Fundamentals of Digital Logic and Microcomputer Design. M. Rafiquzzaman

6

MICROCOMPUTER ARCHITECTURE, PROGRAMMING, AND SYSTEM DESIGN CONCEPTS

This chapter describes the fundamental material needed to understand the basic characteristics of microprocessors. It includes topics such as typical microcomputer architecture, timing signals, internal microprocessor structure, and status flags. The architectural features are then compared to the Intel 8086 architecture. Topics such as microcomputer programming languages and system design concepts are also described.

6.1 Basic Blocks of a Microcomputer

A microcomputer has three basic blocks: a central processing unit (CPU), a memory unit, and an input/output unit. The CPU executes all the instructions and performs arithmetic and logic operations on data. The CPU of the microcomputer is called the "microprocessor." The microprocessor is typically a single VLSI (Very Large-Scale Integration) chip that contains all the registers, control unit, and arithmetic/ logic circuits of the microcomputer.

A memory unit stores both data and instructions. The memory section typically contains ROM and RAM chips. The ROM can only be read and is nonvolatile, that is, it retains its contents when the power is turned off. A ROM is typically used to store instructions and data that do not change. For example, it might store a table of codes for outputting data to a display external to the microcomputer for turning on a digit from 0 to 9.

One can read from and write into a RAM. The RAM is volatile; that is, it does not retain its contents when the power is turned off. A RAM is used to store programs and data that are temporary and might change during the course of executing a program. An I/O (Input/Output) unit transfers data between the microcomputer and the external devices via I/O ports (registers). The transfer involves data, status, and control signals.

In a single-chip microcomputer, these three elements are on one chip, whereas with a single-chip microprocessor, separate chips for memory and I/O are required. Microcontrollers evolved from single-chip microcomputers. The microcontrollers are typically used for dedicated applications such as automotive systems, home appliances, and home entertainment systems. Typical microcontrollers, therefore, include on-chip timers and A/D (analog to digital) and D/A (digital to analog) converters. Two popular

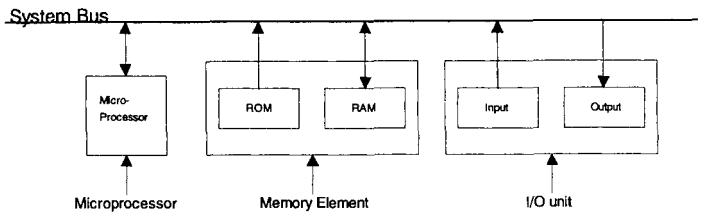


FIGURE 6.1 Basic blocks of a microcomputer

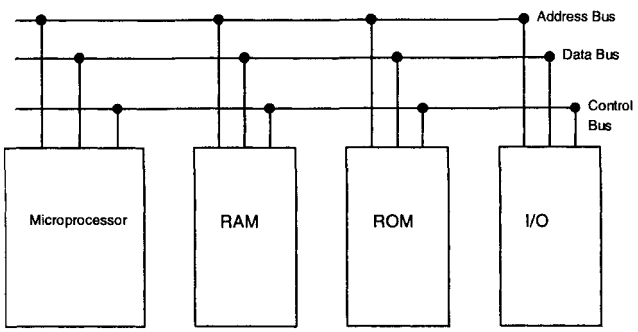


FIGURE 6.2 Simplified version of a typical microcomputer

microcontrollers are the Intel 8751 (8 bit)/8096 (16 bit) and the Motorola HC11 (8 bit)/ HC16 (16 bit). The 16-bit microcontrollers include more on-chip ROM, RAM, and I/O than the 8-bit microcontrollers. Figure 6.1 shows the basic blocks of a microcomputer. The System bus (comprised of several wires) connects these blocks.

6.2 Typical Microcomputer Architecture

In this section, we describe the microcomputer architecture in more detail. The various microcomputers available today are basically the same in principle. The main variations are in the number of data and address bits and in the types of control signals they use.

To understand the basic principles of microcomputer architecture, it is necessary to investigate a typical microcomputer in detail. Once such a clear understanding is obtained, it will be easier to work with any specific microcomputer. Figure 6.2 illustrates the most simplified version of a typical microcomputer. The figure shows the basic blocks of a microcomputer system. The various buses that connect these blocks are also shown. Although this figure looks very simple, it includes all the main elements of a typical microcomputer system.

6.2.1 The Microcomputer Bus

The microcomputer's system bus contains three buses, which carry all the address, data, and control information involved in program execution. These buses connect the microprocessor (CPU) to each of the ROM, RAM, and I/O chips so that information transfer between the microprocessor and any of the other elements can take place.

In the microcomputer, typical information transfers are carried out with respect to the memory or I/O. When a memory or an I/O chip receives data from the microprocessor , it is called a WRITE operation, and data is written into a selected memory location or an I/O port (register). When a memory or an I/O chip sends data to the microprocessor,

it is called a READ operation, and data is read from a selected memory location or an I/O port.

In the *address bus*, information transfer takes place only in one direction, from the microprocessor to the memory or I/O elements. Therefore, this is called a "unidirectional bus." This bus is typically 20 to 32 bits long. The size of the address bus determines the total number of memory addresses available in which programs can be executed by the microprocessor. The address bus is specified by the total number of address pins on the microprocessor chip. This also determines the direct addressing capability or the size of the main memory of the microprocessor. The microprocessor can only execute the programs located in the main memory. For example, a microprocessor with 20 address pins can generate 2^{20} = 1,048,576 (one megabyte) different possible addresses (combinations of 1's and 0's) on the address bus. The microprocessor includes addresses from 0 to 1,048,575 (00000_{16} through $FFFFF_{16}$). A memory location can be represented by each one of these addresses. For example, an 8-bit data item can be stored at address 00200_{16}.

When a microprocessor such as the 8086 wants to transfer information between itself and a certain memory location, it generates the 20-bit address from an internal register on its 20 address pins A_0–A_{19}, which then appears on the address bus. These 20 address bits are decoded to determine the desired memory location. The decoding process normally requires hardware (decoders) not shown in Figure 6.2.

In the *data bus*, data can flow in both directions, that is, to or from the microprocessor. Therefore, this is a bidirectional bus. In some microprocessors, the data pins are used to send other information such as address bits in addition to data. This means that the data pins are time-shared or multiplexed. The Intel 8086 microprocessor is an example where the 20 bits of the address are multiplexed with the 16-bit data bus and four status lines.

The *control bus* consists of a number of signals that are used to synchronize the operation of the individual microcomputer elements. The microprocessor sends some of these control signals to the other elements to indicate the type of operation being performed. Each microcomputer has a unique set of control signals. However, there are some control signals that are common to most microprocessors. We describe some of these control signals later in this section.

6.2.2 Clock Signals

The system clock signals are contained in the control bus. These signals generate the appropriate clock periods during which instruction executions are carried out by the microprocessor. The clock signals vary from one microprocessor to another. Some microprocessors have an internal clock generator circuit to generate a clock signal. These microprocessors require an external crystal or an RC network to be connected at the appropriate microprocessor pins for setting the operating frequency. For example, the Intel 80186 (16-bit microprocessor) does not require an external clock generator circuit. However, most microprocessors do not have the internal clock generator circuit and require an external chip or circuit to generate the clock signal. Figure 6.3 shows a typical clock signal.

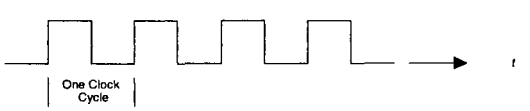


FIGURE 6.3 A typical clock signal

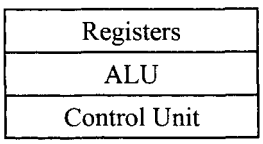


FIGURE 6.4 A microprocessor chip with the main functional elements

6.3 The Single-Chip Microprocessor

As mentioned before, the microprocessor is the CPU of the microcomputer. Therefore, the power of the microcomputer is determined by the capabilities of the microprocessor. Its clock frequency determines the speed of the microcomputer. The number of data and address pins on the microprocessor chip make up the microcomputer's word size and maximum memory size. The microcomputer's I/O and interfacing capabilities are determined by the control pins on the microprocessor chip.

The logic inside the microprocessor chip can be divided into three main areas: the register section, the control unit, and the arithmetic and logic unit (ALU). A microprocessor chip with these three sections is shown in Figure 6.4. We now describe these sections.

6.3.1 Register Section

The number, size, and types of registers vary from one microprocessor to another. However, the various registers in all microprocessors carry out similar operations. The register structures of microprocessors play a major role in designing the microprocessor architectures. Also, the register structures for a specific microprocessor determine how convenient and easy it is to program this microprocessor.

We first describe the most basic types of microprocessor registers, their functions, and how they are used. We then consider the other common types of registers.

Basic Microprocessor Registers

There are four basic microprocessor registers: instruction register, program counter, memory address register, and accumulator.

- **Instruction Register (IR).** The instruction register stores instructions. The contents of an instruction register are always decoded by the microprocessor as an instruction. After fetching an instruction code from memory, the microprocessor stores it in the instruction register. The instruction is decoded internally by the microprocessor, which then performs the required operation. The word size of the microprocessor determines the size of the instruction register. For example, a 16-bit microprocessor has a 16-bit instruction register.
- **Program Counter (PC).** The program counter contains the address of the instruction or operation code (op-code). The program counter normally contains the address of the next instruction to be executed. Note the following features of the program counter:
  1. Upon activating the microprocessor's RESET input, the address of the first instruction to be executed is loaded into the program counter.
  2. To execute an instruction, the microprocessor typically places the contents of the program counter on the address bus and reads ("fetches") the contents of this address, that is, instruction, from memory. The program counter contents are automatically incremented by the microprocessor's internal logic. The microprocessor thus executes a program sequentially, unless the program contains an instruction such as a `JUMP` instruction, which changes the sequence.
  3. The size of the program counter is determined by the size of the address bus.

4. Many instructions, such as JUMP and conditional JUMP, change the contents of the program counter from its normal sequential address value. The program counter is loaded with the address specified in these instructions.

- **Memory Address Register (MAR).** The memory address register contains the address of data. The microprocessor uses the address, which is stored in the memory address register, as a direct pointer to memory. The contents of the address consists of the actual data that is being transferred.
- **Accumulator (A).** For an 8-bit microprocessor, the accumulator is typically an 8-bit register. It is used to store the result after most ALU operations. These microprocessors have instructions to shift or rotate the accumulator 1 bit to the right or left through the carry flag. The accumulator is typically used for inputting a byte into the accumulator from an external device or outputting a byte to an external device from the accumulator. Some microprocessors, such as the Motorola 6809, have more than one accumulator. In these microprocessors, the accumulator to be used by the instruction is specified in the op-code.

Depending on the register section, the microprocessor can be classified either as an accumulator-based or a general-purpose register-based machine. In an accumulator-based microprocessor such as the Intel 8085 and Motorola 6809, the data is assumed to be held in a register called the "accumulator." All arithmetic and logic operations are performed using this register as one of the data sources. The result after the operation is stored in the accumulator. Eight-bit microprocessors are usually accumulator based.

The general-purpose register-based microprocessor is usually popular with 16-, 32-, and 64-bit microprocessors, such as the Intel 8086/80386/80486/Pentium and the Motorola 68000/68020/68030/68040/PowerPC. The term "general-purpose" comes from the fact that these registers can hold data, memory addresses, or the results of arithmetic or logic operations. The number, size, and types of registers vary from one microprocessor to another.

Most registers are general-purpose whereas some, such as the program counter (PC), are provided for dedicated functions. The PC normally contains the address of the next instruction to be executed. As metioned before, upon activating the microprocessor chi p's RESET input pin, the PC is normally initialized with the address of the first instruction. For example, the 80486, upon hardware reset, reads the first instruction from the 32-bit hex address FFFFFFF0. To execute the instruction, the microprocessor normally places the PC contents on the address bus and reads (fetches) the first instruction from external memory. The program counter contents are then automatically incremented by the ALU. The microcomputer thus usually executes a program sequentially unless it encounters a jump or branch instruction. As mentioned earlier, the size of the PC varies from one microprocessor to another depending on the address size. For example, the 68000 has a 24-bit PC, whereas the 68040 contains a 32-bit PC. Note that in general-purpose register-based microprocessors, the four basic registers typically include a PC, an MAR, an IR, and a data register.

Use of the Basic Microprocessor Registers

To provide a clear understanding of how the basic microprocessor registers are used, a binary addition program will be considered. The program logic will be explained by showing how each instruction changes the contents of the four registers. Assume that all numbers are in hex. Suppose that the contents of the memory location 2010 are to be added with the contents of 2012. Assume that [NNNN] represents the contents of the memory

location NNNN. Now, suppose that [2010] = 0002 and [2012] = 0005. The steps involved in accomplishing this addition can be summarized as follows:

1. Load the memory address register (MAR) with the address of the first data item to be added, that is, load 2010 into MAR.
2. Move the contents of this address to a data register, D0; that is, move first data into D0.
3. Increment the MAR by 2 to hold 2012, the address of the second data item to be added.
4. Add the contents of this memory location to the data that was moved to the data register, D0 in step 2, and store the result in the 16-bit data register, D0. The above addition program will be written using 68000 instructions. Note that the 68000 uses 24-bit addresses; 24-bit addresses such as 002000_{16} will be represented as 2000_{16} (16-bit number) in the following.

The following steps will be used to achieve this addition for the 68000:

1. Load the contents of the next 16-bit memory word into the memory address register, A1. Note that register A1 can be considered as MAR in the 68000.
2. Read the 16-bit contents of the memory location addressed by MAR into data register, D0.
3. Increment MAR by 2 to hold 2012, the address of the second data to be added.
4. Add the current contents of data register, D0 to the contents of the memory location whose address is in MAR and store the 16-bit result in D0.

The following steps for the Motorola 68000 will be used to achieve the above addition:

3279_{16}	Load the contents of the next 16-bit memory word into the memory address register, A1.
3010_{16}	Read the 16-bit contents of the memory location addressed by MAR into data register, D0.
5249_{16}	Increment MAR by 2.
$D051_{16}$	Add the current contents of data register, D0, to the contents of the memory location whose address is in MAR and store the 16-bit result in D0.

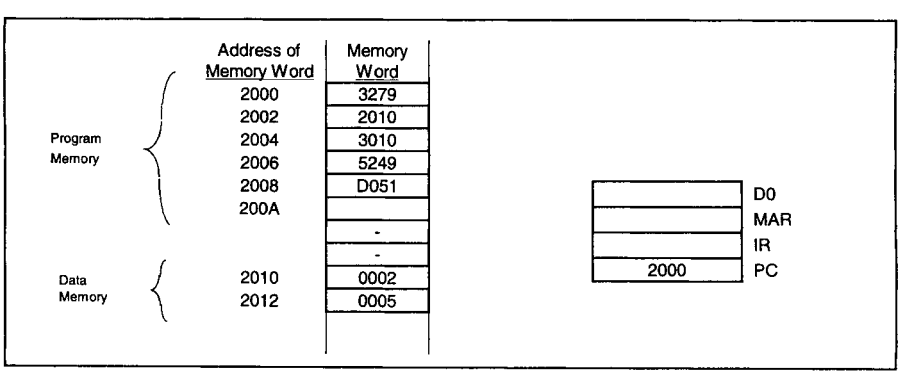


FIGURE 6.5 Microprocessor addition program with initial register and memory

The complete program in hexadecimal, starting at location 2000_{16} (arbitrarily chosen) is given in Figure 6.5. Note that each memory address stores 16 bits. Hence, memory addresses are shown in increments of 2. Assume that the microcomputer can be instructed that the starting address of the program is 2000_{16}. This means that the program counter can be initialized to contain 2000_{16}, the address of the first instruction to be executed. Note that the contents of the other three registers are not known at this point. The microprocessor loads the contents of memory location addressed by the program counter into IR. Thus, the first instruction, 3279_{16}, stored in address 2000_{16} is transferred into IR.

The program counter contents are then incremented by 2 by the microprocessor's ALU to hold 2002_{16}. The register contents that result along with the program are shown in Figure 6.6.

The binary code 3279_{16} in the IR is executed by the microprocessor. The microprocessor then takes appropriate actions. Note that the instruction, 3279_{16}, loads the contents of the next memory location addressed by the PC into the MAR. Thus, 2010_{16} is loaded into the MAR. The contents of the PC are then incremented by 2 to hold 2004_{16}. This is shown in Figure 6.7.

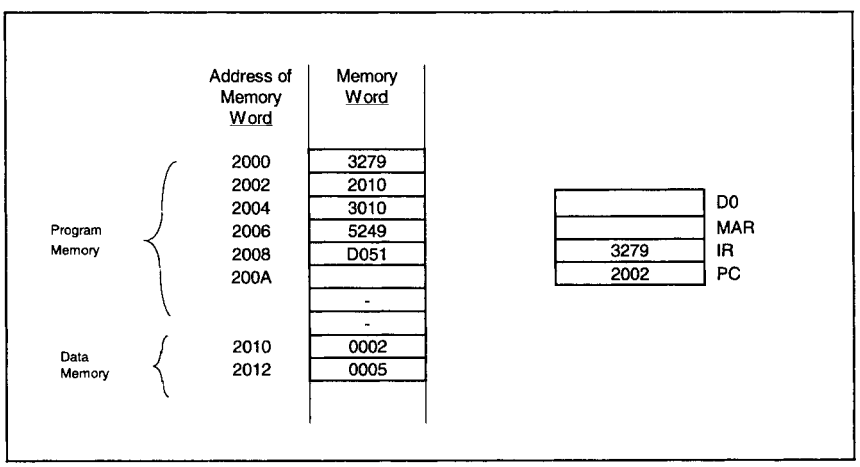


FIGURE 6.6 Microprocessor addition program (modified during execution)

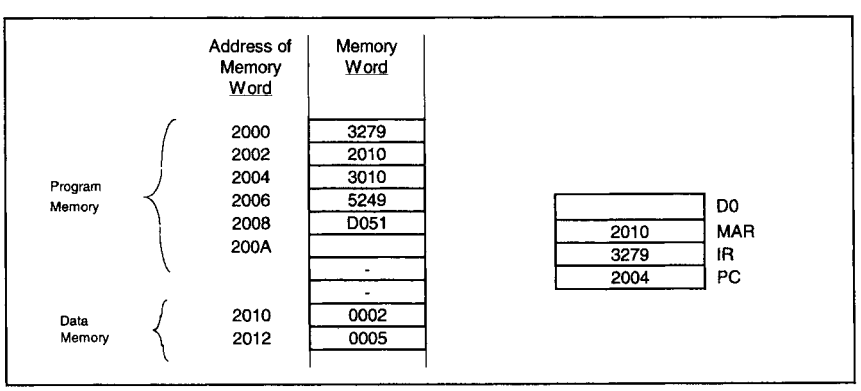


FIGURE 6.7 Microprocessor addition program (modified during execution)

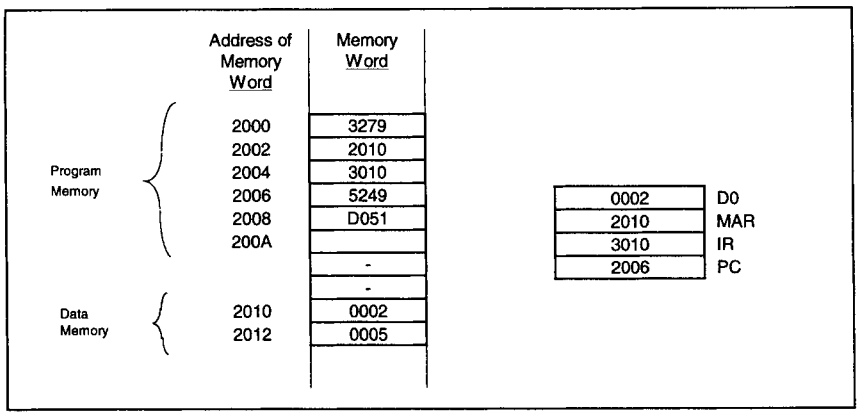


FIGURE 6.8 Microprocessor addition program (modified during execution)

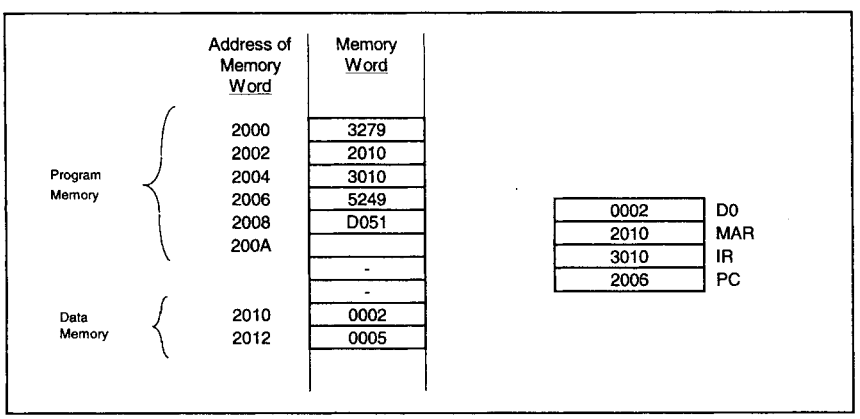


FIGURE 6.9 Microprocessor addition program (modified during execution)

Next, the microprocessor loads the contents of the memory location addressed by the PC into the IR; thus, 3010_{16} is loaded into the IR. The PC contents are then incremented by 2 to hold 2006_{16}. This is shown in Figure 6.8. In response to the instruction 3010_{16}, the contents of the memory location addressed by the MAR are loaded into the data register, D0; thus, 0002_{16} is moved to register D0. The contents of the PC are not incremented this time. This is because 0002_{16} is not immediate data. Figure 6.9 shows the details. Next the microprocessor loads 5249_{16} to IR and then increments PC to contain 2008_{16} as shown in Figure 6.10.

In response to the instruction 5249_{16} in the IR, the microprocessor increments the MAR by 2 to contain 2012_{16} as shown in Figure 6.11. Next, the instruction $D051_{16}$ in location 2008_{16} is loaded into the IR, and the PC is then incremented by 2 to hold $200A_{16}$ as shown in Figure 6.12. Finally, in response to instruction $D051_{16}$, the microprocessor adds the contents of the memory location addressed by MAR (address 2012_{16}) with the contents of register D0 and stores the result in D0. Thus, 0002_{16} is added with 0005_{16}, and the 16-bit result 0007_{16} is stored in D0 as shown in Figure 6.13. This completes the execution of the binary addition program.

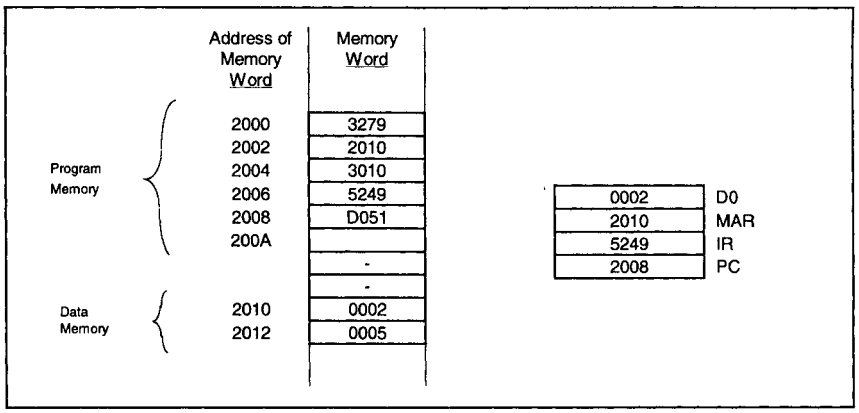


FIGURE 6.10 Microprocessor addition program (modified during execution)

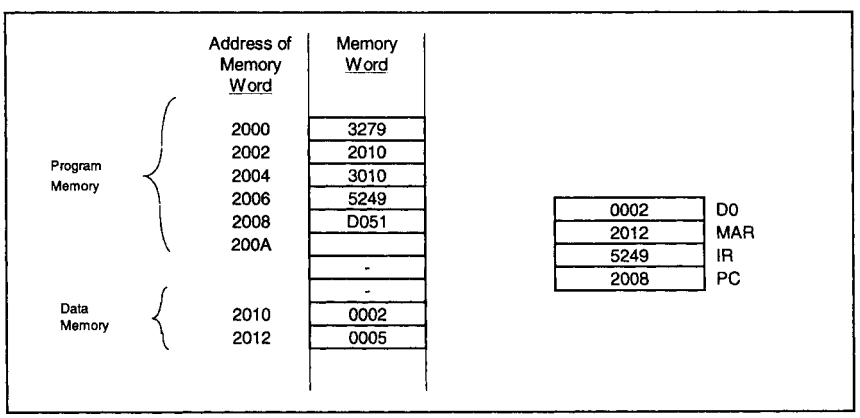


FIGURE 6.11 Microprocessor addition program (modified during execution)

Other Microprocessor Registers

- **General-Purpose Registers**

  The 16-, 32-, and 64-bit microprocessors are register oriented. They have a number of general-purpose registers for storing temporary data or for carrying out data transfers between various registers. The use of general-purpose registers speeds up the execution of a program because the microprocessor does not have to read data from external memory via the data bus if data is stored in one of its general-purpose registers. These registers are typically 16 to 32 bits. The number of general-purpose registers will vary from one microprocessor to another. Some of the typical functions performed by instructions associated with the general-purpose registers are given here. We will use [REG] to indicate the contents of the general-purpose register and [M] to indicate the contents of a memory location.

  1. Move [REG] to or from memory: [M] ← [REG] or [REG] ← [M].
  2. Move the contents of one register to another: [REG1] ← [REG2].
  3. Increment or decrement [REG] by 1: [REG] ← [REG] + 1 or [REG] ← [REG] - 1.
  4. Load 16-bit data into a register [REG] : [REG] ← 16-bit data.

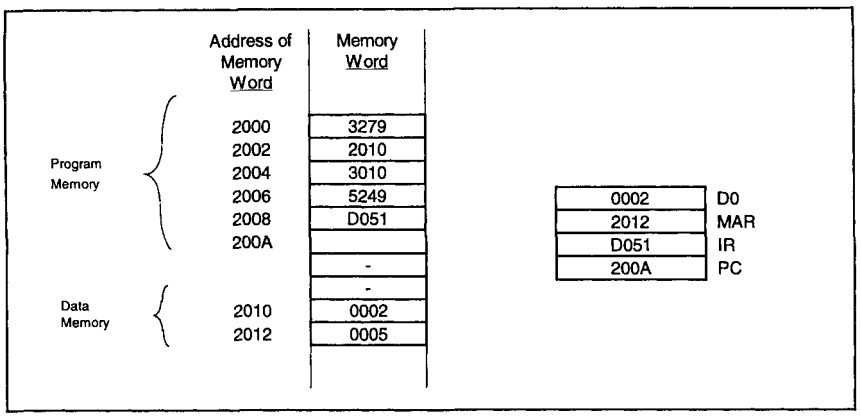


FIGURE 6.12 Microprocessor addition program (modified during execution)

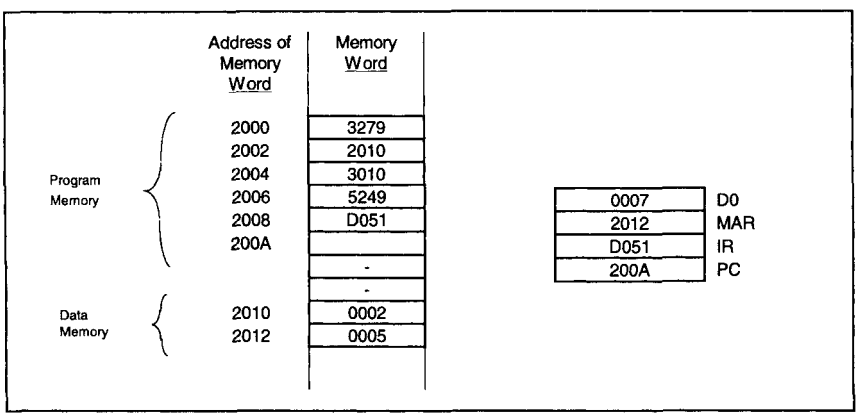


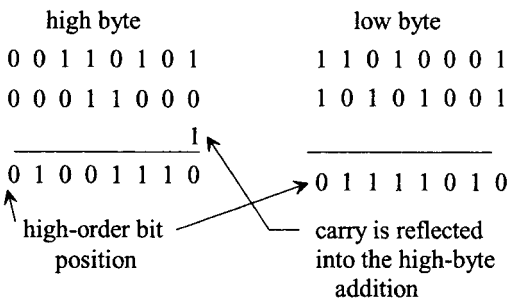
FIGURE 6.13 Microprocessor addition program (modified during execution)

- **Index Register**
  An *index register* is typically used as a counter in address modification for an instruction, or for general storage functions. The index register is particularly useful with instructions that access tables or arrays of data. In this operation the index register is used to modify the address portion of the instruction. Thus, the appropriate data in a table can be accessed. This is called "indexed addressing." This addressing mode is normally available to the programmers of microprocessors. The effective address for an instruction using the indexed addressing mode is determined by adding the address portion of the instruction to the contents of the index register. Index registers are typically 16 or 32 bits long. In a typical 16- or 32-bit microprocessor, general-purpose registers can be used as index registers.

- **Status Register**
  The *status register,* also known as the "processor status word register" or the "condition code register," contains individual bits, with each bit having special significance. The bits in the status register are called "flags." The status of a specific microprocessor operation is indicated by each flag, which is set or reset by the microprocessor's internal logic to indicate the status of certain microprocessor operations such as arithmetic and

logic operations. The status flags are also used in conditional JUMP instructions. We will describe some of the common flags in the following.

The *carry flag* is used to reflect whether or not the result generated by an arithmetic operation is greater than the microprocessor's word size. As an example, the addition of two 8-bit numbers might produce a carry. This carry is generated out of the eighth position, which results in setting the carry flag. However, the carry flag will be zero if no carry is generated from the addition. As mentioned before, in multibyte arithmetic, any carry out of the low-byte addition must be added to the high-byte addition to obtain the correct result. This can illustrated by the following example:



While performing BCD arithmetic with microprocessors, the carry out of the low nibble (4 bits) has a special significance. Because a BCD digit is represented by 4 bits, any carry out of the low 4 bits must be propagated into the high 4 bits for BCD arithmetic. This carry flag is known as the *auxiliary carry flag* and is set to 1 if the carry out of the low 4 bits is 1, otherwise it is 0.

A *zero flag* is used to show whether the result of an operation is zero. It is set to 1 if the result is zero, and it is reset to 0 if the result is nonzero. A *parity flag* is set to 1 to indicate whether the result of the last operation contains either an even number of 1's (even parity) or an odd number of 1's (odd parity), depending on the microprocessor. The type of parity flag used (even or odd) is determined by the microprocessor's internal structure and is not selectable. The sign flag (also sometimes called the negative flag) is used to indicate whether the result of the last operation is positive or negative. If the most significant bit of the last operation is 1, then this flag is set to 1 to indicate that the result is negative. This flag is reset to 0 if the most significant bit of the result is zero, that is, if the result is positive.

As mentioned before, the *overflow flag* arises from the representation of the sign flag by the most significant bit of a word in signed binary operation. The overflow flag is set to 1 if the result of an arithmetic operation is too big for the microprocessor's maximum word size, otherwise it is reset to 0. Let C_f be the final carry out of the most significant bit (sign bit) and C_p be the previous carry. It was shown in Chapter 2 that the overflow flag is the exclusive OR of the carries C_p and C_f.

$$\text{Overflow} = C_p \oplus C_f$$

- **Stack Pointer Register**

The *stack* consists of a number of RAM locations set aside for reading data from or writing data into these locations and is typically used by subroutines (a subroutine is a program that performs operations frequently needed by the main or calling program). The address of the stack is contained in a register called the "stack pointer." Two instructions, PUSH and POP, are usually available with the stack. The PUSH operation

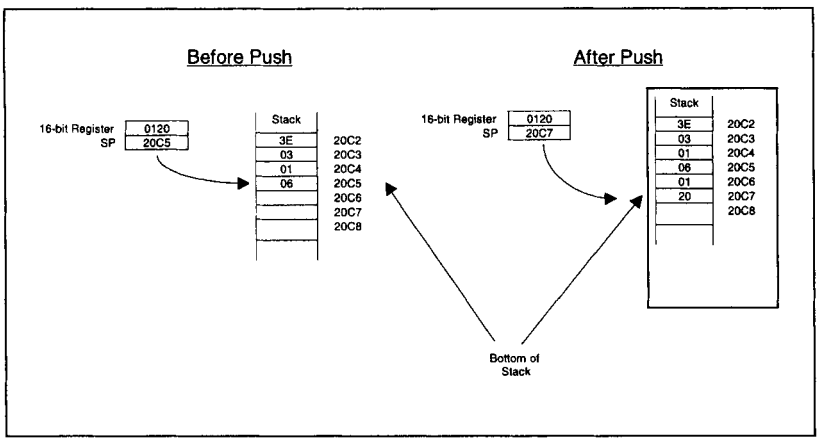


FIGURE 6.14 PUSH operation when accessing stack from bottom

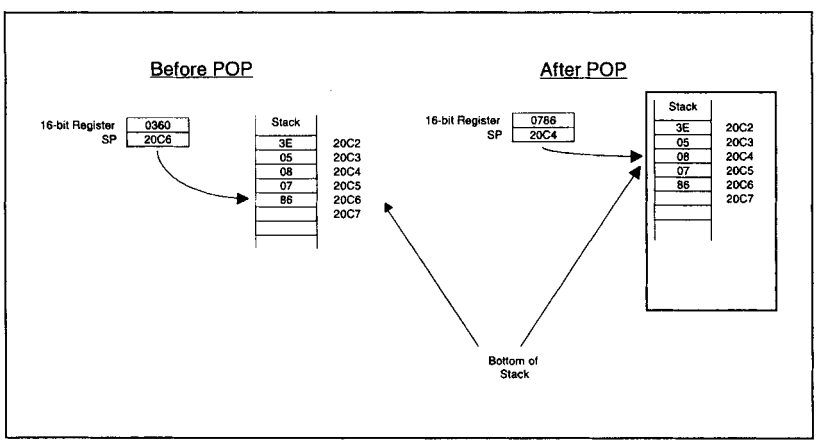


FIGURE 6.15 POP operation when accessing stack from bottom

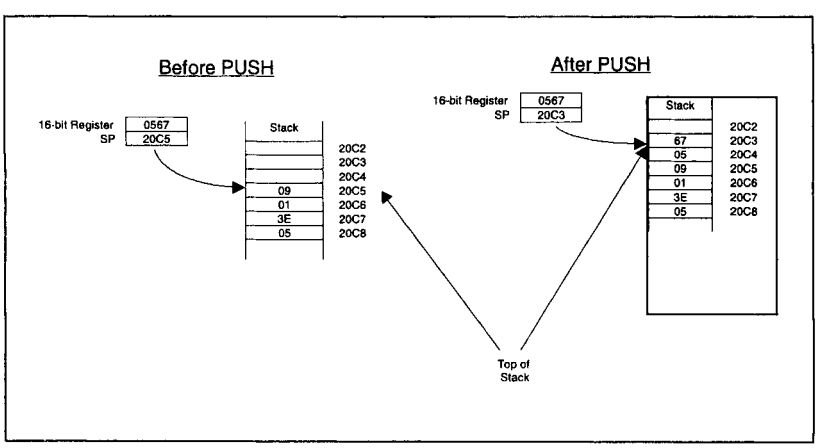


FIGURE 6.16 PUSH operation when accessing stack from top

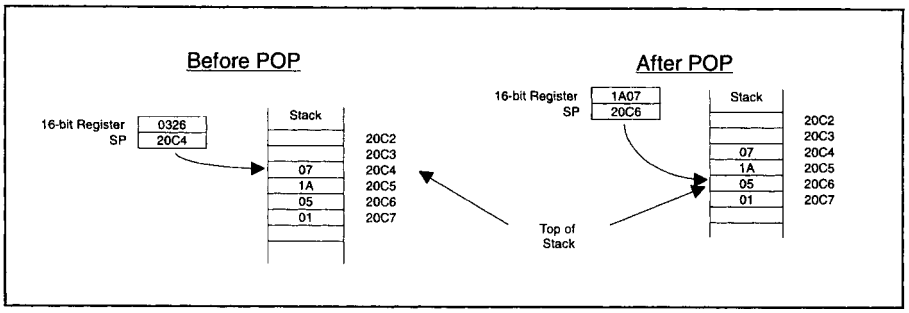


FIGURE 6.17 POP operation when accessing stack from top

is defined as writing to the top or bottom of the stack, whereas the POP operation means reading from the top or bottom of the stack. Some microprocessors access the stack from the top; the others access via the bottom. When the stack is accessed from the bottom, the stack pointer is incremented after a PUSH and decremented after a POP operation. On the other hand, when the stack is accessed from the top, the stack pointer is decremented after a PUSH and incremented after a POP. Microprocessors typically use 16- or 32-bit registers for performing the PUSH or POP operations. The incrementing or decrementing of the stack pointer depends on whether the operation is PUSH or POP and also whether the stack is accessed from the top or the bottom.

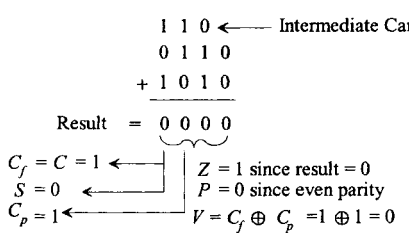
We now illustrate the stack operations in more detail. We use 16-bit registers in Figures 6.14 and 6.15. In Figure 6.14, the stack pointer is incremented by 2 (since 16-bit register) to address location 20C7 after the PUSH. Now consider the POP operation of Figure 6.15. Note that after the POP, the stack pointer is decremented by 2. [20C5] and [20C6] are assumed to be empty conceptually after the POP operation. Finally, consider the PUSH operation of Figure 6.16. The stack is accessed from the top. Note that the stack pointer is decremented by 2 after a PUSH. Next, consider the POP (Figure 6.17). [20C4] and [20C5] are assumed to be empty after the POP.
Note that the stack is a LIFO (Last In First Out) memory.

Example 6.1

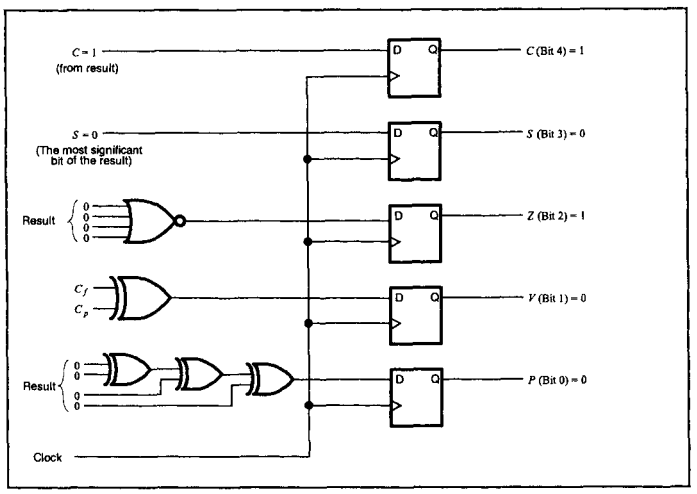
Determine the carry (C), sign (S), zero (Z), overflow (V), and parity (P) flags for the following operation: 0110_2 plus 1010_2.

Assume the parity bit = 1 for ODD parity in the result; otherwise the parity bit = 0. Also, assume that the numbers are signed. Draw a logic diagram for implementing the flags in a 5-bit register using D flip-flops; use P = bit 0, V = bit 1, Z = bit 2, S = bit 3, and C = bit 4. Note that Verilog and VHDL descriptions along with simulation results of this status register are provided in Appendices I and J respectively.

Solution



The flag register can be implemented from the 4-bit result as follows:



6.3.2 Control Unit

The main purpose of the control unit is to read and decode instructions from the program memory. To execute an instruction, the control unit steps through the appropriate blocks of the ALU based on the op-codes contained in the instruction register. The op-codes define the operations to be performed by the control unit in order to execute an instruction. The control unit interprets the contents of the instruction register and then responds to the instruction by generating a sequence of enable signals. These signals activate the appropriate ALU logic blocks to perform the required operation.

The control unit generates the *control signals*, which are output to the other microcomputer elements via the control bus. The control unit also takes appropriate actions in response to the control signals on the control bus provided by the other microcomputer elements.

The control signals vary from one microprocessor to another. For each specific microprocessor, these signals are described in detail in the manufacturer's manual. It is impossible to describe all the control signals for various manufacturers. However, we cover some of the common ones in the following discussion.

- **RESET.** This input is common to all microprocessors. When this input pin is driven to HIGH or LOW (depending on the microprocessor), the program counter is loaded with a predefined address specified by the manufacturer. For example, in the 80486, upon hardware reset, the program counter is loaded with $FFFFFFF0_{16}$. This means that the instruction stored at memory location $FFFFFFF0_{16}$ is executed first. In some other microprocessors, such as the Motorola 68000, the program counter is not loaded directly by activating the RESET input. In this case, the program counter is loaded indirectly from two locations (such as 000004 and 000006) predefined by the manufacturer. This means that these two locations contain the address of the first instruction to be executed.
- **READ/WRITE (R/$\overline{W}$).** This output line is common to all microprocessors. The status of this line tells the other microcomputer elements whether the microprocessor

is performing a READ or a WRITE operation. A HIGH signal on this line indicates a READ operation and a LOW indicates a WRITE operation. Some microprocessors have separate READ and WRITE pins.

- **$\overline{\text{READY}}$.** This is an input to the microprocessor. Slow devices (memory and I/O) use this signal to gain extra time to transfer data to or receive data from a microprocessor. The $\overline{\text{READY}}$ signal is usually an active low signal, that is, LOW means that the microprocessor is ready. Therefore, when the microprocessor selects a slow device, the device places a LOW on the $\overline{\text{READY}}$ pin. The microprocessor responds by suspending all its internal operations and enters a WAIT state. When the device is ready to send or receive data, it removes the $\overline{\text{READY}}$ signal. The microprocessor comes out of the WAIT state and performs the appropriate operation.
- **Interrupt Request (INT or IRQ).** The external I/O devices can interrupt the microprocessor via this input pin on the microprocessor chip. When this signal is activated by the external devices, the microprocessor jumps to a special program, called the "interrupt service routine." This program is normally written by the user for performing tasks that the interrupting device wants the microprocessor to do. After completing this program, the microprocessor returns to the main program it was executing when the interrupt occurred.

6.3.3 Arithmetic and Logic Unit (ALU)

The ALU performs all the data manipulations, such as arithmetic and logic operations, inside the microprocessor. The size of the ALU conforms to the word length of the microcomputer. This means that a 32-bit microprocessor will have a 32-bit ALU. Typically, the ALU performs the following functions:

1. Binary addition and logic operations
2. Finding the ones complement of data
3. Shifting or rotating the contents of a general-purpose register 1 bit to the left or right through carry

6.3.4 Functional Representations of a Simple and a Typical Microprocessor

Figure 6.18 shows the functional block diagram of a simple microprocessor. Note that the

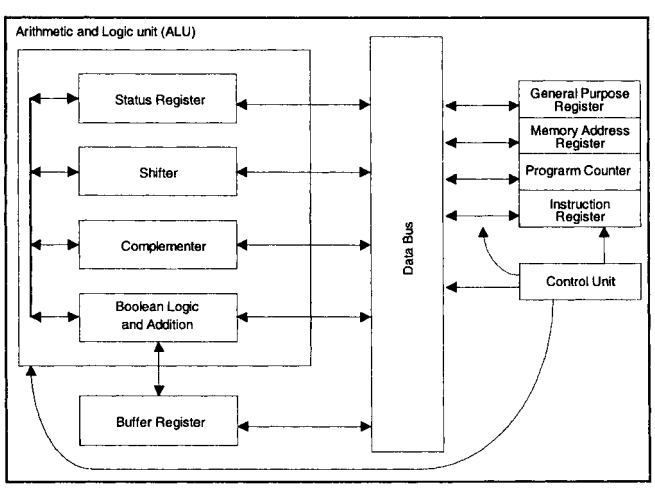


FIGURE 6.18 Functional representation of a simple microprocessor

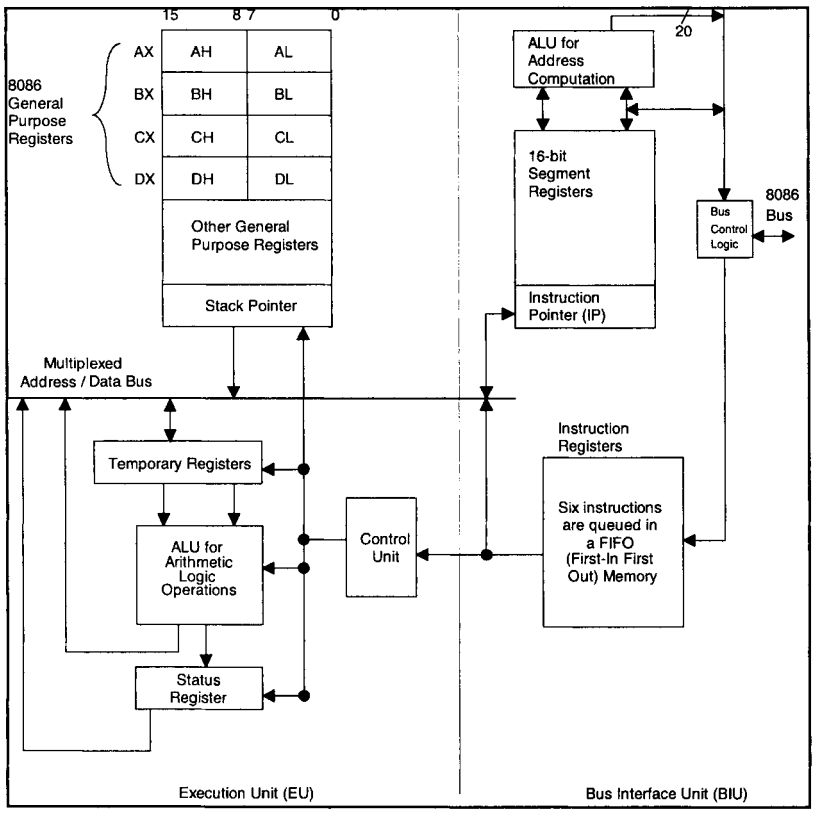


FIGURE 6.19 Simplified block diagram of the 8086

data bus shown is internal to the microprocessor chip and should not be confused with the system bus. The system bus is external to the microprocessor and is used to connect all the necessary chips to form a microcomputer. The buffer register in Figure 6.18 stores any data read from memory for further processing by the ALU. All other blocks of Figure 6.18 have been discussed earlier. Figure 6.19 shows the simplified block diagram of a realistic microprocessor, the Intel 8086.

The 8086 microprocessor is internally divided into two functional units: the bus interface unit (BIU) and the execution unit (EU). The BIU interfaces the 8086 to external memory and I/O chips. The BIU and EU function independently. The BIU reads (fetches) instructions and writes or reads data to or from memory and I/O ports. The EU executes instructions that have already been fetched by the BIU. The BIU contains segment registers, the instruction pointer (IP), the instruction queue registers, and the address generation/bus control circuitry.

The 8086 uses segmented memory. This means that the 8086's 1 MB main memory is divided into 16 segments of 64 KB each. Within a particular segment, the instruction pointer (IP) works as a program counter (PC). Both the IP and the segment registers are 16 bits wide. The 20-bit address is generated in the BIU by using the contents of a 16-bit IP and a 16-bit segment register. The ALU in the BIU is used for this purpose. Memory segmentation is useful in a time-shared system when several users share a microprocessor. Segmentation makes it easy to switch from one user program to another by changing the

contents of a segment register.

The bus control logic of the BIU generates all the bus control signals such as read and write signals for memory and I/O. The BIU's instruction register consist of a first-in–first-out (FIFO) memory in which up to six instruction bytes are preread (prefetched) from external memory ahead of time to speed up instruction execution. The control unit in the EU translates the instructions based on the contents of the instruction registers in the BIU.

The EU contains several 16-bit general-purpose registers. Some of them are AX, BX, CX, and DX. Each of these registers can be used either as an 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) or as a 16-bit register (AX, BX, CX, DX). Register BX can also be used to hold the address in a segment. The EU also contain a 16-bit status register. The ALU in the EU performs all arithmetic and logic operations. The 8086 is covered in detail in Chapter 9.

6.3.5 Microprogramming the Control Unit (A Simplified Explanation)

In this section, we discuss how the op-codes are interpreted by the microprocessor. Most microprocessors have an internal memory, called the "control memory" (ROM). This memory is used to store a number of codes, called the "microinstructions." These microinstructions are combined together to design instructions. Each instruction in the instruction register initiates execution of a set of microinstructions in the control unit to perform the operation required by the instruction. The microprocessor manufacturers define the microinstructions by programming the control memory (ROM) and thus, design the instruction set of the microprocessor. This type of programming is known as "microprogramming." Note that the control units of most 16-, 32-, and 64-bit microprocessors are microprogrammed.

For simplicity, we illustrate the concepts of microprogramming using Figure 6.18. Let us consider incrementing the contents of the register. This is basically an addition operation. The control unit will send an enable signal to execute the ALU adder logic.

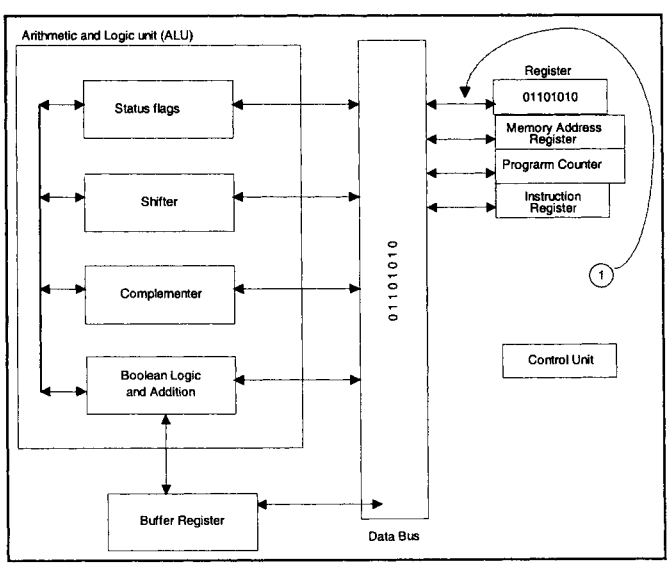


FIGURE 6.20 Transferring register contents to data bus

Incrementing the contents of a register consists of transferring the register contents to the ALU adder and then returning the result to the register. The complete incrementing process is accomplished via the five steps shown in Figures 6.20 through Figure 6.24. In all five steps, the control unit initiates execution of each microinstruction. Figure 6.20 shows the transfer of the register contents to the data bus. Figure 6.21 shows the transfer of the contents of the data bus to the adder in the ALU in order to add 1 to it. Figure 6.22 shows the activation of the adder logic. Figure 6.23 shows the transfer of the result from the adder to the data bus. Finally, Figure 6.24 shows the transfer of the data bus contents to the register.

Microprogramming is typically used by the microprocessor designer to program the logic performed by the control unit. On the other hand, assembly language programming is a popular programming language used by the microprocessor user for programming the microprocessor to perform a desired function. A microprogram is stored in the control unit. An assembly language program is stored in the main memory. The assembly language program is called a macroprogram. A macroinstruction (or simply an instruction) initiates execution of a complete microprogram.

A simplified explanation of microprogramming is provided in this section. This topic will be covered in detail in Chapter 7.

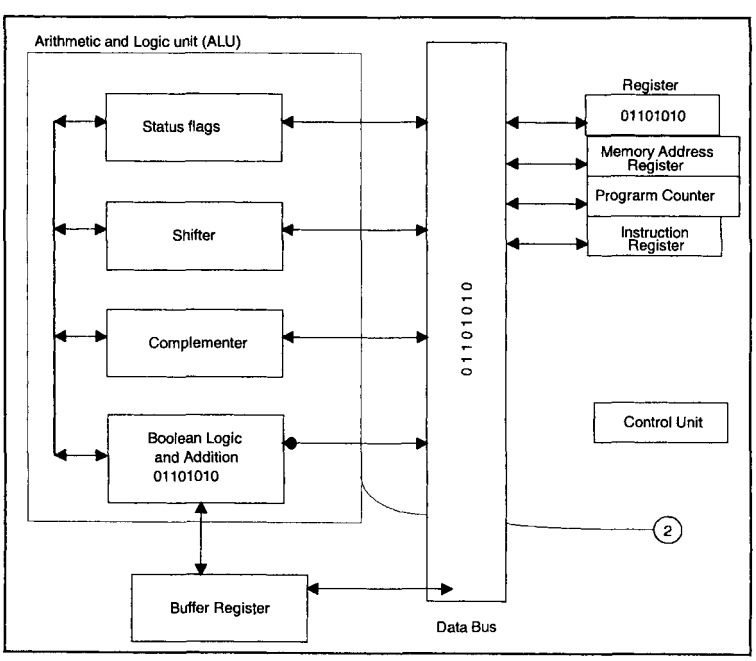


FIGURE 6.21 Transferring data bus contents to the ALU

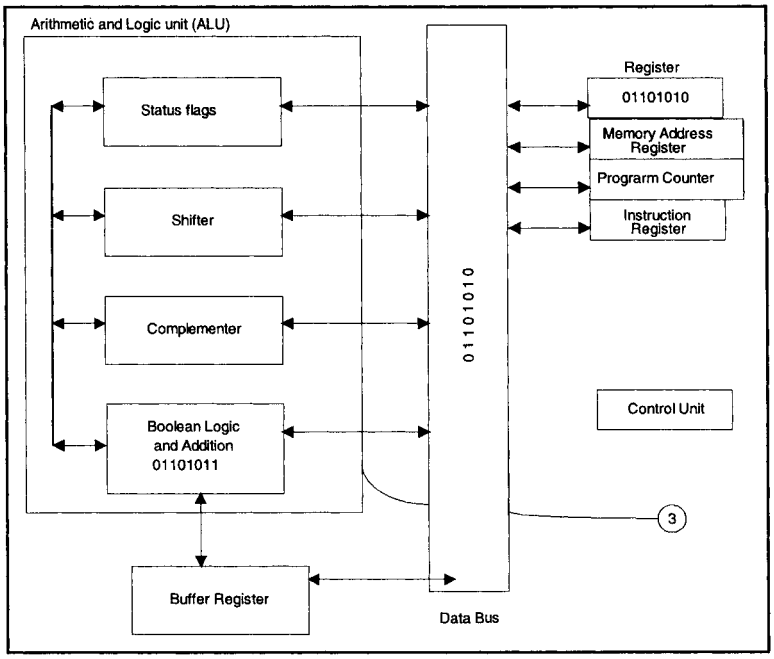


FIGURE 6.22 Activating the ALU logic

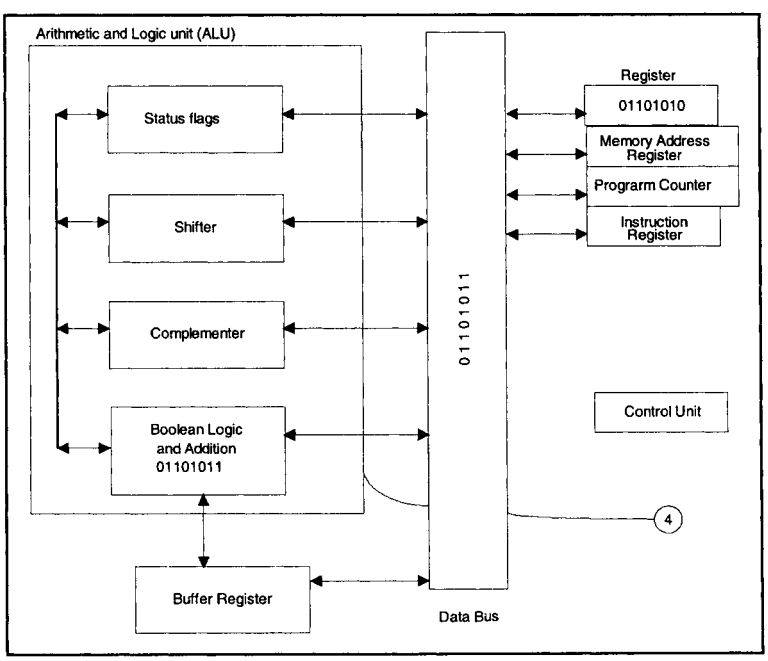


FIGURE 6.23 Transferring the ALU result to the data bus

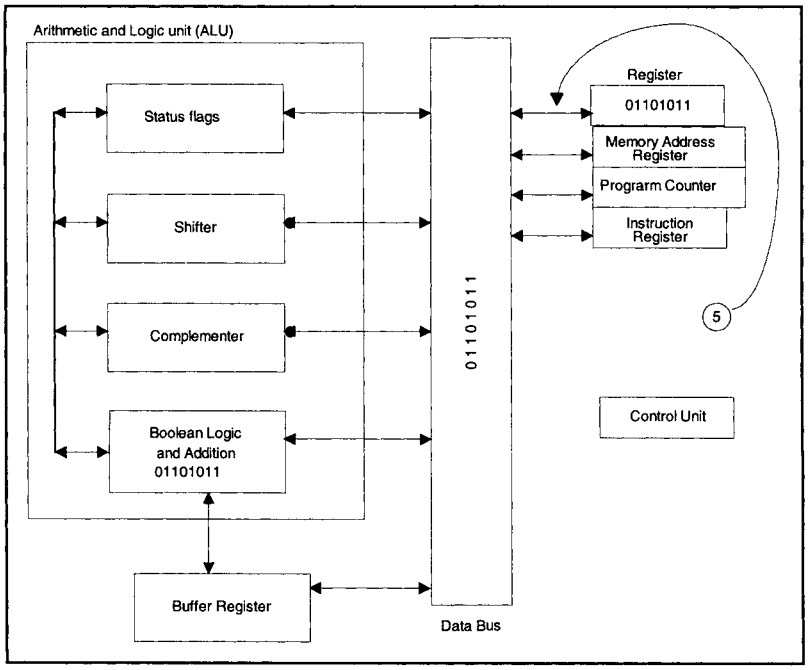


FIGURE 6.24 Transferring the data bus

6.4 The Memory

The main or external memory (or simply the memory) stores both instructions and data. For 8-bit microprocessors, the memory is divided into a number of 8-bit units called "memory words." An 8-bit unit of data is termed a "byte." Therefore, for an 8-bit microprocessor, "memory word" and "memory byte" mean the same thing. For 16-bit microprocessors, a word contains two bytes (16 bits). A memory word is identified in the memory by an address. For example, the 8086 microprocessor uses 20-bit addresses for accessing

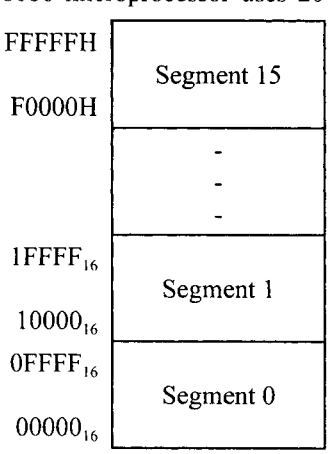


FIGURE 6.25 The main memory of the 8086

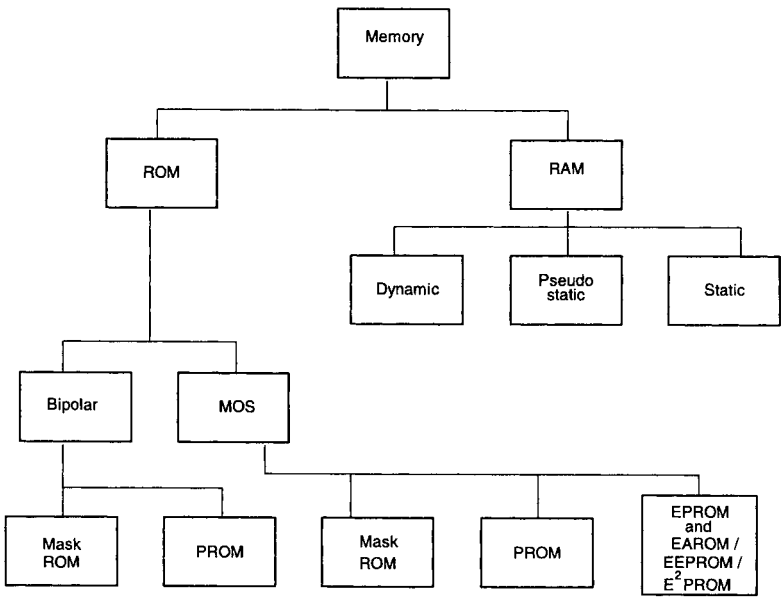


FIGURE 6.26 Summary of available semiconductor memories for microprocessor systems

memory words. This provides a maximum of 2^{20} = 1 MB of memory addresses, ranging from 00000_{16} to $FFFFF_{16}$ in hexadecimal.

As mentioned before, an important characteristic of a memory is whether it is volatile or nonvolatile. The contents of a volatile memory are lost if the power is turned off. On the other hand, a nonvolatile memory retains its contents after power is switched off. Typical examples of nonvolatile memory are ROM and magnetic memory (floppy disk). A RAM is a volatile memory unless backed up by battery.

As mentioned earlier, some microprocessors such as the Intel 8086 divide the memory into segments. For example, the 8086 divides the 1 MB main memory into 16 segments (0 through 15). Each segment contains 64 KB of memory and is addressed by 16 bits. Figure 6.25 shows a typical main memory layout of the 8086. In the figure, the high four bits of an address specify the segment number. As an example, consider address 10005_{16} of segment 1. The high four bits, 0001, of this address define the location is in segment 1 and the low 16 bits, 0005_{16}, specify the particular address in segment 1. The 68000, on the other hand, uses linear or nonsegmented memory. For example, the 68000 uses 24 address pins to directly address 2^{24} = 16 MB of memory with addresses from 000000_{16} to $FFFFFF_{16}$. As mentioned before, memories can be categorized into two main types: read-only memory (ROM) and random-access memory (RAM). As shown in Figure 6.26, ROMs and RAMs are then divided into a number of subcategories, which are discussed next.

6.4.1 Random-Access Memory (RAM)

There are three types of RAM: dynamic RAM, pseudo-static RAM , and static RAM. Dynamic RAM stores data in capacitors, that is, it can hold data for a few milliseconds. Hence, dynamic RAMs are refreshed typically by using external refresh circuitry. Pseudo-static RAMs are dynamic RAMs with internal refresh. Finally, static RAM stores data

in flip-flops. Therefore, this memory does not need to be refreshed. RAMs are volatile unless backed up by battery. Dynamic RAMs (DRAMs) are used in applications requiring large memory. DRAMs have higher densities than Static RAMs (SRAMs). Typical examples of DRAMs are 4464 (64K × 4-bit), 44256 (256K × 4-bit), and 41000 (1M × 1-bit). DRAMs are inexpensive, occupy less space , and dissipate less power compared to SRAMs. Two enhanced versions of DRAM are EDO DRAM (Extended Data Output DRAM) and SDRAM (Synchronous DRAM). The EDO DRAM provides fast access by allowing the DRAM controller to output the next address at the same time the current data is being read. An SDRAM contains multiple DRAMs (typically 4) internally. SDRAMs utilize the multiplexed addressing of conventional DRAMs . That is, SDRAMs provide row and column addresses in two steps like DRAMs. However, the control signals and address inputs are sampled by the SDRAM at the leading edge of a common clock signal (133 MHz maximum). SDRAMs provide higher densities by further reducing the need for support circuitry and faster speeds than conventional DRAMs. The SDRAM has become popular with PC (Personal Computer) memory.

6.4.2 Read-Only Memory (ROM)

ROMs can only be read. This memory is nonvolatile. From the technology point of view, ROMs are divided into two main types, bipolar and MOS. As can be expected, bipolar ROMs are faster than MOS ROMs. Each type is further divided into two common types, mask ROM and programmable ROM. MOS ROMs contain one more type, erasable PROM (EPROM such as Intel 2732 and EAROM or EEPROM or E^2PROM such as Intel 2864). Mask ROMs are programmed by a masking operation performed on the chip during the manufacturing process. The contents of mask ROMs are permanent and cannot be changed by the user. On the other hand, the programmable ROM (PROM) can be programmed by the user by means of proper equipment. However, once this type of memory is programmed, its contents cannot be changed. Erasable PROMs (EPROMs and EAROMs) can be programmed, and their contents can also be altered by using special equipment, called the PROM programmer. When designing a microcomputer for a particular application, the permanent programs are stored in ROMs. Control memories are ROMs. PROMs can be programmed by the user. PROM chips are normally designed using transistors and fuses.

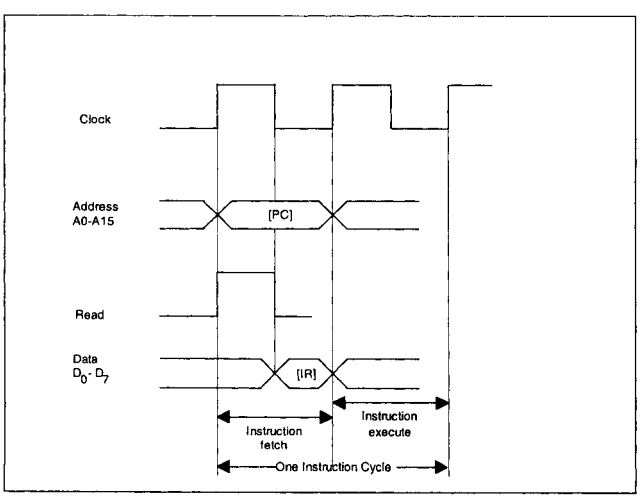


FIGURE 6.27 Typical Instruction Fetch Timing Diagram for an 8-bit Microprocessor

These transistors can be selected by addressing via the pins on the chip. In order to program this memory, the selected fuses are "blown" or "burned" by applying a voltage on the appropriate pins of the chip. This causes the memory to be permanently programmed.

Erasable PROMs (EPROMs) can be reprogrammed and erased. The chip must be removed from the microcomputer system for programming. This memory is erased by exposing the chip via a lid or window on the chip to ultraviolet light. Typical erase times vary between 10 and 30 min. The EPROM can be programmed by inserting the chip into a socket of the PROM programmer and providing proper addresses and voltage pulses at the appropriate pins of the chip. Electrically alterable ROMs (EAROMs) can be programmed without removing the memory from the ROM's sockets. These memories are also called read mostly memories (RMMs), because they have much slower write times than read times. Therefore, these memories are usually suited for operations when mostly reading rather that writing will be performed. Another type of memory called "Flash memory" (nonvolatile) invented in the mid 1980s by Toshiba is designed using a combination of EPROM and E^2PROM technologies. Flash memory can be reprogrammed electrically while being embedded on the board. One can change multiple bytes at a time. An example of Flash memory is the Intel 28F020 (256K x 8). Flash memory is typically used in cellular phones and digital cameras.

6.4.3 READ and WRITE Operations

To execute an instruction, the microprocessor reads or fetches the op-code via the data bus from a memory location in the ROM/RAM external to the microprocessor. It then places the op-code (instruction) in the instruction register. Finally, the microprocessor executes the instruction. Therefore, the execution of an instruction consists of two portions, instruction fetch and instruction execution. We will consider the instruction fetch, memory READ and memory WRITE timing diagrams in the following using a single clock signal. Figure 6.27 shows a typical instruction fetch timing diagram.

In Figure 6.27, to fetch an instruction, when the clock signal goes to HIGH, the microprocessor places the contents of the program counter on the address bus via the address pins A_0–A_{15} on the chip. Note that since each one of these lines A_0–A_{15} can be either HIGH or LOW, both transitions are shown for the address in Figure 6.27. The instruction fetch is basically a memory READ operation. Therefore, the microprocessor raises the signal

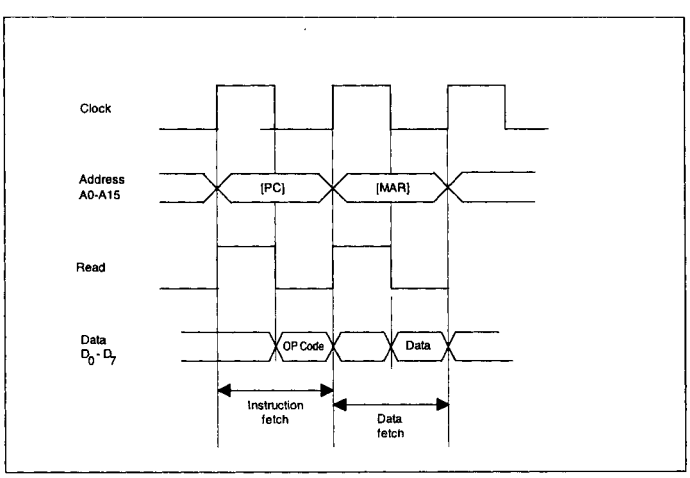


FIGURE 6.28 Typical Memory READ Timing Diagram

on the READ pin to HIGH. As soon as the clock goes to LOW, the logic external to the microprocessor gets the contents of the memory location addressed by A_0–A_{15} and places them on the data bus D_0–D_7. The microprocessor then takes the data and stores it in the instruction register so that it gets interpreted as an instruction. This is called "instruction fetch." The microprocessor performs this sequence of operations for every instruction.

We now describe the READ and WRITE timing diagrams. A typical READ timing diagram is shown in Figure 6.28. Memory READ is basically loading the contents of a memory location of the main ROM/RAM into an internal register of the microprocessor. The address of the location is provided by the contents of the memory address register (MAR). Let us now explain the READ timing diagram of Figure 6.28 as follows:

1. The microprocessor performs the instruction fetch cycle as before to READ the op-code.
2. The microprocessor interprets the op-code as a memory READ operation.
3. When the clock pin signal goes to HIGH, the microprocessor places the contents of the memory address register on the address pins A_0–A_{15} of the chip.
4. At the same time, the microprocessor raises the READ pin signal to HIGH.
5. The logic external to the microprocessor gets the contents of the location in the main ROM/RAM addressed by the memory address register and places them on the data bus.
6. Finally, the microprocessor gets this data from the data bus via its pins D_0 – D_7 and stores it in an internal register.

Memory WRITE is basically storing the contents of an internal register of the microprocessor into a memory location of the main RAM. The contents of the memory address register provide the address of the location where data is to be stored. Figure 6.29 shows a typical WRITE timing diagram. It can be explained in the following way:

1. The microprocessor fetches the instruction code as before.
2. The microprocessor interprets the instruction code as a memory WRITE instruction and then proceeds to perform the DATA STORE cycle.
3. When the clock pin signal goes to HIGH, the microprocessor places the contents of the

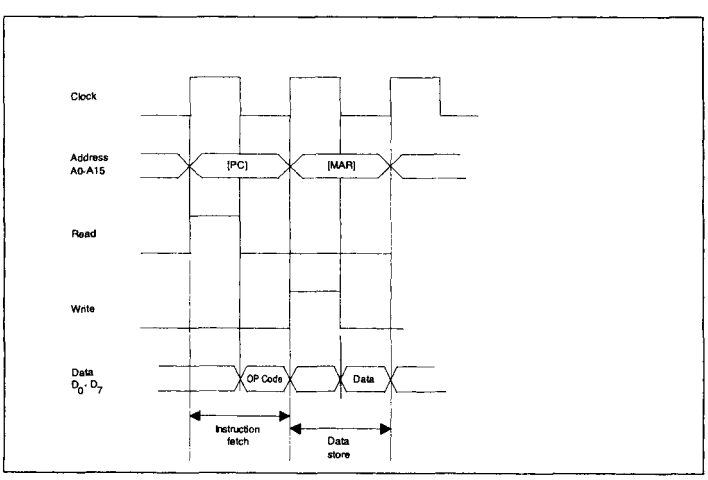


FIGURE 6.29 Typical Memory WRITE Timing Diagram

memory address register on the address pins A_0–A_{15} of the chip.
4. At the same time, the microprocessor raises the WRITE pin signal to HIGH.
5. The microprocessor places data to be stored from the contents of an internal register onto the data pins D_0–D_7.
6. The logic external to the microprocessor stores the data from the register into a RAM location addressed by the memory address register.

6.4.4 Memory Organization

Microcomputer memory typically consists of ROMs / EPROMs, and RAMs. Because RAMs can be both read from and written into, the logic required to implement RAMs is more complex than that for ROMs / EPROMs. A microcomputer system designer is normally interested in how the microcomputer memory is organized or, in other words, how to connect the ROMS /EPROMs and RAMs and then determine the memory map of the microcomputer. That is, the designer would be interested in finding out what memory locations are assigned to the ROMs / EPROMs and RAMs. The designer can then implement the permanent programs in ROMs / EPROMs and the temporary programs in RAMs. Note that RAMs are needed when subroutines and interrupts requiring stack are desired in an application.

As mentioned before, DRAMs (Dynamic RAMs) use MOS capacitors to store information and need to be refreshed. DRAMs are inexpensive compared to SRAMs, provide larger bit densities and consume less power. DRAMs are typically used when memory requirements are 16k words or larger. DRAM is addressed via row and column addressing. For example, one megabit DRAM requiring 20 address bits is addressed using 10 address lines and two control lines, $\overline{RAS}$ (Row Address Strobe) and $\overline{CAS}$ (Column Address Strobe). To provide a 20-bit address into the DRAM, a LOW is applied to $\overline{RAS}$ and 10 bits of the address are latched. The other 10 bits of the address are applied next and $\overline{CAS}$ is then held LOW.

The addressing capability of the DRAM can be increased by a factor of 4 by adding one more bit to the address line. This is because one additional address bit results into one additional row bit and one additional column bit. This is why DRAMs can be expanded to larger memory very rapidly with inclusion of additional address bits. External logic is required to generate the $\overline{RAS}$ and $\overline{CAS}$ signals, and to output the current address bits to the DRAM.

DRAM controller chips take care of refreshing and timing requirements needed by the DRAMs. DRAMs typically require 4 millisecond refresh time. The DRAM controller performs its task independent of the microprocessor. The DRAM controller sends a wait signal to the microprocessor if the microprocessor tries to access memory during a refresh cycle.

Because of large memory, the address lines should be buffered using 74LS244 or 74HC244 (Unidirectional buffer), and data lines should be buffered using 74LS245 or 74HC245 (Bidirectional buffer) to increase the drive capability. Also, typical multiplexers such as 74LS157 or 74HC157 can be used to multiplex the microprocessors address lines into separate row and column addresses.

6.5 Input/Output

Input/Output (I/O) operation is typically defined as the transfer of information between the microcomputer system and an external device. There are typically three main ways of

transferring data between the microcomputer system and the external devices. These are programmed I/O, interrupt I/O, and direct memory access. We now define them.

- **Programmed I/O.** Using this technique, the microprocessor executes a program to perform all data transfers between the microcomputer system and the external devices. The main characteristic of this type of I/O technique is that the external device carries out the functions as dictated by the program inside the microcomputer memory. In other words, the microprocessor completely controls all the transfers.
- **Interrupt I/O.** In this technique, an external device or an exceptional condition such as overflow can force the microcomputer system to stop executing the current program temporarily so that it can execute another program, known as the "interrupt service routine." This routine satisfies the needs of the external device or the exceptional condition. After having completed this program, the microprocessor returns to the program that it was executing before the interrupt.
- **Direct Memory Access (DMA).** This is a type of I/O technique in which data can be transferred between the microcomputer memory and external devices without any microprocessor (CPU) involvement. Direct memory access is typically used to transfer blocks of data between the microcomputer's main memory and an external device such as hard disk. An interface chip called the DMA controller chip is used with the microprocessor for transferring data via direct memory access.

6.6 Microcomputer Programming Concepts

This section includes the fundamental concepts of microcomputer programming. Typical programming characteristics such as programming languages, microprocessor instruction sets, addressing modes, and instruction formats are discussed.

6.6.1 Microcomputer Programming Languages

Microcomputers are typically programmed using semi-English-language statements (assembly language). In addition to assembly languages, microcomputers use a more understandable human-oriented language called the "high-level language." No matter what type of language is used to write the programs, the microcomputers only understand binary numbers. Therefore, the programs must eventually be translated into their appropriate binary forms. The main ways of accomplishing this are discussed later.

Microcomputer programming languages can typically be divided into three main types:

1. Machine language
2. Assembly language
3. High-level language

A machine language program consists of either binary or hexadecimal op-codes. Programming a microcomputer with either one is relatively difficult, because one must deal only with numbers. The architecture and microprograms of a microprocessor determine

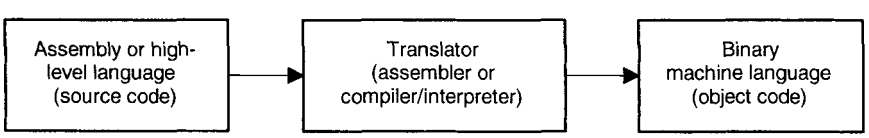


FIGURE 6.30 Translating assembly or a high-level language into binary machine language

all its instructions. These instructions are called the microprocessor's "instruction set." Programs in assembly and high-level languages are represented by instructions that use English- language-type statements. The programmer finds it relatively more convenient to write the programs in assembly or a high-level language than in machine language. However, a translator must be used to convert the assembly or high-level programs into binary machine language so that the microprocessor can execute the programs. This is shown in Figure 6.30.

An assembler translates a program written in assembly language into a machine language program. A compiler or interpreter, on the other hand, converts a high-level language program such as C or C++ into a machine language program. Assembly or high-level language programs are called "source codes." Machine language programs are known as "object codes." A translator converts source codes to object codes. Next, we discuss the three main types of programming language in more detail.

6.6.2 Machine Language

A microprocessor has a unique set of machine language instructions defined by its manufacturer. No two microprocessors by two different manufacturers have the same machine language instruction set. For example, the Intel 8086 microprocessor uses the code $01D8_{16}$ for its addition instruction whereas the Motorola 68000 uses the code $D282_{16}$. Therefore, a machine language program for one microcomputer will not usually run on another microcomputer of a different manufacturer.

At the most elementary level, a microprocessor program can be written using its instruction set in binary machine language. As an example, a program written for adding two numbers using the Intel 8086 machine language is

```
1011 1000 0000 0001 0000 0000
1011 1011 0000 0010 0000 0000
0000 0001 1101 1000
1111 0100
```

Obviously, the program is very difficult to understand, unless the programmer remembers all the 8086 codes, which is impractical. Because one finds it very inconvenient to work with 1's and 0's, it is almost impossible to write an error-free program at the first try. Also, it is very tiring for the programmer to enter a machine language program written in binary into the microcomputer's RAM. For example, the programmer needs a number of binary switches to enter the binary program. This is definitely subject to errors.

To increase the programmer's efficiency in writing a machine language program, hexadecimal numbers rather than binary numbers are used. The following is the same addition program in hexadecimal, using the Intel 8086 instruction set:

```
B80100
BB0200
01D8
F4
```

It is easier to detect an error in a hexadecimal program, because each byte contains only two hexadecimal digits. One would enter a hexadecimal program using a hexadecimal

keyboard. A keyboard monitor program in ROM, usually offered by the manufacturer, provides interfacing of the hexadecimal keyboard to the microcomputer. This program converts each key actuation into binary machine language in order for the microprocessor to understand the program. However, programming in hexadecimal is not normally used.

6.6.3 Assembly Language

The next programming level is to use the assembly language. Each line in an assembly language program includes four fields:

1. Label field
2. Instruction, mnemonic, or op-code field
3. Operand field
4. Comment field

As an example, a typical program for adding two 16-bit numbers written in 8086 assembly language is

Label	Mnemonic	Operand	Comment
START	MOV	AX, 1	move 1 into AX
	MOV	BX, 2	move 2 into BX
	ADD	AX, BX	add the contents of AX with BX
	JMP	START	jump to the beginning of the program

Obviously, programming in assembly language is more convenient than programming in machine language, because each mnemonic gives an idea of the type of operation it is supposed to perform. Therefore, with assembly language, the programmer does not have to find the numerical op-codes from a table of the instruction set, and programming efficiency is significantly improved.

The assembly language program is translated into binary via a program called an "assembler." The assembler program reads each assembly instruction of a program as ASCII characters and translates them into the respective binary op-codes. As an example, consider the HLT instruction for the 8086. Its binary op-code is 1111 0100. An assembler would convert HLT into 111 0100 as shown in Figure 6.31.

An advantage of the assembler is address computation. Most programs use addresses within the program as data storage or as targets for jumps or calls. When programming in machine language, these addresses must be calculated by hand. The assembler solves this problem by allowing the programmer to assign a symbol to an address. The programmer may then reference that address elsewhere by using the symbol. The assembler computes the actual address for the programmer and fills it in automatically. One can obtain hands-

Assembly Code	Binary form of ASCII Codes as Seen by Assembler	Binary OP Code Created by Assembler
H	0100 1000	
L	0100 1100	1111 0100
T	0101 0100	

FIGURE 6.31 Conversion of HLT into its binary op-code

on experience with a typical assembler for a microprocessor by downloading it from the Internet.

Most assemblers use two passes to assemble a program. This means that they read the input program text twice. The first pass is used to compute the addresses of all labels in the program. In order to find the address of a label, it is necessary to know the total length of all the binary code preceding that label. Unfortunately, however, that address may be needed in that preceding code. Therefore, the first pass computes the addresses of all labels and stores them for the next pass, which generates the actual binary code. Various types of assemblers are available today. We define some of them in the following paragraphs.

- **One-Pass Assembler.** This assembler goes through the assembly language program once and translates it into a machine language program. This assembler has the problem of defining forward references. This means that a JUMP instruction using an address that appears later in the program must be defined by the programmer after the program is assembled.
- **Two-Pass Assembler.** This assembler scans the assembly language program twice. In the first pass, this assembler creates a symbol table. A symbol table consists of labels with addresses assigned to them. This way labels can be used for JUMP statements and no address calculation has to be done by the user. On the second pass, the assembler translates the assembly language program into the machine code. The two-pass assembler is more desirable and much easier to use.
- **Macroassembler.** This type of assembler translates a program written in macrolanguage into the machine language. This assembler lets the programmer define all instruction sequences using macros. Note that, by using macros, the programmer can assign a name to an instruction sequence that appears repeatedly in a program. The programmer can thus avoid writing an instruction sequence that is required many times in a program by using macros. The macroassembler replaces a macroname with the appropriate instruction sequence each time it encounters a macroname.

  It is interesting to see the difference between a subroutine and a macroprogram. A specific subroutine occurs once in a program. A subroutine is executed by CALLing it from a main program. The program execution jumps out of the main program and then executes the subroutine. At the end of the subroutine, a RET instruction is used to resume program execution following the CALL SUBROUTINE instruction in the main program. A macro, on the other hand, does not cause the program execution to branch out of the main program. Each time a macro occurs, it is replaced with the appropriate instruction sequence in the main program. Typical advantages of using macros are shorter source programs and better program documentation. A disadvantage is that effects on registers and flags may not be obvious.

  Conditional macroassembly is very useful in determining whether or not an instruction sequence is to be included in the assembly depending on a condition that is true or false. If two different programs are to be executed repeatedly based on a condition that can be either true or false, it is convenient to use conditional macros. Based on each condition, a particular program is assembled. Each condition and the appropriate program are typically included within IF and ENDIF pseudo-instructions.
- **Cross Assembler.** This type of assembler is typically resident in a processor and assembles programs for another for which it is written. The cross assembler program is written in a high-level language so that it can run on different types of processors that understand the same high-level language.
- **Resident Assembler.** This type of assembler assembles programs for a processor

in which it is resident. The resident assembler may slow down the operation of the processor on which it runs.

- **Meta-assembler.** This type of assembler can assemble programs for many different types of processors. The programmer usually defines the particular processor being used.

As mentioned before, each line of an assembly language program consists of four fields: label, mnemonic or op-code, operand, and comment. The assembler ignores the comment field but translates the other fields. The label field must start with an uppercase alphabetic character. The assembler must know where one field starts and another ends. Most assemblers allow the programmer to use a special symbol or delimiter to indicate the beginning or end of each field. Typical delimiters used are spaces, commas, semicolons, and colons:

- Spaces are used between fields.
- Commas (,) are used between addresses in an operand field.
- A semicolon (;) is used before a comment.
- A colon (:) or no delimiter is used after a label.

To handle numbers, most assemblers consider all numbers as decimal numbers unless specified. Most assemblers will also allow binary, octal, or hexadecimal numbers. The user must define the type of number system used in some way. This is usually done by using a letter following the number. Typical letters used are

- `B` for binary
- `Q` for octal
- `H` for hexadecimal

Assemblers generally require hexadecimal numbers to start with a digit. A 0 is typically used if the first digit of the hexadecimal number is a letter. This is done to distinguish between numbers and labels. For example, most assemblers will require the number A5H to be represented as 0A5H.

Assemblers use pseudo-instructions or directives to make the formatting of the edited text easier. These pseudo-instructions are not directly translated into machine language instructions. They equate labels to addresses, assign the program to certain areas of memory, or insert titles, page numbers, and so on. To use the assembler directives or pseudo-instructions, the programmer puts them in the op-code field, and, if the pseudo-instructions require an address or data, the programmer places them in the label or data field. Typical pseudo-instructions are ORIGIN (`ORG`), EQUATE (`EQU`), DEFINE BYTE (`DB`), and DEFINE WORD (`DW`).

ORIGIN (ORG)

The pseudo-instruction **ORG** lets the programmer place the programs anywhere in memory. Internally, the assembler maintains a program-counter-type register called the "address counter." This counter maintains the address of the next instruction or data to be processed.

An `ORG` pseudo-instruction is similar in concept to the `JUMP` instruction. Recall that the `JUMP` instruction causes the processor to place a new address in the program counter. Similarly, the `ORG` pseudo-instruction causes the assembler to place a new value in the address counter.

Typical `ORG` statements are

```
ORG 7000H
CLC
```

The 8086 assembler will generate the following code for these statements:

```
7000 F8
```

Most assemblers assign a value of zero to the starting address of a program if the programmer does not define this by means of an ORG.

Equate (EQU)

The pseudo-instruction **EQU** assigns a value in its operand field to an address in its label field. This allows the user to assign a numeric value to a symbolic name. The user can then use the symbolic name in the program instead of its numeric value. This reduces errors.

A typical example of EQU is `START EQU 0200H`, which assigns the value 0200 in hexadecimal to the label START. Another example is

```
PORTA       EQU     40H
            MOV     AL, 0FFH
            OUT     PORTA, AL
```

In this example, the EQU gives PORTA the value 40 hex, and FF hex is the data to be written into register AL by `MOV AL,0FFH`. `OUT PORTA,AL` then outputs this data FF hex to port 40, which has already been equated to PORTA before.

Note that, if a label in the operand field is equated to another label in the label field, then the label in the operand field must be previously defined. For example, the EQU statement

```
BEGIN   EQU     START
```

will generate an error unless START is defined previously with a numeric value.

Define Byte (DB)

The pseudo-instruction **DB** is usually used to set a memory location to certain byte value. For example,

```
START   DB      45H
```

will store the data value 45 hex to the address START.

With some assemblers, the DB pseudo-instruction can be used to generate a table of data as follows:

```
                ORG     7000H
TABLE   DB      20H,30H,40H,50H
```

In this case, 20 hex is the first data of the memory location 7000; 30 hex, 40 hex, and 50 hex occupy the next three memory locations. Therefore, the data in memory will look like this:

```
7000    20
7001    30
7002    40
7003    50
```

Note that some assemblers use DC.B instead of DB. DC stands for Define Constant.

Define Word (DW)

The pseudo-instruction DW is typically used to assign a 16-bit value to two memory locations. For example,

```
                ORG     7000H
START   DW      4AC2H
```

will assign C2 to location 7000 and 4A to location 7001. It is assumed that the assembler will assign the low byte first (C2) and then the high byte (4A).

With some assemblers, the DW pseudo-instruction can be used to generate a table of 16-bit data as follows:

```
                ORG     8000H
POINTER         DW      5000H,6000H,7000H
```

In this case, the three 16-bit values 5000H, 6000H, and 7000H are assigned to memory locations starting at the address 8000H. That is, the array would look like this:

```
8000        00
8001        50
8002        00
8003        60
8004        00
8005        70
```

Note that some assemblers use DC.W instead of DW.

Assemblers also use a number of housekeeping pseudo-instructions. Typical housekeeping pseudo-instructions are `TITLE`, `PAGE`, `END`, and `LIST`. The following are the housekeeping pseudo-instructions that control the assembler operation and its program listing.

`TITLE` prints the specified heading at the top of each page of the program listing. For example,

```
TITLE "Square Root Algorithm"
```

will print the name "Square Root Algorithm" on top of each page.

`PAGE` skips to the next line.

`END` indicates the end of the assembly language source program.

`LIST` directs the assembler to print the assembler source program.

In the following, assembly language instruction formats, instruction sets, and addressing modes available with typical microprocessors will be discussed.

Assembly Language Instruction Formats

Depending on the number of addresses specified, we have the following instruction formats:

- Three address
- Two address
- One address
- Zero address

Because all instructions are stored in the main memory, instruction formats are designed in such a way that instructions take less space and have more processing capabilities. It should be emphasized that the microprocessor architecture has considerable influence on a specific instruction format. The following are some important technical points that have to be considered while designing an instruction format:

- The size of an instruction word is chosen in such a way that it facilitates the specification of more operations by a designer. For example, with 4- and 8-bit op-code fields, we can specify 16 and 256 distinct operations respectively.
- Instructions are used to manipulate various data elements such as integers, floating-point numbers, and character strings. In particular, all programs written in a symbolic language such as C are internally stored as characters. Therefore, memory space will not be wasted if the word length of the machine is some integral multiple of the number

of bits needed to represent a character. Because all characters are represented using typical 8-bit character codes such as ASCII or EBCDIC, it is desirable to have 8-, 16-, 32-, or 64-bit words for the word length.

- The size of the address field is chosen in such a way that a high resolution is guaranteed. Note that in any microprocessor, the ultimate resolution is a bit. Memory resolution is function of the instruction length, and in particular, short instructions provide less resolution. For example, in a microcomputer with 32K 16-bit memory words, at least 19 bits are required to access each bit of the word. (This is because $2^{15} = 32K$ and $2^4 = 16$)

The general form of a *three address instruction* is shown below:

<op-code> Addr1, Addr2, Addr3

Some typical three-address instructions are

```
MUL   A, B, C          ;     C  <- A * B
ADD   A, B, C          ;     C  <- A + B
SUB   R1, R2, R3       ;     R3 <- R1 - R2
```

In this specification, all alphabetic characters are assumed to represent memory addresses, and the string that begins with the letter R indicates a register. The third address of this type of instruction is usually referred to as the "destination address." The result of an operation is always assumed to be saved in the destination address.

Typical programs can be written using these three address instructions. For example, consider the following sequence of three address instructions

```
MUL   A, B, R1         ;     R1 <- A * B
MUL   C, D, R2         ;     R2 <- C * D
MUL   E, F, R3         ;     R3 <- E * F
ADD   R1, R2, R1       ;     R1 <- R1 + R2
SUB   R1, R3, Z        ;     Z  <- R1 - R3
```

This sequence implements the statement `Z = A * B + C * D - E * F`. The three-address format is normally used by 32-bit microprocessors in addition to the other formats.

If we drop the third address from the three-address format, we obtain the two-address format. Its general form is

<op-code> Addr1, Addr2

Some typical *two-address* instructions are

```
MOV   A, R1        ;     R1 <- A
ADD   C, R2        ;     R2 <- R2 + C
SUB   R1, R2       ;     R2 <- R2 - R1
```

In this format, the addresses Addr1 and Addr2 respectively represent source and destination addresses. The following sequence of two-address instructions is equivalent to the program using three-address format presented earlier:

```
MOV   A, R1        ;     R1 <- A
MUL   B, R1        ;     R1 <- R1 * B
MOV   C, R2        ;     R2 <- C
MUL   D, R2        ;     R2 <- R2 * D
MOV   E, R3        ;     R3 <- E
MUL   F, R3        ;     R3 <- R3 * F
ADD   R2, R1       ;     R1 <- R1 + R2
SUB   R3, R1       ;     R1 <- R1 - R3
MOV   R1, Z        ;     Z  <- R1
```

This format is predominant in typical general-purpose microprocessors such as the Intel 8086 and the Motorola 68000. Typical 8-bit microprocessors such as the Intel 8085 and the Motorola 6809 are accumulator based. In these microprocessors, the accumulator register is assumed to be the destination for all arithmetic and logic operations. Also, this register always holds one of the source operands. Thus, we only need to specify one address in the instruction, and therefore, this idea reduces the instruction length. The one-address format is predominant in 8-bit microprocessors. Some typical one-address instructions are

```
LDA   B     ;     Acc <- B
ADD   C     ;     Acc <- Acc + C
MUL   D     ;     Acc <- Acc * D
STA   E     ;     E <- Acc
```

The following program illustrates how one can translate the statement `Z = A * B + C * D - E * F` into a sequence of one-address instructions:

```
LDA   E     ;     Acc <- E
MUL   F     ;     Acc <- Acc * F
STA   T1    ;     T1  <- Acc
LDA   C     ;     Acc <- C
MUL   D     ;     Acc <- Acc * D
STA   T2    ;     T2 <- Acc
LDA   A     ;     Acc <- A
MUL   B     ;     Acc <- Acc * B
ADD   T2    ;     Acc <- Acc + T2
SUB   T1    ;     Acc <- Acc - T1
STA   Z     ;     Z <- Acc
```

In this program, T1 and T2 represent the addresses of memory locations used to store temporary results. Instructions that do not require any addresses are called "zero-address instructions." All microprocessors include some zero-address instructions in the instruction set. Typical examples of zero-address instructions are `CLC` (clear carry) and `NOP`.

Typical Assembly Language Instruction Sets

An instruction set of a specific microprocessor consists of all the instructions that it can execute. The capabilities of a microprocessor are determined, to some extent, by the types of instructions it is able to perform. Each microprocessor has a unique instruction set designed by its manufacturer to do a specific task. We discuss some of the instructions that are common to all microprocessors. We will group chunks of these instructions together which have similar functions. These instructions typically include

- **Data Processing Instructions.** These operations perform actual data manipulations. The instructions typically include arithmetic/logic operations and increment/ decrement and rotate/shift operations. Typical arithmetic instructions include ADD, SUBTRACT, COMPARE, MULTIPLY, AND DIVIDE. Note that the SUBTRACT instruction provides the result and also affects the status flags while the COMPARE instruction performs subtraction without any result and affects the flags based on the result. Typical logic instructions perform traditional Boolean operations such as AND, OR, and EXCLUSIVE-OR. The AND instruction can be used to perform a masking operation. If the bit value in a particular bit position is desired in a word, the

word can be logically ANDed with appropriate data to accomplish this. For example, the bit value at bit 2 of an 8-bit number 0100 1Y10 (where unknown bit value of Y is to be determined) can be obtained as follows:

```
          0 1 0  0  1 Y 1 0  -- 8-bit number
AND       0 0 0  0  0 1 0 0  -- Masking data
          ---------------------
          0 0 0  0  0 Y 0 0  -- Result
```

If the bit value Y at bit 2 is 1, then the result is nonzero (Flag Z=0); otherwise, the result is zero (Flag Z=1). The Z flag can be tested using typical conditional JUMP instructions such as JZ (Jump if Z=1) or JNZ(Jump if Z=0) to determine whether Y is 0 or 1. This is called masking operation. The AND instruction can also be used to determine whether a binary number is ODD or EVEN by checking the Least Significant bit (LSB) of the number (LSB=0 for even and LSB=1 for odd). The OR instruction can typically be used to insert a 1 in a particular bit position of a binary number without changing the values of the other bits. For example, a 1 can be inserted using the OR instruction at bit number 3 of the 8-bit binary number 0 1 1 1 0 0 1 1 without changing the values of the other bits as follows:

```
          0 1 1 1 0 0 1 1 -- 8-bit number
OR        0 0 0 0 1 0 0 0 -- data for inserting a 1 at bit number 3
          -------------------
          0 1 1 1 1 0 1 1 -- Result
```

The Exclusive-OR instruction can be used to find the ones complement of a binary number by XORing the number with all 1's as follows:

```
          0 1 0 1 1 1 0 0 -- 8-bit number
XOR       1 1 1 1 1 1 1 1 -- data
          -------------------------
          1 0 1 0 0 0 1 1 -- Result (Ones Complement of the 8-bit number
                             0 1 0 1 1 1 0 0)
```

- **Instructions for Controlling Microprocessor Operations.** These instructions typically include those that set the reset specific flags and halt or stop the microprocessor.
- **Data Movement Instructions.** These instructions move data from a register to memory and vice versa, between registers, and between a register and an I/O device.
- **Instructions Using Memory Addresses.** An instruction in this category typically contains a memory address, which is used to read a data word from memory into a microprocessor register or for writing data from a register into a memory location. Many instructions under data processing and movement fall in this category.
- **Conditional and Unconditional JUMPS.** These instructions typically include one of the following:
  1. Unconditional JUMP, which always transfers the memory address specified in the instruction into the program counter.
  2. Conditional JUMP, which transfers the address portion of the instruction into the program counter based on the conditions set by one of the status flags in the flag register.

Typical Assembly Language Addressing Modes

One of the tasks performed by a microprocessor during execution of an instruction is the determination of the operand and destination addresses. The manner in which a microprocessor accomplishes this task is called the "addressing mode." Now, let us present the typical microprocessor addressing modes, relating them to the instruction sets of Motorola 68000.

An instruction is said to have "implied or inherent addressing mode" if it does not have any operand. For example, consider the following instruction: RTS, which means "return from a subroutine to the main program." The RTS instruction is a no-operand instruction. The program counter is implied in the instruction because although the program counter is not included in the RTS instruction, the return address is loaded in the program counter after its execution.

Whenever an instruction/operand contains data, it is called an "immediate mode" instruction. For example, consider the following 68000 instruction:

```
ADD    #15, D0        ;       D0 <- D0 + 15
```

In this instruction, the symbol # indicates to the assembler that it is an immediate mode instruction. This instruction adds 15 to the contents of register D0 and then stores the result in D0. An instruction is said to have a register mode if it contains a register as opposed to a memory address. This means that the operand values are held in the microprocessor registers. For example, consider the following 68000 instruction:

```
ADD    D1, D0         ; D0 <- D1 + D0
```

This ADD instruction is a two-operand instruction. Both operands (source and destination) have register mode. The instruction adds the 16-bit contents of D0 to the 16-bit contents of D1 and stores the 16-bit result in D0.

An instruction is said to have an absolute or direct addressing mode if it contains a memory address in the operand field. For example, consider the 68000 instruction

```
ADD    3000, D2
```

This instruction adds the 16-bit contents of memory address 3000 to the 16-bit contents of D2 and stores the 16-bit result in D2. The source operand to this ADD instruction contains 3000 and is in absolute or direct addressing mode. When an instruction specifies a microprocessor register to hold the address, the resulting addressing mode is known as the "register indirect mode." For example, consider the 68000 instruction:

```
CLR (A0)
```

This instruction clears the 16-bit contents of a memory location whose address is in register A0 to zero. The instruction is in register indirect mode.

The conditional branch instructions are used to change the order of execution of a program based on the conditions set by the status flags. Some microprocessors use conditional branching using the absolute mode. The op-code verifies a condition set by a particular status flag. If the condition is satisfied, the program counter is changed to the value of the operand address (defined in the instruction). If the condition is not satisfied, the program counter is incremented, and the program is executed in its normal order.

Typical 16-bit microprocessors use conditional branch instructions. Some conditional branch instructions are 16 bits wide. The first byte is the op-code for checking a particular flag. The second byte is an 8-bit offset, which is added to the contents of the program counter if the condition is satisfied to determine the effective address. This offset is considered as a signed binary number with the most significant bit as the sign bit. It means that the offset can vary from -128_{10} to $+127_{10}$ (0 being positive). This is called relative mode.

Consider the following 68000 example, which uses the branch not equal (BNE) instruction:

```
BNE 8
```

Suppose that the program counter contains 2000 (address of the next instruction to be executed) while executing this BNE instruction. Now, if $Z = 0$, the microprocessor will load 2000 + 8 = 2008 into the program counter and program execution resumes at address 2008. On the other hand, if $Z = 1$, the microprocessor continues with the next instruction.

In the last example the program jumped forward, requiring positive offset. An example for branching with negative offset is

```
BNE -14
```

Suppose that the current program counter value = 2004_{16}

$= 0010\ 0000\ \ 0000\ 0100$

offset = 2's complement of $14_{10} = F2_{16}$

$= 1111\ 1111\ \ 1111\ 0010$

$1\ 0001\ 1111\ \ 1111\ 0110$

ignore

$1\ \ F\ \ \ F\ \ 6_{16}$

reflect this 1 to the high byte (sign extension)

Therefore, to branch backward to $1FF6_{16}$, the assembler uses an offset of F2 following the op-code for BNE.

An advantage of relative mode is that the destination address is specified relaive to the address of the instruction after the instruction. Since these conditional Jump instructions do not contain an absolute address, the program can be placed anywhere in memory which can still be excuted properly by the microprocessor. A program which can be placed anywhere in memory, and can still run correctly is called a "relocatable" program. It is a good practice to write relocatable programs.

Subroutine Calls in Assembly Language

It is sometimes desirable to execute a common task many times in a program. Consider the case when the sum of squares of numbers is required several times in a program. One could write a sequence of instructions in the main program for carrying out the sum of squares every time it is required. This is all right for short programs. For long programs, however, it is convenient for the programmer to write a small program known as a "subroutine" for performing the sum of squares, and then call this program each time it is needed in the main program.

Therefore, a subroutine can be defined as a program carrying out a particular function that can be called by another program known as the "main program." The subroutine only needs to be placed once in memory starting at a particular memory location. Each time the main program requires this subroutine, it can branch to it, typically by using a jump to subroutine (JSR) instruction along with its starting address. The subroutine is then executed. At the end of the subroutine, a RETURN instruction takes control back to the main program.

The 68000 includes two subroutine call instructions. Typical examples include `JSR 4000` and `BSR 24`. `JSR 4000` is an instruction using absolute mode. In response to the execution of JSR, the 68000 saves (pushes) the current program counter contents (address of the next instruction to be executed) onto the stack. The program counter is then

loaded, with 4000 included in the `JSR` instruction. The starting address of the subroutine is 4000. The `RTS` (return from subroutine) at the end of the subroutine reads (pops) the return address saved into the stack before jumping to the subroutine into the program counter. The program execution thus resumes in the main program. `BSR 24` is an instruction using relative mode. This instruction works in the same way as the `JSR 4000` except that displacement `24` is added to the current program counter contents to jump to the subroutine.

The stack must always be balanced. This means that a PUSH instruction in a subroutine must be followed by a POP instruction before the RETURN from subroutine instruction so that the stack pointer points to the right return address saved onto the stack. This will ensure returning to the desired location in the main program after execution of the subroutine. If multiple registers are PUSHED in a subroutine, one must POP them in the reverse order before the subroutine RETURN instruction.

6.6.4 High-Level Languages

As mentioned before, the programmer's efficiency with assembly language increases significantly compared to machine language. However, the programmer needs to be well acquainted with the microprocessor's architecture and its instruction set. Further, the programmer has to provide an op-code for each operation that the microprocessor has to carry out in order to execute a program. As an example, for adding two numbers, the programmer would instruct the microprocessor to load the first number into a register, add the second number to the register, and then store the result in memory. However, the programmer might find it tedious to write all the steps required for a large program. Also, to become a reasonably good assembly language programmer, one needs to have a lot of experience.

High-level language programs composed of English-language-type statements rectify all these deficiencies of machine and assembly language programming. The programmer does not need to be familiar with the internal microprocessor structure or its instruction set. Also, each statement in a high-level language corresponds to a number of assembly or machine language instructions. For example, consider the statement `F = A + B` written in a high-level language called FORTRAN. This single statement adds the contents of A with B and stores the result in F. This is equivalent to a number of steps in machine or assembly language, as mentioned before. It should be pointed out that the letters A, B, and F do not refer to particular registers within the microprocessor. Rather, they are memory locations.

A number of high-level languages such as C and C++ are widely used these days. Typical microprocessors, namely, the Intel 8086, the Motorola 68000, and others, can be programmed using these high-level languages. A high-level language is a problem-oriented language. The programmer does not have to know the details of the architecture of the microprocessor and its instruction set. Basically, the programmer follows the rules of the particular language being used to solve the problem at hand. A second advantage is that a program written in a particular high-level language can be executed by two different microcomputers, provided they both understand that language. For example, a program written in C for an Intel 8086–based microcomputer will run on a Motorola 68000-based microcomputer because both microprocessors have a compiler to translate the C language into their particular machine language; minor modifications are required for input/output programs.

As mentioned before, like the assembly language program, a high-level language

program requires a special program for converting the high-level statements into object codes. This program can be either an interpreter or a compiler. They are usually very large programs compared to assemblers.

An interpreter reads each high-level statement such as `F = A + B` and directs the microprocessor to perform the operations required to execute the statement. The interpreter converts each statement into machine language codes but does not convert the entire program into machine language codes prior to execution. Hence, it does not generate an object program. Therefore, an interpreter is a program that executes a set of machine language instructions in response to each high-level statement in order to carry out the function. A compiler, however, converts each statement into a set of machine language instructions and also produces an object program that is stored in memory. This program must then be executed by the microprocessor to perform the required task in the high-level program. In summary, an interpreter executes each statement as it proceeds, without generating an object code, whereas a compiler converts a high-level program into an object program that is stored in memory. This program is then executed. Compilers normally provide inefficient machine codes because of the general guidelines that must be followed for designing them. C, C++, and Java are the only high-level languages that include Input/Output instructions. However, the compiled codes generate many more lines of machine code than an equivalent assembly language program. Therefore, the assembled program will take up less memory space and will execute much faster compared to the compiled C, C++, or Java codes. I/O programs written in C are compared with assembly language programs written in 8086 and 68000 in Chapters 9 and 10. C language is a popular high-level language, the C++ language, based on C, is also very popular, and Java, developed by Sun Microsystems, is gaining wide acceptance.

Therefore, one of the main uses of assembly language is in writing programs for real-time applications. "Real-time" means that the task required by the application must be completed before any other input to the program can occur which will change its operation. Typical programs involving non-real-time applications and extensive mathematical computations may be written in C, C++, or Java. A brief description of these languages is given in the following.

C Language

The C Programming language was developed by Dennis Ritchie of Bell Labs in 1972. C has become a very popular language for many engineers and scientists, primarily because it is portable except for I/O and however, can be used to write programs requiring I/O operations with minor modifications. This means that a program written in C for the 8086 will run on the 68000 with some modifications related to I/O as long as C compilers for both microprocessors are available.

C is case sensitive. This means that uppercase letters are different from lowercase letters. Hence Start and start are two different variables. C is a general-purpose programming language and is found in numerous applications as follows:

- **Systems Programming.** Many operating systems, compilers, and assemblers are written in C. Note that an operating system typically is included with the personal computer when it is purchased. The operating system provides an interface between the user and the hardware by including a set of commands to select and execute the software on the system
- **Computer-Aided Design (CAD) Applications.** CAD programs are written in C. Typical tasks to be accomplished by a CAD program are logic synthesis and

simulation.

- **Numerical Computation.** To solve mathematical problems such as integration and differentiation
- **Other Applications.** These include programs for printers and floppy disk controllers, and digital control algorithms using single-chip microcomputers.

A C program may be viewed as a collection of functions. Execution of a C program will always begin by a call to the function called "main." This means that all C programs should have its main program named as **`main`**. However, one can give any name to other functions.

A simple C program that prints "I wrote a C-program" is

```
/* First C-program */
#include <stdio.h>
main ( )
{
   printf("I wrote a C-program");
}
```

Here, `main` is a function of no arguments, indicated by ( ). The parenthesis must be present even if there are no arguments. The braces { } enclose the statements that make up the function.

The line `printf("I wrote a C-program");` is a function call that calls a function named `printf`, with the argument "`I wrote a C-program.`" `printf` is a library function that prints output on the terminal. Note that `/* */` is used to enclose comments. These are not translated by the compiler.

A variation of the C program just described is

```
/* Another C program */
#include <stdio.h>
main ( )
{
   printf("I wrote");
   printf(" a C-");
   printf("program");
   printf("\n");
}
```

Here, `#include` is a preprocessor directive for the C language compiler. These directives give instructions to the compiler that are performed before the program is compiled. The directive `#include <stdio.h>` inserts additional statements in the program. These statements are contained in the file stdio.h. The file `stdio.h` is included with the standard C library. The `stdio.h` file contains information related to the input/output statement.

The `\n` in the last line of the program is C notation for the newline character. Upon printing, the cursor moves forward to the left margin on the next line. `printf` never supplies a newline automatically. Therefore, multiple `printf`'s may be used to output "I wrote a C-program" on a single line in a few steps. The escape sequence `\n` can be used to print three statements on three different lines. An illustration is given in the following:

```
#include <stdio.h>
main ( )
{
   printf("I wrote a C-Program \n");
```

```
    printf("This will be printed on a new line \n");
    printf("So also is this line \n");
}
```

All variables in C must be declared before use, normally at the start of the function before any executable statements. The compiler provides an error message if one forgets a declaration. A declaration includes a type and a list of variables that have that type. For example, the declaration `int a, b` implies that the variables `a` and `b` are integers. Next, write a program to add and subtract two integers a and b where a = 100 and b = 200. The C program is

```
#include <stdio.h>
main ( )
{
    int a = 100, b = 200;          /*a and b are integers
*/
    printf("The sum is: %d \n", a + b);
    printf("The difference is: %d \n", a - b);
}
```

The `%d` in the `printf` statement represents "decimal integer." Note that `printf` is not part of the C language; there is no input or output defined in C itself. `printf` is a function that is contained in the standard library of routines that can be accessed by C programs. The values of `a` and `b` can be entered via the keyboard by using the scanf function. The scanf allows the programmer to enter data from the keyboard. A typical expression for `scanf` is

```
scanf("%d%d", &a, &b);
```

This expression indicates that the two values to be entered via the keyboard are in decimal. These two decimal numbers are to be stored in addresses `a` and `b`. Note that the symbol `&` is an address operator.

The C program for adding and subtracting two integers `a` and `b` using `scanf` is

```
/* C Program that performs basic I/O */
#include <stdio.h>
main ( )
{
    int a,b;
    printf("Input two integers: ");
    scanf("%d%d", &a, &b);
    printf("Their sum is: %d\n", a + b);
    printf("Their difference is: %d\n", a - b);
}
```

In summary, writing a working C program involves four steps as follows:

Step 1: Using a text editor, prepare a file containing the C code. This file is called the "source file."

Step 2 Preprocess the code. The preprocessor makes the code ready for compiling. The preprocessor looks through the source file for lines that start with a #. In the previous programming examples, `#include <stdio.h>` is a preprocessor. This preprocessor instruction copies the contents of the standard header file `stdio.h` into the source code. This header file `stdio.h` describes typical input/output functions such as `scanf( )` and `printf( )` functions.

Step 3: The compiler translates the preprocessed code into machine code. The output from the compiler is called object code.

Step 4: The linker combines the object file with code from the C libraries. For instance, in the examples shown here, the actual code for the library function `printf( )` is inserted from the standard library to the object code by the linker. The linker generates an executable file. Thus, the linker makes a complete program.

Before writing C programs, the programmer must make sure that the computer runs either the UNIX or MS-DOS operating system. Two essential programming tools are required. These are a text editor and a C compiler. The text editor is a program provided with a computer system to create and modify compiler files. The C compiler is also a program that translates C code into machine code.

C++

C++ is a modified version of C language. C++ was developed by Bjarne Stroustrup of Bell Labs in 1980. It includes all features of C and also supports object-oriented programming (OOP). A program can be divided into subprograms using OOP. Each subprogram is an independent object with its own instructions and data. Thus, complexity of programming is reduced. It is therefore easier for the programmer to manage larger programs.

All OOP languages including C++, have three characteristics: encapsulation, polymorphism, and inheritance. *Encapsulation* is a technique that keeps code and data together in such a way that they are protected form outside interference and misuse. A subprogram thus created is called an "object."

Code, data, or both may be private or public. Private code and/or data may be accessed by another part of the same object. On the other hand, public code and/or data may be accessed by a program resident outside the object containing them. One of the most important characteristic of C++ is the class. The class declaration is a technique for creating an object. Note that a class consists of data and functions.

Encapsulation is available with C to some extent. For example, when a library function such as `printf` is used, one uses a black box program. When `printf` is used, several internal variables are created and intialized that are not accessible to the programmer.

Polymorphism (from Greek word meaning "several forms") allows one to define a general class of actions. Within a general class, the specific action is determined by the type of data. For example, in C, the absolute value actions `abs( )` and `fabs( )` compute the absolute values of an integer and a floating point number respectively. In C++, on the other hand, one absolute value action, `abs( )` is used for both data types. The type of data is then used to call `abs( )` to determine which specific version of the function is actually used. Thus, one function name for two different data items is used.

Inheritance is the ability by which one class called subclass obtains the properties of another class called a superclass. Inheritance is convenient for code reusability. Inheritance supports hierarchy classes.

Following are some basic differences between C and C++:

1. In C, one must use `void` with the prototype for a function with no arguments. For example, in C, the prototype `int rand(void);` returns an integer that is a random number.

   In C++, the `void` is optional. Therefore, in C++, the prototype for `rand( )` can be written as `int rand( );`. Of course, `int rand(void);` is a

valid prototype in C++. This means that both prototypes are allowed in C++

2. C++ can use the C type of comment mechanism. That is, a comment can start with `/*` and end with `*/`. C++ can also use a simple line comment that starts with a `//` and stops at the end of the line terminated by a carriage return. Typically, C++ uses C-like comments for multiline comments and the C++ comment mechanism for short comments.
3. In C++, local variables can be declared anywhere. In contrast, in C, local variables must be declared at the start of a block before any action statements.
4. In C++, all functions need to be prototyped. In C, prototypes are optional. Note that a function prototype allows the compiler to check that the function is called with the proper number and types of arguments. It also tells the compiler the type of value that the function is supposed to return. In C, if the function prototype is omitted, the compiler will return an integer. An example of a prototype function is `int abs(int n)` , this provides an integer that is an absolute value of `n`.

Java

Introduced in 1991 by Sun MicroSystems, Java is based on C++ and is a true object oriented language. That is, everything in a Java program is an object and everything is obtained from a single object class.

A Java program must include at least one class. A class includes data type declarations and statements. Every Java standalone program requires a main method at the beginning. Java only supports class methods and not separate functions. There is no preprocessor in Java. However, there is an `import` statement, which is similar to the `#include` preprocessor statement in C. The purpose of the `import` statement in Java is to instruct the interpreter to load the class, which exists in another compilation statement. Java uses the same comment syntax, `/* */` and `//`, as C and C++. In addition, a special comment syntax, `/** */`, that can precede declarations is used in Java.

Java does not require pointers. In C, a pointer may be substituted for the array name to access array elements. In Java, arrays are created by using the "new" operator by including the size of the array in the new expression (rather than in the declaration) as follows:

```
int array [ ] = new int[6];
```

Also, all arrays store the specified size in a variable named `length` as follows:

```
int stringsize = array.length;
```

Therefore, in Java, arrays and strings are not subject to the errors or confusion that is common to arrays and strings in C.

6.7 Monitors

A monitor consists of a number of subroutines grouped together to provide "intelligence" to a microcomputer system. This intelligence gives the microcomputer with the capabilities for software development of user programs such as assembling and debugging. The monitor is typically offered by the microprocessor manufacturers and others in a ROM or CD memory. When a microcomputer is designed by connecting the microprocessor, memory, and I/O, a monitor program can be used for development of user programs.

An example of a monitor is the Intel SDK-86 monitor, which contains debugging

routines, a display routine, and many other programs. The user can assemble, debug, execute and display results for user-written 8086 assembly language programs using the monitor provided by Intel with the SDK-86 microcomputer.

6.8 Flowcharts

Before writing an assembly language program for a specific operation, it is convenient to represent the program in a schematic form called *flowchart.* A brief listing of the basic shapes used in a flowchart and their functions is given in Figure 6.32.

6.9 Basic Features of Microcomputer Development Systems

A microcomputer development system is a tool that allows the designer to develop, debug, and integrate error-free application software in microprocessor systems.

Development systems fall into one of two categories: systems supplied by the device manufacturer (nonuniversal systems) and systems built by after-market manufacturers (universal systems). The main difference between the two categories is the range of microprocessors that a system will accommodate. Nonuniversal systems are supplied by the microprocessor manufacturer (Intel, Motorola) and are limited to use for the particular microprocessor manufactured by the supplier. In this manner, an Intel development system may not be used to develop a Motorola-based system. The universal development systems (Hewlett-Packard, Tektronix) can develop hardware and software for several microprocessors.

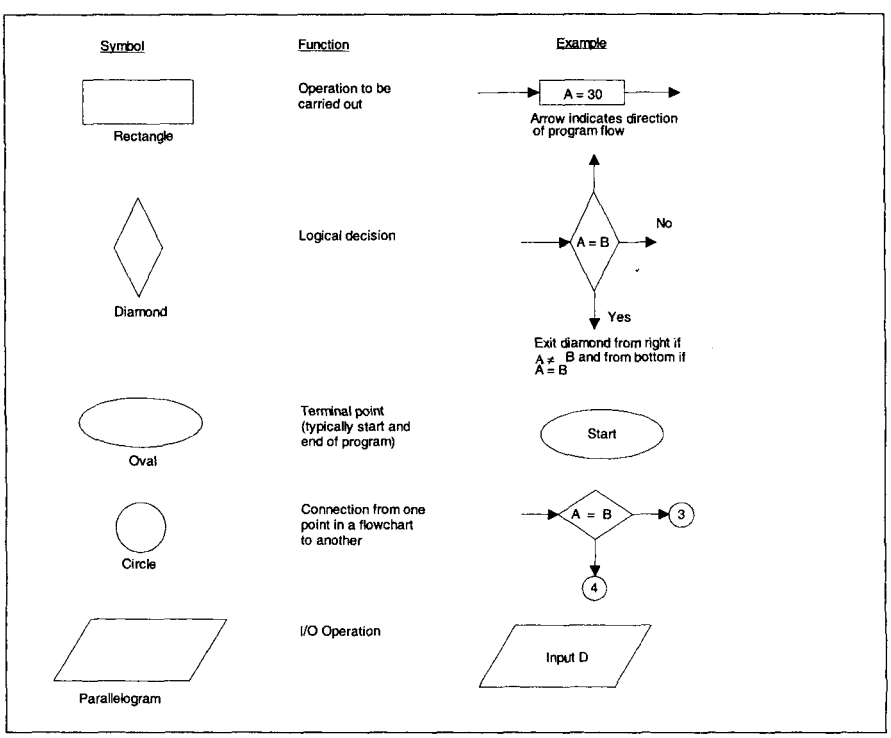


FIGURE 6.32 Flowchart symbols

Within both categories of development systems, there are basically three types available: single-user systems, time-shared systems, and networked systems. A single-user system consists of one development station that can be used by one user at a time. Single-user systems are low in cost and may be sufficient for small systems development. Time-shared systems usually consist of a "dumb" type of terminal connected by data lines to a centralized microcomputer-based system that controls all operations. A networked system usually consists of a number of smart cathode ray tubes (CRTs) capable of performing most of the development work and can be connected over data lines to a central microcomputer. The central microcomputer in a network system usually is in charge of allocating disk storage space and will download some programs into the user's workstation microcomputer. A microcomputer development system is a combination of the hardware necessary for microprocessor design and the software to control the hardware. The basic components of the hardware are the central processor, the CRT terminal, mass storage device (floppy or hard disk), and usually an in-circuit emulator (ICE).

In a single-user system, the central processor executes the operating system software, handles the input/output (I/O) facilities, executes the development programs (editor, assembler, linker), and allocates storage space for the programs in execution. In a large multiuser networked system the central processor may be responsible for the I/O facilities and execution of development programs. The CRT terminal provides the interface between the user and the operating system or program under execution. The user enters commands or data via the CRT keyboard, and the program under execution displays data to the user via the CRT screen. Each program (whether system software or user program) is stored in an ordered format on disk. Each separate entry on the disk is called a *file*. The operating system software contains the routines necessary to interface between the user and the mass storage unit. When the user requests a file by a specific *file name*, the operating system finds the program stored on disk by the file name and loads it into mean memory. More advanced development systems contain *memory management* software that protects a user's files from unauthorized modification by another user. This is accomplished via a unique user identification code called USER ID. A user can only access files that have the user's unique code. The equipment listed here makes up a basic development system, but most systems have other devices such as printers and EPROM and PAL programmers attached. A printer is needed to provide the user with a hard copy record of the program under development.

After the application system software has been completely developed and debugged, it needs to be permanently stored for execution in the target hardware. The EPROM (erasable/programmable read-only memory) programmer takes the machine code and programs it into an EPROM. EPROMs are more generally used in system development because they may be erased and reprogrammed if the program changes. EPROM programmers usually interface to circuits particularly designed to program a specific EPROM.

Most development systems support one or more in-circuit emulators (ICEs). The ICE is one of the most advanced tools for microprocessor hardware development. To use an ICE, the microprocessor chip is removed from the system under development (called the target processor) and the emulator is plugged into the microprocessor socket. The ICE will functionally and electrically act identically to the target processor with the exception that the ICE is under the control of development system software. In this manner the development system may exercise the hardware that is being designed and monitor all status information available about the operation of the target processor. Using an ICE,

processor register contents may be displayed on the CRT and operation of the hardware observed in a single-stepping mode. In-circuit emulators can find hardware and software bugs quickly that might take many hours to locate using conventional hardware testing methods.

Architectures for development systems can be generally divided into two categories: the master/slave configuration and the single-processor configuration. In a master/slave configuration, the master (host) processor controls the mass storage device and processes all I/O (CRT, printer). The software for development systems is written for the master processor, which is usually not the same as the slave (target) processor. The slave microprocessor is typically connected to the user prototype via a connector which links the slave processor to the master processor.

Some development systems such as the HP 64000 completely separate the system bus from the emulation bus and therefore use a separate block of memory for emulation. This separation allows passive monitoring of the software executing on the target processor without stopping the emulation process. A benefit of the separate emulation facilities allows the master processor to be used for editing, assembling, and so on while the slave processor continues the emulation. A designer may therefore start an emulation running, exit the emulator program, and at some future time return to the emulation program.

Another advantage of the separate bus architecture is that an operating system needs to be written only once for the master processor and will be used no matter what type of slave processor is being emulated. When a new slave processor is to be emulated, only the emulator probe needs to be changed.

A disadvantage of the master/slave architecture is that it is expensive. In single-processor architecture, only one processor is used for system operation and target emulation. The single processor does both jobs, executing system software as well as acting as the target processor. Because there is only one processor involved, the system software must be rewritten for each type of processor that is to be emulated. Because the system software must reside in the same memory used by the emulator, not all memory will be available to the emulation process, which may be a disadvantage when large prototypes are being developed. The single-processor systems are inexpensive.

The programs provided for microprocessor development are the operating system, editor, assembler, linker, compiler, and debugger. The operating system is responsible for executing the user's commands. The operating system handles I/O functions, memory management, and loading of programs from mass storage into RAM for execution. The editor allows the user to enter the source code (either assembly language or some high-level language) into the development system.

Almost all current microprocessor development systems use the character-oriented editor, more commonly referred to as the screen editor. The editor is called a "screen editor" because the text is dynamically displayed on the screen and the display automatically updates any edits made by the user.

The screen editor uses the pointer concept to point to the character(s) that need editing. The pointer in a screen editor is called the "cursor," and special commands allow the user to position the cursor to any location displayed on the screen. When the cursor is positioned, the user may insert characters, delete characters, or simply type over the existing characters.

Complete lines may be added or deleted using special editor commands. By placing the editor in the insert mode, any text typed will be inserted at the cursor position when the cursor is positioned between two existing lines. If the cursor is positioned on a

line to be deleted, a single command will remove the entire line from the file.

Screen editors implement the editor commands in different fashions. Some editors use dedicated keys to provide some cursor movements. The cursor keys are usually marked with arrows to show the direction of the cursor movement. More advanced editors (such as the HP 64000) use soft keys. A soft key is an unmarked key located on the keyboard directly below the bottom of the CRT screen. The mode of the editor decides what functions the keys are to perform. The function of each key is displayed on the screen directly above the appropriate key. The soft key approach is valuable because it allows the editor to reassign a key to a new function when necessary.

The source code generated on the editor is stored as ASCII or text characters and cannot be executed by a microprocessor. Before the code can be executed, it must be converted to a form accessible by the microprocessor. An assembler is the program used to translate the assembly language source code generated with an editor into object code (machine code), which may be executed by a microprocessor.

The output file from most development system assemblers is an object file. The object file is usually relocatable code that may be configured to execute at any address. The function of the linker is to convert the object file to an *absolute* file, which consists of the actual machine code at the correct address for execution. The absolute files thus created are used for debugging and finally for programming EPROMs.

Debugging a microprocessor-based system may be divided into two categories: software debugging and hardware debugging. Both debugging processes are usually carried out separately because software debugging can be carried out on an out-of-circuit emulator (OCE) without having the final system hardware.

The usual software development tools provided with the development system are

- Single-step facility
- Breakpoint facility

A single stepper simply allows the user to execute the program being debugged one instruction at a time. By examining the register and memory contents during each step, the debugger can detect such program faults as incorrect jumps, incorrect addressing, erroneous op-codes, and so on. A breakpoint allows the user to execute an entire section of a program being debugged.

There are two types of breakpoints: hardware and software. The hardware breakpoint uses the hardware to monitor the system address bus and detect when the program is executing the desired breakpoint location. When the breakpoint is detected, the hardware uses the processor control lines to halt the processor for inspection or cause the processor to execute an interrupt to a breakpoint routine. Hardware breakpoints can be used to debug both ROM- and RAM-based programs. Software breakpoint routines may only operate on a system with the program in RAM because the breakpoint instruction must be inserted into the program that is to be executed.

Single-stepper and breakpoint methods complement each other. The user may insert a breakpoint at the desired point and let the program execute up to that point. When the program stops at the breakpoint the user may use a single-stepper to examine the program one instruction at a time. Thus, the user can pinpoint the error in a program.

There are two main hardware-debugging tools: the logic analyzer and the in-circuit emulator. Logic analyzers are usually used to debug hardware faults in a system. The logic analyzer is the digital version of an oscilloscope because it allows the user to view logic levels in the hardware. In-circuit emulators can be used to debug and integrate software and hardware. PC-based workstations are extensively used as development systems.

6.10 System Development Flowchart

The total development of a microprocessor-based system typically involves three phases: software design, hardware design, and program diagnostic design. A systems programmer will be assigned the task of writing the application software, a logic designer will be assigned the task of designing the hardware, and typically both designers will be assigned the task of developing diagnostics to test the system. For small systems, one engineer may do all three phases, while on large systems several engineers may be assigned to each phase. Figure 6.33 shows a flowchart for the total development of a system. Notice that software and hardware development may occur in parallel to save time.

The first step in developing the software is to take the system specifications and write a flowchart to accomplish the desired tasks that will implement the specifications. The assembly language or high-level source code may now be written from the system flowchart. The complete source code is then assembled. The assembler is the object code and a program listing. The object code will be used later by the linker. The program listing may be sent to a disk file for use in debugging, or it may be directed to the printer.

The linker can now take the object code generated by the assembler and create

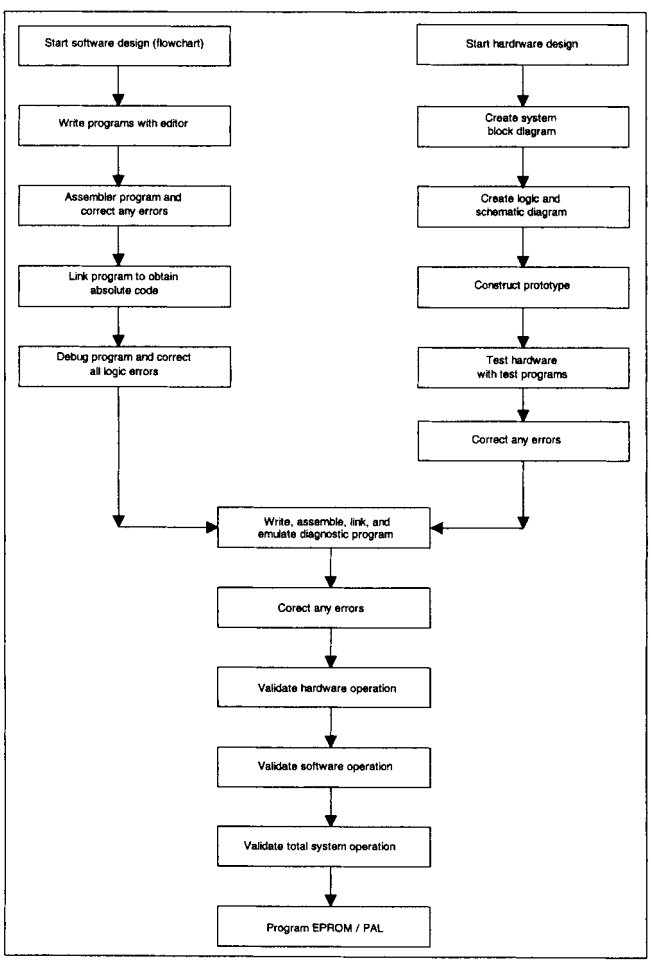


FIGURE 6.33 Microprocessor system development flowchart

the final absolute code that will be executed on the target system. The emulation phase will take the absolute code and load it into the development system RAM. From here, the program may be debugged using breakpoints or single stepping.

Working from the system specifications, a block diagram of the hardware must be developed. The logic diagram and schematics may now be drawn using the block diagram as a guide, and a prototype may now be constructed and tested for wiring errors. When the prototype has been constructed it may be debugged for correct operation using standard electronic testing equipment such as oscilloscopes, meters, logic probes, and logic analyzers, all with test programs created for this purpose. After the prototype has been debugged electrically, the development system in-circuit emulator may be used to check it functionally. The ICE will verify the memory map, correct I/O operation, and so on. The next step in system development is to validate the complete system by running operational checks on the prototype with the finalized application software installed. The EPROMs and/or PALs are then programmed with the error-free programs.

QUESTIONS AND PROBLEMS

6.1 What is the difference between a single-chip microprocessor and a single-chip microcomputer?

6.2 What is a microcontroller? Name one commercially available microcontroller.

6.3 What is the difference between:
(a) The program counter (PC) and the memory address register (MAR)?
(b) The accumulator (A) and the instruction register (IR)?
(c) General-purpose register-based microprocessor and accumulator-based microprocessor. Name a commercially available microprocessor of each type.

6.4 Assuming signed numbers, find the sign, carry, zero, and overflow flags of:
(a) $09_{16} + 17_{16}$.
(b) $A5_{16} - A5_{16}$
(c) $71_{16} - A9_{16}$
(d) $6E_{16} + 3A_{16}$
(e) $7E_{16} + 7E_{16}$

6.5 What is meant by PUSH and POP operations in the stack?

6.6 Suppose that an 8-bit microprocessor has a 16-bit stack pointer and uses a 16-bit register to access the stack from the top. Assume that initially the stack pointer and the 16-bit register contain $20C0_{16}$ and 0205_{16} respectively. After the PUSH operation:
(a) What are the contents of the stack pointer?
(b) What are the contents of memory locations $20BE_{16}$ and $20BF_{16}$?

6.7 Assuming the microprocessor architecture of Figure 6.18, write down a possible sequence of microinstructions for finding the ones complement of an 8-bit number. Assume that the number is already in the register.

6.8 What do you mean by a multiplexed address and data bus?

6.9 Name four general-purpose registers in the 8086.

6.10 Name one 8086 register that can be used to hold an address in a segment.

6.11 What is the difference between EPROM and PROM? Are both types available with bipolar and also MOS technologies?

6.12 Assuming a single clock signal and four registers (PC, MAR, Reg, and IR) for a microprocessor, draw a timing diagram for loading the memory address register. Explain the sequence of events relating them to the four registers.

6.13 Given a memory with a 14-bit address and 8-bit word size.
(a) How many bytes can be stored in this memory?
(b) If this memory were constructed from 1K × 1-bit RAMs, how many memory chips would be required?
(c) How many bits would be used for chip select?

6.14 Define the three types of I/O. Identify each one as either "microprocessor initiated" or "device initiated."

6.15 What is the basic difference between a compiler and an assembler?

6.16 Write a program equivalent to the Pascal assignment statement:

```
Z := (A + (B * C) + (D * E) - (F / G) - (H * I)
```

Use only
(a) Three-address instructions
(b) Two-address instructions

6.17 Describe the meaning of each one of the following addressing modes.

(a)	Immediate	(d)	Register indirect
(b)	Absolute	(e)	Relative
(c)	Register	(f)	Implied

6.18 Assume that a microprocessor has only two registers R1 and R2 and that only the following instruction is available:

```
XOR   Ri, Rj        ;      Rj <- Ri ⊕ Rj
                    ;      i,j = 1,2
```

Using this XOR instruction, find an instruction sequence in order to exchange the contents of registers R1 and R2

6.19 What are the advantages of subroutines?

6.20 Explain the use of a stack in implementing subroutine calls.

6.21 Determine the contents of address 5004_{16} after assembling the following:

(a)
```
ORG  5002H
DB   00H, 05H, 07H, 00H, 03H
```

(b)
```
ORG  5000H
DW   0702H, 123FH, 7020H, 0000H
```

6.22 What is the difference between:

(a) A cross assembler and a resident assembler

(b) A two-pass assembler and meta-assembler

(c) Single step and breakpoint

6.23 Identify some of the differences between C, C++, and Java.

6.24 How does a microprocessor obtain the address of the first instruction to be executed?

6.25 Summarize the basic features of a typical microcomputer development system.

6.26 Discuss the steps involved in designing a microprocessor-based system.

Fundamentals of Digital Logic and Microcomputer Design. M. Rafiquzzaman

7

DESIGN OF COMPUTER INSTRUCTION SET AND THE CPU

This chapter describes the design of the instruction set and the central processor unit (CPU). Topics include op-code encoding, design of typical microprocessor registers, the arithmetic logic unit (ALU), and the control unit.

7.1 Design of the Computer Instructions

A program consists of a sequence of instructions. An instruction performs operations on stored data. There are two components in an instruction: an op-code field and an address field. The op-code field defines the type of operation to be performed on data, which may be stored in a microprocessor register or in the main memory. The address field may contain one or more addresses of data. When data are read from or stored into two or more addresses by the instruction, the address field may contain more than one address. For example, consider the following instruction:

```
MOVE        D0,D1
```

Op-code field Address field

Assume that this computer uses D0 as the source register and D1 as the destination register. This instruction moves the contents of the microprocessor register D0 to register D1. The number and types of instructions supported by a microprocessor vary from one microprocessor to another and primarily depend on the microprocessor architecture. The number of instructions supported by a typical microprocessor depends on the size of the op-code field. For example, an 8-bit op-code can specify a maximum of 256 unique instructions.

As mentioned before, a computer only understands 1's and 0's. This means that the computer can execute an instruction only if it is in binary. A unique binary pattern must be assigned to each op-code by a process called "op-code encoding."

The Block code method is one of the simplest techniques of designing instructions. In this approach, a fixed length of binary pattern is assigned to each op-code. For example, an n-bit binary number can represent 2^n unique op-codes. Consider for example, a hypothetical instruction set shown in Figure 7.1. In this figure, there are 8 different instructions that can be encoded using three bits i_2, i_1, i_0 as shown in Figure 7.2. A 3-to-8 decoder can be used to encode the 8 hypothetical instructions as shown in Figure 7.3.

An n-to-2^n decoder is required for an n-bit op-code. As n increases, the cost of the decoder and decoding time will also increase. In some op-code encoding techniques such as

the "expanding op-code" method, the length of the instruction is a function of the number of addresses used by the instruction. For example, consider a 16-bit instruction in which the lengths of the op-code and address fields are 5 bits and 11 bits respectively. Using such an instruction format, 32 (2^5) operations allowing access to 2048 (2^{11}) memory locations can be specified. Now, if the size of the instruction is kept at 16 bits but the address field is increased to 12 bits, the op-code length will then be decreased to 4 bits. This change will specify 16 (2^4) operations with access to 4096 (2^{12}) memory locations. Thus, the number of

Instruction	Operation Performed
MOVE reg_1, reg_2	$reg_2 \leftarrow reg_1$
CLR reg	reg ← 0
ADD reg_1, reg_2	$reg_2 \leftarrow reg_1 + reg_2$
SUB reg_1, reg_2	$reg_2 \leftarrow reg - reg_1$
AND reg_1, reg_2	$reg_2 \leftarrow reg_1$ AND reg_2
OR reg_1, reg_2	$reg_2 \leftarrow reg_1$ OR reg_2
INC reg	reg ← reg + 1
JMP addr	PC ← addr; Unconditionally Jump to addr

FIGURE 7.1 A hypothetical instruction set

Instruction	3-Bit Op-Code		
	i_2	i_1	i_0
MOVE	0	0	0
CLR	0	0	1
ADD	0	1	0
SUB	0	1	1
AND	1	0	0
OR	1	0	1
INC	1	1	0
JMP	1	1	1

FIGURE 7.2 Op-code encoding using block code

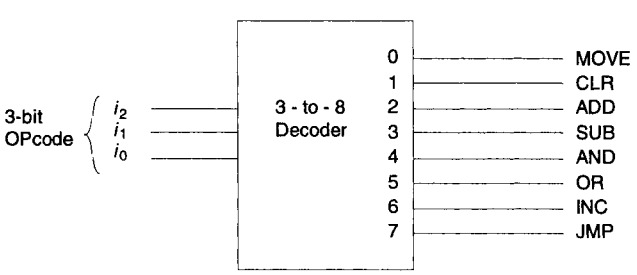


FIGURE 7.3 Instruction decoder

operations is reduced by 50% and the number of memory locations is increased by 100%. This concept is used in designing instructions with expanding op-code technique.

Consider an instruction format with 8-bit instruction length and a 2-bit op-code field. Four unique two-address (3 bits for each address) instructions can be specified. This is depicted in Figure 7.4. If three rather than four two-address instructions are used, eight one-address instructions can be specified. This is shown in Figure 7.5. The length of the op-code field for each one-address instruction is 5 bits. Thus, the length of the op-code field increases as the number of address field is decreased. Now, if the total number of one-address instructions is reduced from 8 to 7, then eight 0-address instructions can also be specified. This is shown in Figure 7.6.

7.2 Reduced Instruction Set Computer (RISC)

RISC, which stands for reduced instruction set computer, is a generation of faster and inexpensive machines. The initial application of RISC principles has been in desktop workstations. Note that the PowerPC is a RISC microprocessor. The basic idea behind

OP- Code (2-bits)	Address 1 (3-bits)	Address 2 (3-bits)
$i_1\ i_0$		
0 0	$x_2\ x_1\ x_0$	$y_2\ y_1\ y_0$
0 1	$x_2\ x_1\ x_0$	$y_2\ y_1\ y_0$
1 0	$x_2\ x_1\ x_0$	$y_2\ y_1\ y_0$
1 1	$x_2\ x_1\ x_0$	$y_2\ y_1\ y_0$

FIGURE 7.4 Four two-address instructions

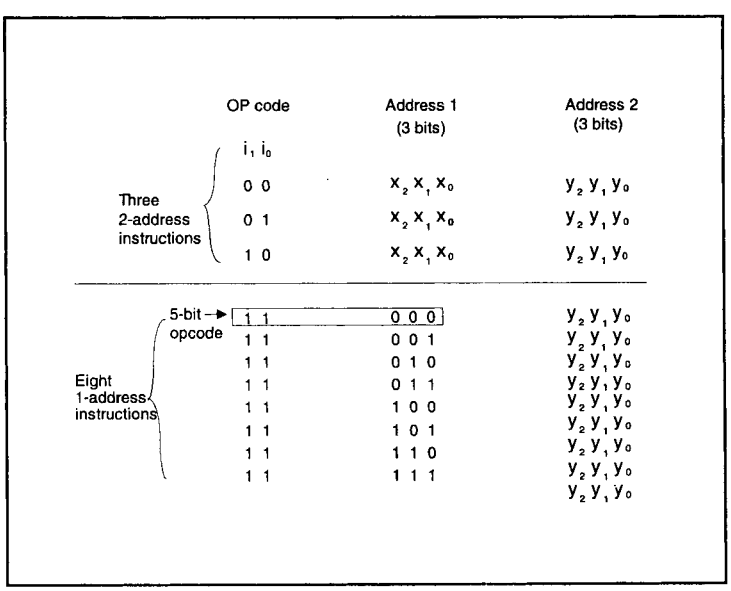


FIGURE 7.5 Three 2-address and eight 1-address instructions

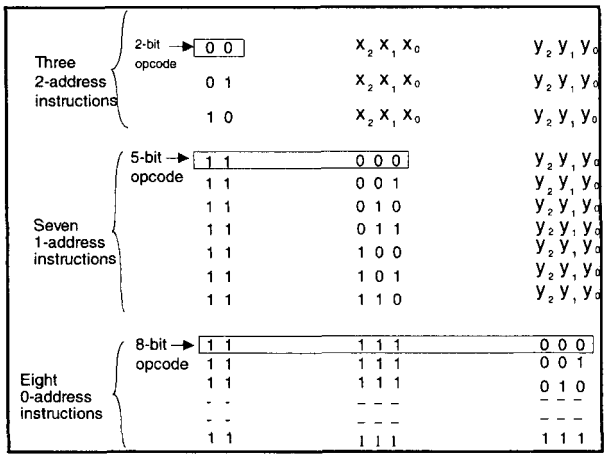


FIGURE 7.6 3 two-address, 7 one-address, and 8 zero-address instructions

RISC is for machines to cost less yet run faster, by using a small set of simple instructions for their operations. Also, RISC allows a balance between hardware and software based on functions to be achieved to make a program run faster and more efficiently. The philosophy of RISC is based on six principles: reliance on optimizing compilers, few instructions and addressing modes, fixed instruction format, instructions executed in one machine cycle, only call/return instructions accessing memory, and hardwired control.

The trend has always been to build CISCs (complex instruction set computers), which use many detailed instructions. However, because of their complexity, more hardware would have to be used. The more instructions, the more hardware logic is needed to implement and support them. For example, in a RISC machine, an ADD instruction takes its data from registers. On a CISC, each operand can be stored in any of many different forms, so the compiler must check several possibilities. Thus, both RISC and CISC have advantages and disadvantages. However, the principles of understanding optimizing compilers and what actually happens when a program is executed lead to RISC.

Case Study: RISC I (University of California, Berkeley)

The RISC machine presented in this section is the one investigated at the University of California, Berkeley. The RISC I is designed with the following design constraints:

1. Only one instruction is executed per cycle.
2. All instructions have the same size.
3. Only load and store instructions can access memory.
4. High-level languages (HLL) are supported.

Two high level Languages (C and Pascal) were supported by RISC I. A simple architecture implies a fewer transistors, and this leads to the fact that most pieces of a RISC HLL system are in software. Hardware is utilized for time-consuming operations. Using C and Pascal, a comparison study was made to determine the frequency of occurrence of particular variable and statement types. Studies revealed that integer constants appeared most frequently, and a study of the code produced revealed that the procedure calls are the most time-consuming operations.

i) Basic RISC Architecture

The RISC I instruction set contains a few simple operations (arithmetic, logical, and shift). These instructions operate on registers. Instruction, data, addresses and registers are all 32 bits long. RISC instructions fall in four categories: ALU, memory access, branch, and miscellaneous. The execution time is given by the time taken to read a register, perform an ALU operation, and store the result in a register. Register 0 always contains a 0. Load and store instructions move data between registers and memory. These instructions use two CPU cycles. Variations of memory-access instructions exist in order to accommodate sign-extended or zero-extended 8-bit, 16-bit and 32-bit data. Though absolute and register indirect addressing are not directly available, they may be synthesized using register 0. Branch instructions include CALL, RETURN, and conditional and unconditional jumps. The following instruction format is used:

opcode(7)	scc(1)	dest(5)	source1(5)	imm(1)	source2(13)

For register-to-register instructions, dest selects one of the 32 registers as destination of the result of the operation that is itself performed on registers source 1 and source2. If imm equals 0, the low-order 5 bits of source2 specify another register. If imm equals 1, then source2 is regarded as a sign-extended 13-bit constant. Since the frequency of integer constants is high, the immediate field has been made an option in every instruction. Also, Scc determines whether the condition codes are set. Memory-access instructions use source 1 to specify the index register and source2 to specify offset.

ii) Register Windows

The procedure-call statements take the maximum execution time. A RISC program has more call statements, since the complex instructions available in CISC are subroutines in RISC. The RISC register window scheme strives to make the call operation as fast as possible and also to reduce the number of accesses to data memory. The scheme works as follows.

Using procedures involve two groups of time-consuming operations, namely, saving or restoring registers on each call/return and passing parameters and results to and from the procedure. Statistics indicate that local variables are the most frequent operands.

This creates a need to support the allocation of locals in the registers. One available scheme is to provide multiple banks of registers on the chip to avoid saving and restoring of registers. Thus each procedure call results in a new set of registers being allocated for use by that procedure. The return alters a pointer that restores the old set. A similar scheme is adopted by RISC. However, there are some registers that are not saved or restored; these are called global registers. In addition, the sets of registers used by different processes are overlapped in order to allow parameters to be passed. In other machines, parameters are usually passed on the stack with the calling procedure using a register to point to the beginning of the parameters (and also to the end of the locals). Thus all references to parameters are indexed references to memory. In RISC I the set of window registers (r10 to r31) is divided into three parts. Registers r26 to r31 (HIGH) contain parameters passed from the calling procedure. Registers rl6 to r25 (LOCAL) are for local storage. Registers r10 to rl5 (LOW) are for local storage and for parameters to be passed to the called procedure. On each call, a new set of r10 to r31 registers is allocated. The LOW registers of the caller are required to become the HIGH registers of the called procedure. This is accomplished by having the hardware overlap the LOW registers of the calling frame with the HIGH registers of the called frame. Thus without actually moving the information, parameters are

transferred.

Multiple register banks require a mechanism to handle the case in which there are no free register banks available. RISC handles this problem with a separate register-overflow stack in memory and a stack pointer to it. Overflow and underflow are handled with a trap to a software routine that adjusts the stack. The final step in allocating variables in registers is handling the problem of pointers. RISC resolves this by giving addresses to the window registers. If a portion of the address space is reserved, we can determine with one comparison whether an address points to a register or to memory. Load and store are the only instructions that access memory and they take an extra cycle already. Hence this feature may be added without reducing the performance of the load and store instructions. This permits the use of straightforward computer technology and still leaves a large fraction of the variables in registers.

iii) Delayed Jump

A normal RISC I instruction cycle is long enough to execute the following sequence of operations:

1. Read a register.
2. Perform an ALU operation.
3. Store the result back into a register.

Performance is increased by prefetching the next instruction during the current instruction. To facilitate this, jumps are redefined such that they do not occur until after the following instruction. This is called delayed jump.

7.3 Design of the CPU

The CPU contains three elements: registers, the ALU (Arithmetic Logic Unit), and the control unit. These topics are discussed next. Verilog and VHDL descriptions along with simulation results of a typical CPU are provided in Appendices I and J respectively.

7.3.1 Register Design

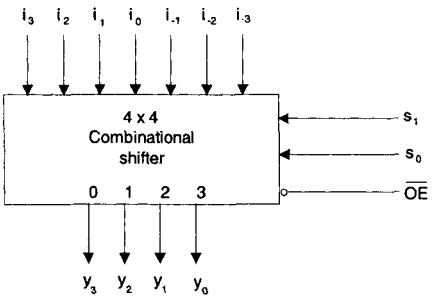
The concept of general-purpose and flag registers is provided in Chapters 5 and 6. The main purpose of a general-purpose register is to store address or data for an indefinite period of time. The computer can execute an instruction to retrieve the contents of this register when needed. A computer can also execute instructions to perform shift operations on the contents of a general-purpose register. This section includes combinational shifter design and the concepts associated with barrel shifters.

A high-speed shifter can be designed using combinational circuit components such as a multiplexer. The block diagram, internal organization, and truth table of a typical combinational shifter are shown in Figure 7.7. From the truth table, the following equations can be obtained:

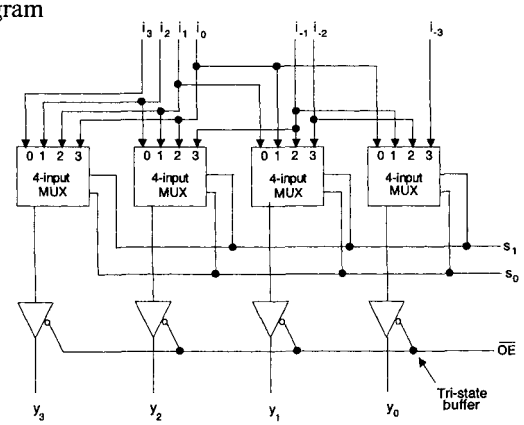
$$y_3 = \overline{s_1}\,\overline{s_0} i_3 + \overline{s_1} s_0 i_2 + s_1 \overline{s_0} i_1 + s_1 s_0 i_0$$
$$y_2 = \overline{s_1}\,\overline{s_0} i_2 + \overline{s_1} s_0 i_1 + s_1 \overline{s_0} i_0 + s_1 s_0 i_{-1}$$
$$y_1 = \overline{s_1}\,\overline{s_0} i_1 + \overline{s_1} s_0 i_0 + s_1 \overline{s_0} i_{-1} + s_1 s_0 i_{-2}$$
$$y_0 = \overline{s_1}\,\overline{s_0} i_0 + \overline{s_1} s_0 i_{-1} + s_1 \overline{s_0} i_{-2} + s_1 s_0 i_{-3}$$

The 4 × 4 shifter of Figure 7.7 can be expanded to obtain a system capable of rotating 16-bit data to the left by 0, 1, 2, or 3 positions, which is shown in Figure 7.8.

This design can be extended to obtain a more powerful shifter called the *barrel*



(a) Block Diagram



(b) Internal Schematic

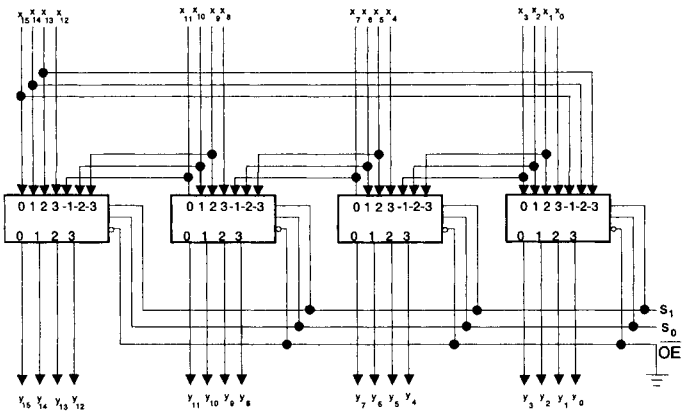
	Shift Count		Output				Comment
$\overline{OE}$	s_1	s_0	y_3	y_2	y_1	y_0	
1	X	X	Z	Z	Z	Z	Outputs are tristated
0	0	0	i_3	i_2	i_1	i_0	Pass (no shift)
0	0	1	i_2	i_1	i_0	i_{-1}	Left Shift once
0	1	0	i_1	i_0	i_{-1}	i_{-2}	Left shift twice
0	1	1	i_0	i_{-1}	i_{-2}	i_{-3}	Left shift three times

(c) Truth Table (X is don't care in the above)

FIGURE 7.7 4 × 4 combinational shifter

shifter. The shift is a cycle rotation, which means that the input binary information is shifted in one direction; the most significant bit is moved to the least significant position.

The block-diagram representation of a 16 × 16 barrel shifter is shown in Figure 7.9. This shifter is capable of rotating the given 16-bit data to the left by n positions, where $0 \leq n \leq 15$. Figure 7.9 shows the truth table representing the operation of the shifter. The barrel shifter is an on-chip component for typical 32-bit and 64-bit microprocessors.



(a) Logic Diagram

Shift Count		Output															
S_1	S_0	y_{15}	y_{14}	y_{13}	y_{12}	y_{11}	y_{10}	y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
0	1	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}
1	0	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}
1	1	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}

(b) Truth Table

FIGURE 7.8 Combinational shifter capable of rotating 16-bit data to the left by 0, 1, 2, or 3 positions

7.3.2 Adders

Addition is the basic arithmetic operation performed by an ALU. Other operations such as subtraction and multiplication can be obtained via addition. Thus, the time required to add two numbers plays an important role in determining the speed of the ALU.

The basic concepts of half-adder, full adder, and binary adder are discussed in Section 4.5.1. The following equations for the full-adder were obtained. Assume $x_i = x$, $y_i = y$, $c_i = z$, and $C_{i+1} = C$ in Table 4.6.

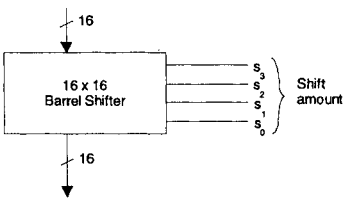
$$\text{Sum, } S_i = \bar{x}_i\bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i$$
$$= x_i \oplus y_i \oplus c_i$$

From Table 4.6,

$$\text{Carry, } C_{i+1} = \bar{x}_i y_i c_i + x_i \bar{y}_i c_i + x_i y_i \bar{c}_i + x_i y_i c_i$$
$$= (\bar{x}_i y_i c_i + x_i y_i c_i) + (x_i \bar{y}_i c_i + x_i y_i c_i) + (x_i y_i \bar{c}_i + x_i y_i c_i)$$
$$= y_i c_i + x_i c_i + x_i y_i$$

The logic diagrams for implementing these equations are given in Figure 7.10.

As has been made apparent by Figure 7.10, for generating C_{i+1} from c_i, two gate delays are required. To generate S_i from c_i, three gate delays are required because c_i must be inverted to obtain $\bar{c}_i$. Note that no inverters are required to get $\bar{x}_i$ or $\bar{y}_i$ from x_i or y_i, respectively, because the numbers to be added are usually stored in a register that is a collection of flip-flops. The flip-flop generates both normal and complemented outputs.



(a) Block Diagram of a 16 × 16 Barrel Shifter

Shift Count				Output															
S_3	S_2	S_1	S_0	y_{15}	y_{14}	y_{13}	y_{12}	y_{11}	y_{10}	y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
0	0	0	1	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}
0	0	1	0	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}
0	0	1	1	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}
0	1	0	0	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}
0	1	0	1	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}
0	1	1	0	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}
0	1	1	1	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9
1	0	0	0	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8
1	0	0	1	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7
1	0	1	0	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6
1	0	1	1	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5
1	1	0	0	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4
1	1	0	1	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3
1	1	1	0	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2
1	1	1	1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1

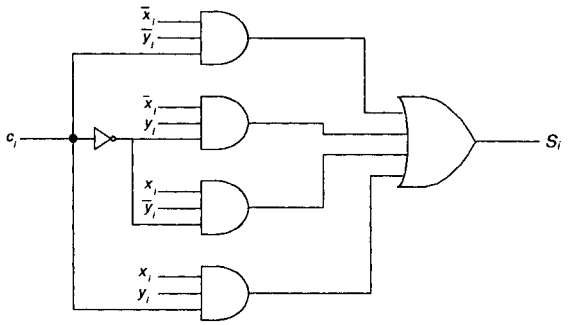
(b) Truth Table of the 16 × 16 Barrel Shifter

FIGURE 7.9 Barrel shifter

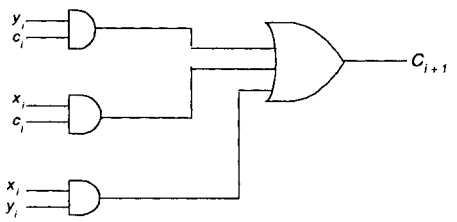
For the purpose of discussion, assume that the gate delay is Δ time units, and the actual value of Δ is decided by the technology. For example, if transistor translator logic (TTL) circuits are used, the value of Δ will be 10 ns.

By cascading n full adders, an n-bit binary adder capable of handling two n-bit operands (X and Y) can be designed. The implementation of a 4-bit ripple-carry or binary adder is shown in Figure 7.11. When two unsigned integers are added, the input carry, c_0, is always zero. The 4-bit adder is also called a "carry-propagate adder" (CPA), because the carry is propagated serially through each full adder. This hardware can be cascaded to obtain a 16-bit CPA, as shown in Figure 7.12; $c_0 = 0$ or 1 for multiprecision addition.

Although the design of an n-bit CPA is straightforward, the carry propagation time limits the speed of operation. For example, in the 16-bit CPA (see Figure 7.12), the

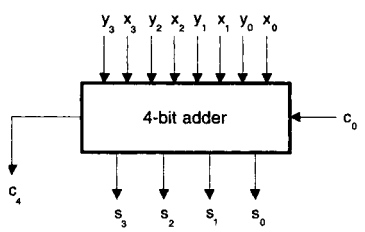


(a) Full adder

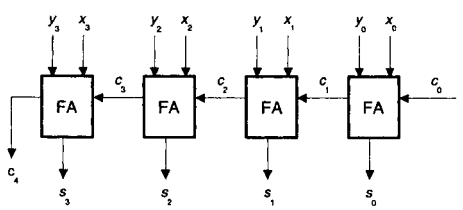


(b) Carry

FIGURE 7.10 Logic circuit of full adder



(a) Block Diagram of a 4-bit Ripple-Carry Adder



(b) Four 4-bit Full Adders are Cascaded to implement a 4-Bit Ripple-Carry Adder

FIGURE 7.11 Implementation of a 4-bit Ripple-Carry Adder

addition operation is completed only when the sum bits s_0 through s_{15} are available.

To generate s_{15}, c_{15} must be available. The generation of c_{15} depends on the availability of c_{14}, which must wait for c_{13} to become available. In the worst case, the carry process propagates through 15 full adders. Therefore, the worst-case add-time of the 16-bit CPA can be estimated as follows:

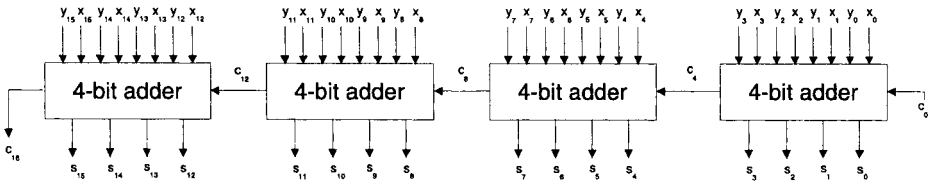


FIGURE 7.12 Implementation of a 16-bit adder using 4-Bit Adders as Building Blocks

Time taken for carry to propagate through 15 full adders (the delay involved in the path from c_0 to c_{15})	$= 15 * 2\ \Delta$
Time taken to generate s_{15} from c_{15}	$= 3\ \Delta$
Total	$= 33\ \Delta$

If $\Delta = 10$ ns, then the worst-case add-time of a 16-bit CPA is 330 ns. This delay is prohibitive for high-speed systems, in which the expected add-time is typically less than 100 ns, which makes it necessary to devise a new technique to increase the speed of operation by a factor of 3. One such technique is known as the "carry look-ahead." In this approach the extra hardware is used to generate each carry (c_i, $i > 0$) directly from c_0. To be more practical, consider the design of a 4-bit carry look-ahead adder (CLA). Let us see how this may be used to obtain a 16-bit adder that operates at a speed higher than the 16-bit CPA.

Recall that in a full adder for adding X_i, Y_i, and C_i, the output carry C_{i+1} is related to its carry input C_i, as follows:

$$C_{i+1} = X_iY_i + X_iC_i + Y_iC_i$$

The result can be rewritten as

$$C_{i+1} = G_i + P_iC_i$$

where $G_i = X_iY_i$ and $P_i = X_i + Y_i$

The function G_i is called the carry-generate function, because a carry is generated when $X_i = Y_i = 1$. If X_i or Y_i is a 1, then the input carry C_i is propagated to the next stage. For this reason, the function P_i is often referred to as the "carry-propagate" function. Using G_i and P_i, C_1, C_2, C_3, and C_4 can be expressed as follows:

$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1$$
$$C_3 = G_2 + P_2C_2$$
$$C_4 = G_3 + P_3C_3$$

All high-order carries can be generated in terms of C_0 as follows:

$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0)$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0)$$
$$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$
$$C_4 = G_3 + P_3C_3 = G_3 + P_3(G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0)$$
$$= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

$g_0 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$

$p_0 = P_3P_2P_1P_0$

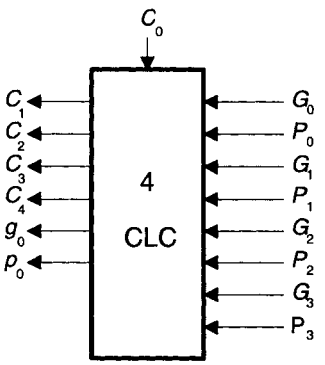


FIGURE 7.13 A Four-Stage Carry Look-ahead Circuit

Therefore C_1, C_2, C_3, and C_4 can be generated directly from C_0. For this reason, these equations are called "carry look-ahead equations," and the hardware that implements these equations is called a "4-stage look-ahead circuit" (4-CLC). The block diagram of such circuit is shown in Figure 7.13.

The following are some important points about this system:

- A 4-CLC can be implemented as a two-level AND-OR logic circuit (The first level consists of AND gates, whereas the second level includes OR gates).
- The outputs g_0 and p_0 are useful to obtain a higher-order look-ahead system.

To construct a 4-bit CLA, assume the existence of the basic adder cell shown in Figure 7.14. Using this basic cell and 4-bit CLC, the design of a 4-bit CLA can be completed as shown in Figure 7.15. Using this cell as a building block, a 16-bit adder can be designed as shown in Figure 7.16.

The worst-case add-time of this adder can be calculated as follows:

		Delay
For P_i, G_i generation		
from X_i, Y_i $(0 \le i \le 15)$	...	Δ
To generate C_4 from C_0	...	2Δ
To generate C_8 from C_4	...	2Δ
To generate C_{12} from C_8	...	2Δ
To generate C_{15} from C_{12}	...	2Δ
To generate S_{15} from C_{15}	...	3Δ
Total delay	...	12Δ

A graphical illustration of this calculation can be shown as follows:

$$\text{Data available} \xrightarrow{\Delta} G_iP_i \xrightarrow{2\Delta} C_4 \xrightarrow{2\Delta} C_8 \xrightarrow{2\Delta} C_{12} \xrightarrow{2\Delta} C_{15} \xrightarrow{3\Delta} S_{15}$$

From this calculation, it is apparent that the new 16-bit adder is faster than the 16-bit CPA by a factor of 3. In fact, this system can be speeded up further by employing another 4-bit CLC and eliminating the carry propagation between the 4-bit CLA blocks. For this purpose, the g_i and p_i outputs generated by the 4-bit CLA are used. This design task is left as an exercise to the reader.

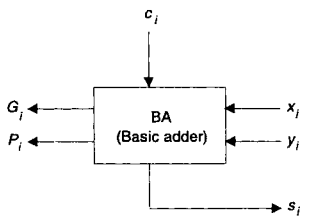


FIGURE 7.14 Basic CLA cell

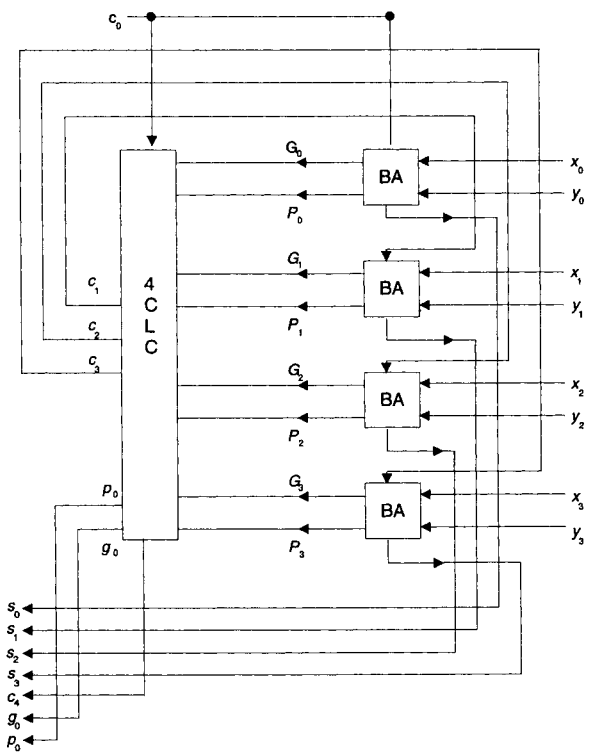


FIGURE 7.15 A 4-bit CLA

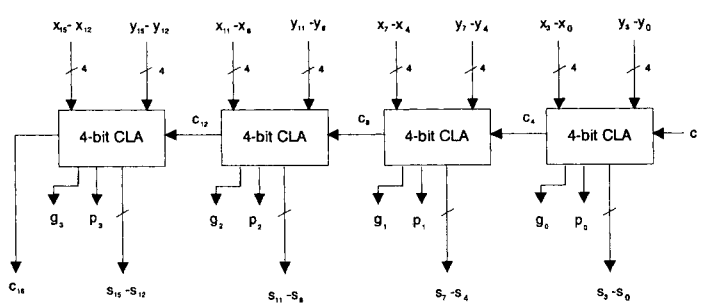


FIGURE 7.16 Design of a 16-bit adder using 4-bit CLAs

If there is a need to add more than 3 operands, a technique known as "carry-save addition" is used. To see its effectiveness, consider the following example:

44
28
32
79
63←Sum vector
12 ←Carry vector
183←Final answer

In this example, four decimal numbers are added. First, the unit digits are added, producing a sum of 3 and a carry digit of 2. Similarly, the tens digits are added, producing a sum digit of 6 and a carry digit of 1. Because there is no carry propagation from the unit digit to the tenth digit, these summations can be carried out in parallel to produce a sum vector of 63 and a carry vector of 12. When all operands are exhausted, the sum and the shifted carry vector are added in the conventional manner, which produces the final answer. Note that the carry is propagated only in the last step, which generates the final answer no matter how many operands are added. The concept is also referred to as "addition by deferred carry assimilation."

7.3.3 Addition, Subtraction, Multiplication and Division of unsigned and signed numbers

The procedure for addition and subtraction of two's complement signed binary numbers is straightforward. The procedure for adding unsigned numbers is discussed in Chapter 2. Also, addition of two 2's complement signed numbers was included in Chapter 2. Note that binary numbers represented in two's complement form contain both unsigned numbers (Most Significant Bit = 0) and signed numbers (Most Significant Bit = 1). The procedure for adding two 2's complement signed numbers using pencil and paper is provided below:

Add the two numbers along with the sign bits. Check the overflow bit (V) using $V = C_f \oplus C_p$ where C_f is the final carry and C_p is the previous carry. If V = 0, then the result of addition is correct. On the other hand, if V = 1, then the result is incorrect; one needs to increase the number of bits for each number, and repeat the addition operation until V = 0 to obtain the correct result.

Subtraction of two 2's complement signed binary numbers using pencil and paper can be performed as follows:

Take the 2's complement of subtrahend along with the sign bit and add it to the minuend . The result is correct if there is no overflow. The result is wrong if there is an overflow. In case of overflow, increase the number of bits for each number, repeat the subtraction operation until the overflow is zero to obtain the correct result. Note that if there is a final carry after performing the 2's complement subtraction, the result is positive. On the other hand, if there is no final carry after 2's complement subtraction, the result is negative.

Computers utilize common hardware to perform addition and subtraction operations for both unsigned and signed numbers. The instruction set of computers typically include the same ADD and SUBTRACT instructions for both unsigned and signed numbers. The interpretations of unsigned and signed ADD and SUBTRACT operations are performed by the programmer. For example, consider adding two 8-bit numbers, A and B ($A = FF_{16}$ and $B = FF_{16}$) using the ADD instruction by a computer as follows:

```
                1111 111← Intermediate carries
    FF16 = 1111 1111
+   FF16 = 1111 1111
    ---------------------------
Final carry →1  1111 1110 = FE16
```

When the above addition is interpreted as an unsigned operation by the programmer, the result will be
$A + B = FF_{16} + FF_{16} = 255_{10} + 255_{10} = 510_{10}$ which is FE_{16} with a carry as shown above. However, if the addition is interpreted as a signed operation, then, $A + B = FF_{16} + FF_{16} = (-1_{10}) + (-1_{10}) = -2_{10}$ which is FE_{16} as shown above, and the final carry must be discarded by the programmer. Similarly, the unsigned and signed subtraction can be interpreted by the programmer.

Typical 8-bit microprocessors, such as the Intel 8085 and Motorola 6809, do not include multiplication and division instructions due to limitations in the circuit densities that can be placed on the chip. Due to advances in semiconductor technology, 16-, 32-, and 64-bit microprocessors usually include multiplication and division algorithms in a ROM inside the chip. These algorithms typically utilize an ALU to carry out the operations. one can write a program that multiplies two numbers. Although this solution seems viable, the operational speed is unsatisfactory.

For application environments such as real-time digital filtering, in which the processor is expected to perform 32 to 64 eight-bit multiplication operations within 100 μsec (sampling frequency = 10 kHz), speed is an important factor. New device technologies such as BICMOS and HCMOS, allow manufacturers to pack millions of transistors in a chip. Consequently, state-of-the-art 32-bit microprocessors such as the Motorola 68060 (HCMOS) and Intel Pentium (BICMOS) designed using these technologies, have a larger instruction set than their predecessors, which includes multiplication and division instructions. In this section, multiplier design principles are discussed. Two unsigned integers can be multiplied using repeated addition as mentioned in Chapter 2. Also, they can be multiplied in the same way as two decimal numbers are multiplied by paper and pencil method. Consider the multiplication of two unsigned integers, where the multiplier Q = 15 and the multiplicand is M = 14, as illustrated:

```
M ——————→ 1110 ——— Multiplicand (14₁₀)
Q ——————→ 1111 ——— Multiplier (15₁₀)

          ————————————
           1110 ←——
           1110 ←——
           1110 ←——— Partial products
          1110 ←———
          ————————————
P ——————→ 11010010 ←——— Final product
```

In the paper and pencil algorithm, shifted versions of multiplicands are added.

$$
\begin{array}{cccc}
 & & m_3 & m_2 & m_1 & m_0 \\
 & & q_3 & q_2 & q_1 & q_0 \\
\hline
 & & m_3q_0 & m_2q_0 & m_1q_0 & m_0q_0 \leftarrow \text{Partial product } PR_0 \\
 & m_3q_1 & m_2q_1 & m_1q_1 & m_0q_1 \leftarrow \text{Partial product } PR_1 \\
 & m_3q_2 & m_2q_2 & m_1q_2 & m_0q_2 \leftarrow \text{Partial product } PR_2 \\
m_3q_3 & m_2q_3 & m_1q_3 & m_0q_3 \leftarrow \text{Partial product } PR_3 \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

FIGURE 7.17 Generalized Version of the Multiplication of Two 4-bit Numbers Using the Paper and Pencil Algorithm

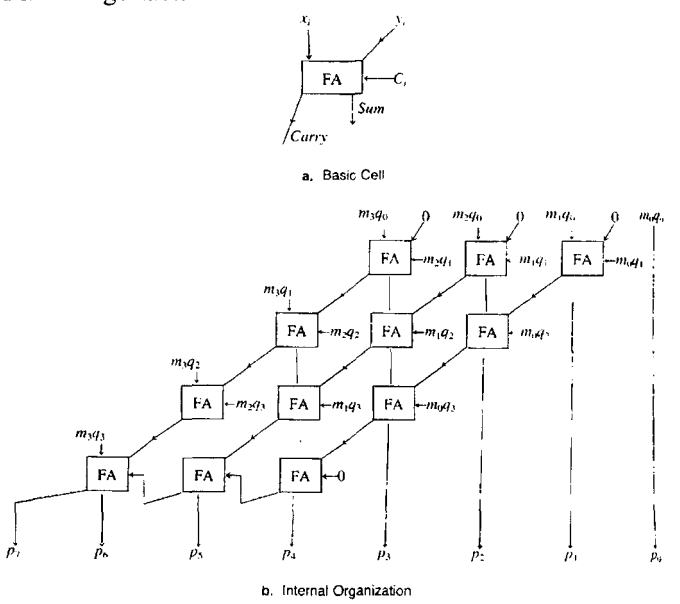


FIGURE 7.18 4 ×4 Array Multiplier

This procedure can be implemented by using combinational circuit elements such as AND gates and FULL adders. Generally, a 4-bit unsigned multiplier Q and a 4-bit unsigned multiplicand M can be written as M: $m_3\ m_2\ m_1\ m_0$ and Q: $q_3\ q_2\ q_1\ q_0$.The process of generating the partial products and the final product can also be generalized as shown in

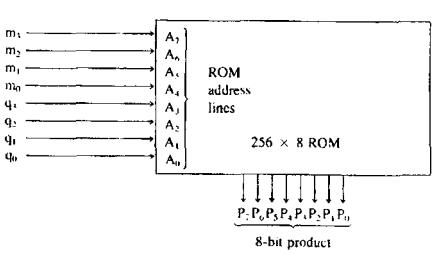


FIGURE 7.19 ROM-based 4x4 Multiplier

Figure 7.17. Each cross-product term ($m_i\, q_j$) in this figure can be generated using an AND gate. This requires 16 AND gates to generate all cross-product terms that are summed by full adder arrays, as shown in Figure 7.18.

Consider the generation of p_2 in Figure 7.18(b). From Figure 7.17, p_2 is the sum of m_2q_0, m_1q_1 and m_0q_2. The sum of these three elements is obtained by using two full adders. (See column for p_2 in Figure 7.18). The top full-adder in this column generates the sum $m_2q_0 + m_1q_1$. This sum is then added to m_0q_2 by the bottom full-adder along with any carry from the previous full-adder for p_1.

The time required to complete the multiplication can be estimated by considering the longest carry propagation path comprising of the rightmost diagonal (which includes the full-adder for p_1 and the bottom full-adders for p_2 and p_3), and the last row (which includes the full-adder for p_6 and the bottom full-adders for p_4 and p_5). The time taken to multiply two n-bit numbers can be expressed as follows:

$$T(n) + \Delta_{AND\ gate} + (n - 1)\,\Delta_{carry\ propagation} + (n - 1)\,\Delta_{carry\ propagation}$$

In this equation, all cross-product terms m_iq_i can be generated simultaneously by an array of AND gates. Therefore, only one AND gate delay is included in the equation. Also, the rightmost diagonal and the bottom row contain (n - 1) full-adders each for the n × n multiplier.

Assuming that $\Delta_{AND\ gate} = \Delta_{carry\ propagation} = 2gate\ delays = 2\Delta$, the preceding expression can be simplified as shown:

$T(n) = 2\Delta + (2n - 2)2\Delta = (4n - 2)\Delta$.

The array multiplier that has been considered so far is known as Braun's multiplier. The hardware is often called a nonadditive multiplier (NM), since it does not include any additive inputs. An additive multiplier (AM) includes an extra input R; it computes products of the form

$P = M * Q + R$

This type of multiplier is useful in computing the sum of products of the form $\sum XiYi$.

Both an NM and an AM are available as standard 1C blocks. Since these systems require more components, they are available only to handle 4- or 8-bit operands.

Alternatively, the same 4x4 NM discussed earlier can be obtained using a 256 × 8 ROM as shown in Figure 7.19.

It can be seen that a given MQ pair defines a ROM address, where the corresponding 8-bit product is held. The ROM approach can be used for small-scale multipliers because:

- The technological advancements allow the manufacturers to produce low-cost ROMs.
- The design effort is minimum.

In case of large multipliers, ROM implementation is unfeasible, since large-size ROMs are required. For example, in order to implement an 8 × 8 multiplier, a 2^{16} × 16 ROM is required. If the required 8 × 8 product is decomposed into a linear combination of four 4x4 products, an 8 × 8 multiplier can be implemented using four 256 × 8 ROMs and a few 4-bit parallel adders. However, PLDs can be used to accomplish this.

Signed multiplication can be performed using various algorithms. A simple algorithm follows.

In the case of signed numbers, there are three possibilities:

1. M and Q are in sign-magnitude form.
2. M and Q are in ones complement form.
3. M and Q are in twos complement form.

For the first case, perform unsigned multiplication of the magnitudes without the sign

bits. The sign bit of the product is determined as $M_n \oplus Q_n$, where M_n and Q_n are the most significant bits (sign bits) of the multiplicand (M) and the multiplier (Q), respectively. For the second case, proceed as follows:

Step 1: If $M_n = 1$, then compute the ones complement of M.
Step 2: If $Q_n = 1$, then compute the ones complement of Q.
Step 3: Multiply the $n - 1$ bits of the multiplier and the multiplicand.
Step 4: $S_n = M_n \oplus Q_n$
Step 5: If $S_n = 1$, then compute the ones complement of the result obtained in Step 3.

Whenever the ones complement of a negative number (sign bit = 1) is taken, the sign is reversed. Hence, with respect to the multiplier, the inputs are always a positive quantity. When the sign of the bit is negative, however ($M_n \oplus Q_n = 1$), the result must be presented in the ones complement form. This is why the ones complement of the product found by the unsigned multiplier is computed. When M and Q are in twos complement form, the same procedure is repeated, with the exception that the twos complement must be determined when $Q_n = 1$, $M_n = 1$, or $M_n \oplus Q_n = 1$. Consider M and Q as twos complement numbers. Suppose $M = 1100_2$ and $Q = 0111_2$. Because $M_n = 1$, take the twos complement of $M = 0100_2$; because $Q_n = 0$, do not change Q. Multiply 0111_2 and 0100_2 using the unsigned multiplication method discussed before. The product is 00011100_2. The sign of the product $S_n = M_n \oplus Q_n = 1 \oplus 0 = 1$. Hence, take the twos complement of the product 00011100_2 to obtain 11100100_2, which is the final answer: -28_{10}.

As mentioned in Chapter 2, unsigned division can be performed using repeated subtraction. However, the general equation for division can be used for signed division. Note that the general equation for division is *Dividend = Quotient * Divisor + Remainder*. For example, consider dividend = – 9, divisor = 2. Three possible solutions are shown below:

(a) $-9 = -4 * 2 - 1$, Quotient = – 4, Remainder = – 1.
(b) $-9 = -5 * 2 + 1$, Quotient = – 5, Remainder = +1.
(c) $-9 = -6 * 2 + 3$, Quotient = – 6, Remainder = +3.

However, the correct answer is shown in (a) in which, Quotient = – 4 and Remainder = – 1. Hence, for signed division, the sign of the remainder is the same as the sign of the dividend, unless the remainder is zero. Typical microprocessors such as Motorola 68XXX follow this convention.

7.3.4 ALU Design

Functionally, an ALU can be divided up into two segments: the arithmetic unit and the logic unit. The arithmetic unit performs typical arithmetic operations such as addition, subtraction, and increment or decrement by 1. Usually, the operands involved may be signed or unsigned integers. In some cases, however, an arithmetic unit must handle 4-bit binary-coded decimal (BCD) numbers and floating-point numbers. Therefore, this unit must include the circuitry necessary to manipulate these data types. As the name implies, the logic unit contains hardware elements that perform typical operations such as Boolean NOT and OR. In this section, the design of a simple ALU using typical combinational elements such as gates, multiplexers, and a 4-bit parallel adder is discussed. For this approach, an arithmetic unit and a logic unit are first designed separately; then they are combined to obtain an ALU.

For the first step, a two-function arithmetic unit, as shown in Figure 7.20 is designed. The key element of this system is a 4-bit parallel adder. The multiplexers select

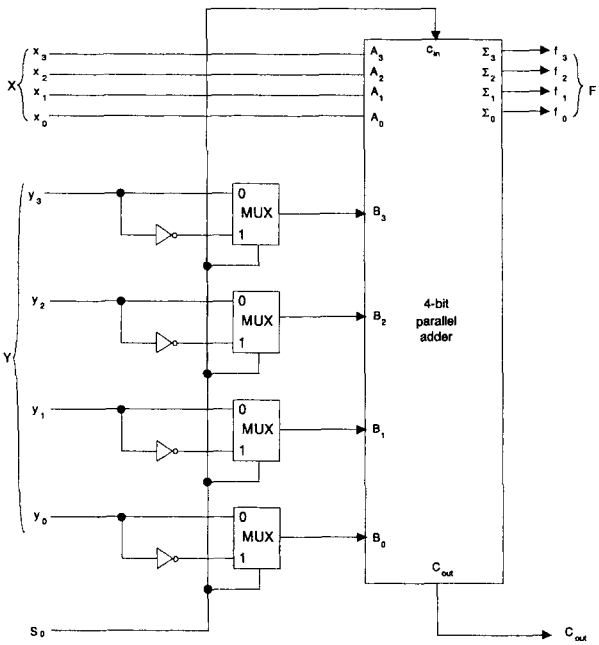


FIGURE 7.20 Organization of an arithmetic unit

either Y or $\overline{Y}$ for the 3-input of the parallel adder. In particular, if $s_0 = 0$, then $B = Y$; otherwise $B = \overline{Y}$. Because the selection input (s_0) also controls the input carry (c_{in}), the following results:

$$\begin{aligned} \text{If } s_0 = 0 \text{ then } F &= X \text{ plus } Y \\ \text{else } F &= X \text{ plus } \overline{Y} \text{ plus } 1 \\ &= X \text{ minus } Y \end{aligned}$$

This arithmetic unit generates addition and subtraction operations. For the second step, let us design a two-function logic unit; this is shown in Figure 7.21. From Figure 7.21 it can be seen that when $s_0 = 0$, the output $G = X$ AND Y; otherwise the output $G = X \oplus Y$. Note that from these two Boolean operations, other operations such as NOT and OR can be derived by the following Boolean identities:

$1 \oplus x = \overline{x}$

x OR $y = x \oplus y \oplus xy$

Therefore, NOT and OR operations can be obtained by using additional hardware and the circuit of Figure 7.21. The outputs generated by the arithmetic and logic units can be combined by using a set of multiplexers, as shown in Figure 7.22. From this organization it can be seen that when the select line $s_1 = 1$, the multiplexers select outputs generated by the logic unit; otherwise, the outputs of the arithmetic unit are selected.

More commonly, the select line, s_1, is referred to as the *mode input* because it selects the desired mode of operation (arithmetic or logic). A complete block diagram schematic of this ALU is shown in Figure 7.23. The truth table illustrating the operation of this ALU is shown in Figure 7.24. This table shows that this ALU is capable of performing 2 arithmetic and 2 logic operations on the 4-bit operands X and Y.

The rapid growth in IC technology permitted the manufacturers to produce an ALU as an MSI block. Such systems implement many operations, and their use as a system

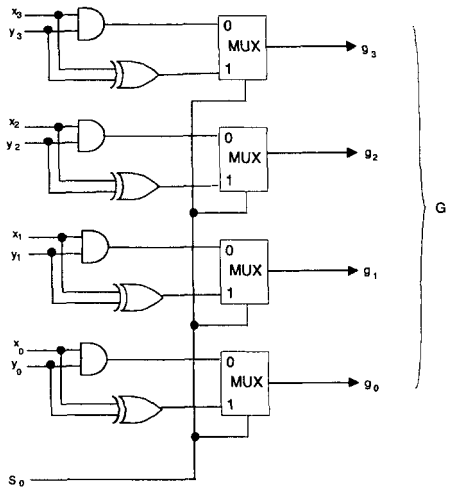


FIGURE 7.21 Organization of a 4-bit two-function logic unit

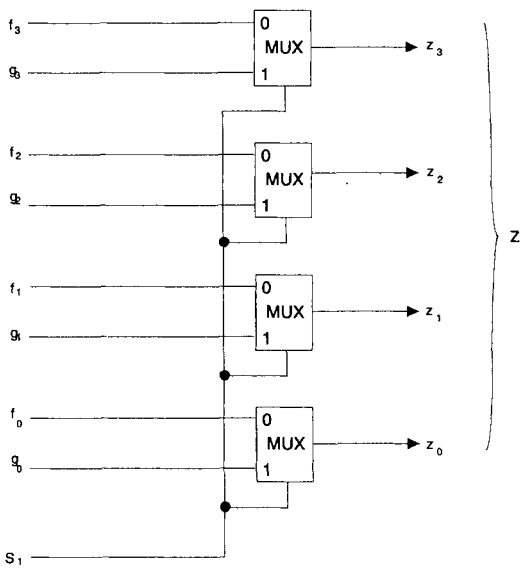


FIGURE 7.22 Combining the outputs generated by the arithmetic and logic units

component reduces the hardware cost, board space, debugging effort, and failure rate. Usually, each MSI ALU chip is designed as a 4-bit slice. However, a designer can easily interconnect n such chips to get a $4n$-bit ALU. Some popular 4-bit ALU chips are the 74381 and 74181. The 74381 ALU performs 3 arithmetic and 2 miscellaneous operations on 4-bit operands. The 74181 ALU performs 16 arithmetic and 16 Boolean operations on two 4-bit operands, using either active high or active low data. A complete description and operational characteristics of these devices may be found in the data books.

Typical 8-bit microprocessors, such as the Intel 8085 and Motorola 6809, do not include multiplication and division instructions due to limitations in the circuit densities that can be placed on the chip. Due to advanced semiconductor technology, 16-, 32-, and 64-bit

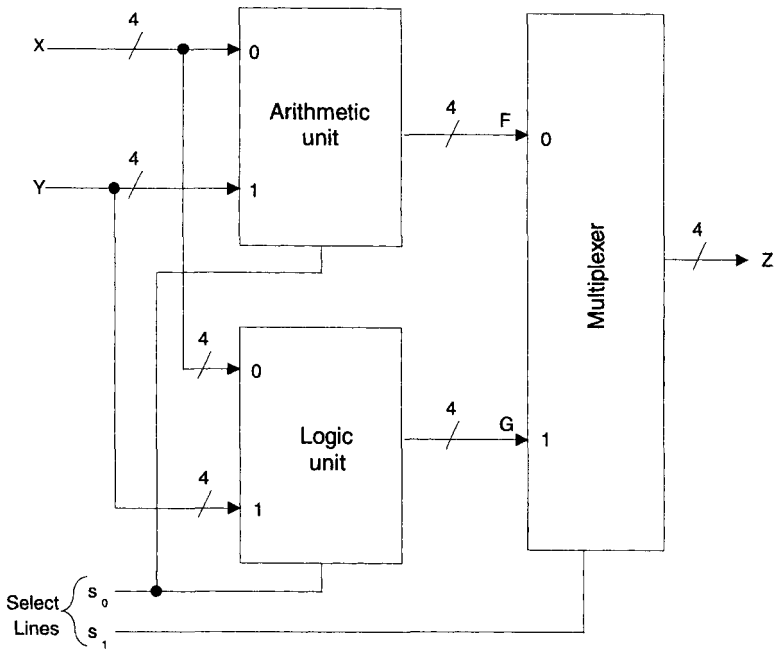


FIGURE 7.23 Schematic representation of the four functions

Select Lines		Output Z	Comment
s_1	s_0		
0	0	X plus Y	Addition
0	1	X plus $\overline{Y}$ Plus 1	2's Complement subtraction
1	0	X ∧ Y	Boolean AND
1	1	X ⊕ Y	Exclusive-OR

FIGURE 7.24 Truth table controlling the operations of the ALU of Figure 7.23

microprocessors usually include multiplication and division algorithms in a ROM inside the chip. These algorithms typically utilize an ALU to carry out the operations. Verilog and VHDL descriptions along with simulation results of typical ALU's are included in Appendices I and J respectively.

7.3.5 Design of the Control Unit

The main purpose of the control unit is to translate or decode instructions and generate appropriate enable signals to accomplish the desired operation. Based on the contents of the instruction register, the control unit sends the selected data items to the appropriate processing hardware at the right time. The control unit drives the associated processing hardware by generating a set of signals that are synchronized with a master clock.

The control unit performs two basic operations: instruction interpretation and instruction sequencing. In the interpretation phase, the control unit reads (fetches) an instruction from the memory addressed by the contents of the program counter into

the instruction register. The control unit inputs the contents of the instruction register. It recognizes the instruction type, obtains the necessary operands, and routes them to the appropriate functional units of the execution unit (registers and ALU). The control unit then issues the necessary signals to the execution unit to perform the desired operation and routes the results to the specified destination.

In the sequencing phase, the control unit generates the address of the next instruction to be executed and loads it into the program counter. To design a control unit, one must be familiar with some basic concepts such as register transfer operations, types of bus structures inside the control unit, and generation of timing signals. These are described in the next section.

There are two methods for designing a control unit: hardwired control and microprogrammed control. In the hardwired approach, synchronous sequential circuit design procedures are used in designing the control unit. Note that a control unit is a clocked sequential circuit. The name "hardwired control" evolved from the fact that the final circuit is built by physically connecting the components such as gates and flip-flops. In the microprogrammed approach, on the other hand, all control functions are stored in a ROM inside the control unit. This memory is called the "control memory." RAMs and PALs are also used to implement the control memory. The words in this memory are called "control words," and they specify the control functions to be performed by the control unit. The control words are fetched from the control memory and the bits are routed to appropriate functional units to enable various gates. An instruction is thus executed. Design of control units using microprogramming (sometimes called *firmware* to distinguish it from hardwired control) is more expensive than using hardwired controls. To execute an instruction, the contents of the control memory in microprogrammed control must be read, which reduces the overall speed of the control unit.The most important advantage of microprogramming is its flexibility; many additions and changes are made by simply changing the microprogram in the control memory. A small change in the hardwired approach may lead to redesigning the entire system.

There are two types of microprocessor architectures: CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer). CISC microprocessors contain a large number of instructions and many addressing modes while RISC microprocessors include a simple instruction set with a few addressing modes. Almost all computations can be obtained from a few simple operations. RISC basically supports a small set of commonly used instructions which are executed at a fast clock rate compared to CISC which contains a large instruction set (some of which are rarely used) executed at a slower clock rate. In order to implement fetch /execute cycle for supporting a large instruction set for CISC, the clock is typically slower. In CISC, most instructions can access memory while RISC contains mostly load/store instructions. The complex instruction set of CISC requires a complex control unit, thus requiring microprogrammed implementation. RISC utilizes hardwired control which is faster. CISC is more difficult to pipeline while RISC provides more efficient pipelining. An advantage of CISC over RISC is that complex programs require fewer instructions in CISC with a fewer fetch cycles while the RISC requires a large number of instructions to accomplish the same task with several fetch cycles. However, RISC can significantly improve its performance with a faster clock, more efficient pipelining and compiler optimization. PowerPC and Intel 80XXX utilize RISC and CISC architectures respectively. Intel Pentium family, on the other hand, utilizes a combination of RISC and CISC architectures for providing high performance. The Pentium uses RISC (hardwired control) to implement efficient pipelining for simple

$$R_1 \leftarrow R_0$$

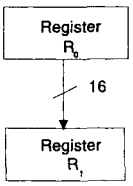


FIGURE 7.25 16-Bit register transfer from R_0 to R_1

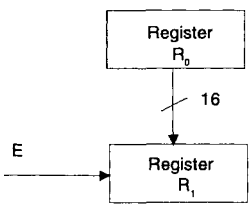


FIGURE 7.26 An enable input controlling register transfer

instructions. CISC (microprogrammed control) for complex instructions is utilized by the Pentium to provide upward compatibility with the Intel 8086/80X86 family.

Basic Concepts

Register transfer notation is the fundamental concept associated with the control unit design. For example, consider the register transfer operation of Figure 7.25. The contents of 16-bit register R_0 are transferred to 16-bit register R_1 as described by the following notation:

$$R_1 \leftarrow R_0$$

The symbol $\leftarrow$ is called the transfer operator. However, this notation does not indicate the number of bits to be transferred. A declaration statement specifying the size of each register is used for the purpose:

```
Declare registers R0 [16], R1 [16]
```

The register transfer notation can also be used to move a specific bit from one register to a particular bit position in another. For example, the statement

$$R_1[1] \leftarrow R_0[14]$$

means that bit 14 of register R_0 is moved to bit 1 of register R_1.

An enable signal usually controls transfer of data from one register to another. For example, consider Figure 7.26. In the figure, the 16-bit contents of register R_0 are transferred to register R_1 if the enable input E is HIGH; otherwise the contents of R_0 and R_1 remain the same. Such a conditional transfer can be represented as

$$E\colon R_1 \leftarrow R_0$$

Figure 7.27 shows a hardware implementation of transfer of each bit of R_0 and R_1. The enable input may sometimes be a function of more than one variable. For example, consider the following statement involving three 16-bit registers: If $R_0 < R_1$ and $R_2[1] = 1$ then $R_1 \leftarrow R_0$.

The condition $R_0 < R_1$ can be determined by an 8-bit comparator such that the output y of the comparator goes to 0 if $R_0 < R_1$. The conditional transfer can then be

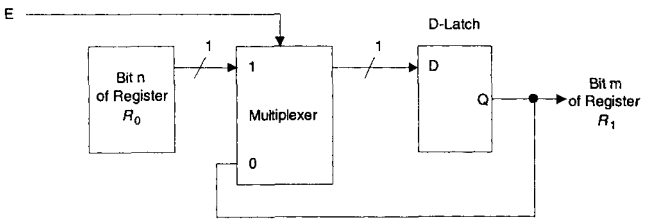


FIGURE 7.27 Hardware for each bit transfer from R_0 to R_1

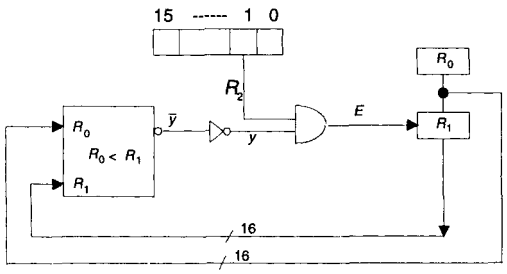


FIGURE 7.28 Hardware implementation E: $R_1 \leftarrow R_0$ where $E = y \cdot R_2$ [1]

```
Declare registers R[8],M[8],Q[8];
Declare buses inbus[8],outbus[8];
Start:    R ← 0, M ← inbus;          Clear register R to 0 and move
                                     multiplicand
          Q ← inbus;                 Transfer multiplier
Loop:     R ← R + M, Q ← Q-1;        Add multiplicand
          If Q < > 0 then go to loop; repeat if Q≠ 0
          Outbus ← R;
  Halt:   Go to Halt;
```

FIGURE 7.29 Register transfer description of 8 × 8 unsigned multiplication (Assume 8-bit result)

expressed as follows: E: $R_1 \leftarrow R_0$ where $E = y \cdot R_2$ [1]. Figure 7.28 depicts the hardware implementation.

A number of wires called "buses" are normally used to transfer data in and out of a digital processing system. Typically, there will be a pair of buses ("inbuses" and "outbuses") inside the CPU to transfer data from the external devises into the processing section and vice versa. Like the registers, these buses are also represented using register transfer notations and declaration statements. For example, "Declare inbus [16] and outbus [16]" indicate that the digital system contains two 16-bit wide data buses (inbus and outbus). $R_0 \leftarrow$ inbus means that the data on the inbus is transferred into register R_0 when the next clock arrives. An equate (=) symbol can also be used in place of $\leftarrow$. For example, "outbus = R_1 [15:8]" means that the high-order 8 bits of the 16-bit register R_1 are made available on the outbus for one clock period. An algorithm implemented by a digital system can be described by using a set of register transfer notations and typical control structures such as if-then and go to. For example, consider the description shown in Figure 7.29 for

multiplying two 8-bit unsigned numbers (Multiplication of an 8-bit unsigned multiplier by an 8-bit multiplicand) using repeated addition.

The hardware components for the preceding description include an 8-bit inbus, an 8-bit outbus, an 8-bit parallel adder, and three 8-bit registers, R, M, and Q. This hardware performs unsigned multiplication by repeated addition. This is equivalent to unsigned multiplication performed by assembly language instruction.

A distinguishing feature of this description is to describe concurrent operations. For example, the operations $R \leftarrow 0$ and $M \leftarrow$ inbus can be performed simultaneously. As a general rule, a comma is inserted between operations that can be executed concurrently. On the other hand, a semicolon between two transfer operations indicates that they must be performed serially. This restriction is primarily due to the data path provided in the hardware. For example, in the description, because there is only one input bus, the operations $M \leftarrow$ inbus and $Q \leftarrow$ inbus cannot be performed simultaneously. Rather, these two operations must be carried out serially. However, one of these operations may be overlapped with the operation $R \leftarrow 0$ because the operation does not use the inbus. The description also includes labels and comments to improve readability of the task description. Operations such as $R \leftarrow 0$ and M $\leftarrow$ inbus are called "micro-operations", because they can be completed in one clock cycle. In general, a computer instruction can be expressed as a sequence of micro-operations.

The rate at which a microprocessor completes operations such as $R \leftarrow R + M$ is determined by its bus structure inside the microprocessor chip. The cost of the microprocessor increases with the complexity of the bus structure. Three types of bus structures are typically used: single-bus, two-bus, and three-bus architectures.

The simplest of all bus structures is the single-bus organization shown in Figure 7.30. At any time, data may be transferred between any two registers or between a register and the ALU. If the ALU requires two operands such as in response to an ADD instruction, the operands can only be transferred one at a time. In single-bus architecture, the bus must be multiplexed among various operands. Also, the ALU must have buffer registers to hold the transferred operand.

In Figure 7.30, an add operation such as $R_0 \leftarrow R_1 + R_2$ is completed in three clock cycles as follows:

First clock cycle: The contents of R_1 are moved to buffer register B_1 of the ALU.
Second clock cycle: The contents of R_2 are moved to buffer register B_2 of the ALU.
Third clock cycle: The sum generated by the ALU is loaded into R_0.

A single-bus structure slows down the speed of instruction execution even though data may already be in the microprocessor registers. The instruction's execution time is longer if the operands are in memory; two clock cycles may be required to retrieve the operands into the microprocessor registers from external memory.

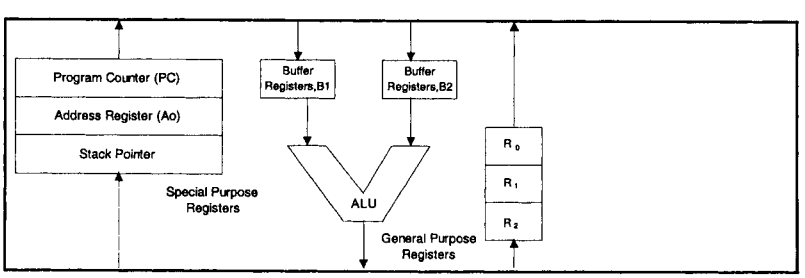


FIGURE 7.30 Single-bus architecture

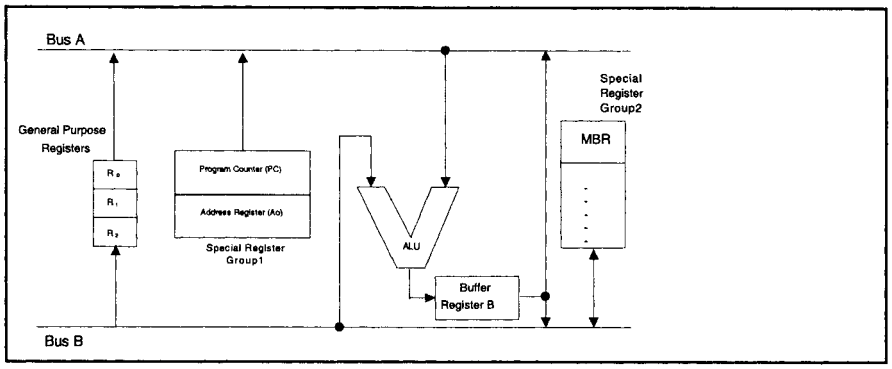


FIGURE 7.31 Two-bus architecture

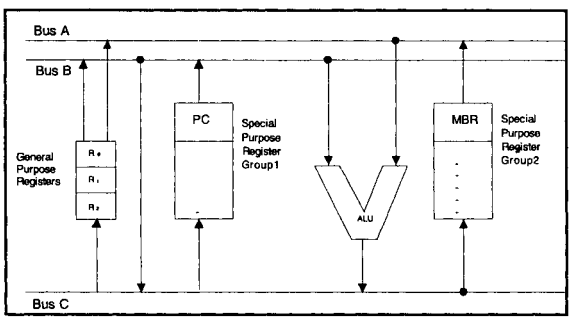


FIGURE 7.32 Three-bus architecture

To execute an instruction such as ADD between two operands already in register, the control logic in a single-bus structure must follow a three-step sequence. Each step represents a control state. Therefore, a single-bus architecture requires a large number of states in the control logic, so more hardware may be needed to design the control unit. Because all data transfers take place through the same bus one at a time, the design effort to build the control logic is greatly reduced.

Next, consider a two-bus architecture, shown in Figure 7.31. All general-purpose registers are connected to both buses (bus *A* and bus *B*) to form a two-bus architecture. The two operands required by the ALU are, therefore, routed in one clock cycle. Instruction execution is faster because the ALU does not have to wait for the second operand, unlike the single-bus architecture. The information on a bus may be from a general-purpose register or a special-purpose register. In this arrangement, special-purpose registers are often divided into two groups. Each group is connected to one of the buses. Data from two special-purpose registers of the same group cannot be transferred to the ALU at the same time.

In the two-bus architecture, the contents of the program counter are always transferred to the right input of the ALU because it is connected to bus *A*. Similarly, the contents of the special register MBR (memory buffer register, to hold up data retrieved from external memory) are always transferred to the left input of the ALU because it is connected to bus *B*.

In Figure 7.31, an add operation such as $R_0 \leftarrow R_1 + R_2$ is completed in two clock cycles as follows:

First clock cycle: The contents of R_1 and R_2 are moved to the inputs of ALU. The ALU then generates the sum in the output register.

Second clock cycle: The sum from the output register is routed to R_0.

The performance of a two-bus architecture can be improved by adding a third bus (bus *C*), at the output of the ALU. Figure 7.32 depicts a typical three-bus architecture. The three-bus architecture perform the addition operation $R_0 \leftarrow R_1 + R_2$ in one cycle as follows:

First cycle: The contents of R_1 and R_2 are moved to the inputs of the ALU via bus A and bus *B* respectively. The sum generated by the ALU is then transferred to R_0 via bus *C*.

The addition of the third bus will increase the system cost and also the complexity of the control unit design.

Note that the bus architectures described so far are inside the microprocessor chip. On the other hand, the system bus connecting the microprocessor, memory, and I/O are external to the microprocessor.

Another important concept required in the design of a control unit is the generation of timing signals. One of the main tasks of a control unit is to properly sequence a set of operations such as a sequence of *n* consecutive clock pulses. To carry out an operation, timing signals are generated from a master clock. Figure 7.33 shows the input clock pulse and the four timing signals T_0, T_1, T_2, and T_3. A ring counter (described in Chapter 5) can be used to generate these timing signals. To carry out an operation P_i at the *i*th clock pulse, a control unit must count the clock pulses and produce a timing signal T_i.

Hardwired Control Design

The steps involved in hardwired control design are summarized as follows:

1. Derive a flowchart from the problem definition and validate the algorithm by using trial data.
2. Obtain a register transfer description of the algorithm from the flowchart.
3. Specify a processing hardware along with various components.
4. Complete the design of the processing section by establishing the necessary control inputs.
5. Determine a block diagram of the controller.

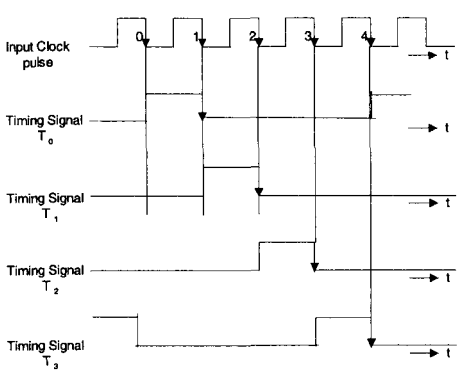


FIGURE 7.33 Timing signals

6. Obtain the state diagram of the controller.
7. Specify the characteristic of the hardware for generating the required timing signals used in the controller.
8. Draw the logic circuit of the controller.

The following example is provided to illustrate the concepts associated with implementation of a typical instruction in a control unit using hardwired control. The unsigned multiplication by repeated addition discussed earlier is used for this purpose. A 4-

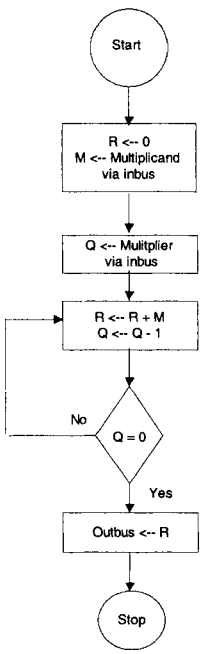


FIGURE 7.34 Flowchart for 4-bit × 4-bit multiplication

	R	M	Q
Initialization	0 0 0 0	0 1 0 0	0 0 1 1
Iteration 1 R <-- R + M Q <-- Q - 1	0 1 0 0	0 1 0 0	0 0 1 0
Iteration 2 R <-- R + M Q <-- Q - 1	1 0 0 0	0 1 0 0	0 0 0 1
Iteration 3 R <-- R + M Q <-- Q - 1	1 1 0 0	0 1 0 0	0 0 0 0

Product $=12_{10}$

FIGURE 7.35 Verification of the unsigned multiplication algorithm

bit by 4-bit unsigned multiplication will be considered. Assume the result of multiplication is 4 bits.

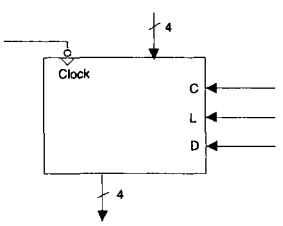
Step 1: Derive a flowchart from the problem definition and then validate the algorithm using trial data.

Figure 7.34 shows the flowchart. In the figure, M and Q are two 4-bit registers containing the unsigned multiplicand and unsigned multiplier respectively. Assume that the result of multiplication is 4-bit wide. The 4-bit result of the multiplication called the "product" will be stored in the 4-bit register, R. The contents of R are then output to the outbus.

The flowchart in Figure 7.34 is similar to an ASM chart and provides a hardware description of the algorithm. The sequence of events and their timing relationships are described in the flowchart. For example, the operations, $R \leftarrow 0$ and $M \leftarrow$ multiplicand shown in the same block are executed simultaneously. Note that $M \leftarrow$ multiplicand via inbus and $Q \leftarrow$ multiplier via inbus must be performed serially because both operations use a single input bus for loading data. These operations are, therefore, shown in different

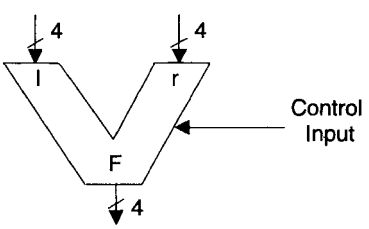
```
Start: R ← 0, M ← inbus;          Clear Register to 0 and move multiplicand
       Q ← inbus;                 Transfer Multiplier
Loop:  R ← R + M, Q ← Q -1;       Perform addition, decrement counter
       If Q < > 0 then goto Loop; Repeat if Q ≠ 0
       outbus ← R;
Halt:  Go to Halt;
```

FIGURE 7.36 Register transfer description 4-bit × 4-bit unsigned multiplication



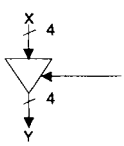
C	L	D	Clock	Action
1	0	0	↓	Clear
0	1	0	↓	Load External data
0	0	1	↓	Decrement by one
0	0	0	↓	No Change

(a) General Purpose Register



Control Input	F
1	I + r
0	No operation

(b) 4-bit Adder



Control Input	Y
1	X
0	High Impedence State

(c) Tristate Buffer

FIGURE 7.37 Components of the processing section of 4-bit by 4-bit unsigned multiplication

blocks. Because $R \leftarrow 0$ does not use the inbus, this operation is overlapped, in our case, with initializing of M via the inbus. This simultaneous operation is indicated by placing them in the same block.

The algorithm will now be verified by means of a numerical example as shown in Figure 7.35. Suppose $M = 0100_2 = 4_{10}$ and $Q = 0011_2 = 3_{10}$; then R = product = $1100_2 = 12_{10}$

Step 2: Obtain a register transfer description of the algorithm from the flowchart. Figure 7.36 shows the description of the algorithm.

Step 3: Specify a processing hardware along with various components.

The processing section contains three main components:

- General-purpose registers
- 4-bit adder
- Tristate buffer

Figure 7.37 shows these components. The general-purpose register is a trailing edge-triggered device.

Three operations (clear, parallel load, and decrement) can be performed by applying the appropriate inputs at C, L, and D. All these operations are synchronized at the trailing (high to low) edge of the clock pulse.

The 4-bit adder can be implemented using 4-bit adder circuits. The tristate buffer is used to control data transfer to the outbus.

Step 4: Complete the design of the processing section by establishing the necessary control inputs.

Figure 7.38 shows the detailed logic diagram of the processing section, along with the control inputs.

Step 5: Determine a block diagram of the controller. Figure 7.39 shows the block diagram.

The controller has three inputs and seven outputs. The Reset input is an asynchronous input used to reset the controller so that a new computation can begin. The Clock input is used to synchronize the controller's action. All activities are assumed to be synchronized with the trailing edge of the clock pulse.

Step 6: Obtain the state diagram of the controller.

The controller must initiate a set of operations in a specified sequence. Therefore, it is modeled as a sequential circuit. The state diagram of the unsigned multiplier controller is shown in Figure 7.40.

Initially, the controller is in state T_0. At this point, the control signals C_0 and C_1 are HIGH. Operations $R \leftarrow 0$ and $M \leftarrow$ inbus are carried out with the trailing edge of the next clock pulse. The controller moves to state T_1 with this clock pulse. When the controller is in T_2, $R \leftarrow R + M$ and $Q \leftarrow Q - 1$ are performed.

All these operations take place at the trailing edge of the next clock pulse. The controller moves to state T_5 only when the unsigned multiplication is completed. The controller then stays in this state forever. A hardware reset input causes the controller to move to state T_0, and a new computation will start.

In this state diagram, selection of states is made according to the following guidelines:

- If the operations are independent of each other and can be completed within one clock cycle, they are grouped within one control state. For example, in Figure 7.40, operations $R \leftarrow 0$ and $M \leftarrow$ inbus are independent of each other. With this hardware, they can be executed in one clock cycle. That is, they are

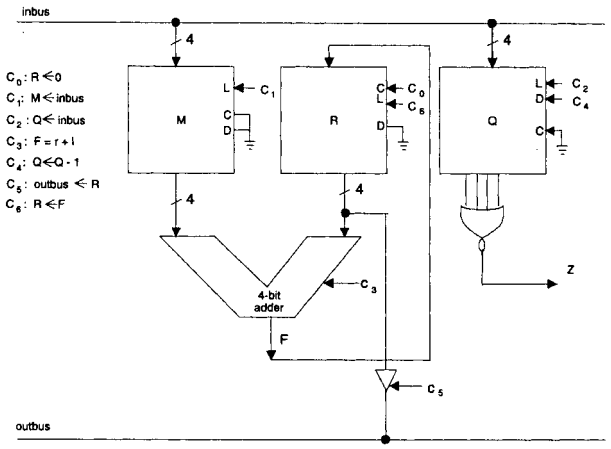


FIGURE 7.38 Detailed logic diagram of the processing section

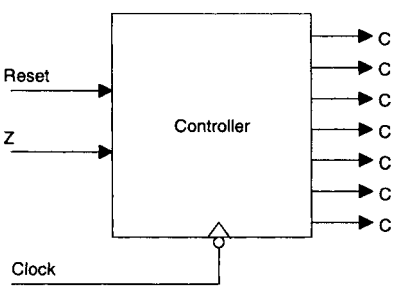
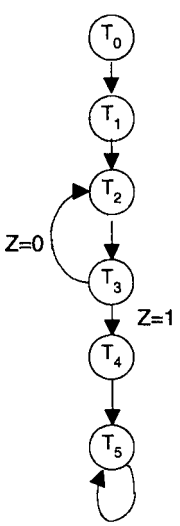


FIGURE 7.39 Block diagram of the unsigned multiplier controller



Control State	Operation Performed	Control Signal to be activated
T_0	R ← 0, M ← inbus	C_0, C_1
T_1	Q ← inbus	C_2
T_2	R ← R + M, Q ← Q - 1	C_3, C_4, C_6
T_3	None	None
T_4	outbus ← R	C_5
T_5	None	None

(a) State Diagram (b) Controller action

FIGURE 7.40 Controller description

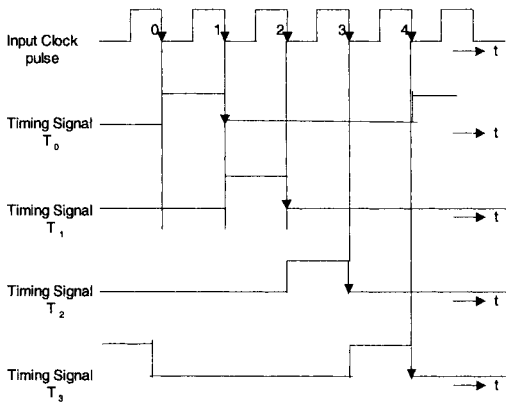


FIGURE 7.41 Timing signals generated by the controller

microoperations. However, if they cannot be completed within the T_0 clock cycle, either clock duration must be increased or the operations should be divided into a sequence of microoperations.

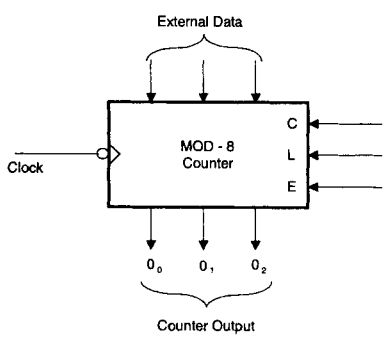
- Conditional testing normally implies the introduction of new states. For example, in the figure, conditional testing of Z introduces the new state T_3.
- One should not attempt to minimize the number of states. When in doubt, new states must be introduced. The correctness of the control logic is more important than the cost of the circuit.

Step 7: Specify the characteristics of the hardware for generating the required timing signals.

There are six states in the controller state diagram. Six nonoverlapping timing signals (T_0 through T_5) must be generated so that only one will be high for a clock pulse. For example, Figure 7.41 shows the four timing signals T_0, T_1, T_2, and T_3. A mod-8 counter and a 3-to-8 decoder can be used to accomplish this task. Figure 7.42 shows the mod-8 counter.

Step 8: Draw the logic circuit of the controller.

Figure 7.43 shows the logic circuit of the controller. The key element of the implementation in Figure 7.43 is the sequence controller (SC) hardware, which sequences



(a) Block Diagram

C	L	E	Clock	Action
1	X	X	X	Clear
0	1	X	↓	Load external data
0	0	1	↓	Count up
0	0	0	↓	No operation

Note: X = don't care

(b) Function Table

FIGURE 7.42 Characteristics of the counter used in the controller design

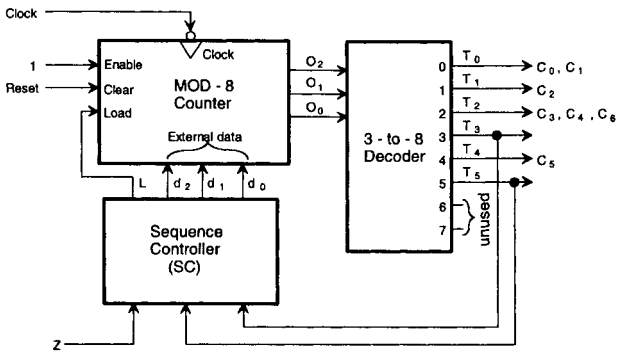
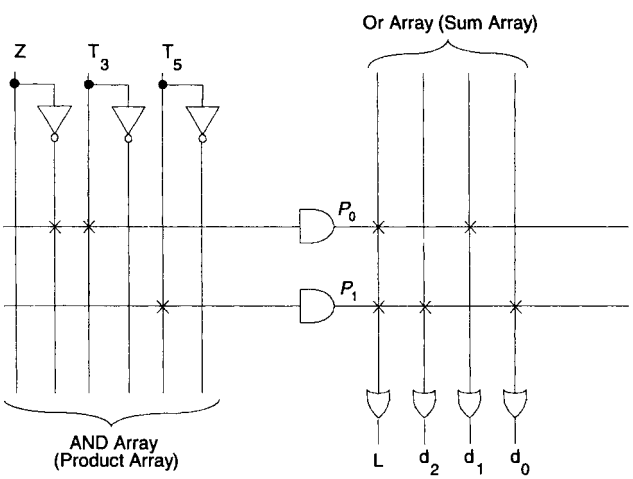


FIGURE 7.43 Logic diagram of the unsigned multiplier controller

Inputs			Outputs			
Z	T_3	T_5	L	d_2	d_1	d_0
0	1	x	1	0	1	0
x	x	1	1	1	0	1

Note: x = don't care

(a) Truth Table



(b) PLA Implementation

FIGURE 7.44 Sequence controller design

the controller according to the state diagram of Figure 7.40. Figure 7.44(a) shows the truth table for the SC controller.

Consider the logic involved in deriving the entries of the SC truth table. The mod-8 counter is loaded (or initialized) with the specified external data if the counter control inputs *C* and *L* are 0 and 1 respectively from Figure 7.42. In this counter, the counter load

control input L overrides the counter enable control input E.

From the controller's state diagram of Figure 7.40, the controller counts up automatically in response to the next clock pulse when the counter load control input L = 0 because the enable input E is tied to HIGH. Such normal sequencing activity is desirable for the following situations:

- Present control state is T_0, T_1, T_2, T_4.
- Present control state is T_3 and $Z = 1$; the next state is T_4.

The SC must load the counter with the appropriate count when the counter is required to load the count out of its normal sequence.

For example, from the controller's state diagram of Figure 7.40, if the present control state is T_3 (counter output $O_2O_1O_0$= 011) and if $Z = 0$, the next state is T_2. When these input conditions occur, the counter must be loaded with external value 010 at the trailing edge of the next clock pulse ($T_2 = 1$ only when $O_2O_1O_0$= 010. Therefore, the SC generates $L = 1$ and $d_2d_1d_0 = 010$.

Similarly, from the controller's state diagram of Figure 7.40, if the present state is T_5, the next control state is also T_5. The SC must generate the outputs $L = 1$ and $d_2d_1d_0 =$ 101. The SC truth table of Figure 7.41 shows these out-of-sequence counts. For each row of the SC truth table of Figure 7.44(a), a product term is generated in the PLA:

$$P_0 + \overline{Z}T_3 \text{ and } P_1 = T_5.$$

The PLA (Figure 7.44b) generates four outputs: L, d_2, d_1, and d_0. Each output is directly generated by the SC truth table and the product terms. The PLA outputs are as follows:

$$\begin{aligned} L &= P_0 + P_1 \\ d_2 &= P_1 \\ d_1 &= P_0 \\ d_0 &= P_1 \end{aligned}$$

The controller design is completed by relating the control states (T_0 through T_5) to the control signals (C_0 though C_6) as follows:

$$\begin{aligned} C_0 &= C_1 = T_0 \\ C_2 &= T_1 \\ C_3 &= C_4 = C_6 = T_2 \\ C_5 &= T_4 \end{aligned}$$

From these equations, when the control is in state T_0 or T_2, multiple micro-operations are performed. Otherwise,when the control is in state T_1 or T_4, a single micro-operation is performed.

The unsigned multiplication algorithm just implemented using hardwired control can be considered as an unsigned multiplication instruction with a microprocessor. To execute this instruction, the microcomputer will read (fetch) this multiplication instruction from external memory into the instruction register located inside the microprocessor. The contents of this instruction register will be input to the control unit for execution. The control unit will generate the control signals C_0 through C_6 as shown in Figure 7.43. These control signals will then be applied to the appropriate components of the processing section in Figure 7.38 at the proper instants of time shown in Figure 7.40. Note that the control signals are physically connected to the hardware elements of Figure 7.38. Thus, the execution of the unsigned multiplication instruction will be completed by the microprocessor.

Microprogrammed Control Unit Design

As mentioned earlier, a microprogrammed control unit contains programs written

using microinstructions. These programs are stored in a control memory normally in a ROM inside the CPU. To execute instructions, the microprocessor reads (fetches) each instruction into the instruction register from external memory. The control unit translates the instruction for the microprocessor. Each control word contains signals to activate one or more microoperations. A program consisting of a set of microinstructions is executed in a sequence of micro-operations to complete the instruction execution. Generally, all microinstructions have two important fields:

- Control word
- Next address

The control field indicates which control lines are to be activated. The next address field specifies the address of the next microinstruction to be executed. The concept of microprogramming was first proposed by W. V. Wilkes in 1951 utilizing a decoder and an 8 × 8 ROM with a diode matrix. This concept is extended further to include a control memory inside the CPU. The cost of designing a CPU primarily depends on the size of the control memory. The length of a microinstruction, on the other hand, affects the size of the control memory. Therefore, a major design effort is to minimize the cost of implementing a microprogrammed CPU by reducing the length of the microinstruction.

The length of a microinstruction is directly related to the following factors:

- The number of micro-operations that can be activated simultaneously. This is called the "degree of parallelism."
- The method by which the address of the next microinstruction is determined.

All microinstructions executed in parallel can be included in a single microinstruction with a common op-code. The result is a short microprogram. However, the length of the microinstruction increases as parallelism grows.

The control bits in a microinstruction can be organized in several ways. One obvious way is to assign a single bit for each control line. This will provide full parallelism. No decoding of the control field is necessary. For example, consider Figure 7.45 with two registers, X and Y with one outbus.

In figure 7.45, the contents of each register are transferred to the outbus when the

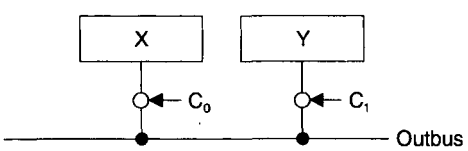


FIGURE 7.45 An example of a register transfer

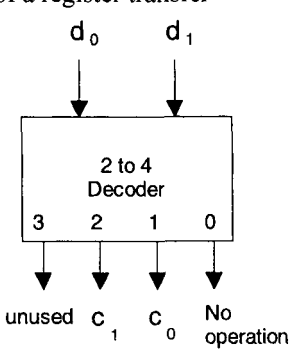


FIGURE 7.46 Encoded format

appropriate control line is activated:

C_0: outbus ← X

C_1: outbus ← Y

Here, each operation can be performed one at a time because there is only one outbus. A single bit can be assigned to perform each transfer as follows:

Control Bits		Operation Performed
C_0	C_1	
1	0	Outbus ← X
0	1	Outbus ← Y
0	0	No operation

This method is called "unencoded format."

The three operations can be implemented using two bits and a 2-to-4 decoder as shown in Figure 7.46. This is called "encoded format." The relationship between the encoded and actual control information is as follows:

Encoded Bits		Operation Performed
d_1	d_0	
0	0	No operation
0	1	Outbus ← x
1	0	Outbus ← y

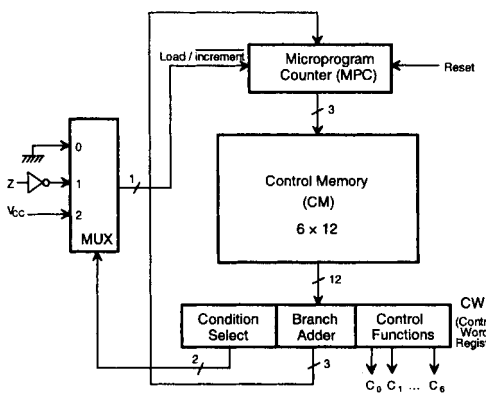
Note that a 5-bit control field is required for five operations. However, three encoded bits are required for five operations using a 3 to 8 decoder. Hence, the encoded format typically provides a short control field and thus results in short microinstructions. However, the need for a decoder will increase the cost. Therefore, there is a trade-off between the degree of parallelism and the cost. Microinstructions can be classified into two groups: horizontal and vertical. The horizontal microinstruction mechanism provides long microinstructions, a high degree of parallelism, and little or no encoding. The vertical microinstruction method, on the other hand, offers short microinstructions, limited parallelism, and considerable decoding.

Microprogramming is the technique of writing microprograms in a microprogrammed control unit. Writing microprograms is similar to writing assembly language programs. Microprograms are basically written in a symbolic language called microassembly language. These programs are translated by a microassembler to generate microcodes, which are then stored in the control memory.

In the early days, the control memory was implemented using ROMs. However, these days control memories are realized in writeable memories. This provides the flexibility of interpreting different instruction set by rewriting the original microprogram, which allows implementation of different control units with the same hardware. Using this approach, one CPU can interpret the instruction set of another CPU. The design of a microprogrammed control unit is considered next. The 4-bit × 4-bit unsigned multiplication

Control Memory Address		*Control Word*
0	START	`R ← 0, M ← inbus;`
1		`Q ← inbus;`
2	LOOP	`R ← R + M, Q ← Q - 1;`
3		`If Z = 0 then goto Loop;`
4		`outbus ← R;`
5	HALT	`Go to HALT`

FIGURE 7.47 Symbolic microprogram for 4-bit x 4-bit unsigned multiplication using repeated addition



C_0 : R ← 0
C_1 : M ← inbus
C_2 : Q ← inbus
C_3 : F ← 1 + r
C_4 : Q ← Q − 1
C_5 : outbus ← R
C_6 : R ← F

FIGURE 7.48 Microprogrammed unsigned multiplier control unit

using hardwired control (presented earlier) is implemented by microprogramming. The register transfer description shown in Figure 7.36 is rewritten in symbolic microprogram language as shown in Figure 7.47. Note that the unsigned 4-bit x 4-bit multiplication uses repeated addition. The result (product) is assumed to be 4 bits wide.

To implement the microprogram, the hardware organization of the control unit shown in Figure 7.48 can be used. The various components of the hardware of Figure 7.48 are described in the following:

1. **Microprogram Counter (MPC).** The MPC holds the address of the next microinstruction to be executed. It is initially loaded from an external source to point to the starting address of the microprogram. The MPC is similar to the program counter (PC). The MPC is incremented after each microinstruction fetch. If a branch instruction is encountered, the MPC is loaded with the contents of the branch address field of the microinstruction.
2. **Control Word Register (CWR).** Each control word in the control memory in this example is assumed to contain three fields: condition select, branch address, and control function. Each microinstruction fetched from the Control Memory is loaded into the CWR. The organization of the CWR is same for each control word

and contains the three fields just mentioned. In the case of a conditional branch microinstruction, if the condition specified by the condition select field is true, the MPC is loaded with the branch address field of the CWR; otherwise, the MPC is incremented to point to the next microinstruction. The control function field contains the control signals.

3. **MUX (Multiplexer).** The MUX is a condition select multiplexer. It selects one of the external conditions based on the contents of the condition select field of the microinstruction fetched into the CWR.

In Figure 7.48, a 2-bit condition select field is required as follows:

Condition Select Field		Interpretation
0	0	No branching (no condition)
0	1	Branch if $Z = 0$
1	0	Unconditional branching

From Figure 7.47 six control memory address (addresses 0 through 5) are required for the control memory to store the microprogram. Therefore, a 3-bit address is necessary for each microinstruction. Hence, three bits for the branch address field are required. From Figure 7.48 seven control signals (C_0 through C_6) are required. Therefore, the size of the control function field is 7 bits wide. Thus, the size of each control word can be determined as follows:

size of *a* control word	=	size of the condition select field	+	size of the branch address field	+	number of control signals
	=	2	+	3	+	7
	=	12 bits				

Therefore, the size of the control memory is 6 bits × 12 bits because the microprogram requires six addresses (0 through 5) and each control word is 12 bits wide. The size of the CWR is 12 bits. The complete binary listing of the microprogram is shown in Figure 7.49.

ROM Address			Control Word											Comments
In decimal	In binary		Condition Select		Branch Address			Control Function						
								C_0	C_1	C_2	C_3	C_4	C_5	C_6	
0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	R← 0, M ← inbus
1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	Q ← inbus
2	0	1	0	0	0	0	0	0	0	0	1	1	0	1	R ← R + M, Q ← Q –1, R ← F
3	0	1	1	0	1	0	1	0	0	0	0	0	0	0	If Z = 0 then go to address 2 (loop)
4	1	0	0	0	0	0	0	0	0	0	0	0	1	0	outbus ← R
5	1	0	1	1	0	1	0	1	0	0	0	0	0	0	Go to address 5 (HALT)

FIGURE 7.49 Binary listing of the microprogram for 4-bit × 4-bit unsigned multiplication

Let us now explain the binary program. Consider the first line of the program. The instruction contains no branching. Therefore, the condition select field is 00. The contents of the branch in this case filled with 000. In the control function field, two micro-operations, C_0 and C_1, are activated. Therefore, both C_0 and C_1 are set to 1; C_2 through C_6 are set to 0.

This results in the following binary microinstruction shown in the first line (address 0) of Figure 7.49:

Condition Select	Branch Address	Control Function
00	000	1100000

Next, consider the conditional branch instruction of Figure 7.49. This microinstruction implements the conditional instruction "If $Z = 0$ then go to address 2." In this case, the microinstruction does not have to activate any control signal of the control function field. Therefore, C_0 through C_6 are zero. The condition select field is 01 because the condition is based on $Z = 0$. Also, if the condition is true ($Z = 0$), the program branches to address *2*. Therefore, the branch address field contains 010_2. Thus, the following binary microinstruction is obtained:

Condition Select	Branch Address	Control Function
01	010	000000

The other lines in the binary representation of the microprogram can be explained similarly. To execute an unsigned multiplication instruction implemented using the repeated addition just described, a microprogrammed microprocessor will fetch the instruction from external memory into the instruction register. To execute this instruction, the microprocessor uses the control unit of Figure 7.48 to generate the control word based on the microprogram of Figure 7.49 stored in the control memory. The control signals C_0 through C_6 of the control function field of the CWR will be connected to appropriate components of Figure 7.38 The instruction will thus be executed by the microprocessor.

By examining the microprogram in Figure 7.49, it is obvious that the control function field contains all zeros in case of branch instructions. In a typical microprogram, there may be several conditional and unconditional branch instructions. Therefore, a lot of valuable memory space inside the control unit will be wasted if the control field is filled with zeros. In practice, the format of the control word is organized in a different manner to minimize its size. This reduces the implementation cost of the control unit. Whenever there are several branch instructions, the microinstructions, can be formatted by using a method called multiple microinstruction format. In this approach, the microinstructions are divided into two groups: operate and branch instructions.

An operate instruction initiates one or more microoperations. For example, after the execution of an operate instruction, the MPC will be incremented by 1. In the case of a branch instruction, no microoperation will usually be initiated, and the MPC may be loaded with a new value.

This means that the branch address field can be removed from the microinstruction format. Therefore, the control function field is used to specify the branch address itself. Typically,

ROM Address		Control Word													Comments
In decimal	In binary	Condition Select		Branch Address			Control Function								
							C_0	C_1	C_2	C_3	C_4	C_5	C_6		
0	0 0 0	0	0	0	0	0	1	1	0	0	0	0	0		R← 0, M ← inbus
1	0 0 1	0	0	0	0	0	0	0	1	0	0	0	0		Q ← inbus
2	0 1 0	0	0	0	0	0	0	0	0	1	1	0	1		R ← R + M, Q ← Q −1, R ← F
3	0 1 1	0	1	0	1	0	0	0	0	0	0	0	0		If Z = 0 then go to address 2 (loop)
4	1 0 0	0	0	0	0	0	0	0	0	0	0	1	0		outbus ← R
5	1 0 1	1	0	1	0	1	0	0	0	0	0	0	0		Go to address 5 (HALT)

FIGURE 7.50 Reduction of the length of microinstruction of Figure 7.49

each microinstruction will have two fields, as shown next:

CONDITION-SELECT FIELD		CONTROL FUNCTION FIELD						
S_1	S_0	C_6	C_5	C_4	C_3	C_2	C_1	C_0

If S_1 S_0 = 00, the microinstruction is considered as an operate instruction, and the contents of the control function field are treated as the control signals. Assume the Condition Select Field is encoded as follows:

S_1	S_0	
0	0	No branch
0	1	Branch if cond-1 = 1
1	1	Branch if cond-2 = 1
1	0	Unconditional branch

If S_1 S_0 = 01, the instruction is regarded as a branch instruction, and the contents of the control field are assumed to be a 7-bit branch address. In this example, it is assumed that when S_1 S_0 = 01, the MPC will be loaded with the appropriate address specified by C_6 C_5 C_4 C_3 C_2 C_1 C_0 if the condition Z = 0 is satisfied; on the other hand, if S_1 S_0 = 10, an unconditional branch to the address specified by the Control Function / Branch Address Field occurs.

In order to illustrate this concept, the microprogram for 4-bit by 4-bit unsigned multiplication of Figure 7.49 is rewritten using the multiple instruction format as shown in Figure 7.50.

It can be seen from the figure 7.50 that the total size of the control store is 54 bits (6 × 9 = 54). In contrast, the control store of figure 7.49 contains 72 bits. For large microprograms with many branch instructions, tremendous memory savings can be accomplished using the multiple microinstructon format. Addresses 0, 1, 2, and 4 contain microinstructions with the contents of the conditional select field as 00, and are considered as operate instructions. In this case, the contents of the control function field are directed to the processing hardware.

Address 3 contains a conditional branch instruction since the contents of the condition select field are 01; while address 5 contains an unconditional branch instruction

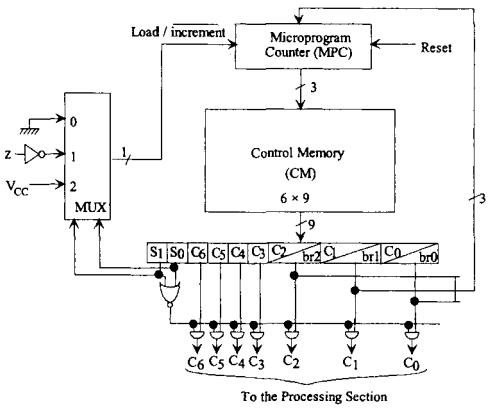


FIGURE 7.51 Microprogrammed Controller for the Microprogram of Figure 7.50.

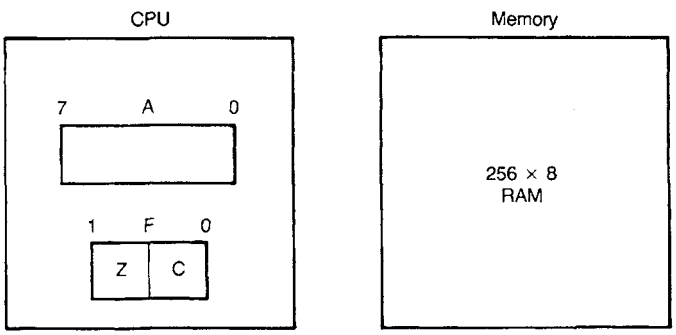


FIGURE 7.52 Programming Model of a Simple Processor

(halt instruction; that is, jump to the same address) since the condition select field is 10. Hence, the 7-bit control function field directly specifies the desired branch addresses 2 and 5, respectively. Figure 7.51 shows the hardware schematic.

7.4 Design of a Microprogrammed CPU

Next, the design of a microprogrammed processor is illustrated. The programming model of this processor is shown in Figure 7.52.

The CPU contains two registers:

1. An 8-bit register A
2. A 2-bit flag register F

The flag register holds only zero (Z) and carry (C) flags. All programs and data are stored in the 256 x 8 RAM. The detailed hardware schematic of the data-flow part of this processor is shown in Figure 7.53.

From Figure 7.53, it can be seen that the hardware organization includes four more 8-bit registers, PC, IR, MAR, and BUFFER. These registers are transparent to a programmer. The 8-bit register BUFFER is used to hold the data that is retrieved from memory. In this system, only a restricted number of data paths are available. These paths are controlled by the control inputs C_0 through C_9, as defined in Table 7.1.

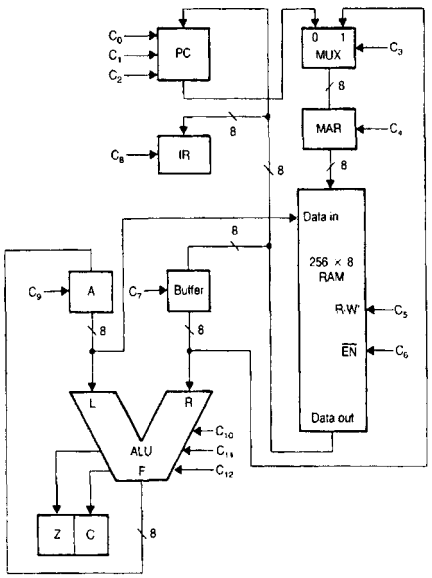


FIGURE 7.53 Hardware Schematic of the Simple Processor (Note: 8-bit PC is connected to eight 2 to 1 MUXs-- Not shown above)

From Figure 7.54, notice that the proposed instruction set contains 11 instructions. The first 7 instructions are classified as memory reference instructions, since they all require a memory address (which is an 8-bit number in this case). The last 4 instructions do not require any memory address; they are called nonmemory reference instructions. Each memory reference instruction is assumed to occupy 2 consecutive bytes in the RAM. The first byte is reserved for the op-code, and the second byte indicates the 8-bit memory address. In contrast, a nonmemory reference instruction takes only one byte of storage. This instruction set supports only two addressing modes: implicit and direct. Both branch instructions are assumed to be absolute mode branch instructions. The op-code encoding for this instruction set is carried out in a logical manner, as explained in Figure 7.55.
The bit I3 of Figure 7.55 decides the instruction type. If I3 = 1, it is a memory reference instruction (MRI), otherwise it is a nonmemory reference instruction (NMRI).
Within the memory reference category, instructions are classified into four groups, as follows:

GROUP NO.	INSTRUCTIONS
0	Load and store
1	Add and subtract
2	Jumps
3	Logical

There are two instructions in the first three groups. Bit I_0 is used to determine the desired instruction of a particular group. If Io of group 0 equals zero, it is the load (LDA) instruction; otherwise it is the store (STA) instruction. Nevertheless, no such classification is required for group 3 and the nonmemory reference instructions.

As mentioned before, the instruction execution involves the following steps:

TABLE 7.1 Definitions of the Control Inputs C_0-C_9

MICROOPERATION	COMMENT
C_0: PC ← 0	Clear PC to zero.
C_1: PC ← PC + 1	Advance the PC.
$C_2C_5\overline{C_6}$: PC ← M ((MAR))	Read the data from the memory and save it in the PC.
$\overline{C_3}C_4$: MAR ← PC	Transfer the contents of the PC into MAR.
$C_5\overline{C_6}C_7$: BUFFER ← M ((MAR))	Read the data from the memory and save the result in BUFFER.
C_3C_4: MAR ← BUFFER	Transfer the content of the BUFFER into MAR.
$C_5\overline{C_6}C_8$: IR ← M ((MAR))	Read the data from memory and save the result into IR.
C_9: A ← F	Transfer the ALU output into the A register.
$\overline{C_5}\,\overline{C_6}$: M ((MAR)) ← A	Save contents of register A into memory.

The eight ALU operations performed by the CPU are defined by $C_{10}C_{11}C_{12}$ as follows:

C_{11}	C_{11}	C_{12}	F
0	0	0	0
0	0	1	R
0	1	0	L+R
0	1	1	L-R
1	0	0	L+1
1	0	1	L-1
1	1	0	L AND R
1	1	1	NOT L

Step 1: Fetch the instruction.
Step 2: Decode the instruction to find out the required operation.
Step 3: If the required operation is a halt operation, then go to Step 6; otherwise continue.
Step 4: Retrieve the operands and perform the desired operation.
Step 5: Go to Step 1.
Step 6: Execute an infinite LOOP.

The first step is known as the fetch cycle, and the rest are collectively known as the execution cycle. To decode the instruction, the hardware shown in Figure 7.56 is used.

With this hardware and the status flags (Z and C), a microprogram to implement the instruction set can be written. The symbolic version of this microprogram is shown in

General Format	Instruction Length in Bytes	Object Code		Instruction Type	Operation	Comment
		In binary	In hex			
LDA ‹addr›	2	0000 1000	08	MRI	A ← M (‹addr›)	Load register A direct
		‹addr8›	‹addrH)			
STA ‹addr›	2	0000 1001	09	MRI	M (‹addr›) ← A	Store register A direct
		‹addr8›	‹addrH›			
ADD ‹addr›	2	0000 1010	0A	MRI	A ← A + M (‹addr›)	Add register A direct
		‹addr8›	‹addrH›			
SUB ‹addr›	2	0000 1011	0B	MRI	A ← A – M (‹addr›)	Subtract register A direct
		‹addr8›	‹addrH›			
JZ ‹addr›	2	0000 1100	0C	MRI	If Z=l then PC ← ‹addr› else PC ← PC + 1	Jump on zero flag set
		‹addr8›	‹addrH›			
JC ‹addr›	2	0000 1101	0D	MRI	If C = 1 then PC ← ‹addr› else PC ← PC + 1	Jump on carry flag set
		‹addr8›	‹addrH›			
AND ‹addr›	2	0000 1110	0E	MRI	A ← A ^ M (‹addr›)	And register A direct
		‹addr8›	‹addrH›			
CMA	1	0000 0000	00	NMRI	A ← $\overline{A}$	Comple-ment register A
INCA	1	0000 0010	02	NMRI	A ← A + 1	Increment register A
DCRA	1	0000 0100	04	NMRI	A ← A - 1	Decrement register A
HLT	1	0000 0110	06	NMRI	Halt	Halt CPU.

‹addr8›: 8-bit memory address in binary
‹addrH›: 8-bit memory address in hex
MRI: memory reference instruction
NMRI: nonmemory reference instruction.

FIGURE 7.54 Instruction Set to be Implemented

Figure 7.57.

The hardware organization of the microprogrammed control unit for this situation shown in Figure 7.58 directly follows the symbolic listing shown in Figure 7.57. No attempt has been made toward arriving at a minimal microprogram. Rather, the concept was presented. The task of translating the symbolic microprogram of Figure 7.57 into a binary microprogram is left as an exercise.

Mnemonic	Op-code Bit and Their Interpretations							
					TC	GN		SC
	I7	I6	I5	I4	I3	I2	I1	I0
LDA	0	0	0	0	1	0	0	0
STA	0	0	0	0	1	0	0	1
ADD	0	0	0	0	1	0	1	0
SUB	0	0	0	0	1	0	1	1
JZ	0	0	0	0	1	1	0	0
JC	0	0	0	0	1	1	0	1
AND	0	0	0	0	1	1	1	0
CMA	0	0	0	0	0	0	0	0
INCA	0	0	0	0	0	0	1	0
DCRA	0	0	0	0	0	1	0	0
HLT	0	0	0	0	0	1	1	0

Note:

TC: Type classifier (if I3 = 1, then it is a MRI; otherwise it is a NMRI)

GN: Group number within a type

(I2	I1	Group no.	
0	0	0	
0	1	1	
1	0	2	
1	1	3	)

SC: Subcategory within a group

FIGURE 7.55 Op-code Encoding Logic

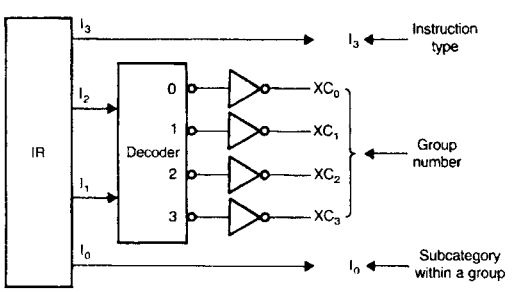


FIGURE 7.56 Instruction-decoding Hardware

Symbolic Microprogram:

ROM Address			
0		PC ← 0;	These operations constitute the fetch cycle.
1	FETCH	MAR ← PC;	
2		IR ← M ((MAR)), PC ← PC + 1;	
3		IF I_3 = 1 then go to MEMREF;	
4		IF XC_0 = 1 then go to CMA;	Here we decode the instructions.
5		IF XC_1 = 1 then go to INCA;	
6		IF XC_2 = 1 then go to DCRA;	
7		Go to HALT;	
8	CMA	A ← $\overline{A}$;	Execute CMA instructions.
9		Go to FETCH;	
10	INCA	A ← A + 1;	Execute INCA instruction.
11		Go to FETCH;	
12	DCRA	A ← A - 1;	Execute DCRA instruction.
13		Go to FETCH;	
14	MEMREF	IF XC_0 = 1 then go to LDSTO;	Here we branch to the various groups of the memory reference instruction.
15		IF XC_1 = 1 then go to ADSUB;	
16		IF XC_2 = 1 then go to JMPS;	
17	AND	MAR ← PC;	
18		BUFFER ← M ((MAR)), PC ← PC + 1;	Execute AND instruction.
19		MAR ← BUFFER;	
20		BUFFER ← M ((MAR));	
21		A ← A ∧ BUFFER;	
22		Go to FETCH;	
23	LDSTO	MAR ← PC;	
24		BUFFER ← M ((MAR)), PC ← PC + 1;	
25		MAR ← BUFFER;	
26		IF I_0 = 1 then go to STO;	
27	LOAD	BUFFER ← M ((MAR));	
28		A ← BUFFER;	
29		Go to FETCH;	
30	STO	M ((MAR)) ← A;	
31		Go to FETCH;	

FIGURE 7.57 Symbolic Microprogram that implements the instruction set of figure 7.54

```
32  ADSUB    MAR ← PC;
33           BUFFER ← M ((MAR)), PC ← PC + 1;
34           MAR ← BUFFER ;
35           BUFFER ← M ((MAR));
36           IF I0 = 1 then go to SUB;
37  ADD      A ← A + BUFFER;                    Execute ADD instruction
38           Go to FETCH;
39  SUB      A ← A – BUFFER;                    Execute SUB instruction
40           Go to FETCH;
41  JMPS     MAR ← PC;
42           IF I0 = 1 then go to JOC;
43           IF I0 = 1 then go to JOC;
44  JOZ      IF Z = 1 then go to LOADPC;        Execute JZ instruction
45           PC ← PC + 1;
46           Go to FETCH;
47  JOC      IF C = 1 then go to LOADPC;        Execute JC instruction
48           PC ← PC + 1;
49           Go to FETCH;
50  LOADPC   PC ← M((MAR));
51           Go to FETCH;
52  HALT     Go to HALT;                        Execute HALT instruction
```

FIGURE 7.57 *Continued*

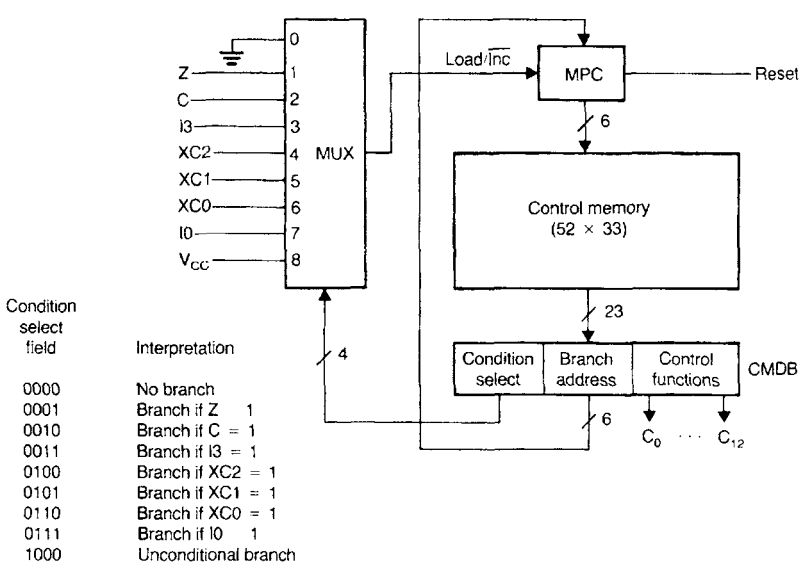


FIGURE 7.58 Microprogrammed Controller for the CPU

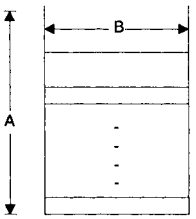


FIGURE 7.59 A microprogram of size A × B

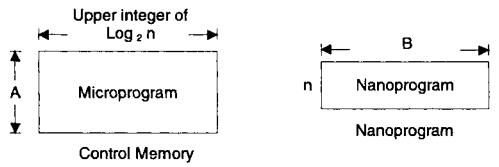


FIGURE 7.60 Nanomemory

000	0100
001	0000
010	0100
011	0100
100	0000
101	1010
110	1010

FIGURE 7.61 7 × 4-bit single control memory

000	00
001	01
010	00
011	00
100	01
101	10
110	10

7 × 2-bit microcontrol store

00	0100
01	0000
10	1010

3 × 4 nanocontrol store

FIGURE 7.62 Two-level store (nanomemory)

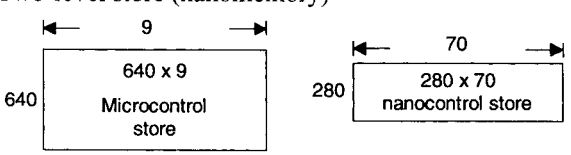


FIGURE 7.63 68000 nanomemory

Example 7.1

If the following two instructions are to be added to the instruction set of Figure 7.54, write a symbolic microprogram for the CPU of section 7.3 that describes the execution of each instruction:

	GENERAL FORMAT	OPERATION	DESCRIPTION
(a)	CLRA	A←0	Clear register A
(b)	PRSA	A← 11111111	Set register A to all ones

Solution:

```
(a)  CLRA:   A←0              ; Use ALU's zero output (C10C11C12=000)
             go to FETCH      ;
(b)  PRSA:   A←0              ; Use ALU's zero output (C10C11C12=000)
             A←Ā              ;
             go to FETCH      ;
```

Nanomemory is another approach for reducing the size of the control memory. This technique contains a two-level memory: control memory and nanomemory. At the outset, are may feel that the two-level memory will increase the overall cost. In fact, it reduces the cost of the system by minimizing the memory size.

The concept of nanomemory is derived from a combination of horizontal and vertical instructions. However, this method provides trade-offs between them.

Motorola uses nanomemory to design the control units of their popular 16-bit and 32-bit microprocessors, including the 68000, 68020, 68030, and 68040. The nanomemory method provides significant savings in memory when a group of micro-operations occur several times in a microprogram. Consider the microprogram of Figure 7.59, which contains A microinstructions B bits wide. The size of the control memory to store this microprogram is AB bits. Assume that the microprogram has n ($n < A$) unique microinstructions. These n microinstructions can be held in a separate memory called the "nanomemory" of size nB bits. Each of these n instructions occurs once in the nanomemory. Each microinstruction in the original microprogram is replaced with the address that specifies the location of the nanomemory in which the original B-bit-wide microinstructions are held.

Because the nanomemory has n addresses, only the upper integer of $\log_2 n$ bits is required to specify a nanomemory address. This is illustrated in Figure 7.60. The operation of microprocessor employing a nanomemory can be explained as follows: The microprocessor's control unit reads an address from the microprogram. The content of this address in the nanomemory is the desired control word. The bits in the control word are used by the control unit to accomplish the desired operation. Note that a control unit employing nanomemory (two-level memory) is slower than the one using a conventional control memory (single memory). This is because the nanomemory requires two memory reads (one for the control memory and the other for the nanomemory). For a single conventional control memory, only one memory fetch is necessary. This reduction in control unit speed is offset by the cost of the memory when the same microinstructions occur many times in the microprogram.

Consider the 7 x 4-bit microprogram stored in the single control memory of Figure 7.61. This simplified example is chosen to illustrate the nanomemory concept even though this is not a practical example. In this program, 3 out of 7 microinstructions are unique.

Therefore, the size of the microcontrol store is 7 × 2 bits and the size of the nanomemory is 3 × 4 bits. This is shown in Figure 7.62.

Memory requirements for the single control memory = 7 × 4 = 28 bits. Memory requirements for nanomemory = (7 × 2 + 3 × 4) bits = 26 bits. Therefore, the saving using nanomemory = 28 - 26 = 2 bits. For a simple example like this, 2 bits are saved. The HMOS 68000 control unit nanomemory includes a 640 × 9-bit microcontrol store and a 280 × 70-bit nanocontrol store as shown in Figure 7.63. In Figure 7.63, out of 640 microinstructions, 280 are unique. If the 68000 were implemented using a single control memory, the requirements would have been 640 × 70 bits. Therefore,

Memory savings = (640 × 70) – (640 × 9 + 280 × 70) bits
= 44,800 – 25,360
= 19,440 bits

This is a tremendous memory savings for the 68000 control unit.

QUESTIONS AND PROBLEMS

7.1 It is desired to implement the following instructions using block code: ADD, SUB, XOR, MOVE, HALT. Draw a block diagram.

7.2 The instruction length and the size of an address field are 9 bits and 3 bits respectively. Is it possible to have

6 two-address instructions
15 one-address instructions
8 zero-address instructions

using expanding op-code technique? Justify your answer.

7.3 Using the instruction format of Problem 7.2, is it possible to have

7 two-address instructions
7 one-address instructions
8 zero-address instructions

using expanding opcode technique? Justify your answer.

7.4 Assume that it is desired to have 2 two-address, 7 one-address, and 25 zero-address instructions in a computer instruction set. Using expanding op-code technique with a 2-bit op-code and 3-bit address field, is it possible to accomplish the above? If so, justify your answer and determine the instruction length.

7.5 Assume that using an instruction length of 9 bits and the address field size of 3 bits, 5 two-address and 10 one-address instructions have already been designed, using expanding op-code technique. Is it possible to have at least 48 zero-address instructions that can be added to the instruction set?

7.6 Design a combinational logic shifter with 4-bit input and 4-bit output as follows:

$\overline{OE}$	Shift Count S_1	Shift Count S_0	4 - bit output
1	X	X	High Impedance output lines
0	0	0	No Shift
0	0	1	Right Shift once
0	1	0	Right Shift twice
0	1	1	Right Shift three times

where X means don't care. Using multiplexers and tristate buffers, draw a logic diagram.

7.7 Draw a logic diagram for a 4 × 4 barrel shifter.

7.8 Using a minimum number of full adders and multiplexers, design an incrementer/ decrementer circuit as follows: If $S = 0$, output $y = x + 1$; otherwise, $y = x - 1$. Assume x and y are 4-bit signed numbers and the result is 4 bits wide.

7.9 Design a combinational circuit to compute the absolute value of an 8-bit twos complement number. Use 8-bit binary adder and exclusive-OR gates. Draw a logic circuit.

7.10 Using a 4-bit CLA as the building block, design an 8-bit adder.

7.11 Design:

(a) a 16-bit adder whose worst-case add-time is 10Δ using a 4-bit CLA as a building block.

(b) the fastest 64-bit adder using a 4-bit CLA as the building block. Estimate the worst-case add-time of your design.

(c) a combinational circuit to compute the function $f(x) = (3/8) * x$ where x is a 4-bit 2's complement number.

7.12 Design an arithmetic logic unit to perform the following functions:

S_1	S_0	F
0	0	A plus B
0	1	A minus B
1	0	A AND B
1	1	A OR B

Use multiplexers, binary adders, and gates as needed. Assume that A and B are 4-bit numbers. Draw a logic circuit.

7.13 Design a combinational circuit that will perform the following operations:

S_1	S_0	Y
0	0	0
0	1	A
1	0	B
1	1	15_{10}

Assume that A is a 4-bit number and $B = \overline{a_3}\,\overline{a_2}\,\overline{a_1}\,\overline{a_0}$. Draw a logic diagram.

7.14 Design a 4-bit ALU to perform the following operations:

S	F
0	Logical Left Shift A once
1	0

Assume that A is a 4-bit number. Draw a logic diagram using a binary adder, multiplexers, and inverters as necessary.

7.15 Design a 4-bit arithmetic unit as follows:

S	F
0	A plus B
1	A plus 1

Assume that A and B are 4-bit numbers

7.16 Design an ALU to perform the following operations:

S_1	S_0	F
0	0	x plus y
0	1	x
1	0	B
1	1	x $\oplus$ y

Assume that x and y are 4-bit numbers, and $B = \overline{y_3}\,\overline{y_2}\,\overline{y_1}\,\overline{y_0}$. Draw a logic diagram.

7.17 Assume two 2's complement signed numbers, $M = 11111111_2$ and $Q = 11111100_2$. Perform the signed multiplication using the algorithm described in Section 7.2.2.

7.18 What is the purpose of the control unit in a microprocessor?

7.19 Draw a logic diagram to implement the following register transfers:

(a) If the content of the 8-bit register R is odd, then

$$x \leftarrow x \oplus y$$

else $x \leftarrow x$ AND y

Assume x and y are 4 bits wide.

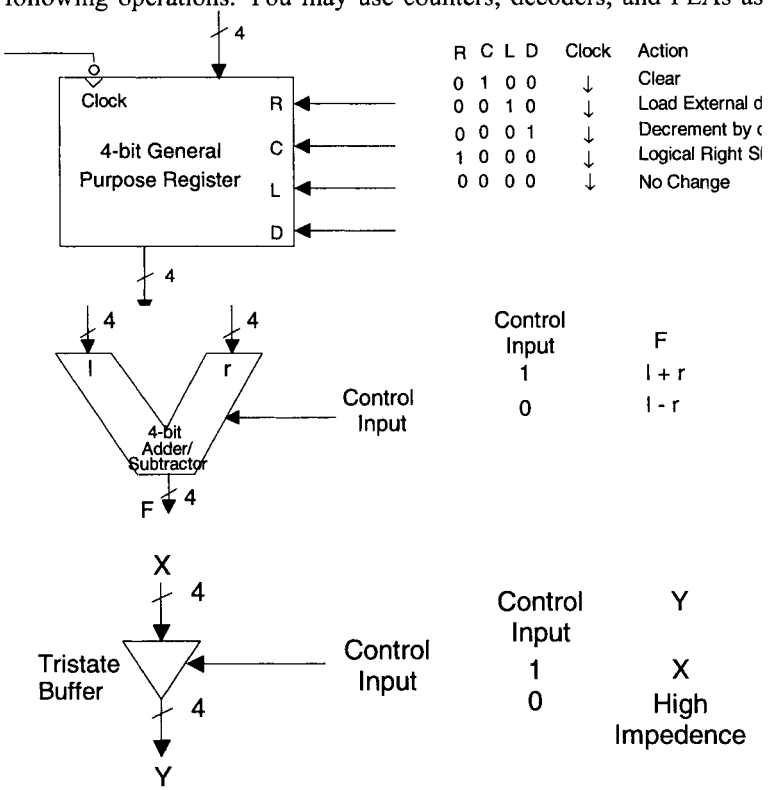
(b) If the number in the 8-bit register R is negative, then $x \leftarrow x - 1$ else $x \leftarrow x + 1$. Assume x and y are 4 bits wide.

7.20 Discuss briefly the merits and demerits of single-bus, two-bus, and three-bus architectures inside a control unit.

7.21 What is the basic difference between hardwired control, microprogramming, and nanoprogramming? Name the technique used for designing the control units of the Intel 8086, Motorola 68000, and PowerPC.

7.22 Using the following components: 4-bit general-purpose register, 4-bit adder/subtractor, and tristate buffer, and assuming the inbus and outbus are

4 bits wide, design a control unit using hardwired control to perform the following operations. You may use counters, decoders, and PLAs as required.



(a) Outbus ← 4 × *A*. Assume *A* is a 4-bit unsigned number and the result is 4 bits wide.

(b) If the 4-bit number in register *B* is odd, outbus ← 0; otherwise outbus ← *A* + (*B* / 2). Assume *A* and *B* are unsigned 4 bit numbers. Also, assume data is already loaded into *B*.

(c) If the content of a 4-bit register *Q* = 0, perform *R* ← *M* and then transfer the 4-bit result to outbus. On the other hand, if the content of the 4-bit register *Q* ≠ 0, perform *R* ← 0 and then transfer the 4-bit result to the outbus. Assume *M* and *R* are 4 bits wide.

7.23 Repeat Problem 7.22 using microprogramming.

7.24 Discuss the basic differences between microprogramming and nano-programming.

7.25 (a) A conventional microprogrammed control unit includes 1024 words by 85 bits. Each of 512 microinstructions are unique. Calculate the savings if any by having a nanomemory. Calculate the sizes of microcontrol memory and nanomemory.

(b) Consider the following 14 × 6 microprogram using a conventional control memory:

0000	000001
0001	000010
0010	000001
0011	000011
0100	000001
0101	000010
0110	000001
0111	000011
1000	000010
1001	000001
1010	000011
1011	000010
1100	000011
1101	000010
1110	000001

Implement this microprogram in a nanomemory. Justify the use of either a single-control memory or a two-level memory for the program.

7.26 Discuss the basic differences between CISC and RISC.

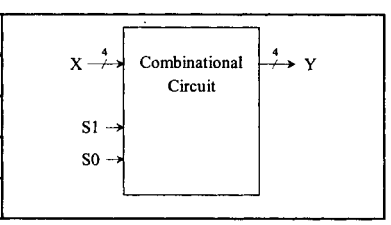
7.27 Design and implement a combinational circuit that will work as follows:

S1	S0	F
0	0	A plus B
0	1	Shift left (A)
1	0	A plus B plus 1
1	1	Shift left (A) + 1

Note that A and B are 4-bit operands

7.28 i) Design a combinational circuit that will satisfy the following specification.

S1	S0	Y_i
0	0	0
0	1	X_i
1	0	$\overline{X_i}$
1	1	1



ii) Using the results of part i), design a 4-bit, 8-function arithmetic unit that ii) will function as described next:

S2	S1	S0	F
0	0	0	A
0	0	1	A plus B
0	1	0	A plus $\overline{B}$
0	1	1	A minus 1

1	0	0	A plus 1
1	0	1	A plus B plus 1
1	1	0	A plus $\overline{B}$ plus 1
1	1	1	A

7.29 Design a 4-bit, 8-function arithmetic unit that will meet the following specifications:

S2	S1	S0	F
0	0	0	2A
0	0	1	A plus $\overline{B}$
0	1	0	A plus B
0	1	1	A minus 1
1	0	0	2A plus 1
1	0	1	A plus $\overline{B}$ plus 1
1	1	0	A plus B plus 1
1	1	1	A

7.30 (a) Using a 4-bit binary adder with inputs (A, B, and C_{in}), outputs (F and C_{out}), and one selection bit (S0), design an arithmetic circuit as follows:

S0	FUNCTION TO BE PERFORMED
0	A plus B
1	B plus 1

(b) Using another selection bit S1, modify the circuit of i) to include the arithmetic and logic functions as follows:

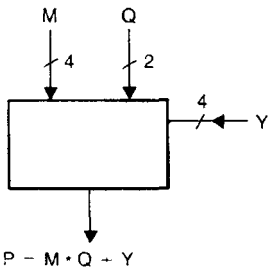
S1	S0	FUNCTION TO BE PERFORMED
0	0	F = A plus B
0	1	F = B
1	0	F = shift left (logical) A
1	1	F = $\overline{A}$

(c) Design a 4-bit logic unit that will function as follows:

S1	S0	F
0	0	A + B
0	1	A • B
1	0	$\overline{A}$
1	1	A ⊕ B

7.31 Design and implement a 6 × 6 array multiplier.

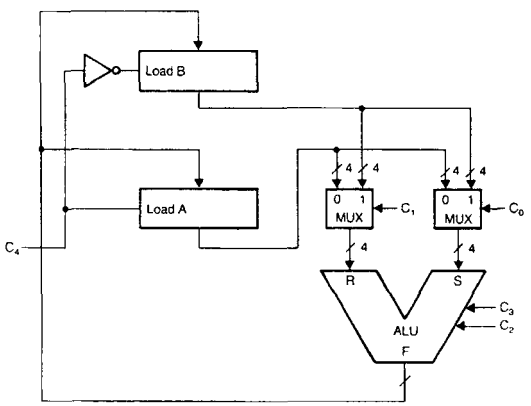
7.32 Design an unsigned 8 × 4 non-additive multiplier using additive-multiplier-module whose block diagram representation is as follows:



Assume that M, Q, and Y are unsigned integers.

7.33 Using four 256 x 8 ROMS and 4-bit parallel adders, design a 8 x 8 unsigned, nonadditive multiplier. Draw a logic diagram of your implementation.

7.34 Consider the registers and ALU shown in Figure P7.34:



The interpretation of various control points are summarized as follows:

C_3	C_2	F
0	0	R plus S
0	1	R minus S
1	0	R and S
1	1	R EX-OR S

C_1	C_0	R-INPUT	S-INPUT
0	0	A	A
0	1	A	B
1	0	B	A
1	1	B	B

C_4	ACTION
0	B ← F
1	A ← F

FIGURE P7.34

Answer the following questions by writing suitable control word(s). Each control word must be specified according to the following format: $C_4\ C_3\ C_2\ C_1\ C_0$
For example:

$C_4\ C_3\ C_2\ C_1\ C_0$
1 0 0 0 1 ;A ← A plus B

(a) How will the A register be cleared? (Suggest at least two possible ways.) DIRECT CLEAR input is not available.

(b) Suggest a sequence of control words that exchanges the contents of A and B registers (exchange means A ← B and B ← A).

7.35 Consider the following algorithm:

```
Declare registers A [8], B [8], C [8];
START: A ← 0; B ← 00001010;
LOOP:        A ← A + B; B ← B – 1;
             If B < > 0 then go to LOOP
             C ← A;
HALT:        Go to HALT
```

Design a hardwired controller that will implement this algorithm.

7.36 It is desired to build an interface in order establish communication between a 32-bit host computer and a front end 8-bit microcomputer (See Figure P7.36). The operation of this system is described as follows:

Step 1: First the host processor puts a high signal on the line "want" (saying that it needs a 32-bit data) for one clock period.

Step 2: The interface recognizes this by polling the want line.

Step 3: The interface unit puts a high signal on the line "fetch" for one clock period (that is it instructs the microcomputer to fetch an 8-bit data).

Step 4: In response to this, the microcomputer samples the speech signal, converts it into an 8-bit digital data and informs the interface that the data is ready by placing a high signal on the "ready" line for one clock period.

Step 5: The interface recognizes this (by polling the ready line), and it reads the 8-bit data into its internal register.

Step 6: The interface unit repeats the steps 3 through 5 for three more times (so that it acquires 32-bit data from the microcomputer).

Step 7: The interface informs the host computer that the latter can read the 32-bit data by placing a high signal on the line "takeit" for one clock period.

Step 8: The interface unit maintains a valid 32-bit data on the 32-bit output bus until the host processor says that it is done (the host puts a high signal on the line "done" for one clock period). In this case, the interface proceeds to step 1 and looks for a high on the "want" line.

(a) Provide a Register Transfer Language description of the interface.

(b) Design the processing section of the interface.

(c) Draw a block diagram of the interface controller.

(d) Draw a state diagram of the interface controller.

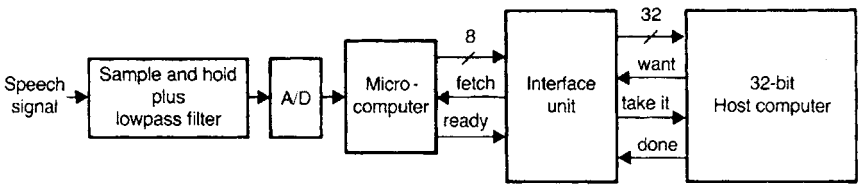


FIGURE P7.36

7.37 Solve Problem 7.35 using the microprogrammed approach.

7.38 Design a microprogrammed system to add numbers stored in the register pair AB and CD. A, B, C, and D are 8-bit registers. The sum is to be saved in the register pair AB. Assume that only an 8-bit adder is available.

7.39 The goal of this problem is to design a microprogrammed 3rd order FIR (Finite impulse response) digital filter. In this system, there are 4 coefficients w_0, w_1, w_2, and w_3. The output y_k (at the kth clock period) is the discrete convolution product of the inputs (x_ks) and the filter coefficients. This is formally expressed as follows:

$$y_k = w_0 * x_k + w_1 * x_{k-1} + w_2 * x_{k-2} + w_3 * x_{k-3}$$

In the above summation, x_k represents the input at the kth clock period while x_{k-i} represents input at $(k-i)$th sample period. For all practical purposes, we assume that our system is causal and so $x_i = 0$ for $i < 0$. The processing hardware is shown in Figure P7.39. This unit includes 8 eight-bit registers (to hold data and coefficients), A/D (Analog digital converter), MAC (multiplier accumulator), and a D/A (Digital analog converter). The processing sequence is shown below:

1 Initialize coefficient registers
2 Clear all data registers except x_i
3 Start A/D conversion (first make $sc = 1$ and then retract it to 0)
4 Wait for one control state (To make sure that the conversion is complete)
5 Read the digitized data into the register x_k
6 Iteratively calculate filter output y_k (use MAC for this)
7 Pass y_k to D/A (Pass Accumulator's output to D/A via Rounding ROM)
8 Move the data to reflect the time shift ($x_{k-3} = x_{k-2}$, $x_{k-2} = x_{k-1}$, $x_{k-1} = x_k$)
9 Go to 3

(a) Specify the controller organization.
(b) Produce a well documented listing of the binary microprogram

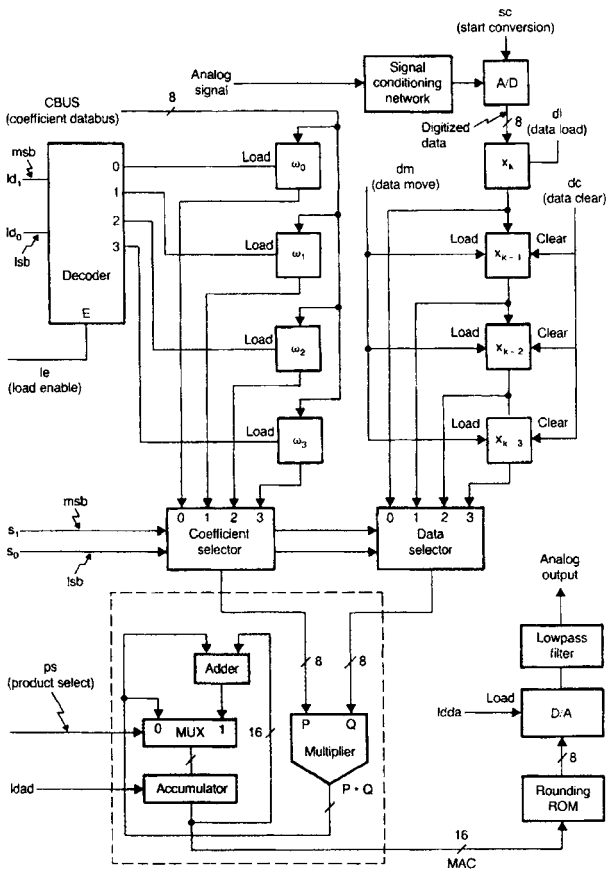
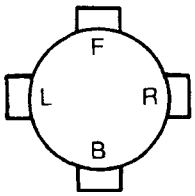


FIGURE P7.39

7.40 Your task is to design a microprogrammed controller for a simple robot with 4 sensors (see Fig. A). The sensor output will go high only if there is a wall or an obstruction within a certain distance. For example, if F= 1, there is an obstruction or wall in the forward direction. In particular, your controller is supposed to communicate with a motor controller unit shown in Fig. B. The flow chart that describes the control algorithm is shown in Fig. C. The outputs such as MFTS, MRT, MLT, MUT, and STP, andd the status signals such as FMC, and TC will be high for one clock period. Assume that a power on reset causes the controller to go the WAIT STATE 0.



F: forward direction sensor
R: right direction sensor
L: left direction sensor
B: backward direction sensor

Figure A

FIGURE P7.40a

(a) Specify the controller organization.

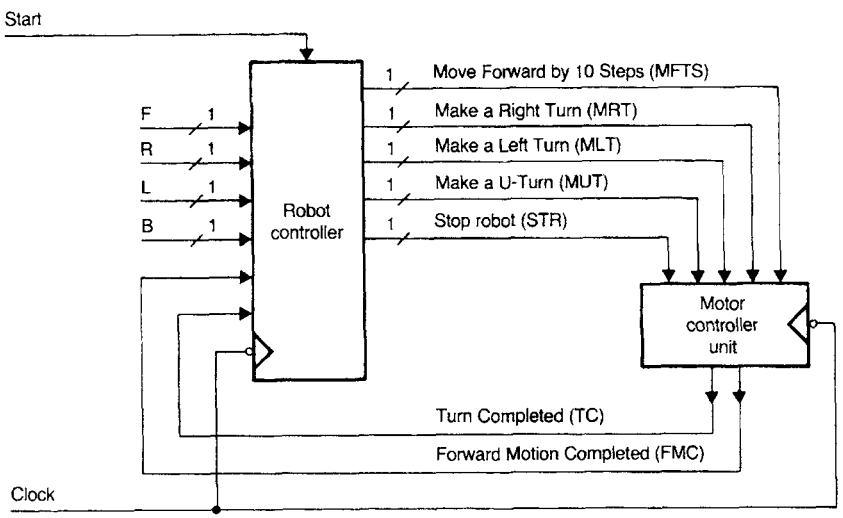


Figure B

FIGURE P7.40b

(b) Provide a well documented listing of the binary microprogram.

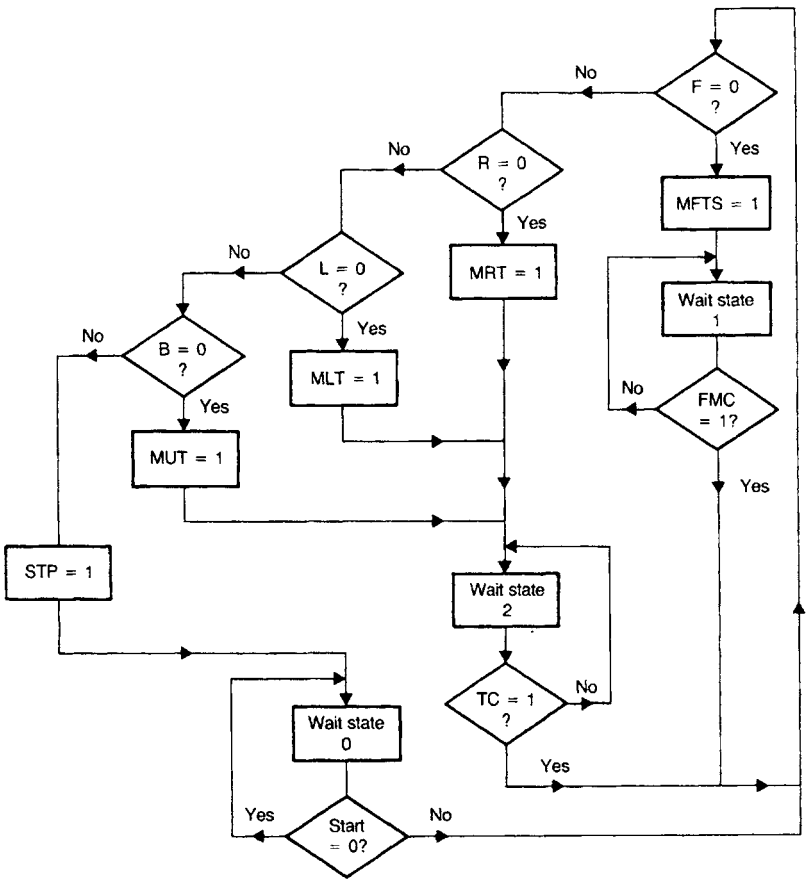


Figure C

FIGURE P7.40c

7.41 It is desired to add the following instructions to the instruction set shown in Figure 7.54.

	GENERAL FORMAT	OPERATION	DESCRIPTION
(a)	MVIA ‹data8›	A ← ‹data8›	This is an immediate mode move instruction. The first byte contains the op-code while the second byte contains the 8-bit data.
(b)	NEGA	A ← − A	This instruction negates the contents of A

Write a symbolic microprogram that describes the execution of each instruction.

7.42 Explain how the effect of an unconditional branch instruction of the following form is simulated:
JP ‹addr›

Use the instruction set shown in Figure 7.54.

7.43 Using the instruction set shown in Figure 7.54, write a program to add the contents of the memory locations 64_{16} through $6D_{16}$ and save the result in the address $6E_{16}$.

7.44 Show that it is possible to specify 675 microoperations using a 10 bit control function field.

7.45 A microprogram occupies 100 words and each word typically emits 70 control signals. The architect claims that by using a $2^i \times 70$ nanomemory (for some $i > 0$), it is possible to save 4260 bits. If this were true, determine the number of distinct control states in the original microprogram (Note that here when we say a control state we refer only to the control function field).
Hint: You may have to employ a trial and error approach to solve this problem.

8

MEMORY, I/O, AND PARALLEL PROCESSING

This chapter describes the basics of memory, input/output(I/O) techniques, and parallel processing. Topics include memory array design, memory management concepts, cache memory organization, input/output methods utilized by typical microprocessors, and fundamentals of parallel processing.

8.1 Memory Organization

8.1.1 Introduction

A memory unit is an integral part of any microcomputer system, and its primary purpose is to hold instructions and data. The major design goal of a memory unit is to allow it to operate at a speed close to that of the processor. However, the cost of a memory unit is so prohibitive that it is practically not feasible to design a large memory unit with one technology that guarantees a high speed. Therefore, in order to seek a trade-off between the cost and operating speed, a memory system is usually designed with different technologies such as solid state, magnetic, and optical.

In a broad sense, a microcomputer memory system can be divided into three groups:

- Processor memory
- Primary or main memory
- Secondary memory

Processor memory refers to a set of microprocessor registers. These registers are used to hold temporary results when a computation is in progress. Also, there is no speed disparity between these registers and the microprocessor because they are fabricated using the same technology. However, the cost involved in this approach limits a microcomputer architect to include only a few registers in the microprocessor. The design of typical registers is described in Chapters 5, 6 and 7.

Main memory is the storage area in which all programs are executed. The microprocessor can directly access only those items that are stored in main memory. Therefore, all programs must be within the main memory prior to execution. CMOS technology is normally used these days in main memory design. The size of the main memory is usually much larger than processor memory and its operating speed is slower than the processor registers. Main memory normally includes ROMs and RAMs. These are described in Chapter 6.

Electromechanical memory devices such as disks are extensively used as microcomputer's secondary memory and allow storage of large programs at a low cost. These secondary memory devices access stored data serially. Hence, they are significantly slower than the main memory. Popular secondary memories include hard disk and floppy disk systems. Programs are stored on the disks in files. Note that the floppy disk is removable whereas the hard disk is not. Secondary memory stores programs in excess of the main memory. Secondary memory is also referred to as "auxiliary" or "virtual" memory. The microcomputer cannot directly execute programs stored in the secondary memory, so in order to execute these programs, the microcomputer must transfer them to its main memory by a program called the "operating system."

Programs in disk memories are stored in tracks. A track is a concentric ring of programs stored on the surface of a disk. Each track is further subdivided into several sectors. Each sector typically stores 512 or 1024 bytes of information. All secondary memories use magnetic media except the optical memory, which stores programs on a plastic disk. CD-ROM is an example of a popular optical memory used with microcomputer systems. The CD-ROM is used to store large programs such as a C++ compiler. Other state-of-the-art optical memories include CD-RAM, DVD-ROM and DVD-RAM. These optical memories are discussed in Chapter 1.

In the past, one of the most commonly used disk memory with microcomputer systems was the floppy disk. The floppy disk is a flat, round piece of plastic coated with magnetically sensitive oxide material. The floppy disk is provided with a protective jacket to prevent fingerprint or foreign matter from contaminating the disk's surface. The 3½-inch floppy disk was very popular because of its smaller size and because it didn't bend easily. All floppy disks are provided with an off-center index hole that allows the electronic system reading the disk to find the start of a track and the first sector.

The storage capacity of a hard disk varied from 10 megabytes (MB) in 1981 to hundreds of gigabytes (GB) these days. The 3 ½-inch floppy disk, on the other hand, can typically store 1.44 MB. Zip disks were an enhancement in removable disk technology providing storage capacity of 100 MB to 750 MB in a single disk with access speed similar to the hard disk. Zip disk does not use a laser. Rather, it uses a magnetic-coated Myler inside, along with smaller read/write heads, and a rotational speed of 3000 rpm. The smaller heads allow the Zip drive to store programs using 2,118 tracks per inch, compared to 135 tracks per inch on a floppy disk. Floppy disks are being replaced these days by USB (Universal Serial Bus) Flash memory. Note that USB is a standard connection for computer peripherals such as CD burners. Also, flash memory gets its name because the technology uses microchips that allow a section of memory cells called blocks to be erased in a single action called a "flash". USB flash memory offers much more storage capacity than floppy disks, and can typically store 16 megabytes up to multiple gigabytes of information.

8.1.2 Main Memory Array Design

From the previous discussions, we notice that the main memory of a microcomputer is fabricated using solid-state technology. In a typical microcomputer application, a designer has to implement the required capacity by interconnecting several small memory chips. This concept is known as the "memory array design." In this section, we address this topic. We also show how to interface a memory system with a typical microprocessor.

Now let us discuss how to design ROM/RAM arrays. In particular, our discussion is focused on the design of memory arrays for a hypothetical microcomputer. The pertinent signals of a typical microprocessor necessary for main memory interfacing are shown in

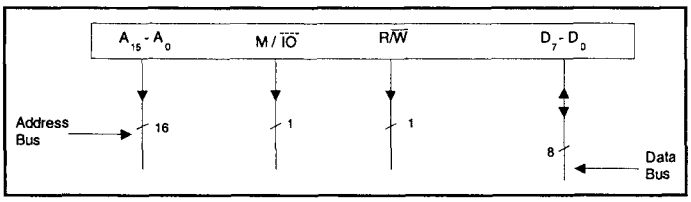


FIGURE 8.1 Pertinent signals of a typical microprocessor required for main memory interfacing

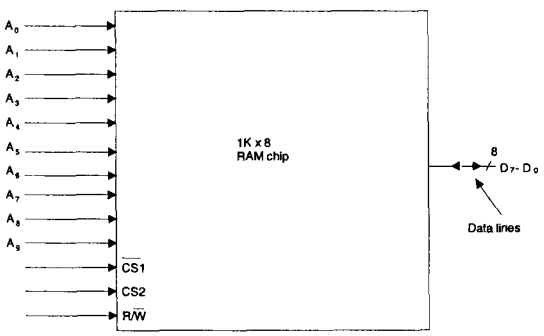


FIGURE 8.2 A typical 1K × 8 RAM chip

Figure 8.1. In Figure 8.1, there are 16 address lines, A_{15} through A_0, with A_0 being the least significant bit. This means that this microprocessor can directly address a maximum of 2^{16} = 65,536 or 64K bytes of memory locations. The control line M/$\overline{IO}$ goes to LOW if the microprocessor executes an I/O instruction, and it is held HIGH if the processor executes a memory instruction. Similarly, the control line R/$\overline{W}$ goes to HIGH to indicate that the operation is READ and it goes to LOW for WRITE operation. Note that all 16 address lines and the two control lines described so far are unidirectional in nature; that is, information can always travel on these lines from the processor to external units. Also, in Figure 8.1 eight bidirectional data lines D_7 through D_0 (with D_0 being the least significant bit) are shown. These lines are used to allow data transfer from the processor to external units and vice versa.

In a typical application, the total amount of main memory connected to a microprocessor consists of a combination of both ROMs and RAMs. However, in the following we will illustrate for simplicity how to design memory array using only the RAM chips.

The pin diagram of a typical 1K × 8 RAM chip is shown in Figure 8.2. In this RAM chip there are 10 address lines, A_9 through A_0, so one can read or write 1024 (2^{10} = 1024) different memory words. Also, in this chip there are 8 bidirectional data lines D_7 through D_0 so that information can travel back and forth between the microprocessor and the memory unit. The three control lines $\overline{CS1}$, CS2, and R/$\overline{W}$ are used to control the RAM unit according to the truth table shown in Figure 8.3. From this truth table it can be concluded that the RAM unit is enabled only when $\overline{CS1}$ = 0 and CS2 = 1. Under this condition, R/$\overline{W}$ = 0 and R/$\overline{W}$ = 1 imply write and read operations respectively.

To connect a microprocessor to ROM/RAM chips, three address-decoding techniques are usually used: linear decoding, full decoding, and memory decoding using

$\overline{CS1}$	CS2	R/$\overline{W}$	Function
0	1	0	Write Operation
0	1	1	Read Operation
1	X	X	The chip is not selected
X	0	X	The chip is not selectd

X means Don't Care

FIGURE 8.3 Truth table for controlling RAM

PLD. Let us first discuss how to interconnect a microprocessor with a 4K RAM chip array comprised of the four 1K RAM chips of Figure 8.2 using the linear decoding technique. Figure 8.4 uses the linear decoding to accomplish this.

In this approach, the address lines A_9 through A_0 of the microprocessor are connected to all RAM chips. Similarly, the control lines M/$\overline{IO}$ and R/$\overline{W}$ of the microprocessor are connected to the control lines CS2 and R/$\overline{W}$ respectively of each RAM chip. The high-order address bits A_{10} through A_{13} directly act as chip selects.

In particular, the address lines A_{10} and A_{11} select the RAM chips I and II respectively. Similarly, the address lines A_{12} and A_{13} select the RAM chips III and IV respectively. A_{15} and A_{14} are don't cares and are assumed to be 0. Figure 8.5 describes how

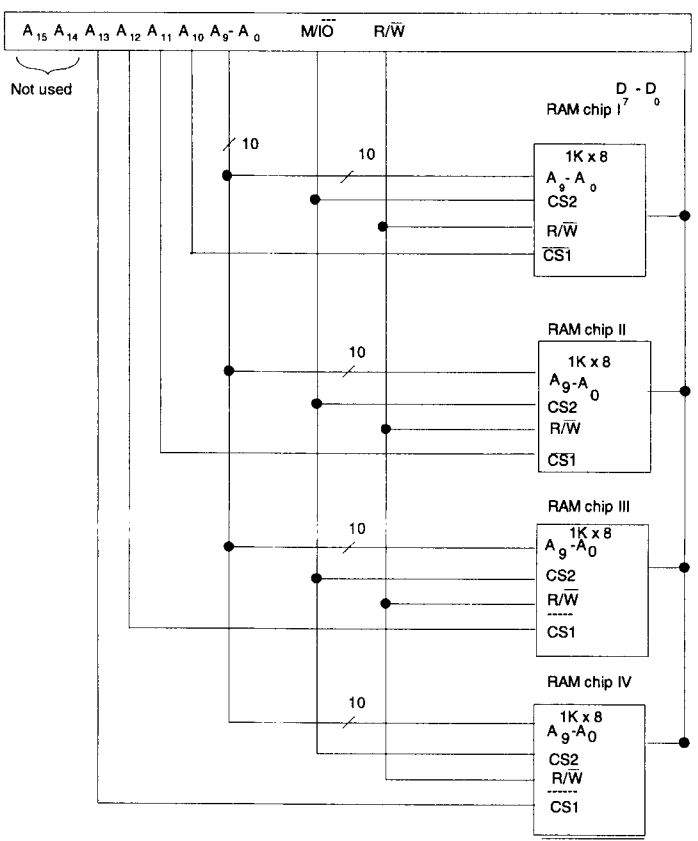


FIGURE 8.4 Microprocessor connected to 4K RAM using linear select decoding technique

Address Range in Hexadecimal	RAM Chip Number
3800-3BFF	I
3400-37FF	II
2C00-2FFF	III
1C00-1FFF	IV

FIGURE 8.5 Address map of the memory organization of Figure 8.4

the addresses are distributed among the four 1K RAM chips. This method is known as "linear select decoding," and its primary advantage is that it does not require any decoding hardware. However, if two or more lines of A_{10} through A_{13} are low at the same time, more than one RAM chip are selected, and this causes a bus conflict. Because of this potential problem, the software must be written in such a way that it never reads into or writes from any address in which more than one of the bits A_{13} through A_{10} are low. Another disadvantage of this method is that it wastes a large amount of address space. For example,

A_{12}	A_{11}	A_{10}	Selected RAM Chip
0	0	0	RAM chip I
0	0	1	RAM chip II
0	1	0	RAM chip III
0	1	1	RAM chip IV

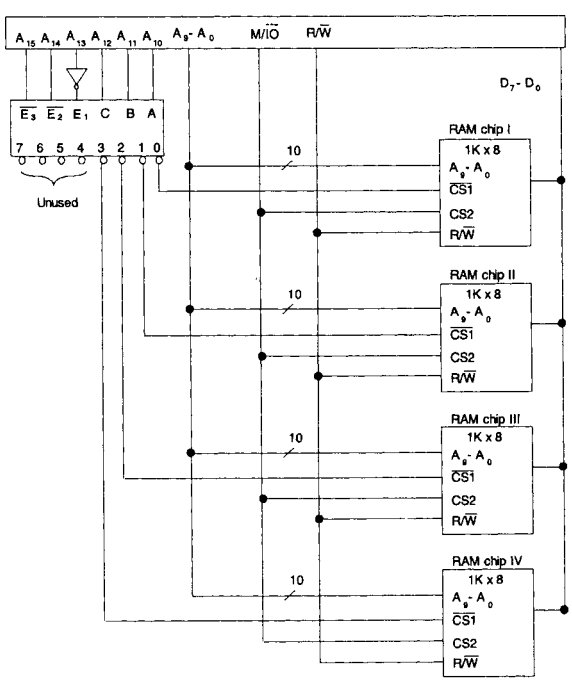


FIGURE 8.6 Interconnecting a microprocessor with a 4K RAM using full decoded memory addressing

whenever the address value is B800 or 3800, the RAM chip I is selected. In other words, the address 3800 is the mirror reflection of the address B800 (this situation is also called "memory foldback"). This technique is, therefore, limited to a small system. In particular, we can extend the system of Figure 8.4 up to a total capacity of 6K using A_{14} and A_{15} as chip selects for two more 1K RAM chips.

To resolve the problems with linear decoding, we use the full decoded memory addressing. In this technique, we use a decoder. The same 4K memory system designed using this technique is shown in Figure 8.6. Note that the decoder in the figure is very similar to a practical decoder such as the 74LS138 with three chip enables. In Figure 8.6 the decoder output selects one of the four 1K RAM chips depending on the values of A_{12}, A_{11}, and A_{10}. Note that the decoder output will be enabled only when $\overline{E3} = \overline{E2} = 0$ and E1 = 1. Therefore, in the organization of Figure 8.6, when any one of the high-order bits A_{15}, A_{14}, or A_{13} is 1, the decoder will be disabled, and thus none of the RAM chips will be selected. In this arrangement, the memory addresses are assigned as shown in Figure 8.7.

This approach does not waste any address space since the unused decoder outputs (don't cares) can be used for memory expansion. For example, the 3-to-8 decoder of Figure 8.6 can select eight 1K RAM chips. Also, this method does not generate any bus conflict. This is because the selected decoder output ensures enabling of one memory chip at a time.

As mentioned before, a Programmable Logic Device (PLD) is similar to a ROM in concept except that it does not provide full decoding of the input lines. Instead, a PLD provides a partial sum of products that can be obtained via programming and saves a lot of space on the board. For example, a PAL chip contains a fused programmable AND array and a fixed OR array. Note that both AND and OR arrays are programmable in a PLA. The AND and OR gates are fabricated inside the PLD without interconnections. The specific functions desired are implemented during programming via software. For example, programming of the PAL provides connections of the AND gates to the inputs of the OR gates. Therefore, the PAL implements the sum of the products of the inputs. PLDs are used extensively these days with 32- and 64-bit microprocessors such as the Intel 80386/80486/Pentium and Motorola 68030/68040/PowerPC for performing the memory decode function. PLDs connect these microprocessors to memory, I/O devices, and other chips without the use of any additional logic gates or circuits.

8.1.3 Virtual Memory and Memory Management concepts

Due to the massive amount of information that must be saved in most systems, the mass storage device is often a disk. If each access is to a disk (even a hard disk), then system throughput will be reduced to unacceptable levels.

An obvious solution is to use a large and fast locally accessed semiconductor memory. Unfortunately the storage cost per bit for this solution is very high. A combination

Address Range in Hexadecimal	RAM Chip Number
0000-03FF	I
0400-07FF	II
0800-0BFF	III
0C00-0FFF	IV

FIGURE 8.7 Address map of the memory organization of Figure 8.6

of both off-board disk (secondary memory) and on-board semiconductor main memory must be designed into a system. This requires a mechanism to manage the two-way flow of information between the primary (semiconductor) and secondary (disk) media. This mechanism must be able to transfer blocks of data efficiently, keep track of block usage, and replace them in a nonarbitrary way. The main memory system must, therefore, be able to dynamically allocate memory space.

An operating system must have resource protection from corruption or abuse by users. Users must be able to protect areas of code from each other while maintaining the ability to communicate and share other areas of code. All these requirements indicate the need for a device, located between the microprocessor and memory, to control accesses, perform address mappings, and act as an interface between the logical (Programmer's memory) and the physical (Microprocessor's directly addressable memory) address spaces. Because this device must manage the memory use configuration, it is appropriately called the "memory management unit (MMU)." Typical 32-bit processors such as the Motorola 68030/68040 and the Intel 80486/Pentium include on-chip MMUs. The MMU reduces the burden of the memory management function of the operating system.

The basic functions provided by the MMU are address translation and protection. The MMU translates logical program addresses to physical memory address. Note that in assembly language programming, addresses are referred to by symbolic names. These addresses in a program are called logical addresses because they indicate the logical positions of instructions and data. The MMU translates these logical addresses to physical addresses provided by the memory chips. The MMU can perform address translation in one of two ways:

1. By using the substitution technique as shown in Figure 8.8(a)
2. By adding an offset to each logical address to obtain the corresponding physical address as shown in Figure 8.8(b)

Address translation using the substitution technique is faster than the offset method. However, the offset method has the advantage of mapping a logical address to any physical address as determined by the offset value.

Memory is usually divided into small manageable units. The terms "page" and "segment" are frequently used to describe these units. Paging divides the memory into equal-sized pages; segmentation divides the memory into variable-sized segments. It is relatively easier to implement the address translation table if the logical and main memory spaces are divided into pages.

There are three ways to map logical addresses to physical addresses: paging,

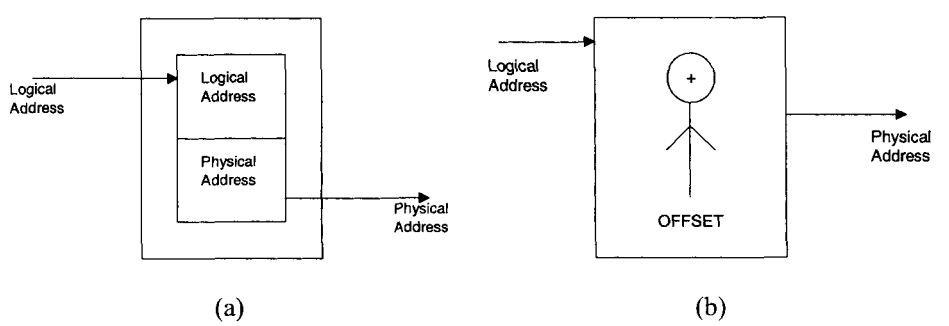


FIGURE 8.8 (a) Address translation using the substitution technique; (b) Address translation by the offset technique

segmentation, and combined paging/segmentation. In a paged system, a user has access to a larger address space than physical memory provides. The virtual memory system is managed by both hardware and software. The hardware included in the memory management unit handles address translation. The memory management software in the operating system performs all functions including page replacement policies to provide efficient memory utilization. The memory management software performs functions such as removal of the desired page from main memory to accommodate a new page, transferring a new page from secondary to main memory at the right instant of time, and placing the page at the right location in memory.

If the main memory is full during transfer from secondary to main memory, it is necessary to remove a page from main memory to accommodate the new page. Two popular page replacement policies are first-in–first-out (FIFO) and least recently used (LRU). The FIFO policy removes the page from main memory that has been resident in memory for the longest amount of time. The FIFO replacement policy is easy to implement, but one of its main disadvantages is that it is likely to replace heavily used pages. Note that heavily used pages are resident in main memory for the longest amount of time. Sometimes this replacement policy might be a poor choice. For example, in a time-shared system, several users normally share a copy of the text editor in order to type and correct programs. The FIFO policy on such a system might replace a heavily used editor page to make room for a new page. This editor page might be recalled to main memory immediately. The FIFO, in this case, would be a poor choice. The LRU policy, on the other hand, replaces the page that has not been used for the longest amount of time.

In the segmentation method, the MMU utilizes the segment selector to obtain a descriptor from a table in memory containing several descriptors. A descriptor contains the physical base address for a segment, the segment's privilege level, and some control bits. When the MMU obtains a logical address from the microprocessor, it first determines whether the segment is already in the physical memory. If it is, the MMU adds an offset component to the segment base component of the address obtained from the segment descriptor table to provide the physical address. The MMU then generates the physical address on the address bus for selecting the memory. On the other hand, if the MMU does not find the logical address in physical memory, it interrupts the microprocessor. The microprocessor executes a service routine to bring the desired program from a secondary memory such as disk to the physical memory. The MMU determines the physical address using the segment offset and descriptor as described earlier and then generates the physical address on the address bus for memory. A segment will usually consist of an integral number of pages, each, say, 256 bytes long. With different-sized segments being swapped in and out, areas of valuable primary memory can become unusable. Memory is unusable for segmentation when it is sandwiched between already allocated segments and if it is not

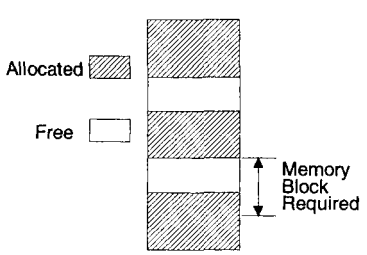


FIGURE 8.9 Memory fragmentation (external)

large enough to hold the latest segment that needs to be loaded. This is called "external fragmentation" and is handled by MMUs using special techniques. An example of external fragmentation is given in Figure 8.9. The advantages of segmented memory management are that few descriptors are required for large programs or data spaces and that internal fragmentation (to be discussed later) is minimized. The disadvantages include external fragmentation, the need for involved algorithms for placing data, possible restrictions on the starting address, and the need for longer data swap times to support virtual memory.

Address translation using descriptor tables offers a protection feature. A segment or a page can be protected from access by a program section of a lower privilege level. For example, the selector component of each logical address includes one or two bits indicating the privilege level of the program requesting access to a segment. Each segment descriptor also includes one or two bits providing the privilege level of that segment. When an executing program tries to access a segment, the MMU can compare the selector privilege level with the descriptor privilege level. If the segment selector has the same or higher privilege level, then the MMU permits the access. If the privilege level of the selector is lower than that of the descriptor, the MMU can interrupt the microprocessor, informing it of a privilege-level violation. Therefore, the indirect technique of generating a physical address provides a mechanism of protecting critical program sections in the operating system. Because paging divides the memory into equal-sized pages, it avoids the major problem of segmentation—external fragmentation. Because the pages are of the same size, when a new page is requested and an old one swapped out, the new one will always fit into the vacated space. However, a problem common to both techniques remains—internal fragmentation.

Internal fragmentation is a condition where memory is unused but allocated due to memory block size implementation restrictions. This occurs when a module needs, say, 300 bytes and page is 1K bytes, as shown in Figure 8.10

In the paged-segmentation method, each segment contains a number of pages. The logical address is divided into three components: segment, page, and word. The segment component defines a segment number, the page component defines the page within the segment, and the word component provides the particular word within the page. A page component of n bits can provide up to 2^n pages. A segment can be assigned with one or more pages up to maximum of 2^n pages; therefore, a segment size depends on the number of pages assigned to it.

A protection mechanism can be assigned to either a physical address or a logical address. Physical memory protection can be accomplished by using one or more protection bits with each block to define the access type permitted on the block. This means that

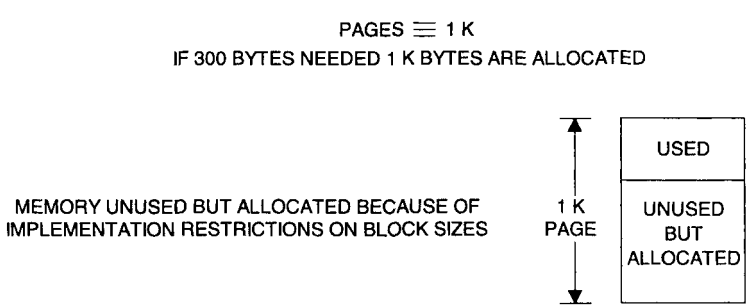


FIGURE 8.10 Memory fragmentation (internal)

each time a page is transferred from one block to another, the block protection bits must be updated. A more efficient approach is to provide a protection feature in logical address space by including protection bits in descriptors of the segment table in the MMU.

Virtual memory is the most fundamental concept implemented by a system that performs memory-management functions such as space allocation, program relocation, code sharing and protection.The key idea behind this concept is to allow a user program to address more locations than those available in a physical memory. An address generated by a user program is called a virtual address. The set of virtual addresses constitutes the virtual address space. Similarly, the main memory of a computer contains a fixed number of addressable locations and a set of these locations forms the physical address space. The basic hardware for virtual memory is implemented in modern microprocessors as an on-chip feature. These contemporary processors support both cache and virtual memories. The virtual addresses are typically converted to physical addresses and then applied to cache.

In the early days, when a programmer used to write a large program that could not fit into the main memory, it was necessary to divide the program into small portions so each one could fit into the primary memory. These small portions are called overlays. A programmer has to design overlays so that they are independent of each other. Under these circumstances, one can successively bring each overlay into the main memory and execute them in a sequence.

Although this idea appears to be simple, it increases the program-development time considerably.

However, in a system that uses a virtual memory, the size of the virtual address space is usually much larger than the available physical address space. In such a system, a programmer does not have to worry about overlay design, and thus a program can be written assuming a huge address space is available. In a virtual memory system, the programming effort can be greatly simplified. However, in reality, the actual number of physical addresses available is considerably less than the number of virtual addresses provided by the system. There should be some mechanism for dividing a large program into small overlays automatically. A virtual memory system is one that mechanizes the process of overlay generation by performing a series of mapping operations.

A virtual memory system may be configured in one of the following ways:

- Paging systems
- Segmentation systems

In a paging system, the virtual address space is divided into equal-size blocks called pages. Similarly, the physical memory is also divided into equal-size blocks called frames. The size of a page is the same as the size of a frame. The size of a page may be 512, 1024 or 2048 words.

In a paging system, each virtual address may be regarded as an ordered pair (p, n), where p is the page number and n is the word number within the page p. Sometimes the quantity n is referred to as the displacement, or offset. A user program may be regarded as a sequence of pages, and a complete copy of the program is always held in a backup store such as a disk. A page p of the user program can be placed in any available page frame p' of the main memory. A program may access a page if the page is in the main memory. In a paging scheme, pages are brought from secondary memory and are stored in main memory in a dynamic manner. All virtual addresses generated by a user program must be translated into physical memory addresses. This process is known as dynamic address translation and is shown in Figure 8.11.

When a running program accesses a virtual memory location v = (p, n), the

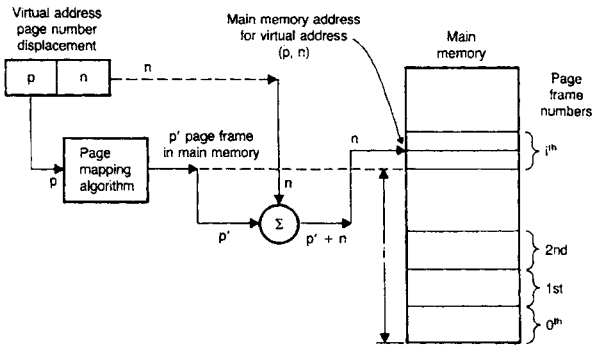


FIGURE 8.11 Paging Systems—Virtual versus Main Memory Mapping

mapping algorithm finds that the virtual page p is mapped to the physical frame p'. The physical address is then determined by appending p' to n.

This dynamic address translator can be implemented using a page table. In most systems, this table is maintained in the main memory. It will have one entry for each virtual page of the virtual address space. This is illustrated in the following example.

Example 8.1

Design a mapping scheme with the following specifications:

- Virtual address space = 32K words
- Main memory size = 8K words
- Page size = 2K words
- Secondary memory address = 24 bits

Solution

32K words can be divided into 16 virtual pages with 2K words per page, as follows:

VIRTUAL ADDRESS	PAGE NUMBER
0-2047	0
2048-4095	1
4096-6143	2
6144-8191	3
8192-10239	4
10240-12287	5
12288-14335	6
14336-16383	7
16384-18431	8
18432-20479	9
20480-22527	10
22528-24575	11
24576-26623	12

26624-28671	13
28672-30719	14
30720-32767	15

Since there are 8K words in the main memory, 4 frames with 2K words per frame are available:

PHYSICAL ADDRESS	FRAME NUMBER
0-2047	0
2048-4095	1
4096-6143	2
6144-8191	3

Since there are 32K addresses in the virtual space, 15 bits are required for the virtual address. Because there are 16 virtual pages, the page map table contains 16 entries. The 4 most-significant bits of the virtual address are used as an index to the page map table, and the remaining 11 bits of the virtual address are used as the displacement to locate a word within the page frame. Each entry of the page table is 32 bits long. This can be obtained as follows:

1 bit for determining whether the page table is in main memory or not (residence bit).
2 bits for main memory page frame number.
24 bits for secondary memory address
5 bits for future use. (Unused)
32 bits total

The complete layout of the page table is shown in Figure 8.12. Assume the virtual address generated is 0111 000 0010 1101. From this, compute the following:

Virtual page number = 7_{10}
Displacement = 43_{10}

From the page-map table entry corresponding to the address 0111, the page can be found in the main memory (since the page resident bit is 1).

The required virtual page is mapped to main memory page frame number 2. Therefore, the actual physical word is the 43rd word in the second page frame of the main memory.

So far, a page referenced by a program is assumed always to be found in the main memory. In practice, this is not necessarily true. When a page needed by a program is not assigned to the main memory, a page fault occurrs. A page fault is indicated by an interrupt, and when this interrupt occurs, control is transferred to a service routine of the operating system called the page-fault handler. The sequence of activities performed by the page-fault handler are summarized as follows:

- The secondary memory address of the required page p is located from the page table.
- Page p from the secondary memory is transferred into one of the available main memory frames by performing a block-move operation.
- The page table is updated by entering the frame number where page p is loaded and by setting the residence bit to 1 and the change bit to 0.

When a page-fault handler completes its task, control is transferred to the user program, and the main memory is accessed again for the required data or instruction. All

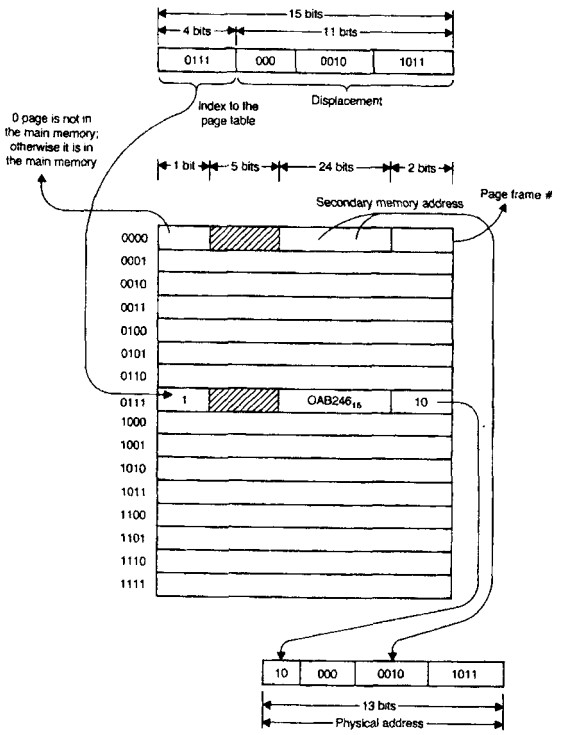


FIGURE 8.12 Mapping Scheme for the Paging System of Example 8.1

these activities are kept hidden from a user. Pages are transferred to main memory only at specified times. The policy that governs this decision is known as the fetch policy. Similarly, when a page is to be transferred from the secondary memory to main memory, all frames may be full. In such a situation, one of the frames has to be removed from the main memory to provide room for an incoming page. The frame to be removed is selected using a replacement policy. The performance of a virtual memory system is dependent upon the fetch and replacement strategies. These issues are discussed later.

The paging concept covered so far is viewed as a one-dimensional technique because the virtual addresses generated by a program may linearly increase from 0 to some maximum value M. There are many situations where it is desirable to have a multidimensional virtual address space. This is the key idea behind segmentation systems.

Each logical entity such as a stack, an array, or a subroutine has a separate virtual address space in segmentation systems. Each virtual address space is called a segment, and each segment can grow from zero to some maximum value. Since each segment refers to a separate virtual address space, it can grow or shrink independently without affecting other segments.

In a segmentation system, the details about segments are held in a table called a segment table. Each entry in the segment table is called a segment descriptor, and it typically includes the following information:

- Segment base address b (starting address of the segment in the main memory)
- Segment length *l* (size of a segment)

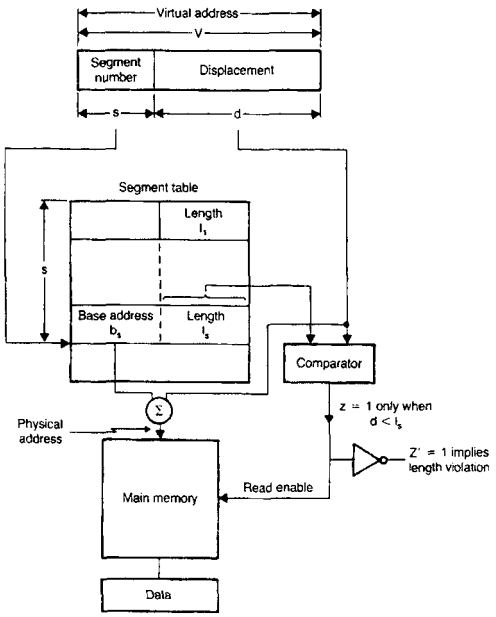


FIGURE 8.13 Address Translation in a Segmentation System. (Note that $\overline{Z} = Z'$)

- Segment presence bit
- Protection bits

From the structure of a segment descriptor, it is possible to create two or more segments whose sizes are different from one another. In a sense, a segmentation system becomes a paging system if all segments are of equal length. Because of this similarity, there is a close relationship between the paging and segmentation systems from the viewpoint of address translation.

A virtual address, V, in a segmentation system is regarded as an ordered pair (s, d), where s is the segment number and d is the displacement within segment s. The address translator for a segmentation system can be implemented using a segment table, and its organization is shown in Figure 8.13.

The details of the address translation process is briefly discussed next.

Let V be the virtual address generated by the user program. First, the segment number field, s, of the virtual address V is used as an index to the segment table. The base address and length of this segment are b_s and l_s, respectively. Then, the displacement d of the virtual address V is compared with the length of the segment l_s to make sure that the required address lies within the segment. If d is less than or equal to l_s, then the comparator output Z will be high. When $d \leq l_s$, the physical address is formed by adding b_s and d. From this physical address, data is retrieved and transferred to the CPU. However, when $d > l_s$, the required address lies out of the segment range, and thus an address out of range trap will be generated. A trap is a nonmaskable interrupt with highest priority.

In a segmentation system, a segment needed by a program may not reside in main memory. This situation is indicated by a bit called a valid bit. A valid bit serves the same purpose as that of a page resident bit, and thus it is regarded as a component of the segment descriptor. When the valid bit is reset to 0, it may be concluded that the required segment is not in main memory.

This means that its secondary memory address must be included in the segment descriptor. Recall that each segment represents a logical entity. This implies that we can protect segments with different protection protocols based on the logical contents of the segment. The following are the common protection protocols used in a segmentation system:

- Read only
- Execute only
- Read and execute only
- Unlimited access
- No access

Thus it follows that these protection protocols have to be encoded into some protection codes and these codes have to be included in a segment descriptor.

In a segmented memory system, when a virtual address is translated into a physical address, one of the following traps may be generated:

- Segment fault trap is generated when the required segment is not in the main memory.
- Address violation trap occurs when d $>l_s$.
- Protection violation trap is generated when there is a protection violation.

When a segment fault occurs, control will be transferred to the operating system. In response, the operating system has to perform the following activities:

- First, it finds the secondary memory address of the required segment from its segment descriptor.
- Next, it transfers the required segment from the secondary to primary memory.
- Finally, it updates the segment descriptor to indicate that the required segment is in the main memory.

After performing the preceding activities, the operating system transfers control to the user program and the data or instruction retrieval or write operation is repeated.

A comparison of the paging and segmentation systems is provided next. The primary idea behind a paging system is to provide a huge virtual space to a programmer, allowing a programmer to be relieved from performing tedious memory-management tasks such as overlay design. The main goal of a segmentation system is to provide several virtual address spaces, so the programmer can efficiently manage different logical entities such as a program, data, or a stack.

The operation of a paging system can be kept hidden at the user level. However, a programmer is aware of the existence of a segmented memory system.

To run a program in a paging system, only its current page is needed in the main memory. Several programs can be held in the main memory and can be multiplexed. The paging concept improves the performance of a multiprogramming system. In contrast, a segmented memory system can be operated only if the entire program segment is held in the main memory.

In a paging system, a programmer cannot efficiently handle typical data structures such as stacks or symbol tables because their sizes vary in a dynamic fashion during program execution. Typically, large pages for a symbol table or small pages for a stack cannot be created. In a segmentation system, a programmer can treat these two structures as two logical entities and define the two segments with different sizes.

The concept of segmentation encourages people to share programs efficiently. For example, assume a copy of a matrix multiplication subroutine is held in the main memory. Two or more users can use this routine if their segment tables contain copies of

the segment descriptor corresponding to this routine. In a paging system, this task cannot be accomplished efficiently because the system operation is hidden from the user. This result also implies that in a segmentation system, the user can apply protection features to each segment in any desired manner. However, a paging system does not provide such a versatile protection feature.

Since page size is a fixed parameter in a paging system, a new page can always be loaded in the space used by a page being swapped out. However, in a segmentation system with uneven segment sizes, there is no guarantee that an incoming segment can fit into the free space created by a segment being swapped out.

In a dynamic situation, several programs may request more space, whereas some other programs may be in the process of releasing the spaces used by them. When this happens in a segmented memory system, there is a possibility that uneven-sized free spaces may be sparsely distributed in the physical address space. These free spaces are so irregular in size that they cannot normally be used to satisfy any new request. This is called an external fragmentation, and an operating system has to merge all free spaces to form a single large useful segment by moving all active segments to one end of the memory. This activity is known as memory compaction. This is a time-consuming operation and is a pure overhead. Since pages are of equal size, no external fragmentation can occur in a paging system.

In a segmented memory system, a programmer defines a segment, and all segments are completely filled.

The page size is decided by the operating system, and the last page of a program may not be filled completely when a program is stored in a sequence of pages. The space not filled in the last page cannot be used for any other program. This difficulty is known as internal fragmentation—a potential disadvantage of a paging system.

In summary, the paging concept simplifies the memory-management tasks to be performed by an operating system and therefore, can be handled efficiently by an operating system. The segmentation approach is desirable to programmers when both protection and

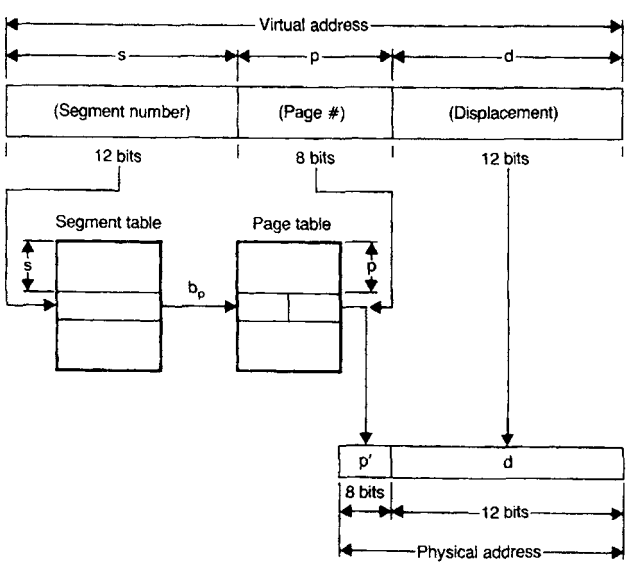


FIGURE 8.14 Address-translation Scheme for a Paged-segmentation System

sharing of logical entities among a group of programmers are required.

To take advantage of both paging and segmentation, some systems use a different approach, in which these concepts are merged. In this technique, a segment is viewed as a collection of pages. The number of pages per segment may vary. However, the number of words per page still remains fixed. In this situation, a virtual address V is an ordered triple (s, p, d), where s is the segment number and p and d are the page number and the displacement within a page, respectively.

The following tables are used to translate a virtual address into a physical address:

- Page table: This table holds pointers to the physical frames.
- Segment table: Each entry in the segment table contains the base address of the page table that holds the details about the pages that belong to the given segment.

The address-translation scheme of such a paged-segmentation system is shown in Figure 8.14:

- First, the segment number s of the virtual address is used as an index to the segment table, which leads to the base address b_p of the page table.
- Then, the page number p of the virtual address is used as an index to the page table, and the base address of the frame number p' (to which the page p is mapped) can be found.
- Finally, the physical memory address is computed by adding the displacement d of the virtual address to the base address p' obtained before.

To illustrate this concept, the following numerical example is provided.

Example 8.2

Assume the following values for the system of Figure 8.14:

- Length of the virtual address field =32 bits
- Length of the segment number field =12 bits
- Length of the page number field = 8 bits
- Length of the displacement field =12 bits

Now, determine the value of the physical address using the following information:

- Value of the virtual address field = $000FA0BA_{16}$
- Contents of the segment table address $(000)_{16} = 0FF_{16}$
- Contents of the page table address $(1F9_{16}) = AC_{16}$

Solution

From the given virtual address, the segment table address is 000_{16} (three high-order hexadecimal digits of the virtual address). It is given that the contents of this segment-able address is $0FF_{16}$. Therefore, by adding the page number p (fourth and fifth hexadecimal digits of the virtual address) with $0FF_{16}$, the base address of the page table can be determined as:

$$0FF_{16} + FA_{16} = 1F9_{16}$$

Since the contents of the page table address $1F9_{16}$ is AC_{16}, the physical address can be obtained by adding the displacement (low-order three hexadecimal digits of the virtual address) with AC_{16} as follows:

$AC000_{16} + 000BA_{16} = AC0BA_{16}$

In this addition, the displacement value 0BA is sign-extended to obtain a 20-bit number that can be directly added to the base value p'. The same final answer can be obtained if p'

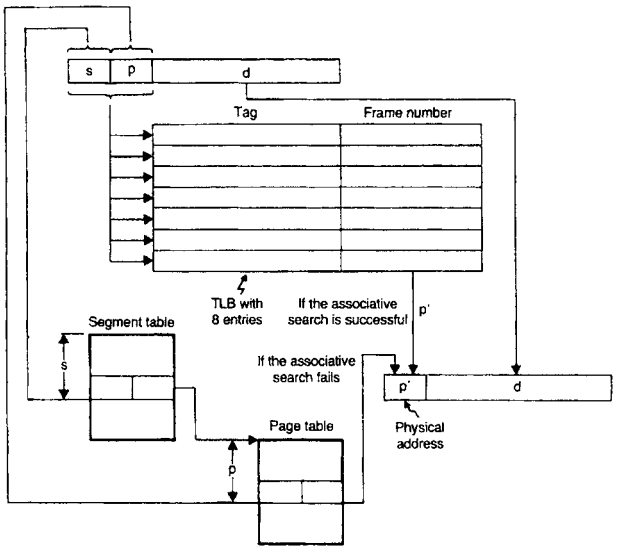


FIGURE 8.15 Address Translation Using a TLB

and d are first concatenated. Thus, the value of the physical address is $AC0BA_{16}$.

The virtual space of some computers use both paging and segmentation, and it is called a linear segmented virtual memory system. In this system, the main memory is accessed three times to retrieve data (one for accessing the page table; one for accessing the segment table; and one for accessing the data itself).

Accessing the main memory is a time-consuming operation. To speed up the retrieval operation, a small associative memory (implemented as an on-chip hardware in modern microprocessors) called the translation lookaside buffer (TLB) is used. The TLB stores the translation information for the 8 or 16 most recent virtual addresses. The organization of a address translation scheme that includes a TLB is shown in Figure 8.15.

In this scheme, assume the TLB is capable of holding the translation information about the 8 most recent virtual addresses.

The pair (s, p) of the virtual address is known as a tag, and each entry in the TLB is of the form:

(s,p) or tag	Base address of the frame p'

When a user program generates a virtual address, the (s, p) pair is associatively compared with all tags held in the TLB for a match. If there is a match, the physical address is formed by retrieving the base address of the frame p' from the TLB and concatenating this with the displacement d. However, in the event of a TLB miss, the physical address is generated after accessing the segment and page tables, and this information will also be loaded in the TLB. This ensures that translation information pertaining to a future reference is confined to the TLB. To illustrate the effectiveness of the TLB, the following numerical example is provided.

Example 8.3

The following measurements are obtained from a computer system that uses a linear segmented memory system with a TLB:

- Number of entries in the TLB =16
- Time taken to conduct an associative search in the TLB = 160 ns
- Main memory access time = 1 μs

Determine the average access time assuming a TLB hit ratio of 0.75.

Solution

In the event of a TLB hit, the time needed to retrieve the data is:

t1 = TLB search time + time for one memory access
= 160 ns + 1 μs
= 1.160 μs

However, when a TLB miss occurs, the main memory is accessed three times to retrieve the data. Therefore, the retrieval time t2 in this case is

t2 = TLB search time + 3 (time for one memory access)
= 160 ns + 3 μs
= 3.160 μs

The average access time,

t_{av} = ht1 + (1 - h)t2

where h is the TLB hit ratio.

The average access time t_{av} = 0.75 (1.6) + 0.25 (3.160) μsec
= 1.2 + 0.79 μsec
=1.99 μsec

This example shows that the use of a small TLB significantly improves the efficiency of the retrieval operation (by 33%). There are two main reasons for this improvement. First, the TLB is designed using the associated memory. Second, the TLB hit ratio may be attributed to the locality of reference. Simulation studies indicate that it is possible to achieve a hit ratio in the range of 0.8 to 0.9 by having a TLB with 8 to 16 entries.

In a computer based on a linear segmented virtual memory system, the performance parameters such as storage use are significantly influenced by the page size p. For instance, when p is very large, excessive internal fragmentation will occur. If p is small, the size of the page table becomes large. This results in poor use of valuable memory space. The selection of the page size p is often a compromise. Different computer systems use different page sizes. In the following, important memory-management strategies are described. There are three major strategies associated with the management:

- Fetch strategies
- Placement strategies
- Replacement strategies

All these strategies are governed by a set of policies conceived intuitively. Then they are validated using rigorous mathematical methods or by conducting a series of simulation experiments. A policy is implemented using some mechanism such as hardware, software, or firmware.

Fetch strategies deal with when to move the next page to main memory. Recall that when a page needed by a program is not in the main memory, a page fault occurs. In the event of a page fault, the page-fault handler will read the required page from the secondary memory and enter its new physical memory location in the page table, and the instruction execution continues as though nothing has happened.

In a virtual memory system, it is possible to run a program without having any page in the primary memory. In this case, when the first instruction is attempted, there is a page fault. As a consequence, the required page is brought into the main memory, where

the instruction execution process is repeated again. Similarly, the next instruction may also cause a page fault. This situation is handled exactly in the same manner as described before. This strategy is referred to as demand paging because a page is brought in only when it is needed. This idea is useful in a multiprogramming environment because several programs can be kept in the main memory and executed concurrently.

However, this concept does not give best results if the page fault occurs repeatedly. For instance, after a page fault, the page-fault handler has to spend a considerable amount of time to bring the required page from the secondary memory. Typically, in a demand paging system, the effective access time t_{av} is the sum of the main memory access time t and μ, where μ is the time taken to service a page fault. Example 8.4 illustrates the concept.

Example 8.4

(a) Assuming that the probability of a page fault occurring is p, derive an expression for t_{av} in terms of t, μ, and p.

(b) Suppose that t = 500 ns and μ = 30 ms, calculate the effective access time t_{av} if it is given that on the average, one out of 200 references results in a page fault.

Solution

(a) If a page fault does not occur, then the desired data can be accessed within a time t. (From the hypothesis the probability for a page fault not to occur is $1 - p$). If the page fault occurs, then μ time units are required to access the data. The effective access time is

$$t_{av} = (1 - p)t + p\mu$$

(b) Since it is given that one out of every 200 references generates a page fault, p = 1/200. Using the result derived in part (a):

$$t_{av} = [(1 - 0.005) \times 0.5 + 0.005 \times 30{,}000]\ \mu s$$
$$= [0.995 \times 0.5 + 150]\ \mu s = [0.4975 + 150]\ \mu s$$
$$= 150.4975\ \mu s$$

These parameters have a significant impact on the performance of a time-sharing system.

As an alternative approach, anticipatory fetching can be adapted. This conclusion is based on the fact that in a short period of time addresses referenced by a program are

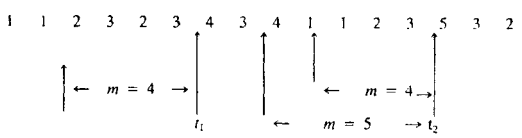


FIGURE 8.16 Stream of Page References

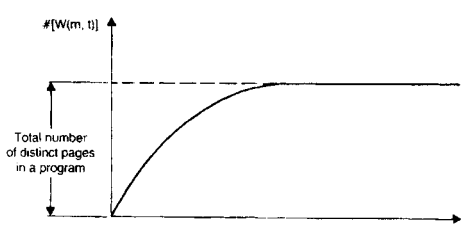


FIGURE 8.17 Relationship between One Cardinality of the Working Set and the Window Size m

clustered around a particular region of the address space. This property is known as locality of reference.

The working set of a program W(m, t) is defined as the set of m most recently needed pages by the program at some instant of time t. The parameter m is called the window of the working set. For example, consider the stream of references shown in Figure 8.16:

From this figure, determine that:

$W(4, t_1) = (2, 3)$ $W(4, t_2) = \{1, 2, 3\}$ $W(5, t_2) = \{1,2,3,4\}$

In general, the cardinality of the set W(0, t) is zero, and the cardinality of the set W(∞, t) is equal to the total number of distinct pages in the program. Since m + 1 most-recent page references include m most-recent page references:

$$\#[W(m + 1, t)] \subseteq \#[W(m, t)]$$

In this equation, the symbol # is used to indicate the cardinality of the set W(m, t). When m is varied from 0 to ∞, #W(m, t) increases exponentially. The relationship between m and #W(m, t) is shown in Figure 8.17.

In practice, the working set of program varies slowly with respect to time. Therefore, the working set of a program can be predicted ahead of time. For example, in a multiprogramming system, when the execution of a suspended program is resumed, its present working set can be reasonably estimated based on the value of its working set at the time it was suspended. If this estimated working set is loaded, page faults are less likely to occur. This anticipatory fetching further improves the system performance because the working set of a program can be loaded while another program is being executed by the CPU. However, the accuracy of a working set model depends on the value of m. Larger values of m result in more-accurate predictions. Typical values of m lie in the range of 5000 to 10,000.

To keep track of the working set of a program, the operating system has to perform time-consuming housekeeping operations. This activity is pure overhead, and thus the system performance may be degraded.

Placement strategies are significant with segmentation systems, and they are concerned with where to place an incoming program or data in the main memory. The following are the three widely used placement strategies:

- First-fit technique
- Best-fit technique
- Worst-fit technique

The first-fit technique places the program in the first available free block or hole that is adequate to store it. The best-fit technique stores the program in the smallest free hole of all the available holes able to store it. The worst-fit technique stores the program in the largest free hole. The first-fit technique is easy to implement and does not have to scan the entire space to place a program. The best-fit technique appears to be efficient because it finds an optimal hole size. However, it has the following drawbacks:

- It is very difficult to implement.
- It may have to scan the entire free space to find the smallest free hole that can hold the incoming program. Therefore, it may be time-consuming.
- It has the tendency continuously to divide the holes into smaller sizes. These smaller holes may eventually become useless.

Worst-fit strategy is sometimes used when the design goal is to avoid creating small holes. In general, the operating system maintains a list known as the available space list (ASL) to indicate the free memory space. Typically, each entry in this list includes the following information:

- Starting address of the free block
- Size of the free block

After each allocation or release, the operating system updates the ASL. In the following example, the mechanics of the various placement strategies presented earlier are explained.

Example 8.5

The available space list of a computer memory system is specified as follows:

STARTING ADDRESS	BLOCK SIZE (IN WORDS)
100	50
200	150
450	600
1,200	400

Determine the available space list after allocating the space for the stream of requests consisting of the following block sizes:

25, 100, 250, 200, 100, 150

a) Use the first-fit method.

b) Use the best-fit method.

c) Use the worst-fit method.

Solution

a) First-fit method. Consider the first request with a block size of 25. Examination of the block sizes of the available space list reveals that this request can be satisfied by allocating from the first available block. The block size (50) is the first of the available space list and is adequate to hold the request (25 blocks). Therefore, the first request with 25 blocks will be allocated from the available space list starting at address 100 with a block size of 50. Request 1 will be allocated starting at an address of 100 ending at an address 100 + 24 = 124 (25 locations including 100). Therefore, the first block of the available space list will start at 125 with a block size of 25. The starting address and block size of each request can be calculated similarly.

b) Best-fit method. Consider request 1. Examination of the available block size reveals that this request can be satisfied by allocating from the first smallest available block capable of holding it. Request 1 will be allocated starting at address 100 and ending at 124. Therefore, the available space list will start at 125 with a block size of 25.

c) Worst-fit method. Consider request 1. Examination of the available block sizes reveals that this request can be satisfied by allocating from the third block (largest) starting at 450. After this allocation the starting address of the available list will be 500 instead of 450 with a block size of 600 - 25 = 575. Various results for all the other requests are shown in Figure 8.18.

In a multiprogramming system, programs of different sizes may reside in the main memory. As these programs are completed, the allocated memory space becomes free. It may happen that these unused free spaces, or holes, become available between two allocated blocks, or partitions. Some of these holes may not be large enough to satisfy the memory request of a program waiting to run. Thus valuable memory space may be wasted. One way to get around this problem is to combine adjacent free holes to make the hole size larger and usable by other jobs. This technique is known as coalescing of holes.

It is possible that the memory request made by a program may be larger than

	Request 1 (25)		Request 2 (100)		Request 3 (250)		Request 4 (200)		Request 5 (100)		Request 6 (150)	
	Start address	Block size	Start address	Block size	Start address	Block size	Start address	Block size	Start address	Block size	Start address	Block size
First fit	125	25	125	25	125	25	125	25	125	25	125	25
	200	150	300	50	300	50	300	50	300	50	300	50
	450	600	450	600	700	350	900	150	1000	50	1000	50
	1200	400	1200	400	1200	400	1200	400	1200	400	1350	250
Best fit	125	25	125	25	125	25	125	25	125	25	125	25
	200	150	300	50	300	50	300	50	300	50	300	50
	450	600	450	600	450	600	650	400	650	400	800	250
	1200	400	1200	400	1450	150	1450	150	1550	50	1550	50
Worst fit	100	50	100	50	100	50	100	50	100	50	100	50
	200	150	200	150	200	150	200	150	200	150	200	150
	500	575	600	475	850	225	850	225	950	125	850	125
	1200	400	1200	400	1200	400	1400	200	1400	200	1550	50

FIGURE 8.18 Memory Map after Allocating Space for All Requests Given Example Using Different Placement Strategies

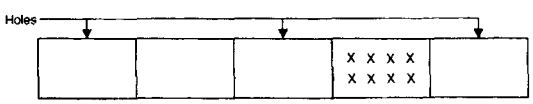


FIGURE 8.19 Memory Status before Compaction

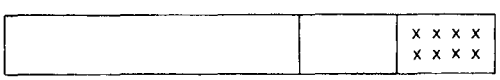


FIGURE 8.20 Memory Status after Compaction

any free hole but smaller than the combined total of all available holes. If the free holes are combined into one single hole, the request can be satisfied. This technique is known as memory compaction. For example, the status of a computer memory before and after memory compaction is shown in Figures 8.19 and 8.20, respectively.

Placement strategies such as first-fit and best-fit are usually implemented as software procedures. These procedures are included in the operating system's software. The advent of high-level languages such as Pascal and C greatly simplify the programming effort because they support abstract data objects such as pointers. The available space list discussed in this section can easily be implemented using pointers.

The memory compaction task is performed by a special software routine of the operating system called a garbage collector. Normally, an operating system runs the garbage collector routine at regular intervals.

In a paged virtual memory system, when no frames are vacant, it is necessary

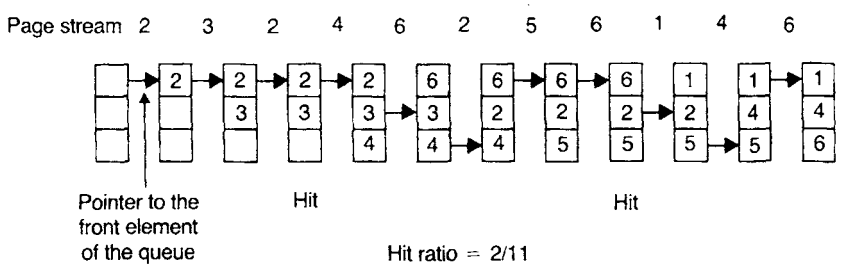


FIGURE 8.21 Hit Ratio Computation for Example 8.6

to replace a current main memory page to provide room for a newly fetched page. The page for replacement is selected using some replacement policy. An operating system implements the chosen replacement policy. In general, a replacement policy is considered efficient if it guarantees a high hit ratio. The hit ratio h is defined as the ratio of the number of page references that did not cause a page fault to the total number of page references.

The simplest of all page replacement policies is the FIFO policy. This algorithm selects the oldest page (or the page that arrived first) in the main memory for replacement. The hit ratio h for this algorithm can be analytically determined using some arbitrary stream of page references as illustrated in the following example.

Example 8.6

Consider the following stream of page requests.

2, 3, 2, 4, 6, 2, 5, 6, 1, 4, 6

Determine the hit ratio h for this stream using the FIFO replacement policy. Assume the main memory can hold 3 page frames and initially all of them are vacant.

Solution

The hit ratio computation for this situation is illustrated in Figure 8.21.

From Figure 8.21, it can be seen that the first two page references cause page faults. However, there is a hit with the third reference because the required page (page 2) is already in the main memory. After the first four references, all main memory frames are completely used. In the fifth reference, page 6 is required. Since this page is not in the main memory, a page fault occurs. Therefore, page 6 is fetched from the secondary memory. Since there are no vacant frames in the main memory, the oldest of the current main memory pages is selected for replacement. Page 6 is loaded in this position. All other data tabulated in this figure are obtained in the same manner. Since 9 out of 11 references generate a page fault, the hit ratio is 2/11.

The primary advantage of the FIFO algorithm is its simplicity. This algorithm can be implemented by using a FIFO queue. FIFO policy gives the best result when page references are made in a strictly sequential order. However, this algorithm fails if a program loop needs a variable introduced at the beginning. Another difficulty with the FIFO algorithm is it may give anomalous results.

Intuitively, one may feel that an increase in the number of page frames will also increase the hit ratio. However, with FIFO, it is possible that when the page frames are increased, there is a drop in the hit ratio. Consider the following stream of requests:

1, 2, 3, 4, 5, 1, 2, 5, 1, 2, 3, 4, 5, 6, 5

Assume the main memory has 4 page frames; then using the FIFO policy there is a hit ratio of 4/15. However, if the entire computation is repeated using 5 page frames, there

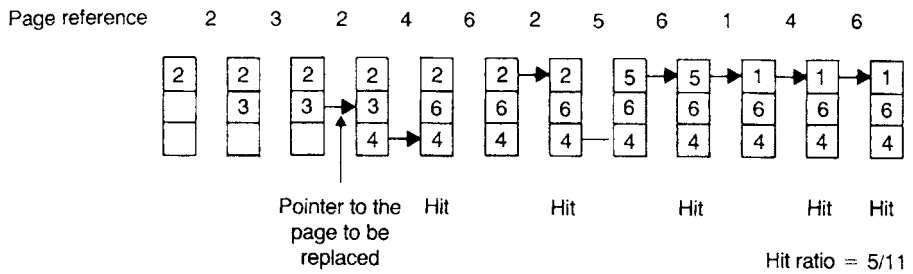


FIGURE 8.22 Hit Ratio Computation for Example 8.7

is a hit ratio of 3/15. This computation is left as an exercise.

Another replacement algorithm of theoretical interest is the optimal replacement policy. When there is a need to replace a page, choose that page which may not be needed again for the longest period of time in the future.

The following numerical example explains this concept.

Example 8.7

Using the optimal replacement policy, calculate the hit ratio for the stream of page references specified in Example 8.6. Assume the main memory has three frames and initially all of them are vacant.

Solution

The hit ratio computation for this problem is shown in Figure 8.22.

From Figure 8.22, it can be seen that the first two page references generate page faults. There is a hit with the sixth page reference, because the required page (page 2) is found in the main memory. Consider the fifth page reference. In this case, page 6 is required. Since this page is not in the main memory, it is fetched from the secondary memory. Now, there are no vacant page frames. This means that one of the current pages in the main memory has to be selected for replacement. Choose page 3 for replacement because this page is not used for the longest period of time. Page 6 is loaded into this position. Following the same procedure, other entries of this figure can be determined. Since 6 out of 11 page references generate a page fault, the hit ratio is 5/11.

The decision made by the optimal replacement policy is optimal because it makes a decision based on the future evolution. It has been proven that this technique does not give any anomalous results when the number of page frames is increased. However, it is not possible to implement this technique because it is impossible to predict the page references well ahead of time. Despite this disadvantage, this procedure is used as a standard to determine the efficiency of a new replacement algorithm. Since the optimal replacement policy is practically unfeasible, some method that approximates the behavior of this policy is desirable. One such approximation is the least recently used (LRU) policy.

According to the LRU policy, the page that is selected for replacement is that page that has not been referenced for the longest period of time. Example 8.8 illustrates this.

Example 8.8

Solve Example 8.7 using the LRU policy.

Solution

The hit ratio computation for this problem is shown in Figure 8.23.

In the figure we again notice that the first two references generate a page fault,

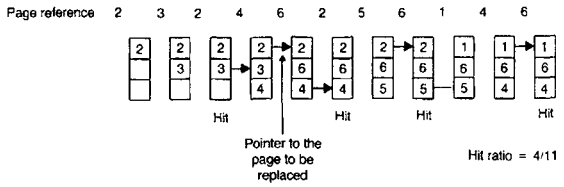


FIGURE 8.23 Hit Ratio Computation for Example 8.9

whereas the third reference is a hit because the required page is already in the main memory. Now, consider what happens when the fifth reference is made. This reference requires page 6, which is not in the memory.

Also, we need to replace one of the current pages in the main memory because all frames are filled. According to the LRU policy, among pages 2, 3, and 4, page 3 is the page that is least recently referenced. Thus we replace this page with page 6. Following the same reasoning the other entries of Figure 8.23 can be determined. Note that 7 out of 11 references generate a page fault; therefore, the hit ratio is 4/11. From the results of the example, we observe that the performance of the LRU policy is very close to that of the optimal replacement policy. Also, the LRU obtains a better result than the FIFO because it tries to retain the pages that are used recently.

Now, let us summarize some important features of the LRU algorithm.

- In principle, the LRU algorithm is similar to the optimal replacement policy except that it looks backward on the time axis. Note that the optimal replacement policy works forward on the time axis.
- If the request stream is first reversed and then the LRU policy is applied to it, the result obtained is equivalent to the one that is obtained by the direct application of the optimal replacement policy to the original request stream.
- It has been proven that the LRU algorithm does not exhibit Belady's anamoly. This is because the LRU algorithm is a stack algorithm. A page-replacement algorithm is said to be a stack algorithm if the following condition holds:

  $$P_t(i) \subset P_t(i + 1)$$

  In the preceding relation the quantity Pt(i) refers to the set of pages in the main memory whose total capacity is i frames at some time t. This relation is called the inclusion property. One can easily demonstrate that FIFO replacement policy is not a stack algorithm. This task is left as an exercise.
- The LRU policy can be easily implemented using a stack. Typically, the page numbers of the request stream are stored in this stack. Suppose that p is the page number being referenced. If p is not in the stack, then p is pushed into the stack. However, if p is in the stack, p is removed from the stack and placed on the top of the stack. The top of the stack always holds the most recently referenced page number, and the bottom of the stack always holds the least-recent page number. To see this clearly, consider Figure 8.24, in which a stream of page references and the corresponding stack instants are shown. The principal advantage of this approach is that there is no need to search for the page to be replaced because it is always the bottom most element of the stack. This approach can be implemented using either software or microcodes. However, this method takes more time when a page number is moved from the middle of the stack.
- Alternatively, the LRU policy can be implemented by adding an age register to each entry of the page table and a virtual clock to the CPU. The virtual clock is organized so that it is incremented after each memory reference. When a page is referenced, its

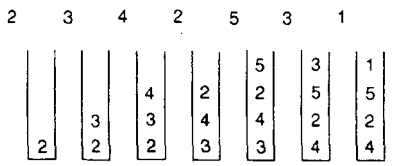


FIGURE 8.24 Implementation of the LRU Algorithm Using a Stack

age register is loaded with the contents of the virtual clock. The age register of a page holds the time at which that page was most recently referenced. The least-recent page is that page whose age register value is minimum. This approach requires an operating system to perform time-consuming housekeeping operations. Thus the performance of the system may be degraded.

- To implement these methods, the computer system must provide adequate hardware support. Incrementing the virtual clock using software takes more time. Thus the operating speed of the entire system is reduced. The LRU policy can not be implemented in systems that do not provide enough hardware support. To get around this problem, some replacement policy is employed that will approximate the LRU policy.
- The LRU policy can be approximated by adding an extra bit called an activity bit to each entry of the page table. Initially all activity bits are cleared to 0. When a page is referenced, its activity bit is set to 1. Thus this bit tells whether or not the page is used. Any page whose activity bit is 0 may be a candidate for replacement. However, the activity bit cannot determine how many times a page has been referenced.
- More information can be obtained by adding a register to each page table entry. To illustrate this concept, assume a 16-bit register has been added to each entry of the page table. Assume that the operating system is allowed to shift the contents of all the registers 1 bit to the right at regular intervals. With one right shift, the most-significant bit position becomes vacant. If it is assumed that the activity bit is used to fill this vacant position, some meaningful conclusions can be derived. For example, if the content of a page register is 0000_{16}, then it can be concluded that this page was not in use during the last 16 time-interval periods. Similarly, a value $FFFF_{16}$ for page register indicates that the page should have been referenced at least once in the last 16 time-interval periods. If the content of a page register is $FF00_{16}$ and the content of another one is $00F0_{16}$, the former was used more recently.
- If the content of a page register is interpreted as an integer number, then the least-recent page has a minimum page register value and can be replaced. If two page registers hold the minimum value, then either of the pages can be evicted, or one of them can be chosen on a FIFO basis.
- The larger the size of the page register, the more time is spent by the operating system in the update operations. When the size of the page register is 0, the history of the system can only be obtained via the activity bits. If the proposed replacement procedure is applied on the activity bits alone, the result is known as the second-chance replacement policy.
- Another bit called a dirty bit may be appended to each entry of the page table. This bit is initially cleared to 0 and set to 1 when a page is modified.
- This bit can be used in two different ways:
    - The idea of a dirty bit reduces the swapping overhead because when the dirty bit of a page to be replaced is zero, there is no need to copy this page into the

secondary memory, and it can be overwritten by an incoming page. A dirty bit can be used in conjunction with any replacement algorithm.

- A priority scheme can be set up for replacement using the values of the dirty and activity bits, as described next.

PRIORITY LEVEL	ACTIVITY BIT	DIRTY BIT	MEANING
0	0	0	Neither used nor modified.
1	0	1	Not recently used but modified.
2	1	0	Used but not modified.
3	1	1	Used as well as dirty.

Using the priority levels just described, the following replacement policy can be formulated: When it is necessary to replace a page, choose that page whose priority level is minimum. In the event of a tie, select the victim on a FIFO basis.

In some systems, the LRU policy is approximated using the least frequently used (LFU) and most frequently used (MFU) algorithms. A thorough discussion of these procedures is beyond the scope of this book.

- One of the major goals in a replacement policy is to minimize the page-fault rate. A program is said to be in a thrashing state if it generates excessive numbers of page faults. Replacement policy may not have a complete control on thrashing. For example, suppose a program generates the following stream of page references:

  1,2,3,4, 1,2,3,4, 1,2,3,4, . . .

  If it runs on a system with three frames it will definitely enter into thrashing state even if the optimal replacement policy is implemented.
- There is a close relationship between the degree of multiprogramming and thrashing. In general, the degree of multiprogramming is increased to improve the CPU use. However, in this case more thrashing occurs. Therefore, to reduce thrashing, the degree of multiprogramming is reduced. Now the CPU utilization drops. CPU utilization and thrashing are conflicting performance issues.

8.1.4 Cache Memory Organization

The performance of a microcomputer system can be significantly improved by introducing a small, expensive, but fast memory between the microprocessor and main memory. This memory is called "cache memory" and this idea was first introduced in the IBM 360/85 computer. Later on, this concept was also implemented in minicomputers such as the PDP-11/70. With the advent of VLSI technology, the cache memory technique is gaining acceptance in the microprocessor world. Studies have shown that typical programs spend most of their execution times in loops. This means that the addresses generated by a microprocessor have a tendency to cluster around a small region in the main memory, a phenomenon known as "locality of reference." Typical 32-bit microprocessors can execute the same instructions in a loop from the on-chip cache rather than reading them repeatedly from the external main memory. Thus, the performance is greatly improved. For example, an on-chip cache memory is implemented in Intel's 32-bit microprocessor, the 80486/Pentium, and Motorola's 32-bit microprocessor, the MC 68030/68040. The 80386 does not have an on-chip cache, but external cache memory can be interfaced to it.

The block diagram representation of a microprocessor system that employs a cache memory is shown in Figure 8.25. Usually, a cache memory is very small in size and

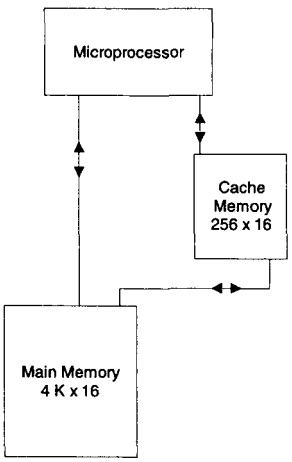


FIGURE 8.25 Memory organization of a microprocessor system that employs a cache memory

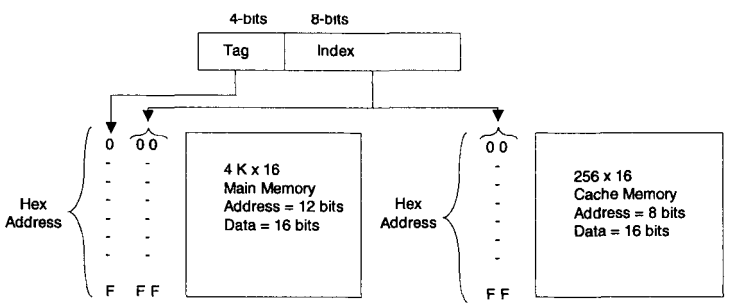


FIGURE 8.26 Addresses for main memory and cache memory

its access time is less than that of the main memory by a factor of 5. Typically, the access times of the cache and main memories are 100 and 500 ns, respectively. If a reference is found in the cache, we call it a "cache hit," and the information pertaining to the microprocessor reference is transferred to the microprocessor from the cache. However, if the reference is not found in the cache, we call it a "cache miss." When there is a cache miss, the main memory is accessed by the microprocessor and, the instructions and/or data are then transferred to the microprocessor from the main memory. At the same time, a block containing the desired information needed by the microprocessor is transferred from the main memory to cache. The block normally contains 4 to 16 words, and this block is placed in the cache using the standard replacement policies such as FIFO or LRU. This block transfer is done with a hope that all future references made by the microprocessor will be confined to the fast cache.

The relationship between the cache and main memory blocks is established using mapping techniques. Three widely used mapping techniques are *Direct mapping, Fully associative mapping, and Set-associative mapping*. In order to explain these three mapping techniques, the memory organization of Figure 8.26 will be used. The main memory is capable of storing 4K words of 16 bits each. The cache memory, on the other hand, can store 256 words of 16 bits each. An identical copy of every word stored in cache exists in main

memory. The microprocessor first accesses the cache. If there is a hit, the microprocessor accepts the 16-bit word from the cache. In case of a miss, the microprocessor reads the desired 16-bit word from the main memory and this 16-bit word is then written to the cache. A cache memory may contain instructions only (Instruction cache) or data only (Data cache) or both instructions and data (Unified cache).

Direct mapping uses a RAM for the cache. The microprocessor's 12-bit address is divided into two fields, an index field and a tag field. Because the cache address is 8 bits wide ($2^8 = 256$), the low-order 8 bits of the microprocessor's address form the index field, and the remaining 4 bits constitute the tag field. This is illustrated in Figure 8.26.

In general, if the main memory address field is m bits wide and the cache memory address is n bits wide, the index field will then require n bits and the tag field will be (m - n) bits wide. The n-bit address will access the cache. Each word in the cache will include the data word and its associated tag. When the microprocessor generates an address for main memory, the index field is used as the address to access the cache. The tag field of

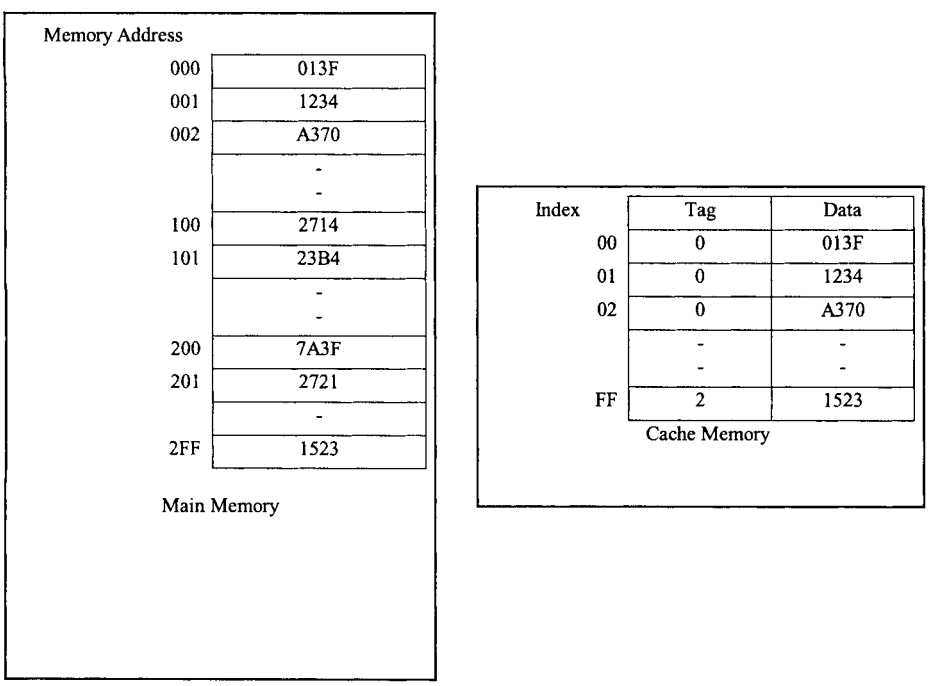


FIGURE 8.27 Direct mapping numerical example

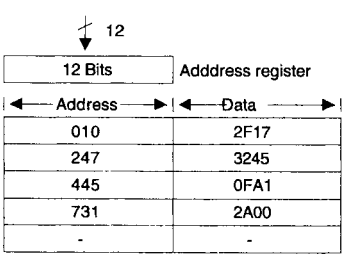


FIGURE 8.28 Associative mapping, numerical example

the main memory is compared with the tag field in the word read from cache. A hit occurs if the tags match. This means that the desired data word is in cache. A miss occurs if there is no match, and the required word is read from main memory. It is written in the cache along with the tag. One of the main drawbacks of direct mapping is that numerous misses may occur if two or more words with addresses having the same index but with different tags are accessed several times. This situation should be avoided or can be minimized by having such words far apart in the address lines. Let us now illustrate the concept of direct mapping for a data cache by means of a numerical example of Figure 8.27. All numbers are in hexadecimal.

The content of index address 00 of cache is tag = 0 and data = 013F. Suppose that the microprocessor wants to access the memory address 100. The index address 00 is used to access the cache. The memory address tag 1 is compared with the cache tag of 0. This does not produce a match. Therefore, the main memory is accessed and the data 2714 is transferred into the microprocessor. The cache word at index address 00 is then replaced with a tag of 1 and data of 2714.

The fastest and the most expensive cache memory utilizes an associative memory. This method is known as "fully associative mapping." Each element in associative memory contains a main memory address and its content (data). When the microprocessor generates a main memory address, it is compared associatively (simultaneously) with all addresses in the associative memory. If there is a match, the corresponding data word is read from the associative cache memory and sent to the microprocessor. If a miss occurs, the main memory is accessed and the address along with its corresponding data are written to the associative cache memory. If the cache is full, certain policies such as FIFO are used as replacement algorithms for the cache. The associative cache is expensive but provides fast operation. The concept of an associative cache is illustrated by means of a numerical example in Figure 8.28. Assume all numbers are in hexadecimal.

The associative memory stores both the memory address and its contents (data). The figure shows four words stored in the associative cache. Each word in the cache is the 12-bit address along with its 16-bit contents (data). When the microprocessor wants to access memory, the 12-bit address is placed in an address register and the associative cache memory is searched for a matching address. Suppose that the content of the microprocessor address register is 445. Because there is a match, the microprocessor reads the corresponding data 0FA1 into an internal data register.

Set-associative mapping is a combination of direct and associative mapping. Each cache word stores two or more main memory words using the same index address. Each main memory word consists of a tag and its data word. An index with two or more tags and data words forms a set. When the microprocessor generates a memory request, the index of the main memory address is used as the cache address. The tag field of the main memory address is then compared associatively (simultaneously) with all tags stored under the index. If a match occurs, the desired data word is read. If a match does not occur, the

Index	Tag	Data	Tag	Data
00	0	013F	2	7A3F
01	1	23B4	2	2721

FIGURE 8.29 Set-associative mapping, numerical example with set size of 2

data word, along with its tag, is read from main memory and also written into the cache.

The hit ratio improves as the set size increases because more words with the same index but different tags can be stored in the cache. The concept of set-associative mapping can be illustrated by the numerical example shown in figure 8.29. Assume that all numbers are in hexadecimal.

Each cache word can store two or more memory words under the same index address. Each data item is stored with its tag. The size of a set is defined by the number of tag and data items in a cache word. A set size of two is used in this example. Each index address contains two data words and their associated tags.Each tag includes 4 bits, and each data word contains 16 bits. Therefore, the word length = 2 × (4 + 16) = 40 bits. An index address of 8 bits can represent 256 words. Hence, the size of the cache memory is 256 × 40. It can store 512 main memory words because each cache word includes two data words.

The hex numbers shown in Figure 8.29 are obtained from the main memory contents shown in Figure 8.27. The words stored at addresses 000 and 200 of main memory of figure 8.27 are stored in cache memory (shown in Figure 8.29) at index address 00. Similarly, the words at addresses 101 and 201 are stored at index address 01. When the microprocessor wants to access a memory word, the index value of the address is used to access the cache. The tag field of the microprocessor address is then compared with both tags in the cache associatively (simultaneously) for a cache hit. If there is a match, appropriate data is read into the microprocessor. The hit ratio will improve as the set size increases because more words with the same index but different tags can be stored in the cache. However, this may increase the cost of comparison logic.

There are two ways of writing into cache: the write-back and write-through methods. In the write-back method, whenever the microprocessor writes something into a cache word, a "dirty" bit is assigned to the cache word. When a dirty word is to be replaced with a new word, the dirty word is first copied into the main memory before it is overwritten by the incoming new word. The advantage of this method is that it avoids unnecessary writing into main memory.

In the write-through method, whenever the microprocessor alters a cache address, the same alteration is made in the main memory copy of the altered cache address. This policy can be easily implemented and also ensures that the contents of the main memory are always valid. This feature is desirable in a multiprocesssor system, in which the main memory is shared by several processors. However, this approach may lead to several unnecessary writes to main memory.

One of the important aspects of cache memory organization is to devise a method that ensures proper utilization of the cache. Usually, the tag directory contains an extra bit for each entry, called a "valid" bit. When the power is turned on, the valid bit corresponding to each cache block entry of the tag directory is reset to zero. This is done in order to indicate that the cache block holds invalid data. When a block of data is first transferred from the main memory to a cache block, the valid bit corresponding to this cache block is set to 1. In this arrangement, whenever the valid bit is zero, it implies that a new incoming block can overwrite the existing cache block. Thus, there is no need to copy the contents of the cache block being replaced into the main memory.

The performance of a system that employs a cache can be formally analyzed as follows: If t_c, h, and t_m specify the cache-access time, hit ratio, and the main memory access time, respectively; then the average access time can be determined as shown in the equation below:

$$t_{av} = ht_c + (1 - h)(t_c + t_m)$$

The hit ratio h always lies in the closed interval 0 and 1, and it specifies the relative number of successful references to the cache. In the above equation, when there is a cache hit, the main memory will not be accessed; and in the event of a cache miss, both main memory and cache will be accessed. Suppose the ratio of main memory access time to cache access time is γ, then an expression for the efficiency of a system that employs a cache can be derived as follows:

$$\begin{aligned} Efficieny = E &= \frac{t_c}{t_{av}} \\ &= \frac{t_c}{ht_c + (1-h)(t_c + t_m)} \\ &= \frac{1}{h + (1-h)(1 + \frac{t_m}{t_c})} \\ &= \frac{1}{h + (1-h)(1+\gamma)} \\ &= \frac{1}{1 + \gamma(1-h)} \end{aligned}$$

Note that E is maximum when $h = 1$ (when all references are confined to the cache). A hit ratio of 90% (h = 0.90) is not uncommon with many contemporary systems.

Example 8.9

Calculate t_{av}, γ, and E of a memory system whose parameters are as indicated:

$t_c = 160$ ns

$t_m = 960$ ns

$h = 0.90$

Solution

$$\begin{aligned} t_{av} &= ht_c + (1 - h)(t_c + t_m) \\ &= 0.9\,(160) + (0.1)\,(960 + 160) \\ &= 144 + 112 \\ &= 256 \text{ ns} \end{aligned}$$

$$\gamma = \frac{t_m}{t_c} = \frac{960}{160} = 6$$

$$E = \frac{1}{1 + \gamma(1-h)} = \frac{1}{1 + 6(0.1)} = 0.625$$

This result indicates that by employing a cache, efficiency is improved by 62.5%. Assume the unit of mapping is a block; then the relationship between the main and cache memory blocks can be established by using a specific mapping technique.

In fully associative mapping, a main memory block i can be mapped to any cache block j, where $0 \leq i \leq M - 1$ *and* $0 \leq j \leq N - 1$ Note that the main memory has M blocks and the cache is divided into N blocks. To determine which block of main memory is stored into the cache, a tag is required for each block. Hence,

Tag (j) = address of the main memory block stored in the cache block j.

Suppose $M = 2^m$ and $N = 2^n$; then m and n bits are required to specify the addresses of a main and cache memory block, respectively. Also, block size = 2^w, where w bits are required to specify a word in a block.

For Associative mapping : m bits of the main memory are used as a tag; and N tags are

needed since there are N cache blocks.

Main memory address = (Tag + w)bits.

For Direct mapping: High order (m-n) bits are used as a tag.

Main memory address = (Tag + n + w)bits

For Set-associative mapping:

Tag field = (m - n + s) bits, where Blocks/set = 2^s

Cache set number = (n - s) bits

Main memory address = (Tag size + cache set number + w) bits.

Example 8.10

The parameters of a computer memory system are specified as follows:

- Main memory size = 8K blocks
- Cache memory size =512 blocks
- Block size = 8 words

Determine the sizes of the tag field along with the main memory address using each of the following methods:

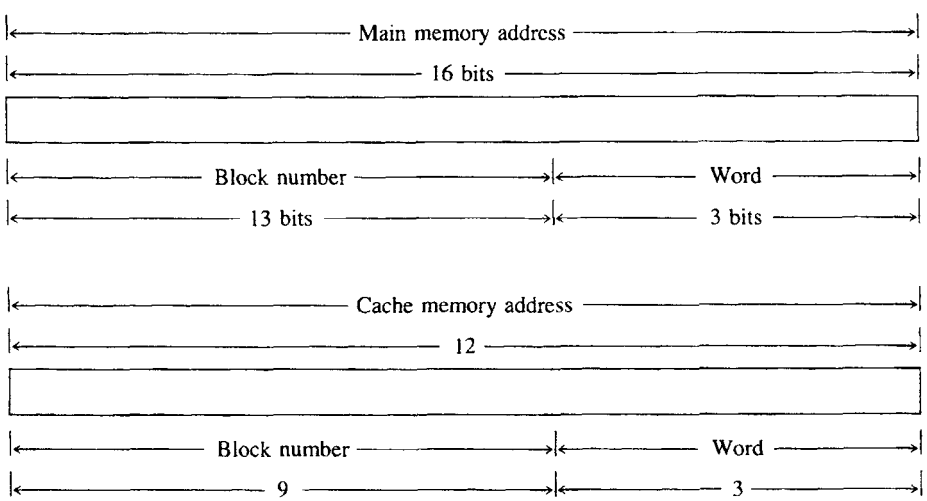
(a) Fully associative mapping
(b) Direct mapping
(c) Set associative mapping with 16 blocks/set

Solution

With the given data, compute the following:

- M = 8K = 8192 = 2^{13}, and thus m = 13.
- N = 512 = 2^9, and thus n = 9.
- Block size = 8 words = 2^3 words, and thus we require 3 bits to specify a word within a block.

Using this information, we can determine the main and cache memory address formats as shown next:



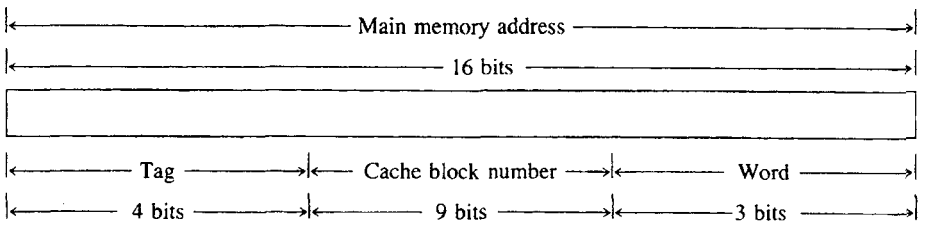
(a) In this case, the size of the tag field is m = 13 = bits:

Size of the main memory address = Tag (bits) + Word (bits)

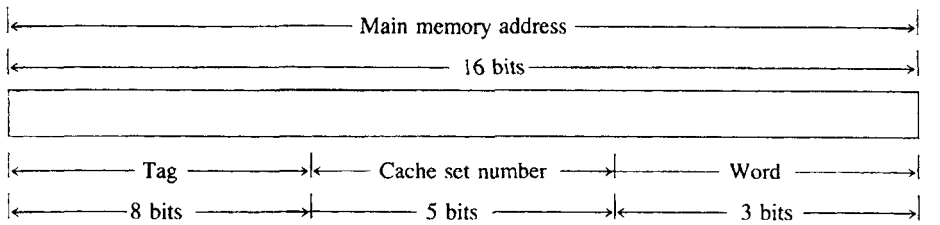
= 13 bits + 3 bits

= 16 bits

(b) In this case, the size of the tag field is m - n = 13-9=4 bits:



(c) s = 16 = 2^4, and thus s = 4. Therefore, the size of the tag field is m - n + s =13-9+4=8 bits:



Example 8.11

The access time of a cache memory is 50 ns and that of the main memory is 500 ns. It is estimated that 80% of the main memory requests are for read and the remaining are for write. The hit ratio for read access only is 0.9 and a write-through policy is used.

(a) Determine the average access time considering only the read cycles.

(b) What is the average time if the write requests are also taken into consideration

Solution

(a) $t_{av} = ht_c + (1 - h)(t_c + t_m)$
$= 0.9 \times 50 + (0.1)(550)$
$= 45 + 55$ ns
$= 100$ ns

(b) $t_{read/write}$ = *(read request probability)* × $t_{av\ read}$ + (1- *read request probability*) × $t_{av\ write}$
read request probability = 0.8
write request probability = 0.2
$t_{av\ read} = t_{av} = 100$ ns (result of part (a))
$t_{av\ write} = 500$ ns (because both the main and cache memories are updated at the same time)
$t_{read/write} = 0.8 \times 100 + 0.2 \times 500$
$= 80 + 100$ ns
$= 180$ ns

The growth in 1C technology has allowed manufacturers to fabricate a cache on the CPU chip. The on-chip cache of Motorola's 32-bit microprocessor, the MC68020, is discussed next.

The MC68020 on-chip cache is a direct mapped instruction cache. Only instructions are cached; data items are not. This cache is a collection of 64 entries, where each cache entry consists of a 26-bit tag field and 32-bit instruction data. The tag field

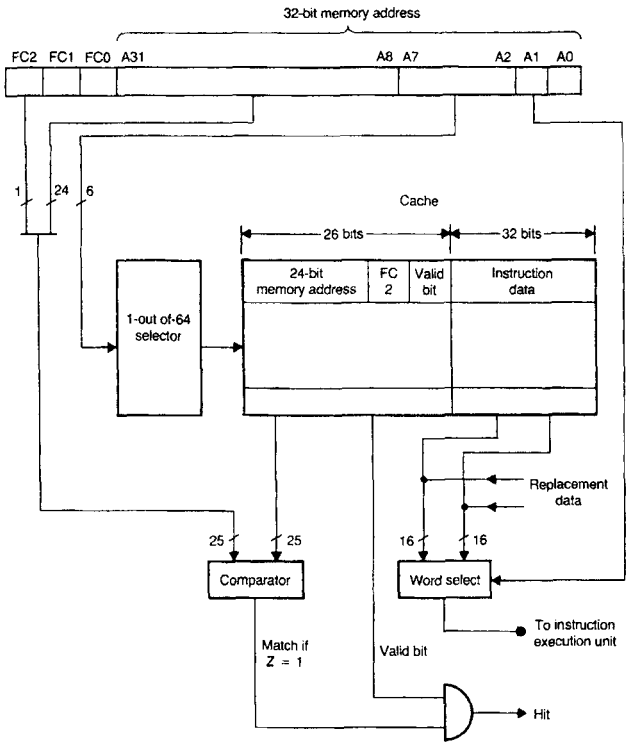


FIGURE 8.30 MC68020 On-chip Cache Organization

includes the following components:

- High-order 24 bits of the memory address.
- The most-significant bit FC2 of the function code. In the MC68020 processor, the 3-bit function code combination FC2 FC1 FC0 is used to identify the status of the processor and the address space (discussed in Chapter 10) of the bus cycle. For example, FC2 = 1 means the processor operates in the supervisory or privileged mode. Otherwise, it operates in the user mode. Similarly, when FC1 FC0 = 01, the bus cycle is made to access data. When FC1 FC0 = 10, the bus cycle is made to access code.
- Valid bit.

A block diagram of the MC68020 on-chip cache is shown in Figure 8.30.

If an instruction fetch occurs when the cache is enabled, the cache is first checked to determine if the word requested is in the cache. This is achieved by first using 6 bits of the memory address (A7-A2) to select one of the 64 entries of the cache. Next, address bits A31-A8 and function bit FC2 are compared to the corresponding values of the selected cache entry. If there is a match and the valid bit is set, a cache hit is occurs.

In this case, the address bit Al is used to select the proper instruction word stored in the cache and the cycle ends. If there is no match or the valid bit is cleared, and a cache miss occurs. In this case, the instruction is fetched from external memory. This new instruction is automatically written into the cache and the valid bit is set. Since the processor always pre fetches instructions from the external memory in the form of long words, both instruction data words of the cache will be updated regardless of which word caused the miss.

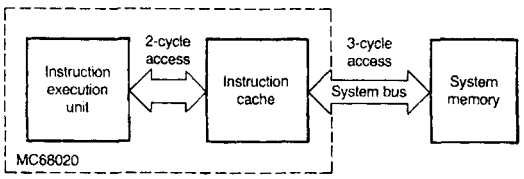


FIGURE 8.31 MC68020 Instruction Cache.

The MC68020 on-chip instruction cache obtains a significant increase in performance by reducing the number of fetches required to external memory. Typically, this cache reduces the instruction execution time in two ways. First, it provides a two-clock-cycle access time for an instruction that hits in the cache (see Figure 8.31); second, if the access hits in the cache, it allows simultaneous instruction and data access to occur. Of these two benefits, simultaneous access is more significant, since it allows 100% reduction in the time required to access the instruction rather than the 33% reduction afforded by going from three to two clocks.

Finally, microprocessors such as Intel Pentium II support two-levels of cache. These are L1 (Level 1) and L2 (Level 2) cache memories. The L1 cache (Smaller in size) is contained inside the processor chip while the L2 cache (Larger in size) is interfaced external to the microprocessor. The L1 cache normally provides separate instruction and data caches. The processor can directly access the L1 cache while the L2 cache normally supplies instructions and data to the L1 cache. The L2 cache is usually accessed by the microprocessor only if L1 misses occur. This two-level cache memory enhances the performance of the microprocessor.

8.2 Input/Output

One communicates with a microcomputer system via the I/O devices interfaced to it. The user can enter programs and data using the keyboard on a terminal and execute the programs to obtain results. Therefore, the I/O devices connected to a microcomputer system provide an efficient means of communication between the microcomputer and the outside world. These I/O devices are commonly called "peripherals" and include keyboards, CRT displays, printers, and disks.

The characteristics of the I/O devices are normally different from those of the microcomputer. For example, the speed of operation of the peripherals is usually slower than that of the microcomputer, and the word length of the microcomputer may be different from the data format of the peripheral devices. To make the characteristics of the I/O devices compatible with those of the microcomputer, interface hardware circuitry between the microcomputer and I/O devices is necessary. Interfaces provide all input and output transfers between the microcomputer and peripherals by using an I/O bus. An I/O bus carries three types of signals: device address, data, and command.

The microprocessor uses the I/O bus when it executes an I/O instruction. A typical I/O instruction has three fields. When the computer executes an I/O instruction, the control unit decodes the op-code field and identifies it as an I/O instruction. The CPU then places the device address and command from respective fields of the I/O instruction on the I/O bus. The interfaces for various devices connected to the I/O bus decode this address, and

an appropriate interface is selected. The identified interface decodes the command lines and determines the function to be performed. Typical functions include receiving data from an input device into the microprocessor or sending data to an output device from the microprocessor. In a typical microcomputer system, the user gets involved with two types of I/O devices: physical I/O and virtual I/O. When the computer has no operating system, the user must work directly with physical I/O devices and perform detailed I/O design.

There are three ways of transferring data between the microcomputer and physical I/O device:

1. Programmed I/O
2. Interrupt I/O
3. Direct memory access (DMA)

The microcomputer executes a program to communicate with an external device via a register called the "I/O port" for programmed I/O. An external device requests the microcomputer to transfer data by activating a signal on the microcomputer's interrupt line during interrupt I/O. In response, the microcomputer executes a program called the interrupt-service routine to carry out the function desired by the external device. Data transfer between the microcomputer's memory and an external device occurs without microprocessor involvement with direct memory access.

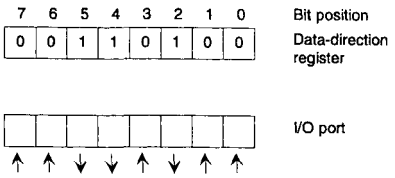
In a microcomputer with an operating system, the user works with virtual I/O devices. The user does not have to be familiar with the characteristics of the physical I/O devices. Instead, the user performs data transfers between the microcomputer and the physical I/O devices indirectly by calling the I/O routines provided by the operating system using virtual I/O instructions.

Basically, an operating system serves as an interface between the user programs and actual hardware. The operating system facilitates the creation of many logical or virtual I/O devices, and allows a user program to communicate directly with these logical devices. For example, a user program may write its output to a virtual printer. In reality, a virtual printer may refer to a block of disk space. When the user program terminates, the operating system may assign one of the available physical printers to this virtual printer and monitor the entire printing operation. This concept is known as "spooling" and improves the system throughput by isolating the fast processor from direct contact with a slow printing device. A user program is totally unaware of the logical-to-physical device-mapping process. There is no need to modify a user program if a logical device is assigned to some other available physical device. This approach offers greater flexibility over the conventional hardware-oriented techniques associated with physical I/O.

8.2.1 Programmed I/O

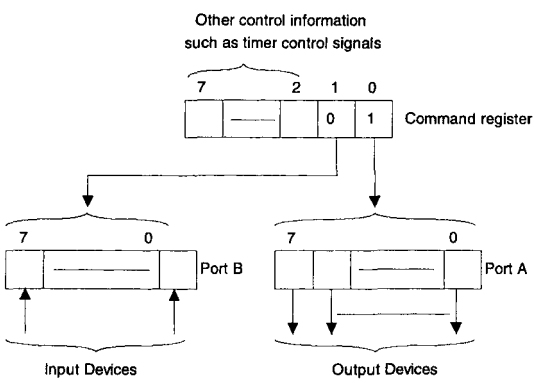
A microcomputer communicates with an external device via one or more registers called "I/O ports" using programmed I/O. I/O ports are usually of two types. For one type, each bit in the port can be individually configured as either input or output. For the other type, all bits in the port can be set up as all parallel input or output bits. Each port can be configured as an input or output port by another register called the "command" or "data-direction register." The port contains the actual input or output data. The data-direction register is an output register and can be used to configure the bits in the port as inputs or outputs.

Each bit in the port can be set up as an input or output, normally by writing a 0 or a 1 in the corresponding bit of the data-direction register. As an example, if an 8-bit data-direction register contains 34H, then the corresponding port is defined as follows:



In this example, because 34H (0011 0100) is sent as an output into the data-direction register, bits 0, 1, 3, 6, and 7 of the port are set up as inputs, and bits 2, 4, and 5 of the port are defined as outputs. The microcomputer can then send output to external devices, such as LEDs, connected to bits 2, 4, and 5 through a proper interface. Similarly, the microcomputer can input the status of external devices, such as switches, through bits 0, 1, 3, 6, and 7. To input data from the input switches, the microcomputer assumed here inputs the complete byte, including the bits to which LEDs are connected. While receiving input data from an I/O port, however, the microcomputer places a value, probably 0, at the bits configured as outputs and the program must interpret them as "don't cares." At the same time, the microcomputer's outputs to bits configured as inputs are disabled.

For parallel I/O, there is only one data-direction register, usually known as the "command register" for all ports. A particular bit in the command register configures all bits in the port as either inputs or outputs. Consider two I/O ports in an I/O chip along with one command register. Assume that a 0 or a 1 in a particular bit position defines all bits of ports A or B as inputs or outputs. An example is depicted in the following:



Some I/O ports are called "handshake ports." Data transfer occurs via these ports through exchanging of control signals between the microcomputer and an external device.

I/O ports are addressed using either *standard I/O* or *memory-mapped I/O* techniques. The "standard I/O" (also called "isolated I/O" by Intel) uses an output pin such as M/$\overline{IO}$ pin on the Intel 8086 microprocessor chip. The processor outputs a HIGH on this pin to indicate to memory and the I/O chips that a memory operation is taking place. A LOW output from the processor to this pin indicates an I/O operation. Execution of IN or OUT instruction makes the M/$\overline{IO}$ LOW, whereas memory-oriented instructions, such as MOVE, drive the M/$\overline{IO}$ to HIGH. In standard I/O, the processor uses the M/$\overline{IO}$ pin to distinguish between I/O and memory. For typical processors, an 8-bit address is commonly used for each I/O port. With an 8-bit I/O port address, these processors are capable of addressing 256 ports. In addition, some processors can also use 16-bit I/O ports. However, in a typical application, four or five I/O ports may usually be required. Some of the address bits of the microprocessor are normally decoded to obtain the I/O port addresses. With

"memory-mapped I/O", the processor, on the other hand, does not differentiate between I/O and memory, and therefore, does not use the M/$\overline{IO}$ control pin. The processor uses a portion of the memory addresses to represent I/O ports. The I/O ports are mapped as part of the processor's main memory addresses which may not physically exist, but are used by the microprocessor's memory-oriented instructions such as MOVE to generate the necessary control signals to perform I/O. Motorola microprocessors do not have the control pin such as M/$\overline{IO}$ pin and use only "memory-mapped I/O" while Intel microprocessors can use both types.

When standard I/O is used, typical processors normally use 2-byte IN or OUT instruction as follows:

IN port number	{	2-byte instruction for inputting data from the specified I/O port into the processor's register
OUT port number	{	2-byte instruction for outputting data from the register into the specified I/O port

With memory-mapped I/O, the processor normally uses instructions, namely, MOVE, as follows:

MOVE M, reg	where M= Port address mapped into memory	{	instruction for inputting I/O data into a register
MOVE reg, M	where M= Port address mapped into memory	{	instruction for outputting data from a register into the specified port

There are typically two ways via which programmed I/O can be utilized. These are *unconditional I/O* and *conditional I/O*. The processor can send data to an external

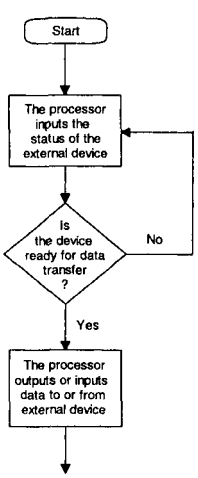


FIGURE 8.32 Flowchart for conditional programmed I/O

device at any time using unconditional I/O. The external device must always be ready for data transfer. A typical example is when the processor outputs a 7-bit code through an I/O port to drive a seven-segment display connected to this port. In conditional I/O, the processor outputs data to an external device via *handshaking*. This means that data transfer occurs via exchanging of control signals between the processor and an external device. The processor inputs the status of the external device to determine whether the device is ready for data transfer. Data transfer takes place when the device is ready. The flow chart in Figure 8.32 illustrates the concept of conditional programmed I/O.

The concept of conditional I/O will now be demonstrated by means of data transfer between a processor and an analog-to-digital (A/D) converter. Consider, for example, the A/D converter shown in Figure 8.33. This A/D converter transforms an analog voltage V_x into an 8-bit binary output at pins D_7-D_0. A pulse at the START conversion pin initiates the conversion. This drives the BUSY signal LOW. The signal stays LOW during the conversion process. The BUSY signal goes to HIGH as soon as the conversion ends. Because the A/D converter's output is tristated, a LOW on the $\overline{\text{OUTPUT ENABLE}}$ transfers the converter's outputs. A HIGH on the $\overline{\text{OUTPUT ENABLE}}$ drives the converter's outputs to a high impedance state.

The concept of conditional I/O can be demonstrated by interfacing the A/D converter to a typical processor. Figure 8.34 shows such an interfacing example. The user writes a program to carry out the conversion process. When this program is executed, the processor sends a pulse to the START pin of the converter via bit 2 of port A. The processor then checks the BUSY signal by inputting bit 1 of port A to determine if the conversion is completed. If the BUSY signal is HIGH (indicating the end of conversion), the processor sends a LOW to the $\overline{\text{OUTPUT ENABLE}}$ pin of the A/D converter. The processor then inputs the converter's D_0-D_7 outputs via port B. If the conversion is not completed, the

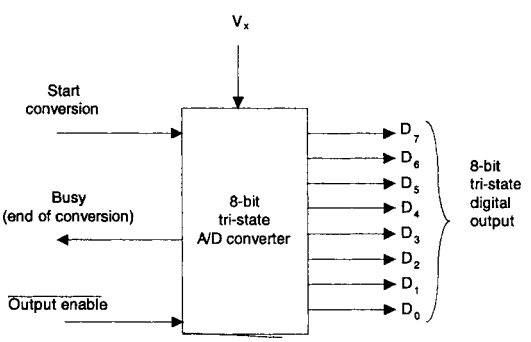


FIGURE 8.33 A/D converter

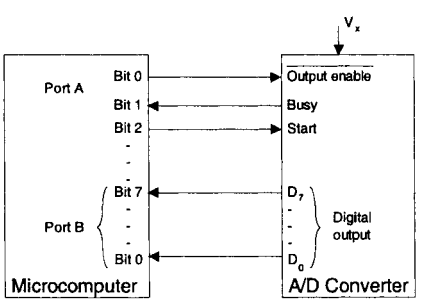


FIGURE 8.34 Interfacing an A/D converter to a microcomputer

processor waits in a loop checking for the BUSY signal to go to HIGH.

8.2.2 Interrupt I/O

A disadvantage of conditional programmed I/O is that the microcomputer needs to check the status bit (BUSY signal for the A/D converter) by waiting in a loop. This type of I/O transfer is dependent on the speed of the external device. For a slow device, this waiting may slow down the microcomputer's capability of processing other data. The interrupt I/O technique is efficient in this type of situation.

Interrupt I/O is a device-initiated I/O transfer. The external device is connected to a pin called the "interrupt (INT) pin" on the processor chip. When the device needs an I/O transfer with the microcomputer, it activates the interrupt pin of the processor chip. The microcomputer usually completes the current instruction and saves the contents of the current program counter and the status register in the stack.

The microcomputer then automatically loads an address into the program counter to branch to a subroutine-like program called the "interrupt-service routine." This program is written by the user. The external device wants the microcomputer to execute this program to transfer data. The last instruction of the service routine is a RETURN, which is typically similar in concept to the RETURN instruction used at the end of a subroutine. The RETURN from interrupt instruction normally loads the program counter and the status register with the information saved in the stack before going to the service routine . Then, the microcomputer continues executing the main program. An example of interrupt I/O is shown in Figure 8.35.

Assume the microcomputer is MC68000 based and executing the following program:

```
        ORG       $2000
        MOVE.B    #$81, DDRA       ;    configure bits 0 and 7
                                   ;    of port A as outputs
        MOVE.B    #$00, DDRB       ;    configure Port B as input
        MOVE.B    #$81, PORTA      ;    send start pulse to A/D
                                   ;    and HIGH to OUTPUT ENABLE
        MOVE.B    #$01, PORTA
        CLR.W     D0               ;    clear 16-bit register D0 to 0
BEGIN   MOVE.W    D1, D2
          :
```

The extensions .B and .W represent byte and word operations. Note that the symbols $ and # indicate hexadecimal number and immediate mode respectively. The preceding program is arbitrarily written. The program logic can be explained using the MC68000 instruction set. Ports DDRA and DDRB are assumed to be the data-direction registers for ports A and B, respectively. The first four `MOVE` instructions configure bits 0 and 7 of port A as outputs and port B as the input port, and then send a trailing START pulse (HIGH and then LOW) to the A/D converter along with a HIGH to the $\overline{\text{OUTPUT ENABLE}}$. This HIGH $\overline{\text{OUTPUT ENABLE}}$ is required to disable the A/D's output. The microcomputer continues with execution of the `CLR.W D0` instruction. Suppose that the BUSY signal becomes HIGH, indicating the end of conversion during execution of the `CLR.W D0` instruction. This drives the INT signal to HIGH, interrupting the microcomputer. The microcomputer completes execution of the current instruction, `CLR.W D0`. It then saves the current contents of the program counter (address BEGIN) and status register automatically and executes a subroutine-like program called the service routine. This program is usually written by the user. The microcomputer manufacturer normally specifies the starting address of the

service routine, or it may be provided by the user via external hardware. Assume this address is $4000, where the user writes a service routine to input the A/D converter's output as follows:

```
ORG       $4000
MOVE.B    #$00, PORTA     ;      Activate OUTPUT ENABLE.
MOVE.B    PORTB, D1       ;      Input A/D
RTE                       ;      Return and restore PC and SR.
```

In this service routine, the microcomputer inputs the A/D converter's output. The return instruction RTE, at the end of the service routine, pops address BEGIN and the previous status register contents from the stack and loads the program counter and status register with them. The microcomputer executes the `MOVE.W D1,D2` instruction at address BEGIN and continues with the main program. The basic characteristics of interrupt I/O have been discussed so far. The main features of interrupt I/O provided with a typical microcomputer are discussed next.

Interrupt Types

There are typically three types of interrupts: external interrupts, traps or internal interrupts, and software interrupts. External interrupts are initiated through the microcomputer's interrupt pins by external devices such as A/D converters. External interrupts can further be divided into two types: maskable and nonmaskable. Nonmaskable interrupt can not be enabled or disabled by instructions while microprocessor's instruction set contains instructions to enable or disable maskable interrupt. For example, Intel 8086 can disable or enable maskable interrupt by executing instructions such as CLI (Clear interrupt flag in the Status register to 0) or STI (Set interrupt flag in the Status register to 1) . The 8086 recognizes the maskable interrupt after execution of the STI while ignores it upon execution of the CLI. Note that the 8086 has an interrupt-flag bit in the Status register. The nonmaskable interrupt has a higher priority than the maskable interrupt. If both maskable and nonmaskable interrupts are activated at the same time, the processor will service the nonmaskable interrupt first. The nonmaskable interrupt is typically used as a power failure interrupt. Processors normally use +5 V DC, which is transformed from 110 V AC. If the power falls below 90 V AC, the DC voltage of +5 V cannot be maintained. However, it will take a few milliseconds before the AC power drops below 90 V AC. In these few milliseconds, the power-failure-sensing circuitry can interrupt the processor. The interrupt-service routine can be written to store critical data in nonvolatile memory such as battery-backed CMOS RAM, and the interrupted program can continue without any loss of data when the power returns.

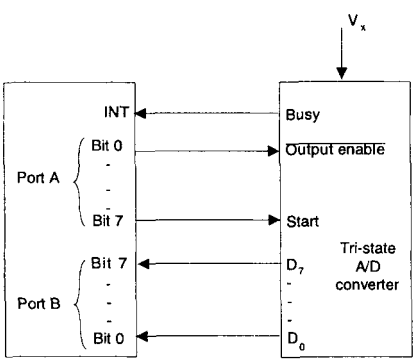


FIGURE 8.35 Microcomputer A/D converter interface via interrupt I/O

Some processors such as the 8086 are provided with a maskable handshake interrupt. This interrupt is usually implemented by using two pins — INTR and $\overline{\text{INTA}}$. When the INTR pin is activated by an external device, the processor completes the current instruction, saves at least the current program counter onto the stack, and generates an interrupt acknowledge ($\overline{\text{INTA}}$). In response to the $\overline{\text{INTA}}$, the external device provides an 8-bit number, using external hardware on the data bus of the microcomputer. This number is then read and used by the microcomputer to branch to the desired service routine.

Internal interrupts, or traps, are activated internally by exceptional conditions such as overflow, division by zero, or execution of an illegal op-code. Traps are handled in the same way as external interrupts. The user writes a service routine to take corrective measures and provide an indication to inform the user that an exceptional condition has occurred. Many processors include software interrupts, or system calls. When one of these instructions is executed, the processor is interrupted and serviced similarly to external or internal interrupts. Software interrupt instructions are normally used to call the operating system. These instructions are shorter than subroutine calls, and no calling program is needed to know the operating system's address in memory. Software interrupt instructions allow the user to switch from user to supervisor mode. For some processors, a software interrupt is the only way to call the operating system, because a subroutine call to an address in the operating system is not allowed.

Interrupt Address Vector

The technique used to find the starting address of the service routine (commonly known as the interrupt address vector) varies from one processor to another. With some processors, the manufacturers define the fixed starting address for each interrupt. Other manufacturers use an indirect approach by defining fixed locations where the interrupt address vector is stored.

Saving the Microprocessor Registers

When a processor is interrupted, it saves at least the program counter on the stack so that the processor can return to the main program after executing the service routine. Typical processors save one or two registers, such as the program counter and status register, before going to the service routine. The user should know the specific registers the processor saves prior to executing the service routine. This will allow the user to use the appropriate return instruction at the end of the service routine to restore the original conditions upon return to the main program.

Interrupt Priorities

A processor is typically provided with one or more interrupt pins on the chip. Therefore, a special mechanism is necessary to handle interrupts from several devices that share one of these interrupt lines. There are two ways of servicing multiple interrupts: polled and daisy chain techniques.

i) Polled Interrupts

Polled interrupts are handled by software and are therefore are slower than daisy chaining. The processor responds to an interrupt by executing one general-service routine for all devices. The priorities of devices are determined by the order in which the routine polls each device. The processor checks the status of each device in the general-service routine, starting with the highest-priority device, to service an interrupt. Once the processor determines the source of the interrupt, it branches to the service routine for the device.

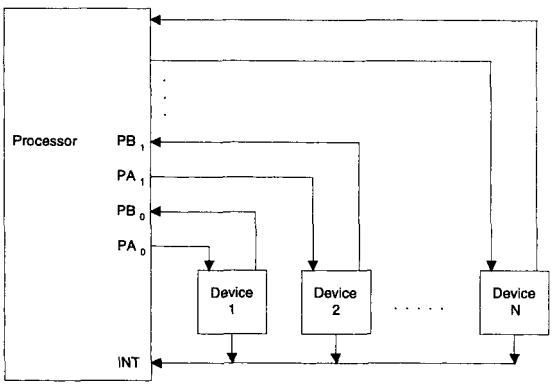


FIGURE 8.36 Polled interrupt

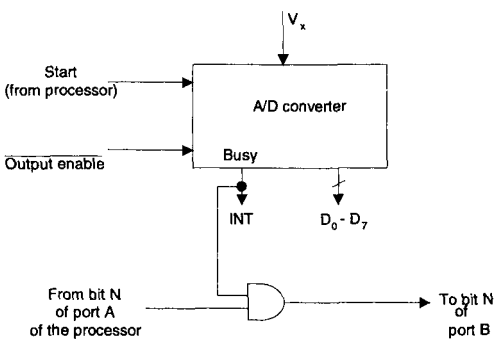


FIGURE 8.37 Device N and associated logic for polled interrupt

Figure 8.36 shows a typical configuration of the polled-interrupt system.

In Figure 8.36, several external devices (device 1, device 2,..., device N) are connected to a single interrupt line of the processor via an OR gate (not shown in the figure). When one or more devices activate the INT line HIGH, the processor pushes the program counter and possibly some other registers onto the stack. It then branches to an address defined by the manufacturer of the processor. The user can write a program at this address to poll each device, starting with the highest-priority device, to find the source of the interrupt. Suppose the devices in Figure 8.36 are A/D converters. Each converter, along with the associated logic for polling, is shown in Figure 8.37.

Assume that in Figure 8.36 two A/D converters (device 1 and device 2) are provided with the START pulse by the processor at nearly the same time. Suppose the user assigns device 2 the higher priority. The user then sets up this priority mechanism in the general-service routine. For example, when the BUSY signals from device 1 and/or 2 become HIGH, indicating the end of conversion, the processor is interrupted. In response, the processor pushes at least the program counter onto the stack and loads the PC with the interrupt address vector defined by the manufacturer.

The general interrupt-service routine written at this address determines the source of the interrupt as follows: A 1 is sent to PA1 for device 2 because this device has higher priority. If this device has generated an interrupt, the output (PB1) of the AND gate in Figure 8.37 becomes HIGH, indicating to the processor that device 2 generated the interrupt. If the output of the AND gate is 0, the processor sends a HIGH to PA0 and checks the output

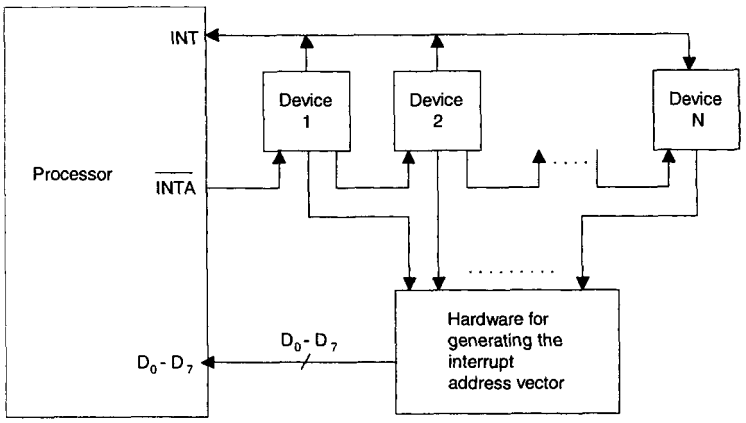


FIGURE 8.38 Daisy chain interrupt

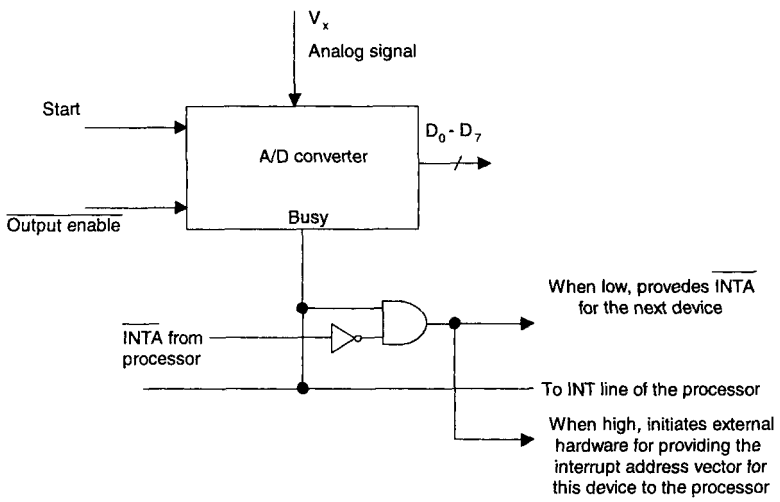


FIGURE 8.39 Each device and the associated logic in a daisy chain

(PB0) for HIGH. Once the source of the interrupt is determined, the processor can be programmed to jump to the service routine for that device. The service routine enables the A/D converter and inputs the converter's outputs to the processor.

Polled interrupts are slow, and for a large number of devices, the time required to poll each device may exceed the time to service the device. In such a case, a faster mechanism, such as the daisy chain approach, can be used.

ii) Daisy Chain Interrupts

Devices are connected in a daisy chain fashion, as shown in Figure 8.38, to set up priority systems. Suppose one or more devices interrupt the processor. In response, the processor pushes at least the PC and generates an interrupt acknowledge ($\overline{\text{INTA}}$) signal to the highest-priority device (device 1 in this case). If this device has generated the interrupt, it will accept the $\overline{\text{INTA}}$; otherwise, it will pass the $\overline{\text{INTA}}$ onto the next device until the $\overline{\text{INTA}}$ is accepted.

Once accepted, the device provides a means for the processor to find the interrupt-

address vector by using external hardware. Assume the devices in Figure 8.38 are A/D converters. Figure 8.39 provides a schematic for each device and the associated logic.

Suppose the processor in Figure 8.39 sends a pulse to start the conversions of the A/D converters of devices 1 and 2 at nearly the same time. When the BUSY signal goes to HIGH, the processor is interrupted through the INT line. The processor pushes the program counter and possibly some other registers. It then generates a LOW at the interrupt-acknowledge $\overline{\text{INTA}}$ for the highest-priority device (device 1 in Figure 8.38). Device 1 has the highest priority—it is the first device in the daisy chain configuration to receive $\overline{\text{INTA}}$. If A/D converter 1 has generated the BUSY HIGH, the output of the AND gate becomes HIGH. This signal can be used to enable external hardware to provide the interrupt-address vector on the processor's data lines. The processor then branches to the service routine. This program enables the converter and inputs the A/D output to the processor via Port B. If A/D converter #1 does not generate the BUSY HIGH, however, the output of the AND gate in Figure 8.39 becomes LOW (an input to device 2's logic) and the same sequence of operations takes place. In the daisy chain, each device has the same logic with the exception of the last device, which must accept the $\overline{\text{INTA}}$. Note that the outputs of all the devices are connected to the INT line via an OR gate (not shown in Figure 8.38)

8.2.3 Direct Memory Access (DMA)

Direct memory access (DMA) is a technique that transfers data between a microcomputer's memory and an I/O device without involving the microprocessor. DMA is widely used in transferring large blocks of data between a peripheral device such as a hard disk and the microcomputer's memory. The DMA technique uses a DMA controller chip for the data-transfer operations. The DMA controller chip implements various components such as a counter containing the length of data to be transferred in hardware in order to speed up data transfer. The main functions of a typical DMA controller are summarized as follows:

- The I/O devices request DMA operation via the DMA request line of the controller chip.
- The controller chip activates the microprocessor HOLD pin, requesting the microprocessor to release the bus.
- The processor sends HLDA (hold acknowledge) back to the DMA controller, indicating that the bus is disabled. The DMA controller places the current value of its internal registers, such as the address register and counter, on the system bus and sends a DMA acknowledge to the peripheral device. The DMA controller completes the DMA transfer.

There are three basic types of DMA: block transfer, cycle stealing, and interleaved DMA. For block-transfer DMA, the DMA controller chip takes over the bus from the microcomputer to transfer data between the microcomputer memory and I/O device. The microprocessor has no access to the bus until the transfer is completed. During this time, the microprocessor can perform internal operations that do not need the bus. This method is popular with microprocessors. Using this technique, blocks of data can be transferred.

Data transfer between the microcomputer memory and an I/O device occurs on a word-by-word basis with cycle stealing. Typically, the microprocessor is generated by ANDing an $\overline{\text{INHIBIT}}$ signal with the system clock. The system clock has the same frequency as the microprocessor clock. The DMA controller controls the $\overline{\text{INHIBIT}}$ line. During normal operation, the $\overline{\text{INHIBIT}}$ line is HIGH, providing the microprocessor clock. When DMA operation is desired, the controller makes the $\overline{\text{INHIBIT}}$ line LOW for one clock cycle. The microprocessor is then stopped completely for one cycle. Data transfer

between the memory and I/O takes place during this cycle. This method is called "cycle stealing" because the DMA controller takes away or steals a cycle without microprocessor recognition. Data transfer takes place over a period of time.

With interleaved DMA, the DMA controller chip takes over the system bus when the microprocessor is not using it. For example, the microprocessor does not use the bus while incrementing the program counter or performing an ALU operation. The DMA controller chip identifies these cycles and allows transfer of data between the memory and I/O device. Data transfer takes place over a period of time for this method.

Because block-transfer DMA is common with microprocessors, a detailed description is provided. Figure 8.40 shows a typical diagram of the block-transfer DMA. In the figure, the I/O device requests the DMA transfer via the DMA request line connected to the controller chip. The DMA controller chip then sends a HOLD signal to the microprocessor, and it then waits for the HOLD acknowledge (HLDA) signal from the microprocessor. On receipt of the HLDA, the controller chip sends a DMA ACK signal to the I/O device. The controller takes over the bus and controls data transfer between the RAM and I/O device. On completion of the data transfer, the controller interrupts the microprocessor by the INT line and returns the bus to the microprocessor by disabling the HOLD and DMA ACK signals.

The DMA controller chip usually has at least three registers normally selected by the controller's register select (RS) line: an address register, a terminal count register, and a status register. Both the address and terminal counter registers are initialized by the microprocessor. The address register contains the starting address of the data to be transferred, and the terminal counter register contains the desired block to be transferred. The status register contains information such as completion of DMA transfer. Note that the

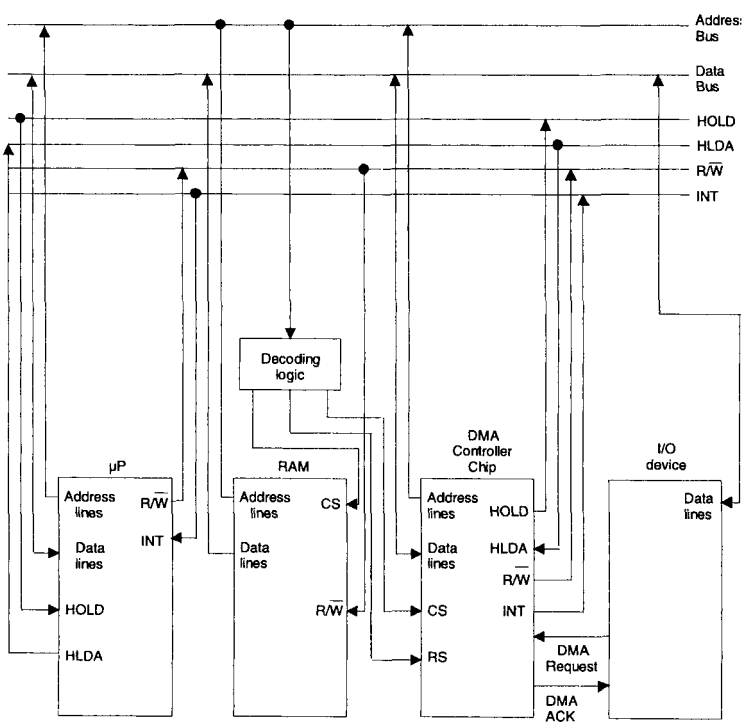


FIGURE 8.40 Typical block transfer

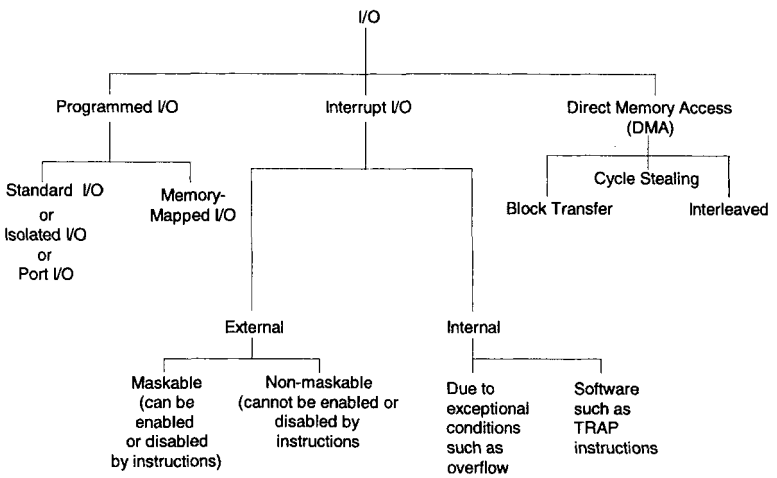


FIGURE 8.41 I/Ostructure of a typical microcomputer

DMA controller implements logic associated with data transfer in hardware to speed up the DMA operation.

8.3 Summary of I/O

Figure 8.41 summarizes various I/O devices associated with a typical microprocessor.

8.4 Fundamentals of Parallel Processing

The term "parallel processing" means improving the performance of a computer system by carrying out several tasks simultaneously. A high volume of computation is often required in many application areas, including real-time signal processing. A conventional single computer contains three functional elements: CPU, memory, and I/O. In such a uniprocessor system, a reasonable degree of parallelism was achieved in the following manner:

1. The IBM 370/168 and CDC 6600 computers included a dedicated I/O processor. This additional unit was capable of performing all I/O operations by employing the DMA technique discussed earlier. In these systems, parallelism was achieved by keeping the CPU and I/O processor busy as much as possible with program execution and I/O operations respectively.

2. In the CDC 6600 CPU, there were 24 registers and 10 execution units. Each execution unit was designed for a specific operation such as addition, multiplication, and shifting. Since all units were independent of each other, several operations were performed simultaneously.

3. In many uniprocessor systems such as IBM 360, parallelism was achieved by using high-speed hardware elements such as carry-look-ahead adders and carry-save adders.

4. In several conventional computers, parallelism is incorporated at the instruction-execution level. Recall that an instruction cycle typically includes activities such as op code fetch, instruction decode, operand fetch, operand execution, and result saving. All these operations can be carried out by overlapping the instruction fetch phase with the

instruction execution phase. This is known as instruction pipelining. This pipelining concept is implemented in the state-of-the-art microprocessors such as Intel's Pentium series.

5. In many uniprocessor systems, high throughput is achieved by employing high speed memories such as cache and associative memories. The use of virtual memory concepts such as paging and segmentation also allows one to achieve high processing rates because they reduce speed imbalance between a fast CPU and a slow periphal device such as a hard disk. These concepts are also implemented in today's microprocessors to achieve high performance.

6. It is a common practice to achieve parallelism by employing software methods such as multiprogramming and time sharing in uniprocessors. In both techniques, the CPU is multiplexed among several jobs. This results in concurrent processing, which improves the overall system throughput.

8.4.1 General Classifications of Computer Architectures

Over the last two decades, parallel processing has drawn the attention of many research workers, and several high-speed architectures have been proposed. To present these results in a concise manner, different architectures must be classified in well defined groups.
All computers may be categorized into different groups using one of three classification methods:

1. Flynn
2. Feng
3. Handler

The two principal elements of a computer are the processor and the memory. A processor manipulates data stored in the memory as dictated by the instruction. Instructions are stored in the memory unit and always flow from memory to processor. Data movement

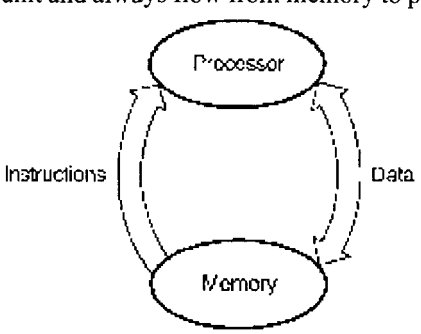


FIGURE 8.42 Processor-Memory Interaction

NAME OF THE ARCHITECTURE	NAME OF THE ARCHITECTURE IN ABBREVIATED FORM
Single-instruction stream-single-data stream	SISD
Single-instruction stream-multiple-data stream	SIMD
Multiple-instruction stream-single-data stream	MISD
Multiple-instruction stream-multiple-data stream	MIMD

FIGURE 8.43 Classification of Computers Using Flynn's Method

is bidirectional, meaning data may be read from or written into the memory. Figure 8.42 shows the processor-memory interaction.

The number of instructions read and data items manipulated simultaneously by the processor form the basis for Flynn's classification. Figure 8.43 shows the four types of computer architectures that are defined using Flynn's method. The SISD computers are capable of manipulating a single data item by executing one instruction at a time. The SISD classification covers the conventional uniprocessor systems such as the VAX-11, IBM 370, Intel 8085, and Motorola 6809. The processor unit of a SISD machine may have one or many functional units. For example, the VAX-11/780 is a SISD machine with a single functional unit. CDC 6600 and IBM 370/168 computers are typical examples of SISD systems with multiple functional units. In a SISD machine, instructions are executed in a strictly sequential fashion. The SIMD system allows a single instruction to manipulate several data elements. These machines are also called vector machines or array processors. Examples of this type of computer are the ILLIAC-IV and Burroughs Scientific Processor (BSP).

The ILLIAC-IV was an experimental parallel computer proposed by the University of Illinois and built by the Burroughs Corporation. In this system, there are 64 processing elements. Each processing element has its own small local memory unit. The operation of all the processing elements is under the control of a central control unit (CCU). Typically, the CCU reads an instruction from the common memory and broadcasts the same to all processing units so the processing units can all operate on their own data at the same time. This configuration is very useful for carrying out a high volume of computations that are encountered in application areas such as finite-element analysis, logic simulation, and spectral analysis. Modern microprocessors such as Intel Pentium II use the SIMD architecture.

By definition, MISD refers to a computer in which several instructions manipulate the same data stream concurrently. The notion of pipelining is very close to the MISD definition.

A set of instructions constitute a program, and a program operates on several data elements. MIMD organization refers to a computer that is capable of processing several programs simultaneously. MIMD systems include all multiprocessing systems. Based on the degree of processor interaction, multiprocessor systems may be further divided into two groups: loosely coupled and tightly coupled. A tightly coupled system has high interaction between processors. Multiprocessor systems with low interprocessor communications are referred to as loosely coupled systems.

In Feng's approach, computers are classified according to the number of bits processed within a unit time. However, Handler's classification scheme categorizes computers on the basis of the amount of parallelism found at the following levels:

- CPU
- ALU
- Bit

A thorough discussion of these schemes is beyond the scope of this book. Since contemporary microprocessors such as Intel Pentium II use SIMD architechture, a basic coverage of SIMD is provided next. The SIMD computers are also called array processors. A synchronous array processor may be defined as a computer in which a set of identical processing elements act under the control of a master controller (MC). A command given by the MC is simultaneously executed by all processing elements, and a SIMD system is formed. Since all processors execute the same instruction, this organization offers a great

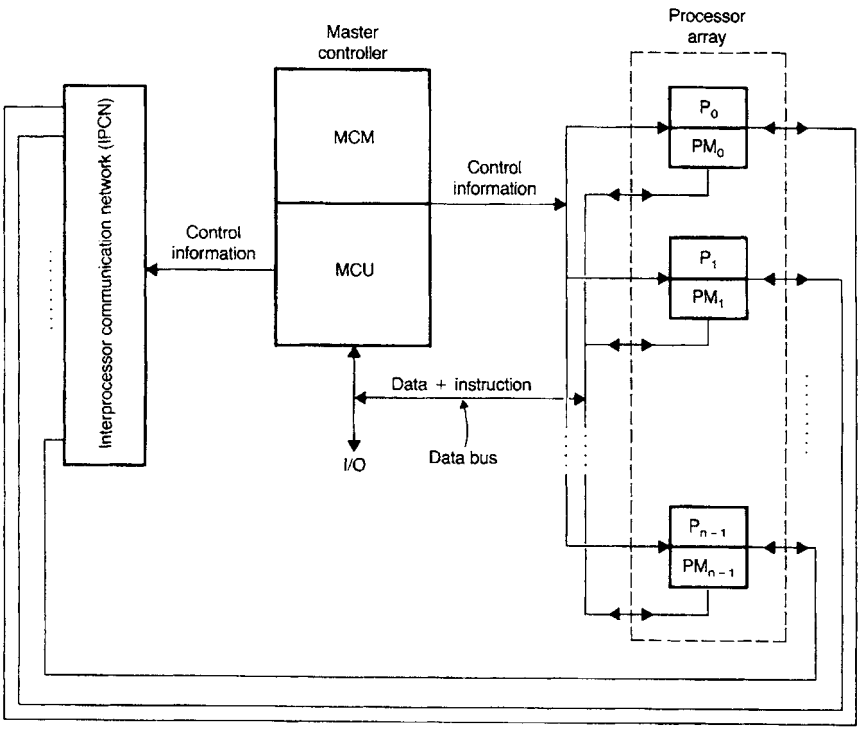


FIGURE 8.44 A Typical Array Processor Organization

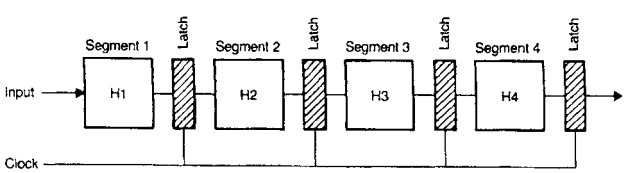


FIGURE 8.45 A Four-segment Pipeline

attraction for vector processing applications such as matrix manipulation.

A conceptual organization of a typical array processor is shown in Figure 8.44. The Master Controller (MC) controls the operation of the processor array. This array consists of N identical processing elements (P_0 through P_{n-1}). Each processing element P_i is assumed to have its own memory, PM^i, to store its data. The MC of Figure 8.44 contains two major components:

- The master control unit (MCU)
- The master control memory (MCM)

The MCU is the CPU of the master controller and includes an ALU and a set of registers. The purpose of the MCM is to hold the instructions and common data.
Each instruction of a program is executed under the supervision of the MCU in a sequential fashion. The MCU fetches the next instruction, and the execution of this instruction will take place in one of the following ways:

- If the instruction fetched is a scalar or a branch instruction, it is executed by the MC itself.
- If the instruction fetched is a vector instruction, such as vector add or vector multiply, then the MCU broadcasts the same instruction to each P^i, of the processor array, allowing all P_i's to execute this instruction simultaneously.

Assume the required data is already within the processing element's private memory. Before execution of a vector instruction, the system ensures that appropriate data values are routed to each processing element's private memory. Such an operation can be performed in two ways:

- All data values can be transferred to the private memories from an external source via the system data bus.
- The MCU can transfer the data values to the private memories via the control bus.

In an array processor like the one shown in Figure 8.44, it may be necessary to disable some processing elements during a vector operation. This is accomplished by including a mask register, M, in the MCU. The mask register contains a bit, m_i, for each processing element, p_i. A particular processing element, p_i, will respond to a vector instruction broadcast by the MCU only when its mask bit, m_i, is set to 1; otherwise, the processing element. P_i, will not respond to the vector instruction and is said to be disabled.

In an array processor, it may be necessary to exchange data between processing elements. Such an exchange of data between processing elements takes place through the path provided by the interprocessor communication network (IPCN). Data exchanges refers to exchanges between scratchpad registers of the processing elements and exchanges between private memories of the processing elements.

8.4.2 Pipeline Processing

The purpose of this section is to provide a brief overview of pipelining.

Basic Concepts

Assume a task T is carried out by performing four activities: Al, A2, A3, and A4, in that order. Hardware Hi is designed to perform the activity Ai. Hi is referred to as a segment, and it essentially contains combinational circuit elements. Consider the arrangement shown in Figure 8.45.

In this configuration, a latch is placed between two segments so the result computed by one segment can serve as input to the following segment during the next clock period. The execution of four tasks Tl, T2, T3, and T4 using the hardware of Figure 8.45 is described using a space-time chart shown in Figure 8.46.

Initially, task Tl is handled by segment 1. After the first clock, segment 2 is busy with Tl while segment 1 is busy with T2. Continuing in this manner, the task Tl is completed at the end of the fourth clock. However, following this point, one task is shipped out per clock. This is the essence of the pipeline concept. A pipeline gains efficiency for the same reason as an assembly line does: Several activities are performed but not on the same material.

Suppose ti and L denote the propagation delays of segment i and the latch, respectively. Then the pipeline clock period T can be expressed as follows:

$$T = \max (Tl, T2, \ldots Tn) + L$$

The segment with the maximum delay is known as the bottleneck, and it decides the pipeline clock period T. The reciprocal of T is referred to as the pipeline frequency.

Consider the execution of m tasks using an n-segment pipeline. In this case, the

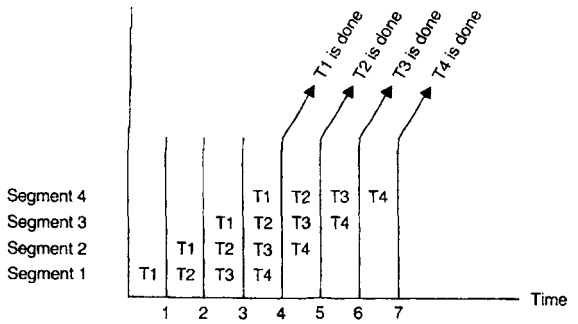


FIGURE 8.46 Overlapped Execution of Four Tasks Using a Pipeline

first task will be completed after n clocks (because there are n segments) and the remaining m-1 tasks are shipped out at the rate of one task per pipeline clock.

Therefore, n + (m – 1) clock periods are required to complete m tasks using an n-segment pipeline. If all m tasks are executed without any overlap, mn clock periods are needed because each task has to pass through all n segments. Thus speed gained by an n segment pipeline can be shown as follows:

$$\text{speedup } P(n) = \frac{\text{number of clocks required when there is no overlap}}{\text{number of clocks required when tasks are overlapped in time}} = \frac{mn}{n + m - 1}$$

P(n) approaches n when m approaches infinity. This implies that when a large number of tasks are carried out using an *n*-segment pipeline, an *n*-fold increase in speed can be expected.

The previous result shows that the pipeline completes m tasks in the m + n - 1 clock periods. Therefore, its throughput can be defined as follows:

$$\text{throughput of an } n\text{-segment pipeline} = U(n) = \frac{\text{number of tasks computed per unit time}}{} = \frac{m}{(n + m - 1)T}$$

For a large value of m, U(n) approaches 1/T, which is the pipeline frequency. Thus the throughput of an ideal pipeline is equal to the reciprocal of its clock period. The efficiency of an *n*-segment pipeline is defined as the ratio of the actual speedup to the maximum speedup realized.

$$\text{efficiency of an } n\text{-segment pipeline} = E(n) = \frac{\text{actual speedup}}{\text{maximum speedup}} = \frac{P(n)}{n}$$

This illustrates that when m is very large, E(n) approaches 1 as expected.

In many modem computers, the pipeline concept is used in carrying out two tasks: arithmetic operations and instruction execution.

Arithmetic Pipelines

The pipeline concept can be used to build high-speed multipliers. Consider the multiplication P = M * Q, where M and Q are 8-bit numbers. The 16-bit product P can be expressed as:

$P = M(q_7 2^7 + q_6 2^6 + q_5 2^5 + q_4 2^4 + q_3 2^3 + q_2 2^2 + q_1 2^1 + q_0 2^0)$. Hence, $P = \sum_{i=0}^{7} Mq_i 2^i$. This result can also be rewritten as: $P = \sum_{i=0}^{7} S_i$

where, $S_i = Mq_i 2^i$ and each S_i represents a 16-bit partial product. Each partial product is the shifted multiplicand. All 8 partial products can be added using several carry-save adders.

This concept can be extended to design an n × n pipelined multiplier. Here n partial products must be summed with 2n bits per partial product. So, as n increases, the hardware cost associated with a fully combinational multiplier increases in an exponential fashion. To reduce the hardware cost, large multipliers are designed.

The pipeline concept is widely used in designing floating-point arithmetic units. Consider the process of adding two floating point numbers A = 0.9234 * 10^4 and B = 0.48 * 10^2. First, notice that the exponents of A and B are unequal. Therefore, the smaller number should be modified so that its exponent is equal to the exponent of the greater number. For this example, modify B to 0.0048 * 10^4. This modification step is known as exponent alignment. Here the decimal point of the significand 0.48 is shifted to the right to obtain the desired result. After the exponent alignment, the significands 0.9234 and 0.0048 are added to obtain the final solution of 0.9282 * 10^4.

For a second example, consider the operation A - B, where A = 0.9234 * 10^4 and B = 0.9230 * 10^4. In this case, no exponent alignment is necessary because the exponent of A equals to the exponent of B. Therefore, the significand of B is subtracted from the significand
of A to obtain 0.9234 - 0.9230 = 0.0004. However, 0.0004 * 10^4 cannot be the final answer because the significand, 0.0004, is not normalized. A floating-point number with base b is said to be normalized if the magnitude of its significand satisfies the following inequality: $1/b \leq |\text{significand}| < 1$.

In this example, since b = 10, a normalized floating-point number must satisfy the condition:

$$0.1 \leq |\text{significand}| < 1$$

(Note that normalized floating-point numbers are always considered because for each real-world number there exists one and only one floating-point representation. This uniqueness property allows processors to make correct decisions while performing compare operations).

The final answer is modified to 0.4 * 10^1. This modification step is known as postnormalization, and here the significand is shifted to the left to obtain the correct result.

In summary, addition or subtraction of two floating-point numbers calls for four activities:

1. Exponent comparison
2. Exponent alignment
3. Significand addition or subtraction
4. Postnormalization

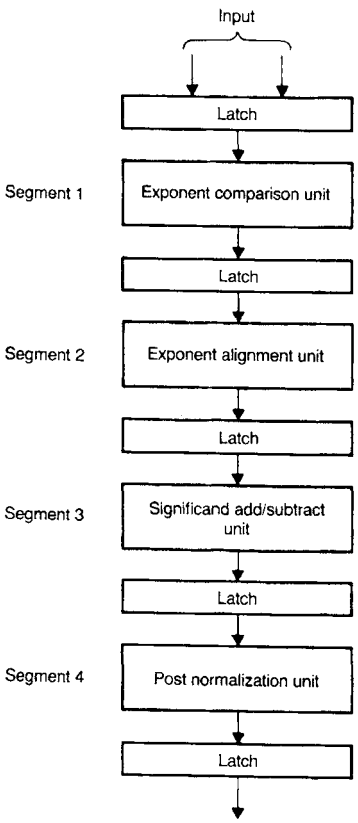


FIGURE 8.47 A Pipelined Floating-point Add/Subtract Unit

Based on this result, a four-segment floating-point adder/subtracter pipeline can be built, as shown in Figure 8.47.

It is important to realize that each segment in this pipeline is primarily composed of combinational components such as multiplexers. The shifter used in this system is the barrel shifter discussed earlier. Modern microprocessors such as Motorola MC 68040 include a 3-stage floating-point pipeline consisting of operand conversion, execute, and result normalization.

Instruction Pipelines

Modern microprocessors such as Motorola MC 68020 contain a 3-stage instruction pipeline. Recall that an instruction cycle typically involves the following activities:

1. Instruction fetch 2. Instruction decode 3. Operand fetch
4. Operation execution 5. Result routing.

This process can be effectively carried out by using the pipeline shown in Figure 8.48. As mentioned earlier, in such a pipelined scheme the first instruction requires five clocks to complete its execution. However, the remaining instructions are completed at a rate of one per pipeline clock. Such a situation prevails as long as all the segments are busy.

In practice, the presence of branch instructions and conflicts in memory accesses poses a great problem to the efficient operation of an instruction pipeline.

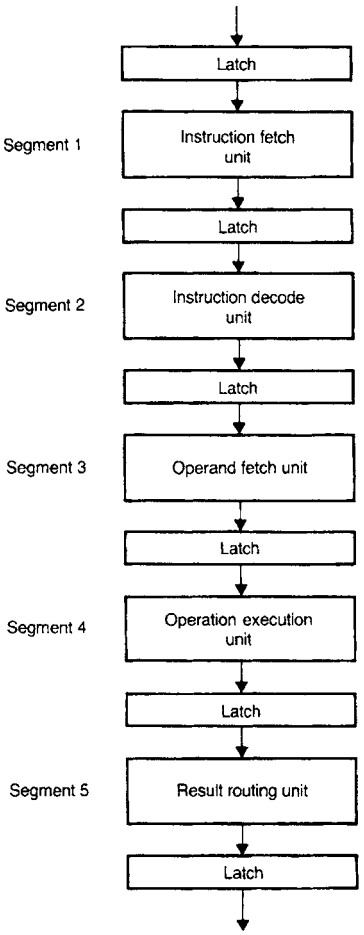


FIGURE 8.48 A Five-segment Instruction Pipeline

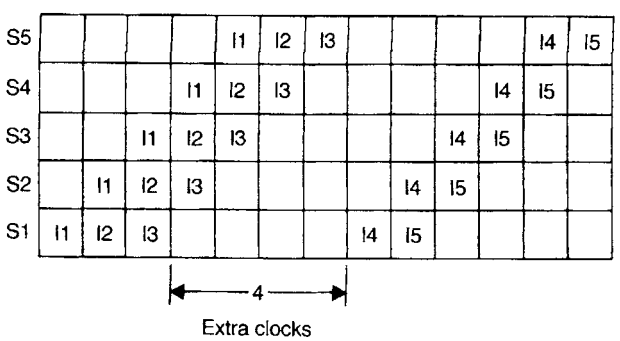


FIGURE 8.49 Pipelined Execution Of A Stream of Five instructions that Includes a Branch Instruction

For example, consider the execution of a stream of five instructions: I1, I2, I3, I4, and I5 in which I3 is a conditional branch instruction. This stream is processed by the instruction pipeline (Figure 8.48) as depicted in Figure 8.49.

When a conditional branch instruction is fetched, the next instruction cannot be fetched because the exact target is not known until the conditional branch instruction has been executed. The next fetch can occur once the branch is resolved. Four additional clocks are required due to I3.

Suppose a stream of s instructions is to be executed using an n-segment pipeline. If c is the probability for an instruction to be a conditional branch instruction, there will be sc conditional branch instructions in a stream of s instructions. Since each branch instruction requires $n - 1$ additional clocks, the total number of clocks required to process a stream of s instructions is $(n + s - 1) + sc(n - 1)$

An instruction cycle constitutes n pipeline clocks. Therefore, the total number of instruction cycles required to execute an instruction is

$$I = \frac{(n+s-1)+sc(n-1)}{n}$$

The average number of instructions executed per instruction cycle is

$$\frac{s}{I} = \frac{sn}{(n+s-1)+sc(n-1)} = \frac{n}{\frac{n}{s}+\frac{(s-1)}{s}+c(n-1)}$$

For a large value of s, the preceding result can be simplified as shown on the following page:

$$\lim_{s\to\infty} \frac{s}{I} = \frac{n}{1+c(n-1)}$$

For n = 5, the equation becomes:

$$\frac{5}{1+4c}$$

For no conditional branch instructions (c = 0), 5 instructions per instruction cycle are executed. This is the best result produced by a five-segment pipeline. If 25% of the

MEMORY ADDRESS	INSTRUCTION
2000	LDA X
2001	INC Y
2002	JMP 2050
2003	SUB Z
.	
.	
.	
.	
.	
2050	STA W
.	
.	
.	
.	
.	

FIGURE 8.50 A Hypothetical Program

MEMORY ADDRESS	INSTRUCTION
2000	LDA X
2001	INC Y
2002	JMP 2051
2003	NOP
2004	SUB Z
.	
.	
.	
2051	STA W

FIGURE 8.51 Modified Sequence

Instruction fetch	LDA X	INC Y	JMP 2051	NOP	STA W
Instruction execute		LDA X	INC Y	JMP 2051	NOP

FIGURE 8.52 Pipelined Execution of a Hypothetical Instruction Sequence

instructions are branch instructions only,

$$\frac{5}{1+4 * 0.25} = 2.5 \text{ instructions}$$

per instruction cycle can be executed. This shows how pipeline efficiency is significantly decreased even with a small percentage of branch instructions.

In many contemporary systems, branch instructions are handled using a strategy called Target Prefetch. When a conditional branch instruction is recognized, the immediate successor of the branch instructions and the target of the branch are prefetched. The latter is saved in a buffer until the branch is executed. If the branch condition is successful, one pipeline is still busy because the branch target is in the buffer.

Another approach to handle branch instructions is the use of the delayed branch concept. In this case, the branch does not take place until after the following instruction. To illustrate

MEMORY ADDRESS	INSTRUCTION
2000	LDA X
2001	JMP 2050
2002	INC Y
2003	SUB Z
.	.
.	.
.	.
2050	STA W

FIGURE 8.53 Instruction Sequence with Branch Instruction Reversed

Instruction fetch	LDA X	JMP 2050	INC Y	STA W	
Instruction execute		LDA X	JMP 2050	INC Y	

FIGURE 8.54 Execution of the Reversed-instruction Sequence

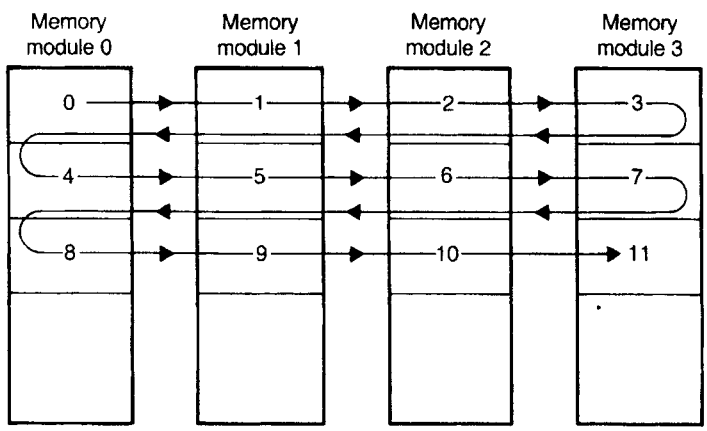


FIGURE 8.55 Memory Interleaving

this, consider the instruction sequence shown in Figure 8.50.

Suppose the compiler inserts a NOP instruction and changes the branch instruction to JMP 2051. The program semantics remain unchanged. This is shown in Figure 8.51.

This modified sequence depicted in Figure 8.51 will be executed by a two-segment pipeline, as shown in Figure 8.52:

- Instruction fetch
- Instruction execute

Because of the delayed branch concept, the pipeline still functions correctly without damage.

The efficiency of this pipeline can be further improved if the compiler produces a new sequence as shown in Figure 8.53.

In this case, the compiler has reversed the instruction sequence. The JMP instruction is placed in the location 2001, and the INC instruction is moved to memory location 2002. This reversed sequence is executed by the same 2-segment pipeline, as shown in Figure 8.54.

It is important to understand that due to the delayed branch rule, the INC Y instruction is fetched before the execution of JMP 2050 instruction; therefore, there is no change in the order of instruction execution. This implies that the program will still produce the same result. Since the NOP instruction was eliminated, the program is executed more efficiently.

The concept of delayed branch is one of the key characteristics of RISC as it makes concurrency visible to a programmer.

As does the presence of branch instructions, memory-access conflicts cause damage to pipeline performance. For example, if the instructions in the operand fetch and result-saving units refer to the same memory address, these operations cannot be overlapped.

To reduce such memory conflicts, a new approach called memory interleaving is often employed. For this case, the memory addresses are distributed among a set of memory modules, as shown in Figure 8.55.

In this arrangement, memory is distributed among many modules. Since consecutive addresses are placed into different modules, the CPU can access several words in one memory access.

QUESTIONS AND PROBLEMS

8.1 What is the basic difference between main memory and secondary memory?

8.2 Compare the basic features of hard disk, floppy disk and Zip disk.

8.3 What are the main differences between CD and DVD memories?

8.4 Name the methods used in main memory array design. What are the advantages and disadvantages of each.

8.5 The block diagram of a 512 × 8 RAM chip is shown in Figure P8.5. In this arrangement, the memory chip is enabled only when $\overline{CS1}$ = L and CS2 = H. Design a 1K × 8 RAM system using this chip as the building block. Draw a neat logic diagram of your implementation. Assume that the microprocessor can directly address 64K with a R/$\overline{W}$ and 8 data pins. Using linear decoding and don't-care conditions as 1's, determine the memory map in hex.

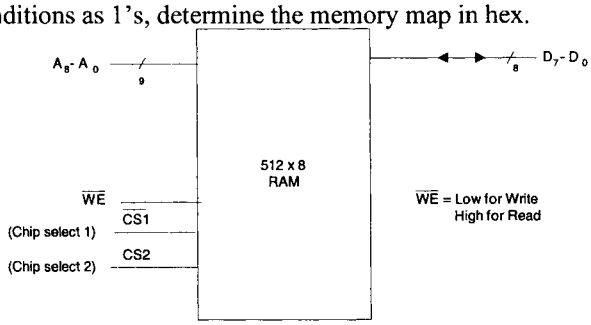


FIGURE P8.5

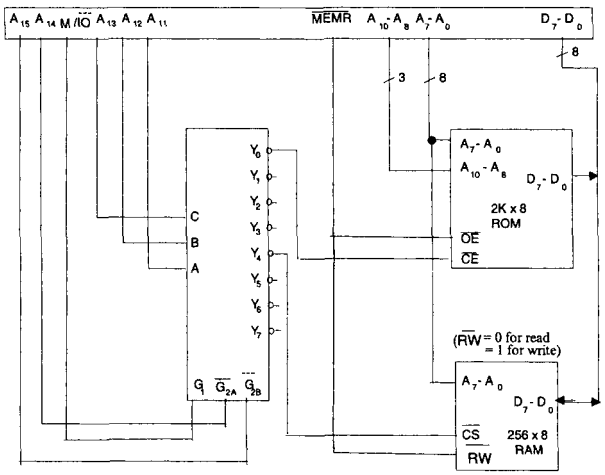


FIGURE P8.6

8.6 Consider the hardware schematic shown in Figure P8.6.

(a) Determine the address map of this system. Note: $\overline{\text{MEMR}}$=0 for read, $\overline{\text{MEMR}}$=1 for write and, M/$\overline{\text{IO}}$=0 for I/O and M/$\overline{\text{IO}}$=1 for memory.

(b) Is there any possibility of bus conflict in this organization? Clearly justify your answer.

8.7 Interface a microprocessor with 16-bit address pins and 8-bit data pins and a R/$\overline{\text{W}}$ pin to a 1K × 8 EPROM chip and two 1K × 8 RAM chips such that the following memory map is obtained:

Device	*Size*	*Address Assignment (in hex)*
EPROM chip	1K × 8	8000–83FF
RAM chip 0	1K × 8	9000–93FF
RAM chip 1	1K × 8	C000–C3FF

Assume that both EPROM and RAM chips contain two enable pins; $\overline{\text{CE}}$ and $\overline{\text{OE}}$ for the EPROM, $\overline{\text{CE}}$ and $\overline{\text{WE}}$ for each RAM. Note that $\overline{\text{WE}}$ =1 and $\overline{\text{WE}}$ = 0 mean read and write operations for the RAM chip. Use a 74138 decoder.

8.8 Repeat Problem 8.7 to obtain the following memory map using a 74138 decoder:

Device	*Size*	*Address Assignment in hex*
EPROM chip	1K × 8	7000–73FF
RAM chip 0	1K × 8	D000–D3FF
RAM chip 1	1K × 8	F000–F3FF

8.9 What is meant by "foldback" in linear decoding?

8.10 Comment on the importance of the following features in an operating system implementation:

(a) Address translation

(b) Protection

8.11 Explain briefly the differences between segmentation and paging.

8.12 Draw a block diagram showing the address and data lines for the 2716, 2732, and 2764 EPROM chips.

8.13 How many address and data lines are required for a 1M × 16 memory chip.

8.14 A microprocessor with 24 address pins and 8 data pins is connected to a 1K × 8 memory chip with one-chip enable. How many unused address bits of the microprocessor are available for interfacing other 1K × 8 memory chips. What is the maximum directly addressable memory available with this microprocessor?

8.15 Design a direct mapped virtual memory system with the following specifications:

- Size of the virtual address space = 64K
- Size of the physical address space = 8K
- Page size = 512 words
- Total length of a page table entry = 24 bits

8.16 A virtual memory system has the following specifications:

- Size of the virtual address space = 64K
- Size of the physical address space = 4K
- Page size = 512

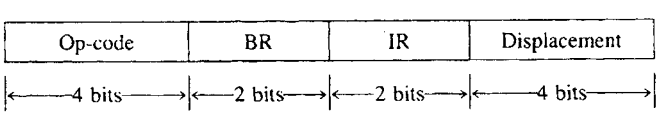
From the page table the following mapping is recognized:

VIRTUAL PAGE NUMBER	PHYSICAL PAGE FRAME NUMBER
0	0
3	1
7	2
4	3
10	4
12	5
24	6
30	7

(a) Find all virtual addresses that will generate a page fault.

(b) Compute the main memory address for the following virtual addresses:
24, 3784, 10250, 30780

8.17 Assume a computer has a segmented memory with paged segments. (Fig. P8.17) The instruction format of this machine is as shown:

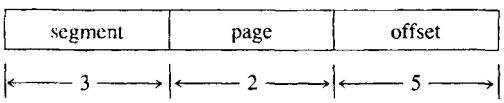


This format has the following fields:

- Op-code field
- 2-bit base register field BR
- 2-bit index register field IR
- 4-bit displacement field

The contents of the specified base and index registers are added with the displacement to produce a virtual address whose format is shown next:

Virtual Address



The virtual address is translated into a physical address by means of segment and page tables, which are stored in the main memory. The segment table entry contains the starting address of its page table and the page table entry contains the address of the location which holds the page frame number. The segment table base address register contains the start address of the segment table. The final physical address is the sum of the page table entry and the offset from the virtual address. Consider the following situation:

(a) Compute the physical address needed by the given situation

(b) Howmany two-operand summations are required to compute one physical address?

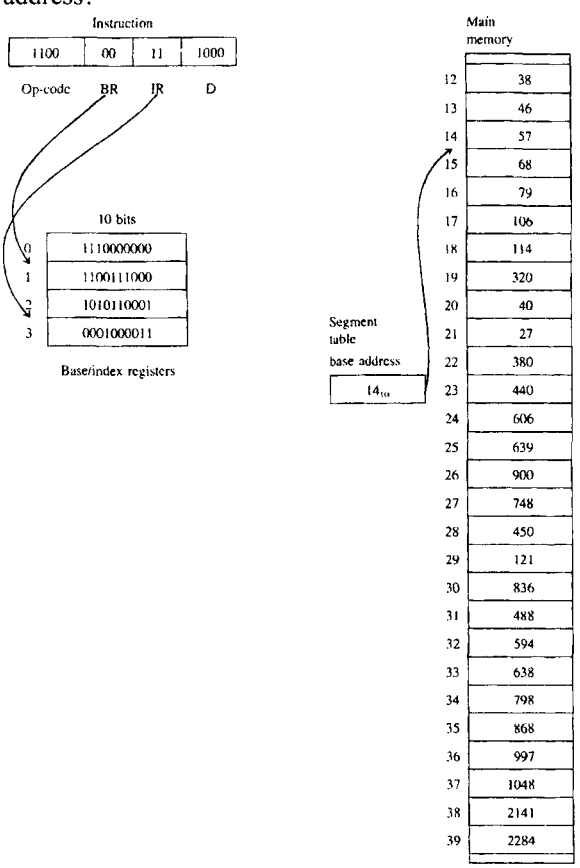


FIGURE P 8.17

8.18 Assume a main memory has 4 page frames and initially all page frames are empty. Consider the following stream of references;

1, 2, 3, 4, 5, 1, 2, 6, 1, 2, 3, 4, 5, 6, 5

Calculate the hit ratio if the replacement policy used is as follows.

(a) FIFO

(b) LRU

8.19 Repeat Problem 8.18 when the main memory has 5 page frames instead of 4. Comment on your results.

8.20 Consider the stream of references given in Problem 8.18. Plot a graph between the hit ratio and the number of frames *(f)* in the main memory after computing the hit ratio for all values *f* in the range of 1 to 8. Assume LRU policy is used. (Hint: Use the stack algorithm.)

8.21 What is the size of a decoder with one chip enable ($\overline{CE}$) to obtain a 64K × 32 memory from the 4K × 8 chips? Where are the inputs and outputs of the decoder connected?

8.22 What is the advantage of having a cache memory? Name a 32-bit microprocessor that does not contain an on-chip cache.

8.23 Discuss the various cache-mapping techniques.

8.24 A microprocessor has a main memory of 8K × 32 and a cache memory of 4K × 32. Using direct mapping, determine the sizes of the tag field, index field, and each word of the cache.

8.25 A microprocessor has a main memory of 4K × 32. Using a cache memory address of 8 bits and set-associative mapping with a set size of 2, determine the size of the cache memory.

8.26 A microprocessor can directly address one megabyte of memory with a 16-bit word size. Determine the size of each cache memory word for associative mapping.

8.27 A typical computer system has a 32K main memory and a 4K fully associative cache memory. The cache block size is 8 words. The access time for the main memory is 10 times that of the cache memory.

(a) How many hardware comparators are needed?

(b) What is the size of the tag field?

(c) If a direct mapping scheme were used instead, what would be the size of the tag field?

(d) Suppose the access efficiency is defined as the ratio of the average access time with a cache to the average access time without a cache, determine the access efficiency assuming a cache hit ratio h of 0.9.

(e) If the cache access time is 200 nanoseconds, what hit ratio would be required to achieve an average access time equal to 500 nanoseconds?

8.28 A set associative cache has a total of 64 blocks divided into sets of 4 blocks each.
(a) Main memory has 1024 blocks with 16 words per block. How many bits are needed in each of the tag, set, and word fields of the main memory address?
(b) A computer system has 32K words of main memory and a set associative cache. The block size is 16 words and the TAG field of the main memory address is 5-bit wide. If the same cache were direct mapped, the main memory will have a 3-bit TAG field. How many words are there in the cache? How many blocks are there in a cache set?

8.29 Under what condition does the set associative mapping method become one of the following?
(a) Direct mapping
(b) Fully associative mapping

8.30 Discuss the main features of Motorola 68020 on-chip cache.

8.31 What is the basic difference between:
(a) Standard I/O and memory-mapped I/O?
(b) Programmed I/O and virtual I/O?
(c) Polled I/O and interrupt I/O?
(d) A subroutine and interrupt I/O?
(e) Cycle-stealing, block transfer, and interleaved DMA?
(f) Maskable and nonmaskable interrupts?
(g) Internal and external interrupts?
(h) Memory mapping in a microprocessor and memory-mapped I/O?

8.32 Explain the significance of interleaved memory organization in pipelined computers .

8.33 Discuss the basic differences between SISD and SIMD.

8.34 The Cray - I computer has one CPU, and 12 functional units. Up to a maximum of 8 functional units can be cascaded to form a chain. Each functional unit is pipelined and the number of pipeline segments vary from 1 to 14. Each functional unit is capable of manipulating 64-bit data. Is it possible to describe this machine using Flynn's approach? Explain.

8.35 Consider a processor array with 4 floating-point processors (FPP). Suppose that each FPP takes 4 time units to produce one result, how long it would take to carry out 100 floating point operations? Is there any performance improvement if the same 100 floating-point operations are carried out using a 4-segment pipelined processor in which each segment takes 1 time unit to produce the result (Ignore latch delay)?

8.36 Explain the significance of masking in array processors.

8.37 Consider the floating-point pipeline discussed in section 8.4.2. Assume:

$T_1 = 40$ ns $T_2 = 100$ ns
$T_3 = 180$ ns $T_4 = 60$ ns
$T_1 = 20$ ns

(a) Determine the pipeline clock rate.
(b) Find the time taken to add 1000 pairs of floating-point numbers using this pipeline.
(c) What is the efficiency of the pipeline when 2000 pairs of floating-point numbers are added?

8.38 Design a pipeline multiplier using carry/save adders (CSA) and carry-look-ahead adders to multiply a stream of input numbers X0, X1, X2, by a fixed number Y. Assume all Xs and Ys are 6-bit numbers. The output should be a stream of 12-bit products YX0, YX1, YX2. Draw a neat schematic diagram of your design.

8.39 Consider the execution of 1000 instructions using a 6-segment pipeline.
(a) What is the average number of instructions executed per instruction cycle when C = 0.2?
(b) What must be the value of C so execution of at least 4 instructions per instruction cycle is always allowed.

8.40 Describe the methods used to handle branches in a pipeline instruction execution unit.

8.41 Modify each of the following programs so the data flow in the 2-segment pipeline (Figure 8.52) is properly regularized:

(a)

MEMORY ADDRESS	INSTRUCTION
2000	LDA X
2001	DCR Y
2002	JMP 2040
2003	SUB Z
:	:
2040	STA W

(b)

MEMORY ADDRESS	INSTRUCTION
2000	LDA X
2001	DCR Y
2002	JNZ 2040
2003	SUB Z
2004	:
:	STAW
2040	