ANNAMACHARYA
INSTITUTE OF TECHNOLOGY AND SCIENCES
(AUTONOMOUS)
Approved by AICTE, New Delhi & Permanent Affiliation to JNTUA, Anantapur.
Three B. Tech Programmes (CSE , ECE & CE) are accredited by NBA, New Delhi,Accredited by NAAC with 'A' Grade , Bangalore.
A-grade awarded by AP Knowledge Mission. Recognized under sections 2(f) & 12(B) of UGC Act 1956.
Venkatapuram Village, Renigunta Mandal, Tirupati, Andhra Pradesh-517520.

Department of Artificial Intelligence

Academic Year 2023-24

II. B.Tech I Semster

Computer Organization (20APC3006)

Prepared By

Mrs. S. Venkata Lakshmi ., M.Tech(Ph.D).
Assistant Professor
Department of CSE, AITS

ANNAMACHARYA INSTITUTE OF TECHNOLOGY AND SCIENCES::TIRUPATI

II B.Tech II Semester

(19APC0506) Computer Organization

L	T	P	C
3	0	0	3

Course Objectives:

- To learn the fundamentals of computer organization and its relevance to classical and modern problems of computer design
- To make the students understand the structure and behavior of various functional modules of a computer.
- To understand the techniques that computers use to communicate with I/O devices
- To study the concepts of pipelining and the way it can speed up processing.
- To understand the basic characteristics of multiprocessors

Unit I:

Basic Structure of Computer: Computer Types, Functional Units, Basic operational Concepts, Bus Structure, Software, Performance, Multiprocessors and Multicomputer.

Machine Instructions and Programs: Numbers, Arithmetic Operations and Programs, Instructions and Instruction Sequencing, Addressing Modes, Basic Input/output Operations, Stacks and Queues, Subroutines, Additional Instructions.

Unit II:

Arithmetic: Addition and Subtraction of Signed Numbers, Design and Fast Adders, Multiplication of Positive Numbers, Signed-operand Multiplication, Fast Multiplication, Integer Division, Floating-Point Numbers and Operations.

Basic Processing Unit: Fundamental Concepts, Execution of a Complete Instruction, Multiple-Bus Organization, Hardwired Control, Multiprogrammed Control.

Unit III:

The Memory System: Basic Concepts, Semiconductor RAM Memories, Read-Only Memories, Speed, Size and Cost, Cache Memories, Performance Considerations, Virtual Memories, Memory Management Requirements, Secondary Storage.

Unit IV:

Input/output Organization: Accessing I/O Devices, Interrupts, Processor Examples, Direct Memory Access, Buses, Interface Circuits, Standard I/O Interfaces.

Unit V:

Pipelining: Basic Concepts, Data Hazards, Instruction Hazards, Influence on Instruction Sets.

Large Computer Systems: Forms of Parallel Processing, Array Processors, The Structure of General-Purpose, Interconnection Networks.

Textbook:

1. “Computer Organization”, Carl Hamacher, Zvonko Vranesic, Safwat Zaky, McGraw Hill Education, 5th Edition, 2013.

Reference Textbooks:

1. Computer System Architecture, M.Morris Mano, Pearson Education, 3rd Edition.
2. Computer Organization and Architecture, Themes and Variations, Alan Clements, CENGAGE Learning.
3. Computer Organization and Architecture, Smruti Ranjan Sarangi, McGraw Hill Education.
4. Computer Architecture and Organization, John P.Hayes, McGraw Hill Education.

Course Outcomes:

- Ability to use memory and I/O devices effectively
- Able to explore the hardware requirements for cache memory and virtual memory
- Ability to design algorithms to exploit pipelining and multiprocessors

UNIT-I

Basic Structure of Computer: Computer Types, Functional Units, Basic operational Concepts, Bus Structure, Software, Performance, Multiprocessors and Multicomputer.

Machine Instructions and Programs: Numbers, Arithmetic Operations and Programs, Instructions and Instruction Sequencing, Addressing Modes, Basic Input/output Operations, Stacks and Queues, Subroutines, Additional Instructions.

Basic Structure of Computer:

1.1 Computer types

A computer can be defined as a fast electronic calculating machine that accepts the (data) digitized input information process it as per the list of internally stored instructions and produces the resulting information. List of instructions are called programs & internal storage is called computer memory.

The different types of computers are

1. Personal computers: - This is the most common type found in homes, schools, Business offices etc., It is the most common type of desk top computers with processing and storage units along with various input and output devices.

2. Note book computers: - These are compact and portable versions of PC

3. Work stations: - These have high resolution input/output (I/O) graphics capability, but with same dimensions as that of desktop computer. These are used in engineering applications of interactive design work.

4. Enterprise systems: - These are used for business data processing in medium to large corporations that require much more computing power and storage capacity than work stations. Internet associated with servers has become a dominant worldwide source of all types of information.

5. Super computers: - These are used for large scale numerical calculations required in the applications like weather forecasting etc.,

1.2 Functional unit:

A computer consists of five functionally independent main parts input, memory, arithmetic logic unit (ALU), and output and control unit.

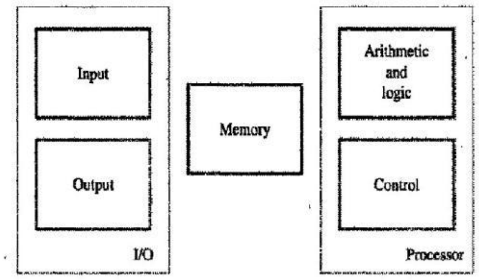


Figure 1.1 Basic functional units of a computer.

Input device accepts the coded information as source program i.e. high level language. This is either stored in the memory or immediately used by the processor to perform the desired operations. The program stored in the memory determines the processing steps. Basically the computer converts one source program to an object program. i.e. into machine language.

Finally the results are sent to the outside world through output device. All of these actions are coordinated by the control unit.

Input unit: -

The source program/high level language program/coded information/simply data is fed to a computer through input devices keyboard is a most common type. Whenever a key is pressed, one corresponding word or number is translated into its equivalent binary code over a cable & fed either to memory or processor. Joysticks, trackballs, mouse, scanners etc are other input devices.

Memory unit: -

Its function is to store programs and data. There are two classes of storage, they are:

1. Primary memory

*2. **Secondary memory***

1. Primary memory: - Is the one exclusively associated with the processor and operates at the electronics speeds programs must be stored in this memory while they are being executed. The memory contains a large number of semiconductors storage cells, each capable of storing one bit of information. These cells are rarely read or written as individual cells but instead are processed in groups of fixed size called words.

To provide easy access to a word in memory, a distinct address is associated with each word location. **Addresses are** numbers that identify memory location. Number of bits in each word is called word length of the computer. Programs must reside in the memory during execution. Instructions and data can be written into the memory or read out under the control of processor.

Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called random-access memory (RAM).

The time required to access one word in called memory access time. Memory which is only readable by the user and contents of which can’t be altered is called read only memory (ROM) it contains operating system.

Caches are the small fast RAM units, which are coupled with the processor and are often contained on the same IC chip to achieve high performance. Although primary storage is essential it tends to be expensive.

2. Secondary Memory: - Is used where large amounts of data & programs have to be stored, particularly information that is accessed infrequently.

Examples: Magnetic disks & tapes, optical disks (ie CD-ROM's), floppies etc.,

Arithmetic logic unit (ALU):-

Most of the computer operators are executed in ALU of the processor like addition, subtraction, division, multiplication, etc. the operands are brought into the ALU from memory and stored in high speed storage elements called register. Then according to the instructions the operation is performed in the required sequence.

The control and the ALU are many times faster than other devices connected to a computer system. This enables a single processor to control a number of external devices such as key boards, displays, magnetic and optical disks, sensors and other mechanical controllers.

Output unit:-

These actually are the counterparts of input unit. Its basic function is to send the processed results to the outside world.

Examples: Printer, speakers, monitor etc.,

Control unit:-

It effectively is the nerve center that sends signals to other units and senses their states. The actual timing signals that govern the transfer of data between input unit, processor, memory and output unit are generated by the control unit.

The operation of a computer can be summarized as follows:

- The computer accepts information in the form of programs and data through an input unit and stores it in the memory.
- Information stored in the memory is fetched, under program controi, into an arithmetic and logic unit, where it is processed.
- Processed information leaves the computer through an output unit.
- All activities inside the machine are directed by the control unit.

1.3 Basic operational concepts

To perform a given task an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be stored are also stored in the memory.

Examples: - Add LOCA, R0

This instruction adds the operand at memory location LOCA, to operand in register R0 & places the sum into register. The original contents of location LOCA are preserved, whereas those of RO are overwritten. This instruction requires the performance of several steps,

1. First the instruction is fetched from the memory into the processor.

2. The operand at LOCA is fetched and added to the contents of R0

3. Finally the resulting sum is stored in the register R0

The preceding **Add** instruction combines a memory access operation with an ALU Operations. In some other type of computers, these two types of operations are performed by separate instructions for performance reasons.

Load LOCA, R1

Add R1, R0

The first of these instructions transfers the contents of memory location **LOCA** into processor register **R1**, and the second instruction adds the contents of registers **RI** and **RO** and places the sum into **RO**.

Transfers between the memory and the processor are started by sending the address of the memory location to be accessed to the memory unit and issuing the appropriate control signals. The data are then transferred to or from the memory.

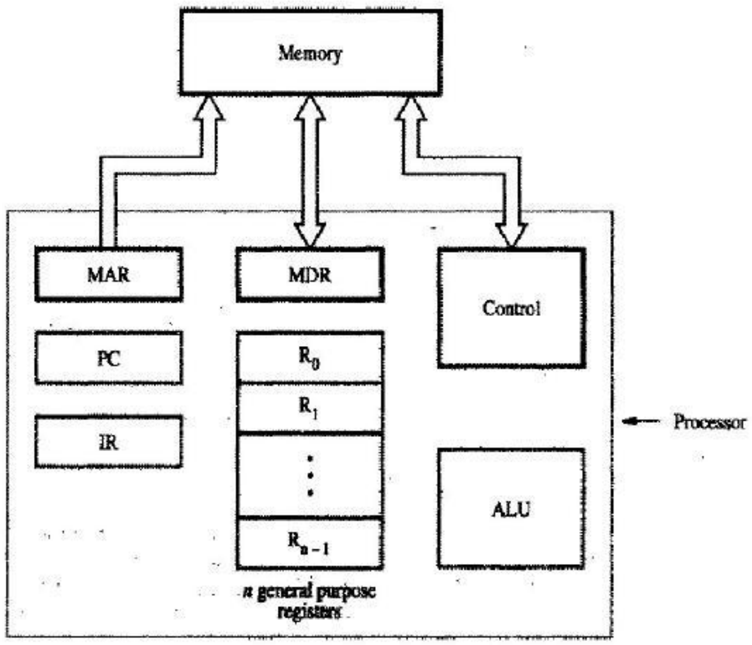


Figure 1.2 Connections between the processor and the memory.

The fig shows how memory & the processor can be connected. In addition to the ALU & the control circuitry, the processor contains a number of registers used for several different purposes.

The instruction register (IR):- Holds the instruction that is currently being executed. Its output is available for the control circuits which generates the timing signals that control the various processing elements in one execution of instruction.

The program counter PC:-

This is another specialized register that keeps track of execution of a program. It contains the memory address of the next instruction to be fetched and executed. Besides **IR** and **PC**, there are n-general purpose registers R_0 through R_{n-1}.

The other two registers which facilitate communication with memory are:

1. MAR – (Memory Address Register):- It holds the address of the location to be accessed.

2. MDR – (Memory Data Register):- It contains the data to be written into or read out of the address location.

Operating steps are

1. Programs reside in the memory & usually get these through the Input unit.
2. Execution of the program starts when the PC is set to point at the first instruction of the program.
3. Contents of PC are transferred to MAR and a Read Control Signal is sent to the memory.
4. After the time required to access the memory elapses, the address word is read out of the memory and loaded into the MDR.
5. Now contents of MDR are transferred to the IR & now the instruction is ready to be decoded and executed.
6. If the instruction involves an operation by the ALU, it is necessary to obtain the required operands.
7. An operand in the memory is fetched by sending its address to MAR & Initiating a read cycle.
8. When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU.
9. After one or two such repeated cycles, the ALU can perform the desired operation.
10. If the result of this operation is to be stored in the memory, the result is sent to MDR.
11. Address of location where the result is stored is sent to MAR & a write cycle is initiated.
12. The contents of PC are incremented so that PC points to the next instruction that is to be executed.

Normal execution of a program may be preempted (temporarily interrupted) if some devices require urgent servicing, to do this one device raises an Interrupt signal.

An interrupt is a request signal from an I/O device for service by the processor. The processor provides the requested service by executing an appropriate interrupt service routine.

The Diversion may change the internal stage of the processor its state must be saved in the memory location before interruption. When the interrupt-routine service is completed the state of the processor is restored so that the interrupted program may continue.

1.4 Bus STRUCTURES

To achieve a reasonable speed of operation, a computer must be organized so that all its units can handle one full word of data at a given time. A group of lines that serve as a connecting port for several devices is called a **bus**.

‘The simplest way to interconnect functional units is to use a single bus, as shown in *Figure 1.3.* Ail units are connected to this bus. Because the bus can be used for only one transfer at a time, only two units can actively use the bus at any given time. Bus control lines are used to arbitrate multiple requests for use of one bus.

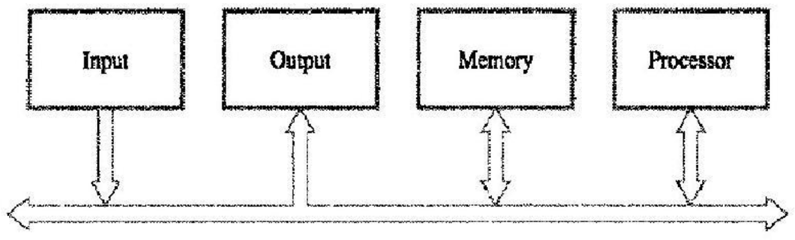


Figure 1.3 Single-bus structure.

Single bus structure is

- Low cost
- Very flexible for attaching peripheral devices

Multiple bus structure certainly increases the performance but also increases the cost significantly.

All the interconnected devices are not of same speed & time leads to a bit of a problem. This is solved by using cache registers (ie buffer registers). These buffers are electronic registers of small capacity when compared to the main memory but of comparable speed.

The instructions from the processor at once are loaded into these buffers and then the complete transfer of data at a fast rate will take place.

1.5 Software

System software is a collection of programs that are executed as needed to perform functions such as,

- Receiving and interpreting user commands
- Entering and editing application programs and storing them as files in secondary storage devices
- Managing the storage and retrieval of files in secondary storage devices
- Running standard application programs such as word processors, spreadsheets, or games, with data supplied by the user
- Controlling 1/O units to receive input information and produce output results
- Translating programs from source form prepared by the user into object form consisting of machine instructions

- Linking and running user-written application programs with existing standard library routines, such as numerical computation packages

System software is thus responsible for the coordination of all activities in a computing system.

Application programs are usually written in a high-level programming language, such as C, C++, Java, or FORTRAN, in which the programmer specifies mathematical or text-processing operations.

A *system software* program called a **compiler** translates the high-level language program into a suitable machine language program.

Another important system program is **Text editor;** it is used for entering and editing application programs. The user of this program interactively executes commands that allow statements of a source program entered at a keyboard to be accumulated in a file.

A **file** is simply a sequence of alphanumeric characters or binary data that is stored in memory or in secondary storage. A file can be referred to by a name chosen by the user.

Operating system (OS) is a large program, or actually a collection of routines, that is used to control the sharing of and interaction among various computer units as they execute application programs. The OS routines perform the tasks required to assign computer resources to individual application programs. These tasks include assigning memory and magnetic disk space to program and data files, moving data between memory and disk units, and handling I/O operations.

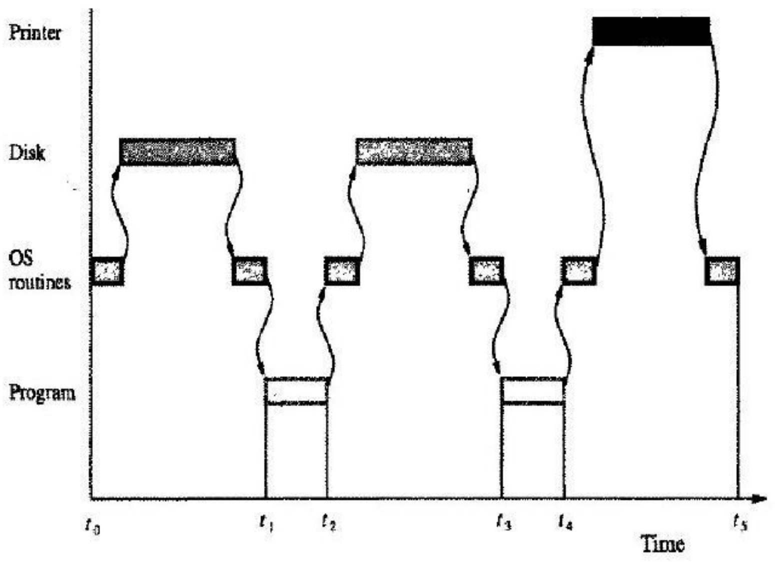


Figure 1.4 User program and OS routine sharing of the processor.

During the time period t_0 to t_1, OS routine initiates loading the application program from disk to memory, waits until the transfer is completed, and then passes execution control to the application program. A similar pattern of activity occurs during period t_2, to t_3 and period t_4 to t_5, when the operating system transfers the data file from the disk and prints the results. At t_5, the operating system may load and execute another application program.

Now, let us point out a way that computer resources can be used more efficiently if several application programs are to be processed. Notice that the disk and the processor are idle during most of the time period t_4 to t_5. The operating system can load the next program to be executed into the memory from the disk while the printer is operating. Sirnilarly, during t_0 to t_1, the operating system can arrange to print the previous program's results while the current program is being loaded from the disk.

Thus, the operating system manages the concurrent execution of several application programs to make the best possible use of computer resources. This pattern of concurrent execution is called **multiprogramming** or **multitasking**.

1.6 Performance

The most important measure of the performance of a computer is how quickly it can execute programs. The speed with which a computer executes program is affected by the design of its hardware. For best performance, it is necessary to design the compiles, the machine instruction set, and the hardware in a coordinated way.

The total time required to execute the program is elapsed time is a measure of the performance of the entire computer system. It is affected by the speed of the processor, the disk and the printer. The time needed to execute a instruction is called the processor time.

Just as the elapsed time for the execution of a program depends on all units in a computer system, the processor time depends on the hardware involved in the execution of individual machine instructions. This hardware comprises the processor and the memory which are usually connected by the bus as shown in the fig c.

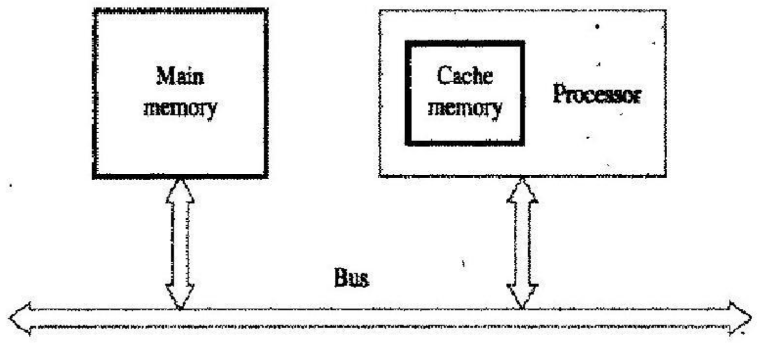


Figure 1.5 The processor cache.

The pertinent parts of the fig. 1.3 are repeated in fig. d which includes the cache memory as part of the processor unit.

Let us examine the flow of program instructions and data between the memory and the processor. At the start of execution, all program instructions and the required data are stored in the main memory. As the execution proceeds, instructions are fetched one by one over the bus into the processor, and a copy is placed in the cache later if the same instruction or data item is needed a second time, it is read directly from the cache.

The processor and relatively small cache memory can be fabricated on a single IC chip. The internal speed of performing the basic steps of instruction processing on chip is very high and is considerably faster than the speed at which the instruction and data can be fetched from the main memory. A program will be executed faster if the movement of instructions and data between the main memory and the processor is minimized, which is achieved by using the cache.

For example: Suppose a number of instructions are executed repeatedly over a short period of time as happens in a program loop. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. The same applies to the data that are used repeatedly.

Processor clock:

Processor circuits are controlled by a timing signal called clock. The clock designer the regular time intervals called clock cycles. To execute a machine instruction the processor divides the action to be performed into a sequence of basic steps that each step can be completed in one clock cycle. The length P of one clock cycle is an important parameter that affects the processor performance.

Processor used in today's personal computer and work station has a clock rates that range from a few hundred million to over a billion cycles per second.

MULTIPROCESSORS AND MULTICOMPUTERS:

- Large computers that contain a number of processor units are called multiprocessor system.
- These systems either execute a number of different application tasks in parallel or execute subtasks of a single large task in parallel.
- All processors usually have access to all memory locations in such system & hence they are called shared memory multiprocessor systems.
- The high performance of these systems comes with much increased complexity and cost.
- In contrast to multiprocessor systems, it is also possible to use an interconnected group of complete computers to achieve high total computational power. These computers normally have access to their own memory units when the tasks they are executing need to communicate data they do so by exchanging messages over a communication network. This properly distinguishes them from shared memory multiprocessors, leading to name message-passing multi computer.

MACHINE INSTRUCTIONS AND PROGRAMS

2.1 NUMBERS, ARITHMETIC OPERATIONS, AND CHARACTERS

Computers are built using logic circuits that operate on information represented by two valued electrical signals. We label the two values as 0 and 1; and we define the amount of information represented by such a signal as a *bit* of information, where *bit* stands for *binary digit*. The most natural way to represent a number in a computer system is by a string of bits, called a binary number. A text character can also be represented by a string of bits called a *character code*.

2.1.1 NUMBER REPRESENTATION

Consider an n-bit vector

$$B=b_{n-1} \ldots B_1 b_0$$

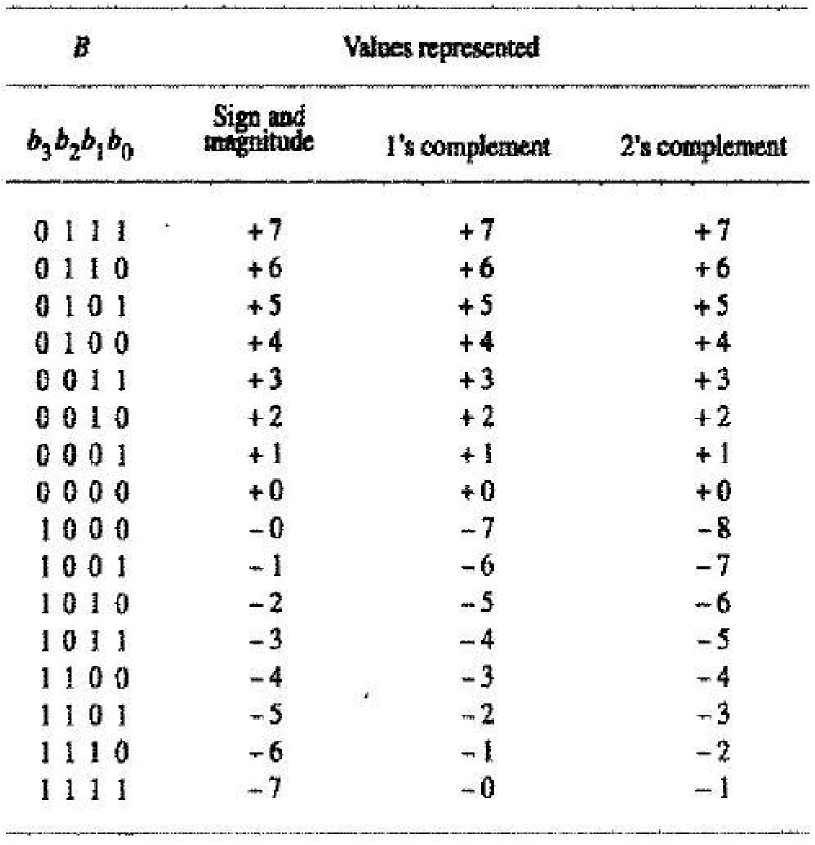
Where b_i=0 or 1for $0 \le i \le n — 1$. This vector can represent unsigned integer values V in the range 0 to 2^n -1, where

$$V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

We obviously need to represent both positive and negative numbers. Three systems are used for representing such numbers:

- ✓ Sign-and-magnitude
- ✓ 1's-complement
- ✓ 2's-complement

In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers. *Fig 2.1* illustrates all three representations using 4-bit numbers. Positive values have identical representations in al systems, but negative values have different representations. In the *sign-and-magnitude* systems, negative values are represented by changing the most significant bit (b3 in figure 2.1) from 0 to 1 in the B vector of the corresponding positive value. For example, +5 is represented by 0101, and -5 is represented by 1101.



B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

Figure 2.1 Binary, signed-integer representations.

In 1's- complement representation, negative values are obtained by complementing each bit of the corresponding positive number. Thus, the representation for -3 is obtained by complementing each bit in the vector 0011 to yield 1100. Clearly, the same operation, bit complementing, is done in converting a negative number to the corresponding positive value. Converting either way is referred to as forming the 1's-complement of a given number. Finally, in the 2's-complement system, forming the 2's-complement of a number is done by subtracting that number from 2^n. Hence, the 2's complement of a number is obtained by adding 1 to the 1's complement of that number.

Addition of Positive numbers:-

Consider adding two 1-bit numbers. The results are shown in figure 2.2. Note that the sum of 1 and 1 requires the 2-bit vector 10 to represent the value 2. We say that the sum is 0 and the carry-out is 1. In order to add multiple-bit numbers, we use a method analogous to that used for manual computation with decimal numbers. We add bit pairs starting from the low-order (right) and of the bit vectors, propagating carries toward the high-order (left) end.

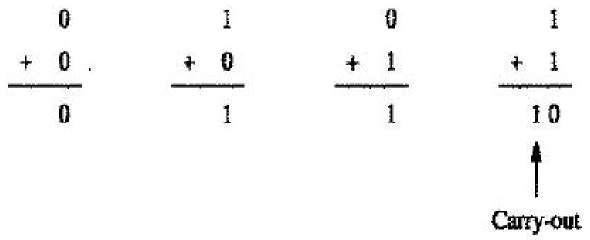


Figure 2.2 Addition of 1-bit numbers.

INSTRUCTIONS AND INSTRUCTION SEQUENCING

A computer must have instructions capable of performing four types of operations.

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

REGISTER TRANSFER NOTATION:-

Transfer of information from one location in the computer to another. Possible locations that may be involved in such transfers are memory locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify a location by a symbolic name standing for its hardware binary address. For *Example*, names for the addresses of memory locations may be **LOC, PLACE, A, VAR2**; processor registers names may be **R0, R5**; and I/O register names may be **DATAIN, OUTSTATUS**, and so on. The contents of a location are denoted by placing square brackets around the name of the location. Thus, the expression

$$R1 \leftarrow [LOC]$$

Means that the contents of memory location **LOC** are transferred into processor register **R1**.

As another example, consider the operation that adds the contents of registers **R1** and **R2**, and then places their sum into register R3. This action is indicated as

$$R3 \leftarrow [R1] + [R2]$$

This type of notation is known as *Register Transfer Notation (RTN)*. Note that "the right-hand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be places, overwriting the old contents of that location".

ASSEMBLY LANGUAGE NOTATION:-

Assembly language format is another type of notation to represent machine instructions and programs. For example, an instruction that causes the transfer described above, from memory location LOC to processor register R1, is specified by the statement

```
MOV LOC,R1
```

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

The second example of adding two numbers contained in processor registers R1 and R2 and placing their sum in R3 can be specified by the assembly language statement

```
Add R1,R2,R3
```

BASIC INSTRUCTIONS:

The operation of adding two numbers is a fundamental capability in any computer. The statement

$$C = A + B$$

In a high-level language program is a command to the computer to add the current values of the two variables called A and B, and to assign the sum to a third variable, C. When the program containing this statement is compiled, the three variables, A, B, and C, are assigned to distinct locations in the memory. We will use the variable names to refer to the corresponding memory location addresses. The contents of these locations represent the values of the three variables. Hence, the above high-level language statement requires the action to take place in the computer.

$$C \leftarrow [A] + [B]$$

To carry out this action, the contents of memory locations **A** and **B** are fetched from the memory and transferred into the processor where their sum is computed. This result is then sent back to the memory and stored in location **C**.

Let us first assume that this action is to be accomplished by a single machine instruction. Furthermore, assume that this instruction contains the memory addresses of the three operands, A, B, and C. This three-address instruction can be represented symbolically as

Add A,B,C

Operands A and B are called the *source operands*, C is called the *destination operand*, and Add is the operation to be performed on the operands. A general instruction of this type has the format

Operation Source1, Source2, Destination

If k bits are needed for specify the memory address of each operand, the encoded form of the above instruction must contain 3k bits for addressing purposes in addition to the bits needed to denote the Add operation.

An alternative approach is to use a sequence of simpler instructions to perform the same task, with each instruction having only one or two operands. Suppose that two- address instructions of the form

Operation Source, Destination

are available. An Add instruction of this type is

Add A,B

which performs the operation **B←[A]+[B]**. When the sum is calculated, the result is sent to the memory and stored in location B, replacing the original contents of this location. This means that operand B is both a *source* and a *destination*.

A single two-address instruction cannot be used to solve our original problem, which is to add the contents of locations A and B, without destroying either of them, and to place the sum in location C. The problem can be solved by using another two-address instruction that copies the contents of one memory location into another. Such an instruction is

Move B,C

Which performs the operations C ← [B], leaving the contents of location B unchanged. Using only one-address instructions, the operation C← [A] + [B] can be performed by two instruction sequence

Move B,C

Add A,C

Thus, the one-address instruction

Add A

means the following: Add the contents of memory location A to the contents of the accumulator register and place the sum back into the accumulator. Let us also introduce the one-address instructions

Load A

And

Store A

The Load instruction copies the contents of memory location A into the accumulator, and the Store instruction copies the contents of the accumulator into memory location A. Using only one-address instructions, the operation C ← [A] + [B] can be performed by executing the sequence of instructions

Load A

Add B

Store C

Some early computers were designed around a single accumulator structure. Most modern computers have a number of general-purpose processor registers – typically 8 to 32, and even considerably more in some cases. Access to data in these registers is much faster than to data stored in memory locations because the registers are inside the processor.

Let Ri represent a general-purpose register. The instructions

Load A, R_i

Store R_i, A and

Add A, R_i

Are generalizations of the Load, Store, and Add instructions for the single-accumulator case, in which register Ri performs the function of the accumulator.

When a processor has several general-purpose registers, many instructions involve only operands that are in the register. In fact, in many modern processors, computations can be performed directly only on data held in processor registers. Instructions such as

Add R_i, R_j

Or

Add R_i, R_j, R_k

In both of these instructions, the source operands are the contents of registers R_i and R_j. In the first instruction, R_j also serves as the destination register, whereas in the second instruction, a third register, R_k, is used as the destination.

It is often necessary to transfer data between different locations. This is achieved with the instruction

Move Source, Destination

When data are moved to or from a processor register, the Move instruction can be used rather than the Load or Store instructions because the order of the source and destination operands determines which operation is intended. Thus,

Move A, Ri

Is the same as

Load A, Ri

And

Move Ri, A

Is the same as

Store Ri, A

In processors where arithmetic operations are allowed only on operands that are processor registers, the C = A + B task can be performed by the instruction sequence

Move A, Ri

Move B, Rj

Add Ri, Rj

Move Rj, C

In processors where one operand may be in the memory but the other must be in register, an instruction sequence for the required task would be

Move A, Ri

Add B, Ri

Move Ri, C

The speed with which a given task is carried out depends on the time it takes to transfer instructions from memory into the processor and to access the operands referenced by these instructions. Transfers that involve the memory are much slower than transfers within the processor.

We have discussed three-, two-, and one-address instructions. It is also possible to use instructions in which the locations of all operands are defined implicitly. Such instructions are found in machines that store operands in a structure called a pushdown stack. In this case, the instructions are called zero-address instructions.

INSTRUCTION EXECUTION AND STRAIGHT-LINE SEQUENCING:

The three instructions of the program are in successive word locations, starting at location i. Since each instruction is 4 bytes long, the second and third instructions start at addresses i + 4 and i + 8.

Let us consider how below program is executed. The processor contains a register called the program counter (PC), which holds the address of the instruction to be executed next.

To begin executing a program, the address of its first instruction (i in our example) must be placed into the *PC*. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called straight-line sequencing. During the execution of each instruction, the PC is *incremented by 4* to point to the next instruction. Thus, after the Move instruction at location i + 8 is executed, the PC contains the value i + 12, which is the address of the first instruction of the next program segment.

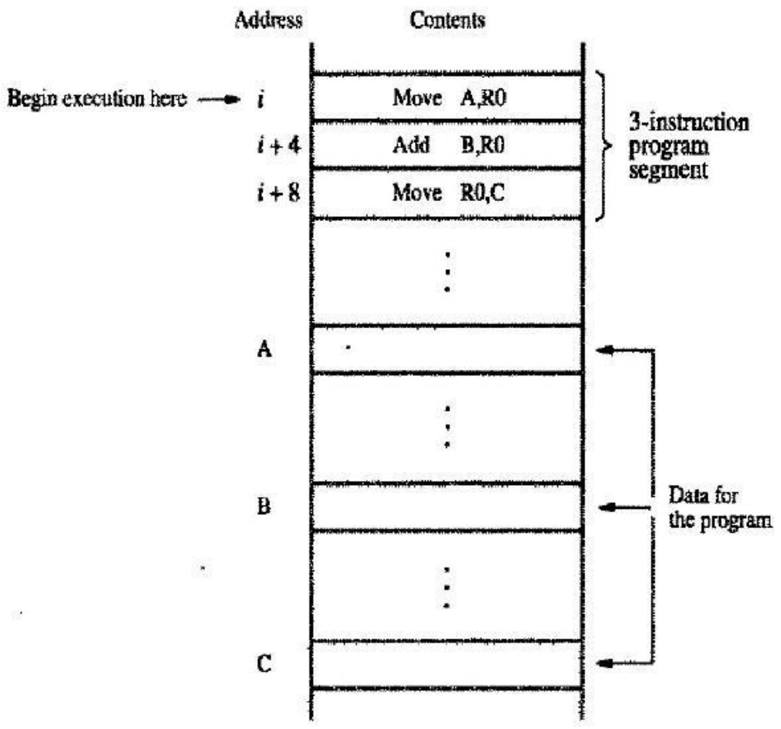


Figure 2.8 A program for C ← [A] + [B].

Executing a given instruction is a two-phase procedure: ***instruction fetch*** & ***instruction execute***.

In the first phase the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the *instruction register (IR)* in the processor.

At the start of the second phase the instruction in *IR* is examined to determine which operation is to be performed.

BRANCHING:

Consider the task of adding a list of n numbers. The addresses of the memory locations containing the **n** numbers are symbolically giver as **NUM1, NUM2... NUMn** and a separate **Add** instruction is used to add each number to the contents of register **R0**. After all the numbers have been added, the result is placed in memory location **SUM**.

The loop is a straight-line sequence of instructions executed as many times as needed. It starts at location **LOOP** and ends at the instruction **Branch>0**. During each pass through this loop, the address of the next list entry is determined, and that entry is fetched and added to RO.

Assume that the number of entries in the list, 2, is stored in memory location N. Register R1 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are leaded into register R1 at the beginning of the program. Then, within the body of the loop, the instruction

Decrement R1

reduces the contents of RI by 1 each time through the Loop. Execution of the loop is repeated as long as the result of the decrement operation is greater than zero.

Branch instruction loads a new value into the program counter. The processor fetches and executes the instruction at this new address, called the *branch target*. Conditional branch instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

Branch>0 LOOP

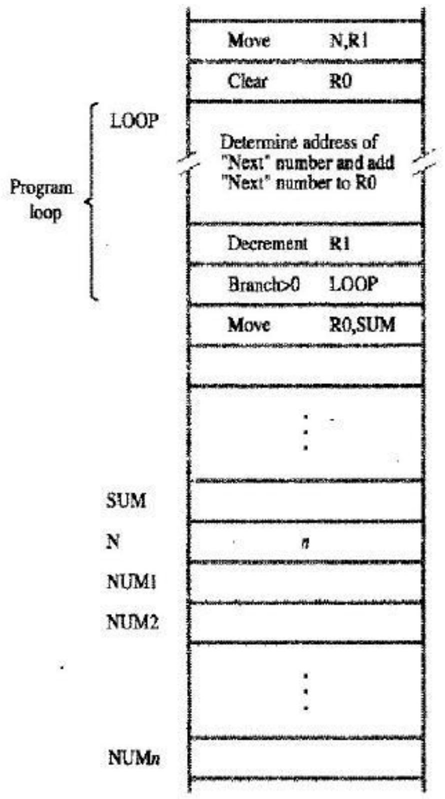


Figure 2.10 Using a loop to add *n* numbers.

The Move instruction is fetched and executed. It moves the final result from R0 into memory location SUM.

CONDITION CODES:

The processor keeps track of information about the results of various operations for use by subsequent conditional branch instructions. This is accomplished by recording the required information in individual bits, often called condition code flags. These flags are usually grouped together in a special processor register called the condition code register or status register. Individual condition code flags are set to 1 or cleared to 0, depending on the outcome of the operation performed.

Four commonly used flags are:

N (negative)	Set to 1 if the result is negative; otherwise, cleared to 0
Z (zero)	Set to 1 if the result is 0; otherwise, cleared to 0
V (overfiow}	Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
C (carry)	Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

The **N** and **Z** flags indicate whether the result of an arithmetic or logic operation is negative or zero. The N and Z flags may also be affected by instructions that transfer data, such as Move, Load, or Store.
The **V** flag indicates whether overflow has taken place. The processor sets the V flag to allow the programmer to test whether overflow has occurred and branch to an appropriate routine that corrects the problem. Instructions such as BranchlfOverfiow are provided for this purpose.

The **C** flag is set to 1 if a carry occurs from the most significant bit position during an arithmetic operation. This flag makes it possible to perform arithmetic operations on operands that are longer than the word length of the processor.

The instruction Branch>0, tests one or more of the condition flags.

GENERATING MEMORY ADDRESSES:

Suppose that a processor register, Ri, is used to hold the memory address of an operand, H it is initially loaded with the address NUM1 before the loop is entered and is then incremented by 4 on each pass through the loop, it can provide the needed capability.

ADDRESSING MODES:

The different ways in which the location of an operand is specified in an instruction are referred to as *addressing modes*.

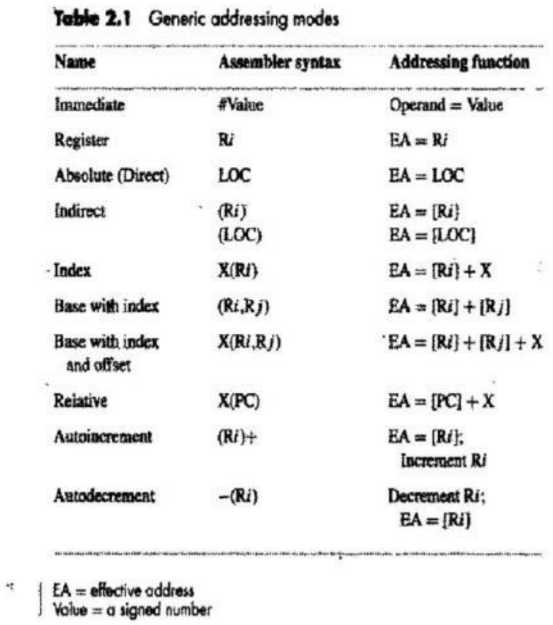


Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC

Indirect	(R*i*)
	(LOC)
		EA = [R*i*]
		EA = [LOC]

Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*]; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i*; EA = [R*i*]

EA = effective address
Value = a signed number

IMPLEMENTATION OF VARIABLES AND CONSTANTS:

Variables and constants are the simplest data types and are found in almost every computer program. In assembly language, a variable is represented by allocating a register or a memory location to hold its value. Thus, the value can be changed as needed using appropriate instructions.

Register mode: *The operand is the contents of a processor register; the name (address) of the register is given in the instruction.*

Absolute mode: *The operand is in a memory location; the address of this location is given explicitly in the instruction. (In some assembly languages, this mode is called Direct.)*

The instruction

```
MOVE LOC, R2
```

uses these two modes. Processor registers are used as temporary storage locations where the data in a register are accessed using the Register mode. The Absolute mode can represent global variables in a program. A declaration such as

```
Integer A, B;
```

in a high-level language program will cause the compiler to allocate a memory location to each of the variables A and B.

Immediate mode: *The operand is given explicitly in the instruction.*

For example, the instruction

$$\text{Move } 200_{\text{immediae}}, \text{R0}$$

places the value 200 in register R0.

The Immediate mode is only used to specify the value of a source operand.

A common convention is to use the sharp sign (#) in front of the value to indicate that this value is to be used as an immediate operand.

```
Move #200, R0
```

Constant values are used frequently in high-level language programs.

For example, the statement

```
A=B+6
```

contains the constant 6, Assuming that A and B have been declared earlier as variables and may be accessed using the Absolute mode, this statement may be compiled asfollows:

```
Move B,R1

Add #6,R1

Move R1,A
```

INDIRECTION AND POINTERS

In the addressing modes that follow, the instruction provides information from which the memory address of the operand can be determined. We refer to this address as the *effective address (EA)* of the operand.

Indirect mode: The effective address of the operand is the contents of a register or memory location whose address appears in the instruction.

Indirection will be denoted by placing the name of the register or the memory address given in the instruction in parentheses.

To execute the Add instruction in Figure 2.11a, the processor uses the value B, which is in register R1, as the effective address of the operand. It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds to the contents of register R0. Indirect addressing through a memory location is also possible as shown in Figure 2.11b. In this case, the processor first reads the contents of memory location A, and then requests a second read operation using the value B as an address to obtain the operand.

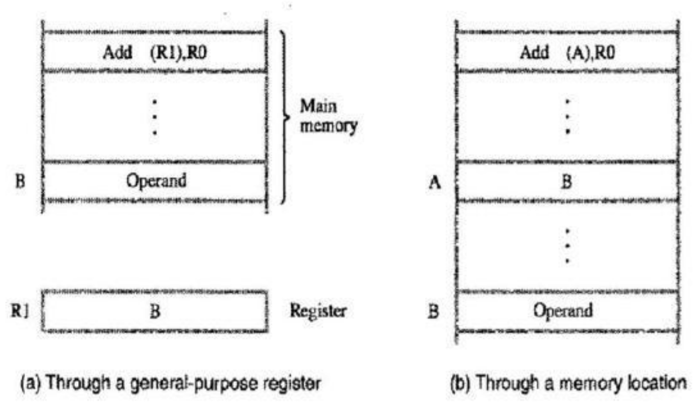

Add (R1),R0
Main memory
B
Operand
R1
B
Register
(a) Through a general-purpose register
Add (A),R0
A
B
B
Operand
(b) Through a memory location

Figure 2.11 Indirect addressing.

The register or memory location that contains the address of an operand is called a pointer.

For adding a list of numbers, indirect addressing can be used to access successive numbers in the list, resulting in the program shown in Figure 2.12. Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2. The initialization section of the program loads the counter value n from memory location N into R1 and uses the immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0. The first time through the loop, the instruction

```
Add (R2),R0
```

fetches the operand at location NUMI and adds it to R0. The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

Consider the C-language statement

```
A=*B
```

where B is a pointer variable. This statement may be compiled into

```
Move B,R1
Move (R1),A
```

Using indirect addressing through memory, the same action can be achieved with

```
Move (B),A
```

INDEXING AND ARRAYS:

It is useful in dealing with lists and arrays.

Index mode: The effective address of the operand is generated by adding a constant value to the contents of a register.

The register used may be either a special register provided for this purpose, or, more commonly, it may be any one of a set of general-purpose registers in the processor. In either case, it is referred to as an *index register*.

Index mode can be indicated symbolically as

`X(Ri)`

where `X` denotes the constant value contained in the instruction and `Ri` is the name of the register involved. The effective address of the operand is given by

`EA =X+[Ri]`

The contents of the index register are not changed in the process of generating the effective address.

In an assembly language program, the constant X may be given cither as an explicit number or as a symbolic name representing a numerical value.

In Figure 2.13a, the index register, Ri, contains the address of a memory location, and the value *X defines an offset* (also called a displacement) from this address to the location where the operand is found.

An alternative use is illustrated in Figure 2.13b. Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand.

In either case, the effective address is the sum of two values; *one is given explicitly in the instruction, and the other is stored in a register*.

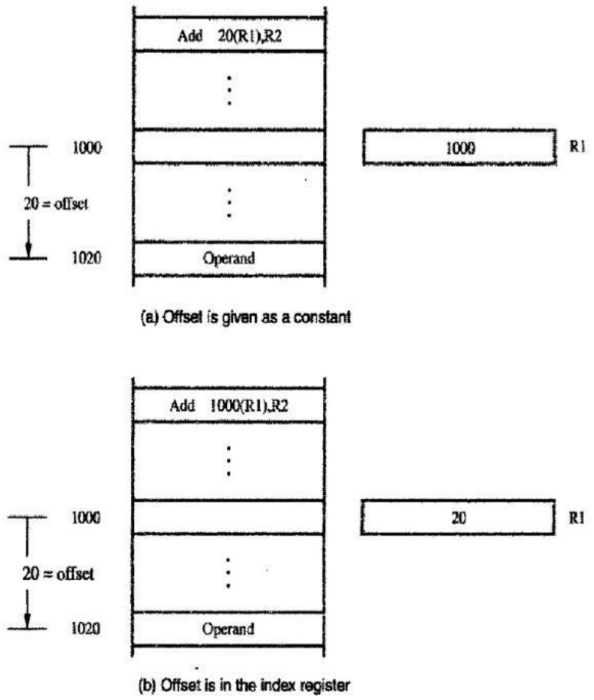


Figure 2.13 Indexed addressing.

To see the usefulness of indexed addressing, consider a simple example involving a list of test scores for students taking a given course. Assume that the list of scores, beginning at location LIST, is structured as shown in Figure 2.14. A four-word memory block comprises a record that stores the relevant information for each student. Each record consists of the student's identification number (ID), followed by the scores the student earned on three tests. There are n students in the class, and the value n is stored in location N immediately in front of the list. The addresses given in the figure for the student IDs and test scores assume that the memory is byte addressable and that the word length is 32 bits

Each row contains the entries for one student, and the columns give the IDs and test scores.

In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears

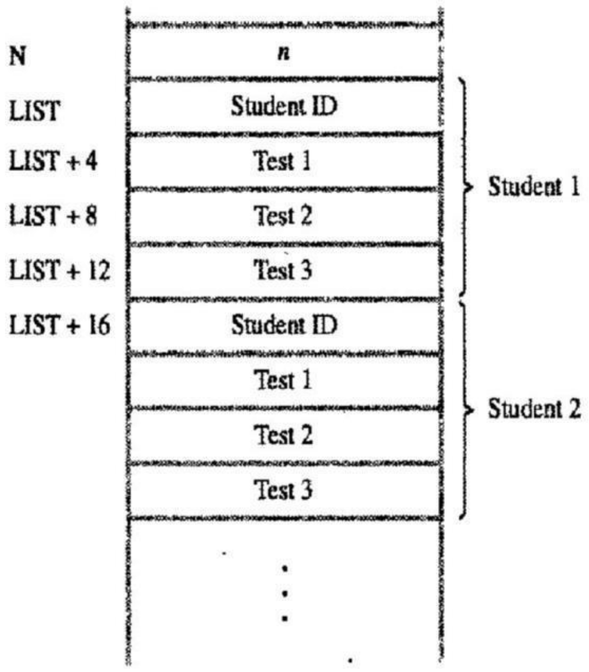
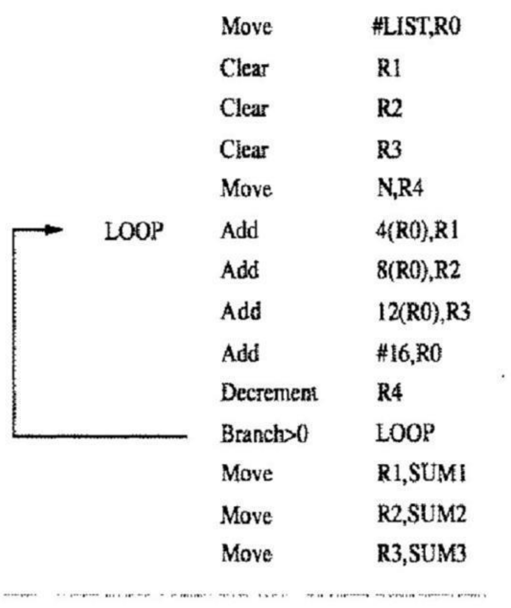


Figure 2.14 A list of students' marks.

Several variations of basic form provide for very efficient access to memory operands in practical programming situations. For Example,

```
(Ri Rj)
```

The effective address is the sum of the contents of registers Ri and Rj. The second register is usually called the base register.



```
        Move        #LIST,R0
        Clear       R1
        Clear       R2
        Clear       R3
        Move        N,R4
LOOP    Add         4(R0),R1
        Add         8(R0),R2
        Add         12(R0),R3
        Add         #16,R0
        Decrement   R4
        Branch>0    LOOP
        Move        R1,SUM1
        Move        R2,SUM2
        Move        R3,SUM3
```

Figure 2.15 Indexed addressing used in accessing test scores in the list in Figure 2.14.

RELATIVE ADDRESSING:

Then, X(PC) can be used to address a memory location that is X bytes away from the location presently pointed to by the program counter. Since the addressed location is identified "relative" to the program counter, which always identifies the current execution point in a program, the name Relative mode is associated with this type of addressing.

Relative mode: The effective address is determined by the Index mode using the program counter in place of the general-purpose register Ri.

This mode can be used to access data operands. But, it's most common use is to specify the target address in branch instructions. An instruction such as

```
Branch>0 LOOP
```

causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied.

APDITIONAL MODES:

Autoincrement mode: The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.

Autoincrement mode can be denoted by putting the specified register in parentheses, to show that the contents of the register are used as the effective address, followed by a plus sign to indicate that these contents are to be incremented after the operand is accessed. Thus, the Autoincrement mode is written as

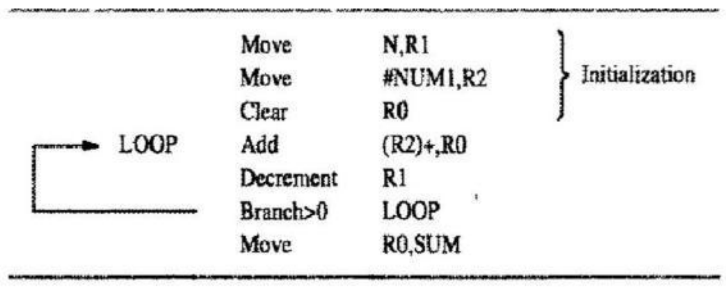
(Ri)+

Implicitly, the increment amount is 1 when the mode is given in this form.

Autodecrement mode: The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.

Autodecrement mode can be denoted by putting the specified register in parentheses, preceded by a minus sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus, we write

-(Ri)



```
        Move        N,R1
        Move        #NUM1,R2
        Clear       R0
LOOP    Add         (R2)+,R0
        Decrement   R1
        Branch>0    LOOP
        Move        R0,SUM
```

Initialization

Figure 2.16 The Autoincrement addressing mode used in the program of Figure 2.12.

In this mode, operands are accessed in descending address order.

2.3 Basic input/output operations

We now examine the means by which data are transferred between the memory of a computer and the outside world. Input/Output (I/O) operations are essential, and the way they are performed can have a significant effect on the performance of the computer.

Consider a task that reads in character input from a keyboard and produces character output on a display screen. A simple way of performing such I/O tasks is to use a method known as program-controlled I/O. The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second. The rate of output transfers from the computer to the display is much higher. It is determined by the rate at which characters can be transmitted over the link between the computer and the display device, typically several thousand characters per second. However, this is still much slower than the speed of a processor that can execute many millions of instructions per second. The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.

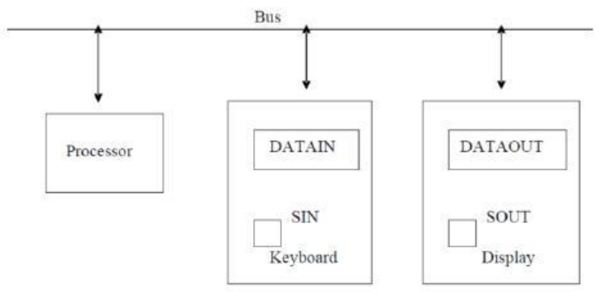


Fig a Bus connection for processor, keyboard, and display

The keyboard and the display are separate device as shown in fig a. the action of striking a key on the keyboard does not automatically cause the corresponding character to be displayed on the screen. One block of instructions in the I/O program transfers the character into the processor, and another associated block of instructions causes the character to be displayed.

Striking a key store the corresponding character code in an 8-bit buffer register associated with the keyboard. Let us call this register DATAIN, as shown in fig a. To inform the processor that a valid character is in DATAIN, a status control flag, SIN, is set to 1. A program monitors SIN, and when SIN is set to 1, the processor reads the contents of DATAIN. When the character is transferred to the processor, SIN is automatically cleared to 0. If a second character is entered at the keyboard, SIN is again set to 1, and the processor repeats.

An analogous process takes place when characters are transferred from the processor to the display. A buffer register, DATAOUT, and a status control flag, SOUT, are used for this transfer. When SOUT equals 1, the display is ready to receive a character.

In order to perform I/O transfers, we need machine instructions that can check the state of the status flags and transfer data between the processor and the I/O device. These instructions are similar in format to those used for moving data between the processor and the memory. For example, the processor can monitor the keyboard status flag SIN and transfer a character from DATAIN to register R1 by the following sequence of operations.

2.4 Stacks and queues

A computer program often needs to perform a particular subtask using the familiar subroutine structure. In order to organize the control and information linkage between the main program and the subroutine, a data structure called a stack is used. This section will describe stacks, as well as a closely related data structure called a queue.

Data operated on by a program can be organized in a variety of ways. We have already encountered data structured as lists. Now, we consider an important data structure known as a stack. A stack is a list of data elements, usually words or bytes, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. Another descriptive phrase, last-in-first-out (LIFO) stack, is also used to describe this type of storage mechanism; the last data item placed on the stack is the first one removed when retrieval begins. The terms push and pop are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

Fig b shows a stack of word data items in the memory of a computer. It contains numerical values, with 43 at the bottom and -28 at the top. A processor register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the stack pointer (SP). It could be one of the general-purpose registers or a register dedicated to this function.

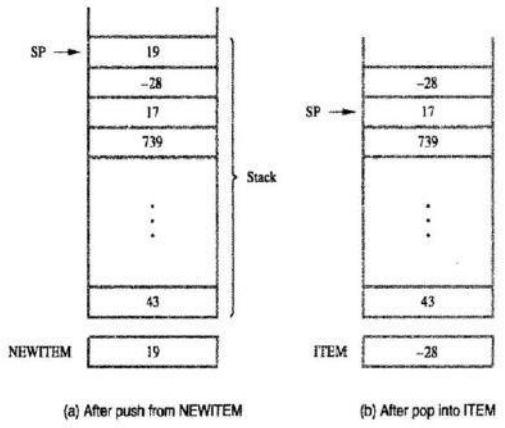


Figure 2.22 Effect of stack operations on the stack in Figure 2.21.

Another useful data structure that is similar to the stack is called a queue. Data are stored in and retrieved from a queue on a first-in-first-out (FIFO) basis. Thus, if we assume that the queue grows in the direction of increasing addresses in the memory, which is a common practice, new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.

There are two important differences between how a stack and a queue are implemented. One end of the stack is fixed (the bottom), while the other end rises and falls as data are pushed and popped. A single pointer is needed to point to the top of the stack at any given time. On the other hand, both ends of a queue move to higher addresses as data are added at the back and removed from the front. So, two pointers are needed to keep track of the two ends of the queue.

Another difference between a stack and a queue is that, without further control, a queue would continuously move through the memory of a computer in the direction of higher addresses. One way to limit the queue to a fixed region in memory is to use a circular buffer. Let us assume that memory addresses from BEGINNING to END are assigned to the queue. The first entry in the queue is entered into location BEGINNING, and successive entries are appended to the queue by entering them at successively higher addresses. By the time the back of the queue reaches END, space will have been created at the beginning if some items have been removed from the queue. Hence, the back pointer is reset to the value BEGINNING and the process continues. As in the case of a stack, care must be taken to detect when the region assigned to the data structure is either completely full or completely empty.

2.5 Subroutines

In a given program, it is often necessary to perform a particular subtask many times on different data-values. Such a subtask is usually called a subroutine. For example, a subroutine may evaluate the sine function or sort a list of values into increasing or decreasing order.

It is possible to include the block of instructions that constitute a subroutine at every place where it is needed in the program. However, to save space, only one copy of the instructions that constitute the subroutine is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location. When a program branches to a subroutine we say that it is calling the subroutine. The instruction that performs this branch operation is named a Call instruction.

After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine. The subroutine is said to return to the program that called it by executing a Return instruction.

The way in which a computer makes it possible to call and return from subroutines is referred to as its subroutine linkage method. The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called the link register. When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register.

The Call instruction is just a special branch instruction that performs the following operations

- Store the contents of the PC in the link register
- Branch to the target address specified by the instruction.

The Return instruction is a special branch instruction that performs the operation.

- Branch to the address contained in the link register.

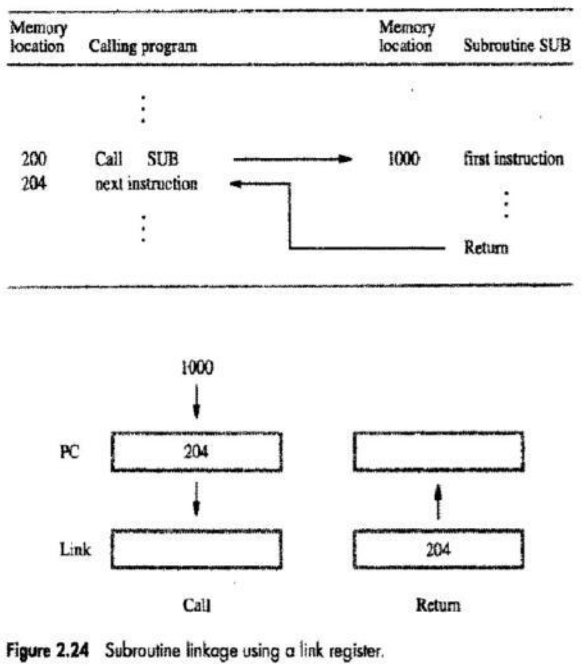


Figure 2.24 Subroutine linkage using a link register.

SUBROUTINE NESTING AND THE PROCESSOR STACK:-

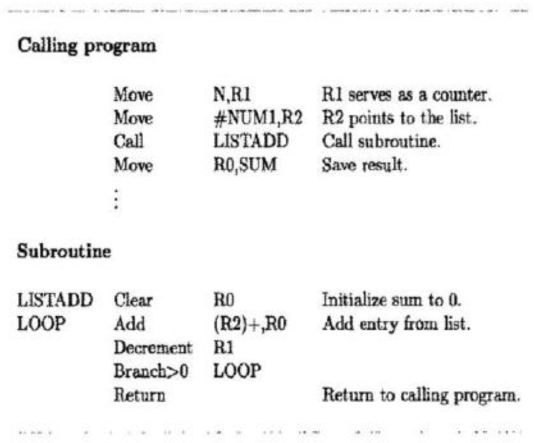
A common programming practice, called subroutine nesting, is to have one subroutine call another. In this case, the return address of the second call is also stored in the link register, destroying its previous contents. Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost.

Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it. The return address needed for this first return is the last one generated in the nested call sequence. That is, return addresses are generated and used in a last-in-first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto a stack. A particular register is designated as the stack pointer, SP, to be used in this operation. The stack pointer points to a stack called the processor stack. The Call instruction pushes the contents of the PC onto the processor stack and loads the subroutine address into the PC. The Return instruction pops the return address from the processor stack into the PC.

PARAMETER PASSING:-

When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, in this case, the results of the computation. This exchange of information between a calling program and a subroutine is referred to as parameter passing. Parameter passing may be accomplished in several ways. The parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the processor stack used for saving the return address.

The purpose of the subroutines is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the address of the first number in the list. This technique is called passing by reference. The second parameter is passed by value, that is, the actual number of entries, n, is passed to the subroutine.



Calling program

```
         Move        N,R1        R1 serves as a counter.
         Move        #NUM1,R2    R2 points to the list.
         Call        LISTADD     Call subroutine.
         Move        R0,SUM      Save result.
         :
```

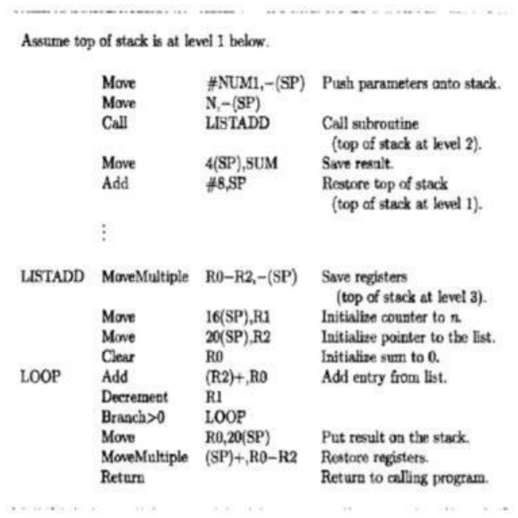
Subroutine

```
LISTADD  Clear       R0          Initialize sum to 0.
LOOP     Add         (R2)+,R0    Add entry from list.
         Decrement   R1
         Branch>0    LOOP
         Return                  Return to calling program.
```

Figure 2.25 Program of Figure 2.16 written as a subroutine; parameters passed through registers.

THE STACK FRAME:-

Now, observe how space is used in the stack in the example. During execution of the subroutine, six locations at the top of the stack contain entries that are needed by the subroutine. These locations constitute a private workspace for the subroutine, created at the time the subroutine is entered and freed up when the subroutine returns control to the calling program. Such space is called a stack frame.

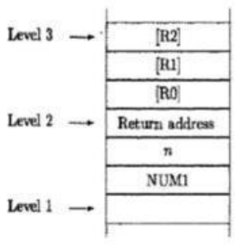


```
Assume top of stack is at level 1 below.

            Move           #NUM1,-(SP)   Push parameters onto stack.
            Move           N,-(SP)
            Call           LISTADD       Call subroutine
                                           (top of stack at level 2).
            Move           4(SP),SUM     Save result.
            Add            #8,SP         Restore top of stack
                                           (top of stack at level 1).
            ⋮

LISTADD     MoveMultiple   R0-R2,-(SP)   Save registers
                                           (top of stack at level 3).
            Move           16(SP),R1     Initialize counter to n.
            Move           20(SP),R2     Initialize pointer to the list.
            Clear          R0            Initialize sum to 0.
LOOP        Add            (R2)+,R0      Add entry from list.
            Decrement      R1
            Branch>0       LOOP
            Move           R0,20(SP)     Put result on the stack.
            MoveMultiple   (SP)+,R0-R2   Restore registers.
            Return                       Return to calling program.
```

(a) Calling program and subroutine



(b) Top of stack at various times

Figure 2.26 Program of Figure 2.16 written as a subroutine; parameters passed on the stack.

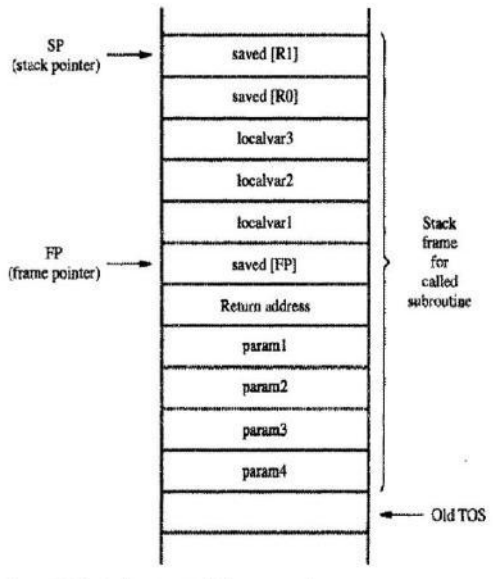


Figure 2.27 A subroutine stack frame example.

fig 2.27 shows an example of a commonly used layout for information in a stack frame. In addition to the stack pointer SP, it is useful to have another pointer register, called the frame pointer (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine. These local variables are only used within the subroutine, so it is appropriate to allocate space for them in the stack frame associated with the subroutine. We assume that four parameters are passed to the subroutine, three local variables are used within the subroutine, and registers R0 and R1 need to be saved because they will also be used within the subroutine.

The pointers SP and FP are manipulated as the stack frame is built, used, and dismantled for a particular of the subroutine. We begin by assuming that SP point to the old top-of-stack (TOS) element in fig b. Before the subroutine is called, the calling program pushes the four parameters onto the stack. The call instruction is then executed, resulting in the return address being pushed onto the stack. Now, SP points to this return address, and the first instruction of the subroutine is about to be executed. This is the point at which the frame pointer FP is set to contain the proper memory address. Since FP is usually a general-purpose register, it may contain information of use to the Calling program. Therefore, its contents are saved by pushing them onto the stack. Since the SP now points to this position, its contents are copied into FP.

Thus, the first two instructions executed in the subroutine are

```
Move FP, -(SP)
Move SP, FP
```

After these instructions are executed, both SP and FP point to the saved FP contents.

```
Subtract #12, SP
```

Finally, the contents of processor registers R0 and R1 are saved by pushing them onto the stack. At this point, the stack frame has been set up as shown in the fig.

The subroutine now executes its task. When the task is completed, the subroutine pops the saved values of R1 and R0 back into those registers, removes the local variables from the stack frame by executing the instruction.

```
Add #12, SP
```

and pops the saved old value of FP back into FP. At this point, SP points to the return address, so the Return instruction can be executed, transferring control back to the calling program.

ADDITIONAL INSTRUCTIONS

Logic instructions

Logic operations such as AND, OR, and NOT, applied to individual bits, are the basic building blocks of digital circuits, as described. It is also useful to be able to perform logic operations is software, which is done using instructions that apply these operations to all bits of a word or byte independently and in parallel. For example, the instruction

```
Not dst
```

SHIFT AND ROTATE INSTRUCTIONS:-

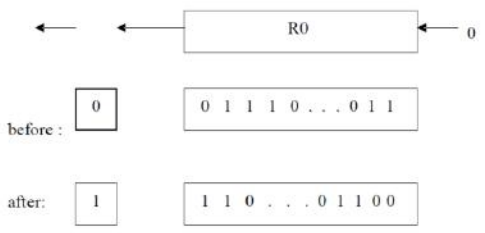
There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions. The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information. For general operands, we use a logical shift. For a number, we use an arithmetic shift, which preserves the sign of the number.

Logical shifts:-

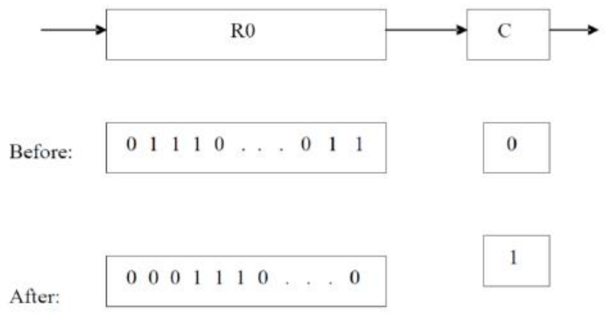
Two logical shift instructions are needed, one for shifting left (LShiftL) and another for shifting right (LShiftR). These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction. The general form of a logical left shift instruction is

```
LShiftL count, dst
```

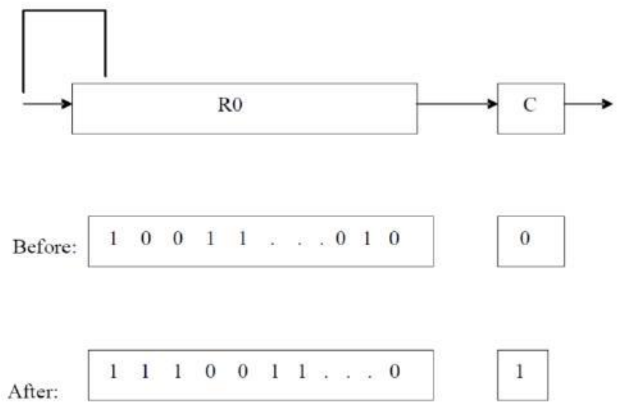
(a) Logical shift left LShiftL #2, R0



(b) Logical shift right LShiftR #2, R0



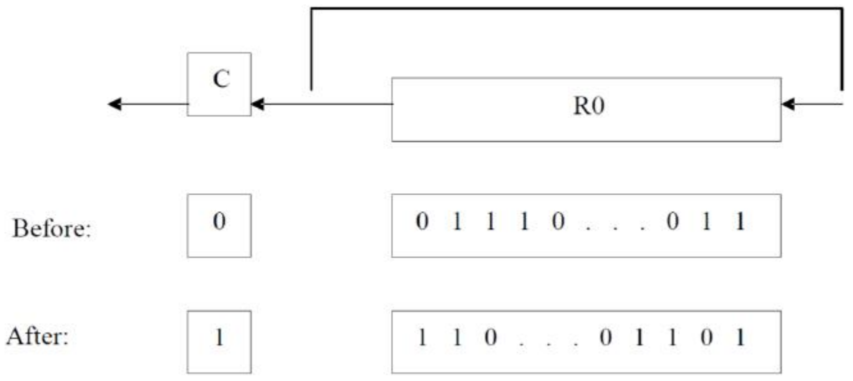
(c) Arithmetic shift right AShiftR #2, R0



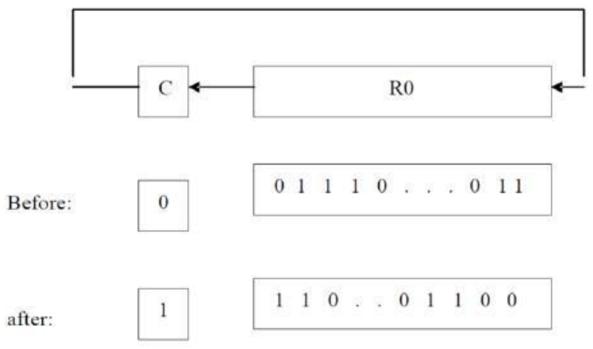
Rotate Operations:-

In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry flag C. To preserve all bits, a set of rotate instructions can be used. They move the bits that are shifted out of one end of the operand back into the other end. Two versions of both the left and right rotate instructions are usually provided. In one version, the bits of the operand are simply rotated. In the other version, the rotation includes the C flag.

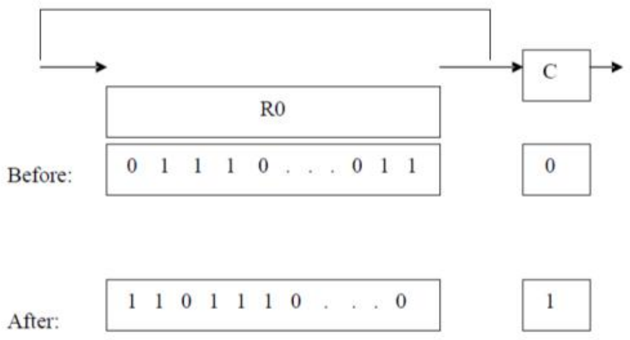
(a) Rotate left without carry RotateL #2, R0



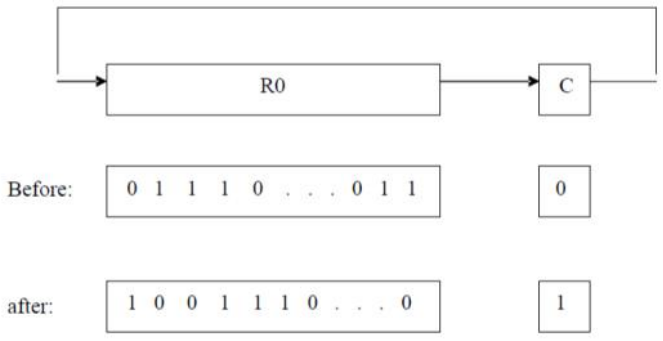
(b) Rotate left with carry RotateLC #2, R0



(c) Rotate right without carry RotateR #2, R0



(d) Rotate right with carry RotateRC #2, R0



2.10.3 MULTIPLICATION AND DIVISION

‘Two signed integers can be multiplied or divided by machine instructions with the same format. The instruction

Multiply Ri,Rj

performs the operation

$$Rj \leftarrow [Ri] \times [Rj]$$

The product of two n-bit numbers can be as large as 2n bits. A number of instruction sets have a Multiply instruction that computes the lew-order n bits of the product and places i in register Rj, as indicated. To accommodate the general 2n-bit product case, some processors produce the product in two registers, usually adjacent registers Rj and R(j + 1), with the high-order half being placed in register R(j + 1).

Some instruction sets provide a signed integer Divide instruction

Divide Ri,Rj

which performs the operation

$$Rj \leftarrow [Rj]/[Ri]$$

placing the quotient in Rj. The remainder may be placed in R(j + 1), or it may be Lost.

-- END --

ANNAMACHARYA INSTITUTE OF TECHNOLOGY AND SCIENCES::TIRUPATI
II B.Tech II Semester

(19APC0506) Computer Organization

L	T	P	C
3	0	0	3

Course Objectives:

- To learn the fundamentals of computer organization and its relevance to classical and modern problems of computer design
- To make the students understand the structure and behavior of various functional modules of a computer.
- To understand the techniques that computers use to communicate with I/O devices
- To study the concepts of pipelining and the way it can speed up processing.
- To understand the basic characteristics of multiprocessors

Unit I:
Basic Structure of Computer: Computer Types, Functional Units, Basic operational Concepts, Bus Structure, Software, Performance, Multiprocessors and Multicomputer.
Machine Instructions and Programs: Numbers, Arithmetic Operations and Programs, Instructions and Instruction Sequencing, Addressing Modes, Basic Input/output Operations, Stacks and Queues, Subroutines, Additional Instructions.
Unit II:
Arithmetic: Addition and Subtraction of Signed Numbers, Design and Fast Adders, Multiplication of Positive Numbers, Signed-operand Multiplication, Fast Multiplication, Integer Division, Floating-Point Numbers and Operations.
Basic Processing Unit: Fundamental Concepts, Execution of a Complete Instruction, Multiple-Bus Organization, Hardwired Control, Multiprogrammed Control.
Unit III:
The Memory System: Basic Concepts, Semiconductor RAM Memories, Read-Only Memories, Speed, Size and Cost, Cache Memories, Performance Considerations, Virtual Memories, Memory Management Requirements, Secondary Storage.
Unit IV:
Input/output Organization: Accessing I/O Devices, Interrupts, Processor Examples, Direct Memory Access, Buses, Interface Circuits, Standard I/O Interfaces.
Unit V:
Pipelining: Basic Concepts, Data Hazards, Instruction Hazards, Influence on Instruction Sets
Large Computer Systems: Forms of Parallel Processing, Array Processors, The Structure of General-Purpose, Interconnection Networks.

Textbook:
1. "Computer Organization", Carl Hamacher, Zvonko Vranesic, Safwat Zaky, McGraw Hill Education, 5th Edition, 2013.

Reference Textbooks:
1. Computer System Architecture, M.Morris Mano, Pearson Education, 3rd Edition.
2. Computer Organization and Architecture, Themes and Variations, Alan Clements, CENGAGE Learning.
3. Computer Organization and Architecture, Smruti Ranjan Sarangi, McGraw Hill Education.
4. Computer Architecture and Organization, John P.Hayes, McGraw Hill Education.

Course Outcomes:

- Ability to use memory and I/O devices effectively
- Able to explore the hardware requirements for cache memory and virtual memory
- Ability to design algorithms to exploit pipelining and multiprocessors

UNIT-II

Arithmetic: Addition and Subtraction of Signed Numbers, Design and Fast Adders, Multiplication of Positive Numbers, Signed-operand Multiplication, Fast Multiplication, Integer Division, Floating-Point Numbers and Operations.

Basic Processing Unit: Fundamental Concepts, Execution of a Complete Instruction, Multiple-Bus Organization, Hardwired Control, Multiprogrammed Control.

Arithmetic

2.1 ADDITION AND SUBTRACTION OF SIGNED NUMBERS

Figure 6.1 shows the logic truth table for the sum and carry-out functions for adding equally weighted bits x; and y, in two numbers X and Y, The figure also shows logic expressions for these functions, along with an example of addition of the 4-bit unsigned numbers 7 and 6. Note that each stage of the addition process must accommodate a carry-in bit. We use c_i to represent the carry-in to the i^{th} stage, which is the same as the carry-out from the (i - 1) st stage.

The logic expression for s_i in Figure 6.1 can be implemented with a 3-input XOR gate, used in Figure 6.2a as part of the logic required for a single stage of binary addition. The carry-out function, c_{i+1}, is implemented with a two-level AND-OR logic circuit. A convenient symbol for the complete circuit for a single stage of addition, called a full adder (FA), is also shown in the figure.

A cascaded connection of such n full adder blocks, as shown in Figure 6.2C, forms a parallel adder & can be used to add two n-bit numbers. Since the carries must propagate, or ripple, through this cascade, the configuration is called an n-bit ripple-carry adder.

The carry-in, C_o, into the *least-significant-bit (LSB)* position [1st stage] provides a convenient means of adding 1 to a number. Take for instance; forming the 2's- complement of a number involves adding 1 to the 1’s-complement of the number. The carry signals are also useful for interconnecting k adders to form an adder capable of handling input numbers that are **kn** bits long, as shown in Figure 6.2c.

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \bar{x}_i\bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Example:

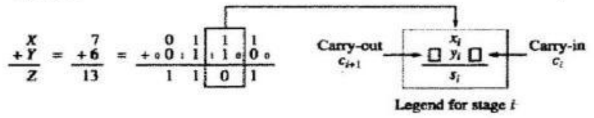


Figure 6.1 Logic specification for a stage of binary addition.

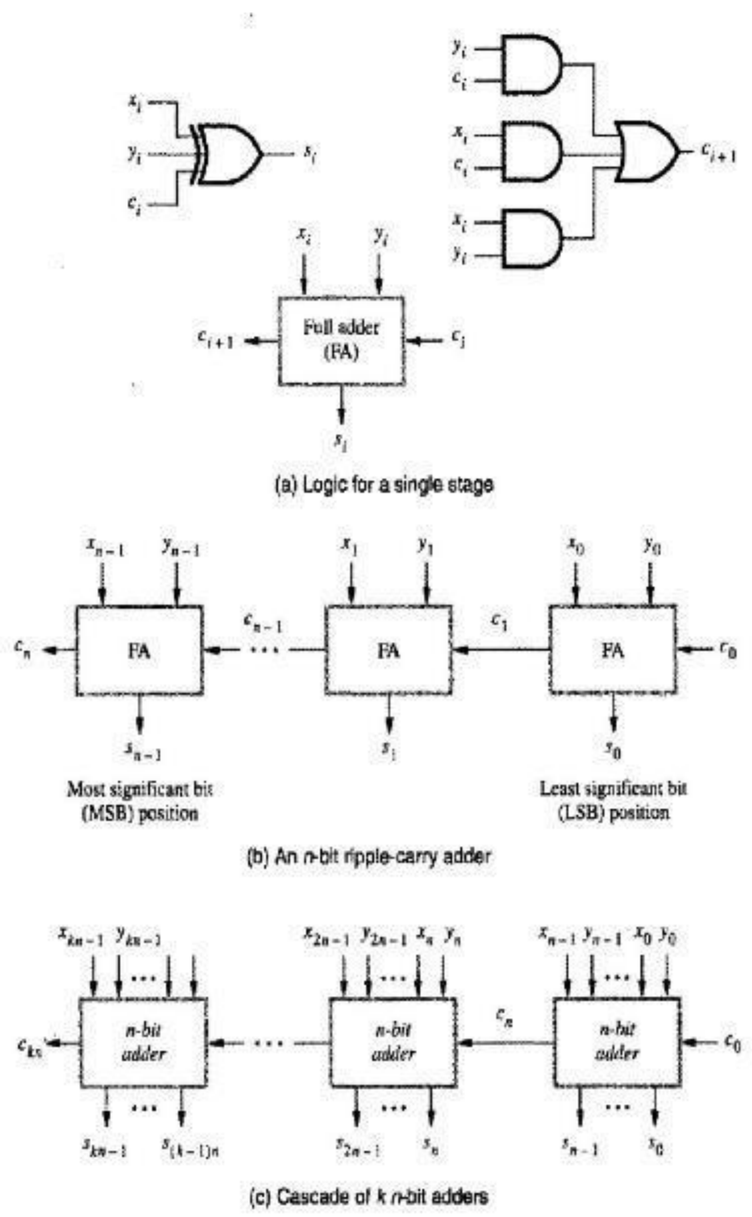

Full adder (FA)
(a) Logic for a single stage
FA
Most significant bit (MSB) position
Least significant bit (LSB) position
(b) An n-bit ripple-carry adder
n-bit adder
(c) Cascade of k n-bit adders

Figure 6.2 Logic for addition of binary vectors.

ADDITION/SUBTRACTION Logic UNIT:

The n-bit adder in Figure 6.28 can be used to add 2's-complement numbers X and Y, where the X_{n-1} and y_{n-1} bits are the sign bits. Overflow can only occur when the signs of the two operands are the same. In this case, overflow obviously occurs if the sign of the result is different, therefore, a circuit to detect overflow can be added to the n-bit adder by implementing the logic expression

$$s_i = \bar{x}_i\bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Overflow can also occur when the carry bits c, and c_{n-1} are different. Therefore, a simpler alternative circuit for detecting overflow can be obtained by implementing the expression $\mathbf{c_n \oplus c_{n-1}}$ with an XOR gate.

In order to perform the subtraction operation X - Y on 2's-complement numbers X and Y, we form the 2's-compiement of Y and add it to X. The logic circuit network shown in Figure 6.3 can be used to perform either addition or subtraction based on the value applied to the Add/Sub input control line. This line is set to 0 for addition, applying the Y vector unchanged to one of the adder inputs along with a carry-in signal, c_o, of 0, When the Add/Sub control line is set to 1, the Y vector is 1's-complemented (that is, bit complemented) by the XOR gates and co is set to 1 to complete the 2's-complementation of Y. An XOR gate can be added to Figure 6.3 to detect the overflow condition $\mathbf{c_n \oplus c_{n-1}}$.

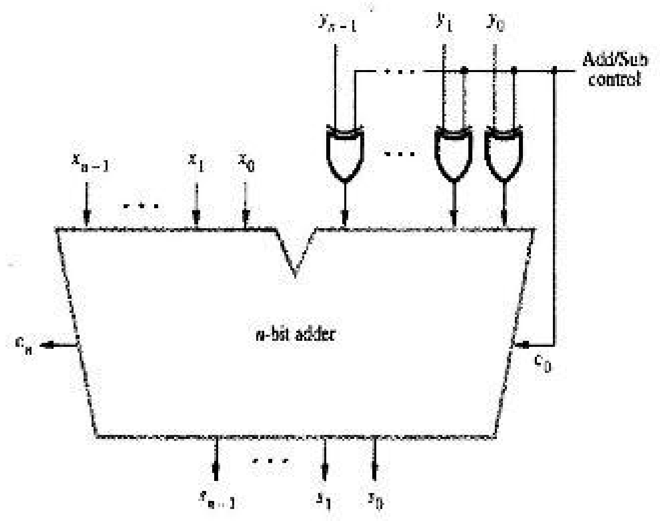


Figure 6.3 Binary addition-subtraction logic network

6.2 DESIGN OF FAST ADDERS:

In an n-bit parallel adder (ripple-carry adder), there is too much delay in developing the outputs, s_o through s_{n-1} and c_n. On many occasions this delay is not acceptable; in comparison with the speed of other processor components and speed of the data transfer between registers and cache memories. The delay through a network depends on the integrated circuit technology used in fabricating the network and on the number of gates in the paths from inputs to outputs (propagation delay). The delay through any combinational logic network constructed from gates in a particular technology is determined by adding up the number of logic-gate delays along the longest signal propagation path through the network. In the case of the n-bit ripple-carry adder, the longest path is from inputs x_0, y_0, and c_0 at the least-significant-bit (LSB) position to outputs c_n and s_{n-1} at the most-significant-bit (MSB) position.

Using the logic implementation indicated in Figure 6.2a, c_{n-1} is available in 2(n—1) gate delays, and s_{n-1} is one XOR gate delay later. The final carry-out, c_n is available after 2n gate delays. Therefore, if a ripple-carry adder is used to implement the addition/subtraction unit shown in Figure-6.3, all sum bits are available in 2n gate delays, including the delay through the XOR gates on the Y input. Using the implementation **$c_n \oplus c_{n-1}$** for overflow, this indicator is available after 2n+2 gate delays. In summary, in a parallel adder an nth stage adder cannot complete the addition process before all its previous stages have completed the addition even with input bits ready. This is because; the carry bit from previous stage has to be made available for addition of the present stage.

In practice, a number of design techniques have been used to implement high- speed adders. In order to reduce this delay in adders, an augmented logic gate network structure may be used. One such method is to use circuit designs for fast propagation of carry signals (carry prediction).

Carry-Look ahead Addition:

As it is clear from the previous discussion that a parallel adder is considerably slow & a fast adder circuit must speed up the generation of the carry signals, it is necessary to make the carry input to each stage readily available along with the input bits.

This can be achieved either by propagating the previous carry or by generating a carry depending on the input bits & previous carry. The logic expressions for s_i (sum) and c_{i+1} (carry-out) of stage i are

$$s_i = x_i \oplus y_i \oplus c_i$$

and

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Factoring the second equation into

$$c_{i+1} = x_i y_i + (x_i + y_i)c_i$$

we can write

$$c_{i+1} = G_i + P_i c_i$$

where

$$G_i = x_i y_i \quad \text{and} \quad P_i = x_i + y_i$$

The expressions G_i and P_i are called *generate* and *propagate* functions for stage i. If the generate function for stage i is equal to 1, then $c_{i+1} = 1$, independent of the input carry, c_i. This occurs when both x_i and y_i are 1. The propagate function means that an input carry will produce an output carry when either x_i or y_i or both equal to 1. Now, using G_i & P_i functions we can decide carry for i^{th} stage even before its previous stages have completed their addition operations. All G_i and P_i functions can be formed independently and in parallel in only one gate delay after the X_i and Y_i inputs are applied to an n-bit adder. Each bit stage contains an AND gate to form G_i, an OR gate to form P_i and a three-input XOR gate to form s_i. However, a much simpler circuit can be derived by considering the propagate function as $P_i = x_i \oplus y_i$ which differs from $P_i = x_i + y_i$ only when $x_i = y_i = 1$ where $G_i = 1$ (so it does not matter whether Pi is 0 or 1). Then, the basic diagram in Figure-6.4a can be used in each bit stage to predict carry ahead of any stage completing its addition.

Consider the c_{i+1} expression

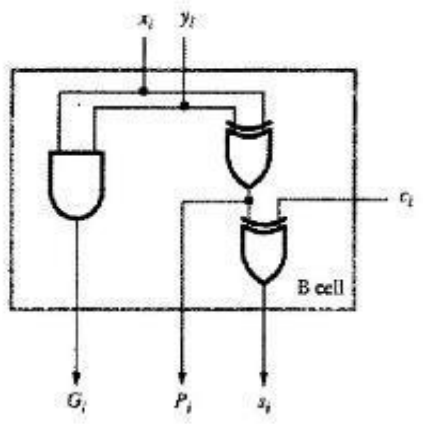
$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1}$$

This is because, $C_i = (G_{i-1} + P_{i-1}C_{i-1})$.

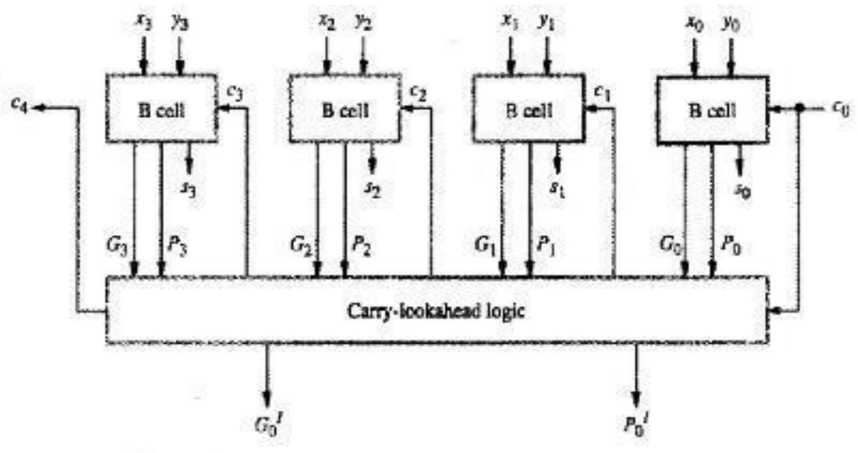
Further, $C_{i-1} = (G_{i-2} + P_{i-2}C_{i-2})$ and so on. Expanding in this fashion, the final carry expression can be written as below;

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} \cdots P_1 G_0 + P_i P_{i-1} \cdots P_0 c_0$$

Thus, all carries can be obtained in three gate delays after the input signals X_i, Y_i and C_{in} are applied at the inputs. This is because only one gate delay is needed to develop all P_i and G_i signals, followed by two gate delays in the AND-OR circuit (SOP expression) for c_i after a further XOR gate delay, all sum bits are available. Therefore, independent of n, the number of stages, the n-bit addition process requires only four gate delays.



(a) Bit-stage cell



(b) 4-bit adder

Figure 6.4 4-bit carry-lookahead adder.

Now, consider the design of a 4-bit parallel adder. The carries can be implemented as

$$c_1 = G_0 + P_0 c_0$$

$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$

$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

The complete 4-bit adder is shown in Figure 6.4b where the B cell indicates G_i, P_i & S_i generator. The carries are implemented in the block labeled carry look-ahead logic. An adder implemented in this form is called a carry look ahead adder. Delay through the adder is 3 gate delays for all carry bits and 4 gate delays for all sum bits. In comparison, note that a 4-bit ripple-carry adder requires 7 gate delays for S3(2n-1) and 8 gate delays(2n) for c4.

If we try to extend the carry lookahead adder of Figure 5b for longer operands, we run into a problem of gate fan-in constraints. From the final expression for C_{i+1} & the carry expressions for a 4 bit adder, we see that the last AND gate and the OR gate require a fan-in of i + 2 in generating cn-1. For c4 (i = 3)in the 4-bit adder, a fan-in of 5 is required. This puts the limit on the practical implementation. So the adder design shown in Figure 4b cannot be directly extended to longer operand sizes. However, if we cascade a number of 4-bit adders, it is possible to build longer adders without the practical problems of fan- in. An example of a 16 bit carry look ahead adder is as shown in figure. Eight 4-bit carry look-ahead adders can be connected as in Figure-6.2 to form a 32-bit adder.

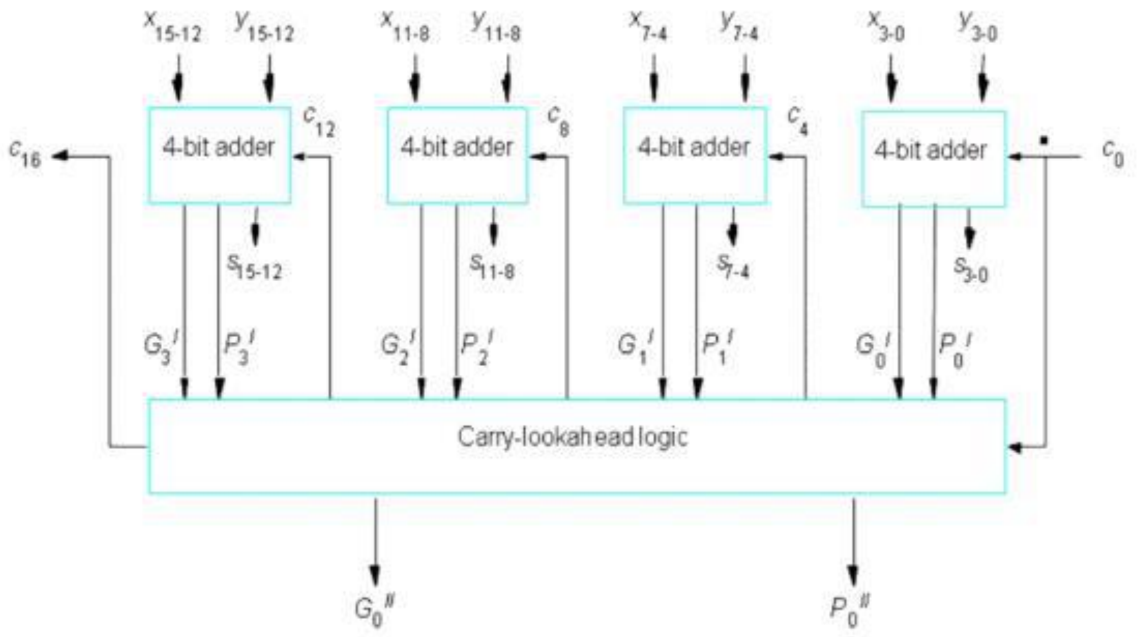


FIG: 16 bit carry-lookahead adder

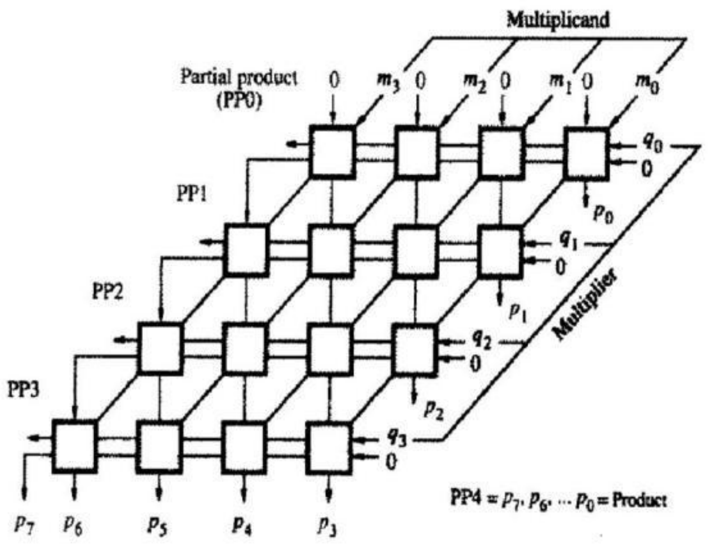
6.3 MULTIPLICATION OF POSITIVE NUMBERS

Consider the multiplication of two integers as in Figure-6a in binary number system. This algorithm applies to unsigned numbers and to positive signed numbers. The product of two n-digit numbers can be accommodated in 2n digits, so the product of the two 4-bit numbers in this example fits into 8 bits. In the binary system, multiplication by the multiplier bit '1' means the multiplicand is entered in the appropriate position to be added to the partial product. If the multiplier bit is '0', then 0s are entered, as indicated in the third row of the shown example.

Binary multiplication of positive operands can be implemented in a combinational (speed up) two-dimensional logic array, as shown in Figure 6.6. Here, M- indicates multiplicand, Q- indicates multiplier & P- indicates partial product. The basic component in each cell is a full adder FA. The AND gate in each cell determines whether a multiplicand bit mj, is added to the incoming partial-product bit, based on the value of the multiplier bit, qi. For i in the range of 0 to 3, if qi = 1, add the multiplicand (appropriately shifted) to the incoming partial product, PPi, to generate the outgoing partial product, PP(i+ 1) & if qi = 0, PPi is passed vertically downward unchanged. The initial partial product PPO is all 0s. PP4 is the desired product. The multiplicand is shifted left one position per row by the diagonal signal path. Since the multiplicand is shifted and added to the partial product depending on the multiplier bit, the method is referred as SHIFT & ADD method. The multiplier array & the components of each bit cell are indicated in the diagram, while the flow diagram shown explains the multiplication procedure.

```
        1 1 0 1      (13) Multiplicand M
      × 1 0 1 1      (11) Multiplier Q
      ---------
        1 1 0 1
      1 1 0 1
    0 0 0 0
  1 1 0 1
---------------
1 0 0 0 1 1 1 1      (143) Product P
```

(a) Manual multiplication algorithm



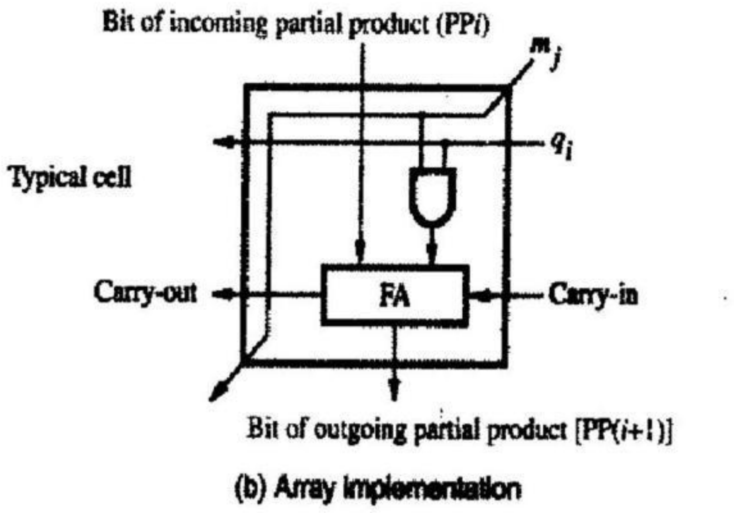
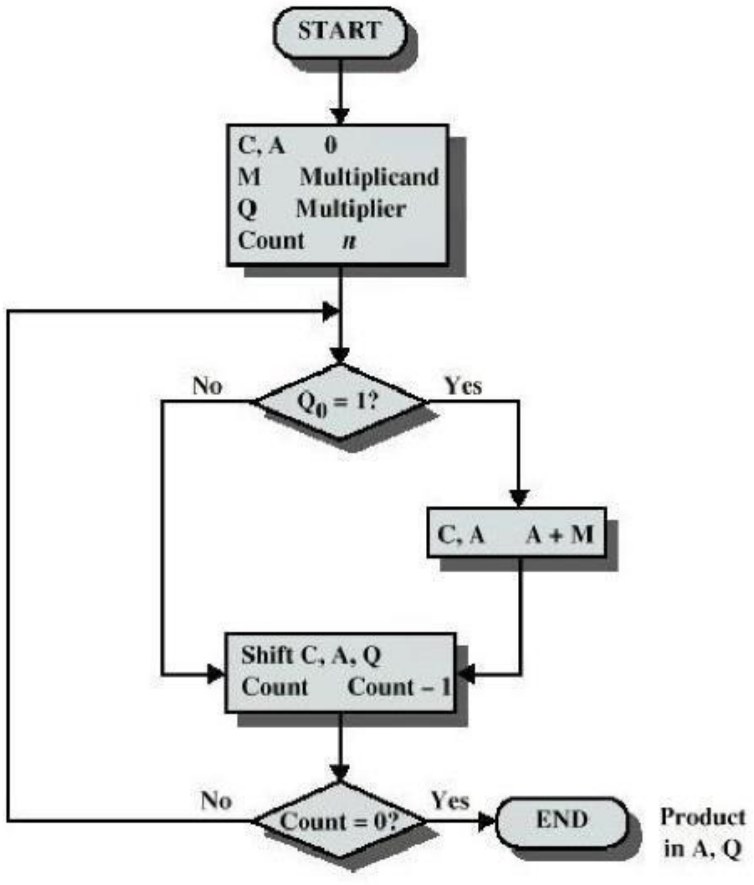

Bit of incoming partial product (PPi)
m_j
q_i
Typical cell
Carry-out
FA
Carry-in
Bit of outgoing partial product [PP(i+1)]
(b) Array implementation

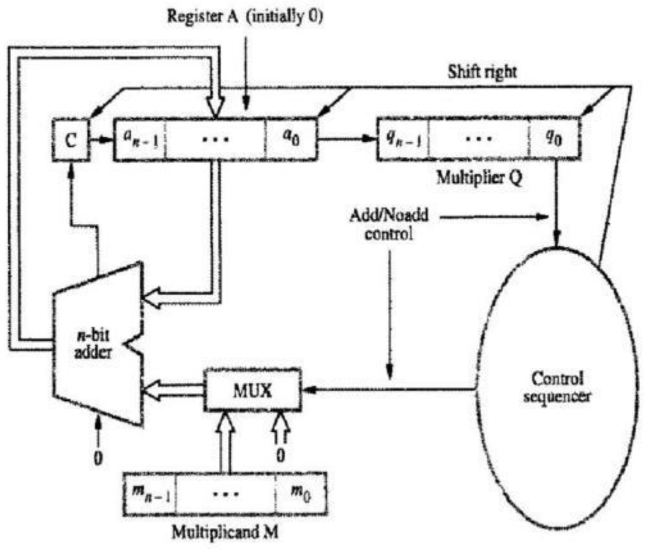
Fig 6.6: Array multiplication of positive binary operands

The following SHIFT & ADD method flow chart depicts the multiplication logic for unsigned numbers.



Despite the use of a combinational network, there is a considerable amount of delay associated with the arrangement shown. Although the preceding combinational multiplier is easy to understand, it uses many gates for multiplying numbers of practical size, such as 32- or 64-bit numbers. The worst case signal propagation delay path is from the upper right corner of the array to the high-order product bit output at the bottom left corner of the array. The path includes the two cells at the right end of each row, followed by all the cells in the bottom row. Assuming that there are two gate delays from the inputs to the outputs of a full adder block, the path has a total of 6(n - 1) - 1 gate delays, including the initial AND gate delay in all cells, for the n x n array. In the delay expression, (n-1) because, only the AND gates are actually needed in the first row of the array because the incoming (initial) partial product PPO is zero

Multiplication can also be performed using a mixture of combinational array techniques (similar to those shown in Figure 7) and sequential techniques requiring less combinational logic. Multiplication is usually provided as an instruction in the machine instruction set of a processor. High-performance processor (DS processors) chips use an appreciable area of the chip to perform arithmetic functions on both integer and floating-point operands. Sacrificing an area on-chip for these arithmetic circuits increases the speed of processing. Generally, processors built for real time applications have an on-chip multiplier.



(a) Register configuration

Another simplest way to perform multiplication is to use the adder circuitry in the ALU for a number of sequential steps. The block diagram in Figure 8a shows the hardware arrangement for sequential multiplication. This circuit performs multiplication by using single n-bit adder n times to implement the spatial addition performed by the n rows of ripple-carry adders. Registers A and Q combined to hold PPi while multiplier bit qi generates the signal Add/No-add. This signal controls the addition of the multiplicand M to PPi to generate PP(i + 1). The product is computed in n cycles. The partial product grows in length by one bit per cycle from the initial vector, PPO, of n 0s in register A. The carry-out from the adder is stored in flip-flop C. To begin with, the multiplier is loaded into register Q, the multiplicand into register M and registers C and A are cleared to 0. At the end of each cycle C, A, and Q are shifted right one bit positions to allow for growth of the partial product as the multiplier is shifted out of register Q. Because of this shifting, multiplier bit qi, appears at the LSB position of Q to generate the Add/No-add signal at the correct time, starting with qo during the first cycle, q1 during the second cycle, and so on.

After they are used, the multiplier bits are discarded by the right-shift operation. Note that the carry-out from the adder is the leftmost bit of PP(i + 1), and it must be held in the C flip-flop to be shifted right with the contents of A and Q. After n cycles, the high-order half-of- the product is held in register A and the low-order half is in register Q. The multiplication example used above is shown in Figure 8b as it would be performed by this hardware arrangement.

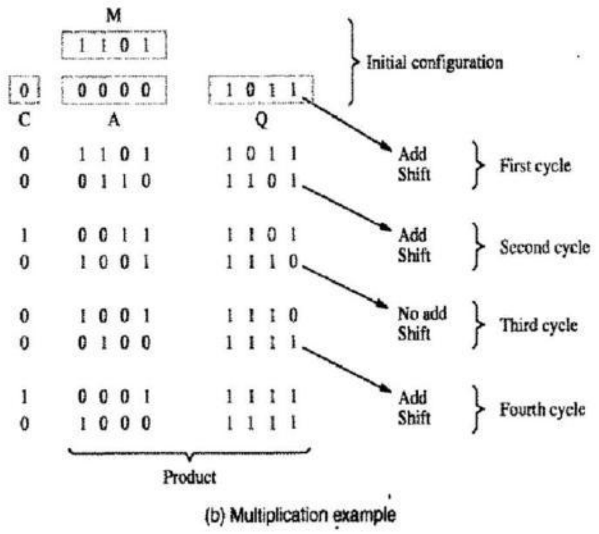


Fig: Sequential circuit binary multiplier.

Using this sequential hardware structure, it is clear that a **multiply** instruction takes much more time to execute than an **Add** instruction. This is because of the sequential circuits associated in a multiplier arrangement. Several techniques have been used to speed up multiplication; bit pair recoding, carry save addition, repeated addition, etc.

6.4 SIGNED-OPERAND MULTIPLIATION:

Multiplication of 2's-complement signed operands, generating a double-length product is still achieved by accumulating partial products by adding versions of the multiplicand as decided by the multiplier bits. First, consider the case of a positive multiplier and a negative multiplicand. When we add a negative multiplicand to a partial product, we must extend the sign-bit value of the multiplicand to the left as far as the product will extend. In Figure 6.8, for example, the 5-bit signed operand, - 13, is the multiplicand, and +11, is the 5 bit multiplier & the expected product -143 is 10-bit wide. The sign extension of the multiplicand is shown in red color. Thus, the hardware discussed earlier can be used for negative multiplicands if it provides for sign extension of the partial products.

For a negative multiplier, a straightforward solution is to form the 2's- complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier. This is possible because complementation of both operands does not change the value or the sign of the product. In order to take care of both negative and positive multipliers, BOOTH algorithm can be used.

```
                                        1  0  0  1  1   (-13)
                                    x   0  1  0  1  1   (+11)
                                    -----------------
                    1  1  1  1  1   1  0  0  1  1

                    1  1  1  1  1   0  0  1  1

Sign extension is   0  0  0  0  0   0  0  0
shown in blue
                    1  1  1  0  0   1  1

                    0  0  0  0  0   0
                    ---------------------------------
                    1  1  0  1  1   1  0  0  0  1   (-143)
```

Figure 6.8 Sign extension of negative multiplicand.

Booth Algorithm

The Booth algorithm generates a 2n-bit product and both positive and negative 2's-complement n-bit operands are uniformly treated. To understand this algorithm, consider a multiplication operation in which the multiplier is positive and has a single block of 1s, for example, 0011110(+30). To derive the product, as in the normal standard procedure, we could add four appropriately shifted versions of the multiplicand,. However, using the Booth algorithm, we can reduce the number of required operations by regarding this multiplier as the difference between numbers 32 & 2 as shown below;

```
  0100000  (32)
 -0000010   (2)
 ---------------
  0011110  (30)
```

This suggests that the product can be generated by adding 25 times the multiplicand to the 2's-complement of 21 times the multiplicand. For convenience, we can describe the sequence of required operations by recoding the preceding multiplier as 0 +1000 - 10. In general, in the Booth scheme, -1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left.

Figure 6.9 illustrates the normal and the Booth algorithms for the said example. The Booth algorithm clearly extends to any number of blocks of 1s in a multiplier, including the situation in which a single 1 is considered a block. See Figure 6.10 for another example of recoding a multiplier. The case when the least significant bit of the multiplier is 1 is handled by assuming that an implied 0 lies to its right. The Booth algorithm can also be used directly for negative multipliers, as shown in Figure 6.11. To verify the correctness of the Booth algorithm for negative multipliers, we use the following property of negative-number representations in the 2's-complement.

```
                0 1 0 1 1 0 1
                0 0+1+1+1+1 0
                -------------
                0 0 0 0 0 0 0
              0 1 0 1 1 0 1
            0 1 0 1 1 0 1
          0 1 0 1 1 0 1
        0 1 0 1 1 0 1
      0 0 0 0 0 0 0
    0 0 0 0 0 0 0
---------------------------
0 0 0 1 0 1 0 1 0 0 0 1 1 0

                0 1 0 1 1 0 1
                0+1 0 0 0-1 0
                -------------
0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0 1 0 0 1 1   <---- 2's complement of
0 0 0 0 0 0 0 0 0 0 0 0           the multiplicand
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 1 0 1
0 0 0 0 0 0 0 0
---------------------------
0 0 0 1 0 1 0 1 0 0 0 1 1 0
```

Figure 6.9 Normal and Booth multiplication schemes.

```
0   0   1   0   1   1   0   0   1   1   1   0   1   0   1   1   0   0
                                  ⇓
0  +1  -1  +1   0  -1   0  +1   0   0  -1  +1  -1  +1   0  -1   0   0
```

Figure 6.10 Booth recoding of a multiplier.

```
  0 1 1 0 1  (+13)                0 1 1 0 1
× 1 1 0 1 0  (-6)     ⟹          0 -1 +1 -1 0
-----------                     -------------
                      0 0 0 0 0 0 0 0 0 0
                      1 1 1 1 1 0 0 1 1
                      0 0 0 0 1 1 0 1
                      1 1 1 0 0 1 1
                      0 0 0 0 0 0
                      -------------------
                      1 1 1 0 1 1 0 0 1 0  (-78)
```

Figure 6.11 Booth multiplication with a negative multiplier.

To demonstrate the correctness of the Booth algorithm for negative multipliers, we use the following property of negative-number representations in the 2’s-complement system: Let the leftmost 0 of a negative number, X, be at bit position k, that is,

$$X = 11\ldots10x_{k-1}\ldots x_0$$

Then the value of X is given by

$$V(X) = -2^{k+1} + x_{k-1} \times 2^{k-1} + \cdots + x_0 \times 2^0$$

The correctness of this expression for V(X) is shown by observing that if X is formed as the sum of two numbers

$$\begin{array}{rl} & 11\ldots100000\ldots0 \\ + & 00\ldots00x_{k-1}\ldots x_0 \\ \hline X = & 11\ldots10x_{k-1}\ldots x_0 \end{array}$$

then the top number is the 2's-complement representation of -2^{k+1}. The recoded multiplier now consists of the part corresponding to the second number, with —1 added inposition k + 1. For example, the multiplier 110110 is recoded as 0 -1 +10 -10.

The Booth technique for recoding multipliers is summarized in Figure 6.12. The transformation 011...110 => +100..,0—10 is called *skipping over 1s*. This term is derived from the case in which the multiplier has its 1s grouped into a few contiguous blocks, Only a few versions of the shifted multiplicand (the summands) must be added to generate the product, thus speeding up the multiplication operation. However, in the worst case — that of alternating 1s and 0s in the multiplier - each bit of the multiplier selects a summand, In fact, this results in more summands than if the Booth algorithm were not used. A 16-bit, worst-case multiplier, an ordinary multiplier, and a good multiplier are shown in Figure 6.13.

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Figure 6.12 Booth multiplier recoding table.

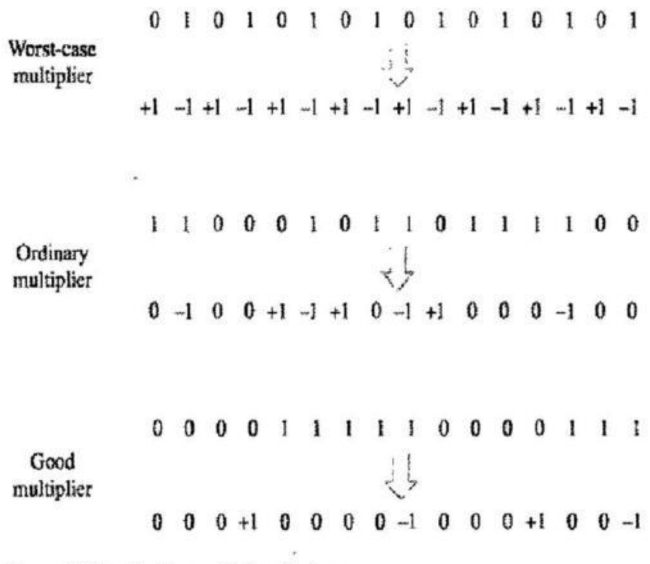


Figure 6.13 Booth recoded multipliers.

The Booth algorithm has two attractive features:

- **First**, it handles both positive and negative multipliers uniformly.
- **Second**, it achieves some efficiency in the number of additions required when the multiplier has a few large blocks of 1s.

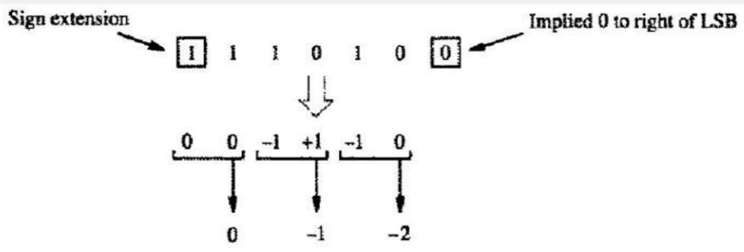
FAST MULIPLICATION:

There are two techniques for speeding up the multiplication operation. The first technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is n/2 for n-bit operands. The second technique reduces the time needed to add the summands (carry-save addition of summands method).

Bit-Pair Recoding of Multipliers:

This bit-pair recoding technique halves the maximum number of summands. It is derived from the Booth algorithm. Group the Booth-recoded multiplier bits in pairs, and observe the following: The pair (+1 -1) is equivalent to the pair (0 +1). That is, instead of adding —1 times the multiplicand M at shift position i to + 1 x M at position i + 1, the same result is obtained by adding +1 x M at position I Other examples are: (+1 0) is equivalent to (0 +2),(-l +1) is equivalent to (0 —1). and so on. Thus, if the Booth-recoded multiplier is examined two bits at a time, starting from the right, it can be rewritten in a form that requires at most one version of the multiplicand to be added to the partial product for each pair of multiplier bits. Figure 6.11a shows an example of bit-pair recoding of the multiplier in Figure 6.11, and Figure 6.14b shows a table of the multiplicand selection decisions for all possibilities.

The multiplication operation in Figure 6.11 is shown in Figure 6.15 as it would be computed using bit-pair recoding of the multiplier.



(a) Example of bit-pair recoding derived from Booth recoding

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	$0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$

(b) Table of multiplicand selection decisions

Figure 6.14 Multiplier bit-pair recoding.

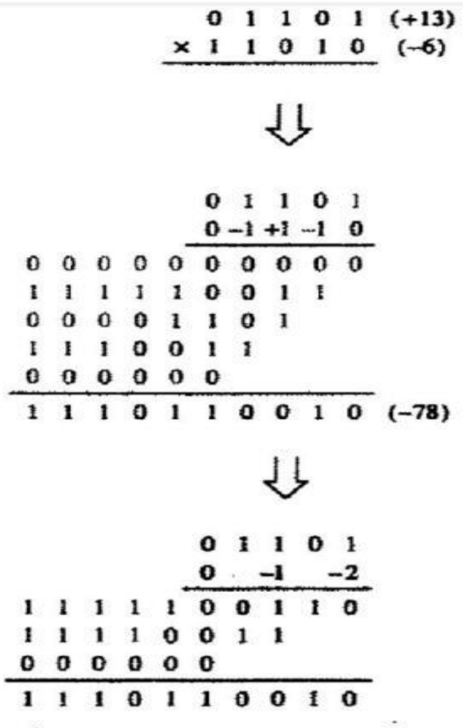
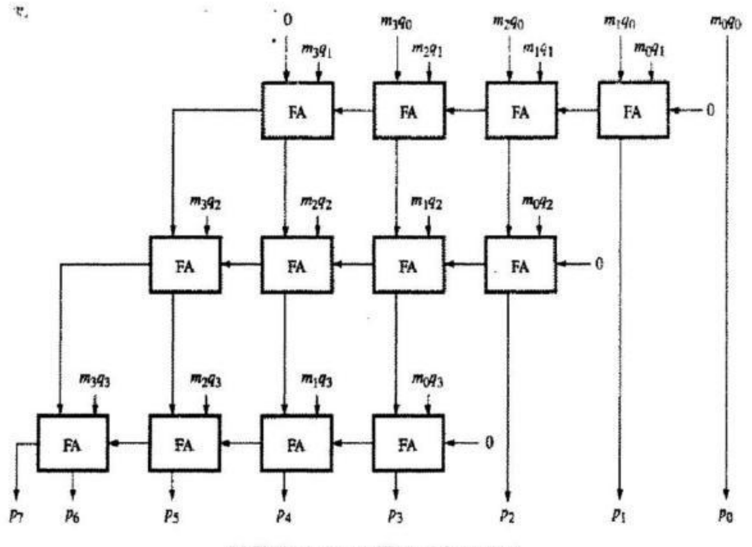


Figure 6.15 Multiplication requiring only $n/2$ summands.

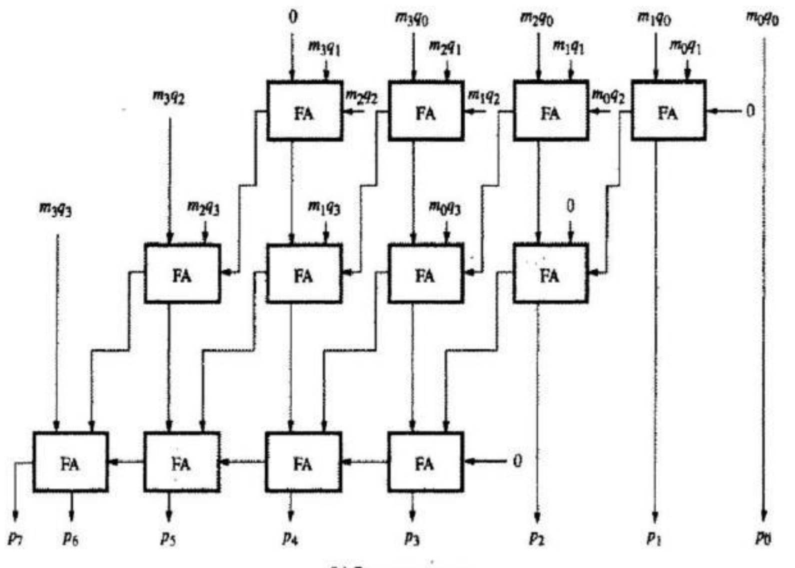
CARRY-SAVE ADDITION OF SUMMANDS

Multiplication requires the addition of several summands. A technique called *carry save addition (CSA)* speeds up the addition process. Consider the array for 4x4 multiplication shown in *Figure 6.16a*. This structure is the general array with the first row consisting of just the AND gates that implement the bit products m3q0, m2q0, m1q0, and m0q0.

Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions, as shown in Figure 6.16b. This frees up an input to three full adders in the first row. These inputs are used to introduce the third summand bit products m2q2, m1q2, and m0q2.



(a) Ripple-carry array (Figure 6.6 structure)



(b) Carry-save array

Figure 6.16 Ripple-carry and carry-save arrays for the multiplication operation M × Q = P for 4-bit operands.

```
                        1  0  1  1  0  1      (45)      M
                     x  1  1  1  1  1  1      (63)      Q
                     -------------------
                        1  0  1  1  0  1                A
                     1  0  1  1  0  1                   B
                  1  0  1  1  0  1                      C
               1  0  1  1  0  1                         D
            1  0  1  1  0  1                            E
         1  0  1  1  0  1                               F
      ----------------------------------
      1  0  1  1  0  0  0  1  0  0  1  1      (2,835)   Product
```

Figure 6.17 A multiplication example used to illustrate carry-save addition as shown in Figure 6.18.

Now, two inputs of each full adder in the second row are fed by sum and carry outputs from the first row. ‘The third input is used to introduce the bit products m2q3, m1q3, and m0q3 of the fourth summand. The high-order bit products m3q2 and m3q3 of the third and fourth summands are introduced into the remaining free inputs at the left end in the second and third rows. The saved carry bits and the sum bits from the second row are now added in the third row to produce the final product bits.

Delay through the carry-save array is somewhat less than delay through the ripple-carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay.

A more significant reduction in delay can be achieved as follows. Consider the addition of many summands, as required in the multiplication of longer operands. We can group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of § and C vectors in one full-adder delay. Next, we group all of the S and C vectors into threes, and perform carry-save addition on them, generating a further set of S and C vectors in one more full-adder delay. We continue with this process until there are only two vectors remaining. They can then be added in a ripple-carry or a carry-lookahead adder to produce the desired product.

Consider the example of adding the six shifted versions of the multiplicand for the case of multiplying two 6-bit unsigned numbers where all six bits of the multiplier are equal to 1. Such an example is shown in Figure 6.17. The six summands, A, B,..., F are added by carry-save addition in Figure 6.18. The, “blue boxes” in these two figures indicate the same operand bits, and show how they are reduced to sum and carry bits in Figure 6.18 by carry-save addition. Three levels of carry-save addition are performed, as shown schematically in Figure 6.19. It is clear from this figure that the final two vectors S4, and C4, are available in three full-adder delays after the six input summands are applied to level 1. The final regular addition operation on S4, and C4, which produces the product, can be done with either a ripple-carry or a carry-lookahead adder.

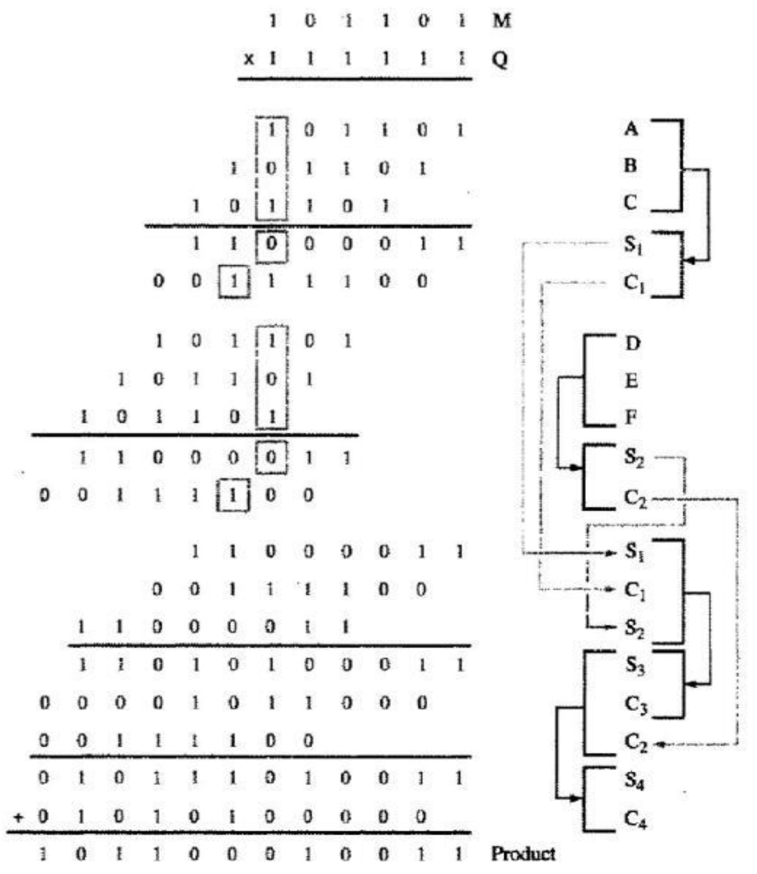


Figure 6.18 The multiplication example from Figure 6.17 performed using carry-save addition.

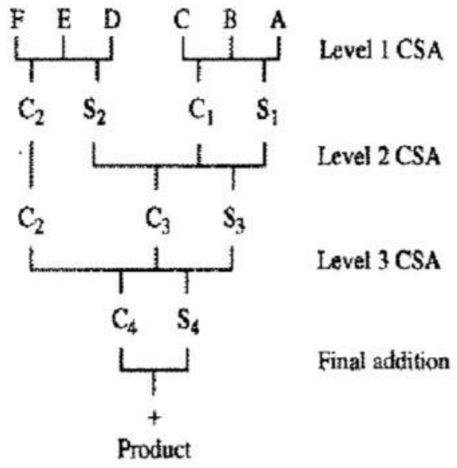


Figure 6.19 Schematic representation of the carry-save addition operations in Figure 6.18.

Summary of Fast Multiplication:

Bit-pair recoding of the multiplier, derived from the Booth algorithm, reduces the number of summands by a factor of 2. These summands can then be reduced to only 2 by using a relatively small number of carry-save addition steps. The final product can be generated by an addition operation that uses a carry-lookahead adder.

All three of these techniques — bit-pair recoding of the multiplier, carry-save addition of the summands, and lookahead addition have been used in various ways by the designers of high-performance processors to reduce the time needed to perform multiplication.

INTEGER DIVISION:

Positive-number multiplication operation is done manually in the way it is done in a logic circuit. A similar kind of approach can be used here in discussing integer division.

First, consider positive-number division. Figure 6.20 shows examples of decimal division and its binary form of division. First, let us try to divide 2 by13, and it does not work. Next, let us try to divide 27 by 13. Going through the trials, we enter 2 as the quotient and perform the required subtraction. The next digit of the dividend, 4, is brought down, and we finish by deciding that 13 goes into 14 once and the remainder is 1. Binary division is similar to this, with the quotient bits only 0 and 1.

A circuit that implements division by this longhand method operates as follows: It positions the divisor appropriately with respect to the dividend and performs a subtraction. If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and sub- traction is performed. On the other hand, if the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor H repositioned for another subtraction

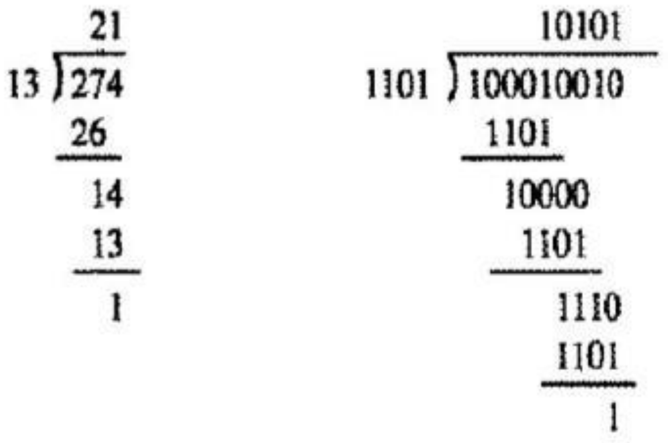


Figure 6.20 Longhand division examples.

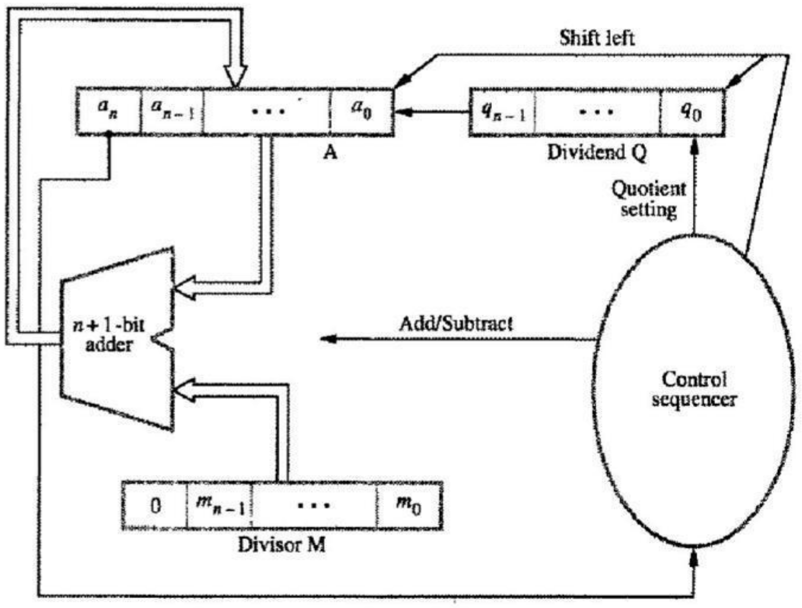


Figure 6.21 Circuit arrangement for binary division.

RESTORING DIVISION

Figure 6.21 shows a logic circuit arrangement that implements restoring division. An n-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of the operation. Register A is set to 0. After the division is complete, the n-bit quotient is in register Q and the remainder is in register A. The required subtractions are facilitated by using 2's complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions.

The following algorithm performs restoring division.

1. Shift A and Q left one binary position.

2. Subtract M from A, and place the answer back in A.

3. If the sign of A is 1, set q_0 to 0 and add M back to A (that is, restore A); otherwise, set $q_0$to 1.

Figure 6.22 shows a 4-bit example as it would be processed by the circuit in Figure 6.21.

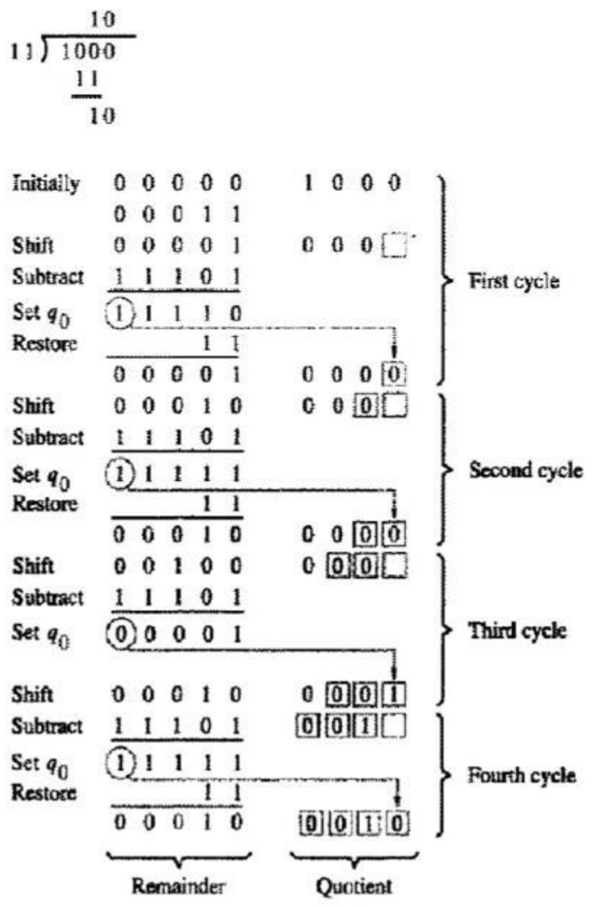


Figure 6.22 A restoring-division example.

NONRESTORING DIVISION

The restoring-division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction. Subtraction is said to be unsuccessful if the result is negative. Consider the sequence of operations that takes place after the subtraction operation in the preceding algorithm. If A is positive, we shift left and subtract M, that is, we perform 2A - M. If A is negative, we restore it by performing A + M, and then we shift it left and subtract M. This is equivalent to performing 2A + M. The q_0 bit is appropriately set to 0 or 1 after the correct operation has been performed.

Algorithm:

Step 1: Do the following n times:

1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.

2. Now, if the sign of A is 0, set q0 to 1; otherwise, set q0 to 0.

Step 2: If the sign of A is 1, add M to A.

Step 2 is needed to leave the proper positive remainder in A at the end of the n cycles of Step 1. The logic circuitry in Figure 16.21 can also be used to perform this algorithm. Note that the Restore operations are no longer needed, and that exactly one Add or Subtract operation is performed per cycle. Figure 6.23 shows how the division example in Figure 6.22 is executed by the nonrestoring-division algorithm.

FLOATING-POINT NUMBERS AND OPERATIONS:

Floating – point arithmetic is an automatic way to keep track of the radix point. The discussion so far was exclusively with fixed-point numbers which are considered as integers, that is, as having an implied binary point at the right end of the number. It is also possible to assume that the binary point is just to the right of the sign bit, thus representing a fraction or anywhere else resulting in real numbers. In the 2's-complement system, the signed value F, represented by the n-bit binary fraction

$$B = b_0.b_{-1}b_{-2} \ldots\ldots b_{-(n-1)} \text{ is given by}$$

$$F(B) = -b_0*2^0 + b_{-1}*2^{-1} + b_{-2}*2^{-2} + \ldots + b_{-(n-1)} \times 2^{-(n-l)}$$

Where the range of F is

$$-1 \leq F \leq 1 -2-(n-1)$$

Consider the range of values representable in a 32-bit, signed, fixed-point format. Interpreted as integers, the value range is approximately 0 to $\pm 2.15 \times 10^9$. If we consider them to be fractions, the range is approximately ±4.55 x 10-10 to ±1. Neither of these ranges is sufficient for scientific calculations, which might involve parameters like Avogadro's number ($6.0247 * 10^{23}$ mole^{-1}) or Planck's constant ($6.6254 * 10^{-27}$erg.s). Hence, we need to easily accommodate both very large integers and very small fractions. To do this, a computer must be able to represent numbers and operate on them in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds. In such a case, the binary point is said to float, and the numbers are called *floating-point numbers*. This distinguishes them from fixed-point numbers, whose binary point is always in the same position.

Because the position of the binary point in a floating-point number is variable, it must be given explicitly in the floating-point representation. For example, in the familiar decimal scientific notation, numbers may be written as 6.0247×10^{23}, $6.6254 * 10^{-27}$, $- 1.0341 \times 10^2$, -7.3000×10^{-14}, and so on. These numbers are said to be given to five significant digits. The scale factors (10^{23}, 10^{-27}, and so on) indicate the position of the decimal point with respect to the significant digits. By convention, when the decimal point is placed to the right of the first (nonzero) significant digit, the number is said to be normalized. Note that the base, 10, in the scale factor is fixed and does not need to appear explicitly in the machine representation of a floating-point number. The sign, the significant digits, and the exponent in the scale factor constitute the representation. We are thus motivated to define a floating-point number representation as one in which a number is represented by its sign, a string of significant digits, commonly called the *mantissa*, and an *exponent* to an implied base for the scale factor.

IEEE STANDARD FOR FLOATING-POINT NUMBERS:

General form and size for floating-point numbers in the decimal system is:

$$\pm X_1.X_2X_3X_4X_5X_6X_7 \times 10^{\pm Y_1Y_2}$$

Where X_i; and Y_i; are decimal digits.

A standard for representing floating-point numbers in 32 bits has been developed and specified in detail by the Institute of Electrical and Electronics Engineers (IEEE). The standard describes both the representation and the way in which the four basic arithmetic operations are to be performed. The 32-bit representation is given in Figure 6.24a. The sign of the number is given in the first bit, followed by a representation for the exponent (to the base 2) of the scale factor. Instead of the signed exponent, E, the value actually stored in the exponent field is an unsigned integer $E' = E + 127$.

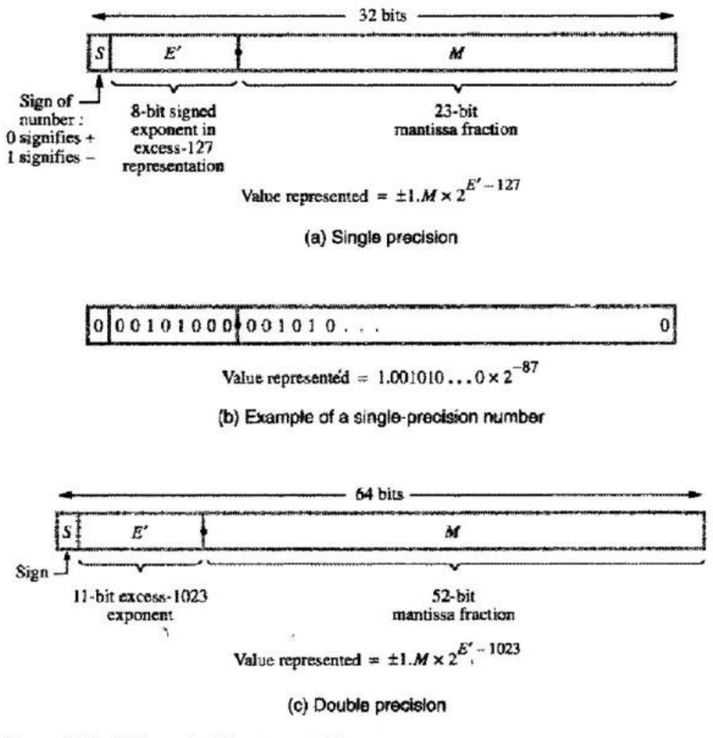

32 bits
S
E′
M
Sign of number : 0 signifies + 1 signifies −
8-bit signed exponent in excess-127 representation
23-bit mantissa fraction
Value represented = ±1.M × 2^(E′ − 127)
(a) Single precision
0 00101000 001010 . . . 0
Value represented = 1.001010...0 × 2^−87
(b) Example of a single-precision number
64 bits
S
E′
M
Sign
11-bit excess-1023 exponent
52-bit mantissa fraction
Value represented = ±1.M × 2^(E′ − 1023)
(c) Double precision

Figure 6.24 IEEE standard floating-point formats.

This is called the excess-127 format. Thus, E′ is in the range 0 < E' < 255. The end values of this range, 0 and 255, are used to represent special values. Therefore, the range of E' for normal values is 1 < E′ < 254. This means that the actual exponent, E, is in the range $-126 \leq E \leq 127$. The last 23 bits represent the mantissa, Since binary normalization is used, the most significant bit of the mantissa is always equal to 1. This bit is not explicitly represented, it is assumed to be to-the immediate left of the binary point, Hence, the 23 bits stored in the M field actually represent the fractional part of the mantissa, that is, the bits to the right of the binary point. An example of a single-precision floating-point number is shown in Figure 6.24b.

The 32-bit standard representation in Figure 6.24a is called a single-precision representation because it occupies a single 32-bit word. The scale factor has a range of 2^{-126} to 2^{+127}, which is approximately equal to $10^{\pm 38}$, The 24-bit mantissa provides approximately the same precision as a 7-digit decimal value.

To provide more precision and range for floating-point numbers, the IEEE standard also specifies a double precision format, as shown in Figure 6.24c. The double-precision format has increased exponent and mantissa ranges. The 11-bit excess-1023 exponent A' has the range i < E' < 2046 for normal values, with 0 and 2047 used to indicate special values, as before. Thus, the actual exponent E is in the range -1022 < E < 1023, providing scale factors of 2^{-1022} to 2^{1023}. The 53-bit mantissa provides a precision equivalent to about 16 decimal digits.

Two basic aspects of operating with floating-point numbers:

First, if a number is not normalized, it can always be put in normalized form by shifting the fraction and adjusting the exponent. Figure 6.25 shows an unnormalized value, 0.0010110... x 2^9, and its normalized version, 1.0110... x 2^6. Since the scale factor is in the form 2^i, shifting the mantissa right or left by one bit position is compensated by an increase or a decrease of 1 in the exponent, respectively. This is occurrence of underflow.

Second, as computations proceed, a number that does not fall in the representable range of normal numbers might be generated. This is occurrence of overflow.

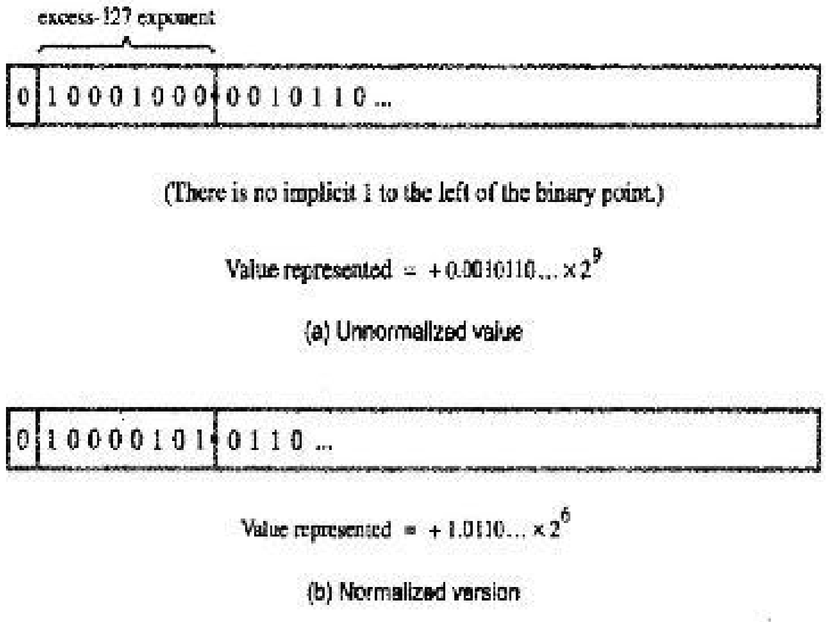

excess-127 exponent
0 10001000 0010110 ...
(There is no implicit 1 to the left of the binary point.)
Value represented = +0.0010110... x 2^9
(a) Unnormalized value
0 10000101 0110 ...
Value represented = +1.0110... x 2^6
(b) Normalized version

Figure 6.25 Floating-point normalization in IEEE single-precision format.

Special Values:

The end values 0 and 255 of the excess-127 exponent E′ are used to represent special values. When E′ = 0 and the mantissa fraction M is zero, the value exact 0 is represented. When E′ = 255 and M = 0, the value ∞ is represented, where ∞ is the result of dividing a normal number by zero. The sign bit is still part of these representations, so there are ±0 and ±∞ representations.

When E' = 0 and M # 0, denormal numbers are represented. Their -value is $\pm 0.M \times 2^{-126}$. Therefore, they are smaller than the smallest normal number. The purpose of introducing denormal numbers is to allow for *gradual underflow*, providing an extension of the range of normal representable numbers that is useful in dealing with very small numbers in certain situations. When E′ = 255 and M ≠ 0, the value represented is called Not Number (NaN).

Exceptions:

In conforming to the IEEE Standard, a processor must set exception flags if any of the following occur in performing operations: underflow, overflow, and divide by zero, inexact, invalid. Inexact is the name for a result that requires rounding in order to be represented in one of the normal formats. An invalid exception occurs if operations such as 0/0 or √-1 are attempted. When exceptions occur, the results are set to special values.

ARITHMETIC OPERATIONS ON FLOATING-POINT NUMBERS:

The rule for addition and subtraction can be stated as follows:

Add/Subtract Rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissas and determine the sign of the result.
4. Normalize the resulting value, if necessary.

Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissas is needed.

Multiply Rule

1. Add the exponents and subtract 127.
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

Divide Rule

1. Subtract the exponents and add 127.
2. Divide the mantissas and determine the sign of the result,
3. Normalize the resulting value, if necessary.

GUARD BITS AND TRUNCATION:

The mantissas of initial operands and final results are limited to 24 bits, including the implicit leading 1, it is important to retain extra bits, often called *guard bits*.

Removing guard bits in generating a final result requires that the extended mantissa be *truncated* to create a 24-bit number that approximates the longer version.

There are several ways to truncate.

- 1st method is to remove the guard bits and make no changes in the retained bits. This is called *chopping*. The result of chopping is a biased approximation because the error range is not symmetrical about 0.
- 2nd method is *Von Neumann rounding*. If the bits to be removed are all Os, they are simply dropped, with no changes to the retained bits. However, if any of the bits to be removed are 1, the least significant bit of the retained bits is set to 1. Although the range of error is larger with this technique than it is with chopping, the maximum magnitude is the same, and the approximation is unbiased because the error range is symmetrical about 0.
- The third truncation method is a rounding procedure. Rounding achieves the closest approximation to the number being truncated and is an unbiased technique. The procedure is as follows: A 1 is added to the LSB position of the bits to be retained if there is a 1 in the MSB position of the bits being removed.

IMPLEMENTING FLOATING-POINT OPERATIONS

An example of the implementation of floating-point operations is shown in Figure 6.26. This is a block diagram of a hardware implementation for the addition and subtraction of 32-bit floating-point operands that have the format shown in Figure 6.24a.

1. In step 1, the sign is sent to the SWAP network in the upper right corner. If the sign is 0, then E'_A, > E'_B, and the mantissas M_A, and M_B are sent straight through the SWAP network. This results in M_B, being sent to the SHIFTER, to be shifted n positions to the right. The other mantissa, M_A, is sent directly to the mantissa adder/subtractor. If the sign is 1, then $E'_A < E'_B$ and the mantissas are swapped before they are sent to the SHIFTER.
2. Step 2 is performed by the two-way multiplexer, MUX, near the bottom left corner of the figure. The exponent of the result, E′, is tentatively determined as E'_A if $E'_A \geq E'_B$, or E'_B if $E'_A < E'_B$, based on the sign of the difference resulting from comparing exponents in step 1.
3. Step 3 involves the major component, the mantissa adder/subtractor in the middle of the figure. The CONTROL logic determines whether the mantissas are to be added or subtracted. This is decided by the signs of the operands (S_A, and S_B) and the operation (Add or Subtract) that is to be performed on the operands. The CONTROL logic also determines the sign of the result, S_R.
4. Step 4 of the Add/Subtract rule consists of normalizing the result of step 3, mantissa M. The number of leading zeros in M determines the number of bit shifts, X, to be applied to M. The normalized value is truncated to generate the 24-bit mantissa, M_R, of the result. The value X is also subtracted from the tentative result exponent E′ to generate the true result exponent E'_R.

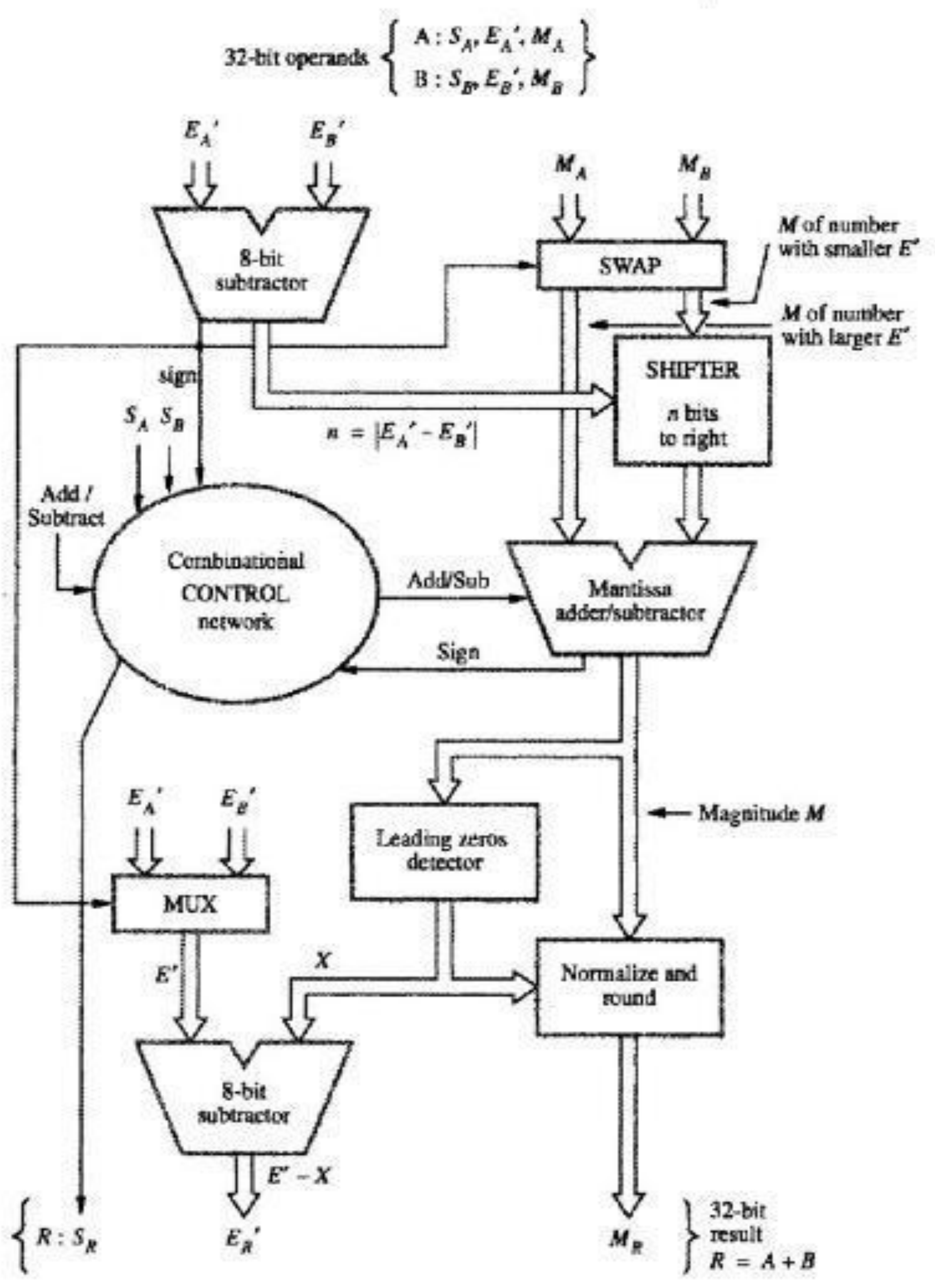


Figure 6.26 Floating-point addition-subtraction unit.

Part-II

BASIC PROCESSING UNIT

The heart of any computer is the central processing unit (CPU). The CPU executes all the machine instructions and coordinates the activities of all other units during the execution of an instruction. This unit is also called as the Instruction Set Processor (ISP). By looking at its internal structure, we can understand how it performs the tasks of fetching, decoding, and executing instructions of a program. The processor is generally called as the central processing unit (CPU) or micro processing unit (MPU).An high-performance processor can be built by making various functional units operate in parallel. High-performance processors have a pipelined organization where the execution of one instruction is started before the execution of the preceding instruction is completed. In another approach, known as superscalar operation, several instructions are fetched and executed at the same time. Pipelining and superscalar architectures provide a very high performance for any processor.

A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program. A program is a set of instructions performing a meaningful task. An instruction is command to the processor & is executed by carrying out a sequence of sub-operations called as micro-operations. Figure1 indicates various blocks of a typical processing unit. It consists of PC, IR, ID, MAR, MDR, a set of register arrays for temporary storage, Timing and Control unit as main units.

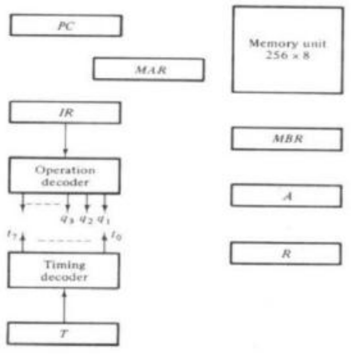


Fig-1

7.1 FUNDAMENTAL CONCEPTS

Execution of a program by the processor starts with the fetching of instructions one at a time, decoding the instruction and performing the operations specified. From memory, instructions are fetched from successive locations until a branch or a jump instruction is encountered. The processor keeps track of the address of the memory location containing the next instruction to be fetched using the program counter (PC) or Instruction Pointer (IP). After fetching an instruction, the contents of the PC are updated to point to the next instruction in the sequence. But, when a branch instruction is to be executed, the PC will be loaded with a different (jump/branch address).

Instruction register, IR is another key register in the processor, which is used to hold the op-codes before decoding. IR contents are then transferred to an instruction decoder (ID) for decoding. The decoder then informs the control unit about the task to be executed. The control unit along with the timing unit generates all necessary control signals needed for the instruction execution. Suppose that each instruction comprises 2 bytes, and that it is stored in one memory word. To execute an instruction, the processor has to perform the following three steps:

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are interpreted as an instruction code to be executed. Hence, they are loaded into the IR/ID. Symbolically, this operation can be written as
   $$IR \leftarrow [[PC]]$$
2. Assuming that the memory is byte addressable, increment the contents of the PC by 4, that is, $PC \leftarrow [PC] + 4$
3. Carry out the actions specified by the instruction in the IR.

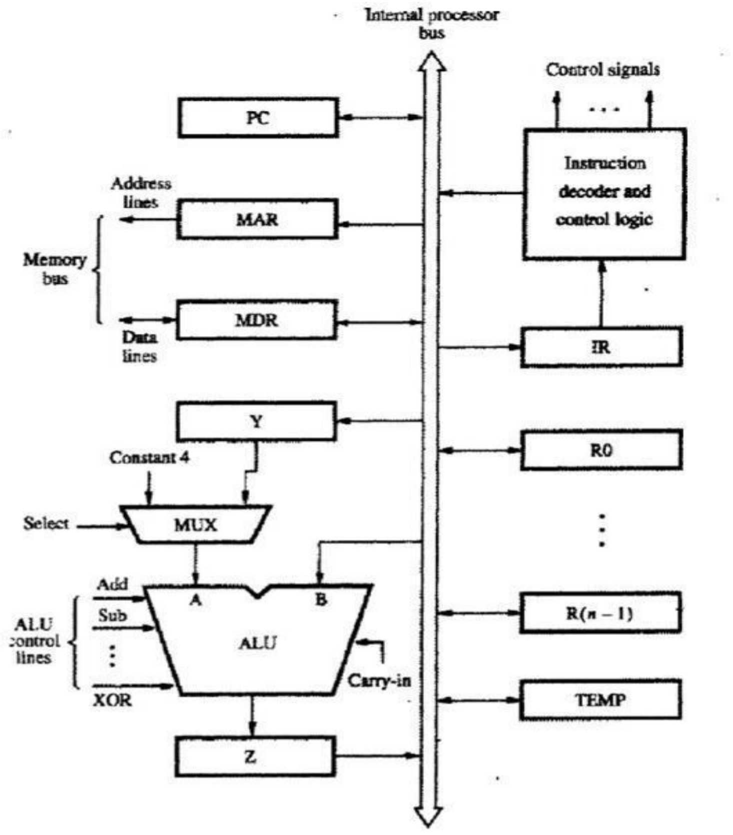


Figure 7.1 Single-bus organization of the datapath inside a processor.

In cases where an instruction occupies more than one word, steps 1 and 2 must be repeated as many times as necessary to fetch the complete instruction. These two steps together are usually referred to as the fetch phase; step 3 constitutes the execution phase.

To study these operations in detail, let us examine the internal organization of the processor. The main building blocks of a processor are interconnected in a variety of ways. A very simple organization is shown in Figure 7.1. A more complex structure that provides high performance will be presented at the end.

Figure shows an organization in which the arithmetic and logic unit (ALU) and all the registers are interconnected through a single common bus, which is internal to the processor. The data and address lines of the external memory bus are shown in Figure 7.1 connected to the internal processor bus via the memory data register, MDR, and the memory address register, MAR, respectively. Register MDR has two inputs and two outputs. Data may be loaded into MDR either from the memory bus or from the internal processor bus. The data stored in MDR may be placed on either bus. The input of MAR is connected to the internal bus, and its output is connected to the external bus. The control lines of the memory bus are connected to the instruction decoder and control logic block. This unit is responsible for issuing the signals that control the operation of all the units inside the processor and for interacting with the memory bus.

The number and use of the processor registers R0 through R(n - 1) vary considerably from one processor to another. Registers may be provided for general-purpose use by the programmer. Some may be dedicated as special-purpose registers, such as index registers or stack pointers. Three registers, Y, Z, and TEMP in Figure 7.1, have not been mentioned before. These registers are transparent to the programmer, that is, the programmer need not be concerned with them because they are never referenced explicitly by any instruction. They are used by the processor for temporary storage during execution of some instructions. These registers are never used for storing data generated by one instruction for later use by another instruction.

The multiplexer MUX selects either the output of register Y or a constant value 4 to be provided as input A of the ALU. The constant 4 is used to increment the contents of the program counter. We will refer to the two possible values of the MUX control input Select as Select4 and Select Y for selecting the constant 4 or register Y, respectively.

As instruction execution progresses, data are transferred from one register to another, often passing through the ALU to perform some arithmetic or logic operation. The instruction decoder and control logic unit is responsible for implementing the actions specified by the instruction loaded in the IR register. The decoder generates the control signals needed to select the registers involved and direct the transfer of data. The registers, the ALU, and the interconnecting bus are collectively referred to as the data path.

With few exceptions, an instruction can be executed by performing one or more of the following operations in some specified sequence:

1. Transfer a word of data from one processor register to another or to the ALU
2. Perform an arithmetic or a logic operation and store the result in a processor register
3. Fetch the contents of a given memory location and load them into a processor register
4. Store a word of data from a processor register into a given memory location

REGISTER TRANSFERS:

Instruction execution involves a sequence of steps in which data are transferred from one register to another. For each register, two control signals are used to place the contents of that register on the bus or to load the data on the bus into the register. This is represented symbolically in Figure 7.2. The input and output of register Ri are connected to the bus via switches controlled by the signals Ri_{in} and Ri_{out} respectively. When Ri_{in} is set to 1, the data on the bus are loaded into Ri. Similarly, when Ri_{out}, is set to 1, the contents of register Ri_{out} are placed on the bus. While Ri_{out} is equal to 0, the bus can be used for transferring data from other registers.

Suppose that we wish to transfer the contents of register RI to register R4. This can be accomplished as follows:

1. Enable the output of register R1 by setting Rl_{out}, to 1. This places the contents of R1 on the processor bus.
2. Enable the input of register R4 by setting $R4_{in}$ to 1. This loads data from the processor bus into register R4.

All operations and data transfers within the processor take place within time period defined by the processor clock. The control signals that govern a particular transfer are asserted at the start of the clock cycle. In our example, Rl_{out} and $R4_{in}$ are set to 1. The registers consist of edge-triggered flip-flops. Hence, at the next active edge of the clock, the flip-flops that constitute R4 will load the data present at their inputs. At the same time, the control signals Rl_{out} and $R4_{in}$ will return to 0. We will use this simple model of the timing of data transfers for the rest of this chapter. However, we should point out that other schemes are possible. For example, data transfers may use both the rising and falling edges of the clock. Also, when edge-triggered flip-flops are not used, two or more clock signals may be needed to guarantee proper transfer of data. This is known as multiphase clocking.

An implementation for one bit of register Ri is shown in Figure 7.3 as an example. A two-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop. When the control input Ri_{in} is equal to 1, the multiplexer selects the data on the bus. This data will be loaded into the flip-flop at the rising edge of the clock. When Ri_{in} is equal to 0, the multiplexer feeds back the value currently stored in the flip-flop.

The Q output of the flip-flop is connected to the bus via a tri-state gate. When Ri_{out}, is equal to 0, the gate's output is in the high-impedance (electrically disconnected) state. This corresponds to the open-circuit state of a switch. When Ri_{out} = 1, the gate drives the bus to 0 or 1, depending on the value of Q.

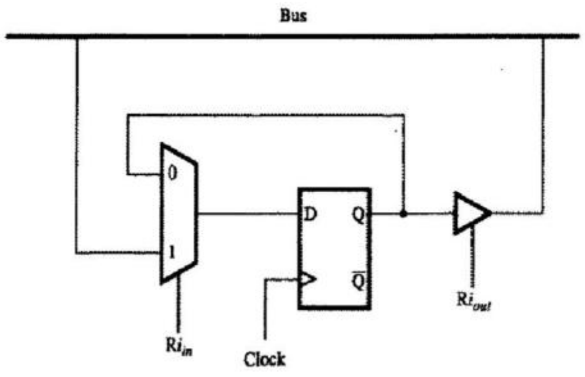


Figure 7.3 Input and output gating for one register bit.

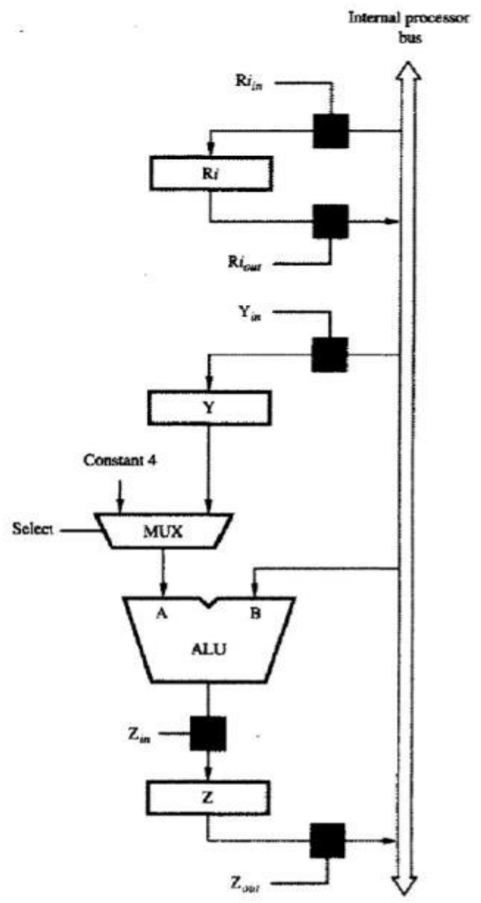


Figure 7.2 Input and output gating for the registers in Figure 7.1.

PERFORMING AN ARITHMETIC OR LOGIC OPERATION:

The ALU is a combinational circuit that has no internal storage. It performs arithmetic and logic operations on the two operands applied to its A and B inputs. In Figures 7.1 and 7.2, one of the operands is the output of the multiplexer MUX and the other operand is obtained directly from the bus. The result produced by the ALU is stored temporarily in register Z. Therefore, a sequence of operations to add the contents of register R1 to those of register R2 and store the result in register R3 is

1. $R1_{out}$, Y_{in}
2. $R2_{out}$, SelectY, Add, Z_{in}
3. Z_{out},$R3_{in}$

The signals whose names are given in any step are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive. Hence,

- In step 1, the output of register R1 and the input of register Y are enabled, causing the contents of R1 to be transferred over the bus to Y.
- In step 2, the multiplexer's Select signal is set to Select, causing the multiplexer to gate the contents of register Y to input A of the ALU. At the same time, the contents of register R2 are gated onto the bus and, hence, to input B. The function performed by the ALU depends on the signals applied to its control lines. In this case, the Add line is set to

1, causing the output of the ALU to be the sum of the two numbers at inputs A and B. This sum is loaded into register Z because its input control signal is activated.

- In step 3, the contents of register Z are transferred to the destination register, R3. This last transfer cannot be carried out during step 2, because only one register output can be connected to the bus during any clock cycle.

Fetching a word from Memory:
To fetch a word of information from memory, the processor has to specify the address of the memory location where this information is stored and request a Read operation. This applies whether the information to be fetched represents an instruction in a program or an operand specified by an instruction. The processor transfers the required address to the MAR, whose output is connected to the address lines of the memory bus, At the same time, the processor uses the control lines of the memory bus to indicate that a Read operation is needed. When the requested data are received from the memory they are stored in register MDR, from where they can be transferred to other registers in the processor.

The connections for register MDR are illustrated in Figure 7.4. It has four control signals: MDR_{in}, and MDR_{out} control the connection to the internal bus, and MDR_{inE} and MDR_{outE} control the connection to the external bus. The circuit in Figure 7.3 is easily modified to provide the additional connections. A three-input multiplexer can be used, with the memory bus data line connected to the third input. This input is selected when MDR_{inE}=1. A second tri-state gate, controlled by MDR_{outE} can be used to connect the output of the flip-flop to the memory bus.

During memory Read and Write operations, the timing of internal processor operations must be coordinated with the response of the addressed device on the memory bus. The processor completes one internal data transfer in one clock cycle. The speed of operation of the addressed device, on the other hand, varies with the device.

A control signal called Memory-Function-Completed (MFC) is used for the processor waits until it receives an indication that the requested Read operation has been completed. The addressed device sets this signal to 1 to indicate that the contents of the specified location have been read and are available on the data lines of the memory bus.

As an example of a read operation, consider the instruction Move (R1),R2. The actions needed to execute this instruction are:

1. MAR $\leftarrow$ [R1]
2. Start a Read operation on the memory bus
3. Wait for the MFC response from the memory
4. Load MDR from the memory bus
5. R2 $\leftarrow$ [MDR]

For simplicity, let us assume that the output of MAR is enabled all the time. When a new address is loaded into MAR, it will appear on the memory bus at the beginning of the next clock cycle, as shown in Figure 7.5. A Read control signal is activated at the same time MAR is loaded. This signal will cause the bus interface circuit to send a read command, MR, on the bus. With this arrangement, we have combined actions 1 and 2 above into a single control step. Actions 3 and 4 can also be combined by activating control signal MDR_{inE} while waiting for a response from the memory. Thus, the data received from the memory are loaded into MDR at the end of the clock cycle in which the MFC signal is received. In the next clock cycle, MDR_{out} is activated to transfer the data to register R2. This means that the memory read operation requires three steps, which can be described by the signals being activated as follows:

1. R1out, MAR_{in}, Read
2. MDR_{inE}, WMFC
3. MDR_{out}, $R2_{in}$

where WMFC is the control signal that causes the processor's control circuitry to wait for the arrival of the MFC signal.

Figure 7.5 shows that MDR_{inE} is set to 1 for exactly the same period as the read command, MR.

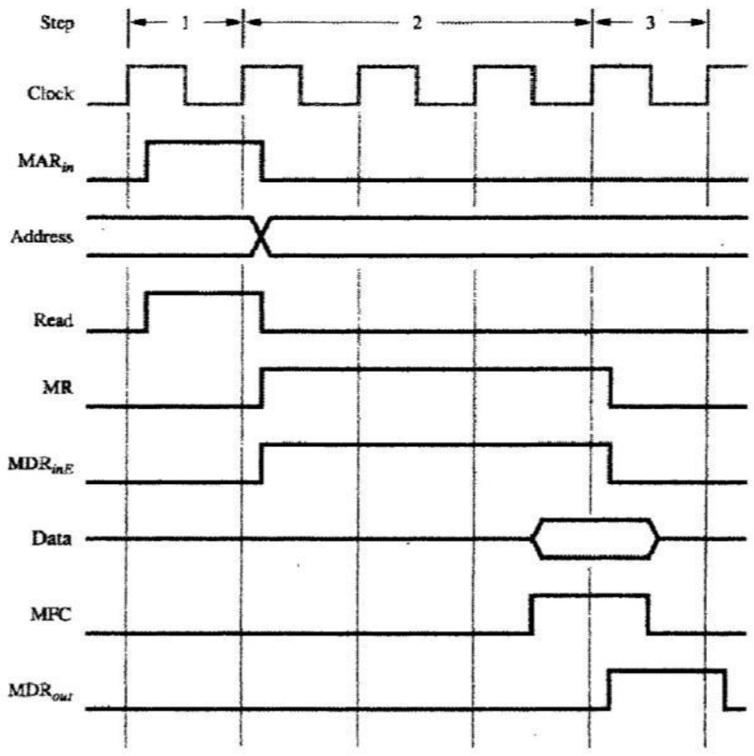


Figure 7.5 Timing of a memory Read operation.

STORING A WORD IN MEMORY

Writing a word into a memory location follows a similar procedure. The desired address is loaded into MAR. Then, the data to be written are loaded into MDR, and a Write command is issued. Hence, executing the instruction Move R2,(R1) requires the following sequence:

1. $R1_{out}$, MAR_{in}
2. $R2_{out}$, MDR_{in}, Write
3. MDR_{outE}, WMFC

The Write control signal causes the memory bus interface hardware to issue a Write command on the memory bus. The processor remains in step 3 until the memory operation is completed and an MFC response is received.

7.2 EXECUTION OF A COMPLETE INSTRUCTION:

Consider the instruction

Add (R3), R1

which adds the contents of a memory location pointed to by R3 to register R1.

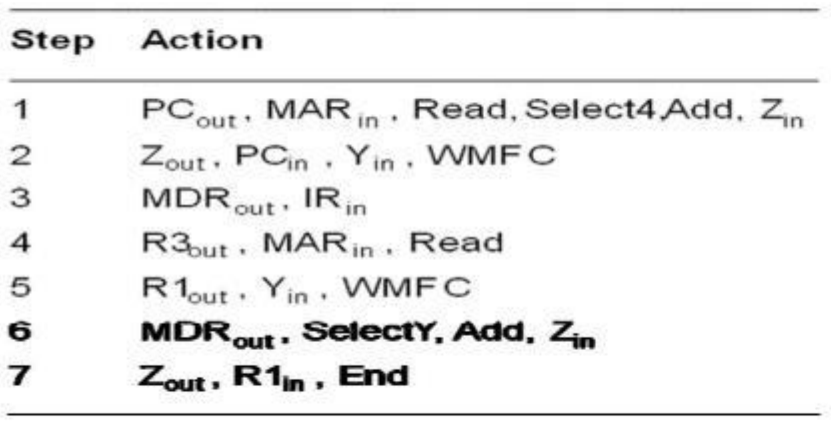
Executing this instruction requires the following actions:

1. Fetch the instruction.

2. Fetch the first operand (the contents of the memory location pointed to by R3).

3. Perform the addition.

4. Load the result into Rl.



Step	Action
1	PC_{out}, MAR_{in}, Read, Select4,Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	**MDR_{out}, SelectY, Add, Z_{in}**
7	**Z_{out}, $R1_{in}$, End**

Figure 7.6 gives the sequence of control steps required to perform these operations for the single-bus architecture of Figure 7.1. Instruction execution proceeds as follows:

In step I, the instruction fetch operation is initiated by loading the contents of the PC into the MAR and sending a Read request to the memory. The Select signal is set to Select4, which causes the multiplexer MUX to select the constant 4. This value is added to the operand at input B, which is the contents of the PC, and the result is stored in register Z. The updated value is moved from register Z back into the PC during step 2, while waiting for the memory to respond. In step 3, the word fetched from the memory is loaded into the IR.

Steps 1 through 3 constitute the instruction fetch phase, which is the same for all instructions. The instruction decoding circuit interprets the contents of the IR at the beginning of step 4. This enables the control circuitry to activate the control signals for steps 4 through 7, which constitute the execution phase. The contents of register R3 are transferred to the MAR in step 4, and a memory read operation is initiated.

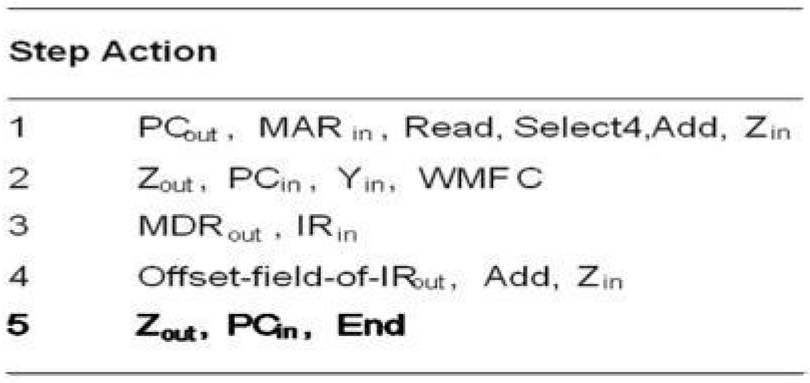
Then the contents of Rl are transferred to register Y in step 5, to prepare for the addition operation. When the Read operation is completed, the memory operand is available in register MDR, and the addition operation is performed in step 6. The contents of MDR are gated to the bus, and thus also to the B input of the ALU, and register Y is selected as the second input to the ALU by choosing Select Y. The sum is stored in register Z, then transferred to Rl in step 7. The End signal causes a new instruction fetch cycle to begin by returning to step 1.

This discussion accounts for all control signals in Figure 7.6 except Y in step 2. There is no need to copy the updated contents of PC into register Y when executing the Add instruction. But, in Branch instructions the updated value of the PC is needed to compute the Branch target address. To speed up the execution of Branch instructions, this value is copied into register Y in step 2. Since step 2 is part of the fetch phase, the same action will be performed for all instructions. This does not cause any harm because register Y is not used for any other purpose at that time.

Branch Instructions:

A branch instruction replaces the contents of the PC with the branch target address. This address is usually obtained by adding an offset X, which is given in the branch instruction, to the updated value of the PC. Listing in figure 8 below gives a control sequence that implements an unconditional branch instruction. Processing starts, as usual, with the fetch phase. This phase ends when the instruction is loaded into the IR in step 3. The offset value is extracted from the IR by the instruction decoding circuit, which will also perform sign extension if required. Since the value of the updated PC is already available in register Y, the offset X is gated onto the bus in step 4, and an addition operation is performed. The result, which is the branch target address, is loaded into the PC in step 5.

The offset X used in a branch instruction is usually the difference between the branch target address and the address immediately following the branch instruction.



Step	Action
1	PC_{out}, MAR_{in}, Read, Select4,Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMF C
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	**Z_{out}, PC_{in}, End**

For example, if the branch instruction is at location 2000 and if the branch target address is 2050, the value of X must be 46. The reason for this can be readily appreciated from the control sequence in Figure 7. The PC is incremented during the fetch phase, before knowing the type of instruction being executed. Thus, when the branch address is computed in step 4, the PC value used is the updated value, which points to the instruction following the branch instruction in the memory.

Consider now a conditional branch. In this case, we need to check the status of the condition codes before loading a new value into the PC. For example, for a Branch-on-negative (Branch<0) instruction, step 4 is replaced with

Offset-field-of-IRout Add, Zin, If N = 0 then End

Thus, if N = 0 the processor returns to step 1 immediately after step 4. If N = 1, step 5 is performed to load a new value into the PC, thus performing the branch operation.

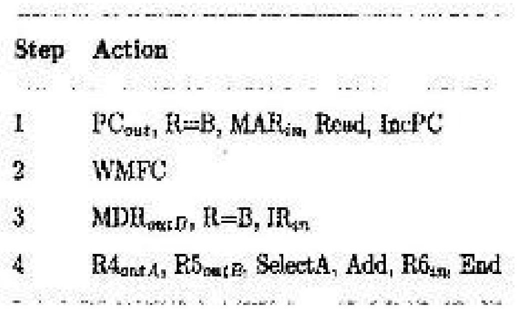
MULTIPLE-BUS ORGANIZATION:

The resulting control sequences shown are quite long because only one data item can be transferred over the bus in a clock cycle. To reduce the number of steps needed, most commercial processors provide multiple internal paths that enable several transfers to take place in parallel.

Figure 7.8 depicts a three-bus structure used to connect the registers and the ALU of a processor. All general-purpose registers are combined into a single block called the register file. In VLSI technology, the most efficient way to implement a number of registers is in the form of an array of memory cells similar to those used in the implementation of random-access memories (RAMs). The register file in Figure 7.8 is said to have three ports. There are two outputs, allowing the contents of two different registers to be accessed simultaneously and have their contents placed on buses A and B. The third port allows the data on bus C to be loaded into a third register during the same clock cycle.

Buses A and B are used to transfer the source operands to the A and B inputs of the ALU, where an arithmetic or logic operation may be performed. The result is transferred to the destination over bus C. If needed, the ALU may simply pass one of its two input operands unmodified to bus C. We will call the ALU control signals for such an operation R=A or R=B. The three-bus arrangement obviates the need for registers Y and Z.

A second feature in Figure 7.8 is the introduction of the Incremental unit, which is used to increment the PC by 4. The source for the constant 4 at the ALU input multiplexer is still useful. It can be used to increment other addresses, such as the memory addresses in Load Multiple and Store Multiple instructions.



Step	Action
1	PC_{out}, R=B, MAR_{in}, Read, IncPC
2	WMFC
3	MDR_{outB}, R=B, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End

Figure 7.9 Control sequence for the instruction Add R4,R5,R6 for the three-bus organization in Figure 7.8.

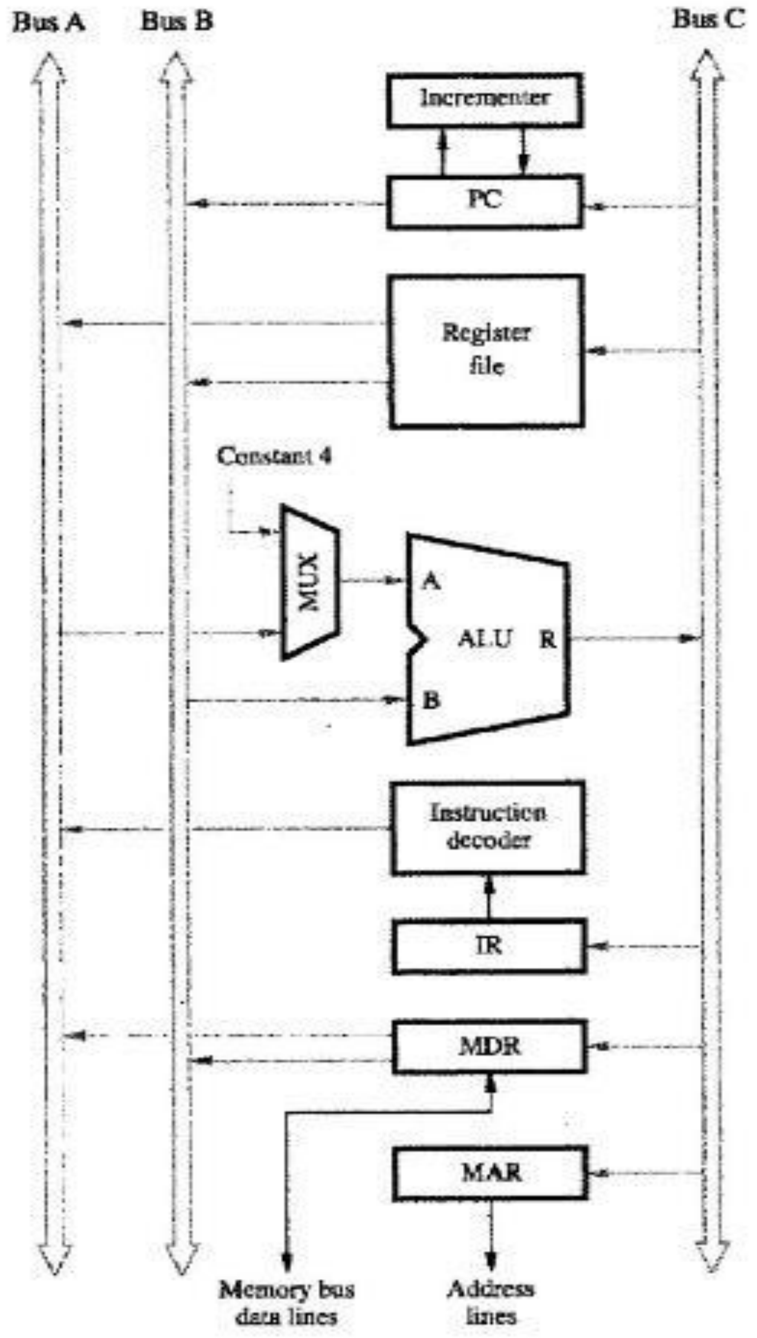


Figure 7.8 Three-bus organization of the datapath.

Consider the three-operand instruction

Add R4,R5,R6

The control sequence for executing this instruction is given in Figure 7.9.

In step 1, the contents of the PC are passed through the ALU, using the R=B control signal, and loaded into the MAR to start a memory read operation. At the same time the PC is incremented by 4. Note that the value loaded into MAR is the original contents of the PC. The incremented value is loaded into the PC at the end of the clock cycle and will not affect the contents of MAR. In step 2, the processor waits for MFC and loads the data received into MDR, then transfers them to IR in step 3. Finally, the execution phase of the instruction requires only one control step to complete, step 4.

By providing more paths for data transfer a significant reduction in the number of clock cycles needed to execute an instruction is achieved.

HARDWIRED CONTROL:

To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence. Computer designers use a wide variety of techniques to solve this problem. The approaches used fall into one of two categories: *hardwired control and micro programmed control*.

The required control signals are determined by the following information:

- Contents of the control step counter
- Contents of the instruction register
- Contents of the condition code flags
- External input signals, such as MFC and interrupt requests

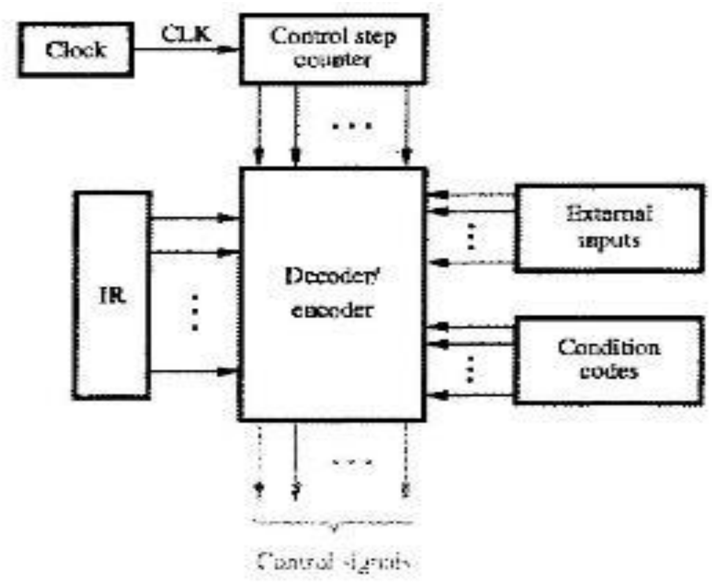


Figure 7.10 Control unit organization.

To gain insight into the structure of the control unit, we start with a simplified view of the hardware involved. The decoder/encoder block in Figure 7.10 is a combinational circuit that generates the required control outputs, depending on the state of all its inputs. By separating the decoding and encoding functions, we obtain the more detailed block diagram in Figure 7.11. The step decoder provides a separate signal line for each step, or time slot, in the control sequence. Similarly, the output of the instruction decoder consists of a separate line for each machine instruction. For any instruction loaded in the IR, one of the output lines INS_1 through INS_m is set to 1, and all other lines are set to 0. The input signals to the encoder block in Figure 7.11 are combined to generate the individual control signals Y_{in}, PC_{out}, Add, End, and so on. An example of how the encoder generates the Z_{in} control signal for the processor organization in Figure 7.1 is given in Figure 7.12. This circuit implements the logic function

$$Zin=T1+T6 - ADD + T4-BR+---$$

This signal is asserted during time slot T_1 for all instructions, during T_6 for an Add instruction, during T4 for an unconditional branch instruction, and so on. The logic function for Z_{in} is derived from the control sequences in Figures 7.6 and 7.7. As another example, Figure 7.13 gives a circuit that generates the End control signal from the logic function

$$End = T7 \bullet ADD + T5 \bullet BR + (T5 \bullet N + T4 \bullet \overline{N}) \bullet BRN + \bullet \bullet \bullet$$

The End signal starts a new instruction fetch cycle by resetting the control step counter to its starting value. Figure 7.11 contains another control signal called RUN. When set to 1, RUN causes the counter to be incremented by one at the end of every clock cycle. When RUN is equal to 0, the counter stops counting. This is needed whenever the WMFC signal is issued, to cause the processor to wait for the reply from the memory.

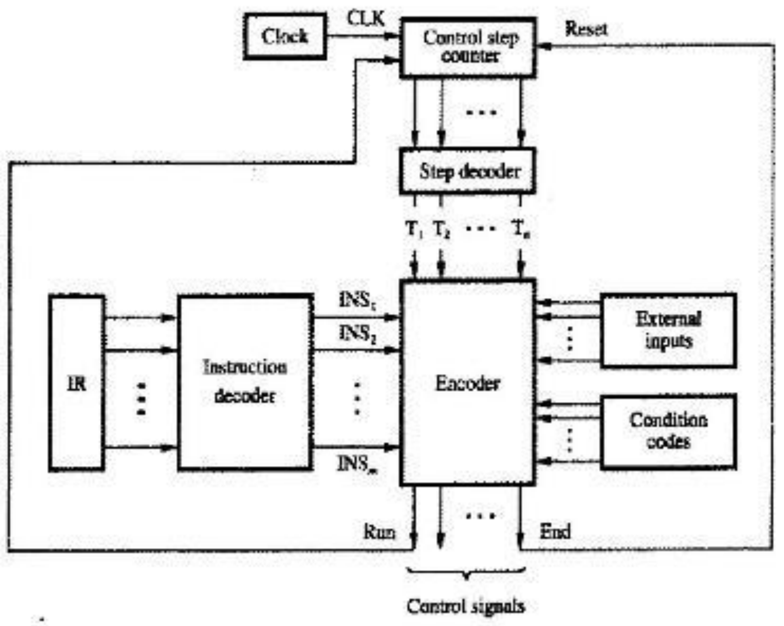


Figure 7.11 Separation of the decoding and encoding functions.

The control hardware shown can be viewed as a state machine that changes from one state to another in every clock cycle, depending on the contents of the instruction register, the condition codes, and the external inputs. The outputs of the state machine are the control signals. The sequence of operations carried out by this machine is determined by the wiring of the logic elements, hence the name "***hardwired***." A controller that uses this approach can operate at high speed. However, it has little flexibility, and the complexity of the instruction set it can implement is limited.

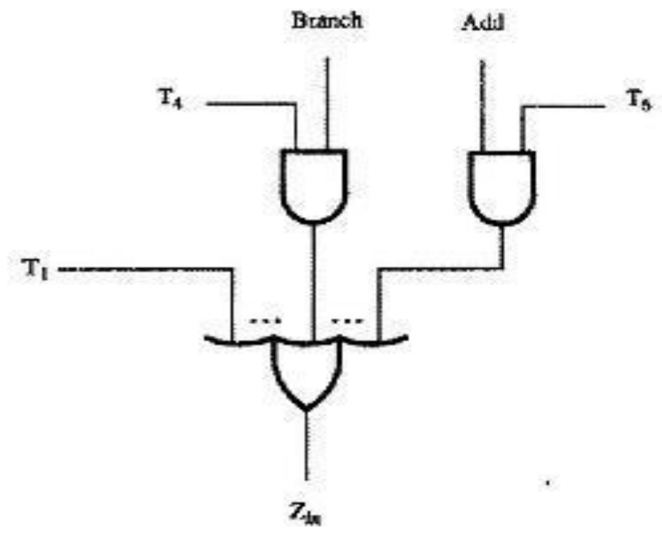


Figure 7.12 Generation of the Z_{in} control signal for the processor in Figure 7.1.

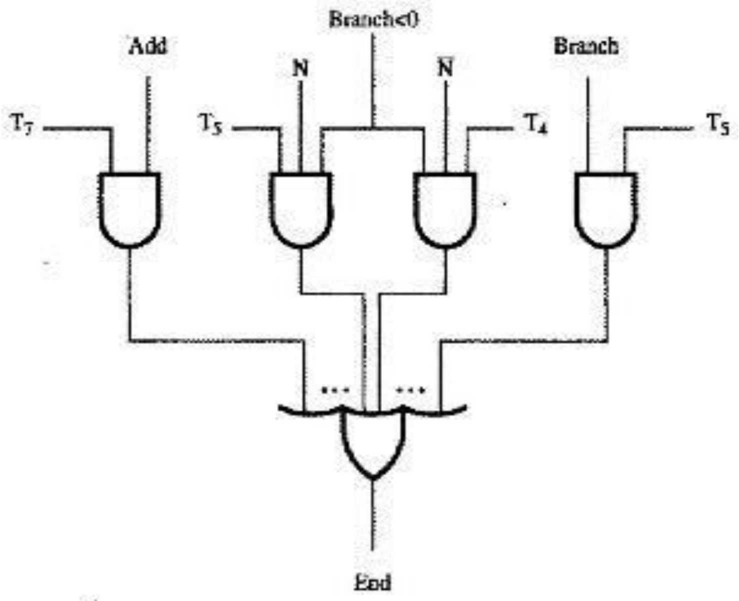


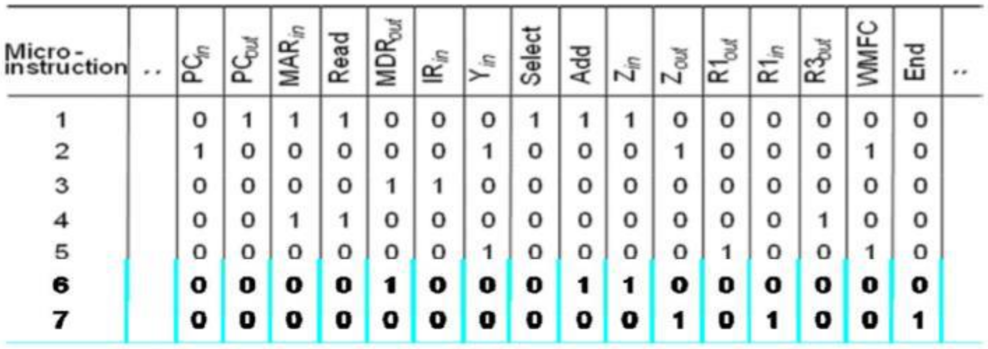
Figure 7.13 Generation of the End control signal.

MICROPROGRAMMED CONTROL:

Microprogrammed control signals are generated by a program similar to machine language programs.

ALU is the heart of any computing system, while Control unit is its brain. The design of a control unit is not unique; it varies from designer to designer. Some of the commonly used control logic design methods are:

- Sequence Reg & Decoder method
- Hard-wired control method
- PLA control method
- Micro-program control method



Micro-instruction	..	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	..
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	
7		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	

Figure 7.15 An example of microinstructions for Figure 7.6.

A control word (CW) is a word whose individual bits represent the various control signals. Each of the control steps in the control sequence of an instruction defines a unique combination of 1s and 0s in the CW. A sequence of CWs corresponding to the control sequence of a machine instruction constitutes the *microroutine* for that instruction, and the individual control words in this microroutine are referred to as *microinstructions*.

The micro routines for all instructions in the instruction set of a computer are stored in a special memory called the control store. The control unit can generate the control signals for any instruction by sequentially reading the CWs of the corresponding micro routine from the control store.

In Figure 7.16 to read the control words sequentially from the control store, a micro program counter (μPC) is used. Every time a new instruction is loaded into the IR, the output of the block labeled "starting address generator" is loaded into the μPC. The μPC is then automatically incremented by the clock, causing successive microinstructions to be read from the control store. Hence, the control signals are delivered to various parts of the processor in the correct sequence.

In microprogrammed control, an alternative approach to control unit is to use conditional branch microinstructions. In addition to the branch address, these microinstructions specify which of the external inputs, condition codes, or, possibly, bits of the instruction register should be checked as a condition for branching to take place.

The instruction Branch <0 may now be implemented by a microroutine such as that shown in Figure 7.17. After loading this instruction into IR, a branch microinstruction transfers control to the corresponding microroutine, which is assumed to start at location 25 in the control store. This address is the output of the starting address generator block in Figure 7.16. The microinstruction at location 25 tests the N bit of the condition codes. If this bit is equal to 0, a branch takes place to location 0 to fetch a new machine instruction. Otherwise, the microinstruction at location 26 is executed to put the branch target address into register Z, as in step 4 in Figure 7.7. The microinstruction in location 27 loads this address into the PC.

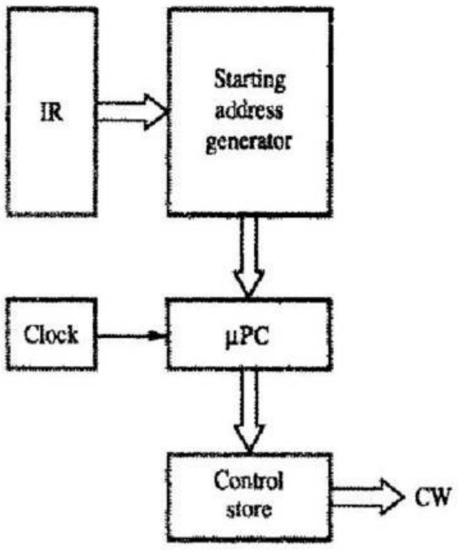
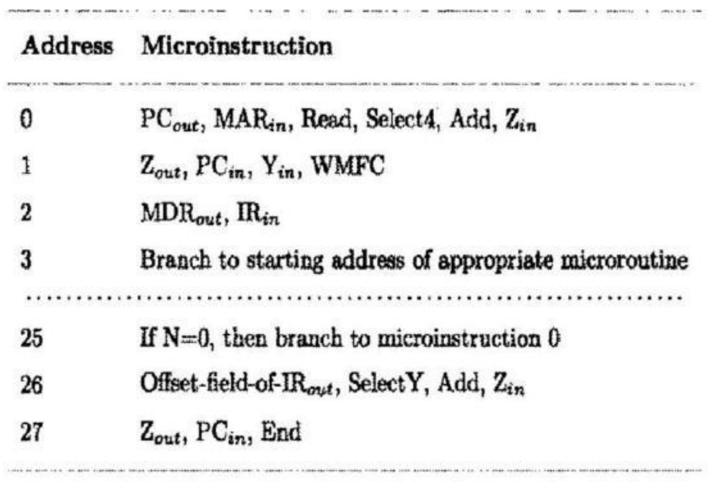


Figure 7.16 Basic organization of a microprogrammed control unit.



Address	Microinstruction
0	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
1	Z_{out}, PC_{in}, Y_{in}, WMFC
2	MDR_{out}, IR_{in}
3	Branch to starting address of appropriate microroutine
……	……
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Figure 7.17 Microroutine for the instruction Branch < 0.

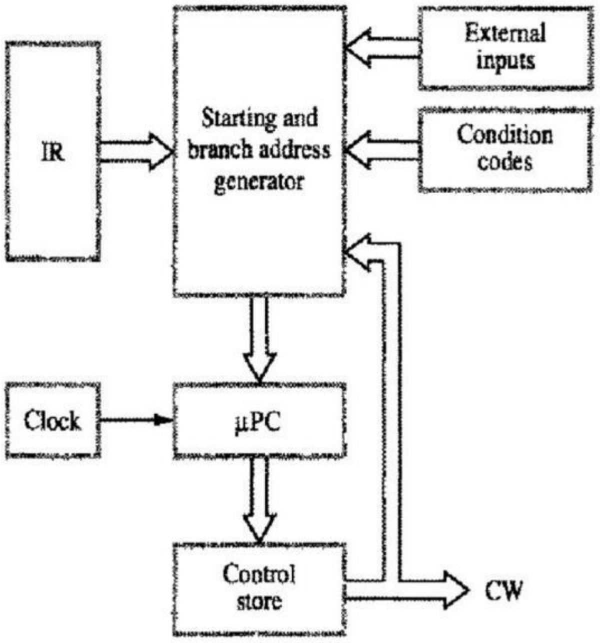


Figure 7.18 Organization of the control unit to allow conditional branching in the microprogram.

To support microprogram branching, the organization of the control unit should be modified as shown in Figure 7.18. The starting address generator block of Figure 7.16 becomes the starting and branch address generator. This block loads a new address into the μPC when a microinstruction instructs it to do so. To allow implementation of a conditional branch, inputs to this block consist of the external inputs and condition codes as well as the contents of the instruction register. In this control unit, the μPC is incremented every time a new microinstruction is fetched from the micro program memory, except in the following situations:

1. When a new instruction is loaded into the IR, the μPC is loaded with the starting address of the micro routine for that instruction.
2. When a Branch microinstruction is encountered and the branch condition is satisfied, the μPC is loaded with the branch address.
3. When an End microinstruction is encountered, the μPC is loaded with the address of the first CW in the micro routine for the instruction fetch cycle.

ANNAMACHARYA INSTITUTE OF TECHNOLOGY AND SCIENCES::TIRUPATI
II B.Tech II Semester

L	T	P	C
3	0	0	3

(19APC0506) Computer Organization

Course Objectives:
- To learn the fundamentals of computer organization and its relevance to classical and modern problems of computer design
- To make the students understand the structure and behavior of various functional modules of a computer.
- To understand the techniques that computers use to communicate with I/O devices
- To study the concepts of pipelining and the way it can speed up processing.
- To understand the basic characteristics of multiprocessors

Unit I:
Basic Structure of Computer: Computer Types, Functional Units, Basic operational Concepts, Bus Structure, Software, Performance, Multiprocessors and Multicomputer.
Machine Instructions and Programs: Numbers, Arithmetic Operations and Programs, Instructions and Instruction Sequencing, Addressing Modes, Basic Input/output Operations, Stacks and Queues, Subroutines, Additional Instructions.
Unit II:
Arithmetic: Addition and Subtraction of Signed Numbers, Design and Fast Adders, Multiplication of Positive Numbers, Signed-operand Multiplication, Fast Multiplication, Integer Division, Floating-Point Numbers and Operations.
Basic Processing Unit: Fundamental Concepts, Execution of a Complete Instruction, Multiple-Bus Organization, Hardwired Control, Multiprogrammed Control.
Unit III:
The Memory System: Basic Concepts, Semiconductor RAM Memories, Read-Only Memories, Speed, Size and Cost, Cache Memories, Performance Considerations, Virtual Memories, Memory Management Requirements, Secondary Storage.
Unit IV:
Input/output Organization: Accessing I/O Devices, Interrupts, Processor Examples, Direct Memory Access, Buses, Interface Circuits, Standard I/O Interfaces.
Unit V:
Pipelining: Basic Concepts, Data Hazards, Instruction Hazards, Influence on Instruction Sets
Large Computer Systems: Forms of Parallel Processing, Array Processors, The Structure of General-Purpose, Interconnection Networks.

Textbook:
1. “Computer Organization”, Carl Hamacher, Zvonko Vranesic, Safwat Zaky, McGraw Hill Education, 5th Edition, 2013.

Reference Textbooks:
1. Computer System Architecture, M.Morris Mano, Pearson Education, 3rd Edition.
2. Computer Organization and Architecture, Themes and Variations, Alan Clements, CENGAGE Learning.
3. Computer Organization and Architecture, Smruti Ranjan Sarangi, McGraw Hill Education.
4. Computer Architecture and Organization, John P.Hayes, McGraw Hill Education.

Course Outcomes:
- Ability to use memory and I/O devices effectively
- Able to explore the hardware requirements for cache memory and virtual memory
- Ability to design algorithms to exploit pipelining and multiprocessors

UNIT-III

The Memory System: Basic Concepts, Semiconductor RAM Memories, Read-Only Memories, Speed, Size and Cost, Cache Memories, Performance Considerations, Virtual Memories, Memory Management Requirements, Secondary Storage.

THE MEMORY SYSTEM

SOME BASIC CONCEPTS

The maximum: size of the memory that can be used in any computer is determined by the *addressing scheme*.

For example,

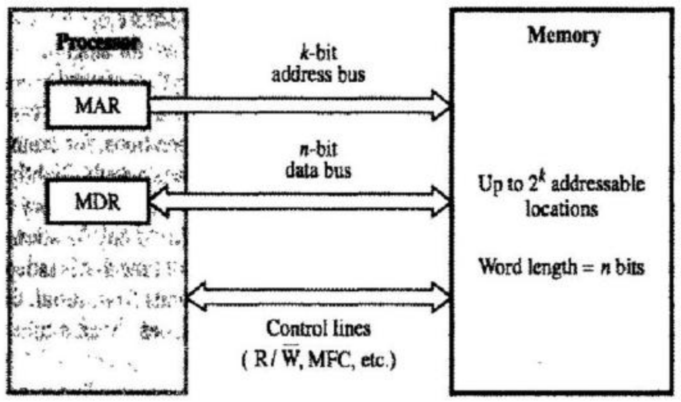
- 16-bit computer that generates 16-bit addresses is capable of addressing up to 2^{16} = 64K memory locations.
- 32-bit addresses can utilize a memory that contains up to 2^{32} = 4G (giga) memory locations.
- 40-bit addresses can access up to 2^{40} = 1T (tera) locations.

The number of locations represents the *size of the address space* of the computer.

Most modern computers are byte addressable. The *big-endian arrangement* is used in the 68000 processor. The *little-endian arrangement* is used in Intel processors.

Word length of a computer is defined as the number of bits actually stored or retrieved in one memory access.

Consider, for example, a byte addressable computer whose instructions generate 32-bit addresses, When a 32-bit address is sent from the processor to the memory unit, the high-order 30 bits determine which word will be accessed. If a byte quantity is specified, the low-order 2 bits of the address specify which byte location is involved. In a *Read operation*, other bytes may be fetched from the memory, but they are ignored by the processor. If the byte operation is a *Write*, however, the control circuitry of the memory must ensure that the contents of other bytes of the same word are not changed.



· **Figure 5.1** Connection of the memory to the processor.

Data transfer between the memory and the processor takes place through the use of two processor registers, usually called MAR and MDR. If MAR is *k* bits long and MDR is *n* bits long, then the memory unit may contain up to 2^k addressable locations. During a memory cycle, *n* bits of data are transferred between the memory and the processor. This transfer takes place over the processor bus, which has *k* address lines and *n* data lines. The bus also includes the control lines Read/Write ($R/\overline{W}$) and Memory Function Completed (MFC) for coordinating data transfers. Other control lines may be added to indicate the number of bytes to be transferred.

The processor reads data from the memory by loading the address of the required memory location into the MAR register and setting the $R/\overline{W}$ line to 1. The memory responds by placing the data from the addressed location onto the data lines, and confirms this action by asserting the MFC signal. Upon receipt of the MFC signal, the processor loads the data on the data lines into the MDR register.

The processor writes data into a memory location by loading the address of this location into MAR and loading the data into MDR. It indicates that a write operation is involved by setting the $R/\overline{W}$ line to 0.

If read or write operations involve consecutive address locations is the main memory, then a “block transfer” operation can be performed in which the only address sent to the memory is the one that identifies the first location.

A useful measure of the *speed* of memory units is the time that elapses between the initiation of an operation and the completion of that operation, for example, the time between the Read and the MFC signals. This is referred to as the memory access time. Another important measure is the memory cycle time, which is the minimum time delay required between the initiations of two successive memory operations, for example, the time between two successive Read operations. The *cycle time* is usually slightly longer than the access time, depending on the implementation details of the memory unit.

A memory unit is called random-access memory (RAM) if any location can be accessed for a Read or Write operation in some fixed amount of time that is independent of the location’s address. This distinguishes such memory units from serial, or partly serial, access storage devices such as magnetic disks and tapes. Access time on the latter devices depends on the address or position of the data.

The processor of a computer can usually process instructions and data faster than they can be fetched from a reasonably priced memory unit.

Cache memory is used to reduce the memory access time. This is a small, fast memory that is inserted between the larger, slower main memory and the processor. It holds the currently active segments of a program and their data.

Virtual memory is another important concept related to memory organization. The memory control circuitry translates the address specified by the program into an address that can be used to access the physical memory. In such a case, an address generated by the processor is referred to as a virtual or logical address. The virtual address space is mapped onto the physical memory where data are actually stored. The mapping function is implemented by a special memory control circuit, often called the memory management unit. This mapping function can be changed during program execution according to system requirements.

Virtual memory is used to increase the apparent size of the physical memory.

SEMICONDUCTOR RAM MEMORIES

Semiconductor memories cycle times range from 100 ns to less than 10 ns. Because of rapid advances in VLSI (Very Large Scale Integration} technology, the cost of semiconductor memories has dropped dramatically.

INTERNAL ORGANIZATION OF MEMORY CHIPS:

Memory cells are usually organized in the form of an array, in which each cell is capable of storing one bit of information. Each row of cells constitutes a memory word, and all cells of a row are connected to a common line referred to as the word line, which is driven by the address decoder on the chip. The cells in each column are connected to a Sense/Write circuit by two bit lines, The Sense/Write circuits are connected to the data input/output lines of the chip, During a Read operation, these circuits sense, or read, the information stored in the cells selected by a word line and transmit this information to the output data lines. During a Write operation, the Sense/Write circuits receive input information and store it in the cells of the selected word.

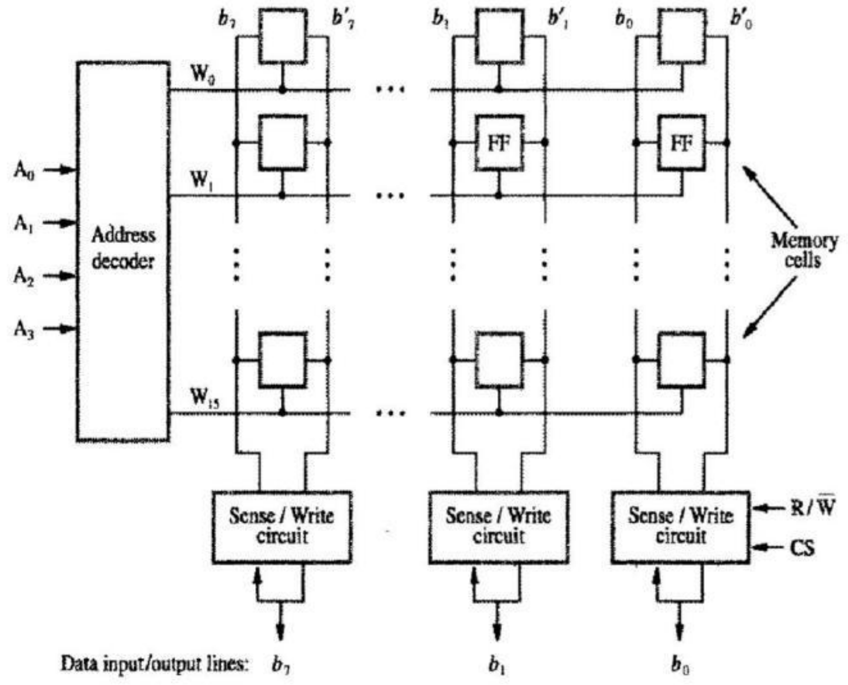


Figure 5.2 Organization of bit cells in a memory chip.

Figure 5.2 is an example of a very small memory chip consisting of 16 words of 8 bits each. This is referred to as a 16 x 8 organization. The data input and the data output of each Sense/Write circuit are connected to a single bidirectional data line that can be connected to the data bus of a computer. Two control lines, $R/\overline{W}$ and CS, are provided in addition to address and data lines.

The $R/\overline{W}$ (Read/$\overline{\text{Write}}$) input specifies the required operation, and the CS (Chip Select) input selects a given chip in a multichip memory system.

The memory circuit in Figure 5.2 stores 128 bits and requires 14 external connections for address, data, and control lines. This circuit requires 14 external connections, and allowing 2 pins for power supply and ground connections, can be manufactured in the form of a 16-pin chip. It can store 16 x 8 = 128 bits.

Another type of organization for 1k x 1 format is shown below: This circuit can be organized as a 128 x 8 memory, requiring a total of 19 external connections. Alternatively, the same number of cells can be organized into a 1K x | format. In this case, a 10-bit address is needed, but there is only one data line, resulting in 15 external connections. Figure 5.3 shows such an organization. The required 10-bit address is divided into two groups of 5 bits each to form the row and column addresses for the cell array. A row address selects a row of 32 cells, all of which are accessed in parallel.

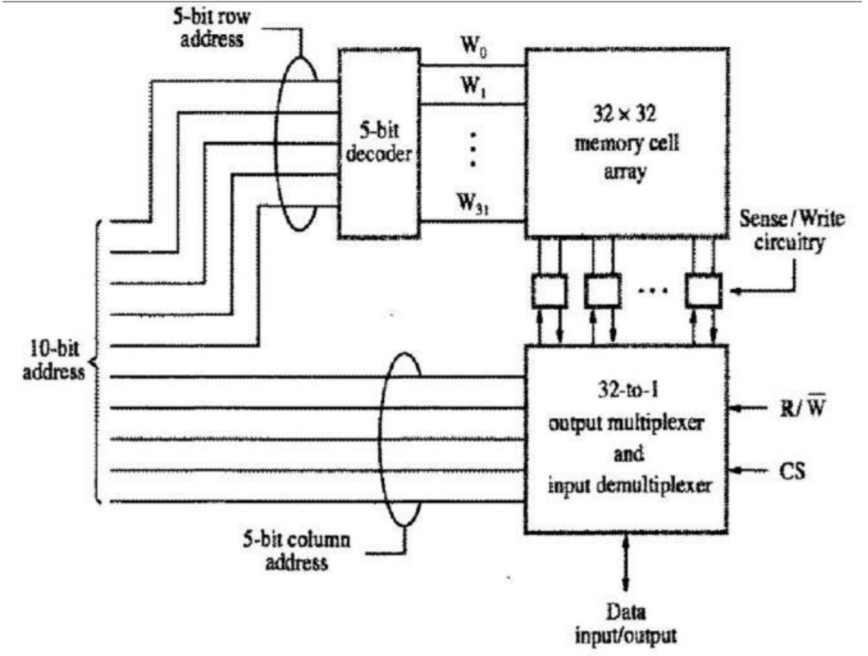


Figure 5.3 Organization of a 1K × 1 memory chip.

STATIC MEMORIES:

Memories that consist of circuits capable of retaining their state as long as power is applied are known as static memories.

Figure 5.4 illustrates how a static RAM (SRAM) cell may be implemented. Two inverters are cross-connected to form a latch. The latch is connected to two bit lines by transistors T_1 and T_2. These transistors act as switches that can be opened or closed under control of the word line. When the word line is at ground level, the transistors are turned off and the latch retains its state. For example, let us assume that the cell is in state 1 if the logic value at point X is 1 and at point Y is 0. This state is maintained as long as the signal on the word line is at ground level.

Read Operation

In order to read the state of the SRAM cell, the word line is activated to close switches T_1 and T_2. If the cell is in state 1, the signal on bit line b is high and the signal on bit line b′ is low. The opposite is true if the cell is in state 0. Thus, b and b′ are complements of each other. Sense/Write circuits at the end of the bit lines monitor the state of b and b′ and set the output accordingly.

Write Operation

The state of the cell is set by placing the appropriate value on bit line b and its complement on b′, and then activating the word line. This forces the cell into the corresponding state. The required signals on the bit lines are generated by the Sense/Write circuit.

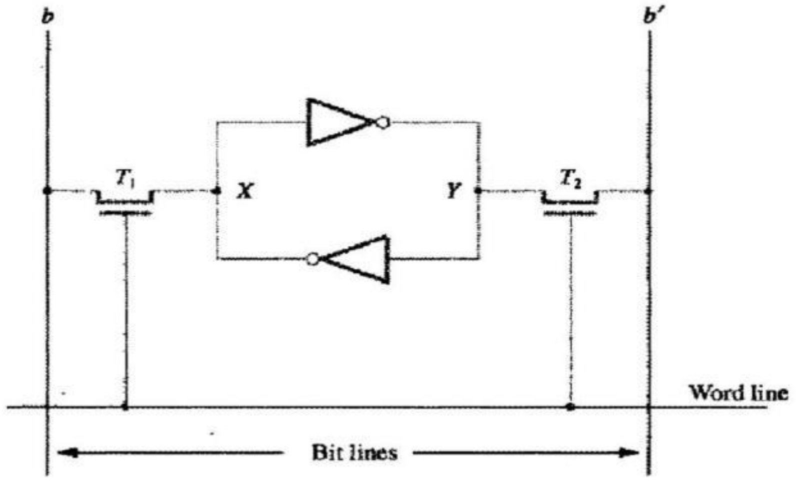


Figure 5.4 A static RAM cell.

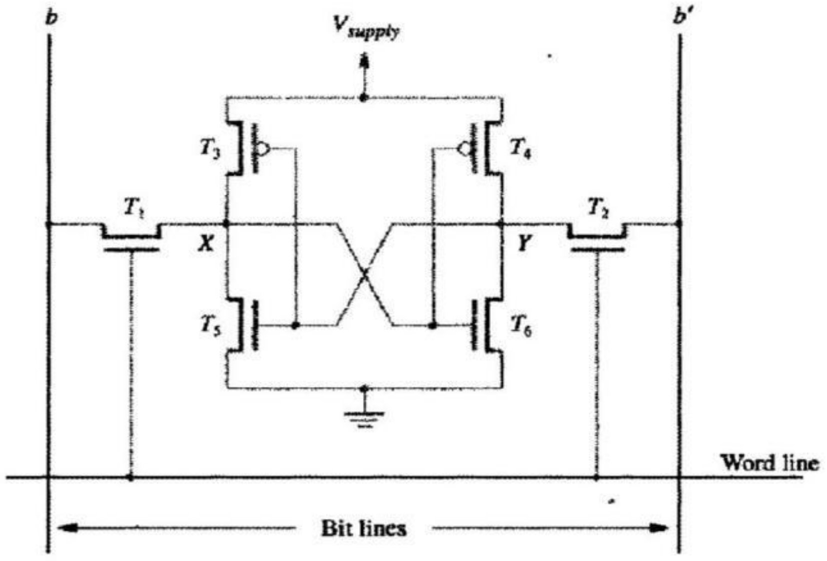


Figure 5.5 An example of a CMOS memory cell.

CMOS Cell:

In figure 5.5, Transistor pairs (T_3, T_5) and (T_4, T_6) form the inverters in the latch. For example, in state 1, the voltage at point X is maintained high by having transistors T_3 and T_6 on, while T_4 and T_5 are off. If T_1 and T_2 are turned on (closed), bit lines b and b′ will have high and low signals, respectively.

The power supply voltage, V_{supply}, is 5 V in older CMOS SRAMs of 3.3 V in new low-voltage versions. Continuous power is needed for the cell to retain its state. If power is interrupted, the cell's contents will be lost. When power is restored, the latch will settle into a stable state, but it will not necessarily be the same state the cell was in before the interruption. Hence, SRAMs are said to be *volatile memories* because their contents are lost when power is interrupted.

A major advantage of CMOS SRAMs is their very low power consumption because current flows in the cell only when the cell is being accessed.

Static RAMs can be accessed very quickly. SRAMs are used in applications where speed is of critical concern.

Static RAMs are fast, but they come at a high cost because their cells require several transistors.

ASYNCHRONOUS DRAMs:

Dynamic RAMs (DRAMs) are less expensive RAMs can be implemented if simpler cells are used. However, such cells do not retain their state indefinitely.

Information is stored in a dynamic memory cell in the form of a charge on a capacitor, and this charge can be maintained for only tens of milliseconds. Since the cell is required to store information for a much longer time, its contents must be periodically refreshed by restoring the capacitor charge to its full value.

After the transistor is turned off, the capacitor begins to discharge. This is caused by the capacitor’s own leakage resistance and by the fact that the transistor continues to conduct a tiny amount of current, measured in picoamperes, after it is turned off. Hence, the information stored in the cell can be retrieved correctly only if it is read before the charge on the capacitor drops below some threshold value. During a Read operation, the transistor in a selected cell is turned on. A sense amplifier connected to the bit line detects whether the charge stored on the capacitor is above the threshold value. If so, it drives the bit line to a full voltage that represents logic value 1. This voltage recharges the capacitor to the full charge that corresponds to logic value 1. If the sense amplifier detects that the charge on the capacitor is below the threshold value, it pulls the bit line to ground level, which ensures that the capacitor will have no charge, representing logic value 0. Thus, reading the contents of the cell automatically refreshes its contents. All cells in a selected row are read at the same time, which refreshes the contents of the entire row.

A 16-megabit DRAM chip, configured as 2M x 8, is shown in Figure 5.7. The cells are organized in the form of a 4K x 4K array. The 4096 cells in each row are divided into 512 groups of 8, so that a row can store 512 bytes of data, Therefore, 12 address bits are needed to select a row. Another 9 bits are needed to specify a group of 8 bits in the selected row. Thus, a 21-bit address is needed to access a byte in this memory. The high-order 12 bits and the low-order 9 bits of the address constitute the row and column addresses of a byte, respectively.

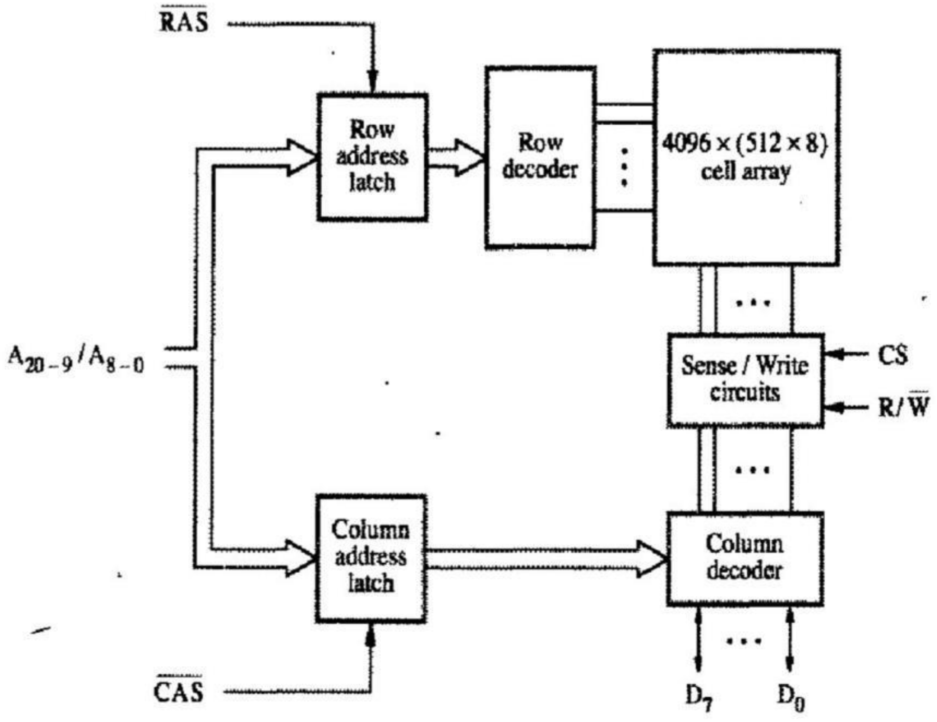


Figure 5.7 Internal organization of a 2M × 8 dynamic memory chip.

To reduce the number of pins needed for extremal connections, the row and column addresses are multiplexed on 12 pins. During a Read or a Write operation, the row address is applied first. It is loaded into the row address latch in response to a signal pulse on the Row Address Strobe (RAS) input of the chip. Then a Read operation is initiated, in which all cells on the selected row are read and refreshed. Shortly after the row address is loaded, the column address is applied to the address pins and loaded into the column address latch under control of the Column Address Strobe (CAS) signal. The information in this latch is decoded and the appropriate group of 8 Sense/Write circuits is selected. If the $R/\overline{W}$ control signal indicates a Read operation, the output values of the selected circuits are transferred to the data lines, D_{7-0}. For a Write operation, the information on the D_{7-0} lines is transferred to the selected circuits.

Applying a row address causes all cells on the corresponding row to be read and refreshed during both Read and Write operations. To ensure that the contents of a DRAM are maintained, each row of cells must be accessed periodically. A refresh circuit usually performs this function automatically. Many dynamic memory chips incorporate a refresh facility within the chips themselves.

A specialized memory controller circuit provides the necessary control signals, RAS and CAS, that govern the timing. The processor must take into account the delay in the response of the memory. Such memories are referred to as asynchronous DRAMS.

Because of their high density and low cost, DRAMs are widely used in the memory units of computers. Available chips range in size from 1M to 256M bits, and even larger chips are being developed. To reduce the number of memory chips needed in a given computer, a DRAM chip is organized to read or write a number of bits in parallel.

Fast Page Mode

When the DRAM in Figure 5.7 is accessed, the contents of all 4096 cells in the selected row ate sensed, but only 8 bits are placed on the data lines $D_{7\text{-}0}$. This byte is selected by the column address bits $A_{8\text{-}0}$. A simple modification can make it possible to access the other bytes in the same row without having to reselect the row. A latch can be added at the output of the sense amplifier in each column. The application of a row address will load the latches corresponding to all bits in the selected row. Then, it is only necessary to apply different column addresses to place the different bytes on the data lines.

The most useful arrangement is to transfer the bytes in sequential order, which is achieved by applying a consecutive sequence of column addresses under the control of successive CAS signals. This scheme allows transferring a block of data at a much faster rate than can be achieved for transfers involving random addresses. The block transfer capability is referred to as the fast page mode feature.

SYNCHRONOUS DRAMS

DRAMs whose operation is directly synchronized with a clock signal, such memories are knows as synchronous DRAMs (SDRAMs).

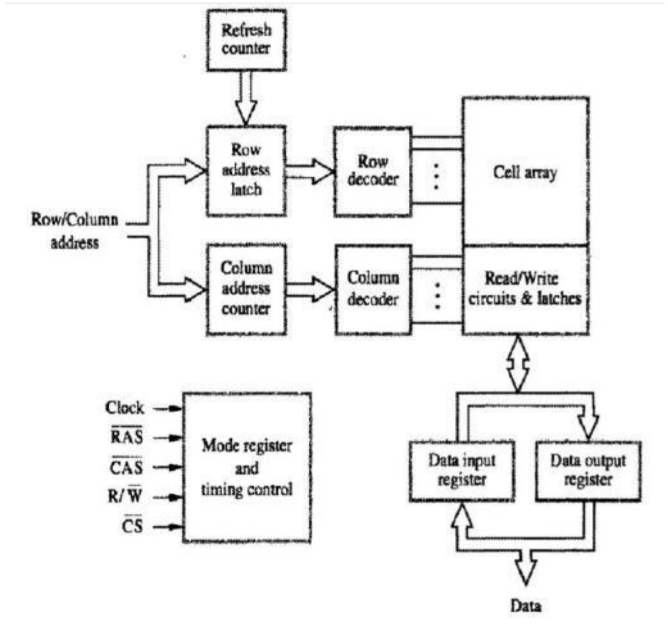


Figure 5.8 Synchronous DRAM.

The cell array is the same as in asynchronous DRAMs. The address and data connections are buffered by means of registers. The output of each sense amplifier is connected to a latch. A Read operation causes the contents of all cells in the selected row to be loaded into these latches. But, if an access is made for refreshing purposes only, it will not change the contents of these latches; it will merely refresh the contents of the cells. Data held in the latches that correspond to the selected column(s) are transferred into the data output register, thus becoming available on the data output pins.

SDRAMs have several different modes of operation, which can be selected by writing control information into a *mode register*. For example, burst operations of different lengths can be specified.

Figure 5.9 shows a timing diagram for a typical burst read of length 4. First, the row address is latched under control of the RAS signal. The memory typically takes 2 or 3 clock cycles (we use 2 in the figure) to activate the selected row. Then, the column address is latched under control of the CAS signal. After a delay of one clock cycle, the first set of data bits is placed on the data lines. The SDRAM automatically increments the column address to access the next three sets of bits in the selected row, which are placed on the data lines in the next 3 clock cycles.

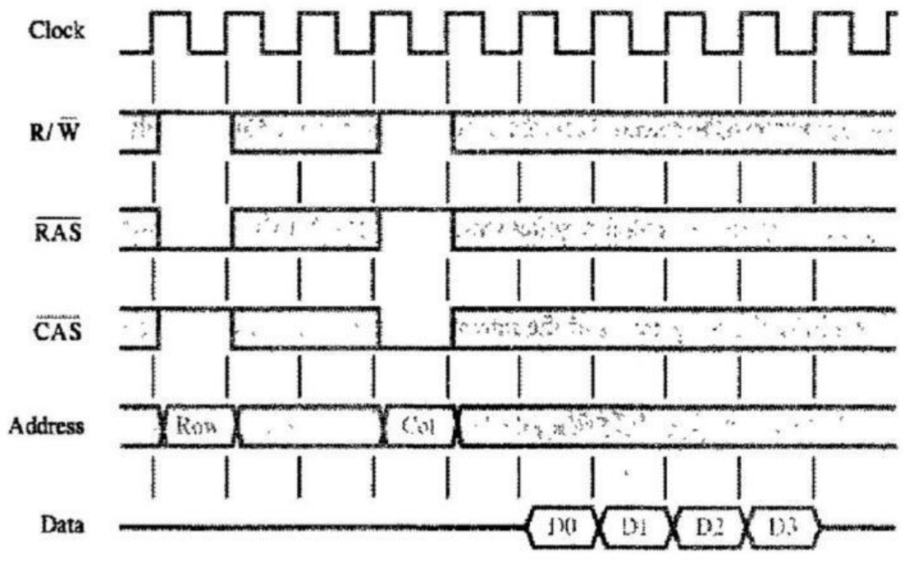


Figure 5.9 Burst read of length 4 in an SDRAM.

SDRAMs have built-in refresh circuitry. A part of this circuitry is a refresh counter, which provides the addresses of the rows that are selected for refreshing.

Commercial SDRAMs can be used with clock speeds above 100 MHz. These chips are designed to meet the requirements of commercially available processors that are used in large volume.

Latency and Bandwidth:

A good indication of the performance of a computer system is given by two parameters: *latency* and *bandwidth*.

The term *memory latency* is used to refer to the amount of time it takes to transfer a word of data to or from the memory. In the case of reading or writing a single word of data, the latency provides a complete indication of memory performance. But, in the case of burst operations that transfer a block of data, the time needed to complete the operation depends also on the rate at which successive words can be transferred and on the size of the block.

In block transfers, the term *latency* is used to denote the time if takes to transfer the first word of data. This time is usually substantially longer than the time needed to transfer each subsequent word of a block.

When transferring blocks of data, since blocks can be variable in size, it is useful to define a performance measure in terms of the number of bits or bytes that can be transferred in one second. This measure is often referred to as the *memory bandwidth*. This measure is often referred to as the memory bandwidth. The bandwidth of a memory unit (consisting of one or more memory chips) depends on the speed of access to the stored data and on the number of bits that can be accessed in parallel. The effective bandwidth in a computer system also depends on the transfer capability of the links that connect the memory and the *processor*, typically the speed of the bus.

The bandwidth clearly depends on *the speed of access and transmission along a single wire, as well as on the number of bits that can be transferred in parallel, namely the number of wires*.

Thus, the bandwidth is the product of the rate at which data are transferred (and accessed) and the width of the data bus.

Double-Data-Rate SDRAM:

The standard SDRAM performs all actions on the rising edge of the clock signal. A similar SDRAM memory device is available, which accesses the cell array in the same way, but transfers data on both edges of the clock. The latency of these devices is the same as for standard SDRAMs. But, since they transfer data on both edges of the clock, their bandwidth is essentially doubled for long burst transfers. Such devices are known as *double-data-rate SDRAMs (DDR SDRAMs)*.

To make it possible to access the data at a high enough rate, the cell array is organized in two banks. Each bank can be accessed separately. Consecutive words of a given block are stored in different banks. Such interleaving of words allows simultaneous access to two words that are transferred on successive edges of the clock.

DDR SDRAMs and standard SDRAMs are most efficiently used in applications where block transfers are prevalent.

STRUCTURE OF LARGER MEMORIES

Static Memory Systems:

Consider a memory consisting of 2M (2,097,152) words of 32 bits each. Figure 5.10 shows how we can implement this memory using 512K x 8 static memory chips. Each column in the figure consists of four chips, which implement one byte position. Four of these sets provide the required 2M x 32 memory. Each chip has a control input called Chip Select. When this input is set to 1, it enables the chip to accept data from or to place date on its data lines. The data output for each chip is of the three-state type. Only the selected chip places data on the data output line, while all other outputs are in the high-impedance state. Twenty one address bits are needed to select a 32-bit word in this memory. The high-order 2 bits of the address are decoded to determine which of the four Chip Select control signals should be activated and the remaining 19 address bits are used to access specific byte locations inside each chip of the selected row. The R/W inputs of all chips are tied together to provide a common Read/Write control.

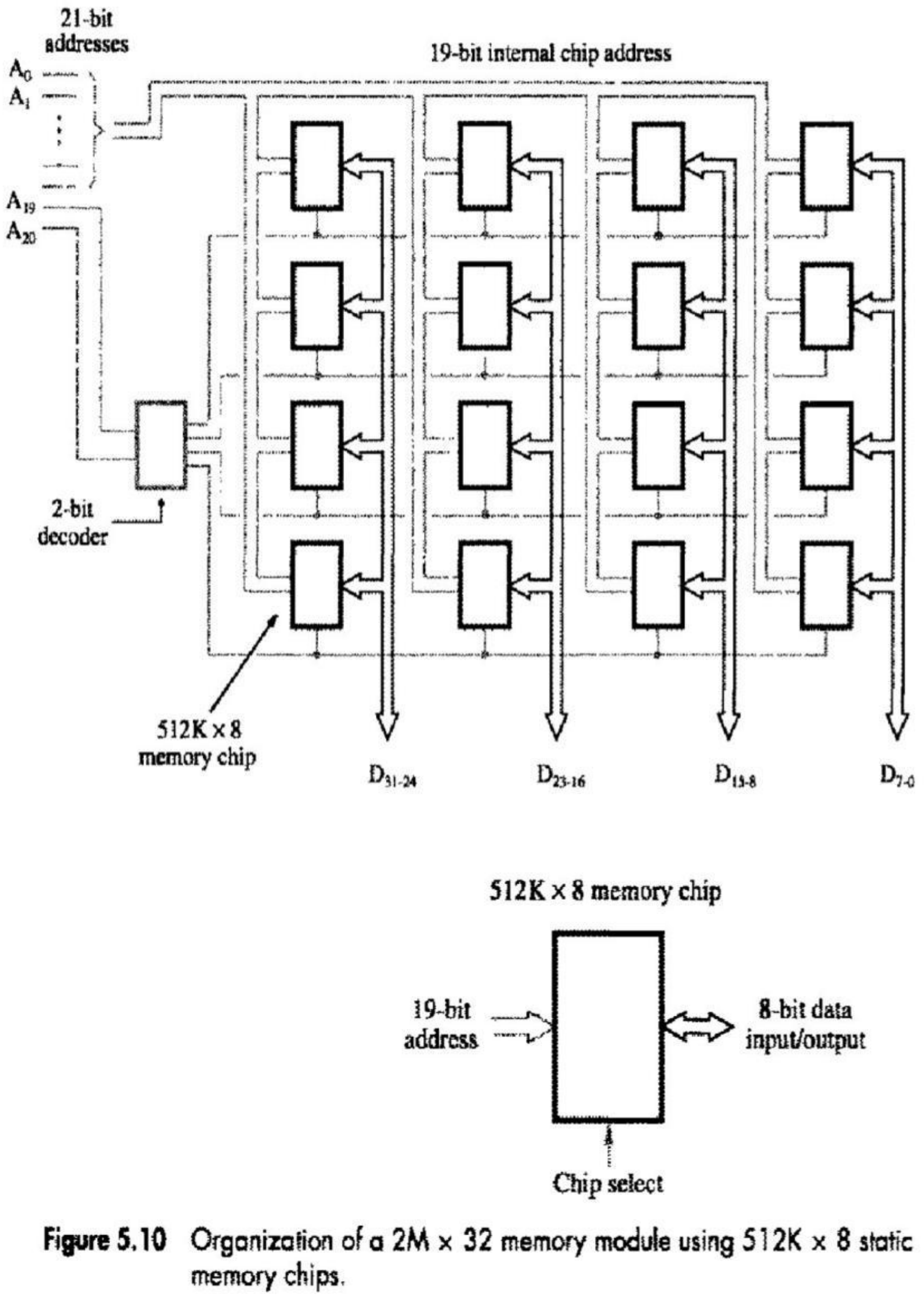


Figure 5.10 Organization of a 2M × 32 memory module using 512K × 8 static memory chips.

Dynamic Memory Systems:

Physical implementation of large dynamic memory systems is often done more conveniently in the form of memory modules.

A large memory leads to better performance because more of the programs and data used in processing can be held in the memory, thus reducing the frequency of accessing the information in secondary storage. However, if a large memory is built by placing DRAM chips directly on the main system printed-circuit board that contains the processor, often referred to as a motherboard, it will occupy an unacceptably large amount of space on the board. These packaging considerations have led to the development of larger memory units known as SIMMs (Single In-line Memory Modules) and DIMMs (Dual In-line Memory Modules). SIMMs and DIMMs of different sizes are designed to use the same size socket. Such modules occupy a smaller amount of space on a motherboard, and they allow easy expansion by replacement if a larger module uses the same socket as the smaller one.

MEMORY SYSTEM CONSIDERS TIONS:

The choice of a RAM chip for a given application depends on several factors. Foremost among these factors are the cost, speed, power dissipation, and size of the chip.

Static RAMs are generally used only when very fast operation is the primary requirement. They are used mostly in cache memories.

Dynamic RAMs are the predominant choice for implementing computer main memories. The high densities achievable in these chips make large memories economically feasible.

Memory Controller:

To reduce the number of pins, the dynamic memory chips use multiplexed address inputs.

The address is divided into two parts.

The *high-order address* bits, which select a row in the cell array, are provided first and latched into the memory chip under control of the RAS signal.

Then, the *low-order address* bits, which select a column, are provided on the same address pins and latched using the CAS signal.

A typical processor issues all bits of an address at the same time. The required multiplexing of address bits is usually performed by a *memory controller* circuit, which is interposed between the processor and the dynamic memory.

The controller accepts a complete address and the R/W signal from the processor, under control of a Request signal which indicates that a memory access operation is needed. The controller then forwards the row and column portions of the address to the memory and generates the RAS and CAS signals. Thus, the controller provides the RAS-CAS timing, in addition to its address multiplexing function. It also sends the $R/\overline{W}$ and CS signals to the memory.

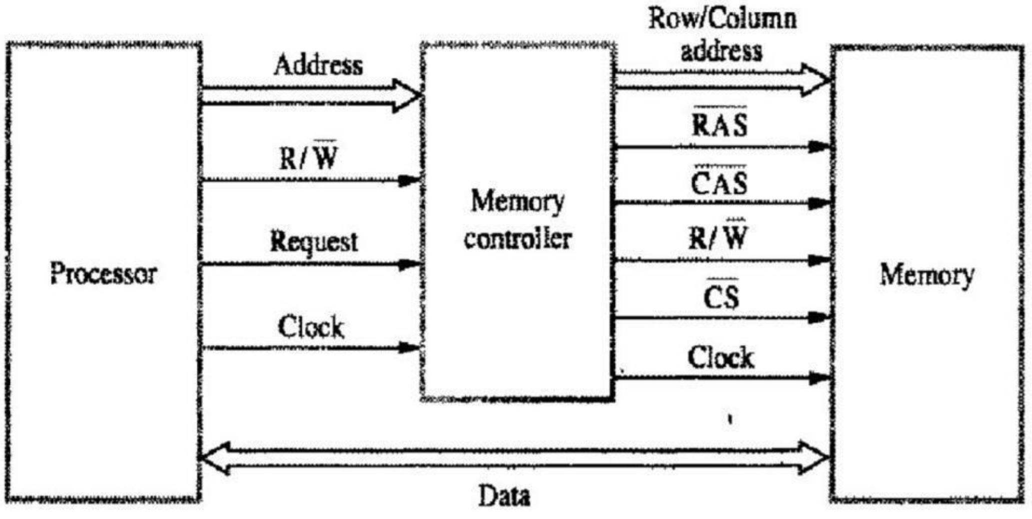


Figure 5.11 Use of a memory controller.

The CS signal is usually active low; hence it is shown as $\overline{CS}$. Data lines are connected directly between the processor and the memory.

When used with DRAM chips, which do not have self-refreshing capability, the memory controller has to provide all the information needed to control the refreshing process. It contains a refresh counter that provides successive row addresses. Its function is to cause the refreshing of all rows to be done within the period specified for a particular device.

Refresh Overhead:

All dynamic memories have to be refreshed. In older DRAMs, a typical period for refreshing all rows was 16ms. In typical SDRAMs, a typical period is 64ms.

Consider an SDRAM whose cells are arranged in 8K (=8192) rows. Suppose that it takes four clock cycles to access (read) each row. Then, it takes 8192 x 4 = 32,768 cycles to refresh all rows. At a clock rate of 133 MHz, the time needed to refresh all rows is 32,768/ (133 x 10^6) = 246 x 10^{-6} seconds. Thus, the refreshing process occupies 0.246ms in each 64-ms time interval. Therefore, the refresh overhead is 0.246/64 = 0.0038, which is less than 0.4 percent of the total time available for accessing the memory.

RAMBUS MEMORY:

The performance of a dynamic memory is characterized by its latency and bandwidth. The only way to increase the amount of data that can be transferred on a speed-limited bus is to increase the width of the bus by providing more data lines, thus widening the bus.

A very wide bus is expensive and requires a lot of space on a motherboard. An alternative approach is to implement a narrow bus that is much faster. This approach was used by Rambus Inc. to develop a proprietary design known as *Rambus*. The *key feature* of Rambus technology is a fast signaling method used to transfer information between chips.

Differential signaling and high transmission rates require special techniques for the design of wire connections that serve as communication links. Rambus provides a complete specification for the design of such communication links, called the *Rambus channel*.

Rambus requires specially designed memory chips. These chips use cell arrays based on the standard DRAM technology. Multiple banks of ceil arrays are used to access more than one word at a time, Circuitry needed to interface to the Rambus Channel is included on the chip. Such chips are known as *Rambus DRAMs (RDRAMs)*.

The original specification of Rambus provided for a channel consisting of 9 data lines and a number of control and power supply lines. Eight of the data lines are intended for transferring a byte of data. The ninth data line can be used for purposes such as parity checking. Subsequent specifications allow for additional channels. A two-channel Rambus, also known as Direct RDRAM, has 18 data lines intended to transfer two bytes of data at a time. There are no separate address lines.

Communication between the processor, or some other device that can serve as a master, and RDRAM modules, which serve as slaves, is carried out by means of packets transmitted on the data lines. There are three types of packets: *request, acknowledge, and data*. A request packet issued by the master indicates the type of operation that is to be performed. It contains the address of the desired memory location and includes an 8-bit count that specifies the number of bytes involved in the transfer. The operation types include memory reads and writes, as well as reading and writing of various control registers in the RDRAM chips. *When the master issues a request packet, the addressed slave responds by returning a positive acknowledgement packet if it can immediately satisfy the request. Otherwise, the slave indicates that itis "busy" by returning a negative acknowledgement packet, in which case the master will try again*.

READ-ONLY MEMORIES:

The contents of non-volatile memory can be read as if they were SRAM or DRAM memories. But, a special writing process is needed to place the information into this memory. Since its normal operation involves only reading of stored data, a memory of this type is called *read-only memory (ROM)*.

ROM

A logic value 0 is stored in the cell if the transistor is connected to ground at point P; otherwise, a 1 is stored. The bit line is connected through a resistor to the power supply. To read the state of the cell, the word line is activated. Thus, the transistor switch is closed and the voltage on the bit line drops to near zero if there is a connection between the transistor and ground. If there is no connection to ground, the bit line remains at the high voltage, indicating a 1. A sense circuit at the end of the bit line generates the proper output value. Data are written into a ROM when it is manufactured.

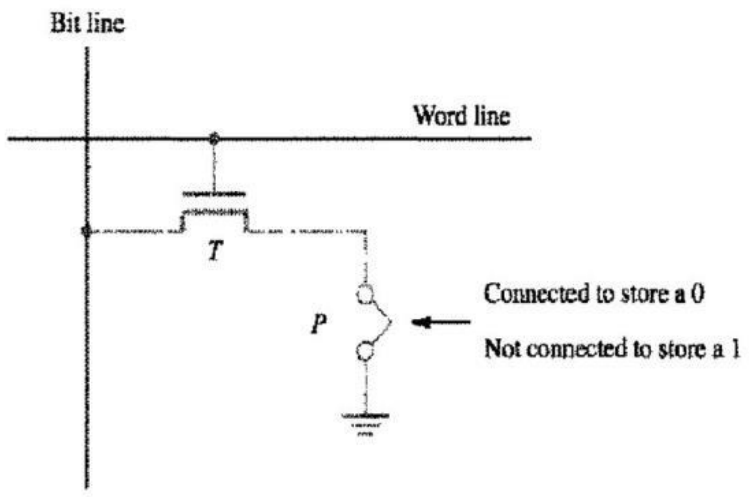


Figure 5.12 A ROM cell.

PROM:

Some ROM designs allow the data to be loaded by the user, thus providing a programmable ROM (PROM). Programmability is achieved by inserting a fuse at point P in Figure 5.12. Before it is programmed, the memory contains all 0s. The user can insert 1s at the required locations by burning out the fuses at these locations using high-current pulses. Of course, this process is irreversible.

PROMs provide flexibility and convenience. PROMs provide a faster and considerably less expensive approach because they can be programmed directly by the user.

EPROM:

Another type of ROM chip allows the stored data to be erased and new data to be loaded. Such an *erasable, reprogrammable* ROM is usually called an *EPROM.* Since EPROMs are capable of retaining stored information for a long time, they can be used in place of ROMs while software is being developed.

The important advantage of EPROM chips is that their contents can be erased and reprogrammed. Erasure requires dissipating the charges trapped in the transistors of memory cells; this can be done by exposing the chip to ultraviolet light. For this reason, EPROM chips are mounted in packages that have transparent windows.

EEPROM:

A significant disadvantage of EPROMs is that a chip must be physically removed from the circuit for reprogramming and that its entire contents are erased by the ultraviolet light. It is possible to implement another version of erasable PROMs that can be both programmed and erased electrically. Such chips, called EEPROMs, do not have to be removed for erasure. Moreover, it is possible to erase the cell contents selectively. The only disadvantage of EEPROMs is that different voltages are needed for erasing, writing, and reading the stored data.

Flash Memory:

In EEPROM it is possible to read and write the contents of a single cell. In a flash device it is possible to read the contents of a single cell, but it is only possible to write an entire block of cells. Prior to writing, the previous contents of the block are erased. Flash devices have greater density, which leads to higher capacity and a lower cost per bit. They require a single power supply voltage, and consume less power in their operation.

Flash memory consumes low power.

Applications:

Typical applications include hand-held computers, cell phones, digital cameras, and MP3 music players. In hand-held computers and cell phones, flash memory holds the software needed to operate the equipment, thus obviating the need for a disk drive. In digital cameras, flash memory is used to store picture image data. In MP3 players, flash memory stores the data that represent sound. Cell phones, digital cameras, and MP3 players are good examples of embedded systems,

There are two popular choices for the implementation of larger modules: flash cards and flash drives.

Flash Cards:

One way of constructing a larger module is to mount flash chips on a small card. Such flash cards have a standard interface that makes them usable in a variety of products. A card is simply plugged into a conveniently accessible slot.

Flash Drives:

The storage capacity of flash drives is significantly lower. Currently, the capacity of flash drives is less than one gigabyte. In contrast, hard disks can store many gigabytes.

The fact that flash drives are solid state electronic devices that have no movable parts provides some important advantages. They have shorter seek and access times, which results in faster response. They have lower power consumption.

The disadvantages of flash drives vis-a-vis hard disk drives are their smaller capacity and higher cost per bit. Disks provide an extremely low cost per bit. Another disadvantage is that the flash memory will deteriorate after it has been written a number of times.

SPEED, SIZE, AND COST

A very fast memory can be implemented if *SRAM* chips are used. But these chips are expensive because their basic cells have six transistors, which preclude packing a very large number of cells onto a single chip. The alternative is to use *Dynamic RAM chips*, which have much simpler basic cells and thus are much less expensive, but such memories are significantly slower.

Secondary storage, mainly magnetic disks, is used to implement large memory spaces. Very large disks are available at a reasonable price, and they are used extensively in computer systems. However, they are much slower than the semiconductor memory units.

"A huge amount of cost-effective storage can be provided by magnetic disks. A large, yet affordable, main memory can be built with dynamic RAM technology". This leaves SRAMs to be used in smaller units where speed is of the essence, such as in cache memories.

The entire computer memory can be viewed as the hierarchy. The fastest access is to data held in processor registers. Therefore, if we consider the registers to be part of the memory hierarchy, then the processor registers are at the top in terms of the speed of access.

At the next level of the hierarchy is a relatively small amount of memory that can be implemented directly on the processor chip. This memory, called a processor cache, holds copies of instructions and data stored in a much larger memory that is provided externally.

There are often two levels of caches.

A primary cache is always located on the processor chip. This cache is small because it competes for space on the processor chip, which must implement many other functions. The *primary cache* is referred to as *level 1 (L1) cache*. A larger, *secondary cache* is placed between the primary cache and the rest of the memory. It is referred to as *level 2 (L2)* cache.

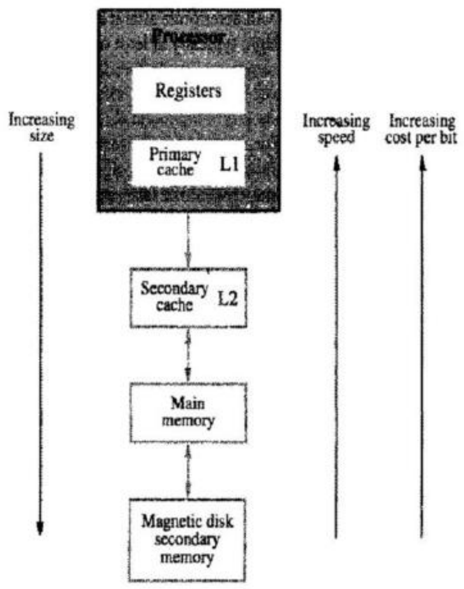


Figure 5.13 Memory hierarchy.

The next level in the hierarchy is called the main memory. This rather large memory is implemented using dynamic memory components, typically in the form of SIMMs, DIMMs, or RIMMs. The main memory is much larger but significantly slower than the cache memory. In a typical computer, the access time for the main memory is about ten times longer than the access time for the L1 cache.

Disk devices provide a huge amount of inexpensive storage. They are very slow compared to the semiconductor devices used to implement the main memory.

CACHE MEMORIES

Fast *cache memory* essentially makes the main memory appear to the processor to be faster than it really is.

The effectiveness of the cache mechanism is based on a property of computer programs called locality of reference.

Many instructions in localized areas of the program are executed repeatedly during some time period, and the remainder of the program is accessed relatively infrequently. This is referred to as locality of reference. It manifests itself in *two* ways: *temporal* and *spatial*. The first means that a recently executed instruction is likely to be executed again very soon. The spatial aspect means that instructions in close proximity to a recently executed instruction are also likely to be executed soon.

The *temporal aspect* of the locality of reference suggests that whenever an information item (instruction or data) is first needed, this item should be brought into the cache where it will hopefully remain until it is needed again. The *spatial aspect* suggests that instead of fetching just one item from the main memory to the cache, it is useful to fetch several items that reside at adjacent addresses as well. The term *block* to refer to a set of contiguous address locations of some size, another term that is often used to refer to a cache block is *cache line*.

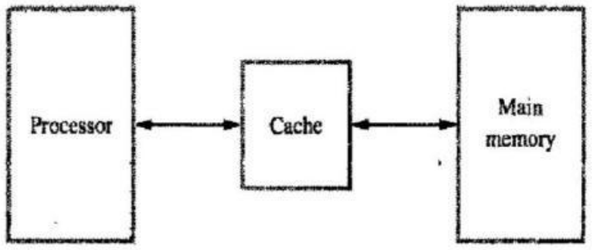


Figure 5.14 Use of a cache memory.

In above figure, when a Read request is received from the processor, the contents of a block of memory words containing the location specified are transferred into the cache one word at a time. Subsequently, when the program references any of the locations in this block, the desired contents are read directly from the cache. Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function. When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the *replacement algorithm*.

The processor does not need to know explicitly about the existence of the cache. It simply issues Read and Write requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache. If it does, the Read or Write operation is performed on the appropriate cache location. In this case, a *read* or *write hit* is said to have occurred.

- In a *Read operation*, the main memory is not involved.
- For a *Write operation*, the system can proceed in two ways. In the first technique, called the *write-through protocol*, the cache location and the main memory location are updated simultaneously. The *second technique* is to update only the cache location and to mark it as updated with an associated flag bit, often called the *dirty or modified bit*.

The main memory location of the word is updated later, when the block containing this marked word is to be removed from the cache to make room for a new block. This technique is known as the *write-back*, or *copy-back, protocol*. The write-through protocol is simpler, but it results in unnecessary Write operations in the main memory when a given cache word is updated several times during its cache residency. Note that the write-back protocol may also result in unnecessary Write operations because when a cache block is written back to the memory all words of the block are written back, even if only a single word has been changed while the block was in the cache.

When the addressed word in a Read operation is not in the cache, a read miss occurs. The block of words that contains the requested word is copied from the main memory into the cache. After the entire block is loaded into the cache, the particular word requested is forwarded to the processor. Alternatively, this word may be sent to the processor as soon as it is read from the main memory. The latter approach, which is called load-through, or early restart, reduces the processor's waiting period somewhat, but at the expense of more complex circuitry.

During a Write operation, if the addressed word is not in the cache, a write miss occurs. Then, if the write-through protocol is used, the information is written directly into the main memory. In the case of the write-back protocol, the block containing the addressed word is first brought into the cache, and then the desired word in the cache is overwritten with the new information.

MAPPING FUNCTIONS:

Consider a cache consisting of 128 blocks of 16 words each, for a total of 2048 (2K) words, and assume that the main memory is addressable by 2 16-bit address. The main memory has 64K words, which we will view as 4K blocks of 16 words each. For simplicity, we will assume that consecutive addresses refer to consecutive words.

Direct Mapping:

The simplest way to determine cache locations in which to store memory blocks is the *direct-mapping technique*. In this technique, block j of the main memory maps onto block j modulo 128 of the cache. Thus, whenever one of the main memory blocks 0, 128, 256,... is loaded in the cache, it is stored in cache block 0. Blocks 1, 129, 257,... are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full. For example, instructions of a program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed, both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block. In this case, the replacement algorithm is trivial.

Placement of a block in the cache is determined from the memory address. The memory address can be divided into three fields, as shown in Figure 5.15,

- The *low-order 4 bits* select one of 16 words in a block. When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored.

- The high-order 5 bits of the memory address of the block are stored in 5 tag bits associated with its location in the cache. They identify which of the 32 blocks that are mapped into this cache position are currently resident in the cache.
- As execution proceeds, the 7-bit cache block field of each address generated by the processor points to a particular block location in the cache.

The high-order 5 bits of the address are compared with the tag bits associated with that cache location. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache.

The direct-mapping technique is easy to implement, but it is not very flexible.

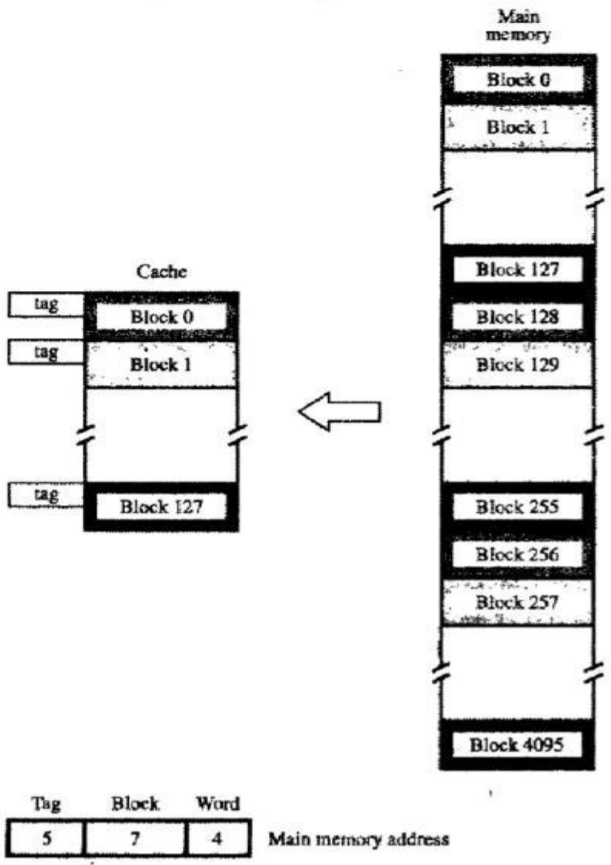


Figure 5.15 Direct-mapped cache.

Associative Mapping:

In this mapping method, a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the *associative-mapping technique*. It gives complete freedom in choosing the cache location in which to place the memory block. Thus, the space in the cache can be used more efficiently. A new block that has to be brought into the cache has to replace (eject) an existing block only if the cache is full. The cost of an associative cache is higher than the cost of a direct-mapped cache because of the need to search all 128 tag patterns to determine whether a given block is in the cache. A search of this kind is called an *associative search*. For performance reasons, the tags must be searched in parallel.

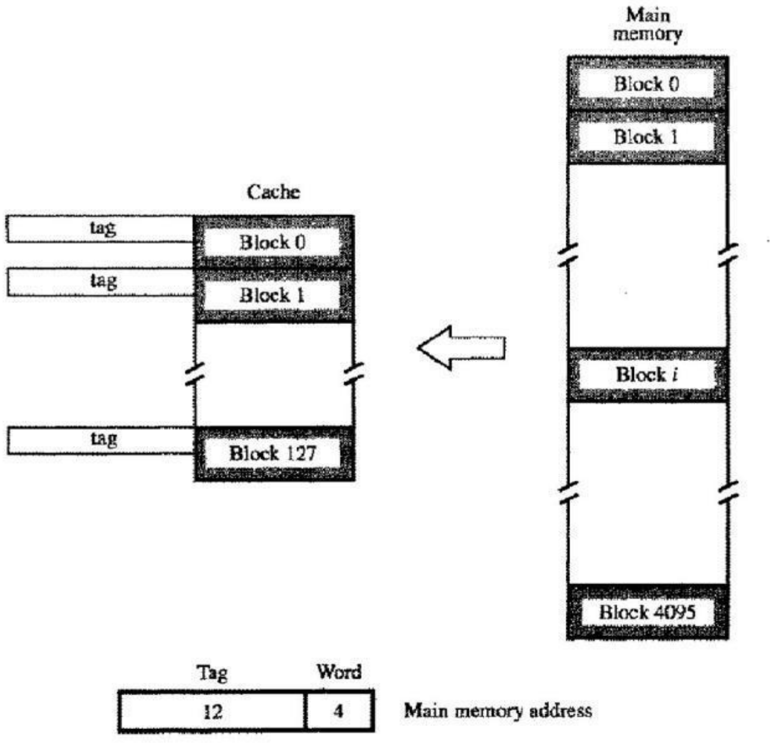


Figure 5.16 Associative-mapped cache.

Set-Associative Mapping:

Blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the *contention problem* of the direct method is eased by having a few choices for block placement. At the same time, the *hardware cost is reduced* by decreasing the size of the associative search. An example of this set-associative-mapping technique is shown in Figure 5.17 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128,...., 4032 map into cache set 0, and they can occupy either of the two block positions within this set. Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

The number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer.

A cache that has k blocks per set is referred to as a *k-way set-associative cache*.

The control bit, *valid bit* indicates whether the block contains valid data.

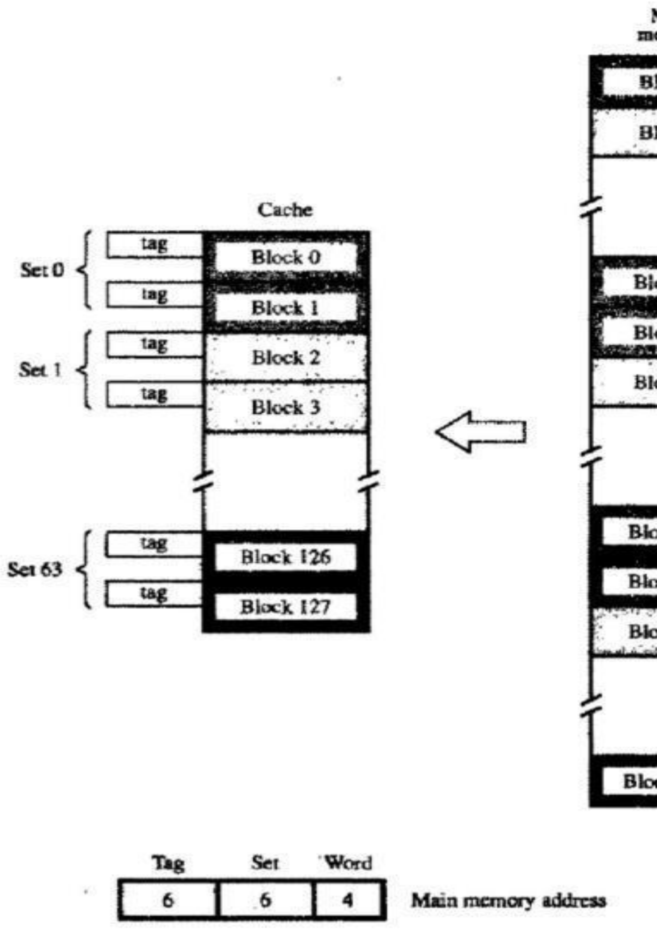


Figure 5.17 Set-associative-mapped cache with two blocks per set.

REPLACEMENT ALGORITHMS

In a direct-mapped cache, the position of each block is predetermined; hence, no replacement strategy exists. In associative and set-associative caches there exists some flexibility. When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite. When a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the least recently used (LRU) block, and the technique is called the LRU replacement algorithm.

To use the LRU algorithm, the cache controller must track references to all blocks as computation proceeds. Suppose it is required to track the LRU block of a four-block set in a set-associative cache. A 2-bit counter can be used for each block. When a hit occurs, the counter of the block that is referenced is set to 0. Counters with values originally lower than the referenced one are incremented by one, and all others remain unchanged. When a miss occurs and the set is not full, the counter associated with the new block loaded from the main memory is set to 0, and the values of all other counters are increased by one. When a miss occurs and the set is fall, the block with the counter value 3 is removed, the new block is put in its place, and its counter is set to 0. The other three block counters are incremented by one, it can be easily verified that the counter values of occupied blocks are always distinct.

The LRU algorithm has been used extensively. Performance of the LRU algorithm can be improved by introducing a small amount of randomness in deciding which block to replace.

PERFORMANCE CONSIDERATIONS

Two key factors in the commercial success of a computer are performance and cost; the best possible performance at the lowest cost is the objective. The challenge in considering design alternatives is to improve the performance without increasing the cost. A common measure of success is the price/performance ratio.

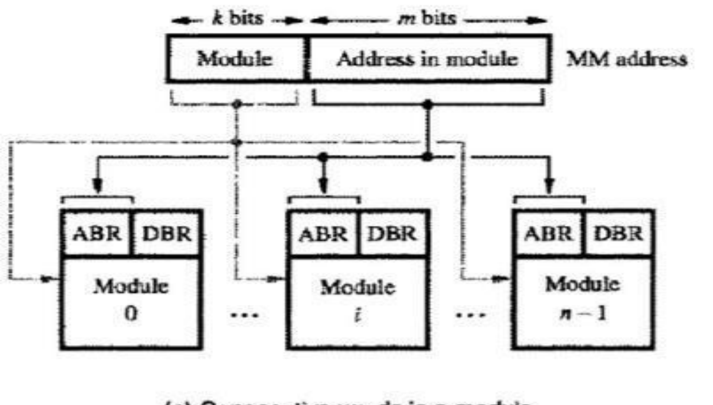
Performance depends on how fast machine instructions can be brought into the processor for execution and how fast they can be executed. The main purpose of the memory hierarchy is to create a memory that the processor sees as having a short access time and 2 large capacities. Each level of the hierarchy plays an important role. It is beneficial if transfers to and from the faster units can be done at a rate equal to that of the faster unit. This is not possible if both the slow and the fast units are accessed in the same manner, but it can be achieved when parallelism is used in the organization of the slower unit. An effective way to introduce parallelism is to use an interleaved organization.

INTERLEAVING:

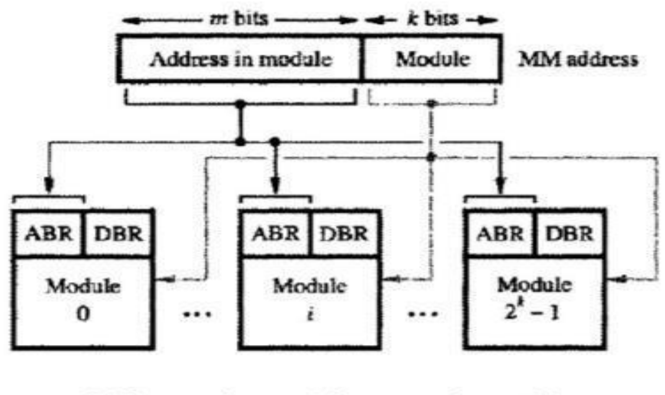
If the main memory of a computer is structured as a collection of physically separate modules, each with its own address buffer register (ABR) and data buffer register (DBR), memory access operations may proceed in more than one module at the same time. Thus, the aggregate rate of transmission of words to and from the main memory system can be increased.

How individual addresses are distributed over the modules is critical in determining the average number of modules that can be kept busy as computations proceed. Two methods of address layout are indicated in Figure 5.25. In the *first case*, the memory address generated by the processor is decoded as shown in Figure 5.25a. The high-order & bits name one of n modules, and the low-order m bits name a particular word in that module. When consecutive locations are accessed, as happens when a block of data is transferred to a cache, only one module is involved. At the same time, however, devices with direct memory access (DMA) ability may be accessing information in other memory modules.

The second and more effective way to address the modules is shown in Figure 5.25b. It is called memory interleaving. The low-order & bits of the memory address select a module, and the high-order m bits name a location within that module. In this way, consecutive addresses are located in successive modules. Thus, any component of the system that generates requests for access to consecutive memory locations can keep several modules busy at any one time. This results in both faster accesses to a block of data and higher average utilization of the memory system as a whole. To implement the interleaved structure, there must be 2 modules; otherwise, there will be gaps of nonexistent locations in the memory address space.



(a) Consecutive words in a module



(b) Consecutive words in consecutive modules

Figure 5.25 Addressing multiple-module memory systems.

Example 1:

The effect of interleaving is substantial. Consider the time needed to transfer 2 block of data from the main memory to the cache when a read miss occurs. Suppose that a cache with 8-word blocks is used, similar to our examples in Section 5.5, On a read miss, the block that contains the desired word must be copied from the memory into the cache. Assume that the hardware has the following properties. It takes one clock cycle to send an address to the main memory. The memory is built with relatively slow DRAM chips that allow the first word to be accessed in 8 cycles, but subsequent words of the block are accessed in 4 clock cycles per word. (Recall from Section 5.2.3 that, when consecutive locations in a DRAM are read from a given row of cells, the row address is decoded only once. Addresses of consecutive columns of the array are then applied to access the desired words, which takes only half the time per access.) Also, one clock cycle is needed to send one word to the cache.

If a single memory module is used, then the time needed to load the desired block into the cache is

$$1+8+(7*4)+1=38 \text{ cycles}$$

Suppose now that the memory is constructed as four interleaved modules, using the scheme in Figure 5.25b, When the starting address of the block arrives at the memory, all four modules begin accessing the required data, using the high-order bits of the address. After 8 clock cycles, each module has one word of data in its DBR. These words are transferred to the cache, one word at a time, during the next 4 clock cycles. During this time, the next word in each module is accessed. Then it takes another 4 cycles to transfer these words to the cache. Therefore, the total time needed to load the block from the interleaved memory is

$$1+8+4+4=17 \text{ cycles}$$

Thus, interleaving reduces the block transfer time by more than a factor of 2.

HIT RATE AND MISS PENALTY

A successful access to data in a cache is called a hit. The number of hits stated as a fraction of all attempted accesses is called the *hit rate*, and the *miss rate* is the number of misses stated as a fraction of attempted accesses.

High hit rates, well over 0.9, are essential for high-performance computers.

Performance is adversely affected by the actions that must be taken after a miss. The extra time needed to bring the desired information into the cache is called the *miss penalty*. This penalty is ultimately reflected in the time that the processor is stalled because the required instructions or data are not available for execution. In general, the miss penalty is the time needed to bring a block of data from a slower unit in the memory hierarchy to a faster unit. The miss penalty is reduced if efficient mechanisms for transferring data between the various units of the hierarchy are implemented.

Example 2:

Consider now the impact of the cache on the overall performance of the computer. Let h be the hit rate, M the miss penalty, that is, the time to access information in the main memory, and C the time to access information in the cache. The average access time experienced by the processor is

$$t_{ave} = hC+(1-h)M$$

We use the same parameters as in Example 5.1. If the computer has no cache, then, using a fast processor and a typical DRAM main memory, it takes 10 clock cycles for each memory read access. Suppose the computer has a cache that holds 8-word blocks and an interleaved main memory. Then, as we showed in Section 5.6.1, 17 cycles are needed to load a block into the cache. Assume that 30 percent of the instructions in a typical program perform a read or a write operation, which means that there are 130 memory accesses for every 100 instructions executed. Assume that the hit rates in the cache are 0.95 for instructions and 0.9 for data. Let us further assume that the miss penalty is the same for both read and write accesses. Then, a rough estimate of the improvement in performance that results from using the cache can be obtained as follows:

$$\frac{\textit{Time without cache}}{\textit{Time with cache}} = \frac{130 \times 10}{100(0.95 \times 1 + 0.05 \times 17) + 30(0.9 \times 1 + 0.1 \times 17)} = 5.04$$

This result suggests that the computer with the cache performs five times better.

It is also interesting to consider how effective this cache is compared to an ideal cache that has a hit rate of 100 percent (in which case, all memory references take one cycle). Our rough estimate of relative performance for these caches is

$$\frac{100(0.95 \times 1 + 0.05 \times 17) + 30(0.9 \times 1 + 0.1 \times 17)}{130} = 1.98$$

How can the hit rate be improved?

- To make the cache larger, but this entails increased cost
- Another possibility is to increase the block size while keeping the total cache size constant, to take advantage of spatial locality, If all items in a larger block are needed in a computation, then it is better to load these items into the cache as a consequence of a single miss, rather than loading several smaller blocks as a result of several misses. The efficiency of parallel access to blocks in an interleaved memory is the basic reason for this advantage.

The miss penalty increases as the block size increases. Since the performance of a computer is affected positively by increased hit rate and negatively by increased miss penalty, the block sizes that are neither very small nor very large give the best results.

Finally, we note that the miss penalty can be reduced if the load-through approach is used when loading new blocks into the cache. Then, instead of waiting for the completion of the block transfer, the processor can continue as soon as the required word is loaded in the cache.

CACHES ON THE PROCESSOR CHIP:

From the speed point of view, the optimal place for a cache is on the processor chip.

All high-performance processor chips include some form of a cache. Some manufacturers have chosen to implement two separate caches, one for instructions and another for data, as in the 68040, Pentium III, and Pentium 4 processors. Others have implemented a single cache for both instructions and data, as in the ARM710T processor.

A combined cache for instructions and data is likely to have a somewhat better hit rate because it offers greater flexibility in mapping new information into the cache. However, if separate caches are used, it is possible to access both caches at the same time, which leads to increased parallelism and, hence, better performance. The disadvantage of separate caches is that the increased parallelism comes at the expense of more complex circuitry.

In high-performance processors two levels of caches are normally used, The L1 cache(s) is on the processor chip. The L2 cache, which is much larger, may be implemented externally using SRAM chips.

If both L1 and L2 caches are used, the L1 cache should be designed to allow very fast access by the processor because its access time will have a large effect on the clock rate of the processor. A practical way to speed up access to the cache is to access more than one word simultaneously and then let the processor use them one at a time,

The L2 cache can be slower, but it should be much larger to ensure a high hit rate. Its speed is less critical because it only affects the miss penalty of the L1 cache. A workstation computer may include an L1 cache with the capacity of tens of kilobytes and an L2 cache of several megabytes.

Including an L2 cache further reduces the impact of the main memory speed on the performance of a computer. The average access time experienced by the processor in a system with two levels of caches is

$$t_{ave}=h1C1+(1-h1)h2C2+(1-h1)(1-h2)M$$

Where

h1 is the hit rate in the L1 cache.

h2 is the hit rate in the L2 cache.

C1 is the time to access information in the LI cache.

C2 is the time to access information in the L2 cache.

M is the time to access information in the main memory

The number of misses in the L2 cache, given by the term (1-h1) (1-h2), should be low. If both h1 and h2 are in the 90 percent range, then the number of misses will be less than 1 percent of the processor’s memory accesses. Thus, the miss penalty M will be less critical from a performance point of view.

OTHER ENHANCEMENTS

Write Buffer:

To improve performance, a write buffer can be included for temporary storage of write requests. The processor places each write request into this buffer and continues execution of the next instruction. The write requests stored in the write buffer are sent to the main memory whenever the memory is not responding to read requests. Note that it is important that the read requests be serviced immediately because the processor usually cannot proceed without the data that are to be read from the memory. Hence, these requests are given priority over write requests.

The write buffer may hold a number of write requests. Thus, it is possible that a subsequent read request may refer to data that are still in the write buffer. To ensure correct operation, the addresses of data to be read from the memory are compared with the addresses of the data in the write buffer. In case of a match, the data in the write buffer are used.

A different situation occurs with the write-back protocol. In this case, the write operations are simply performed on the corresponding word in the cache. But consider what happens when a new block of data is to be brought into the cache as a result of a read miss, which replaces an existing block that has some dirty data. The dirty block has to be written into the main memory. If the required write-back is performed first, then the processor will have to wait longer for the new block to be read into the cache. It is more prudent to read the new block first. This can be arranged by providing a fast write buffer for temporary storage of the dirty block that is ejected from the cache while the new block is being read. Afterward, the contents of the buffer are written into the main memory. Thus, the write buffer also works well for the write-back protocol.

Prefetching:

To avoid stalling the processor, it is possible to prefetch the data into the cache before they are needed; the simplest way to do this is through software. A special prefetch instruction may be provided in the instruction set of the processor. Executing this instruction causes the addressed data to be loaded into the cache, as in the case of a read miss. A prefetch instruction is inserted in a program to cause the data to be loaded in the cache by the time they are needed in the program. The hope is that prefetching will take place while the processor is busy executing instructions that do not result in a read miss, thus allowing accesses to the main memory to be overlapped with computation in the processor.

Prefetch instructions can be inserted into a program either by the programmer or by the compiler. Software prefetching entails a certain overhead because inclusion of prefetch instructions increases the length of programs.

Prefetching can also be done through hardware. This involves adding circuitry that attempts to discover a pattern in memory references and then prefetches data according to this pattern.

Intel’s Pentium 4 processor has facilities for prefetching information into its caches using both software and hardware approaches.

Lockup-Free Cache:

If the action of prefetching stops other accesses to the cache until the prefetch is completed. A cache of this type is said to be locked while it services a miss.

A cache that can support multiple outstanding misses is called lockup-free. Since it can service only one miss at a time, it must include circuitry that keeps track of all outstanding misses.

VIRTUAL MEMORIES

Techniques that automatically move program and data blocks into the physical main memory when they are required for execution are called *virtual-memory techniques*. The binary addresses that the processor issues for either instructions or data are called *virtual or logical addresses*. These addresses are translated into physical addresses by a combination of hardware and software components. If a virtual address refers to a part of the program or data space that is currently in the physical memory, then the contents of the appropriate location in the main memory are accessed immediately. If the referenced address is not in the main memory, its contents must be brought into a suitable location in the memory before they can be used.

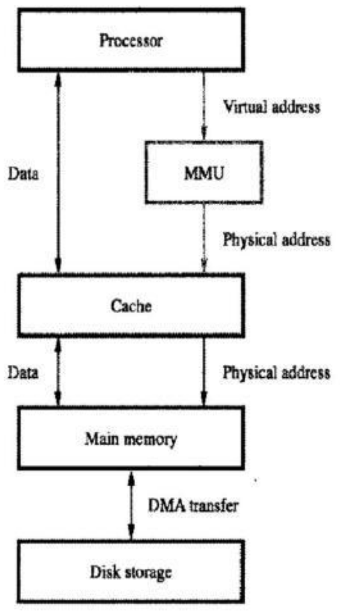


Figure 5.26 Virtual memory organization.

A special hardware unit, called the Memory Management Unit (MMU), translates virtual addresses into physical addresses. When the desired data (or instructions) are in the main memory, these data are fetched. If the data are not in the main memory, the MMU causes the operating system to bring the data into the memory from the disk. Transfer of data between the disk and the main memory is performed using the DMA scheme.

ADDRESS TRANSLATION:

A simple method for translating virtual addresses into physical addresses is to assume that all programs and data are composed of fixed-length units called pages, each of which consists of a block of words that occupy contiguous locations in the main memory. Pages commonly range from 2K to 16K bytes in length.

Pages should not be too small, because the access time of a magnetic disk is much longer (several milliseconds) than the access time of the main memory. The reason for this is that it takes a considerable amount of time to locate the data on the disk, but once located, the data can be transferred at a rate of several megabytes per second. On the other hand, if pages are too large it is possible that a substantial portion of a page may not be used, yet this unnecessary data will occupy valuable space in the main memory.

The cache bridges the speed pap between the processor and the main memory and is implemented in hardware. The virtual-memory mechanism bridges the size and speed gaps between the main memory and secondary storage and is usually implemented in part by software techniques.

A virtual-memory address translation method based on the concept of fixed-length pages is shown schematically in Figure 5.27, Each virtual address generated by the processor, whether it is for an: instruction fetch or an operand fetch/store operation, is interpreted as a *virtual page number* (high-order bits) followed by an *offset* (low-order bits) that specifies the location of a particular byte (or word) within a page. Information about the main memory location of each page is kept in a *page table*. This information includes the main memory address where the page is stored and the current status of the page. An area in the main memory that can hold one page is called a *page frame*. The starting address of the page table is kept in a page table base register. By adding the virtual page number to the contents of this register, the address of the corresponding entry in the page table is obtained. The contents of this location give the starting address of the page if that page currently resides in the main memory.

Each entry in the page table also includes some control bits that describe the status of the page while it is in the main memory. One bit indicates the validity of the page, that is, whether the page is actually loaded in the main memory. Another bit indicates whether the page has been modified during its residency in the memory.

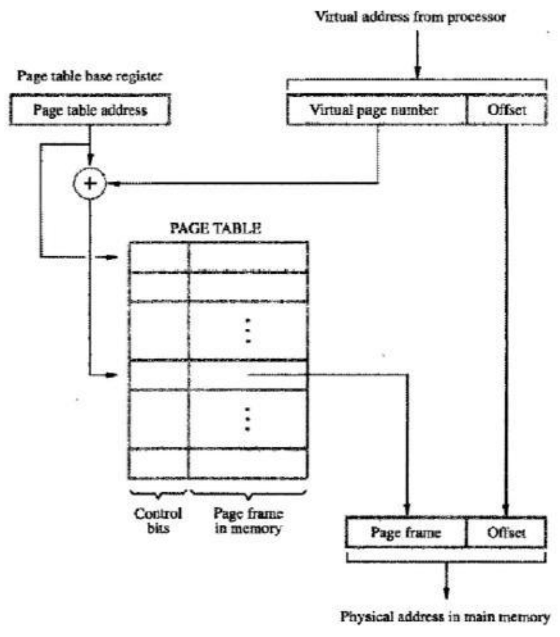


Figure 5.27 Virtual-memory address translation.

The page table information is used by the MMU for every read and write access, so ideally, the page table should be situated within the MMU. A copy of a small portion of the page table can be accommodated within the MMU. This portion consists of the page table entries that correspond to the most recently accessed pages. A small cache, usually called the Translation Lookaside Buffer (TLB) is incorporated into the MMU for this purpose.

When the operating system changes the contents of page tables, it must simultaneously invalidate the corresponding entries in the TLB. When an entry is invalidated, the TLB will acquire the new information as part of the MMU’s normal response to access misses.

Address translation proceeds as follows. Given a virtual address, the MMU looks in the TLB for the referenced page. If the page table entry for this page is found in the TLB, the physical address is obtained immediately. If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated.

When a program generates an access request to a page that is not in the main memory, a page fault is said to have occurred. The whole page must be brought from the disk into the memory before access can proceed. A page fault occurs when some instruction accesses a memory operand that is not in the main memory, resulting in an interruption before the execution of this instruction is completed. Hence, when the task resumes, either the execution of the interrupted instruction must continue from the point of interruption, or the instruction must be restarted.

If a new page is brought from the disk when the main memory is full, it must replace one of the resident pages by using replacement algorithms.

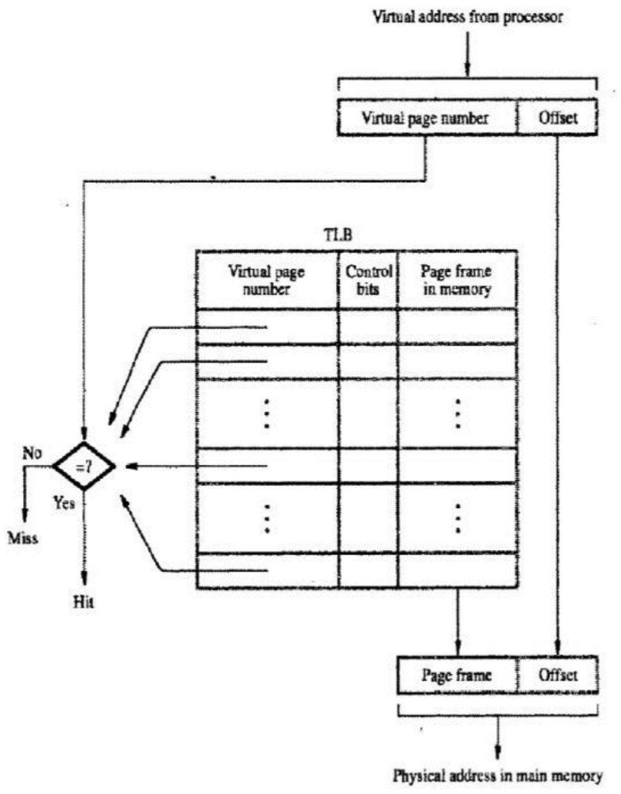


Figure 5.28 Use of an associative-mapped TLB.

A modified page has to be written back to the disk before it is removed from the main memory

MEMORY MANAGEMENT REQUIREMENTS:

Management routines are part of the operating system of the computer. It is convenient to assemble the operating system routines into a virtual address space, called the *system space*, which is separate from the virtual space in which user application programs reside is called the *user space*. By changing the contents of this register, the operating system can switch from one space to another.

In any computer system in which independent user programs coexist in the main memory, the notion of *protection* must be addressed, No program should be allowed to destroy either the data or instructions of other programs in the memory.

Such protection can be provided in several ways:

The processor has two states, the *supervisor state* and the *user state*. As the names suggest, the processor is usually placed in the supervisor state when operating system routines are being executed and in the user state to execute user programs. In the user state, some machine instructions cannot be executed. These privileged instructions, which include such operations as modifying the page table base register, can only be executed while the processor is in the supervisor state. Hence, a user program is prevented from accessing the page tables of other user spaces or of the system space.

SECONDARY STORAGE

MAGNETIC HARD DISK:

The storage medium in a magnetic-disk system consists of one or more disks mounted on a common spindle. A thin magnetic film is deposited on each disk, usually on both sides; the disks are placed in a rotary drive so that the magnetized surfaces move in close proximity to read/write heads, as shown in Figure 5.29a, the disks rotate at a uniform speed. Each head consists of a magnetic yoke and a magnetizing coil, as indicated in Figure 5.29b.

Digital information can be stored on the magnetic film by applying current pulses of suitable polarity to the magnetizing coil. This causes the magnetization of the film in the area immediately underneath the head to switch to a direction parallel to the applied field. The same head can be used for reading the stored information. Only changes in the magnetic field under the head can be sensed during the Read operation. Therefore, if the binary states 0 and 1 are represented by two opposite states of magnetization, a voltage is induced in the head only at 0-to-1 and at 1-to-0 transitions in the bit stream. A long string of 0s or 1s causes an induced voltage only at the beginning and end of the string. To determine the number of consecutive 0s or 1s stored, a clock must provide information for synchronization.

In phase encoding or Manchester encoding changes in magnetization occur for each data bit, as shown in the figure. The drawback of Manchester encoding is its *poor bit-storage density*. We use the Manchester encoding example to illustrate how a *self-clocking* scheme may be implemented, because it is easy to understand.

Read/write heads must be maintained at a very small distance from the moving disk surfaces in order to achieve high bit densities and reliable read/write operations. When the disks are moving at their steady rate, air pressure develops between the disk surface and the head and forces the head away from the surface.

In most modern disk units, the disks and the read/write heads are placed in a sealed, air-filtered enclosure. This approach is known as Winchester technology.

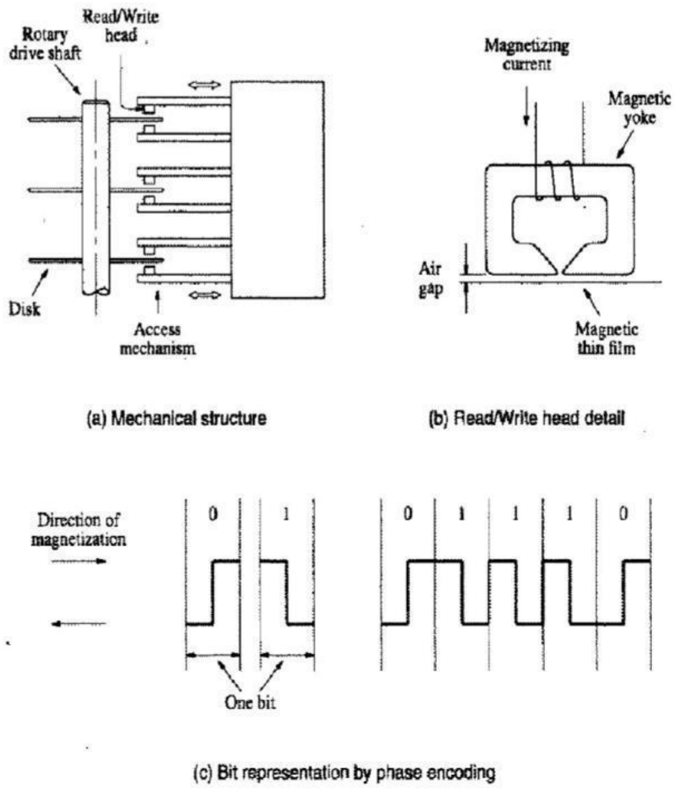


Figure 5.29 Magnetic disk principles.

Organization and Accessing of Data on a Disk:

The organization of data on a disk is illustrated in Figure 5.30. Each surface is divided into concentric *tracks*, and each track is divided into *sectors*. The set of corresponding tracks on all surfaces of a stack of disks forms a *logical cylinder*. The data on all tracks of a cylinder can be accessed without moving the read/write heads. The data are accessed by specifying the *surface number*, the *track number*, and the *sector number*. The Read and Write operations start at sector boundaries.

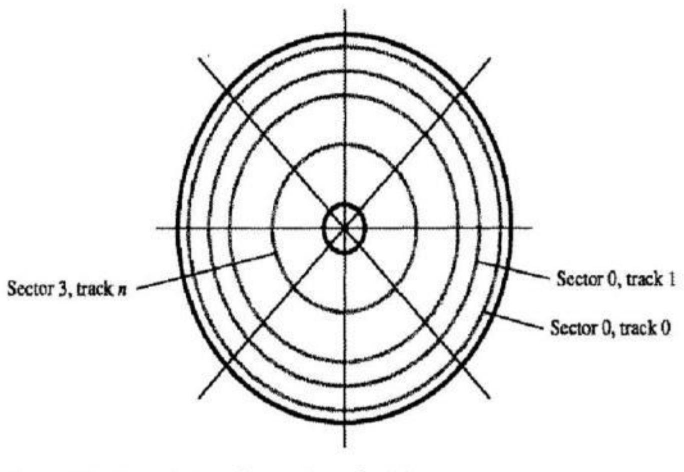


Figure 5.30 Organization of one surface of a disk.

Data bits are stored serially on each track. Each sector usually contains 512 bytes of data, but other sizes may be used. The data are preceded by a *sector header* that contains identification (addressing) information used to find the desired sector on the selected track. Following the data, there are additional bits that constitute an *error correcting code (ECC)*. The ECC bits are used to detect and correct errors that may have occurred in writing or reading of the 512 data bytes. To easily distinguish between two consecutive sectors, there is a small *intersector gap*.

An unformatted disk has no information on its tracks. The formatting process divides the disk physically into tracks and sectors. This process may discover some defective sectors or even whole tracks; the disk controller keeps a record of such defects and excludes them from use.

Access Time:

There are two components involved in the time delay between receiving an address and the beginning of the actual data transfer. The first, called the *seek time*, is the time required to move the read/write head to the proper track. The second component is the *rotational delay*, also called *latency time*. This is the amount of time that elapses after the head is positioned over the correct track until the starting position of the addressed sector passes under the read/write head. The sum of these two delays is called the *disk access time*.

Data Buffer/Cache:

The SCSI bus is capable of transferring data at much higher rates than the rate at which data can be read from disk tracks. An efficient way to deal with the possible differences in transfer rates between the disk and the SCSI bus is to include a *data buffer* in the disk unit. This buffer is a semiconductor memory, capable of storing a few megabytes of data. The requested data are transferred between the disk tracks and the buffer at a rate dependent on the rotational speed of the disk. Transfers between the data buffer and other devices connected to the bus, normally the main memory, can then take place at the maximum rate allowed by the bus.

The data buffer can also be used to provide a caching mechanism for the disk. When a read request arrives at the disk, the controller can first check to see if the desired data are already available in the cache (buffer). If so, the data can be accessed and placed on the SCSI bus in microseconds rather than milliseconds. Otherwise, the data are read from a disk track in the usual way and stored in the cache.

Disk Controller:

Operation of a disk drive is controlled by a disk controller circuit, which also provides an interface between the disk drive and the bus that connects it to the rest of the computer system. The disk controller may be used to control more than one drive. Figure 5.31 shows a disk controller which controls two disk drives.

A disk controller that is connected directly to the processor system bus, or to an expansion bus such as PCI, contains a number of registers that can be read and written by the operating system. The disk controller uses the DMA scheme to transfer data between the disk and the main memory.

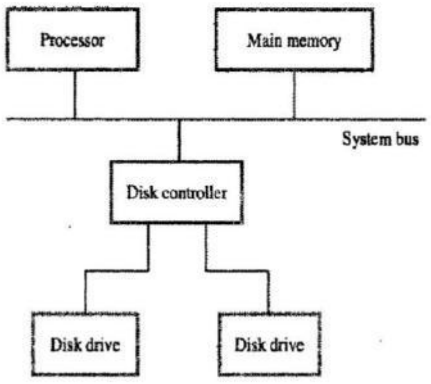


Figure 5.31 Disks connected to the system bus.

The OS initiates the transfers by issuing Read and Write requests, which entail loading the controller's registers with the necessary addressing and control information, typically:

Main memory address- The address of the first main memory location of the block of words involved in the transfer.

Disk address- The location of the sector containing the beginning of the desired block of words

Word count- The number of words in the block to be transferred

The disk address issued by the OS is a logical address. The corresponding physical address on the disk may be different.

On the disk drive side, the controller's major functions are:

Seek— Causes the disk drive to move the read/write head from its current position to the desired track,

Read — Initiates a Read operation, starting at the address specified in the disk address register. Data read serially from the disk are assembled into words and placed into the data buffer for transfer to the main memory. The number of words is determined by the word count register.

Write — Transfers data to the disk, using a control method similar to that for the Read operations.

Error checking - Computes the error correcting code (ECC) value for the data read from 2 given sector and compares it with the corresponding ECC value read from the disk. In case of a mismatch, it corrects the error if possible; otherwise, it raises an interrupt to inform the OS that an error has occurred, During 2 write operation, the controller computes the ECC value for the data to be written and stores this value on the disk.

Software and Operating System implications:

When the power is turned on again, the OS has to be loaded into the main memory, which takes place as part of a process known as *booting*. To initiate booting, a tiny part of main memory is implemented as a nonvolatile ROM. This ROM stores a small monitor program that can read and write main memory locations as well as read one block of data stored on the disk at address 0. This block, referred to as the *boot block*, contains a loader program.

Floppy Disks:

Floppy disks are smaller, simpler, and cheaper disk units that consist of a flexible, removable, plastic diskette coated with magnetic material. The diskette is enclosed in a plastic jacket, which has an opening where the read/write head makes contact with the diskette. A hole in the center of the diskette allows a spindle mechanism in the disk drive to position and rotate the diskette

One of the simplest schemes used in the first floppy disks for recording data is phase or Manchester encoding mentioned earlier. Disks encoded in this way are said to have single density. A more complicated variant of this scheme, called double density, is most often used in current standard floppy disks. It increases the storage density by a factor of 2 but also requires more complex circuits in the disk controller.

Main feature of floppy disks is their low cost and shipping convenience. However, they have much smaller storage capacities, longer access times, and higher failure rates than hard disks. Current standard floppy disks are 3.25 inches in diameter and store 1.44 or 2 Mbytes of date. Larger super-floppy disks are also available.

RAID Disk Arrays:

In 1988, researchers at the University of California-Berkeley proposed a storage system based on multiple disks. They called it RAID, for *Redundant Array of Inexpensive Disks*. Using multiple disks also makes it possible to improve the reliability of the overall system. Six different configurations were proposed. They are known as RAID levels even though there is no hierarchy involved.

RAID0 is the basic configuration intended to enhance performance. A single large file is stored in several separate disk units by breaking the file up into a number of smaller pieces and storing these pieces on different disks. This is called data striping. When the file is accessed for a read, all disks can deliver their data in parallel. In fact, since each disk operates independently of the others, access times vary, and buffering of the accessed pieces of data is needed so that the complete file can be reassembled and sent to the requesting processor as a single entity.

RAID 1 is intended to provide better reliability by storing identical copies of data on two disks rather than just one. The two disks are said to be mirrors of each other. Then, if one disk drive fails, all read and write operations are directed to its mirror drive. This is a costly way to improve the reliability because all disks are duplicated.

RAID 2, RAID 3, and RAID 4 levels achieve increased reliability through various parity checking schemes without requiring a full duplication of disks. All of the parity information is kept on one disk.

RAID 5 also makes use of a parity-based error-recovery scheme. However, the parity information is distributed among all disks, rather than being stored on one disk.

Some hybrid arrangements have subsequently been developed. For example, RAID 10 is an array that combines the features of RAID 0 and RAID 1.

OPTICAL DISKS:

The first generation of CDs was developed in the mid-1980s by the Sony and Philips companies. To provide high-quality sound recording and reproduction, 16-bit samples of the analog signal are taken at a rate of 44,100 samples per second.

CD Technology:

The optical technology that is used for CD systems is based on a laser light source. A laser beam is directed onto the surface of the spinning disk. Physical indentations in the surface are arranged along the tracks of the disk. They reflect the focused beam toward a photodetector, which detects the stored binary patterns.

The laser emits a coherent light beam that is sharply focussed on the surface of the disk. Coherent light consists of synchronized waves that have the same wavelength. If a coherent light beam is combined with another beam of the same kind, and the two beams are in phase, then the result will be a brighter beam. But, if the waves of the two beams are 180 degrees out of phase, they will cancel each other. Thus, if a photodetector is used to detect the beams, it will detect a bright spot in the first case and a dark spot in the second case.

A cross-section of a small portion of a CD is shown in Figure 5.32a. The bottom layer is *polycarbonate plastic*, which functions as a clear glass base. The surface of this plastic is programmed to store data by indenting it with pits. The unindented parts are called ands, A thin layer of reflecting aluminum material is placed on top of a programmed disk. The aluminum is then covered by a protective acrylic. Finally, the topmost layer is deposited and stamped with a label. The total thickness of the disk is 1.2mm.

Figure 5.32b shows what happens as the laser beam scans across the disk and encounters a transition from a pit to a land. ‘Three different positions of the laser source and the detector are shown, as would occur when the disk is rotating. When the light reflects solely from the pit, or solely from the land, the detector will see the reflected beam as a bright spot. But, a different situation arises when the beam moves through the edge where the pit changes to the land, and vice versa.

Figure 5.32c depicts several transitions between lands and pits. If each transition, detected as a dark spot, is taken to denote the binary value 1, and the flat portions represent 0s.

CD is 120 mm in diameter. There is a 15-mm hole in the center. Data are stored on tracks that cover the area from 25-mm radius to 58-mm radius. The space between the tracks is 1.6 microns. Pits are 0.5 microns wide and 0.8 to 3 microns long. There are more than 15,000 tracks on a disk.

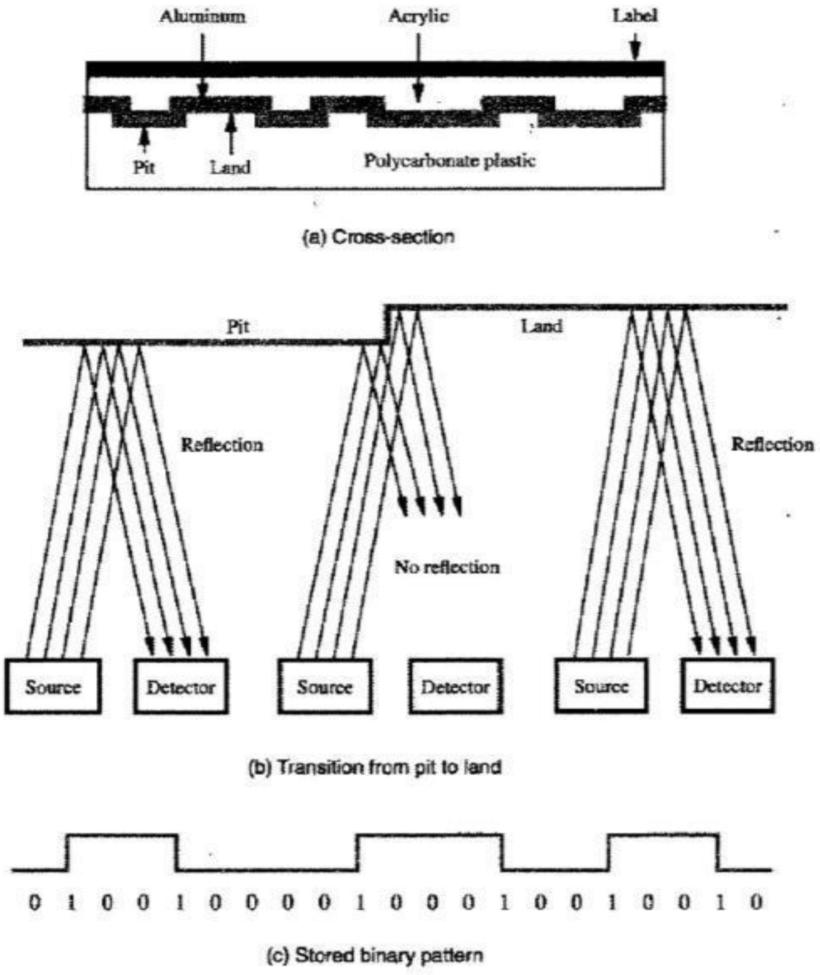


Figure 5.32 Optical disk.

CD-ROM:

CD-ROMs contents can only be read, as with semiconductor ROM chips.

Stored data are organized on CD-ROM tracks in the form of blocks that are called sectors. There are several different formats for a sector. One format, known as Mode 1, uses 2352-byte sectors. There is a 16-byte header that contains a synchronization field used to detect the beginning of the sector and addressing information used to identify the sector. This is followed by 2048 bytes of stored data. At the end of the sector, there are 288 bytes used to implement the error-correcting scheme. The number of sectors per track is variable; there are more sectors on the longer outer tracks.

Error detection and correction is done at more than one level. Each byte of stored information is encoded using a 14-bit code that has some error-correcting capability. This code can correct single-bit errors. Errors that occur in short bursts, affecting several bits, are detected and corrected using the error-checking bits at the end of the sector.

CD-ROM drives operate at a number of different rotational speeds. The basic speed, known as 1X, is 75 sectors per second. This provides a data rate of 153,600 bytes/s (150 Kbytes/s), using the Mode 1 format. With this speed and format, a CD-ROM based on the standard CD designed for 75 minutes of music has a data storage capacity of about 650 Mbytes. *Note that the speed of the drive affects only the data transfer rate bat not the storage capacity of the disk.* A 40X CD-ROM has a data transfer rate that is 40 times higher than that of the 1X CD-ROM.

The importance of CD ROMs for computer systems stems from their large storage capacity and fast access times compared to other inexpensive portable media, such as floppy disks and magnetic tapes. They are widely used for the distribution of software, databases, large texts (books), application programs, and video games.

CD-Recordables:

CD-R was developed in the late 1990s on which data can be easily recorded by a computer user. A spiral track is implemented on a disk during the manufacturing process. A laser in a CD-R drive is used to burn pits into an organic dye on the track. When a burned spot is heated beyond a critical temperature, it becomes opaque. Such burned spots reflect less light when subsequently read. The written data are stored permanently. Unused portions of a disk can be used to store additional data at a later time.

CD-ReWritables:

The most flexible CDs are those that can be written multiple times by the user. They are known as CD-RWs (CD-ReWritables).

The basic structure of CD-RWs is similar to the structure of CD-Rs. Instead of using an organic dye in the recording layer, an alloy of silver, indium, antimony and tellurium is used.

The CD-RW drive uses three different laser powers. The highest power is used to record the pits. The middle power is used to put the alloy into its crystalline state; it is referred to as the “erase power.” The lowest power is used to read the stored information. There is a limit on how many times a CD-RW disk can be rewritten. Presently, this can be done up to 1000 times.

CD-RWs can be used for low-volume distribution of information, just like CD-Rs. The CD-RW drives are now fast enough to be used for daily hard disk backup purposes.

DVD Technology:

The first DVD standard was defined in 1996 by a consortium of companies. The objective is to be able to store a full-length movie on one side of a DVD disk.

The physical size of a DVD disk is the same as for CDs. The disk is 1.2 mm thick, and itis 120 mm in diameter. Its storage capacity is made much larger than that of CDs by several design changes:

- A red light laser with a wavelength of 635 nm is used instead of the infrared light laser used in CDs, which has a wavelength of 780 nm. The shorter wavelength makes it possible to focus the light to a smaller spot.
- Pits are smaller, having a minimum length of 0.4 micron.
- Tracks are placed closer together; the distance between tracks is 0.74 micron

Using these improvements leads to a DVD capacity of 4.7 Gbytes.

The single-layered single-sided disk, defined in the standard as DVD-5. A double-layered disk makes use of two layers on which tracks are implemented on top of each other. The first layer is the clear base, as in CD disks. But, instead of using reflecting aluminum, the lands and pits of this layer are covered by a translucent material that acts as a semireflector. The surface of this material is then also programmed with indented pits to store data. A reflective material is placed on top of the second layer of pits and lands. The disk is read by focusing the laser beam on the desired layer. When the bear is focused on the first layer, sufficient light is reflected by the translucent material to detect the stored binary patterns. When the beam is focused on the second layer, the light reflected by the reflective material corresponds to the information stored on this layer. In both cases, the layer on which the beam is not focused reflects a much smaller amount of light, which is eliminated by the detector circuit as noise. The total storage capacity of both layers is 8.5 Gbytes. This disk is called DVD-9 in the standard.

Two single-sided disks can be put together to form a sandwich-like structure where the top disk is turned upside down. This can be done with single-layered disks, as specified in DVD-10, giving a composite disk with a capacity of 9.4 Gbytes. It can also be done with the double-layered disks, as specified in DVD-18, yielding a capacity of 17 Gbytes.

DVD-RAM:

A rewritable version of DVD devices, known as DVD-RAM, has also been developed. It provides a large storage capacity. Its only disadvantages are the higher price and the relatively slow writing speed. To ensure that the data have been recorded correctly on the disk, a process known as write verification is performed. This is done by the DVD-RAM drive, which reads the stored contents and checks them against the original data

MAGNETIC TAPE SYSTEMS:

Magnetic tapes are suited for off-line storage of large amounts of data. They are typically used for hard disk backup purposes and for archival storage.

Data on the tape are organized in the form of *records* separated by gaps, as shown in Figure 5.33. Tape motion is stopped only when a record gap is underneath the read/write heads. The record gaps are long enough to allow the tape to attain its normal speed before the beginning of the next record is reached. Gaps are identified as areas where there is no change in magnetization. This allows record gaps to be detected independently of the recorded data. To help users organize large amounts of data, a group of related records is called a *file*. The beginning of a file is identified by a file mark, as shown in Figure 5.33. The file mark is a special single- or multiple character record, usually preceded by a gap longer than the interrecord gap.

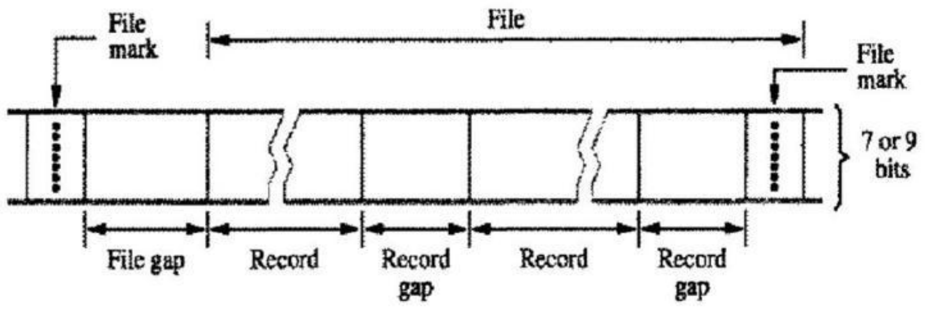


Figure 5.33 Organization of data on magnetic tape.

The first record following a file mark can be used as a *header* or *identifier* for this file. This allows the user to search a tape containing a large number of files for a particular file.

The controller of a magnetic tape drive enables the execution of a number of control commands in addition to read and write commands. Control commands include the following operations:

- Rewind tape
- Rewind and unload tape
- Erase tape
- Write tape mark
- Forward space one record
- Backspace one record
- Forward space one file
- Backspace one file

The tape mark referred to in the operation "Write tape mark" is similar to a file mark except that it is used for identifying the beginning of the tape. The end of the tape is sometimes identified by the EOT (end of tape) character.

Two methods of formatting and using tapes are available. In the *first method*, the records are variable in length. This allows efficient use of the tape, but it does not permit updating or overwriting of records in place. The *second method* is to use fixed-length records. In this case, it is possible to update records in place.

Cartridge Tape System:

Tape systems have been developed for backup of on-line disk storage. One such system uses an 8-mm video format tape housed in a cassette. These units are called *cartridge tapes*. They have capacities in the range of 2 to 5 gigabytes and handle data transfers at the rate of a few hundred kilobytes per second. Reading and writing is done by a helical scan system operating across the tape. Bit densities of tens of millions of bits per square inch are achievable.

ANNAMACHARYA INSTITUTE OF TECHNOLOGY AND SCIENCES::TIRUPATI
II B.Tech II Semester

L	T	P	C
3	0	0	3

(19APC0506) Computer Organization

Course Objectives:

- To learn the fundamentals of computer organization and its relevance to classical and modern problems of computer design
- To make the students understand the structure and behavior of various functional modules of a computer.
- To understand the techniques that computers use to communicate with I/O devices
- To study the concepts of pipelining and the way it can speed up processing.
- To understand the basic characteristics of multiprocessors

Unit I:
Basic Structure of Computer: Computer Types, Functional Units, Basic operational Concepts, Bus Structure, Software, Performance, Multiprocessors and Multicomputer.
Machine Instructions and Programs: Numbers, Arithmetic Operations and Programs, Instructions and Instruction Sequencing, Addressing Modes, Basic Input/output Operations, Stacks and Queues, Subroutines, Additional Instructions.
Unit II:
Arithmetic: Addition and Subtraction of Signed Numbers, Design and Fast Adders, Multiplication of Positive Numbers, Signed-operand Multiplication, Fast Multiplication, Integer Division, Floating-Point Numbers and Operations.
Basic Processing Unit: Fundamental Concepts, Execution of a Complete Instruction, Multiple-Bus Organization, Hardwired Control, Multiprogrammed Control.
Unit III:
The Memory System: Basic Concepts, Semiconductor RAM Memories, Read-Only Memories, Speed, Size and Cost, Cache Memories, Performance Considerations, Virtual Memories, Memory Management Requirements, Secondary Storage.
Unit IV:
Input/output Organization: Accessing I/O Devices, Interrupts, Processor Examples, Direct Memory Access, Buses, Interface Circuits, Standard I/O Interfaces.
Unit V:
Pipelining: Basic Concepts, Data Hazards, Instruction Hazards, Influence on Instruction Sets
Large Computer Systems: Forms of Parallel Processing, Array Processors, The Structure of General-Purpose, Interconnection Networks.

Textbook:
1. "Computer Organization", Carl Hamacher, Zvonko Vranesic, Safwat Zaky, McGraw Hill Education, 5th Edition, 2013.

Reference Textbooks:
1. Computer System Architecture, M.Morris Mano, Pearson Education, 3rd Edition.
2. Computer Organization and Architecture, Themes and Variations, Alan Clements, CENGAGE Learning.
3. Computer Organization and Architecture, Smruti Ranjan Sarangi, McGraw Hill Education.
4. Computer Architecture and Organization, John P.Hayes, McGraw Hill Education.

Course Outcomes:

- Ability to use memory and I/O devices effectively
- Able to explore the hardware requirements for cache memory and virtual memory
- Ability to design algorithms to exploit pipelining and multiprocessors

Unit IV:
Input/output Organization: Accessing I/O Devices, Interrupts, Processor Examples, Direct Memory Access, Buses, Interface Circuits, Standard I/O Interfaces.

INPUT/OUTPUT ORGANIZATION

ACCESSING I/O DEVICES

A simple arrangement to connect I/O devices to a computer is to use a single bus arrangement. The bus enables all the devices connected to it to exchange information. Typically, it consists of three sets of lines used to *carry address*, *data*, and *control signals*. Each I/O device is assigned a unique set of addresses. When the processor places a particular address on the address lines, the device that recognizes this address responds to the commands issued on the control lines. The processor requests either a read or a write operation, and the requested data are transferred over the data lines. When I/O devices and the memory share the same address space, the arrangement is called *memory-mapped I/O*.

With memory-mapped I/O, any machine instruction that can access memory can be used to transfer data to or from an I/O device.

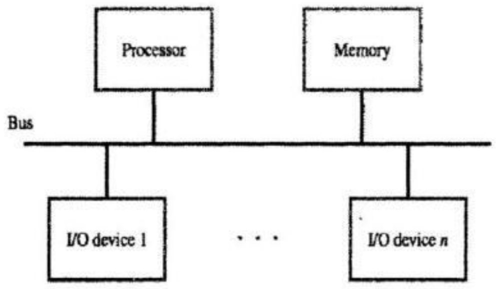


Figure 4.1 A single-bus structure.

For example, if DATAIN is the address of the input buffer associated with the keyboard, the instruction

Move DATAIN,R0

Reads the data from DATAEN and stores them into processor register R0.

Similarly, the instruction

Move R0, DATAOUT

Sends the contents of register R0 to location DATAOUT, which may be the output data buffer of a display unit or a printer.

Figure 4.2 illustrates the hardware required to connect an I/O device to the bus. The *address decoder* enables the device to recognize its address when this address appears on the address lines. The *data register* holds the data being transferred to or from the processor. The *status register* contains information relevant to the operation of the I/O device. Both the data and status registers are connected to the data bus and assigned unique addresses. The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's *interface circuit*.

For an input device such as a keyboard, a status flag, SIN, is included in the interface circuit as part of the status register. This flag is set to **1** when a character is entered at the keyboard and cleared to **0** once this character is read by the processor. Hence, by checking the SIN flag, the software can ensure that it is always reading valid data. A similar procedure can be used to control output operations using an output status flag, SOUT.

Example: Let us consider a simple example of 1/O operations involving a keyboard and a display device in a computer system. The four registers shown in Figure 4.3 are used in the data transfer operations. Register STATUS contains two control flags, SIN and SOUT, which provide status information for the keyboard and the display unit, respectively. The two flags KIRQ and DIRQ in this register are used in conjunction with interrupts. Data from the keyboard are made available in the DATAIN register, and data sent to the display are stored in the DATAOUT register.

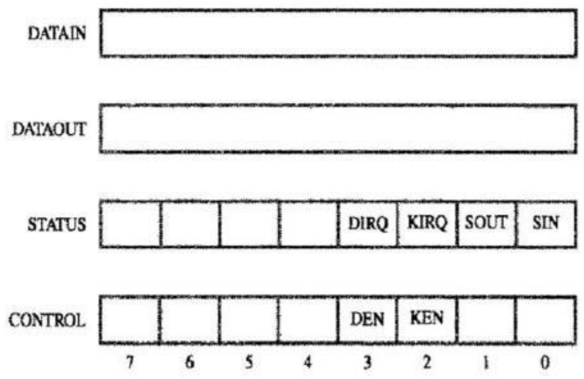


Figure 4.3 Registers in keyboard and display interfaces.

In *program-controlled I/O* the processor repeatedly checks a status flag to achieve the required synchronization between the processor and an input or output device. We say that the processor *polls* the device.

There are two other commonly used mechanisms for implementing I/O operations: *interrupts* and *direct memory access*. In the case of *interrupts*, synchronization is achieved by having the I/O device send a special signal over the bus whenever it is ready for a data transfer operation. *Direct memory access* is a technique used for high-speed I/O devices. It involves having the device interface transfer data directly to or from the memory, without continuous involvement by the processor.

INTERRUPTS

There are many situations where other tasks can be performed while waiting for an I/O device to become ready. To allow this to happen, we can arrange for the I/O device to alert the processor when it becomes ready. It can do so by sending a hardware signal called an *interrupt* to the processor. At least one of the bus control lines, called an *interrupt-request* line, is usually dedicated for this purpose. Since the processor is no longer required to continuously check the status of external devices, it can use the waiting period to perform other useful functions, Indeed, by using interrupts, such waiting periods can ideally be eliminated,

The routine executed in response to an interrupt request is called the *interrupt-service routine*.

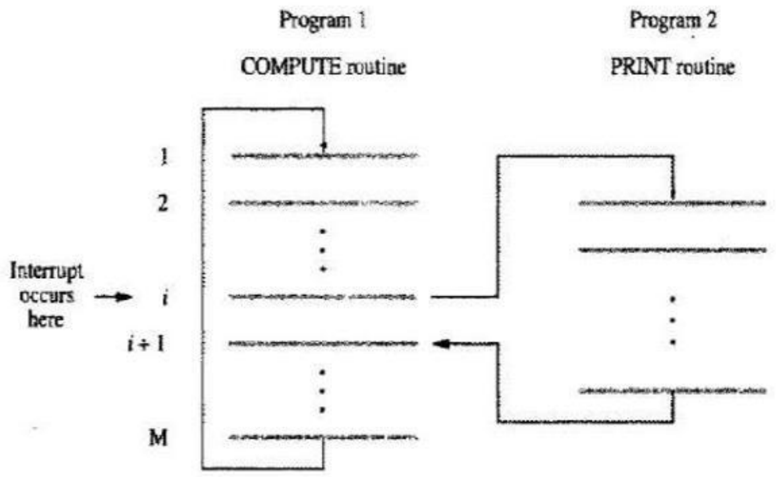


Figure 4.5 Transfer of control through the use of interrupts.

In above figure, the processor first completes execution of instruction *i*. Then, it loads the program counter with the address of the first instruction of the interrupt-service routine. For the time being, let us assume that this address is hardwired in the processor. After execution of the interrupt-service routine, the processor has to come back to instruction *i + 1*. Therefore, when an interrupt occurs, the current contents of the PC, which point to instruction *i + 1*, must be put in temporary storage in a known location. A Return- from-interrupt instruction at the end of the interrupt-service routine reloads the PC from that temporary storage location, causing execution to resume at instruction *i + 1*.

The processor must inform the device that its request has been recognized so that it may remove its interrupt-request signal. This may be accomplished by means of a special control signal on the bus. An *interrupt-acknowledge signal*, used in some of the interrupt schemes serves this function. The execution of an instruction in the interrupt-service routine that accesses a status or data register in the device interface implicitly informs the device that its interrupt request has been recognized.

The task of saving and restoring information can be done automatically by the processor or by program instructions. Most modern processors save only the minimum amount of information needed to maintain the integrity of program execution. This is because the process of saving and restoring registers involves memory transfers that increase the total execution time, and hence represent execution overhead. Saving registers also increases the delay between the time an interrupt request is received and the start of execution of the interrupt-service routine. This delay is called *interrupt latency*.

Interrupts enable transfer of control from one program to another to be initiated by an event external to the computer. Execution of the interrupted program resumes after the execution of the interrupt-service routine has been completed. The concept of interrupts is used in operating systems and in many control applications where processing of certain routines must be accurately timed relative to external events. The latter type of application is referred to as real-time processing.

Interrupt Hardware:

A single interrupt-request line may be used to serve n devices as in below figure.

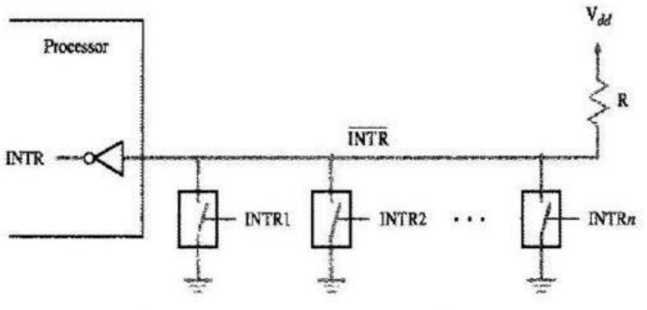


Figure 4.6 An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

All devices are connected to the line via switches to ground. To request an interrupt, a device closes its associated switch. Thus, if all interrupt-request signals $INTR_1$, to $INTR_n$, are inactive, that is, if all switches are open, the voltage on the interrupt-request line will be equal to V_{dd}. This is the inactive state of the line. When a device requests an interrupt by closing its switch, the voltage on the line drops to 0, causing the interrupt-request signal, INTR, received by the processor to go to 1. Since the closing of one or more switches will cause the line voltage to drop to 0, the value of INTR is the logical OR of the requests from individual devices, that is,

$$INTR=INTR_1+_{\dots\dots}+INTR_n$$

It is customary to use the complemented form, $\overline{\text{INTR}}$, to name the interrupt-request signal on the common line, because this signal is active when in the low-voltage state.

Special gates in above figure, known as *open-collector* (for bipolar circuits) or *open-drain* (for MOS circuits) are used to drive the INTR line. The output of an open-collector or an open-drain gate is equivalent to a switch to ground that is open when the gate's input is in the 0 state and closed when it is in the 1 state. Resistor *R* is called a *pull-up resistor* because it pulls the line voltage up to the high-voltage state when the switches are open.

ENABLING AND DISABLING INTERRUPTS:

The arrival of an interrupt request from an external device causes the processor to suspend the execution of one program and start the execution of another. Because interrupts can arrive at any time, they may alter the sequence of events from that envisaged by the programmer.

When a device activates the interrupt-request signal, it keeps this signal activated until it learns that the processor has accepted its request, This means that the interrupt- request signal will be active during execution of the interrupt-service routine, perhaps until an instruction is reached that accesses the device in question. This means that the interrupt- request signal will be active during execution of the interrupt-service routine, perhaps until an instruction is reached that accesses the device in question. It is essential to ensure that this active request signal does not lead to successive interruptions, causing the system to enter an infinite loop from which it cannot recover.

Several mechanisms are available to solve this problem. Among them *three possibilities* are here:

- The *first possibility* is to have the processor hardware ignore the interrupt-request line until the execution of the first instruction of the interrupt-service routine has been completed. Then, by using an Interrupt-disable instruction as the first instruction in the interrupt-service routine, the programmer can ensure that no further interruptions will occur until an Interrupt-enable instruction is executed. Typically, the Interrupt- enable instruction will be the last instruction in the interrupt-service routine before the Return-from-interrupt instruction.
- The *second option*, which is suitable for a simple processor with only one interrupt-request line, is to have the processor automatically disable interrupts before starting the execution of the interrupt-service routine. After saving the contents of the PC and the *processor status register* (PS) on the stack, the processor performs the equivalent of executing an Interrupt-disable instruction. It is often the case that one bit in the PS register, called Interrupt-enable, indicates whether interrupts are enabled. An interrupt request received while this bit is equal to 1 will be accepted. After saving the contents of the PS on the stack, with the Interrupt-enable bit equal to 1, the processor clears the Interrupt-enable bit in its PS register, thus disabling further interrupts. When a Return-from-interrupt instruction is executed, the contents of the PS are restored from the stack, setting the Interrupt-enable bit back to 1.
- In the *third option*, the processor has a special interrupt-request line for which the interrupt-handling circuit responds only to the leading edge of the signal. Such a line is said to be edge-triggered. In this case, the processor will receive only one request, regardless of how long the line is activated.

Before proceeding to study more complex aspects of interrupts, let us summarize the *sequence of events* involved in handling an interrupt request from a single device. Assuming that interrupts are enabled, the following is a typical scenario:

1. *The device raises an interrupt request.*

2. *The processor interrupts the program currently being executed.*
3. *Interrupts are disabled by changing the control bits in the PS (except in the case of edge-triggered interrupts).*
4. *The device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.*
5. *The action requested by the interrupt is performed by the interrupt-service routine.*
6. *Interrupts are enabled and execution of the interrupted program is resumed.*

HANDLING MULTIPLE DEVICES:

Let us now consider the situation where a number of devices capable of initiating interrupts are connected to the processor. Because these devices are operationally independent, there is no definite order in which they will generate interrupts, For example, device X may request an interrupt while an interrupt caused by device Y is being serviced, or several devices may request interrupts at exactly the same time. This gives rise to a number of questions:

1. How can the processor recognize the device requesting an interrupt?
2. Given that different devices are likely to require different interrupt-service routines, how can the processor obtain the starting address of the appropriate routine in each case?
3. Should a device be allowed to interrupt the processor while another interrupt is being serviced?
4. How should two or more simultaneous interrupt requests be handled?

If two devices have activated the line at the same time, it must be possible to break the tie and select one of the two requests for service. When the interrupt- service routine for the selected device has been completed, the second request can be serviced.

The information needed to determine whether a device is requesting an interrupt is available in its *status register*. When a device raises an interrupt request, it sets to 1 one of the bits in its status register, which we will call the IRQ bit. The simplest way to identify the interrupting device is to have the interrupt-service routine poll all the I/O devices connected to the bus. The first device encountered with its IRQ bit set is the device that should be serviced. An appropriate subroutine is called to provide the requested service.

The polling scheme is easy to implement. Its main *disadvantage* is the time spent interrogating the IRQ bits of all the devices that may not be requesting any service. An alternative approach is to use *vectored interrupts*.

Vectored Interrupts:

In *vectored interrupts* approach, to reduce the time involved in the polling process, a device requesting an interrupt may identify itself directly to the processor. Then, the processor can immediately start executing the corresponding interrupt-service routine.

A device requesting an interrupt can identify itself by sending a special code to the processor over the bus. This enables the processor to identify individual devices even if they share a single interrupt-request line. The code supplied by the device may represent the *starting address* of the interrupt-service routine for that device. The code length is typically in the range of 4 to 8 bits. The remainder of the address is supplied by the processor based on the area in its memory where the addresses for interrupt-service routines are located.

The location pointed to by the interrupting device is used to store the starting address of the interrupt-service routine. The processor reads this address, called the *interrupt vector*, and loads it into the PC. The interrupt vector may also include a new value for the processor status register.

In most computers, I/0 devices send the interrupt-vector code over the data bus, using the bus control signals to ensure that devices do not interfere with each other. When a device sends an interrupt request, the processor may not be ready to receive the interrupt-vector code immediately. The interrupting device must wait to put data on the bus only when the processor js ready to receive it, When the processor is ready to receive the interrupt-vector code, if activates the interrupt-acknowledge line, INTA. The I/O device responds by sending its interrupt- vector code and turning off the INTR signal.

Interrupt Nesting:

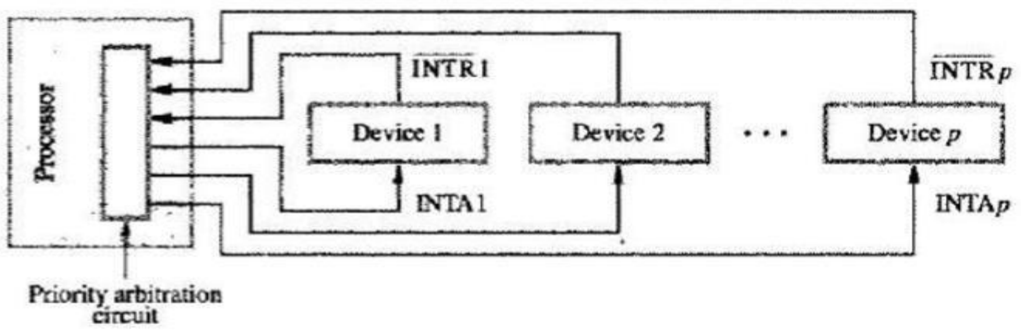
I/O devices should be organized in a priority structure. An interrupt request from a high-priority device should be accepted while the processor is servicing another request from a lower-priority device.

A multiple-level priority organization means that during execution of an interrupt service routine, interrupt requests will be accepted from some devices but not from others, depending upon the device's priority. To implement this scheme, we can assign a priority level to the processor that can be changed under program control. The priority level of the processor is the priority of the program that is currently being executed. The processor accepts interrupts only from devices that have priorities *higher* than its own. At the time the execution of an interrupt-service routine for some device is started, the priority of the processor is raised to that of the device. This action disables interrupts from devices at the same level of priority or lower.

The processor's priority is usually encoded in a few bits of the processor status word. It can be changed by program instructions that write into the PS. These are privileged instructions, which can be executed only while the processor is running in the supervisor mode. The processor is in the supervisor mode only when executing operating system routines. It switches to the user mode before beginning to execute application programs. Thus, a user program cannot accidentally, or intentionally, change the priority of the processor and disrupt the system's operation. An attempt to execute a privileged instruction while in the user mode leads to a special type of interrupt called a *privilege exception*.

A multiple-priority scheme can be implemented easily by using separate interrupt- Request and interrupt-acknowledge lines for each device, as shown in above figure.

Each of the interrupt-request lines is assigned a different priority level. Interrupt requests received over these lines are sent to a priority arbitration circuit in the processor. A request is accepted only if it has a higher priority level than that currently assigned to the processor.



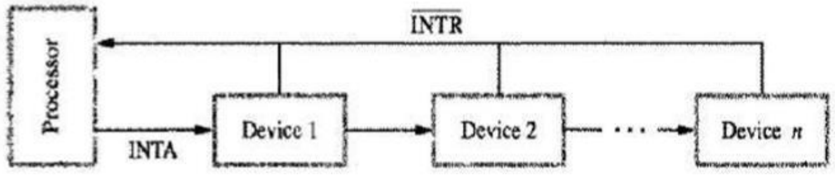
Processor
$\overline{\text{INTR}}$1
$\overline{\text{INTR}}p$
Device 1
Device 2
Device p
INTA1
INTAp
Priority arbitration circuit

Figure 4.7 Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

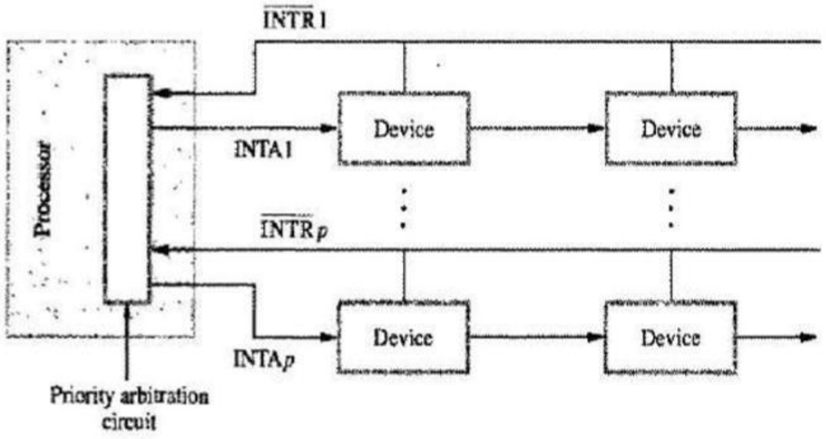
Simultaneous Requests:

In this case, priority is determined by the order in which the devices are polled. When vectored interrupts are used, we must ensure that only one device is selected to send its interrupt vector code. A widely used scheme is to connect the devices to form a *daisy chain*, as shown in Figure 4.8a. The interrupt-request line $\overline{\text{INTR}}$ is common to all devices. The interrupt-acknowledge line, INTA, is connected in a daisy-chain fashion, such that the INTA signal propagates serially through the devices. When several devices raise an interrupt request and the $\overline{\text{INTR}}$ line is activated, the processor responds by setting the INTA line to 1. This signal is received by device 1. Device 1 passes the signal on to device 2 only if it does not require any service. If device 1 has a pending request for interrupt, it blocks the INTA signal and proceeds to put its identifying code on the data lines. Therefore, in the daisy-chain arrangement, the device that is electrically closest to the processor has the highest priority. The second device along the chain has second highest priority, and so on.

In Figure 4.8(b), Devices are organized in groups, and each group is connected at a different priority Level. Within a group, devices are connected in a daisy chain. This organization is used in many computer systems.



(a) Daisy chain



$\overline{\text{INTR}}$1
Processor
INTA1
Device
Device
$\overline{\text{INTR}}p$
INTAp
Device
Device
Priority arbitration circuit

(b) Arrangement of priority groups

Figure 4.8 Interrupt priority schemes.

CONTROLLING DEVICE REQUESTS:

A mechanism is needed in the interface circuits of individual devices to control whether a device is allowed to generate an interrupt request.

The control needed is usually provided in the form of an interrupt-enable bit in the device's interface circuit. The keyboard interrupt-enable, KEN, and display interrupt- enable, DEN, flags in register CONTROL. If either of these flags is set, the interface circuit generates an interrupt request whenever the corresponding status flag in register STATUS is set. At the same time, the interface circuit sets bit KIRQ or DIRQ to indicate that the keyboard or display unit, respectively, is requesting an interrupt. If an interrupt-enable bit is equal to 0, the interface circuit will not generate an interrupt request, regardless of the state of the status flag.

To summarize, *there are two independent mechanisms for controlling interrupt requests. At the device end, an interrupt-enable bit in a control register determines whether the device is allowed to generate an interrupt request. At the processor end, either an interrupt enable bit in the PS register or a priority structure determines whether a given interrupt request will be accepted*.

EXCEPTIONS:

An interrupt is an event that causes the execution of one program to be suspended and the execution of another program to begin.

The term *exception* is often used to refer to any event that causes an interruption.

Other kinds of exceptions:

Recovers from Errors:

Computers use a variety of techniques to ensure that all hardware components are operating properly. For example, many computers include an error-checking code in the main memory, which allows detection of errors in the stored data. If an error occurs, the control hardware detects it and informs the processor by raising an interrupt.

The processor may also interrupt a program if it detects an error or an *unusual condition* while executing the instructions of this program.

When exception processing is initiated as a result of such errors, the processor proceeds in exactly the same manner as in the case of an I/O interrupt request. It suspends the program being executed and starts an exception-service routine. This routine takes appropriate action to recover from the error, if possible, or to inform the user about it. However, when an interrupt is caused by an error, execution of the interrupted instruction cannot usually be completed, and the processor begins exception processing immediately.

Debugging:

System software usually includes a program called a *debugger,* which helps the programmer find errors in a program. The debugger uses exceptions to provide two important facilities called *trace* and *breakpoints*.

When a processor is operating in the trace mode, an exception occurs after execution of every instruction, using the debugging program as the exception-service routine. The debugging program enables the user to examine the contents of registers, memory locations, and so on. On return from the debugging program, the next instruction in the program being debugged is executed, and then the debugging program is activated again. The trace exception is disabled during the execution of the debugging program.

Breakpoints provide a similar facility, except that the program being debugged is interrupted only at specific points selected by the user. An instruction called Trap or Software-interrupt is usually provided for this purpose. Execution of this instruction results in exactly the same actions as when a hardware interrupt request is received. While debugging a program, the user may wish to interrupt program execution after instruction i. The debugging routine saves instruction i + 1 and replaces it with a software interrupt instruction. When the program is executed and reaches that point, it is interrupted and the debugging routine is activated. This gives the user a chance to examine memory and register contents, When the user is ready to continue executing the program being debugged, the debugging routine restores the saved instruction that was at location i + 1 and executes a Return-from-interrupt instruction.

Privilege Exception:

To protect the operating system of a computer from being corrupted by user pro- grams, certain instructions can be executed only while the processor is in the supervisor mode. These are called *privileged instructions*. An attempt to execute such an instruction will produce a *privilege exception*, causing the processor to switch to the supervisor mode and begin executing an appropriate routine in the operating system.

DIRECT MEMORY ACCESS

A special control unit may be provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor. This approach is called *direct memory access*, or *DMA*.

DMA transfers are performed by a control circuit that is part of the 1/0 device interface, called *DMA controller*. The DMA controller performs the functions that ‘would normally be carried out by the processor when accessing the main memory. For each word transferred, it provides the memory address and all the bus signals that control data transfer. Since it has to transfer blocks of data, the DMA controller roust increment the memory address for successive words and keep track of the number of transfers.

To initiate the transfer of a block of words, the processor sends the starting address, the number of words in the block, and the direction of the transfer. On receiving this information, the DMA controller proceeds to perform the requested operation. When the entire block has been transferred, the controller informs the processor by raising an interrupt signal.

I/O operations are always performed by the operating system of the computer in response to a request from an application program. The OS is also responsible for suspending the execution of one program and starting another. Thus, for an I/O operation involving DMA, the OS puts the program that requested the transfer in the Blocked state, initiates the DMA operation, and starts the execution of another program. When the transfer is completed; the DMA controller informs the processor by sending an interrupt request. In response, the OS puts the suspended program in the Runnable state so that it can be selected by the scheduler to continue execution.

Below diagram shows an example of the DMA controller registers that are accessed by the processor to initiate transfer operations. Two registers are used for storing the starting address and the word count. The third register contains status and control flags. The $R/\overline{W}$ bit determines the direction of the transfer. When this bit is set to 1 by a program instruction, the controller performs a *read operation*, that is, it transfers data from the memory to the I/O device. Otherwise, it performs a *write operation*.

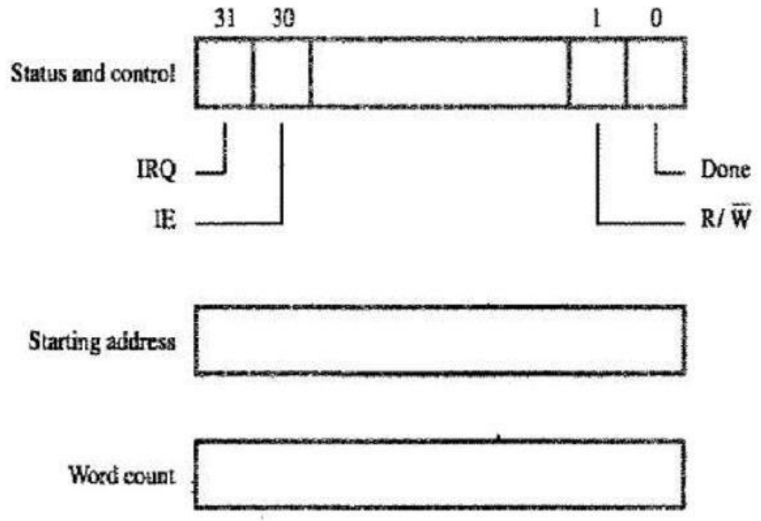


Figure 4.18 Registers in a DMA interface.

When the controller has completed transferring a block of data and is ready to receive another command, it sets the *Done* flag to 1. Bit 30 is the *Interrupt-enable flag, IE*. When this flag is set to 1, it causes the controller to raise an interrupt after it has completed transferring a block of data. Finally, the controller sets the *IRQ* bit to 1 when it has requested an interrupt.

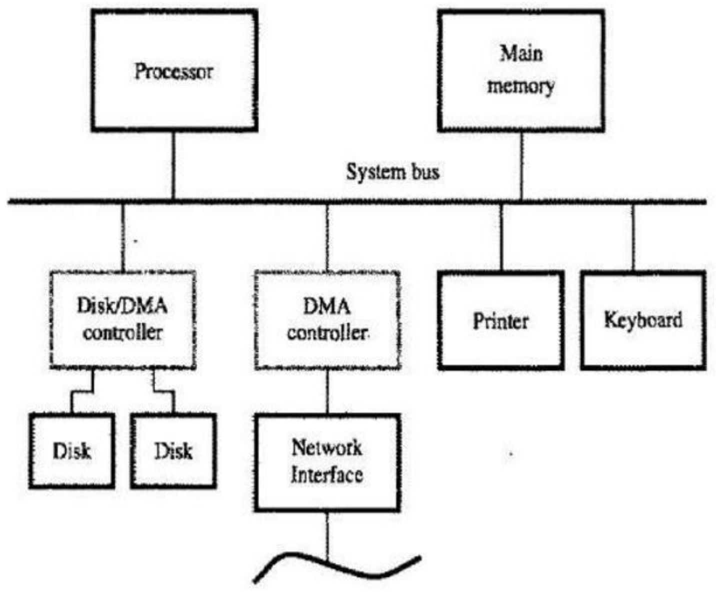


Figure 4.19 Use of DMA controllers in a computer system.

Above diagram shows how DMA controllers may be used. A DMA controller connects a high-speed network to the computer bus. The *disk controller*, which controls two disks, also has DMA capability and provides two DMA channels. It can perform two independent DMA operations, as if each disk had its own DMA controller. The registers needed to store the memory address, the *word count*, and so on are duplicated, so that one set can be used with each device.

To start a DMA transfer of a block of data from the main memory to one of the disks, a program writes the address and word count information into the registers of the corresponding channel of the disk controller. It also provides the disk controller with information to identify the data for future retrieval. If the *IE* bit is set, the controller sends an interrupt request to the processor and sets the IRQ bit. The status register can also be used to record other information, such as whether the transfer took place correctly or errors occurred.

Memory accesses by the processor and the DMA controllers are interwoven. Re- quests by DMA devices for using the bus are always given higher priority than processor requests. Among different DMA devices, top priority is given to high-speed peripherals such as a disk, a high-speed network interface, or a graphics display device. Since the processor originates most memory access cycles, the DMA controller can be said to *"steal"* memory cycles from the processor. Hence, this interweaving technique is usually called *cycle stealing*. Alternatively, the DMA controller may be given exclusive access to the main memory to transfer a block of data without interruption. This is known as *block* or *burst mode*.

BUS ARBITRATION:

The device that is allowed to initiate data transfers on the bus at any given time is called the *bus master*. When the current master relinquishes control of the bus, another device can acquire this status. *Bus arbitration* is the process by which the next device to become the bus master is selected and bus mastership is transferred to it. The selection of the bus master must take into account the needs of various devices by establishing a priority system for gaining access to the bus.

There are two approaches to bus arbitration: *centralized* and *distributed*. In *centralized arbitration*, a single bus arbiter performs the required arbitration. In *distributed arbitration*, all devices participate in, the selection of the next bus master.

Centralized Arbitration:

The bus arbiter may be the processor or a separate unit connected to the bus. Figure 4.20 illustrates a basic arrangement in which the processor contains the bus arbitration circuitry. In this case, the processor is normally the bus master unless it grants bus mastership to one of the DMA controllers. A DMA controller indicates that it needs to become the bus master by activating the *Bus-Request line, BR*. The signal on the Bus-Request line is the *logical OR* of the bus requests from all the devices connected to it. When Bus-Request is activated, the processor activates the *Bus-Grant signal, BG1*, indicating to the DMA controllers that they may use the bus when it becomes free. This signal is connected to all DMA controllers using a daisy-chain arrangement. Thus, if *DMA controller1* is requesting the bus, it blocks the propagation of the grant signal to other devices. Otherwise, it passes the grant downstream by asserting *BG2*. The current bus master indicates to all devices that it is using the bus by activating another open-collector line called *Bus- Busy*, $\overline{BBSY}$, Hence, after receiving the Bus-Grant signal, a DMA controller waits for Bus-Busy to become inactive, then assumes mastership of the bus. At this time, it activates Bus-Busy to prevent other devices from using the bus at the same time.

The timing diagram in Figure 4.21 shows the sequence of events for the devices in Figure 4.20 as DMA controller 2 requests and acquires bus mastership and later releases the bus. During its tenure as the bus master, it may perform one or more data transfer operations, depending on whether it is operating in the cycle stealing or block mode. After it releases the bus, the processor resumes bus mastership.

The arbiter circuit ensures that only one request is granted at any given time, according to a predefined priority scheme. For example, if there are four bus request lines, BR1 through BR4, a *fixed priority* scheme may be used in which BR1 is given top priority and BR4 is given lowest priority. Alternatively, a *rotating priority* scheme may be used to give all devices an equal chance of being serviced. Rotating priority means that after a request on line BR1 is granted, the priority order becomes 2, 3, 4, and 1.

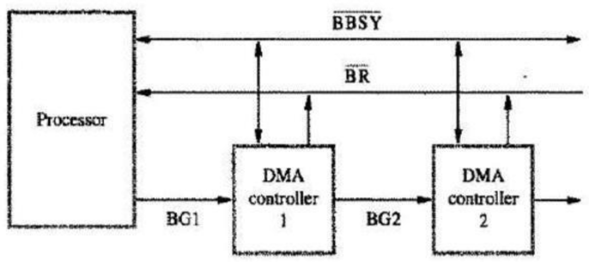
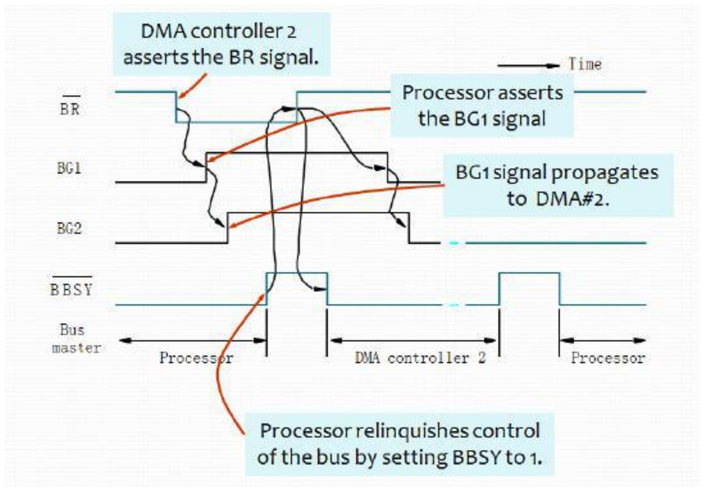


Figure 4.20 A simple arrangement for bus arbitration using a daisy chain.


DMA controller 2 asserts the BR signal.
Time
$\overline{BR}$
Processor asserts the BG1 signal
BG1
BG1 signal propagates to DMA#2.
BG2
$\overline{BBSY}$
Bus master
Processor
DMA controller 2
Processor
Processor relinquishes control of the bus by setting BBSY to 1.

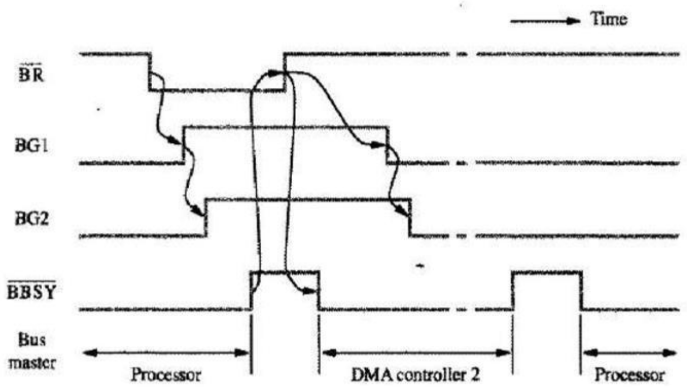


Figure 4.21 Sequence of signals during transfer of bus mastership for the devices in Figure 4.20.

Distributed Arbitration:

Distributed arbitration means that all devices waiting to use the bus have equal responsibility in carrying out the arbitration process, without using central arbiter. A simple method for distributed arbitration is illustrated in Figure 4.22. Each device on the bus is assigned a 4-bit identification number. When one or more devices request the bus, they assert the $\overline{\text{Start-Arbitration}}$ signal and place their 4-bit ID numbers on four open-collector fines, $\overline{\text{ARB0}}$ through ARB3. A winner is selected as a result of the interaction among the signals transmitted over these lines by all contenders. The net outcome is that the code on the four lines represents the request that has the highest ID number.

The drivers are of the open-collector type. Hence, if the input to one driver is equal to one and the input to another driver connected to the same bus line is equal to 0 the bus will be in the low-voltage state. In other words, the connection performs an OR function in which logic 1 wins.

Assume that,

- Device A has the ID 5 and wants to request the bus: Transmits the pattern 0101 on the arbitration lines.
- Device B has the ID 6 and wants to request the bus: Transmits the pattern 0110 on the arbitration lines.
- Pattern that appears on the arbitration lines is the logical OR of the patterns: Pattern 0111 appears on the arbitration lines

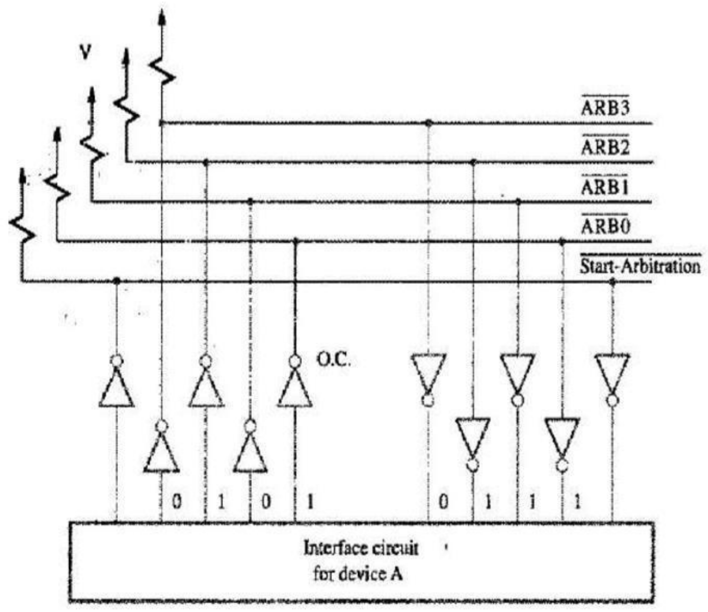


Figure 4.22 A distributed arbitration scheme.

<u>Arbitration process:</u>

- Each device compares the pattern that appears on the arbitration lines to its own ID, starting with MSB.
- If it detects a difference, it transmits 0s on the arbitration lines for that and all lower bit positions.
- Device A compares its ID 5 with a pattern 0101 to pattern 0111.
- It detects a difference at bit position 0, as a result, it transmits a pattern 0100 on the arbitration lines.
- The pattern that appears on the arbitration lines is the logical-OR of 0100 and 0110, which is 0110.
- This pattern is the same as the device ID of B, and hence B has won the arbitration

BUSES

The processor, main memory, and I/O devices can be interconnected by means of a common bus whose primary function is to provide a communications path for the transfer of data.

The *bus lines* used for transferring data may be grouped into three types: *data*, *address*, and *control lines*. The control signals specify whether a read or a write operation is to be performed. Usually, a single an $R/\overline{W}$ line is used. It specifies Read when set to 1 and Write when set to 0. When several operand sizes are possible, such as byte, word, or long word, the required size of data is indicated.

The bus control signals also carry *timing information*. They specify the times at which the processor and the I/O devices may place data on the bus or receive data from the bus. A variety of schemes have been devised for the timing of data transfers over a bus. These can be broadly classified as either *synchronous* or *asynchronous schemes*.

In any data transfer operation, one device plays the role of a *master*. This is the device that initiates data transfers by issuing read or write commands on the bus; hence, it may be called an *initiator*. Normally, the processor acts as the master, but other devices with DMA capability may also become bus masters. The device addressed by the master is referred to as a *slave* or *target*.

SYNCHRONOUS BUS:

In a synchronous bus, all devices derive *timing information* from a common clock line. Equally spaced pulses on this line define equal time intervals. In the simplest form of a synchronous bus, each of these intervals constitutes a bus cycle during which one data transfer can take place. In below figure, the address and data lines in this and subsequent figures are shown as high and low at the same time. This is a common convention indicating that some lines are high and some low, depending on the particular address or data pattern being transmitted. The crossing points indicate the times at which these patterns change. A signal line in an indeterminate or high impedance state is represented by an intermediate level half-way between the low and high signal levels.

Let us consider the sequence of events during an input (read) operation. At time t_0, the master places the device address on the address lines and sends an appropriate command on the control lines. In this case, the command will indicate an input operation and specify the length of the operand to be read, if necessary. Information travels over the bus at a speed determined by its physical and electrical characteristics. The clock pulse width, t_1-t_0, must be longer than the maximum propagation delay between two devices connected to the bus. It also has to be long enough to allow all devices to decode the address and control signals so that the addressed device (the slave) can respond at time t_1. It is important that slaves take no action or place any data on the bus before t_1. The addressed slave places the requested input data on the data lines at time t_1.

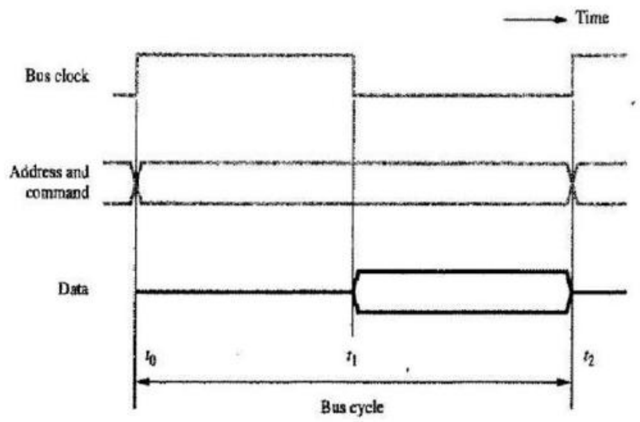


Figure 4.23 Timing of an input transfer on a synchronous bus.

At the end of the clock cycle, at time t_2, the master strobes the data on the data lines into its input buffer. In this context, "strobe" means to capture the values of the data at a given instant and store them into a buffer.

Figure 4.24 gives a more realistic picture of what happens in practice. It shows two views of each signal, except the clock. Because signals take time to travel from one device to another, a given signal transition is seen by different devices at different times. One view shows the signal as seen by the master and the other as seen by the slave.

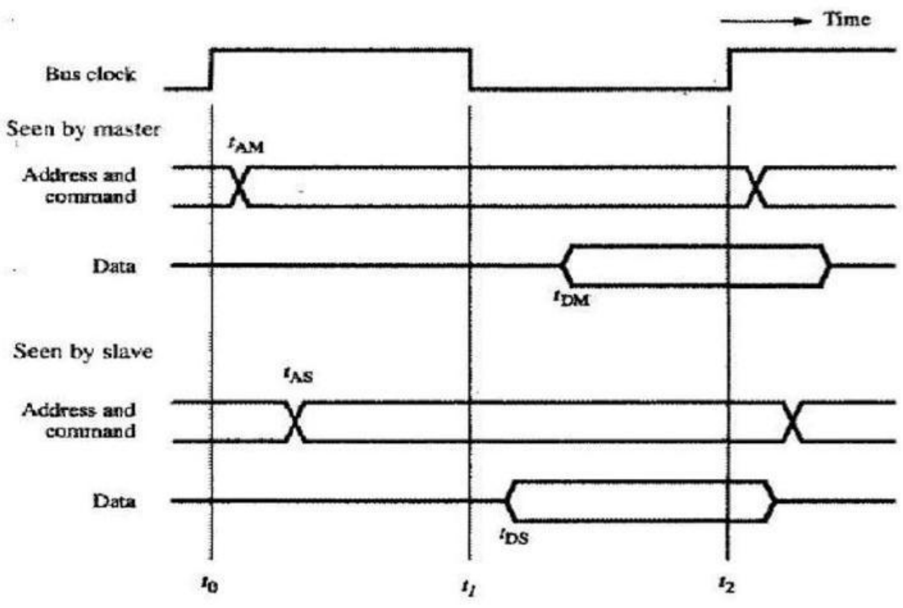


Figure 4.24 A detailed timing diagram for the input transfer of Figure 4.23.

The master sends the address and command signals on the rising edge at the beginning of clock period 1 (t_0). However, these signals do not actually appear on the bus until t_{AM}, largely due to the delay in the bus driver circuit. A while later, at t_{AS}, the signals reach the slave. The slave decodes the address and at t_1, sends the requested data. Here again, the data signals do not appear on the bus until t_{DS}. They travel toward the master and arrive at t_{DM}. At t_2, the master loads the data into its input buffer. Hence the period t_2-t_{DM} is the setup time for the master's input buffer. The data must continue to be valid after t_2 for a period equal to the hold time of that buffer.

Multiple-Cycle Transfers:

Most buses incorporate control signals that represent a response from the device. These signals inform the master that the slave has recognized its address and that it is ready to participate in a data-transfer operation. They also make it possible to adjust the duration of the data-transfer period to suit the needs of the participating devices. To simplify this process, a high-frequency clock signal is used such that a complete data transfer cycle would span several clock cycles. Then, the number of clock cycles involved can vary from one device to another.

An example of this approach is shown in Figure 4.25. During clock cycle 1, the master sends address and command information on the bus, requesting a read operation. The slave receives this information and decodes it. On the following active edge of the clock, that is, at the beginning of clock cycle 2, it makes a decision to respond and begins to access the requested data. The data become ready and are placed on the bus in clock cycle 3. At the same time, the slave asserts a control signal called Slave-ready. The master has been waiting for this signal, strobes the data into its input buffer at the end of clock cycle 3. The bus transfer operation is now complete, and the master may send a new address to start anew transfer in clock cycle 4.

The *Slave-ready signal* is an acknowledgment from the slave to the master, con- firming that valid data have been sent. In the example in Figure 4.25, the slave responds in cycle 3. Another device may respond sooner or later. The Slave-ready signal allows the duration of a bus transfer to change from one device to another. If the addressed device does not respond at all, the master waits for some predefined maximum number of clock cycles, and then aborts the operation. This could be the result of an incorrect address or a device malfunction.

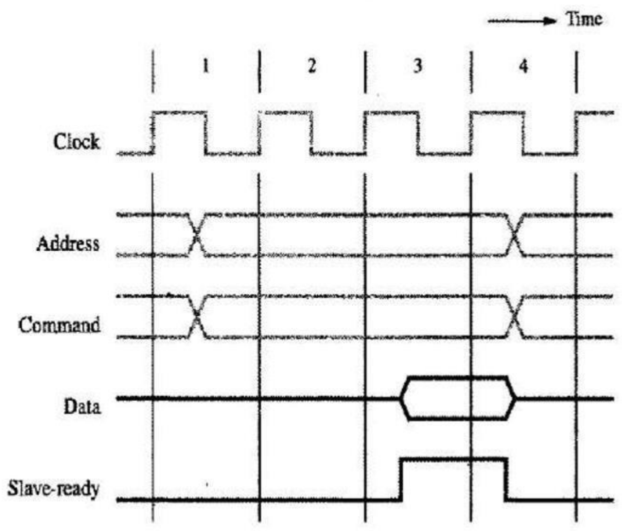


Figure 4.25 An input transfer using multiple clock cycles.

ASYNCHRONOUS BUS:

An alternative scheme for controlling data transfers on the bus is based on the use of a *handshake* between the master and the slave. Here, two timing control lines *Master-ready* and *slave-ready* play a vital role. The first is asserted by the master to indicate that it is ready for a transaction, and the second is a response from the slave.

In principle, a data transfer controlled by a handshake protocol proceeds as follows. The master places the address and command information on the bus. Then it indicates to all devices that it has done so by activating the Master-ready line. This causes all devices on the bus to decode the address. The selected slave performs the required operation and informs the processor it has done so by activating the Slave-ready line. The master waits for Slave-ready to become asserted before it removes its signals from the bus. In the case of a read operation, it also strobes the data into its input buffer

An example of the timing of an input data transfer using the handshake scheme is given in Figure 4.26, which depicts the following sequence of events:

t_0 — The master places the address and command information on the bus, and all devices on the bus begin to decode this information.

t_1 — The master sets the Master-ready line to 1 to inform the I/O devices that the address and command information is ready. The delay t_1- t_0 is intended to allow for any skew that may occur on the bus. Skew occurs when two signals simultaneously transmitted from one source arrive at the destination at different times.

t_2 — The selected slave, having decoded the address and command information, performs the required input operation by placing the data from its data register on the data lines. At the same time, it sets the Slave-ready signal to 1.

t_3 — The Slave-ready signal arrives at the master, indicating that the input data are available on the bus.

T4 — The master removes the address and command information from the bus. The delay between t_3 and t_4 is again intended to allow for bus skew. Erroneous addressing may take place if the address, as seen by some device on the bus, starts to change while the Master-ready signal is still equal to 1.

t_5 - When the device interface receives the 1 to 0 transition of the Master-ready signal, it removes the data and the Slave-ready signal from the bus. This completes the input transfer.

The timing for an output operation, illustrated in Figure 4.27

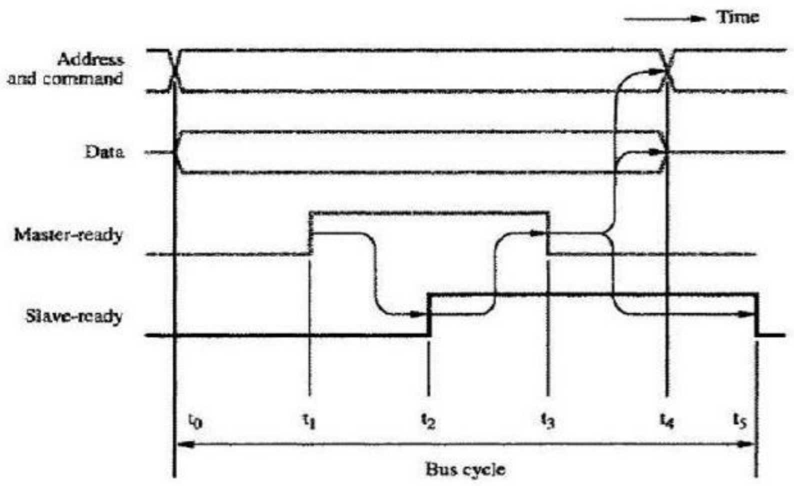


Figure 4.27 Handshake control of data transfer during an output operation.

Difference between Synchronous and Asynchronous Bus:

The choice of a particular design involves trade-offs among factors such as:

- Simplicity of the device interface
- Ability to accommodate device interfaces that introduce different amounts of delay.
- Total time required for a bus transfer.
- Ability to detect errors resulting from addressing a nonexistent device or from an interface malfunction.

The main advantage of the asynchronous bus is that the handshake process eliminates the need for synchronization of the sender and receiver clocks, thus simplifying timing design. Delays, whether introduced by the interface circuits or by propagation over the bus wires, are readily accommodated.

For a synchronous bus, clock circuitry must be designed carefully to ensure proper synchronization, and delays must be kept within strict bounds.

The rate of data transfer on an asynchronous bus controlled by a full handshake is limited by the fact that each transfer involves two round-trip delays.

On synchronous buses, the clock period need only accommodate one end-to-end propagation delay. Hence, faster transfer rates can be achieved. To accommodate a slow device, additional clock cycles are used, as described above. Most of today's high-speed buses use this approach.

INTERFACE CIRCUITS

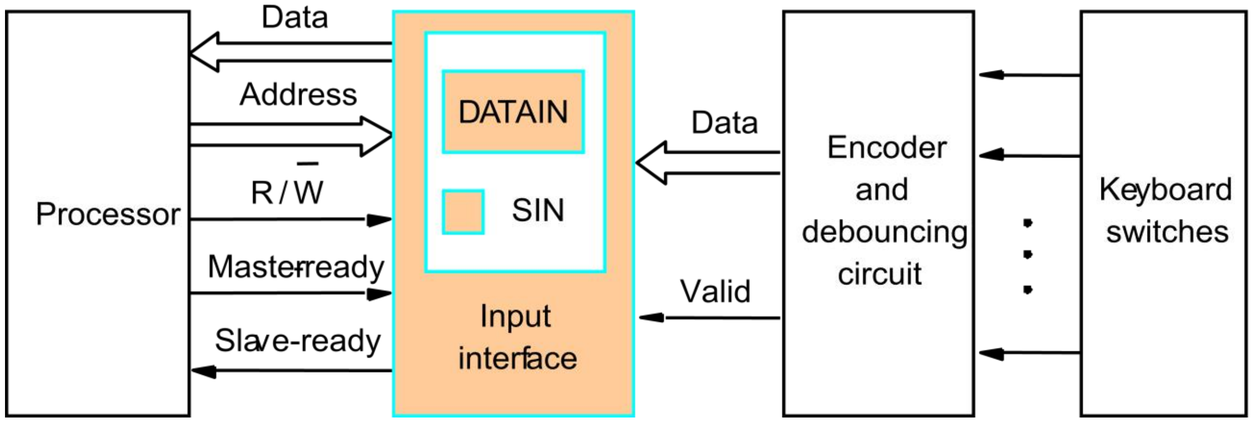
An I/O interface consists of the circuitry required to connect an I/O device to a computer bus. On one side of the interface we have the bus signals for *address*, *data*, and *control*. On the other side we have a *data path* with its associated controls to transfer data between the interface and the I/O device. This side is called a ***port***, and it can be classified as either a *parallel* or a *serial port*.

- ➢ A parallel port transfers data in the form of a number of bits, typically 8 or 16, simultaneously to or from the device.
- ➢ A serial port transmits and receives data one bit at a time.

I/O interface does the following:

1. Provides a storage buffer for at least one word of data (or one byte, in the case of byte-oriented devices)
2. Contains status flags that can be accessed by the processor to determine whether the buffer is full (for input) or empty (for output)
3. Contains address-decoding circuitry to determine when it is being addressed by the processor
4. Generates the appropriate timing signals required by the bus control scheme
5. Performs any format conversion that may be necessary to transfer data between the bus and the I/O device, such as parallel-serial conversion in the case of a serial port

PARALLEL PORT:



Here,

- Keyboard is connected to a processor using a parallel port.
- Processor is 32-bits and uses memory-mapped I/O and the asynchronous bus protocol.
- On the processor side of the interface we have:

- Data lines.
- Address lines
- Control or R/W line.
- Master-ready signal and
- Slave-ready signal

Above diagram shows the hardware components needed for connecting a keyboard to a processor. A typical keyboard consists of mechanical switches that are normally open. When a key is pressed, its switch closes and establishes a path for an electrical signal. This signal is detected by an encoder circuit that generates the ASCII code for the corresponding character. A difficulty with such push-button switches is that the contacts *bounce* when a key is pressed. Bouncing can be eliminated in two ways:

A simple *debouncing* circuit can be included, or a *software approach* can be used. When debouncing is implemented in software, the I/O routine that reads a character from the keyboard wait long enough to ensure that bouncing has subsided. Above Figure illustrates the hardware approach; debouncing circuits are included as a part of the *encoder block*.

The output of the encoder consists of the bits that represent the encoded character and one control signal called *Valid*, which indicates that a key is being pressed. This information is sent to the interface circuit, which contains a data register, DATAIN, and a status flag, SIN.

- When a key is pressed, the V*alid* signal changes from 0 to 1, causing the ASCII code to be loaded into DATAIN and SIN to be set to 1.
- The status flag SIN is cleared to 0 when the processor reads the contents of the DATAIN register.

The interface circuit is connected to an asynchronous bus on which transfers are controlled using the handshake signals *Master-ready* and *Slave-ready*. The third control line, $R/\overline{W}$ distinguishes read and write transfers.

Input Interface:

Figure 4.29 shows a suitable circuit for an input interface. The output lines of the DATAIN register are connected to the data lines of the bus by means of three-state drivers, which are turned on when the processor issues a read instruction with the address that selects this register. The SIN signal is generated by a status flag circuit. This signal is also sent to the bus through a three-state driver. It is connected to bit B0, which means it will appear as bit 0 of the status register. Other bits of this register do not contain valid information. An address decoder is used to select the input interface when the high-order 31 bits of an address correspond to any of the addresses assigned to this interface. Address bit A0 determines whether the status or the data registers is to be read when the Master-ready signal is active. The control handshake is accomplished by activating the Slave-ready signal when either Read-status or Read-data is equal to 1.

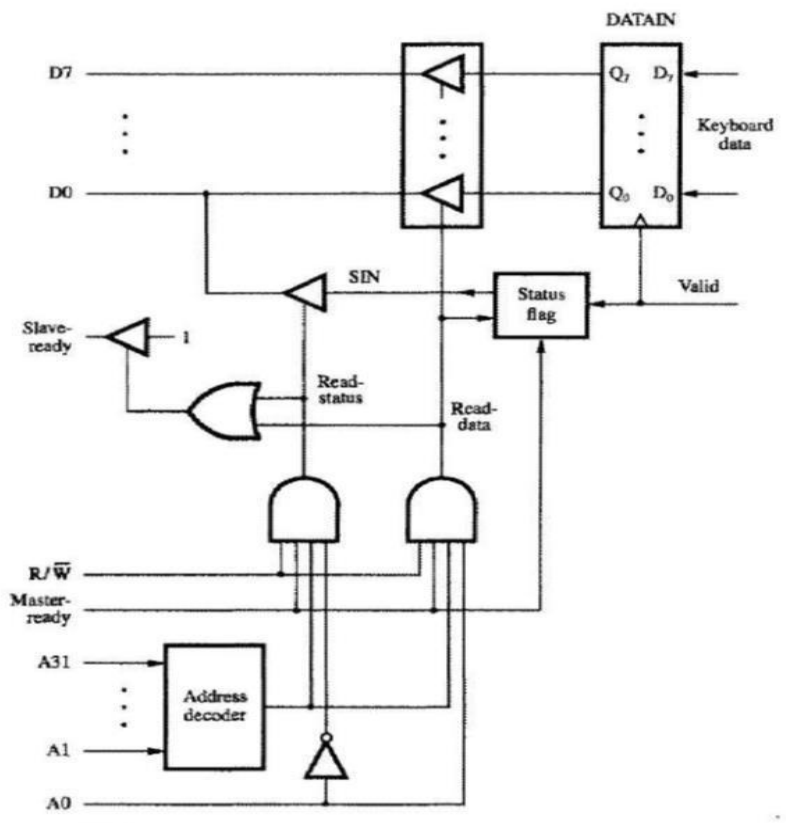


Figure 4.29 Input interface circuit.

Output Interface:

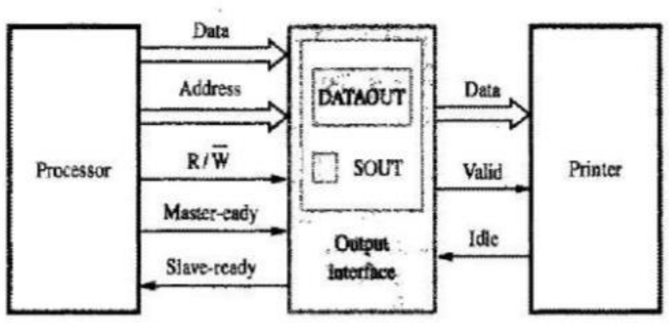


Figure 4.31 Printer to processor connection.

Here,

- Printer is connected to a processor using a parallel port.
- Processor is 32 bits, uses memory-mapped I/O and asynchronous bus protocol.
- On the processor side:

  - Data lines.
  - Address lines
  - Control or R/W line.
  - Master-ready signal and
  - Slave-ready signal.

The printer operates under control of the handshake signals *Valid* and *Idle*. When it is ready to accept a character, the printer asserts its *Idle* signal. The interface circuit can then place a new character on the data lines and activate the *Valid signal*. In response, the printer starts printing the new character and negates the Idle signal, which in turn causes the interface to deactivate the *Valid signal*.

The interface contains a-data register, DATAOUT, and a status flag, SOUT. The SOUT flag is set to 1 when the printer is ready to accept another character, and it is cleared to 0 when a new character is loaded into DATAOUT by the processor.

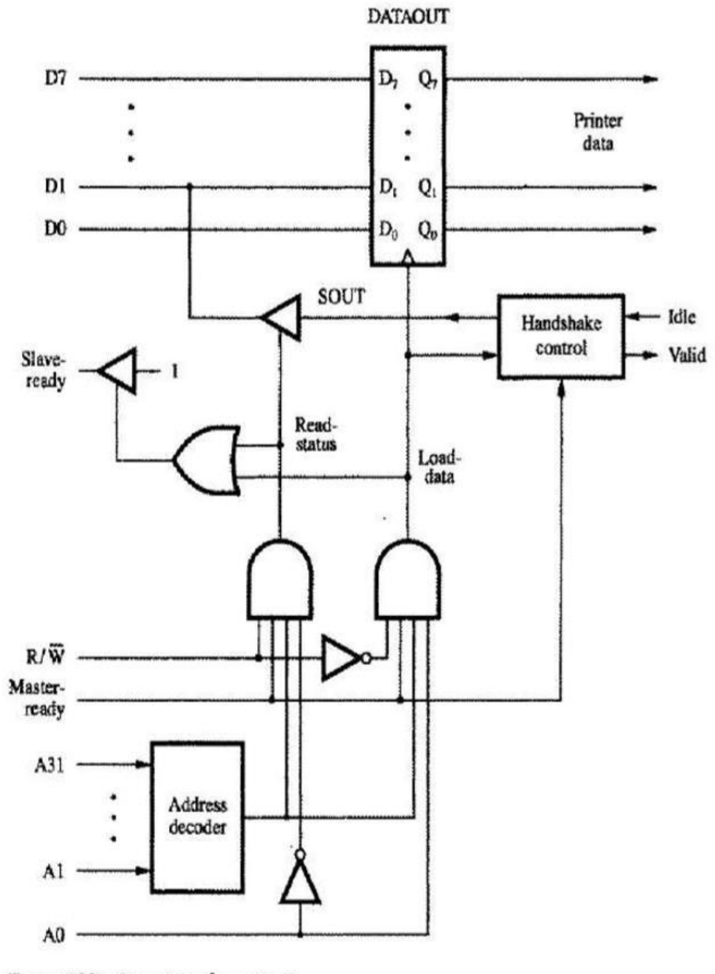


Figure 4.32 Output interface circuit.

The input and output interfaces just described can be combined into a single interface, as shown in below diagram. The overall interface is selected by the high-order 30 bits of the address. Address bits Al and A0 select one of the three addressable locations in the interface, namely, the two data registers and the status register. The status register contains the flags SIN and SOUT in bits 0 and 1, respectively. Labels RS1 and RS0 (for Register Select) are used to denote the two inputs that determine the register being selected.

General-purpose parallel interface circuit that can be configured in a variety of ways is shown below. Data lines P7 through P0 can be used for either input or output purposes. For increased flexibility, the circuit makes it possible for some lines to serve as inputs and some lines to serve as outputs, under program control. The DATAOUT register is connected to these lines via three-state drivers that are controlled by a data direction register, DDR. The processor can write any 8-bit pattern into DDR. For a given bit, if the DDR value is 1, the corresponding data line acts as an output line; otherwise, it acts as an input line.

Two lines, C1 and C2, are provided to control the interaction between the interface circuit and the I/O device it serves. Line C2 is bidirectional to provide several different modes of signaling, including the handshake. The *Ready* and *Accept* lines are the handshake control lines on the processor bus side, and hence would be connected to *Master-ready* and *Slave-ready*. The input signal *My-address* should be connected to the output of an address decoder that recognizes the address assigned to the interface. There are three register select lines, allowing up to eight registers in the interface, input and output data, data direction, and control and status registers for various modes of operation. An interrupt request output, $\overline{\text{INTR}}$, is also provided.

Serial Port:

A serial port is used to connect the processor to I/O devices that require transmission of data one bit at a time. The key feature of an interface circuit for a serial port is that it is capable of communicating in a *bit-serial fashion* on the *device side* and in a *bit-parallel fashion* on the *bus side*. The transformation between the parallel and serial formats is achieved with *shift registers* that have parallel access capability. A block diagram of a typical serial interface is shown in Figure 4.37. It includes the familiar DATAIN and DATAOUT registers. The input shift register accepts bit-serial input from the I/0 device. When all 8 bits of data have been received, the contents of this shift register are loaded in parallel into the DATAIN register. Similarly, output data in the DATAQUT register are loaded into the output shift register, from which the bits are shifted out and sent to the I/O device.

The SIN flag is set to 1 when new data are loaded in DATAIN; it is cleared to 0 when the processor reads the contents of DATAIN. As soon as the data are transferred from the input shift register into the DATAIN register, the shift register can start accepting the next 8-bit character from the I/0 device. The SOUT flag indicates whether the output buffer is available. It is cleared to 0 when the processor writes new data into the DATAOUT register and set to | when data are transferred from DATAOUT into the output shift register.

The double buffering used in the input and output paths is important. A simpler interface could be implemented by turning DATAIN and DATAOUT into shift registers and eliminating the shift registers in Figure 4.37. With the double buffer, the transfer of the second character can begin as soon as the first character is loaded from the shift register into the DATAIN register.

Because it requires fewer wires, serial transmission is convenient for connecting devices that are physically far away from the computer. The speed of transmission, often given as a *bit rate*, depends on the nature of the devices connected.

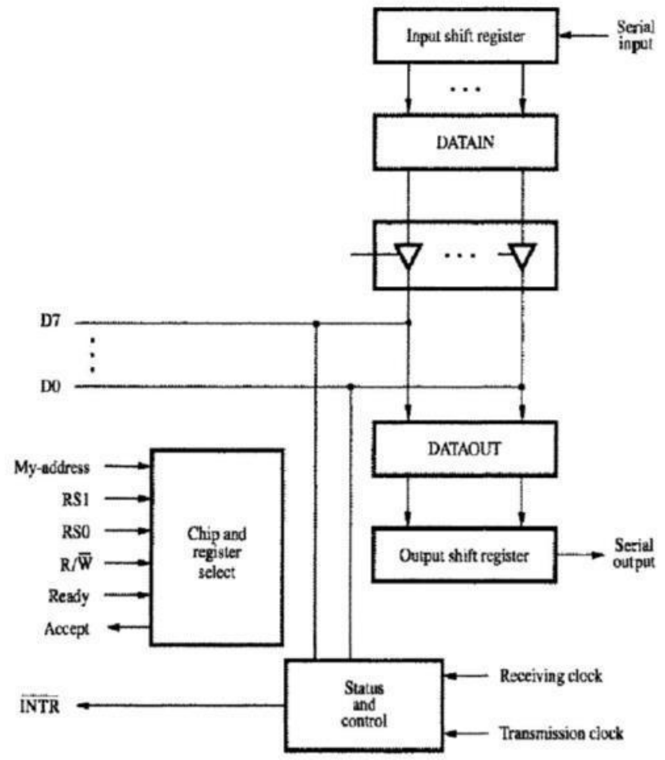


Figure 4.37 A serial interface.

Several standard serial interfaces have been developed:

- Universal Asynchronous Receiver Transmitter (UART) for low-speed serial devices.
- RS-232-C for connection to communication links.

STANDARD I/O INTERFACES

- I/O device is connected to a computer using an interface circuit.
- Do we have to design a different interface for every combination of an I/O device and a computer?
- A practical approach is to develop standard interfaces and protocols.
- A personal computer has:
  - A motherboard which houses the processor chip, main memory and some I/O interfaces.
  - A few connectors into which additional interfaces can be plugged.
- Processor bus is defined by the signals on the processor chip. Devices which require high-speed connection to the processor are connected directly to this bus. Because of electrical reasons only a few devices can be connected directly to the processor bus. Motherboard usually provides another bus that can support more devices. Processor bus and the other bus (called as expansion bus) are interconnected by a circuit called "*bridge*". Devices connected to the expansion bus experience a small delay in data transfers.

- Design of a processor bus is closely tied to the architecture of the processor.
  - No uniform standard can be defined.
- Expansion bus however can have uniform standard defined.

A number of standards have been developed for the expansion bus.

  - Some have evolved by default.

- For example, IBM's Industry Standard Architecture.

Three widely used bus standards:

- PCI (Peripheral Component Interconnect)
- SCSI (Small Computer System Interface)
- USB (Universal Serial Bus)

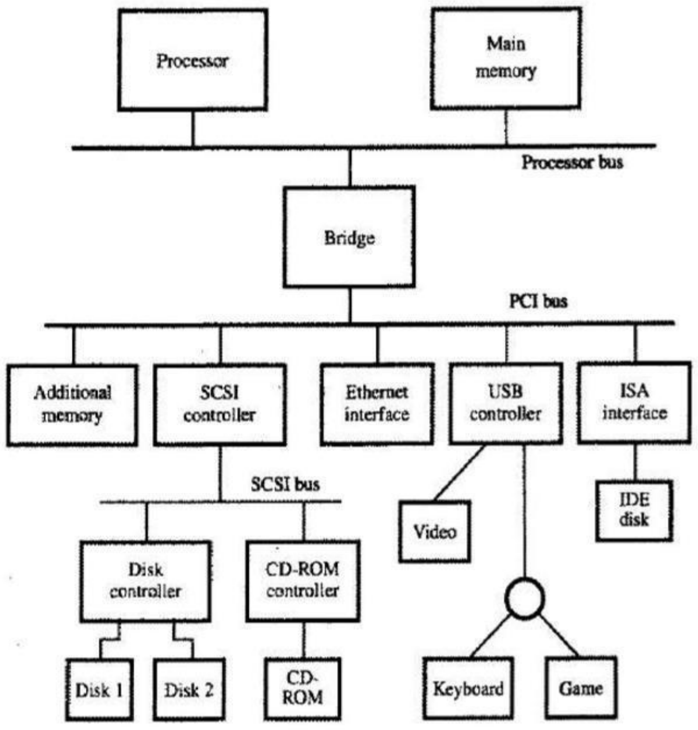


Figure 4.38 An example of a computer system using different interface standards.

PCI BUS:

- *Peripheral Component Interconnect*
- Introduced in 1992
- Low-cost bus
- Processor independent
- Plug-and-play capability
- In today's computers, most memory transfers involve a burst of data rather than just one word. The PCI is designed primarily to support this mode of operation.
- The bus supports three independent address spaces: memory, I/O, and configuration.
- We assumed that the master maintains the address information on the bus until data transfer is completed. But, the address is needed only long enough for the slave to be selected. Thus, the address is needed on the bus for one clock cycle only, freeing the

address lines to be used for sending data in subsequent clock cycles. The result is a significant cost reduction.

- A master is called an initiator in PCI terminology. The addressed device that responds to read and write commands is called a target.

Data Transfer:

Data are transferred between the cache and the main memory in bursts of several words each. The words involved in such a transfer are stored at successive memory locations. When the processor (actually the cache controller) specifies an address and requests a read operation from the main memory, the memory responds by sending a sequence of data words starting at that address. Similarly, during a write operation, the processor sends a memory address followed by a sequence of data words, to be written in successive memory locations starting at that address. The PCI is designed primarily to support this mode of operation. A read or a write operation involving a single word is simply treated as a burst of length one.

The bus supports three independent address spaces: memory, J/O, and configuration. The first two are self-explanatory. The I/O address space is intended for use with processors, such as Pentium, that have a separate I/O address space.

As in below diagram, The PCI bridge provides a separate physical connection for the main memory. For electrical reasons, the bus may be further divided into segments connected via bridges.

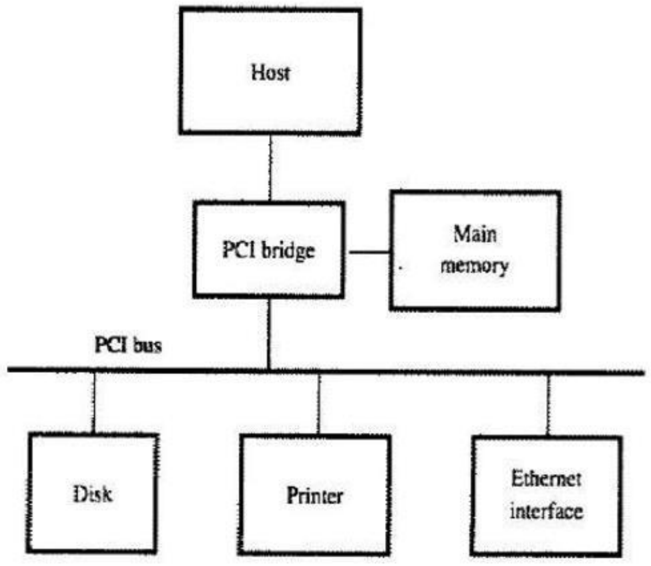
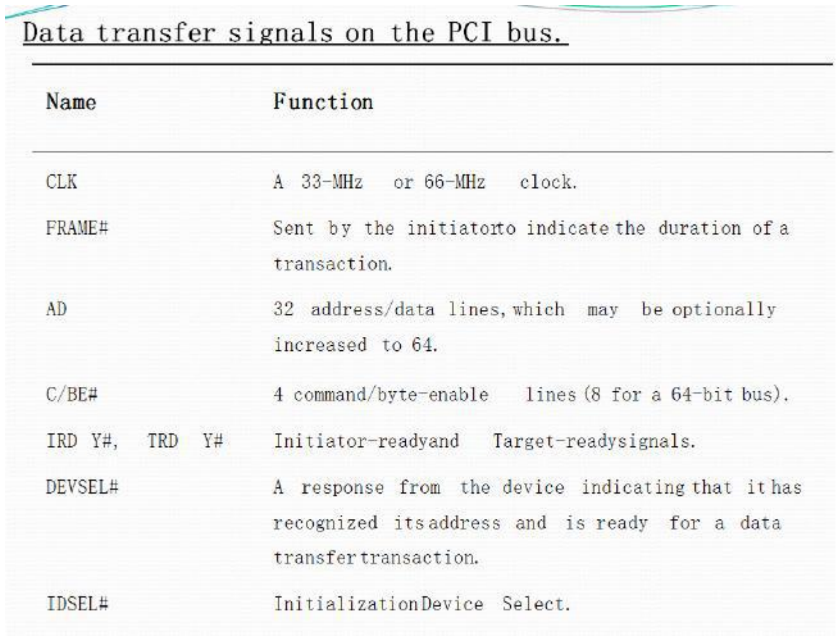


Figure 4.39 Use of a PCI bus in a computer system.



Data transfer signals on the PCI bus.

Name	Function
CLK	A 33-MHz or 66-MHz clock.
FRAME#	Sent by the initiatorto indicate the duration of a transaction.
AD	32 address/data lines, which may be optionally increased to 64.
C/BE#	4 command/byte-enable lines (8 for a 64-bit bus).
IRD Y#, TRD Y#	Initiator-readyand Target-readysignals.
DEVSEL#	A response from the device indicating that it has recognized itsaddress and is ready for a data transfertransaction.
IDSEL#	InitializationDevice Select.

A complete transfer operation on the bus, involving an address and a burst of data, is called a ***transaction***. Individual word transfers within a transaction are called ***phases***.

‘The sequence of events on the bus is shown in Figure 4.40.

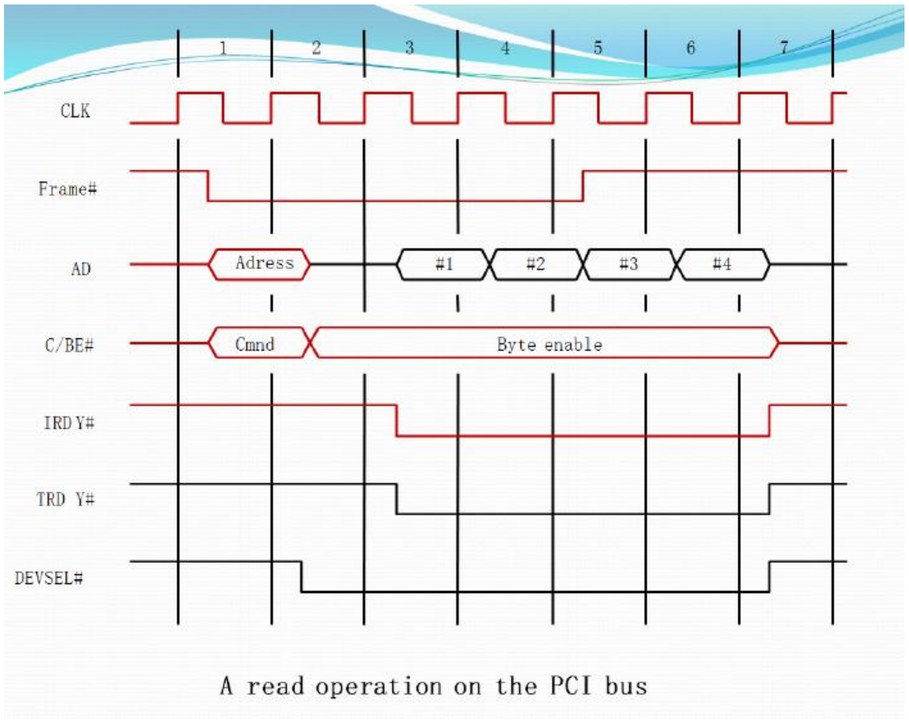


Fig: 4.40 A read operation of the PCI bus

In clock cycle 1, the processor asserts FRAME# to indicate the beginning of a transaction. At the same time, it sends the address on the AD lines and a command on the C/BE# lines. The command indicates that a read operation is requested and that the memory address space is being used.

Clock cycle 2 is used to turn the AD bus lines around. The processor removes the address and disconnects its drivers from the AD lines. The selected target enables its drivers on the AD lines, and fetches the requested data to be placed on the bus during clock cycle 3. It asserts DEVSEL# and maintains it in the asserted state until the end of the transaction.

The C/BE# lines, which were used to send a bus command in clock cycle 1, are used for a different purpose during the rest of the transaction. Each of these four lines is associated with one byte on the AD lines. The initiator sets one or more of the C/BE# lines to indicate which byte lines are to be used for transferring data.

During clock cycle 3, the initiator asserts the initiator ready signal, IRDY#, to indicate that it is ready to receive data. If the target has data ready to send at this time, it asserts target ready, TRDY#, and sends a word of data. The initiator loads the data into its input buffer at the end of the clock cycle. The target sends three more words of data in clock cycles 4 to 6.

The initiator uses the FRAME# signal to indicate the duration of the burst. It negates this signal during the second last word of the transfer. Since it wishes to read four words, the initiator negates FRAME# during clock cycle 5, the cycle in which it receives the third word. After sending the fourth word in clock cycle 6, the target disconnects its drivers and negates DEVSEL# at the beginning of clock cycle 7.

Device Configuration:

- When an I/O device is connected to a computer, several actions are needed to configure both the device and the software that communicates with it.
- PCI incorporates in each I/O device interface a small configuration ROM memory that stores information about that device.
- The configuration ROMs of all devices are accessible in the configuration address space. The PCI initialization software reads these ROMs and determines whether the device is a printer, a keyboard, an Ethernet interface, or a disk controller. It can further learn bout various device options and characteristics.
- Devices are assigned addresses during the initialization process.
- This means that during the bus configuration operation, devices cannot be accessed based on their address, as they have not yet been assigned one.
- Hence, the configuration address space uses a different mechanism. Each device has an input signal called Initialization Device Select, IDSEL#

Electrical characteristics:

- PCI bus has been defined for operation with either a 5 or 3.3 V power supply

SCSI BUS:

- The acronym SCSI stands for Small Computer System Interface.
- It refers to a standard bus defined by the American National Standards Institute (ANSI) under the designation X3.131 .

- In the original specifications of the standard, devices such as disks are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates up to 5 megabytes/s.
- The SCSI bus standard has undergone many revisions, and its data transfer capability has increased very rapidly, almost doubling every two years.
- SCSI-2 and SCSI-3 have been defined, and each has several options.
- Because of various options SCSI connector may have 50, 68 or 80 pins.
- Devices connected to the SCSI bus are not part of the address space of the processor
- The SCSI bus is connected to the processor bus through a SCSI controller. This controller uses DMA to transfer data packets from the main memory to the device, or vice versa.
- A packet may contain a block of data, commands from the processor to the device, or status information about the device.
- A controller connected to a SCSI bus is one of two types – an initiator or a target.
- An initiator has the ability to select a particular target and to send commands specifying the operations to be performed. The disk controller operates as a target. It carries out the commands it receives from the initiator.
- The initiator establishes a logical connection with the intended target.
- Once this connection has been established, it can be suspended and restored as needed to transfer commands and bursts of data.
- While a particular connection is suspended, other device can use the bus to transfer information.
- This ability to overlap data transfer requests is one of the key features of the SCSI bus that leads to its high performance.
- Data transfers on the SCSI bus are always controlled by the target controller.
- To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it.
- Then the controller starts a data transfer operation to receive a command from the initiator.
- Assume that processor needs to read block of data from a disk drive and that data are stored in disk sectors that are not contiguous.
- The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

- The SCSI controller, acting as an initiator, contends for control of the bus.
- When the initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.
- The target starts an output operation (from initiator to target); in response to this, the initiator sends a command specifying the required read operation.
- The target, realizing that it first needs to perform a disk seek operation, sends a message to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
- The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read operation. Then, it reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data to the initiator, the target requests control of the bus. After it wins arbitration, it reselects the initiator controller, thus restoring the suspended connection.
- The target transfers the contents of the data buffer to the initiator and then suspends the connection again
- The target controller sends a command to the disk drive to perform another seek operation. Then, it transfers the contents of the second disk sector to the initiator as before. At the end of this transfer, the logical connection between the two controllers is terminated.
- As the initiator controller receives the data, it stores them into the main memory using the DMA approach.
- The SCSI controller sends as interrupt to the processor to inform it that the requested operation has been completed

Category	Name	Function
Data	– DB(0) to – DB(7)	Data lines: Carry one byte of information during the information transfer phase and iden tify device during arbitration,selection and reselection phases
	– DB(P)	Parity bit for the data bus
Phase	– BSY	Busy: Asserted when the bus is notfree
	–SEL	Selection: Asserted during selection and reselection
Information type	– C/D	Cortrol/Data: Asserted during transfer of control information (command,status or message)
	– MSG	Message:indicates thatthe information being transferred is a message

Main Phases involved:

Arbitration

- A controller requests the bus by asserting BSY and by asserting it's associated data line
- When BSY becomes active, all controllers that are requesting bus examine data lines

Selection

- Controller that won arbitration selects target by asserting SEL and data line of target. After that initiator releases BSY line.
- Target responds by asserting BSY line
- Target controller will have control on the bus from then

Information Transfer

- Handshaking signals are used between initiator and target
- At the end target releases BSY line

Reselection

When a logical connection is suspended and the target is ready to restore it, the target must first gain control of the bus.

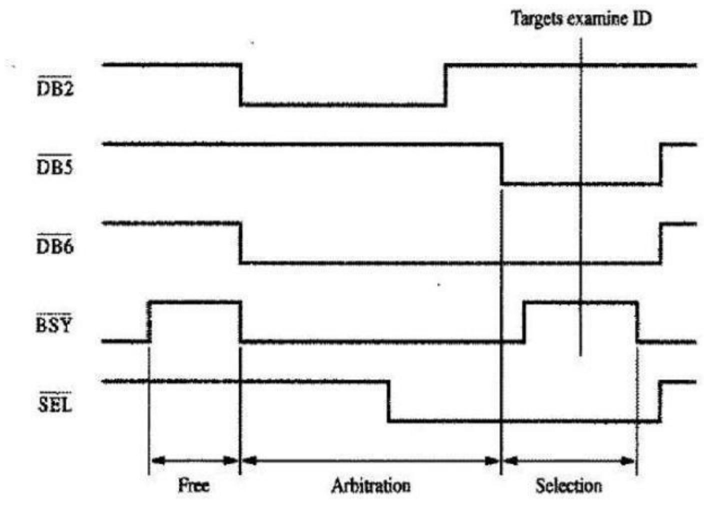


Figure 4.42 Arbitration and selection on the SCSI bus. Device 6 wins arbitration and selects device 2.

UNIVERSAL SERIAL BUS (USB):

- Universal Serial Bus (USB) is an industry standard developed through a collaborative effort of several computer and communication companies, including Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, Nortel Networks, and Philips.
- Speed
    - Low-speed(1.5 Mb/s)
    - Full-speed(12 Mb/s)
    - High-speed(480 Mb/s)
- Port Limitation
- Device Characteristics
- Plug-and-play

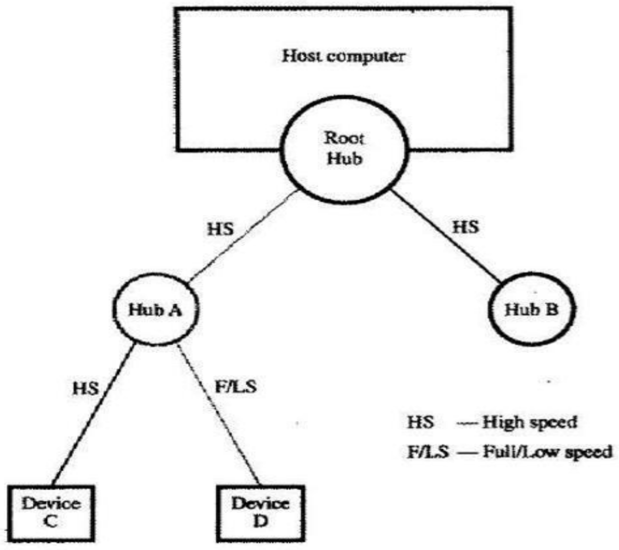


Figure 4.44 Split bus operation.

Universal Serial Bus tree structure:

- To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure as shown in the figure.
- Each node of the tree has a device called a hub, which acts as an intermediate control point between the host and the I/O devices. At the root of the tree, a root hub connects the entire tree to the host computer. The leaves of the tree are the I/O devices being served (for example, keyboard, Internet connection, speaker, or digital TV)
- In normal operation, a hub copies a message that it receives from its upstream connection to all its downstream ports. As a result, a message sent by the host computer is broadcast to all I/O devices, but only the addressed device will respond to that message. However, a message from an I/O device is sent only upstream towards the root of the tree and is not seen by other devices. Hence, the USB enables the host to communicate with the I/O devices, but it does not enable these devices to communicate with each other.

Addressing:

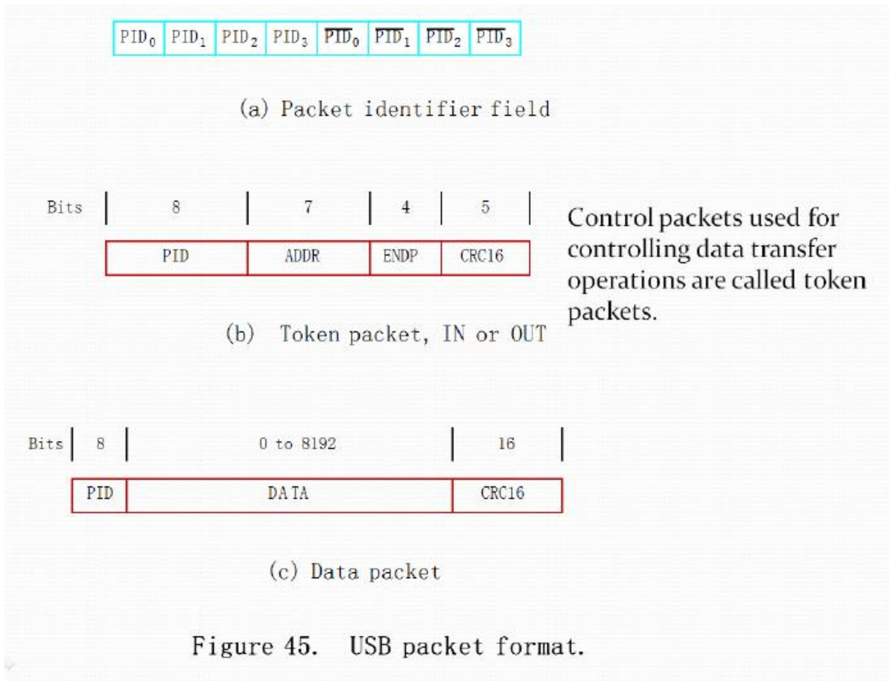
- When a USB is connected to a host computer, its root hub is attached to the processor bus, where it appears as a single device. The host software communicates with individual devices attached to the USB by sending packets of information, which the root hub forwards to the appropriate device in the USB tree.
- Each device on the USB, whether it is a hub or an I/O device, is assigned a 7-bit address. This address is local to the USB tree and is not related in any way to the addresses used on the processor bus.
- A hub may have any number of devices or other hubs connected to it, and addresses are assigned arbitrarily. When a device is first connected to a hub, or when it is powered on,

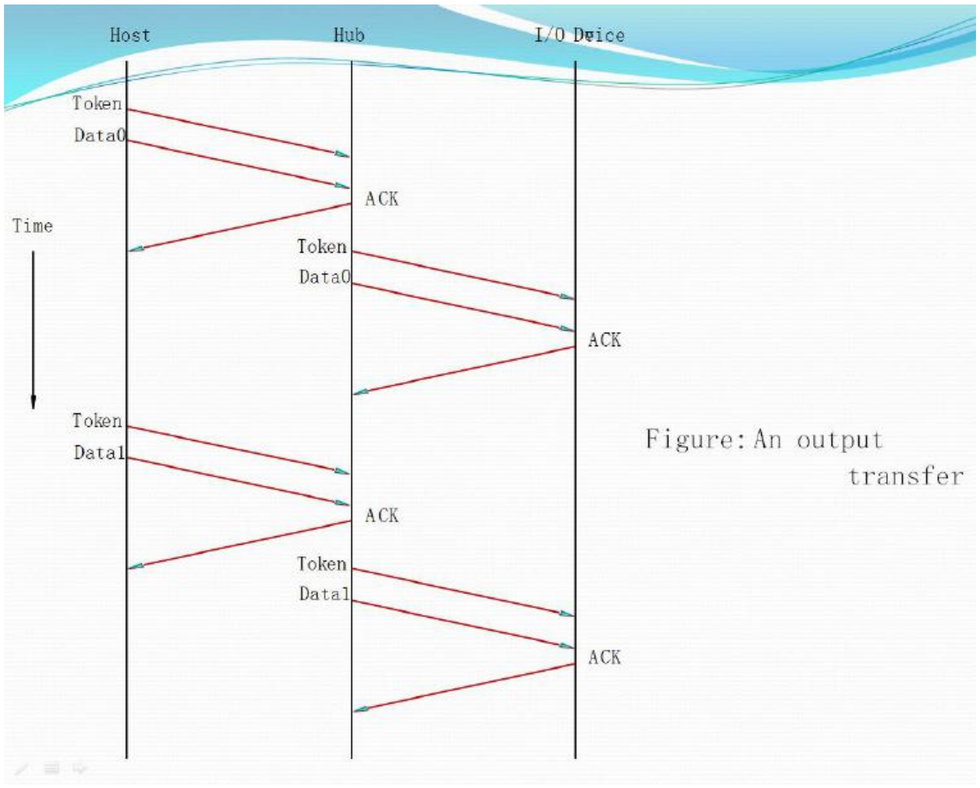
it has the address 0. The hardware of the hub to which this device is connected is capable of detecting that the device has been connected, and it records this fact as part of its own status information. Periodically, the host polls each hub to collect status information and learn about new devices that may have been added or disconnected.

- When the host is informed that a new device has been connected, it uses a sequence of commands to send a reset signal on the corresponding hub port, read information from the device about its capabilities, send configuration information to the device, and assign the device a unique USB address. Once this sequence is completed the device begins normal operation and responds only to the new address.

USB Protocols:

- All information transferred over the USB is organized in packets, where a packet consists of one or more bytes of information. There are many types of packets that perform a variety of control functions.
- The information transferred on the USB can be divided into two broad categories: control and data.
    - Control packets perform such tasks as addressing a device to initiate data transfer, acknowledging that data have been received correctly, or indicating an error.
    - Data packets carry information that is delivered to a device.
- A packet consists of one or more fields containing different kinds of information. The first field of any packet is called the packet identifier, PID, which identifies the type of that packet.
- They are transmitted twice. The first time they are sent with their true values, and the second time with each bit complemented
- The four PID bits identify one of 16 different packet types. Some control packets, such as ACK (Acknowledge), consist only of the PID byte.


PID_0 PID_1 PID_2 PID_3 $\overline{PID_0}$ $\overline{PID_1}$ $\overline{PID_2}$ $\overline{PID_3}$
(a) Packet identifier field
Bits 8 7 4 5
PID ADDR ENDP CRC16
Control packets used for controlling data transfer operations are called token packets.
(b) Token packet, IN or OUT
Bits 8 0 to 8192 16
PID DATA CRC16
(c) Data packet
Figure 45. USB packet format.



Isochronous Traffic on USB:

- One of the key objectives of the USB is to support the transfer of isochronous data.
- Devices that generate or receive isochronous data require a time reference to control the sampling process.
- To provide this reference. Transmission over the USB is divided into frames of equal length.
- A frame is 1ms long for low-and full-speed data.
- The root hub generates a Start of Frame control packet (SOF) precisely once every 1 ms to mark the beginning of a new frame.
- The arrival of an SOF packet at any device constitutes a regular clock signal that the device can use for its own purposes.
- To assist devices that may need longer periods of time, the SOF packet carries an 11-bit frame number.
- Following each SOF packet, the host carries out input and output transfers for isochronous devices.
- This means that each device will have an opportunity for an input or output transfer once every 1 ms.

Electrical Characteristics:

- The cables used for USB connections consist of four wires.
- Two are used to carry power, +5V and Ground.
  - Thus, a hub or an I/O device may be powered directly from the bus, or it may have its own external power connection.
- The other two wires are used to carry data.
- Different signaling schemes are used for different speeds of transmission.
  - At low speed, 1s and 0s are transmitted by sending a high voltage state (5V) on one or the other o the two signal wires. For high-speed links, differential transmission is used.

L	T	P	C
3	0	0	3

(19APC0506) Computer Organization

Course Objectives:

- To learn the fundamentals of computer organization and its relevance to classical and modern problems of computer design
- To make the students understand the structure and behavior of various functional modules of a computer.
- To understand the techniques that computers use to communicate with I/O devices
- To study the concepts of pipelining and the way it can speed up processing.
- To understand the basic characteristics of multiprocessors

Unit I:
Basic Structure of Computer: Computer Types, Functional Units, Basic operational Concepts, Bus Structure, Software, Performance, Multiprocessors and Multicomputer.
Machine Instructions and Programs: Numbers, Arithmetic Operations and Programs, Instructions and Instruction Sequencing, Addressing Modes, Basic Input/output Operations, Stacks and Queues, Subroutines, Additional Instructions.
Unit II:
Arithmetic: Addition and Subtraction of Signed Numbers, Design and Fast Adders, Multiplication of Positive Numbers, Signed-operand Multiplication, Fast Multiplication, Integer Division, Floating-Point Numbers and Operations.
Basic Processing Unit: Fundamental Concepts, Execution of a Complete Instruction, Multiple-Bus Organization, Hardwired Control, Multi-programmed Control.
Unit III:
The Memory System: Basic Concepts, Semiconductor RAM Memories, Read-Only Memories, Speed, Size and Cost, Cache Memories, Performance Considerations, Virtual Memories, Memory Management Requirements, Secondary Storage.
Unit IV:
Input/output Organization: Accessing I/O Devices, Interrupts, Processor Examples, Direct Memory Access, Buses, Interface Circuits, Standard I/O Interfaces.
Unit V:
Pipelining: Basic Concepts, Data Hazards, Instruction Hazards, Influence on Instruction Sets
Large Computer Systems: Forms of Parallel Processing, Array Processors, The Structure of General-Purpose, Interconnection Networks.

Textbook:
1. "Computer Organization", Carl Hamacher, Zvonko Vranesic, Safwat Zaky, McGraw Hill Education, 5th Edition, 2013.

Reference Textbooks:
1. Computer System Architecture, M.Morris Mano, Pearson Education, 3rdEdition.
2. Computer Organization and Architecture, Themes and Variations, Alan Clements, CENGAGE Learning.
3. Computer Organization and Architecture, Smruti Ranjan Sarangi, McGraw HillEducation.
4. Computer Architecture and Organization, John P.Hayes, McGraw Hill Education.

Course Outcomes:

- Ability to use memory and I/O devices effectively
- Able to explore the hardware requirements for cache memory and virtualmemory
- Ability to design algorithms to exploit pipelining and multiprocessors

Unit V:

Pipelining: Basic Concepts, Data Hazards, Instruction Hazards, Influence on Instruction Sets

Large Computer Systems: Forms of Parallel Processing, Array Processors, The Structure of General-Purpose, Interconnection Networks.

Pipelining

Basic Concepts

Pipelining is a particularly effective way of organizing concurrent activity in a computer system. It is frequently encountered in manufacturing plants, cars.

Consider how the idea of pipelining can be used in a computer. The processor executes a program by fetching and executing instructions, one after the other. Let F_i and E_i refer to the fetch and execute steps for instruction I_i. An execution of a program consists of a sequence of fetch and execute steps, as shown in Figure 8.1a.

Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure 8.1b. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction.

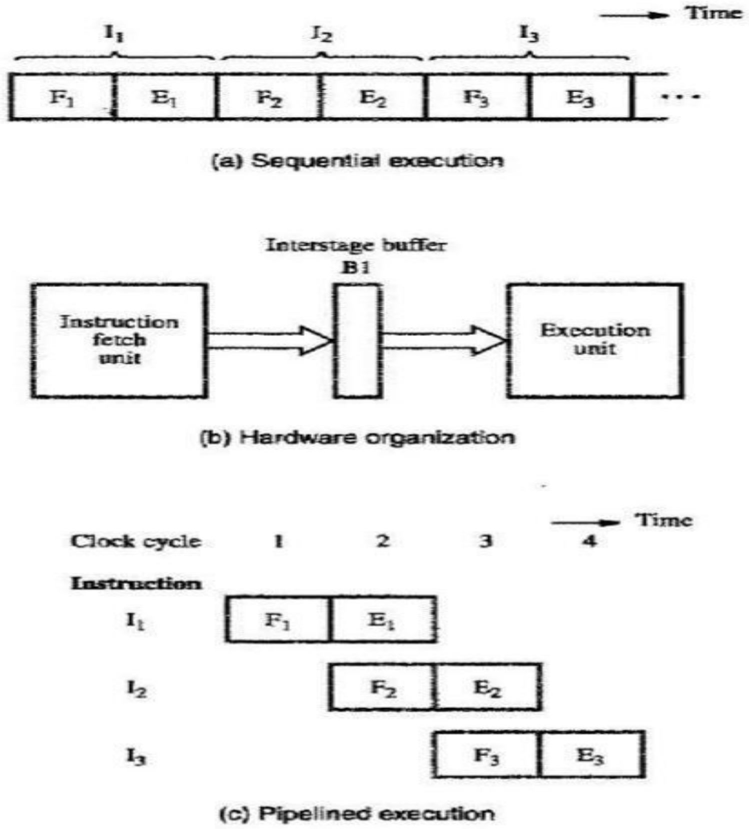


Figure 8.1 Basic idea of instruction pipelining.

The computer is controlled by a clock whose period is such that the fetch and execute steps of any instruction can each be completed in one clock cycle. Operation of the computer proceeds as in Figure 8.1c, In the first clock cycle, the fetch unit fetches an instruction 1_1 (step F_1) and stores it in buffer B1 at the end of the clock cycle. In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I_2 (step F_2). Meanwhile, the execution unit performs the operation specified by instruction $I_{1,}$ which is available to it in buffer B_1 (step E_1). By the end of the second clock cycle, the execution of instruction I_1 is completed and instruction I_2 is available. Instruction I_2 is stored in B1, replacing I_1 which is no longer needed. Step E_2 is performed by the execution unit during the third clock cycle, while instruction I_3 is being fetched by the fetch unit. In this manner, both the fetch and execute units are kept busy all the time.

The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:

F ***Fetch: read the instruction from the memory.***

D ***Decode: decode the instruction and fetch the source operand(s).***

E ***Execute: perform the operation specified by the instruction.***

W ***Write: store the result in the destination location.***

For example, during clock cycle 4, the information in the buffers is as follows:

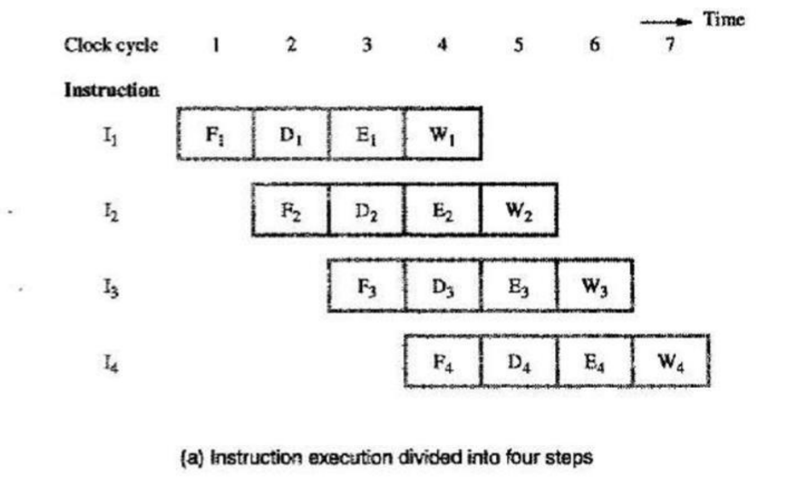
- Buffer B1 holds instruction I_3 which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.
- Buffer B2 holds both the source operands for instruction I_2 and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction I_2, (step W2). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.
- Buffer B3 holds the results produced by the execution unit and the destination information for instruction I_1.

Role of Cache Memory:

Pipelining is most effective in improving performance if the tasks being performed in different stages require about the same amount of time.

This consideration is particularly important for the instruction fetch step, which is assigned one clock period in Figure 8.2a. The clock cycle has to be equal to or greater than the time needed to complete a fetch operation. However, the access time of the main memory may be as much as ten times greater than the time needed to perform basic pipeline stage operations inside the processor.

The use of cache memories solves the memory access problem. In particular, when a cache is included on the same chip as the processor, access time to the cache is usually the same as the time needed to perform other basic operations inside the processor. This makes it possible to divide instruction fetching and processing into steps that are more or less equal in duration. Each of these steps is performed by a different pipeline stage, and the clock period is chosen to correspond to the longest one.


Time
Clock cycle 1 2 3 4 5 6 7
Instruction
I1 F1 D1 E1 W1
I2 F2 D2 E2 W2
I3 F3 D3 E3 W3
I4 F4 D4 E4 W4
(a) Instruction execution divided into four steps

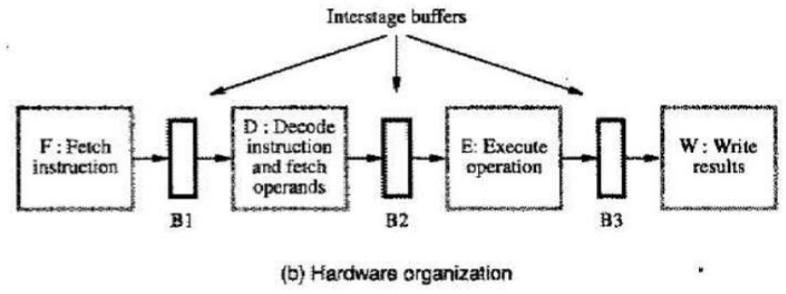


Figure 8.2 A 4-stage pipeline.

PIPELINE PERFORMANCE:

For a variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted.

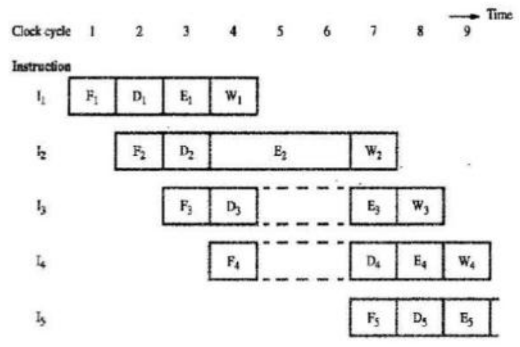
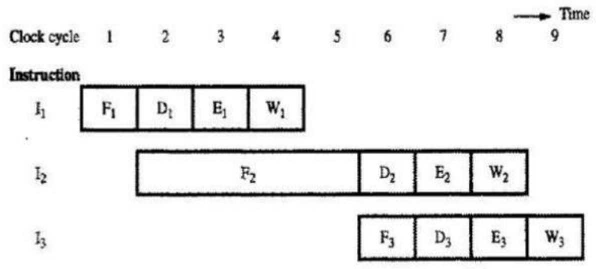


Figure 8.3 Effect of an execution operation taking more than one clock cycle.

Pipelined operation in Figure 8.3 is said to have been *stalled* for two clock cycles. Normal pipelined operation resumes in cycle 7.

- Any condition that causes the pipeline to stall is called a *hazard*.

- A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result some operation has to be delayed, and the pipeline stalls.
- The pipeline may also be stalled because of a delay in the availability of an instruction. For example, this may be a result of a miss in the cache, requiring the instruction to be fetched from the main memory. Such hazards are often called *control hazards* or *instruction hazards*. The effect of a cache miss on pipelined operation is illustrated in Figure 8.4. Instruction I; is fetched from the cache in cycle 1, and its execution proceeds normally. However, the fetch operation for instruction I, which is started in cycle 2, results in a cache miss. The instruction fetch unit must now suspend any further fetch re-Quests and wait for 1, to arrive. We assume that instruction 1, is received and loaded into buffer B1 at the end of cycle 5. The pipeline resumes its normal operation at that point.



(a) Instruction execution steps in successive clock cycles

Clock cycle	1	2	3	4	5	6	7	8	9
Stage									
F: Fetch	F_1	F_2	F_2	F_2	F_2	F_3			
D: Decode		D_1	idle	idle	idle	D_2	D_3		
E: Execute			E_1	idle	idle	idle	E_2	E_3	
W: Write				W_1	idle	idle	idle	W_2	W_3

(b) Function performed by each processor stage in successive clock cycles

Figure 8.4 Pipeline stall caused by a cache miss in F2.

A third type of hazard that may be encountered in pipelined operation is known as a *structural hazard*. This is the situation when two instructions require the use of a given hardware resource at the same time. The most common case in which this hazard may arise is in access to memory.

An example of a structural hazard is shown in Figure 8,5. This figure shows how the load instruction

Load X(R1), R2

can be accommodated in our example 4-stage pipeline. The memory address, X+-[R1], is computed in step E, in cycle 4, then memory access takes place in cycle 5. The operand read from memory is written into register R2 in cycle 6. This means that the execution step of this instruction takes two clock cycles (cycles 4 and 5). It causes the pipeline to stall for one cycle, because both instructions I, and I; require access to the register file in cycle 6. Even though the instructions and their data are all available, the pipeline is stalled because one hardware resource, the register file, cannot handle two operations at once.

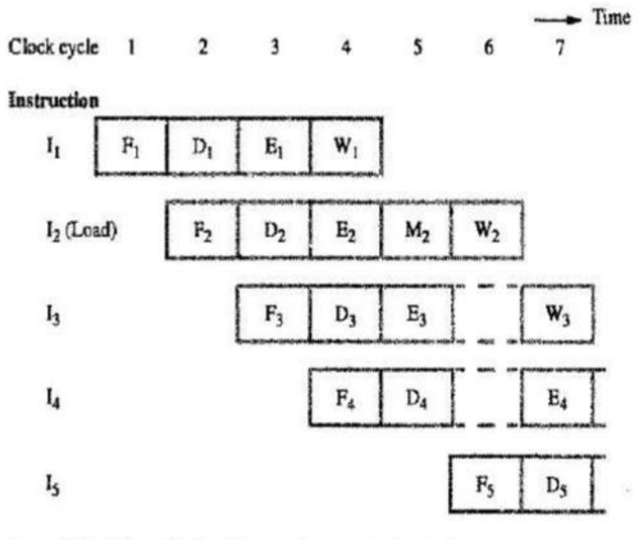


Figure 8.5 Effect of a Load instruction on pipeline timing.

DATA HAZARDS

A *data hazard* is a situation, in which the pipeline is stalled because the data to be operated on are delayed for some reason.

Consider a program that contains two instructions, I_1 followed by I_2. When this program is executed in a pipeline, the execution of I_2 can begin before the execution of I_1 is completed. This means that the results generated by I_1 may not be available for use by I_2. We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially. The potential for obtaining incorrect results when operations are performed concurrently can be demonstrated by a simple example. Assume that A=5, and consider the following two operations:

$$A \leftarrow 3+A$$
$$B \leftarrow 4*A$$

When these operations are performed in the order given, the result is B = 32. But if they are performed concurrently, the value of A used in computing B would be the original value, 5, leading to an incorrect result. If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because the data used in the second instruction depend on the result of the first instruction. On the other hand, the two operations

$$A \leftarrow 5*C$$
$$B \leftarrow 20+C$$

can be performed concurrently, because these operations are independent.

When two operations depend on each other, they must be performed sequentially in the correct order.

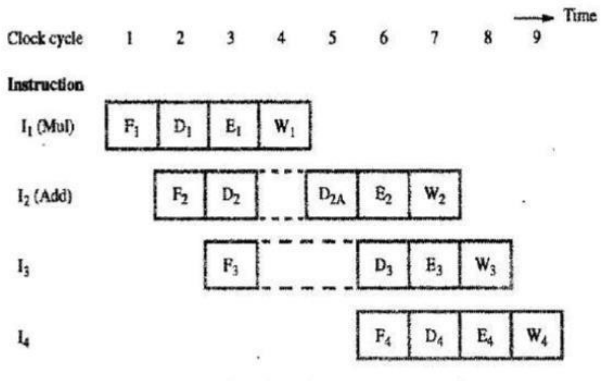


Figure 8.6 Pipeline stalled by data dependency between D_2 and W_1.

The *data dependency* arises when the destination of one instruction is used as a source in the next instruction. For example, the two instructions

Mul R2, R3, R4
Add R5, R4, R6

give rise to a data dependency. The result of the multiply instruction is placed into register R4, which in turn is one of the two source operands of the Add instruction. Assuming that the multiply operation takes one clock cycle to complete, execution would proceed as shown in Figure 8.6. As the Decode unit decodes the Add instruction in cycle 3, it realizes that R4 is used as a source operand. Hence, the D step of that instruction cannot be completed until the W step of the multiply instruction has been completed. Completion of step D_2 must be delayed to clock cycle 5, and is shown as step D_{2A}, in the figure. Instruction I_3 is fetched in cycle 3, but its decoding must be delayed because step D_3 cannot precede D2. Hence, pipelined execution is stalled for two cycles.

OPERAND FORWARDING:

Figure 8.7a shows a part of the processor datapath involving the ALU and the register file. SRC1, SRC2, and RSLT registers constitute the interstage buffers needed for pipelined operation, as illustrated in Figure 8.7b.

The data forwarding mechanism is provided by the blue connection lines. The two multiplexers connected at the inputs to the ALU allow the data on the destination bus to be selected instead of the contents of either the SRC1 or SRC2 register.

After decoding instruction I_2 and detecting the data dependency, a decision is made to use data forwarding. The operand not involved in the dependency, register R2, is read and loaded in register SRC1 in clock cycle 3. In the next clock cycle, the product produced by instruction I_1 is

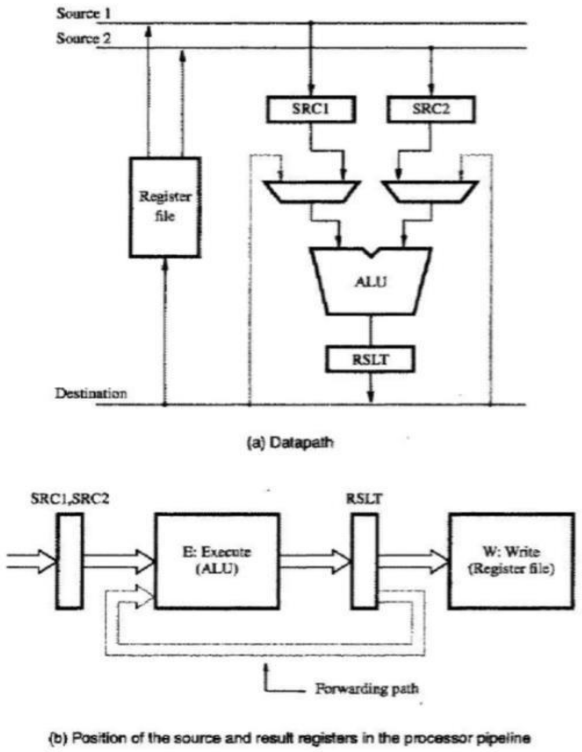


Figure 8.7 Operand forwarding in a pipelined processor.

available in register RSLT, and because of the forwarding connection, it can be used in step E_2. Hence, execution of I_2, proceeds without interruption.

HANDLING DATA HAZARDS IN SOFTWARE:

The control hardware delays reading register R4 until cycle 5, thus introducing a 2-cycle stall unless operand forwarding is used. An alternative approach is to leave the task of detecting data dependencies and dealing with them to the software. In this case, the compiler can introduce the two-cycle delay needed between instructions I_1 and I_2 by inserting NOP (No-operation) instructions, as follows:

```
I1:   Mul R2,R3,R4
      NOP
      NOP
I2:   Add R5,R4,R6
```

If the responsibility for detecting such dependencies is left entirely to the software, the compiler must insert the **NOP** instructions to obtain a correct result. This possibility illustrates the close link between the compiler and the hardware. A particular feature can be either implemented in hardware or left to the compiler. Leaving tasks such as inserting NOP instructions to the compiler leads to simpler hardware. Being aware of the need for a delay, the compiler can attempt to reorder instructions to perform useful tasks in the NOP slots, and thus achieve better performance. On the other hand, the insertion of NOP instructions leads to larger code size. Also, it is often the case that a given processor architecture has several hardware implementations, offering different features. NOP instructions inserted to satisfy the requirements of one implementation may not be needed and, hence, would lead to reduced performance on a different implementation.

INSTRUCTION HAZARDS

UNCONDITIONAL BRANCHES:

Figure 8.8 shows a sequence of instructions being executed in a two-stage pipeline. instructions I_1 to I_3 are stored at successive memory addresses, and I_2 is a branch instruction. Let the branch target be instruction I_k. In clock cycle 3, the fetch operation for instruction I_3 is in progress at the same time that the branch instruction is being decoded and the target address computed. In clock cycle 4, the processor must discard is, which has been incorrectly fetched, and fetch instruction I_k. In the meantime, the hardware unit responsible for the Execute (E) step must be told to do nothing during that clock period. Thus, the pipeline is stalled for one clock cycle.

The time lost as a result of a branch instruction is often referred to as the branch penalty. In Figure 8.8, the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher. For example, Figure 8.9a shows the effect of a branch instruction on a four-stage pipeline. We have assumed that the branch address is computed in step E_2. Instructions I_3 and I_4

must be discarded, and the target instruction, I_k, is fetched in clock cycle 5. Thus, the branch penalty is two clock cycles.

Reducing the branch penalty requires the branch address to be computed earlier in the pipeline. Typically, the instruction fetch unit has dedicated hardware to identify a branch instruction and compute the branch target address as quickly as possible after an instruction is fetched. With this additional hardware, both of these tasks can be performed in step D2, leading to the sequence of events shown in Figure 8.9b. In this case, the branch penalty is only one clock cycle.

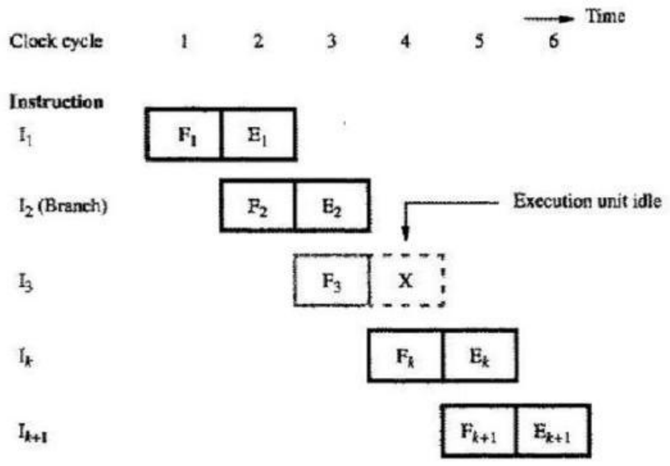
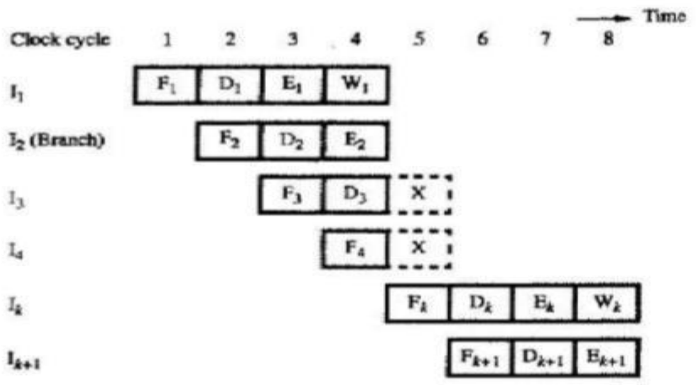


Figure 8.8 An idle cycle caused by a branch instruction.


Time
Clock cycle 1 2 3 4 5 6 7 8
I_1 F_1 D_1 E_1 W_1
I_2 (Branch) F_2 D_2 E_2
I_3 F_3 D_3 X
I_4 F_4 X
I_k F_k D_k E_k W_k
I_{k+1} F_{k+1} D_{k+1} E_{k+1}
(a) Branch address computed in Execute stage

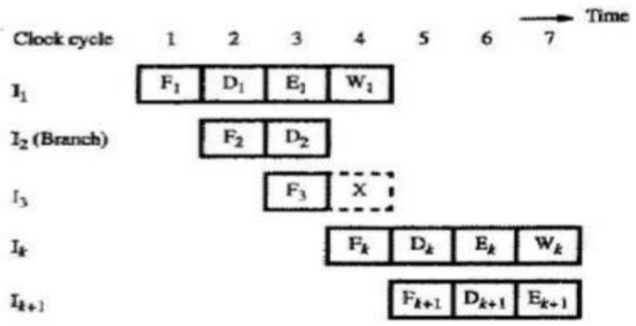

Time
Clock cycle 1 2 3 4 5 6 7
I_1 F_1 D_1 E_1 W_1
I_2 (Branch) F_2 D_2
I_3 F_3 X
I_k F_k D_k E_k W_k
I_{k+1} F_{k+1} D_{k+1} E_{k+1}
(b) Branch address computed in Decode stage

Figure 8.9 Branch timing.

Instruction Queue und Prefetching:

Either a cache miss or a branch instruction stalls the pipeline for one or more clock cycles. To reduce the effect of these interruptions, many processors employ sophisticated fetch units that can fetch instructions before they are needed and put them in a queue. Typically, the instruction queue can store several instructions. A separate unit, which we call the dispatch unit, takes instructions from the front of the queue and sends them to the execution unit. The dispatch unit also performs the decoding function.

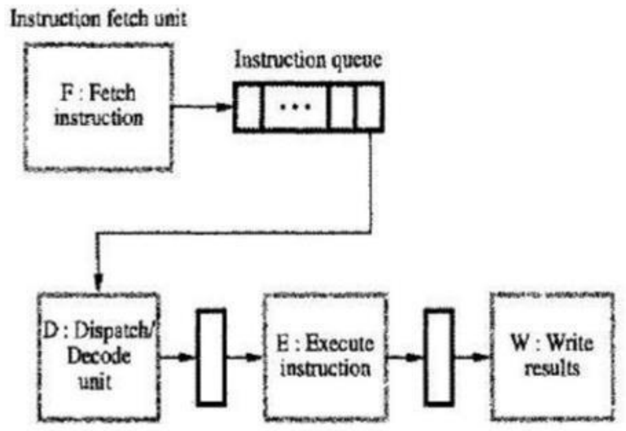


Figure 8.10 Use of an instruction queue in the hardware organization of Figure 8.2*b*.

To be effective, the fetch unit must have sufficient decoding and processing capability to recognize and execute branch instructions. It attempts to keep the instruction queue filled at all times to reduce the impact of occasional delays when fetching instructions. When the pipeline stalls because of a data hazard, for example, the dispatch unit is not able to issue instructions from the instruction queue. However, the fetch unit continues to fetch instructions and add them to the queue. Conversely, if there is a delay in fetching instructions because of a branch or a cache miss, the dispatch unit continues to issue instructions from the instruction queue.

Figure 8.11 illustrates how the queue length changes and how it affects the relationship between different pipeline stages. We have assumed that initially the queue contains one instruction. Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one. Hence, the queue length remains the same for the first four clock cycles. (There is both an F and a D step in each of these cycles.) Suppose that instruction |, introduces a 2-cycle stall. Since space is available in the queue, the fetch unit continues to fetch instructions and the queue length rises to 3 in clock cycle 6.

Instruction I_5 is a branch instruction. Its target instruction, I_k is fetched in cycle 7, and instruction I_6 is discarded, The branch instruction would normally cause a stall in cycle 7 as a result of discarding instruction I_6. Instead, instruction I_4 is dispatched from the queue to the decoding stage. After discarding I_6, the queue length drops to 1 in cycle 8. The queue length will be at this value until another stall is encountered.

Now observe the sequence of instruction completions in Figure 8.11. Instructions I_1, I_2, I_3, I_4 and I_k, complete execution in successive clock cycles.

The instruction fetch unit has executed the branch instruction (by computing the branch address) concurrently with the execution of other instructions. This technique is referred to as **branch folding.**

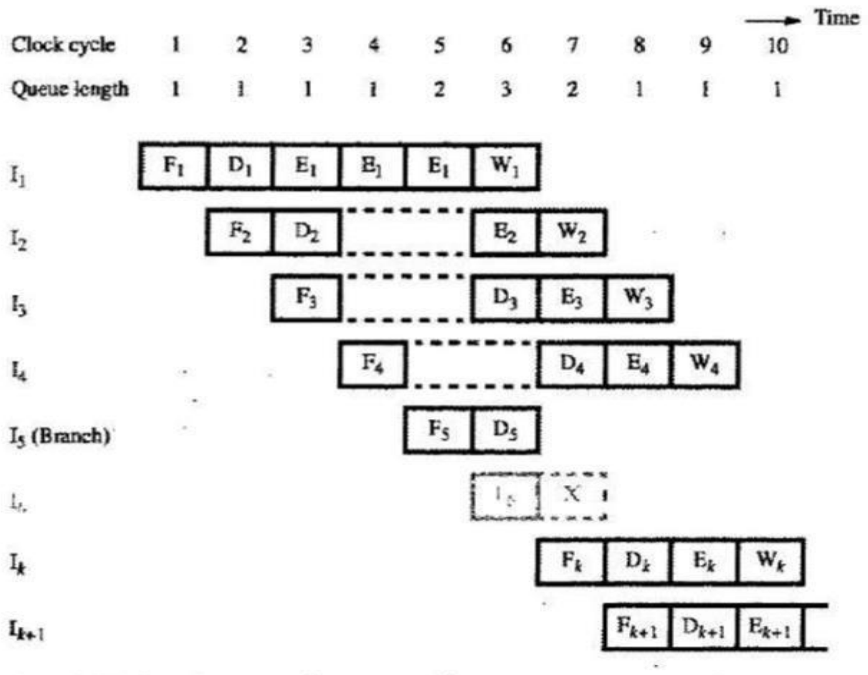


Figure 8.11 Branch timing in the presence of an instruction queue. Branch target address is computed in the D stage.

Note that branch folding occurs only if at the time a branch instruction is encountered, at least one instruction is available in the queue other than the branch instruction.

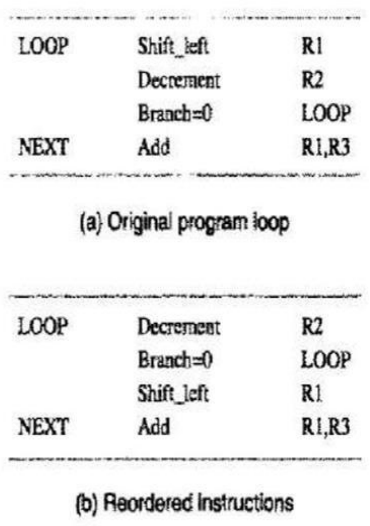
CONDITIONAL BRANCHES AND BRANCH PREDICTION:

Branch instructions can be handled in several ways to reduce their negative impact on the rate of execution of instructions.

Delayed Branch:

A technique called *delayed branching* can minimize the penalty incurred as a result of conditional branch instructions. The instructions in the delay slots ate always fetched instructions. The instructions in the delay slots ate always fetched. Therefore, we would like to arrange for them to be fully executed whether or not the branch is taken. The objective is to be able to place useful instructions in these slots. If no useful instructions can be placed in the delay slots, these slots must be filled with NOP instructions.

Consider the instruction sequence given in Figure 8.12a. Register R2 is used as a counter to determine the number of times the contents of register R1 are shifted left. For a processor with one delay slot, the instructions can be reordered as shown in Figure 8.12b. The shift instruction is fetched while the branch instruction is being executed. After evaluating the branch condition, the processor fetches the instruction at LOOP or at NEXT, depending on whether the branch condition is true or false, respectively. In either case, it completes execution of the shift instruction. The sequence of events during the last two passes in the loop is illustrated in Figure 8.13. Pipelined operation is not interrupted at any time, and there are no idle cycles. Logically, the program is executed as if the branch instruction were placed after the shift instruction. That is, branching takes place one instruction later than where the branch instruction appears in the instruction sequence in the memory, hence the name "delayed branch."



```
LOOP    Shift_left    R1
        Decrement     R2
        Branch=0      LOOP
NEXT    Add           R1,R3
```

(a) Original program loop

```
LOOP    Decrement     R2
        Branch=0      LOOP
        Shift_left    R1
NEXT    Add           R1,R3
```

(b) Reordered instructions

Figure 8.12 Reordering of instructions for a delayed branch.

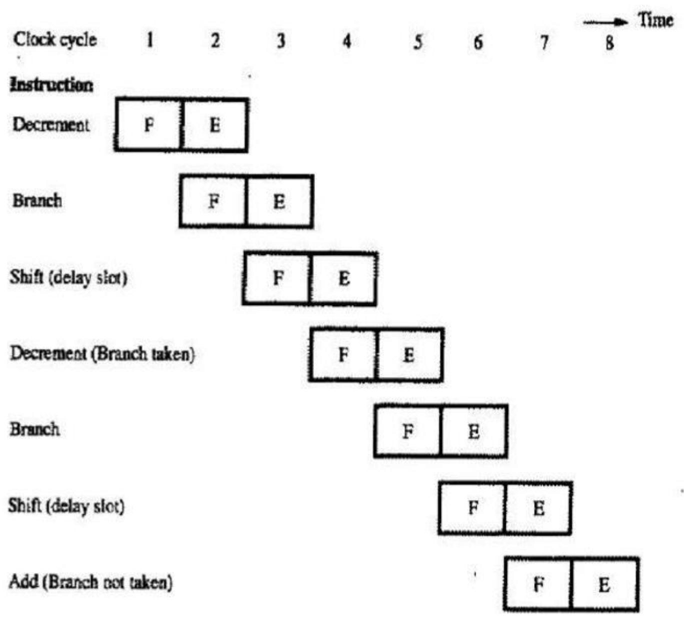


Figure 8.13 Execution timing showing the delay slot being filled during the last two passes through the loop in Figure 8.12*b*.

The effectiveness of the delayed branch approach depends on how often it is possible to reorder instructions.

Branch Prediction:

Another technique for reducing the branch penalty associated with conditional branches is to attempt to predict whether or not a particular branch will be taken. The simplest form of branch prediction is to assume that the branch will not take place and to continue to fetch instructions in sequential address order. Until the branch condition is evaluated, instruction execution along the predicted path must be done on a speculative basis. Speculative execution means that instructions are executed before the processor is certain that they are in the correct execution sequence. Hence, care must be taken that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed. If the branch decision indicates otherwise, the instructions and all their associated data in the execution units must be purged, and the correct instructions fetched and executed.

An incorrectly predicted branch is illustrated in Figure 8.14 for a four-stage pipeline.

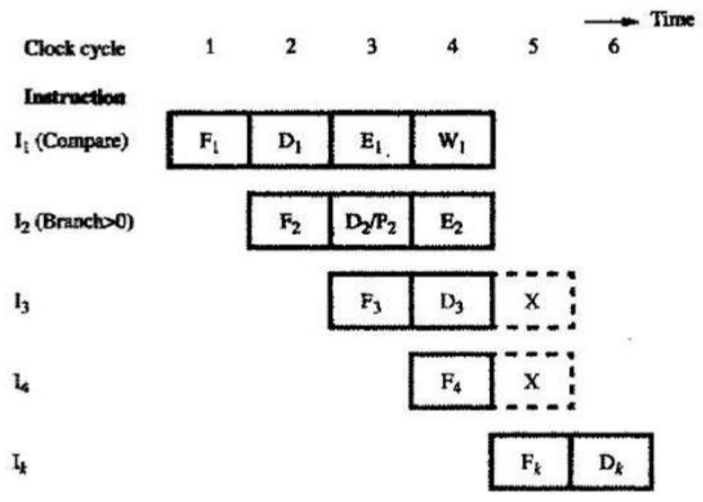


Figure 8.14 Timing when a branch decision has been incorrectly predicted as not taken.

The figure shows a Compare instruction followed by a Branch>0 instruction. Branch prediction takes place in cycle 3, while instruction 1_3 is being fetched. The fetch unit predicts that the branch will not be taken, and it continues to fetch instruction I_4 as I_3 enters the Decode stage. The results of the compare operation are available at the end of cycle 3. Assuming that they are forwarded immediately to the instruction fetch unit, the branch condition is evaluated in cycle 4. At this point, the instruction fetch unit realizes that the prediction was incorrect, and the two instructions in the execution pipe are purged. A new instruction, I_k, is fetched from the branch target address in clock cycle 5.

If branch outcomes were random, then half the branches would be taken. Then the simple approach of assuming that branches will not be taken would save the time lost to conditional branches 50 percent of the time. However, better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken, depending on the expected program behavior.

A decision on which way to predict the result of the branch may be made in hardware by observing whether the target address of the branch is lower than or higher than the address of the branch instruction. A more flexible approach is to have the compiler decide whether a given branch instruction should be predicted taken or not taken. The branch instructions of some processors, such as SPARC, include a branch prediction bit, which is set to 0 or 1 by the compiler to indicate the desired behavior. The instruction fetch unit checks this bit to predict whether the branch will be taken or not taken.

Any approach that has this characteristic is called *static branch prediction*. Another approach in which the prediction decision may change depending on execution history is called *dynamic branch prediction*.

Dynamic Branch Prediction:

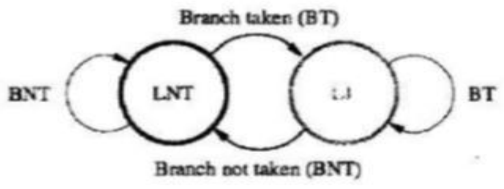
In dynamic branch prediction schemes, the processor hardware assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that instruction is executed.

In its simplest form, the execution history used in predicting the outcome of a given branch instruction is the result of the most recent execution of that instruction. The processor assumes that the next time the instruction is executed, the result is likely to be the same. Hence, the algorithm may be described by the two-state machine in Figure 8.15a. The two states are:

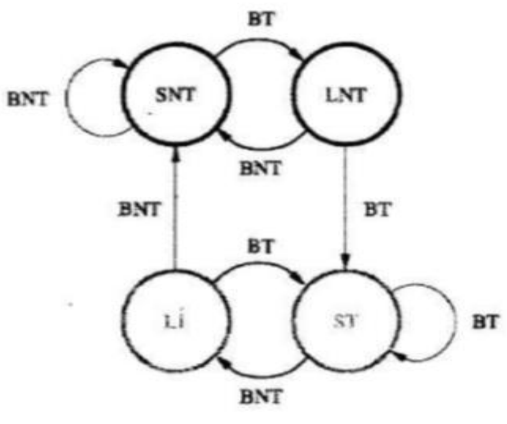
LT: Branch is likely to be taken

LNT: Branch is likely not to be taken

Suppose that the algorithm is started in state LNT. When the branch instruction is executed and if the branch is taken, the machine moves to state LT. Otherwise, it remains in state LNT. The next time the same instruction is encountered, the branch is predicted as taken if the corresponding state machine is in state LT. Otherwise it is predicted as not taken.



(a) A 2-state algorithm



(b) A 4-state algorithm

Figure 8.15 State-machine representation of branch-prediction algorithms.

This simple scheme, which requires one bit of history information for each branch instruction, works well inside program loops. Once a loop is entered, the branch instruction that controls looping will always yield the same result until the last pass through the loop is reached. In the last pass, the branch prediction will turn out to be incorrect, and the branch history state machine will be changed to the opposite state. Unfortunately, this means that the next time this same loop is entered, and assuming that there will be more than one pass through the loop, the machine will lead to the wrong prediction.

Better performance can be achieved by keeping more information about execution history. An algorithm that uses 4 states, thus requiring two bits of history information for each branch instruction, is shown in Figure 8.15b. The four states are:

ST: Strongly likely to be taken

LT: Likely to be taken

LNT: Likely not to be taken

SNT: Strongly likely not to be taken

Again assume that the state of the algorithm is initially set to LNT. After the branch instruction has been executed, and if the branch is actually taken, the state is changed to ST; otherwise, it is changed to SNT. As program execution progresses and the same instruction is encountered again, the state of the branch prediction algorithm continues to change as shown. When a branch instruction is encountered, the instruction fetch unit predicts that the branch will be taken if the state is either LT or ST, and it begins to fetch instructions at the branch target address. Otherwise, it continues to fetch instructions in sequential address order.

INFLUENCES ON INSTRUCTION SETS:

ADDRESSING MODES:

Addressing modes should provide the means for accessing a variety of data structures simply and efficiently. Useful addressing modes include index, indirect, auto-increment, and auto-decrement. Many processors provide various combinations of these modes to increase the flexibility of their instruction sets. Complex addressing modes, such as those involving double indexing, are often encountered. In choosing the addressing modes to be implemented in a pipelined processor, we must consider the effect of each addressing mode on instruction flow in the pipeline. Two important considerations in this regard are the side effects of modes such as auto-increment and auto decrement and the extent to which complex addressing modes cause the pipeline to stall. Another important factor is whether a given mode is likely to be used by compilers. To compare various approaches, we assume a simple model for accessing operands in the memory. The load instruction Load X(R1),R2 takes five cycles to complete execution. The instruction

Load (R1),R2

can be organized to fit a four-stage pipeline because no address computation is required. Access to memory can take place in stage E.

A more complex addressing mode may require several accesses to the memory to reach the named operand. For example, The instruction

Load (X(R1)),R2

may be executed as shown in Figure 8.16a, assuming that the index offset, X, is given in the instruction word. After computing the address in cycle 3, the processor needs to access memory twice - *first* to read location X+[R1] in clock cycle 4 and then to read location [X+[R1]] in cycle 5. If R2 is a source operand in the next instruction, that instruction would be stalled for three cycles, which can be reduced to two cycles with operand forwarding, as shown.

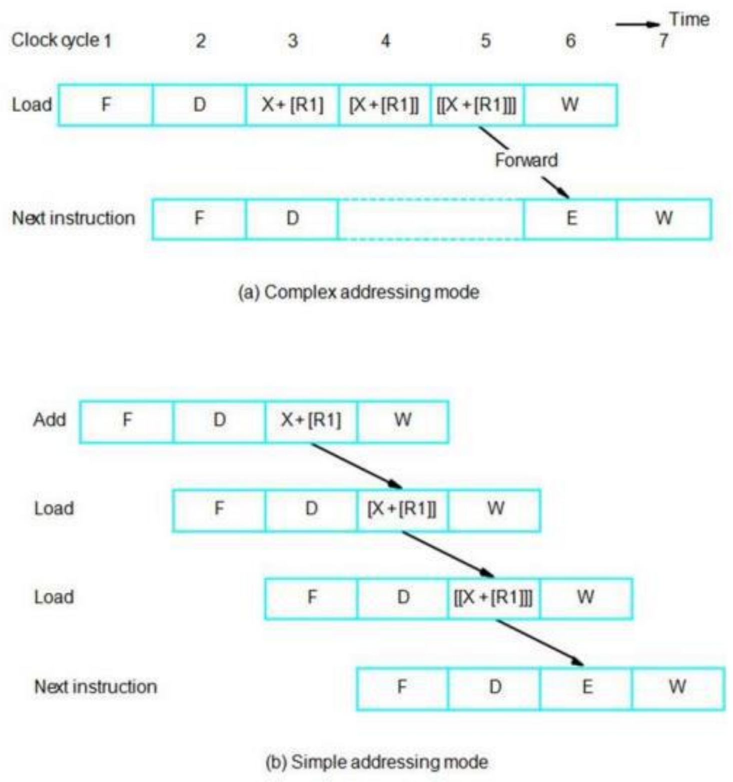


Fig 8.16 Equivalent operations using complex and simple addressing modes

To implement the same Load operation using only simple addressing modes requires several instructions. For example, on a computer that allows three operand addresses, we can use

```
Add    #X,RI,R2
Load   (R2),R2
Load   (R2),R2
```

The Add instruction performs the operation R2 ← X+ [R1]. The two Load instructions fetch the address and then the operand from the memory. This sequence of instructions takes exactly the same number of clock cycles as the original, single Load instruction, as shown in Figure 8.16b.

The addressing modes used in modern processors often have the following features:

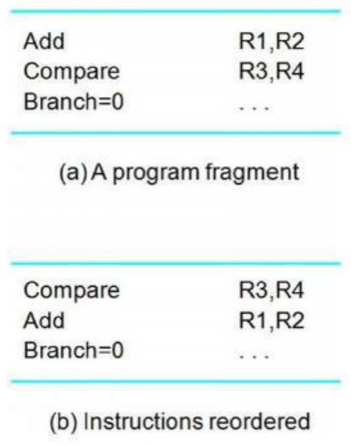
- Access to an operand does not require more than one access to the memory.
- Only load and store instructions access memory operands.
- The addressing modes used do not have side effects.

Three basic addressing modes that have these features are register, register indirect, and index. The first two require no address computation. In the index mode, the address can be computed in one cycle, whether the index value is given in the instruction or in a register.

Condition Codes:

In many processors, the condition code flags are stored in the processor status register. They are either set or cleared by many instructions, so that they can be tested by subsequent conditional branch instructions to change the flow of program execution. An optimizing compiler for a pipelined processor attempts to reorder instructions to avoid stalling the pipeline when branches or data dependencies between successive instructions occur. In doing so, the compiler must ensure that reordering does not cause a change in the outcome of a computation. The dependency introduced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.

Consider the sequence of instructions in Figure 5.17a, and assume that the execution of the Compare and Branch=0 instructions proceeds as in Figure 8.14. The branch decision takes place in step E2 rather than D2 because it must await the result of the Compare instruction. The execution time of the Branch instruction can be reduced by interchanging the **Add** and **Compare** instructions, as shown in Figure 5.17b



```
Add          R1,R2
Compare      R3,R4
Branch=0     . . .
```

(a) A program fragment

```
Compare      R3,R4
Add          R1,R2
Branch=0     . . .
```

(b) Instructions reordered

Figure 5.17 Instruction reordering.

Condition codes can be handled in two ways:

First, to provide flexibility in reordering instructions, the condition-code flags should be affected by as few instructions as possible. Second, the compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not. An instruction set designed with pipelining in mind usually provides the desired flexibility.

LARGE COMPUTER SYSTEMS

FORMS OF PARALLEL PROCESSING

Two fundamental aspects of parallel processing are:

- First, the overall task has the property that some of its subtasks can be done in parallel by different hardware components. For example, a processor computation and an I/O transfer are performed in parallel by the processor and the DMA controller.
- Second, some means must exist for initiating and coordinating the parallel activity. Initiation occurs when the processor sets up the DMA transfer and then continues with another computation. When the transfer is completed, the coordination is achieved by the interrupt signal sent from the DMA controller to the processor. This allows the processor to begin the computation that operates on the transferred data.

CLASSIFICATION OF PARALLEL STRUCTURES:

A general classification of parallel processing has been proposed by Flynn.

They are:

SISD: A single-processor computer system is called a *Single Instruction stream, Single Data stream (SISD)* system. A program executed by the processor constitutes the single instruction stream, and the sequence of data items that it operates on constitutes the single data stream.

SIMD: A single stream of instructions is broadcast to a number of processors. Each processor operates on its own data. This scheme, in which all processors execute the same program but operate on different data, is called a *Single Instruction stream, Multiple Data stream (SIMD)* system.

MIMD: The multiple data streams are the sequences of data items accessed by the individual processors in their own memories. The third scheme involves a number of independent processors, each executing a different program and accessing its own sequence of data items. Such machines are called *Multiple Instruction stream, Multiple Data stream (MIMD)*systems.

MISD: The fourth possibility is a *Multiple instruction stream, Single Data stream (MISD)* system. In such a system, a common data structure is manipulated by separate processors, each executing a different program.

ARRAY PROCESSORS

The SIMD form of parallel processing, also called *array processing,* was the first form of parallel processing. A two-dimensional grid of processing elements executes an instruction stream that is broadcast from a central control processor. As each instruction is broadcast,all elements execute it simultaneously. Each processing element is connected toits four nearest neighbors for purposes of exchanging data.

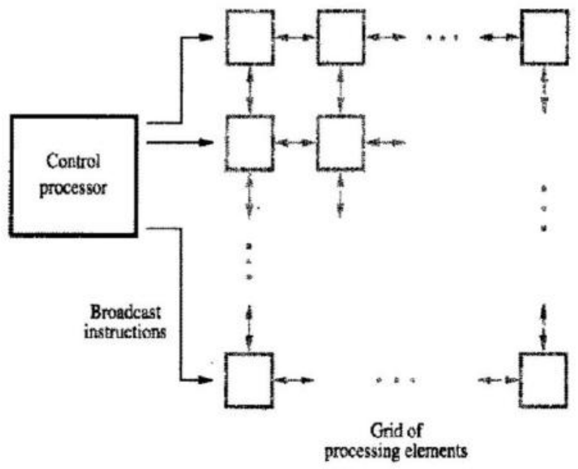


Figure 12.1 An array processor.

The grid of processing elements can be used to solve two dimensional problems. For example, if each element of the grid represents a point in space, the array can be used to compute the temperature at points in the inferior of a conducting plane. Assume that the edges of the plane are held at some fixed temperatures. An approximate solution at the discrete points represented by the processing elements is derived as follows. The outer edges are initialized to the specified temperatures. All interior points are initialized to arbitrary values, not necessarily the same. Iterations are then executed in parallel at each element. Each iteration consists of calculating an improved estimate of the temperature at a point by averaging the current values of its four nearest neighbors. The process stops when changes in the estimates during successive iterations are less than some predefined small quantity.

Each element must be able to exchange values with each of its neighbors over the paths. Each processing element has a few registers and some local memory to store data. It also has a register, which we can call the network register that facilitates movement of values to and from its neighbors. The central processor can broadcast an instruction to shift the values in the network registers one step up, down, left, or right. Each processing element also contains an ALU to execute arithmetic instructions broadcast by the control processor. Using these basic facilities, a sequence of instructions can be broadcast repeatedly to implement the iterative loop. The control processor must be able to determine when each of the processing elements has developed its component of the temperature to the required accuracy. To do this, each element sets an internal status bit to 1 to indicate this condition. The grid interconnections include a facility that allows the controller to detect when all status bits are set at the end of iteration.

Array processors are highly specialized machines. They are well-suited to numerical problems that can be expressed in matrix or vector format. Recall that super computers with vector architecture are also suitable for solving such problems. A key difference between vector-based machines and array processors is that the former achieve high performance through heavy use of pipelining, whereas the latter provide extensive parallelism by replication of computing modules

THE STRUCTURE OF GENERAL-PURPOSE MULTIPROCESSORS

There are three possible ways of implementing a multiprocessor system.

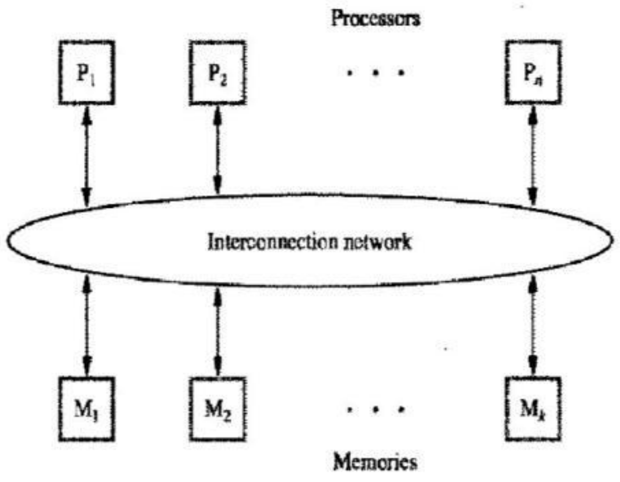


Figure 12.2 A UMA multiprocessor.

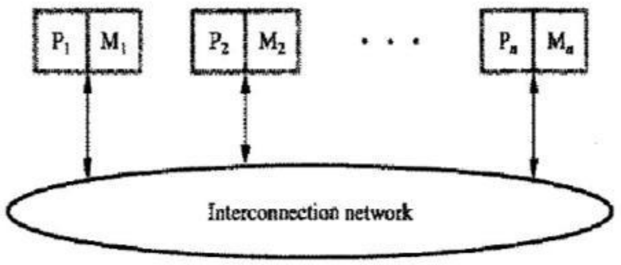


Figure 12.3 A NUMA multiprocessor.

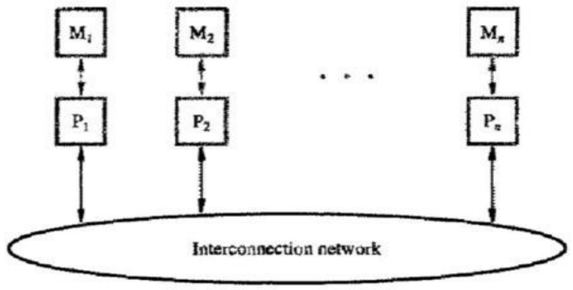


Figure 12.4 A distributed memory system.

The most obvious scheme is given in Figure 12.2.An interconnection network permits n processors to access & memories so that any of the processors can access any of the memories. The interconnection network may introduce considerable delay between a processor and a memory. If this delay is the same for all accesses to memory, which is common for this organization, then such a machine is called a *Uniform Memory Access (UMA)* multiprocessor. Because of the extremely short instruction execution times achievable by processors, the network delay in fetching instructions and data from the memories is unacceptable if it is too long.

An attractive alternative, which allows a high computation rate to be sustained in all processors, is to attach the memory modules directly to the processors. This organization is shown in Figure 12.3, In addition to accessing its local memory, each processor can also access other memories over the network. Since the remote accesses pass through the network, these accesses take considerably longer than accesses to the local memory. Because of this difference in access times, such multiprocessors are called *Non-Uniform Memory Access (NUMA)* multiprocessors.

The organizations of Figures 12.2 and 12.3 provide a global memory, where any processor can access any memory module without intervention by another processor.

A different way of organizing the system is shown in Figure 12.4. Here, all memory modules serve as private memories for the processors that are directly connected to them. A processor cannot access a remote memory without the cooperation of the remote processor. This cooperation takes place in the form of messages exchanged by the processors. Such systems are often called *distributed-memory systems* with a *message-passing protocol*.

INTERCONNECTION NETWORKS

The components that form a multiprocessor system are CPUs, IOPs connected to input output devices, and a memory unit. The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available between the processors and memory in a shared memory system o among the processing elements in a loosely coupled system there are several physical forms available for establishing an interconnection network.

Time-shared common bus

Multiport memory

Crossbar switch

Multistage switching network

Hypercube system

Time Shared Common Bus: A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit.

Disadvantage: Only one processor can communicate with the memory or another processor at any given time. As a consequence, the total overall transfer rate within the system is limited by the speed of the single path. A more economical implementation of a dual bus structure is depicted in Fig., below. Part of the local memory may be designed as a cache memory attached to the CPU.

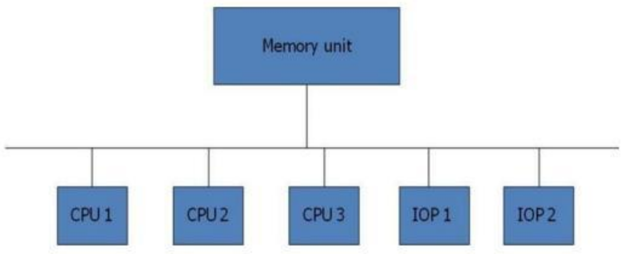


Fig: Time shared common bus organization

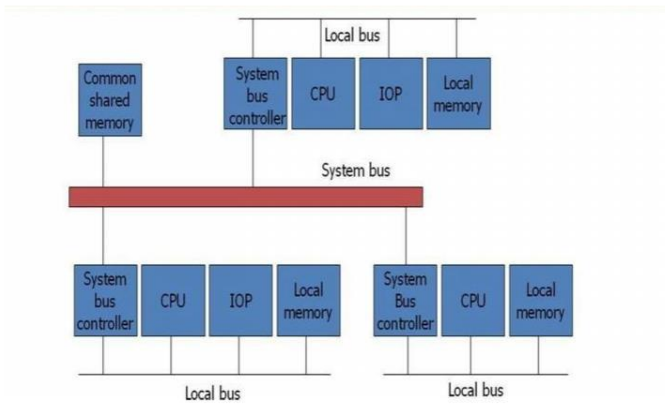


Fig: System bus structure for multiprocessors

Multiport Memory:

A multiport memory system employs separate buses between each memory module and each CPU. The module must have internal control logic to determine which port will have access to memory at any given time. Memory access conflicts are resolved by assigning fixed priorities to each memory port.

<u>**Advantages:**</u> The high transfer rate can be achieved because of the multiple paths.

<u>**Disadvantage:**</u> It requires expensive memory control logic and a large number of cables and connections

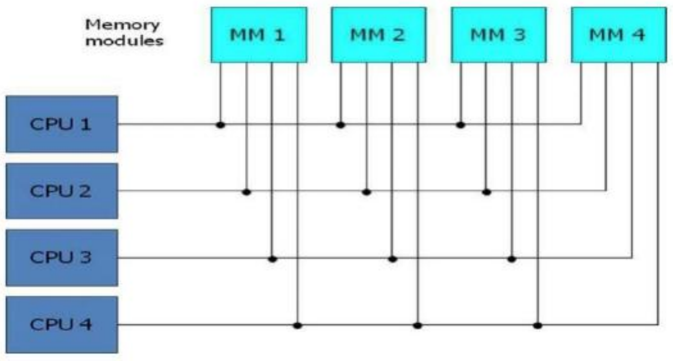


Fig: Multiport memory organization

Crossbar Switch: Consists of a number of cross points that are placed at intersections between processor buses and memory module paths. The small square in each cross point is a switch that determines the path from a processor to a memory module.

Advantages: Supports simultaneous transfers from all memory modules.

Disadvantage: The hardware required to implement the switch can become quite large and complex. Below fig. shows the functional design of a crossbar switch connected to one memory module.

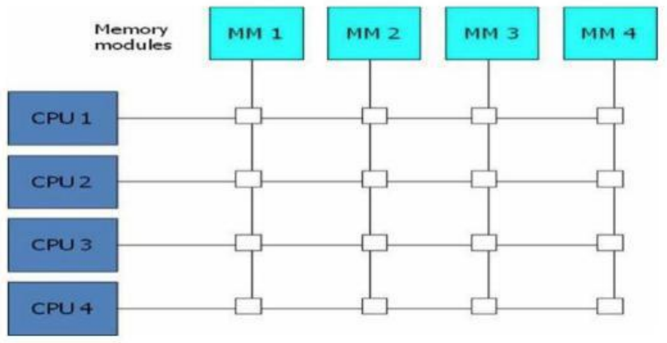
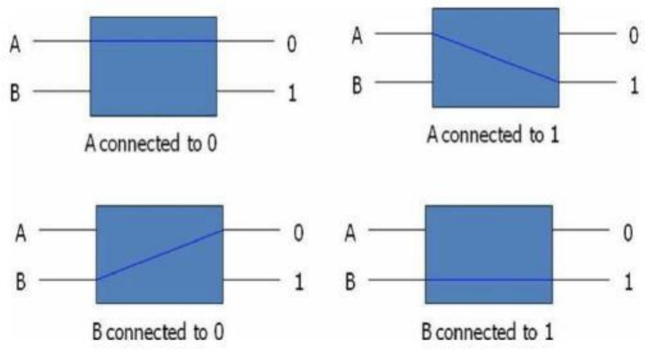
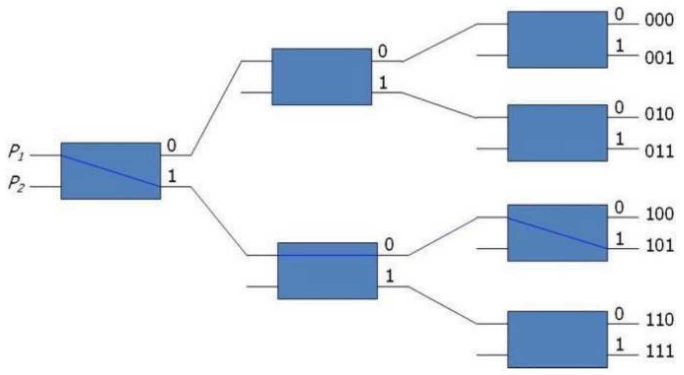


Fig: Crossbar switch

Multistage Switching Network: The basic component of a multistage network is a two input, two-output interchange switch as shown in Fig. below.



Using the 2x2 switch as a building block, it is possible to build a multistage network to control the communication between a number of sources and destinations. To see how this is done, consider the binary tree shown in Fig. below. Certain request patterns cannot be satisfied simultaneously. i.e., if P1 000~011, then P2 100~111 One such topology is the omega switching network shown in Fig. below



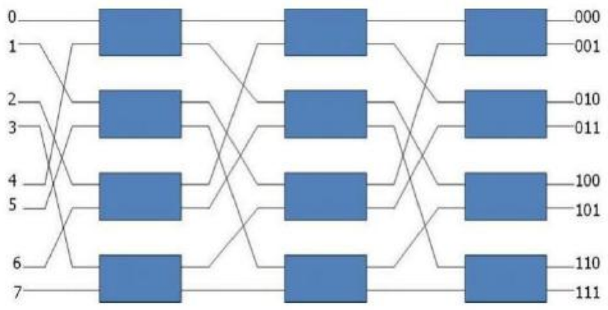


Fig: 8 x 8 Omega Switching Network

Some request patterns cannot be connected simultaneously. i.e., any two sources cannot be connected simultaneously to destination 000 and 001. In a *tightly coupled* multiprocessor system, the source is a processor and the destination is a memory module. In a *loosely coupled* multiprocessor system, both the source and destination are processing elements.

Hypercube System:

The topology of an n-dimensional cube, called a *hypercube*, to implement a network that interconnects 2^n nodes. In addition to the communication circuits, each node usually includes a processor and a memory module as well as some I/O capability.

Figure 12.7 shows a three-dimensional hypercube. The small circles represent the communication circuits in the nodes. The functional units attached to each node are not shown in the figure. The edges of the cube represent bidirectional communication links between neighboring nodes. In an n-dimensional hypercube, each node is directly connected to n neighbors. A useful way to label the nodes is to assign binary addresses to them in such a way that the addresses of any two neighbors differ in exactly one bit position, as shown in the figure.

Routing messages through the hypercube is particularly easy. If the processor at node N_i wishes to send a message to node N_j it proceeds as follows. The binary addresses of the source, i, and the destination, j, are compared from least to most significant bits. Suppose that they differ first in position p. Node N_i then sends the message to its neighbor whose address, k, differs from i in bit position p. Node N_k forwards the message to the appropriate neighbor using the same address comparison scheme. The message gets closer to destination node N_j with each of these hops from one node to another. For example, a message from node N_2 to node N_5 requires 3 hops, passing through nodes N_3 and N_1. The maximum distance that any message needs to travel in an n-dimensional hypercube is n hops.

Scanning address patterns from right to left is only one of the methods that can be used to determine message routing.

Hypercube interconnection networks have been used in a number of machines. The better known examples include Intel's iPSC, which used a 7-dimensional cube to connect up to 128 nodes, and NCUBE's NCUBE/ten, which had up to 1024 nodes in 4 10-dimensional cube.

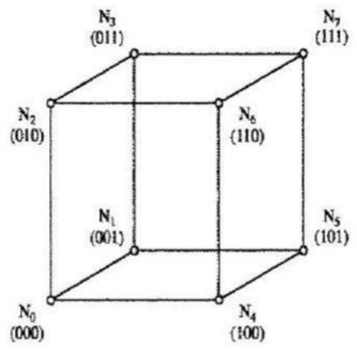


Figure 12.7 A 3-dimensional hypercube network.

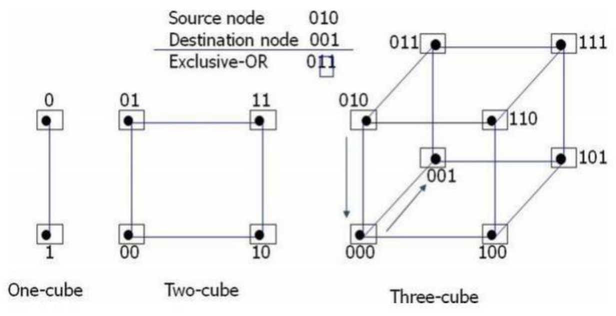


Fig: Hypercube structures for n=1,2,3

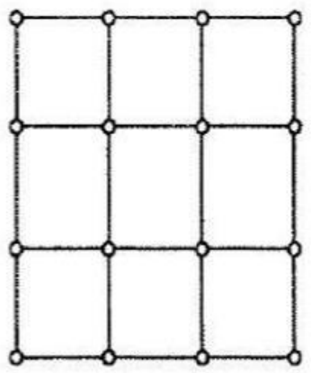


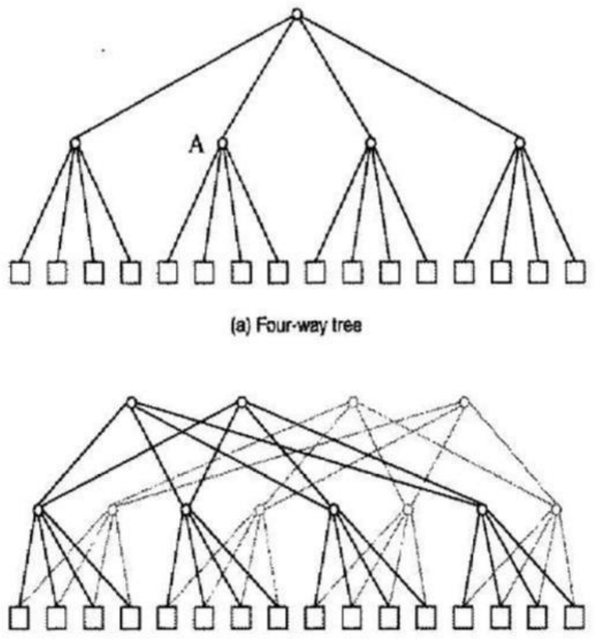
Figure 12.8 A 2-dimensional mesh network.

TREE NETWORKS:

A hierarchically structured network implemented in the form of a tree is another interconnection topology. Figure 12.9a depicts a four-way tree that interconnects 16 modules. In this tree, each parent node allows communication between two of its children at a time. An intermediate-level node, for example node A-in the figure, can provide a connection from one of its child nodes to its parent. This enables two leaf nodes that are any distance apart to communicate. Only one path at a time can be established through a given node in the tree.

A tree network performs well if there is a large amount of locality in communication, that is, if only a small portion of network traffic goes through the single root node. If this is not the case, performance deteriorates rapidly because the root node becomes a bottleneck.

To reduce the possibility of a bottleneck, the number of links in the upper levels of a tree hierarchy can be increased. This is done in a fat tree network, in which each node in the tree (except at the top level) has more than one parent. An example of a fat tree is given in Figure 12.9b. In this case, each node has two parent nodes. A fat tree structure was used in the CM-5 machine by Thinking Machines Corporation.



(a) Four-way tree

(b) Fat tree

Figure 12.9 Tree-based networks.