$\overline{BERR}$	Indicates that an illegal bus operation is being attempted
Clock	CLK	Clock input to the processor
Power supply	VCC	+5 volt ± 5% power supply
Ground	GND	Ground connection

At the start of a bus cycle, the 68020 always transfers data to lines D_0–D_{31}, taking into consideration that the memory or I/O device may be 8, 16, or 32 bits wide. After the first bus cycle, the 68020 knows the device size by checking the $\overline{DSACK0}$ and $\overline{DSACK1}$ pins and generates additional bus cycles if needed to complete the transfer.

Unlike the 68HC000, the 68020 permits word and long word operands to start at an odd address. However, if the starting address is odd, additional bus cycles are required to

complete the transfer. For example, for a 16-bit device, the 68020 requires 2 bus cycles for a write to an even address such as `MOVE.L D1,$40002050` to complete the operation. On the other hand, the 68020 requires 3 bus cycles for `MOVE.L D1,$40002051` for a 16-bit device to complete the transfer. Note that, as in the 68HC000, instructions in the 68020 must start at even addresses.

Next, consider an example of dynamic bus sizing. The four bytes of a 32-bit data can be defined as follows:



If this data is held in a data register D*n* and is to be written to a memory or 1/0 location, then the address lines A_1 and A_0 define the byte position of data. For a 32-bit device, $A_1A_0 = 00$ (addresses 0, 4, 8, ...), $A_1A_0 = 01$ (addresses 1, 5, 9, ...), $A_1A_0 = 10$ (addresses 2, 6, 10, ...), and $A_1A_0 = 11$ (addresses 3, 7, 11, ...) will store OP0, OP1, OP2, and OP3, respectively. This data is written via the 68020 D_{31}–D_0 pins. However, if the device is 16-bit, data is always transferred as follows:

All even-addressed bytes via pins D_{31}–D_{24}.

All odd-addressed bytes via pins D_{23}–D_{16}.

Finally, for an 8-bit device, both even- and odd-addressed bytes are transferred via pins D_{31}–D_{24}.

The 68020 always starts transferring data with the most significant byte first. As an example, consider `MOVE.L D1,$20107420`. In the first bus cycle, the 68020 does not know the size of the device and, hence, outputs all combinations of data on pins D_{31}–D_0, taking into consideration that the device may be 8, 16, or 32 bits wide. Assume that the content of D1 is $02A10512 (OP0 = $02, OP1 = $A1, OP2 = $05, and OP3 = $12). In the first bus cycle, the 68020 sends SIZ1 SIZ0 = 00, indicating a 32-bit transfer, and then outputs data on its D_{31}–D_0 pins as follows:

D_{31}:D_{24}	D_{23}:D_{16}	D_{15}:D_8	D_7:D_0
$02	$A1	$05	$12

If the device is 8-bit, it will take data $02 from pins D_{31}–D_{24} in the first cycle and will then assert $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$ as 10, indicating an 8-bit device. The 68020 then transfers the remaining 24 bits ($A1 first, $05 next, and $12 last) via pins D_{31}–D_{24} in three consecutive cycles, with a total of four cycles being necessary to complete the transfer.

However, if the device is 16-bit, in the first cycle the device will take the 16-bit data $02A1 via pins D_{31}–D_{16} and will then assert $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$ as 01, indicating a 16-bit device. The 68020 then transfers the remaining 16 bits ($0512) via pins D_{31}–D_{16} in the next cycle, requiring a total of two cycles for the transfer.

Finally, if the device is 32-bit, the device receives all 32-bit data $02A10512 via pins D_{31}–D_0 and asserts $\overline{\text{DSACK1}}$ $\overline{\text{DSACK0}}$ = 00 to indicate completion of the transfer. Aligned data transfers for various devices are as follows :

For 8-bit device:

	31 ..... 0 ← Bit number							
Register D1	02	A1	05	12				
68020 pins	D_{31} D_{24}		SIZ1	SIZ0	A_1	A_0	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$
First cycle	02		0	0	0	0	1	0
Second cycle	A1		1	1	0	1	1	0
Third cycle	05		1	0	1	0	1	0
Fourth cycle	12		0	1	1	1	1	0

For 16-bit device:

68020 pins	D_{31} D_{24}	D_{23} D_{16}	SIZ1	SIZ0	A_1	A_0	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$
First cycle	02	A1	0	0	0	0	0	1
Second cycle	05	12	1	0	1	0	0	1

For 32-bit device:

68020 pins	D_{31} D_0	SIZ1	SIZ0	A_1	A_0	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$
First cycle	02 A1 05 12	0	0	0	0	0	0

Next, consider a misaligned transfer such as MOVE.W D1, $02010741 with [D1] = $20F107A4. The 68020 outputs $0707A4XX on its D_{31}-D_0 pins in its first cycle where XX are don't cares. Data transfers to various devices are summarized below:

For 8-bit device:

	31 23 15 7 0 ← Bit number							
Register D1	20	F1	07	A4				
68020 pins	D_{31} D_{24}		SIZ1	SIZ0	A_1	A_0	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$
First cycle	07		1	0	0	1	1	0
Second cycle	A4		0	1	1	0	1	0

For 16-bit device:

68020 pins	D_{31} D_{24} D_{23} D_{16}	SIZ1	SIZ0	A_1	A_0	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$
First cycle	07	1	0	0	1	0	1
Second cycle	A4	0	1	1	0	0	1

For 32-bit device:

68020 pins	D_{31} D_{24} D_{23} D_{16} D_{15} D_8 D_7 D_0	SIZ1	SIZ0	A_1	A_0	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$
First cycle	07 A4	1	0	0	1	0	0

Let us explain some of the other 68020 pins.

The $\overline{\text{ECS}}$ (external cycle start) pin is an 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 $\overline{\text{ECS}}$ must be validated later with $\overline{\text{AS}}$, because the MC68020 may start an instruction fetch cycle and then abort it if the instruction is found in the cache. In the case of a cache hit, the MC68020 does not assert $\overline{\text{AS}}$, but provides A_{31}–A_0, SIZ1, SIZ0, and FC2–FC0 outputs.

The MC68020 $\overline{\text{AVEC}}$ input is activated by an external device to service an autovector interrupt. The $\overline{\text{AVEC}}$ has the same function as $\overline{\text{VPA}}$ on the 68HC000. The

functions of the other signals, such as $\overline{AS}$, R/$\overline{W}$, $\overline{IPL2}$ - $\overline{IPL0}$, $\overline{BR}$, $\overline{BG}$, and $\overline{BGACK}$, are similar to those of the MC68HC000.

The MC68020 system control pins are functionally similar to those of the MC68HC000. However, there are some minor differences. For example, for hardware reset, $\overline{RESET}$ and $\overline{HALT}$ pins need not be asserted simultaneously. Therefore, unlike the 68HC000, the $\overline{RESET}$ and $\overline{HALT}$ pins are not required to be tied together in the MC68020 system.

The $\overline{RESET}$ and $\overline{HALT}$ pins are bidirectional and open drain (external pull-up resistances are required), and their functions are independent. The $\overline{RESET}$ signal is a bidirectional signal. The $\overline{RESET}$ pin, when asserted by an external circuit for a minimum of 520 clock periods, the $\overline{RESET}$ pin 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 content of address $00000000 and loads it into the ISP (the 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 the I2 I1 I0 bits of the SR to 1 1 1, sets the S bit in the SR to 1, and clears the T1, T0, and 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 $\overline{RESET}$ pin for 512 clock cycles and the processor resets all the external devices connected to the $\overline{RESET}$ pin. Software reset does not affect any internal register.

As mentioned earlier while describing dynamic bus sizing, the 68020 always drives all data lines during a write operation. Furthermore, for all inputs there is a sample window of at least 20 ns during which the 68020 latches the input level. 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, 20 ns; otherwise, the level recognized by the MC68020 is unknown or legal.

During data transfer operations, the 68020 can use either synchronous or asynchronous operation. In synchronous operation, the 68020 clock is used to generate $\overline{DSACK1}$, $\overline{DSACK0}$, and other asynchronous inputs. Also, in synchronous operation, if the $\overline{DSACK1}$ and $\overline{DSACK0}$ are asserted for the required window of at least 20 ns (at least 5 ns before and at least15 ns after the falling edge of S2) on the falling edge S2, the 68020 latches valid data on the falling edge of S4 on a read cycle. The 68020 does not generate any wait states if $\overline{DSACK1}$ and $\overline{DSACK0}$ are asserted at the falling edge of S2; otherwise the 68020 inserts wait cycles like the 68HC000 and latches data at the falling edge of the following cycle as soon as $\overline{DSACK1}$ and $\overline{DSACK0}$ are asserted. A minimum of three clock cycles are required for a read operation.

In asynchronous operation, clock frequency independence at a system level is achieved and the 68020 is used in an asynchronous manner. This typically requires using the bus signals such as $\overline{AS}$, $\overline{DS}$, $\overline{DSACK1}$, and $\overline{DSACK0}$ to control data transfer. Using asynchronous operation, $\overline{AS}$ starts the bus cycle and $\overline{DS}$ is used as a condition of valid data on a write cycle. Decoding of SIZ1, SIZ0, A_1, and A_0 provides enable signals, which indicate the portion of the data bus that is used in data transfer. The memory or I/O chip

then responds by placing the requested data on the correct portion of the data bus for a read cycle or latching the data on a write cycle and asserting $\overline{\text{DSACK1}}$, and $\overline{\text{DSACK0}}$, corresponding to the memory or I/O port size (8-bit, 16-bit, or 32-bit), to terminate the bus cycle. If no memory or I/O device responds or the address is invalid, the external control logic asserts the $\overline{\text{BERR}}$ or $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ signal(s) to abort or retry the bus cycle or retries the bus cycle.

In asynchronous operation, the $\overline{\text{DSACK1}}$, and $\overline{\text{DSACK0}}$ signals are allowed to be asserted before the data from memory or an I/O device is valid on a read cycle. The 68020 latches data according to Parameter #31 provided in Motorola manuals. (Parameter #31 is a maximum of 60 ns for the 12.5-MHz 68020, a maximum of 50 ns for the 16.67-MHz 68020, and a maximum of 43 ns for the 20-Mhz 68020, and maximum time is specified from the assertion of $\overline{\text{AS}}$ to the assertion of $\overline{\text{DSACK1}}$, and $\overline{\text{DSACK0}}$. This is because the 68020 will insert wait cycles in one-clock-cycle increments until $\overline{\text{DSACK1}}$, and $\overline{\text{DSACK0}}$ are recognized as asserted.)

MC68020 System Design

The following 8-MHz 68020 system design will use a 128 KB 32-bit wide supervisor data memory. Four 27C256's (32K × 8 HCMOS EPROM with 120-ns access time) are used for this purpose. Because the memory is 32 KB, the 68020 address lines A_2–A_{16} are used for addressing the 27C256's. The 68020 SIZ1, SIZ0, A_1, A_0, $\overline{\text{DSACK1}}$, and $\overline{\text{DSACK0}}$ pins are utilized for selecting the memory chips.

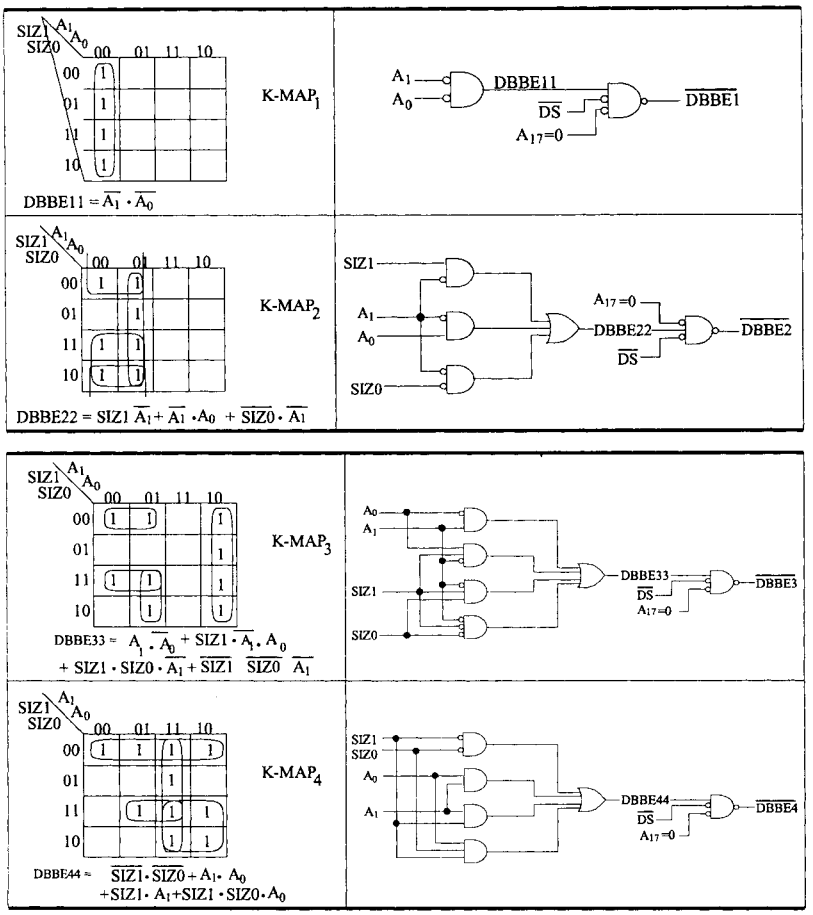
Table 11.12 shows the table for designing the enable logic for the four 27C256 chips. The 68020 A_{17} pin is used to distinguish between memory and I/O. $A_{17} = 0$ is used to select the memory chips; $A_{17} = 1$ is used to select I/O chips (not shown in the design). Table 11.13 shows the K-maps for the enable logic. A logic diagram can be drawn for generating the memory byte enable signals $\overline{\text{DBBE1}}$, $\overline{\text{DBBE2}}$, $\overline{\text{DBBE3}}$, and $\overline{\text{DBBE4}}$.

The 68020 system with 32-bit memory consists of four 27C256's, each connected to its associated portion of the system data bus (D_{31}–D_{24}, D_{23}–D_{16}, D_{15}–D_8, and D_7–D_0).

TABLE 11.12 Table for memory enables for 32-bit memory

SIZ1	*SIZ0*	A_1	A_0	*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

TABLE 11.13 K-maps for Enable Signals for Memory



To manipulate this memory configuration, 32-bit data bus control byte enable logic is incorporated to generate byte enable signals ($\overline{DBBE1}$, $\overline{DBBE2}$, $\overline{DBBE3}$, and $\overline{DBBE4}$). These byte enables are generated by using 68020's SIZ1, SIZ0, A_1, A_0, A_{17}, and $\overline{DS}$ pins as shown in the individual logic diagrams of the byte enable logic. A PAL can be programmed to implement this logic. A schematic of the 68020–27C256 interface is shown in Figure 11.7.

Because the 68020 clock is used to generate $\overline{DSACK1}$, and $\overline{DSACK0}$, the 68020 operates in synchronous mode.

A 74HC138 decoder is used for selecting memory banks to enable the appropriate memory chips. The 74HC138 is enabled by $\overline{AS}$ = 0. The output line 5 (FC2FC1FC0 = 101 for supervisor data) is used to select the memory chips. Assuming don't cares to be zeros and also note that A_{17} = 0 for memory, the supervisor data memory map is obtained as follows:

EPROM #1 $00000000, $00000004, ..., $0001FFFC
EPROM #2 $00000001, $00000005, ..., $0001FFFD
EPROM #3 $00000002, $00000006, ..., $0001FFFE
EPROM #4 $00000003, $00000007, ..., $0001FFFF

$\overline{DSACK1}$ and $\overline{DSACK0}$ are generated by ANDing the $\overline{DBBE1}$, $\overline{DBBE2}$, $\overline{DBBE3}$,

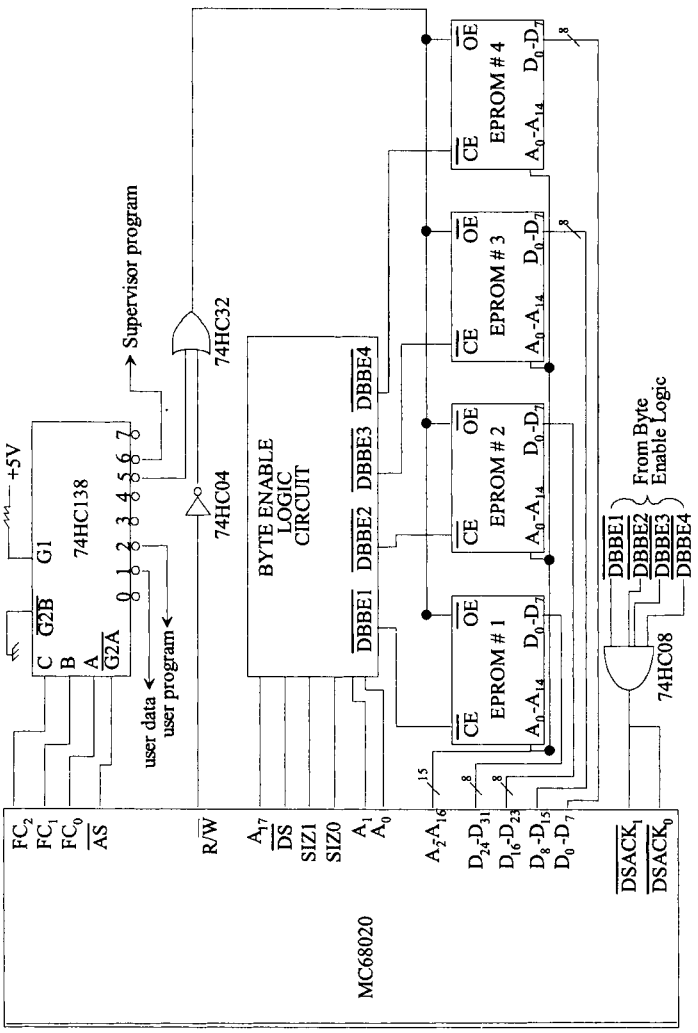


FIGURE 11.7 68020/27C256 System

and $\overline{\text{DBBE4}}$ outputs of the byte enable logic circuit. When one or more EPROM chips are selected, the appropriate enables ($\overline{\text{DBBE1}}$–$\overline{\text{DBBE4}}$) will be low, thus asserting $\overline{\text{DSACK1}}$ = 0 and $\overline{\text{DSACK0}}$ = 0. This will tell the 68020 that the memory is 32 bits wide. Data from the selected memory chip(s) will be placed on the appropriate data pins of the 68020. For example, in response to execution of the instruction `MOVE.W $00000001,D0` in the supervisor mode, the 68020 will generate appropriate signals to generate $\overline{\text{DBBE1}}$= 1, $\overline{\text{DBBE2}}$= 0, $\overline{\text{DBBE3}}$= 0, $\overline{\text{DBBE4}}$= 1, R/$\overline{\text{W}}$ = 1, and output 5 of the decoder = 0

This will select EPROM #2 and EPROM #3 chips. Thus, the contents of address $00000001 are transferred to D0 (bits 8–15) and the contents of address $00000002 are moved to D0 (bits 0–7). The supervisor program, user program, and user data memories can be connected in a similar way (not shown in the figure). For each memory space, four memory chips are required.

Let us discuss the timing requirements of the 68020/27C256 system. Because the

68020 clock is used to generate $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$, the 68020 operates in synchronous mode. This means that the 68020 checks $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$ for LOW at the falling edge of S2 (two cycles). From the 68020 timing diagram (Motorola manual), $\overline{\text{AS}}$, $\overline{\text{DS}}$, and all other output signals used in memory decoding go to LOW at the end of approximately one clock cycle. For an 8-MHz 68020 clock, each cycle is 125 ns. From byte enable logic diagrams, a maximum of four gate delays (40 ns) are required. Therefore, the selected EPROM(s) will be enabled after 165 ns (125 ns + 40 ns). With 120-ns access time, the EPROM(s) will place data on the output lines after approximately 285 ns (165 ns + 120 ns). With an 8-MHz 68020 clock, $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$ will be checked for LOW (32-bit memory) after two cycles (250 ns) and if LOW, the 68020 will latch data after three cycles (375 ns). Hence, no delay circuit is required for $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$. In case a delay circuit is required, a ring counter can be used. Note that the 20-ns window requirement for $\overline{\text{DSACK1}}$ and $\overline{\text{DSACK0}}$ inputs (5 ns before and 15 ns after the falling edge of S2) is satisfied.

MC68020 I/O

The 68020 I/O handling features are very similar to those of the 68000. This means that the 68020 uses memory-mapped I/O, and the 68230 I/O chip can be used for programmed I/O. The external interrupts are handled via the 68020 $\overline{\text{IPL2}}$, $\overline{\text{IPL1}}$, and $\overline{\text{IPL0}}$ pins using autovectoring and nonautovectoring pins. However, the 68020 uses a new pin called $\overline{\text{AVEC}}$ rather than $\overline{\text{VPA}}$ (68HC000) for autovectoring. Nonautovectoring is handled using $\overline{\text{DSACK0}}$ = 0 and $\overline{\text{DSACK1}}$ = 0 rather than $\overline{\text{DTACK0}}$= 0 (as with the 68HC000). Note that the 68020 does not have the $\overline{\text{VPA}}$ pin. Like the 68HC000, the 68020 uses the $\overline{\text{BR}}$, $\overline{\text{BG}}$, and $\overline{\text{BGACK}}$ pins for DMA transfer. The 68020 exceptions are similar to those of the 68000 with some variations such as coprocessor exceptions.

11.7.2 Motorola MC68030

The MC68030 is a virtual memory microprocessor based on the MC68020 with additional features. The MC68030 is designed by using HCMOS technology and can be operated at clock rates of 16.67 and 33 MHz. The MC68030 contains all features of the MC68020, plus some additional ones. The basic differences between the MC68020 and MC68030 are as follows:

Characteristics	*MC 68020*	*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.

11.7.3 Motorola MC68040 / MC68060

This section presents an overview of the Motorola MC68040 and MC 68060 32-bit microprocessors. The MC68040 is Motorola's enhanced 68030, 32-bit microprocessor, implemented in HCMOS technology. Providing balance between speed, power, and physical device size, the MC68040 integrates on-chip 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 an independent 4 KB 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 MC68040 also includes 32-bit nonmultiplexed external address and data buses.
The MC68060 is a superscalar (two instructions per cycle) 32-bit microprocessor. The 68060, like the Pentium, is designed using a combination of RISC and CISC architectures to obtain high performance. For some reason, Motorola does not offer MC68050 microprocessor. The 68060 is fully compatible with the 68040 in the user mode. The 68060 can operate at 50- and 66-MHz clocks with performance much faster than the 68040. An striking feature of the 68060 is the power consumption control. The 68060 is designed using static HCMOS to reduce power during normal operation.

11.7.4 PowerPC Microprocessor

This section provides an overview of the hardware, software, and interfacing features associated with the RISC microprocessor called the PowerPC. Finally, the basic features of both 32-bit and 64-bit PowerPC microprocessors are discussed

Basics of RISC

RISC is an acronym for Reduced Instruction Set Computer. This type of microprocessor emphasizes simplicity and efficiency. RISC designs start with a necessary and sufficient instruction set. The purpose of using RISC architecture is to maximize speed by reducing clock cycles per instruction. Almost all computations can be obtained from a few simple operations. The goal of RISC architecture is to maximize the effective speed of a design by performing infrequent operations in software and frequent functions in hardware, thus obtaining a net performance gain. The following summarizes the typical features of a RISC microprocessor:

1. The RISC microprocessor is designed using hardwired control with little or no microcode. Note that variable-length instruction formats generally require microcode design. All RISC instructions have fixed formats, so microcode design is not necessary.
2. A RISC microprocessor executes most instructions in a single cycle.
3. The instruction set of a RISC microprocessor typically includes only register, load, and store instructions. All instructions involving arithmetic operations use registers, and load and store operations are utilized to access memory.
4. The instructions have a simple fixed format with few addressing modes.
5. A RISC microprocessor has several general-purpose registers and large cache memories.
6. A RISC microprocessor processes several instructions simultaneously and thus includes pipelining.
7. Software can take advantage of more concurrency. For example, Jumps occur after execution of the instruction that follows. This allows fetching of the next instruction during execution of the current instruction.

RISC microprocessors are suitable for embedded applications. Embedded microprocessors or controllers are embedded in the host system. This means that the presence and operation of these controllers are basically hidden from the host system. Typical embedded control applications include office automation systems such as laser

printers. Since a laser printer requires a high performance microprocessor with on-chip floating-point hardware, RISC microprocessors such as PowerPC are ideal for these types of applications.

RISC microprocessors are well suited for applications such as image processing, robotics, graphics, and instrumentation. The key features of the RISC microprocessors that make them ideal for these applications are their relatively low level of integration in the chip and instruction pipeline architecture. These characteristics result in low power consumption, fast instruction execution, and fast recognition of interrupts. Typical 32- and 64-bit RISC microprocessors include PowerPC microprocessors.

IBM/Motorola/Apple PowerPC 601

This section provides an overview of the basic features of PowerPC microprocessors. The PowerPC 601 was jointly developed by Apple, IBM, and Motorola. It is available from IBM as PP 601 and from Motorola as MPC 601. The PowerPC 601 is the first implementation of the PowerPC family of Reduced Instruction Set Computer (RISC) microprocessors. There are two types of PowerPC implementations: 32-bit and 64-bit. The PowerPC 601 implements the 32-bit portion of the IBM PowerPC architectures and Motorola 88100 bus control logic. It includes 32-bit effective (logical) addresses, integer data types of 8, 16, and 32 bits, and floating-point data types of 32 and 64 bits. For 64-bit PowerPC implementations, the PowerPC architecture provides 64-bit integer data types, 64-bit addressing, and other features necessary to complete the 64-bit architecture.

The 601 is a pipelined superscalar processor and is capable of executing three instructions per clock cycle. A pipelined processor is one in which the processing of an instruction is broken down into discrete stages, such as decode, execute, and write-back (the result of the operation is written back in the register file).

Because the tasks required to process an instruction are broken into a series of tasks, an instruction does not require the entire resources of an execution unit. For example, after an instruction completes the decode stage, it can pass on to the next stage, and the subsequent instruction can advance into the decode stage. This improves the throughput of the instruction flow. For example, it may take three cycles for an integer instruction to complete, but if there are no stalls in the integer pipeline, a series of integer instructions can have a throughput of one instruction per cycle. Each unit is kept busy in each cycle.
A superscalar processor is one in which multiple pipelines are provided to allow instructions to execute in parallel. The PowerPC 601 includes three execution units: a 32-bit integer unit (IU), a branch processing unit (BPU), and a pipelined floating-point unit (FPU).

The PowerPC 601 contains an on-chip, 32 KB unified cache (combined instruction and data cache) and an on-chip memory management unit (MMU). It has a 64-bit data bus and a 32-bit address bus. The 601 supports single-beat and four-beat burst data transfer for memory accesses. Note that a single-beat transaction indicates data transfer of up to 64 bits. The PowerPC 601 uses memory-mapped I/O. Input/output devices can also be interfaced to the PowerPC 601 by using the I/O controller. The 601 is designed by using an advanced, CMOS process technology and maintains full compatibility with TTL devices.

The PowerPC 601 contains an on-chip real-time clock (RTC). The RTC was normally an I/O device completely outside the CPU in earlier microcomputers. Although the RTC appearing inside the microcomputer chip is common on single-chip microcomputers, this is the first time the RTC is implemented inside a top-of-the-line microprocessor such as the PowerPC. This implication is that modern multitasking operating systems require time keeping for task switching as well as keeping the calendar date. The 601 real-time

clock (RTC) on-chip hardware provides a measure of real time in terms of time of day and date, with a calendar range of 136.19 years.

To specify the ordering of four bytes (ABCD) within 32 bits, the 601 can use either the ABCD (big-endian) or DCBA (little-endian) ordering. The 601 big- or little-endian modes can be selected by setting the LM bit (bit 28) in the HID0 register. Note that big-endian ordering (ABCD) assigns the lowest address to the highest-order eight bits of the multibyte data. On the other hand, little-endian byte ordering (DCBA) assigns the lowest address to the lowest order (rightmost) 8 bits of the multibyte data.

Note that Motorola 68XXX microprocessors support big-endian byte ordering whereas Intel 80XXX microprocessors support little-endian byte ordering.

PowerPC 601 Registers

PowerPC 601 registers can be accessed depending on the program's access privilege level (supervisor or user mode). The privilege level is determined by the privilege level (PR) bit in the machine status register (MSR). The supervisor mode of operation is typically used by the operating system, and user mode is used by the application software. The PowerPC 601 programming model contains user- and supervisor-level registers. Some of these are

- The user-level register can be accessed by all software with either user or supervisor privileges.
- The 32-bit GPRs (general-purpose registers, GPR0–GPR31) can be used as the data source or destination for all integer instructions. They can also provide data for generating addresses.
- The 32-bit FPRs (floating-point registers, FPR0–FPR31) can be used as data sources and destinations for all floating-point instructions.
- The floating-point status and control register (FPCSR) is a user control register in the floating-point unit (FPU). It contains floating-point status and control bits such as floating-point exception signal bits, exception summary bits, and exception enable bits.
- The condition register (CR) is a 32-bit register, divided into eight 4-bit fields, CR0–CR7. These fields reflect the results of certain arithmetic operations and provide mechanisms for testing and branching.
  The remaining user-level registers are 32-bit special purpose registers—SPR0, SPR1, SPR4, SPR5, SPR8, and SPR9.
- SPR0 is known as the MQ register and is used as a register extension to hold the product for the multiplication instructions and the dividend for the divide instructions. The MQ register is also used as an operand of long shift and rotate instructions.
- SPR1 is called the integer exception register (XER). The XER is a 32-bit register that indicates carries and overflow bits for integer operations. It also contains two fields for load string and compare byte indexed instructions.
- SPR4 and SPR5 respectively represent two 32-bit read only registers and hold the upper (RTCU) and lower (RTCL) portions of the real-time clock (RTC). The RTCU register maintains the number of seconds from a time specified by software. The RTCL register maintains the fraction of the current second in nanoseconds.
- SPR8 is the 32-bit link register (LR). The link register can be used to provide the branch target address and to hold the return address after branch and link instructions.

- SPR9 represents the 32-bit count register (CTR). The CTR can be used to hold a loop count that can be decremented during execution of certain branch instructions. The CTR can also be used to hold the target address for the branch conditional to count register instruction.

PowerPC 601 Addressing Modes

The effective address (EA) is the 32-bit address computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction. Since the PowerPC is based on the RISC architecture, arithmetic and logical instructions do not read or modify memory.

Load and store operations have two types of effective address generation:

i) ***Register Indirect with Immediate Index Mode***

Instructions using this mode contain a signed 16-bit index (d operand in the 32-bit instruction) which is sign extended to 32-bits, and added to the contents of a general-purpose register specified by five bits in the 32-bit instruction (rA operand) to generate the effective address. A zero in the rA operand causes a zero to be added to the immediate index (d operand). The option to specify rA or 0 is shown in the instruction descriptions of the 601 user's manual as the notation (rA|0).

An example is `lbz` rD,d (rA) where rA specifies a general-purpose register (GPR) containing an address, d is the the 16-bit immediate index and rD specifies a general-purpose register as destination. Consider `lbz r1,20(r3)` . The effective address (EA) is the sum r3+20. The byte in memory addressed by the EA is loaded into bits 31 through 24 of register r1. The remaining bits in r1 are cleared to zero. Note that the registers r1 and r3 represent GPR1 and GPR3 respectively.

ii) ***Register Indirect with Index Mode***

Instructions using this addressing mode add the contents of two general-purpose registers (one GPR holds an address and another holds the index). An example is `lbzx` rD, rA, rB where rD specifies a GPR as destination, rA specifies a GPR as the index, and rB specifies a GPR holding an address. Consider `lbzx r1,r4,r6`. The effective address (EA) is the sum (r4|0)+(r6). The byte in memory adressed by the EA is loaded into register r1 (24-31). The remaining bits in register rD are cleared to zero.

PowerPC 601 conditional and unconditional branch instructions compute the effective address (EA) or the next instruction address using various addressing modes A few of them are described below:

- ***Branch Relative*** Branch instructions (32-bit wide) using the relative mode generate the address of the next instruction by adding an offset and the current program counter contents. An example of this mode is an instruction `be start` unconditionally jumps to the address PC + start.
- ***Branch Absolute*** Branch instructions using this mode include the address of the next instruction to be executed. For example, the instruction `ba begin` unconditionally branches to the absolute address "begin" specified in the instruction.
- ***Branch to Link Register*** Branch instructions using this mode branch to the address computed as the sum of the immediate offset and the address of the current instruction. The instruction address following the instruction is placed into the link register. For example, the instruction `bl,start` unconditionally jumps to the address computed from current PC contents plus start. The return

address is placed in the link register.

- ***Branch to Count Register*** Instructions using this mode branch to the address contained in the current register. Consider `bcttr BO, BI` means branch conditional to count register. This instruction branches conditionally to the address specified in the count register.

  The BI operand specifies the bit in the condition register to be used as the condition of the branch. The BO operand specifies how the branch is affected by or affects condition or count registers. Numerical values specifying BI and BO can be obtained from the 601 manual.

Note that some instructions combine the link register and count register modes. An example is `bcctr BO, BI`. This instruction first performs the same operation as the bcttr and then places the instruction address following the instruction into the link register. This instruction is a form of "conditional call" because the return address is saved in the link register.

Typical PowerPC 601 Instructions

The 601 instructions are divided into the following categories:

1. Integer Instructions
2. Floating-point Instructions
3. Load/store Instructions
4. Flow Control Instructions
5. Processor Control Instructions

Integer instructions operate on byte (8-bit), half-word (16-bit), and word (32-bit) operands. Floating-point instructions operate on single-precision and double-precision floating-point operands.

Integer Instructions

The integer instructions include integer arithmetic, integer compare, integer rotate and shift, and integer logical instructions. The integer arithmetic instructions always set the integer exception register bit, CA, to reflect the carry out of bit 7. Integer instructions with the overflow enable (OE) bit set will cause the XER bits SO (summary overflow — overflow bit set due to exception) and OV (overflow bit set due to instruction execution) to be set to reflect overflow of the 32-bit result. Some examples of integer instructions are provided in the following. Note that rS, rD, rA, and rB in the following examples are 32-bit general purpose registers (GPRs) of the 601 and SIMM is 16-bit signed immediate number.

- `add rD,rA,SIMM` performs the following immediate operation: rD ← (rA|0) + SIMM; rA|0) can be either (rA) or 0. An example is `add rD, rA, SIMM` or `add rD, 0, SIMM`.
- `add rD,rA,rB` performs rD ← rA + rB.
- `add.rD,rA,rB` adds with CR update as follows: rD ← rA + rB. The dot suffix enables the update of the condition register.
- `subf rD,rA,rB` performs rD ← rB - rA.
- `sub rD,rA,rB` performs the same operation as subf but updates the condition code register.
- `addme rD,rA` performs the (add to minus one extended) operation: rD ← (rA) + FFFF FFFFH + CA bit in XER.
- `subfme rD,rA` performs the (subtract from minus one extended) operation: rD ←

($\overline{rA}$) + FFFF FFFFH + CA bit in XER, where ($\overline{rA}$) represents the ones complement of the contents of rA.

- `mulhwu rD,rA,rB` performs an unsigned multiplication of two 32-bit numbers in rA and rB. The high-order 32 bits of the 64-bit product are placed in rD.
- `mulhw rD,rA,rB` performs the same operation as the mulhwu except that the multiplication is for signed numbers.
- `mullw rD,rA,rB` places the low order 32-bits of the 64-bit product (rA)*(rB) into rD. The low-order 32-bit products are independent whether the operands are treated as signed or unsigned integers.
- `mulli rD,rA,SIMM` places the low-order 32 bits of the 48-bit product (rA)*$SIMM_{16}$ into rD. The low-order bits of the 32-bit product are independent whether the operands are treated as signed or unsigned integers.
- `divw rD,rA,rB` divides the 32-bit signed dividend in rA by the 32-bit signed divisor in rB. The 32-bit quotient is placed in rD and the remainder is discarded.
- `divwu rD,rA,rB` is the same as the divw instruction except that the division is for unsigned numbers.
- `cmpi crfD,L,rA,SIMM` compares 32 bits in rA with immediate SIMM treating operands as signed integer. The result of comparison is placed in crfd field (0 for CR0, 1 for CR1, and so on) of the condition register. L=0 indicates 32-bit operands while L=1 represents the 64-bit operands. For example, `cmpi 0,0, rA, 200` compares 32 bits in register rA with immediate value 200 and CR0 is affected according to the comparison.
- `xor rA,rS,rB` performs exclusive-or operation between the contents of rS and rB. The result is placed into register rA.
- `extsb rA,rS` places bits 24-31 of rS into bits 24-31 of rA. Bit 24 of rS is then sign extended through bits 0-23 of rA.
- `slw rA,rS,rB` shifts the contents of rS left by the shift count specified by rB [27-31]. Bits shifted out of position 0 are lost. Zeros are placed in the vacated positions on the right. The 32-bit result is placed into rA.
- `srw rA,rS,rB` is similar to `slw rA,rS,rB` except that the operation is for right shift.

Floating-Point Instructions

Some of the 601 floating-point instructions are provided below:

- `fadd frD,frA,frB` adds the contents of the floating-point register, frA to the contents of the floating-point register frB. If the most significant bit of the resultant significand is not a one, then the result is normalized. The result is rounded to the specified position under control of the FPSCR register. The result is rounded to the specified precision under control of the FPSCR register. The result is then placed in frD.

  Note that this fadd instruction requires one cycle in execute stage, assuming normal operations; however, there is an execute stage delay of three cycles if the next instruction is dependent.

  The 601 floating point addition is based on "exponent comparison and add by one" for each bit shifted, until the two exponents are equal. The two significands are then added algebraically to form an intermediate sum. If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one.
- `fsub frD,frA,frB` performs frA – frB, normalization, and rounding of the result

are performed in the same way as the `fadd` instruction.

- `fmul frD,frA,frC` performs frD ← frA * frC. Normalization and rounding of the result are performed in the same way as the fadd. Floating-point multiplication is based on exponent addition and multiplication of the significands.
- `fdiv frD,frA,frB` performs the floating-point division frD ← frA/frB. No remainder is provided. Normalization and rounding of the result are performed in the same way as the `fadd` instruction.
- `fmsub frD,frA,FrC,frB` performs frD ← frA * frC – frB. Normalization and rounding of the result are performed in the same way as the `fadd` instruction.

Load/Store Instructions

Some examples of the 601 load and store instructions are

- `lhzx rD,rA,rB` loads the half word (16 bits) in memory addressed by the sum (rA|0) + (rB) into bits 16 through 31 of rD. The remaining bits of rD are cleared to zero.
- `sthux rS,rA,rB` stores the 16-bit half word from bits 16–31 of register rS in memory addressed by the sum (rA|0) + (rB). The value (rA|0) + rB is placed into register rA.
- `lmw rD,d(rA)` loads *n* (where $n = 32 - D$ and $D = 0$ through 31) consecutive words starting at memory location addressed by the sum (r|0) + d into the general-purpose register specified by rD through r31.
- `stmu rS,d(rA)` is similar to `lmw` except that `stmw` stores *n* consecutive words.

Flow Control Instructions

Flow control instructions include conditional and unconditional branch instructions. An example of one of these instructions is

- `bc` (branch conditional) `BO, BI,` target branch with offset target if the condition bit in CR specified by bit number BI is true (The condition "true" is specified by a value in BO).

  For example, `bc 12,0,target` means that branch with offset target if the condition specified by bit 0 in CR (BI = 0 indicates the result is negative) is true (specified by the value BO = 12 according to Motorola PowerPC 601 manual).

Processor Control Instructions

Processor control instructions are used to read from and write to the machine state register (MSR), condition register (CR), and special status register (SPRs). Some examples of these instructions are

- `mfcr rD` places the contents of the condition register into rD.
- `mtmsr rS` places the contents of rS into the MSR. This is a supervisor-level instruction.
- `mfimsr rD` places the contents of MSR into rD. This is a supervisor-level instruction.

PowerPC 601 Exception Model

All 601 exceptions can be described as either precise or imprecise and either synchronous or asynchronous. Asynchronous exceptions are caused by events external to the processor's execution. Synchronous exceptions, on the other hand, are handled precisely by the 601 and are caused by instructions; precise exception means that the machine state at the time the exception occurs is known and can be completely restored. That is, the instructions

that invoke trap and system call exceptions complete execution before the exception is taken. When exception processing completes, execution resumes at the address of the next instruction.

An example of a maskable asynchronous, precise exception is the external interrupt. When an asynchronous, precise exception such as the external interrupt occurs, the 601 postpones its handling until all instructions and any exceptions associated with those instructions complete execution. System reset and machine check exceptions are two nonmaskable exceptions that are asynchronous and imprecise. These exceptions may not be recoverable or may provide a limited degree of recoverability for diagnostic purpose.

Asynchronous, imprecise exceptions have the highest priority with the synchronous, precise exceptions having the next priority and the asynchronous, precise exceptions the lowest priority.

The 601 exception mechanism allows the processor to change automatically to supervisor state as a result of exceptions. When exceptions occur, information about the state of the processor is saved to certain registers rather than in memory as is usually done with other processors in order to achieve high speeds. The processor then begins execution at an address (exception vector) predetermined for each exception. The exception handler at the specified vector is then processed with processor in supervisor mode.

601 System Interface

The pins and signals of the PowerPC 601 include a 32-bit address bus and 52 control and information signals. Memory access allows transfer sizes of 8, 16, 24, 32, 40, 48, 56, or 64 bits in one bus clock cycle. Data transfer occurs in either single-beat transactions or four-beat burst transactions. Both memory and I/O accesses can use the same bus transfer protocols. The 601 also has the ability to define memory areas as I/O controller interface areas. The 601 uses the $\overline{TS}$ pin for memory-mapped accesses and the XATS pin for I/O controller interface accesses.

Summary of PowerPC 601 Features

The PowerPC 601 is a RISC-based superscalar microprocessor. That is, it can execute two or more instructions per cycle. The PowerPC 601 is based on load/store architectures. This means that all instructions that access memory are either loads or stores, and all operate instructions are from register to register. Both load and store instructions have 32-bit fixed-length instructions along with 32-bit integer and 32-bit floating-point registers.

The PowerPC 601 includes two primary addressing modes: register plus displacement and register plus register. In addition, the 601 load and store instructions perform the load or store operation and also modify the index register by placing the effective address just computed. In the PowerPC 601, Branch target addresses are normally determined by using program counter relative mode. That is, the branch target address is determined by adding a displacement to the program counter. However, as mentioned before, conditional branches in the 601 may test fields in the condition code register and the contents of a special register called the count register (CTR). A single 601 branch instruction can implement a loop-closing branch by decrementing the CTR, testing its value, and branching if it is nonzero.

The PowerPC 601 saves the return address for certain control transfer instructions such as subroutine call in a general-purpose register. The 601 does this in any branch by setting the link (LK) bit to one. The return address is saved in the link register. The PowerPC 601 utilizes sophisticated pipelines. The 601 uses relatively short independent

TABLE 11.14 PowerPC 601 vs. 620

Features	*PowerPC 601*	*PowerPC 620*
Technology	HCMOS	HCMOS
Transistor count	2.8 million	7 million
Clock speed	50 MHz, 66 MHz	133 MHz
Size of the microprocessor	32-bit	64-bit
Address bus	32-bit	40-bit
Data bus	64-bit	128-bit

pipelines with more buffering. The 601 does a lot of computation in each pipe stage. The 601 has a unified (combined) 32 KB cache. That is, instructions and data reside in the same cache in the 601. Finally, the 601 offers high performance by utilizing sophisticated design tricks. For example, the 601 includes powerful instructions such as floating-point multiply-add and update load/store that perform more tasks with fewer instructions.

PowerPC 64-Bit Microprocessors

PowerPC 64-bit microprocessors include the PowerPC 620, 603e, 750/740, and 604e. These microprocessors are 64-bit superscalar processors. This means that they can execute more than one instruction in a cycle. Table 11.14 compares the basic features of the 32-bit PowerPC 601 with the 64-bit PowerPC 620.

There are a few versions of the 64-bit PowerPC available: PowerPC 603e, PowerPC 750/740, and PowerPC 604e. The PowerPC 603e microprocessor is available at speeds of 250, 275, and 300 MHz. The 603e has high performance and low power consumption, which makes it suited for applications found in the embedded system market. The PowerPC 603e is used in the Power Macintosh C500 series, which offers features such as accelerated multimedia, advanced video capture, and publishing. The PowerPC 750/740 is available at speeds up to 266 MHz and uses only 5 watts of power. The unique features offered by this microprocessor are built-in power-saving modes, an on-chip thermal sensor to regulate processor temperature, and a choice of packaging configurations. The PowerPC 604e microprocessor, another member of the PowerPC family, provides speeds of 350 MHz and using 8.0 watts of power. Like Intel, Motorola used the 0.25 micron process technology to achieve this speed. The PowerPC 604e is intended for high-end Macintosh and Mac-compatible systems.

Apple Computer's original G3 (Marketing name used by Apple) utilized PowerPC 750 for Apple's iMac and Power Macintosh personal computers. Apple's G3 (later version) used Motorola's copper-based PowerPC microprocessor, providing speed of up to 400 MHz.

11.7.5 Motorola's State-of-the-art Microprocessors

As part of their plans to carry the PowerPC architecture into the future, Motorola /IBM/ Apple already announced AltiVec extensions for the PowerPC family. The result is the MPC7400 PowerPC microprocessor. This microprocessor is available in 400 MHz, 450 MHz and 500 MHz clock speeds. Motorola's AltiVec technology is the foundation for the Velocity Engine of Apple Computer's next generation desktop computers. For example, Apple rececently announced Power Mac G5 which uses Motorola's 64-bit microprocessor, G5. AltiVec extensions are somewhat comparable to the MMX extensions in Intel's Pentium family. AltiVec has independent processing units while Intel tied MMX to the floating-point unit. Both utilize SIMD (Chapter 8). A comparison of some of the features

of AltiVec vs. MMX is provided below:

Features	**AltiVec**	**MMX**
Size	128 bits at a time	64 bits at a time
Instructions	162 instructions	57 instructions
Registers	32 registers	8 registers
Unit	Independent	tied to Floating-point Unit

In AltiVec, each processing unit can work independent of the others. This provides more parallelism by separate units. Since Intel tied MMX to floating-point unit, Pentiums can perform either floating-point math or switch over to MMX, but not both simultaneously. The switch requires a mode change that can cost hundreds of cycles, both going into and coming out of MMX mode. It may be very tricky with Pentiums to write good and efficient codes when mixing of modes are required in some computing algorithms.

AltiVec can vetorize the floating-point operations. This means that one can use AltiVec to work on some data in the Floating-point Unit, then load the data in the AltiVec side (Vector Unit) without any significant mode switch. This may save hundreds of cycles . Also, this allows programmers to do more with the Vector Unit since they can go back and forth to mix and match.

The biggest drawback with MMX or AltiVec is getting programmers to use them. Programmers are required to use assembly language for MMX. Therefore, a few programmers used MMX for dedicated applications. For example, Intel hand tuned some photoshop filters for Adobe. Programmers can use C language with AltiVec. Therefore, it is highly likely that more programmers will use AltiVec than MMX.

In the future, Motorola and IBM plan to introduce the PowerPC series 2K. It is expected that the chip will contain 100 million transistors and have clock speeds greater than 1 GHz.

QUESTIONS AND PROBLEMS

11.1 Discuss the typical features of 32-bit and 64-bit microprocessors.

11.2 (a) What is the basic difference between the 80386 and 80386SX?
(b) What is the basic difference between the 80386 and 80486?

11.3 What is the difference between the 80386 protected, real-address, and virtual 8086 modes?

11.4 Discuss the basic features of the 80486.

11.5 Assume the following 80386 register contents

(EBX) = 00001000H
(ECX) = 04000002H
(EDX) = 20005000H

prior to execution of each of the following 80386 instructions. Determine the contents of the affected registers and/or memory locations after execution of each of the following instructions and identify the addressing modes:

```
(a)MOV [EBX * 4] [ECX], EDX
(b)MOV [EBX * 2] [ECX + 2020H], EDX
```

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

(a) `MOVZX EAX, CH`

Prior to execution of this `MOVZX` instruction, assume

(EAX) = 80001234H

(ECX) = 00008080H

(b) `MOVSX EDX, BL`

Prior to execution of this `MOVSX` assume

(EDX) = FFFFFFFFH

(EBX) = 05218888H

11.7 Write an 80386 assembly program to add a 64-bit number in ECX: EDX with another 64-bit number in EAX: EBX. Store the result in EAX: EBX.

11.8 Write an 80386 assembly program to divide a signed 32-bit number in DX:AX by an 8-bit signed number in BH. Store the 16-bit quotient and 16-bit remainder in AX and DX respectively.

11.9 Write an 80386 assembly program to compute

$$\sum_{i=1}^{N} X_i^2$$

where $N = 1000$ and the X_i's are signed 32-bit numbers.
Assume that ΣX_i^2 can be stored as a 32-bit number.

11.10 Discuss 80386 I/O.

11.11 Compare the on-chip hardware features of the 80486 and Pentium microprocessors.

11.12 What are the sizes of the address and data buses of the 80486 and the Pentium?

11.13 Identify the main differences between the 80486 and the Pentium.

11.14 What are the clock speed, pipeline model, number of on-chip transistors, and number of pins on the 80486 and Pentium processors?

11.15 Discuss typical applications of Pentium.

11.16 Identify the main differences between the Intel 80386 and 80486.

11.17 What is meant by the 80486 BUS BACKOFF feature?

11.18 How many pipeline stages are in Pentium and Pentium Pro?

11.19 How many new instructions are added to the 80486 beyond those of the 80386?

11.20 Given the following register 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
```

11.21 If (EBX) = 0123A212H and (EDX) = 46B12310H, then what are the contents of EBX and EDX after execution of the 80486 instruction `XADD EBX,EDX`?

11.22 If (BX) = 271AH, (AX) = 712EH, and (CX) = 1234H, what are the contents of AX after execution of the 80486 instruction `CMPXCHG CX,BX`?

11.23 What are three modes of the Pentium processor? Discuss them briefly.

11.24 What is meant by the statement, "The Pentium processor is based on a superscalar design"?

11.25 What are the purposes of the U pipe and V pipe of the Pentium processor?

11.26 What are the sizes of the data and instruction caches in the Pentium?

11.27 Summarize the basic differences among Pentium, Pentium Pro, and Pentium II, Celeron, Pentium II Xeon, Pentium III, and Pentium III Xeon processors.

11.28 Why are the Pentium Pro's complete capabilities not used by the Windows 95 operating system?

11.29 Summarize the basic features of the Intel/Hewlett-Packard "Merced" microprocessor.

11.30 Summarize the basic differences between the 68000, 68020, 68030, 68040 and 68060.

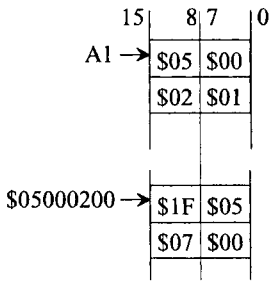
11.31 What is the unique feature of the Power PC microprocessor family?

11.32 Name three new 68020 instructions that are not provided with the 68000.

11.33 Find the contents of the affected registers and memory locations after execution of the 68020 instruction `MOVE ($1000,A5,D3.W*4),D1`. Assume the following data prior to execution of this `MOVE`:

[A5] = $0000F210 , [$ 00014218] = $4567
[D3] = $00001002 , [$ 0001421A] = $2345
[D1] = $F125012A

11.34 Assume the following 68020 memory configuration:



Find the contents of the affected memory locations after execution of `MOVE.W #$1234,([A1])`.

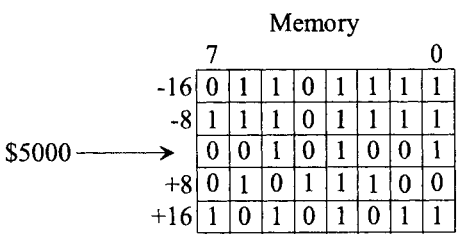
11.35 Find the 68020 compare instruction with the appropriate addressing mode to replace the following 68000 instruction sequence:

```
ASL.L #1,D5
CMP.L 0 (A0,D5.L),D0
```

11.36 Find the contents of D1, D2, A4, and CCR and the memory locations after execution of each of the following 68020 instructions:

```
(a) BFSET $5000 {D1:10}
(b) BFINS D2, (A4) {D1:D4}
```

Assume the data given in Figure P11.36 prior to execution of each of these instructions.



[D1] = $00000004, [D4] = $00000004
[D2] = $12345678, [A4] = $00005000

FIGURE P11.36

11.37 Identify the following 68020 instructions as valid or invalid. Justify your answers.

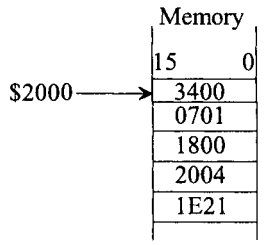
(a) `DIVS A0,D1`
(b) `CHK.B D0,(A0)`
(c) `MOVE.L D0,(A0)`

It is given that [A0] = $1025671A prior to execution of the `MOVE`.

11.38 Determine the values of the Z and C flags after execution of each of the following 68020 instructions:

(a) `CHK2.W (A5),D3`
(b) `CMP2.L $2001,A5`

Assume the following data prior to execution of each of these instructions:



[D3] = $02001740, [A5] = $0002004

11.39 Write a 68020 assembly program to add two 64-bit numbers in D1D0 with another 64-bit number in D2D3. Store the result in D1D0.

11.40 Write a 68020 assembly program to multiply a 32-bit signed number in D5 by another 16-bit signed number in D1. Store the 64-bit result in D5D1.

11.41 Write a subroutine in 68020 assembly language to compute $Y = \sum_{i=1}^{50} \frac{Xi^2}{50}$ Assume the X_i's are signed 32-bit numbers and the array starts at $50000021. Neglect overflow.

11.42 Write a program in 68020 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 that the number of bits to be checked is divisible by 16. If no ones are found, store zero in D3; otherwise store the offset of the first set bit in D3, and then stop. Assume A2 contains the starting address of the array, and D2 contains the number of bits in the array.

11.43 Write a program in 68020 assembly language to multiply a signed byte by a 32-bit signed number to obtain a 64-bit result. Assume that the numbers are respectively pointed to by the addresses that are passed on to the user stack by a subroutine pointed to by (A7+6) and (A7+8). Store the 64-bit result in D2:D1.

11.44 What is meant by 68020 dynamic bus sizing?

11.45 Consider the 68020 instruction `MOVE.B D1,$00000016`. Find the 68020 data pins over which data will be transferred if $\overline{\text{DSACK1}}$ $\overline{\text{DSACK0}}$ = 00. What are the 68020 data pins if $\overline{\text{DSACK1}}$ $\overline{\text{DSACK0}}$ = 10?

11.46 If a 32-bit data is transferred using 68020 `MOVE.L D0,$50607011` instruction to a 32-bit memory with [D0] = $81F27561, how many bus cycles are needed to perform the transfer? What are A_1A_0 equal to during each cycle? What is the SIZ1 SIZ0 code during each cycle? What bytes of data are transferred during each bus cycle?

11.47 Discuss 68020 I/O.

11.48 What do you mean by the unified cache of the 601? What is its size?

11.49 List the user-level and general-purpose registers of the 601.

11.50 Name one supervisor-level register in the 601. What is its purpose?

11.51 How does the 601 MSR indicate the following:
(a) The 601 executes both the user- and supervisor- level instructions.
(b) The 601 executes only the user-level instructions.

11.52 Explain the operation performed by each of the following 601 instructions:

```
(a) add.r1,r2,r3
(b) divwu r2,r3,r4
(c) extsb r1,r2
```

11.53 Discuss briefly the exceptions included in the PowerPC 601.

11.54 Compare the basic features of the 601 with the 620. Discuss PowerPC 64-bit *μp's*.

11.55 Summarize the basic features of Motorola's state-of-the-art microprocessors.