COMPUTER ORGANIZATION

COMPUTER ORGANIZATION

Prof. K. Vikram
M.E., Ph.D., MISTE, MIETE., CSI
Professor & Principal
Narayanadri Institute of Science and Technology
Rajampet, Kadapa Dist., Andhra Pradesh

Paramount Publishing House
• NEW DELHI • HYDERABAD

ComputerOrganization

First Edition - 2015

ISBN : 978-93-85100-18-5

Price : ` **000.00**

Paramount Publishing House
3-5-1105, 2nd Floor, PVR Estates, Behind Hero Honda Showroom, Narayanaguda, Hyderabad-500 029.
Phone: 040-64554822.

Sales Offices :

Hyderabad
3-5-1105, 2nd Floor, PVR Estates, Behind Hero Honda Showroom, Narayanaguda, Hyderabad-500 029.
Phone: 040-64554822.

Visakhapatnam
D.No.28-8-3, First Floor, Opp. Sri Venkateswara Theatre Outgate, Suryabagh, Visakhapatnam-530 002.
Phones : 0891-6639247 & 0891-6646082.

Phones : 0891-6639247 & 0891-6646082.
New Delhi
C/14, SDIDC Work Centre Jhilmil Colony, New Delhi-100095. Phone: 011-22162365.

paramountpublishers@gmail.com | alluriasr2005@yahoo.com

Published by Krishna Prasad Alluri for Paramount Publishing House and printed by him at
Sai Thirumala Printers.

SYLLABUS

PART - A

UNIT-1

Basic Structure of Computers: Computer Types, Functional Units, Basic Operational Concepts, Bus Structures, Performance - Processor Clock, Basic Performance Equation, Clock Rate, Performance Measurement, Historical Perspective Machine Instructions and Programs: Numbers, Arithmetic Operations and Characters, Memory Location and Addresses, Memory Operations, Instructions and Instruction Sequencing,

UNIT - 2

Machine Instructions and Programs contd.: Addressing Modes, Assembly Language, Basic Input and Output Operations, Stacks and Queues, Subroutines, Additional Instructions, Encoding of Machine Instructions

UNIT - 3

Input/Output Organization: Accessing I/O Devices, Interrupts - Interrupt Hardware, Enabling and Disabling Interrupts, Handling Multiple Devices, Controlling Device Requests, Exceptions, Direct Memory Access, Buses

UNIT-4

Input/Output Organization contd.: Interface Circuits, Standard I/O Interfaces - PCI Bus, SCSI Bus, USB

PART - B

UNIT - 5

Memory System: Basic Concepts, Semiconductor RAM Memories, Read Only Memories, Speed, Size, and Cost, Cache Memories - Mapping Functions, Replacement Algorithms, Performance Considerations, Virtual Memories, Secondary Storage

UNIT - 6

Arithmetic: Addition and Subtraction of Signed Numbers, Design of Fast Adders, Multiplication of Positive Numbers, Signed Operand Multiplication, Fast Multiplication, Integer Division, Floating-point Numbers and Operations

UNIT - 7

Basic Processing Unit: Some Fundamental Concepts, Execution of a Complete Instruction, Multiple Bus Organization, Hard-wired Control, Microprogrammed Control

UNIT - 8

Multicores, Multiprocessors, and Clusters: Performance, The Power Wall, The Switch from Uniprocessors to Multiprocessors, Amdahl's Law, Shared Memory Multiprocessors, Clusters and other Message Passing Multiprocessors, Hardware Multithreading, SISD, IMD, SIMD, SPMD, and Vector.

Contents

PART - A

UNIT-1

Basic Structure of Computers: Computer Types, Functional Units, Basic Operational Concepts, Bus Structures, Performance - Processor Clock, Basic Performance Equation, Clock Rate, Performance Measurement, Historical Perspective.

Machine Instructions and Programs: Numbers, Arithmetic Operations and Characters, Memory Location and Addresses, Memory Operations, Instructions and Instruction Sequencing.

Unit-1

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 have 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.,

FUNCTIONAL UNIT

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

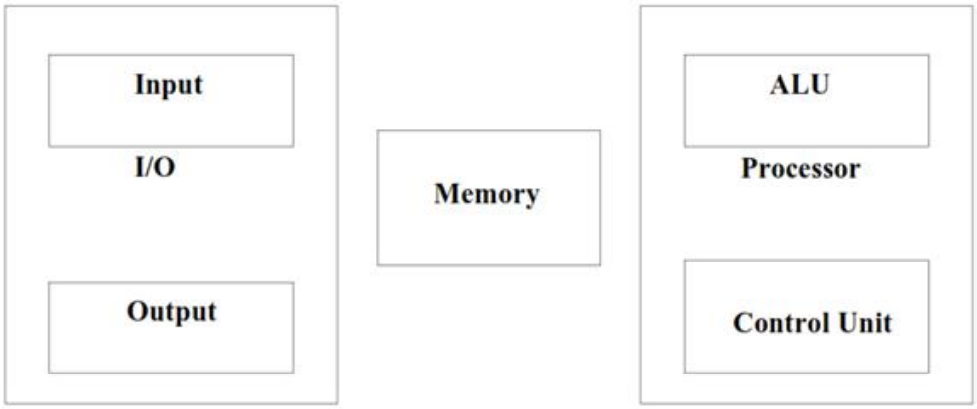


Fig a : Functional units of computer

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 perations. 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 into store programs and data. It is basically to two types

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 are processed in a group of fixed site called word.

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 der 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 aften 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 may 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.

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 xecutes the specified operations. Data to be stored are also stored in the memory.

Examples: Add LOCA, R

This instruction adds the operand at memory location LOCA, to operand in register R & places the sum into register. 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 R
3. Finally the resulting sum is stored in the register R

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

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.

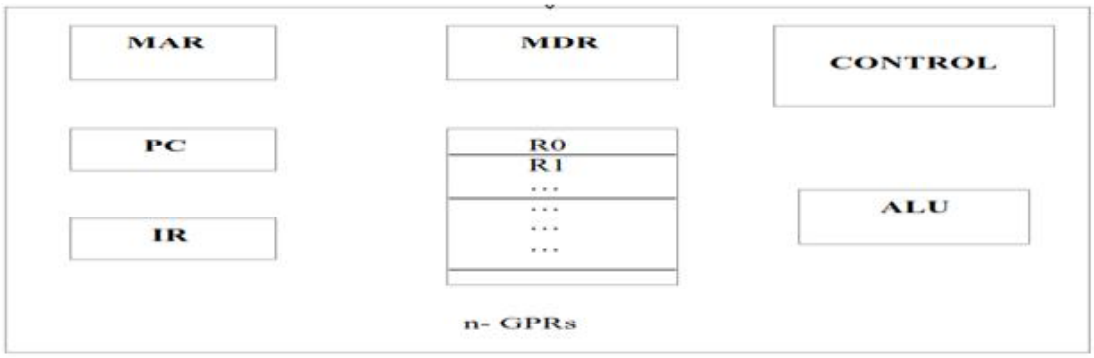


Fig b : Connections between the processor and the memory

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 instructions 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.

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 I/P 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 STRUCTURE

The simplest and most common way of interconnecting various parts of the computer. 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.

In addition to the lines that carry the data, the bus must have lines for address and control purpose. Simplest way to interconnect is to use the single bus as shown.

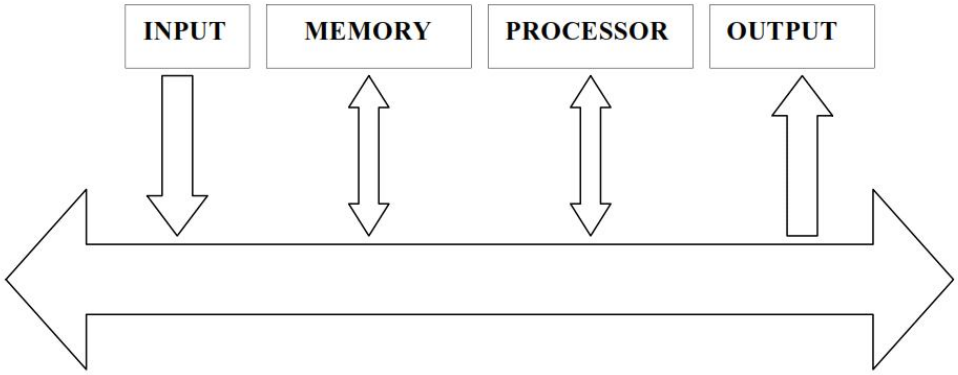


Figure C : Single Bus Structure

Since 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.

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 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.

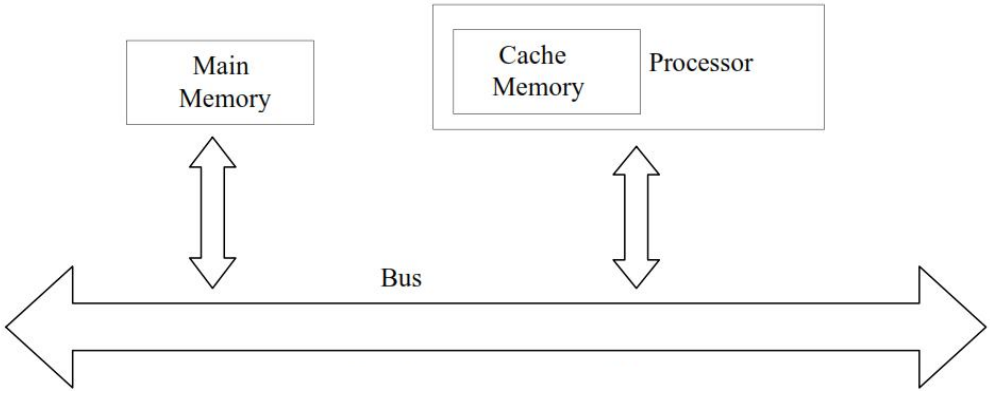


Figure d : The processor cache

The pertinent parts of the fig. c 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 have a clock rates that range from a few hundred million to over a billion cycles per second.

1.6 BASIC PERFORMANCE EQUATION

We now focus our attention on the processor time component of the total elapsed time. Let 'T' be the processor time required to execute a program that has been prepared in some high-level language. The compiler generates a machine language object program that corresponds to the source program. Assume that complete execution of the program requires the execution of N machine cycle language instructions. The number N is the actual number of instruction execution and is not necessarily equal to the number of machine cycle instructions in the object program. Some instruction may be executed more than once, which in the case for instructions inside a program loop others may not be executed all, depending on the input data used.

Suppose that the average number of basic steps needed to execute one machine cycle instruction is S, where each basic step is completed in one clock cycle. If clock rate is 'R' cycles per second, the program execution time is given by

$$T = \frac{N \times S}{R}$$

this is often referred to as the basic performance equation.

We must emphasize that N, S & R are not independent parameters changing one may affect another. Introducing a new feature in the design of a processor will lead to improved performance only if the overall result is to reduce the value of T.

Pipelining and super scalar operation

We assume that instructions are executed one after the other. Hence the value of S is the total number of basic steps, or clock cycles, required to execute one instruction. A substantial mprovement in performance can be achieved by overlapping the execution of successive instructions using a technique called pipelining.

Consider Add R1R2R3

This adds the contents of R1& R2 and places the sum in to R3

The contents of R1 & R2 are first transferred to the inputs of ALU. After the addition operation is performed, the sum is transferred to R . The processor can read the next instruction from the memory, while the addition operation is being performed. Then of that instruction also uses, the ALU, its operand can be transferred to the ALU inputs at the same time that the add instructions is being transferred to R

In the ideal case if all instructions are overlapped to the maximum degree possible the execution proceeds at the rate of one instruction completed in each clock cycle. Individual instructions still require several clock cycles to complete. But for the purpose of computing T, effective value of S is 1.

A higher degree of concurrency can be achieved if multiple instructions pipelines are implemented in the processor. This means that multiple functional units are used creating parallel paths through which different instructions can be executed in parallel with such an arrangement, it becomes possible to start the execution of several instructions in every clock cycle. This mode of operation is called superscalar execution. If it can be sustained for a long time during program execution the effective value of S can be reduced to less than one. But the parallel execution must preserve logical correctness of programs, that is the results produced must be same as those produced by the serial execution of program instructions. Now a days may processor are designed in this manner.

1.7 CLOCK RATE

These are two possibilities for increasing the clock rate 'R'.

1. Improving the IC technology makes logical circuit faster, which reduces the time of execution of basic steps. This allows the clock period P, to be reduced and the clock rate R to be increased.
2. Reducing the amount of processing done in one basic step also makes it possible to reduce the clock period P. however if the actions that have to be performed by an instructions remain the same, the number of basic steps needed may increase.

Increase in the value 'R' that are entirely caused by improvements in IC technology affects all aspects of the processor's operation equally with the exception of the time it takes to access the main memory. In the presence of cache the percentage of accesses to the main memory is small. Hence much of the performance gain excepted from the use of faster technology can be realized.

Instruction set CISC & RISC

Simple instructions require a small number of basic steps to execute. Complex instructions involve a large number of steps. For a processor that has only simple instruction a large number of instructions may be needed to perform a given programming task. This could lead to a large value of 'N' and a small value of 'S' on the other hand if individual instructions perform more complex

operations, a fewer instructions will be needed, leading to a lower value of N and a larger value of S. It is not obvious if one choice is better than the other.

But complex instructions combined with pipelining (effective value of S 1) would achieve one best performance. However, it is much easier to implement efficient pipelining in processors with simple instruction sets.

1.8 PERFORMANCE MEASUREMENTS

It is very important to be able to access the performance of a computer, comp designers use performance estimates to evaluate the effectiveness of new features. The previous argument suggests that the performance of a computer is given by the execution time T, for the program of interest.

Inspite of the performance equation being so simple, the evaluation of 'T' is highly complex. Moreover the parameters like the clock speed and various architectural features are not reliable indicators of the expected performance.

Hence measurement of computer performance using bench mark programs is done to make comparisons possible, standardized programs must be used.

The performance measure is the time taken by the computer to execute a given bench mark. Initially some attempts were made to create artificial programs that could be used as bench mark programs. But synthetic programs do not properly predict the performance obtained when real application programs are run.

A non profit organization called SPEC- system performance evaluation corporation selects and publishes bench marks.

The program selected range from game playing, compiler, and data base applications to numerically intensive programs in astrophysics and quantum chemistry. In each case, the program is compiled under test, and the running time on a real computer is measured. The same program is also compiled and run on one computer selected as reference.

The 'SPEC' rating is computed as follows.

$$\text{SPEC rating} = \frac{\text{Running time on the reference computer}}{\text{Running time on computer under test}}$$

If the SPEC rating = 50

Means that the computer under test is 50 times as fast as the ultra sparc 10. This is repeated for all the programs in the SPEC suit, and the geometric mean of the result is computed.

Let SPECi be the rating for program 'i' in the suite. The overall SPEC rating for the computer is given by

SPEC rating = $\left(\prod_{i=1}^{n} n \, SPEC \right)^{\frac{1}{n}}$

where 'n' = number of programs in suite.

Since actual execution time is measured the SPEC rating is a measure of the combined effect of all factors affecting performance, including the compiler, the OS, the processor, the memory of comp being tested.

Multiprocessor & microprocessors

- 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.

1.10 NUMBER REPRESENTATION

Consider an n-bit vector

$$B = b_{n-1} \ldots b_1 b_0$$

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

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

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 (b 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. 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 2n.

B	Values represented		
b3b2b1 b0	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			

1 0 0 1	+3	+3	+3
0 0 0 0	+2	+2	+2
1 0 0 0	+1	+1	+1
0 1 0 0	+0	+0	+0
0 1 0 0	-0	-0	-0
1 1 0 1	-1	-1	-1
0 1 0 1	-2	-2	-2
1	-3	-3	-3
1 1 0 0 1 1 0	-4	-4	-4
1	-5	-5	-5
1 1 1 0 1 1 1	-6	-6	-6
1	-7	-7	-7

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.

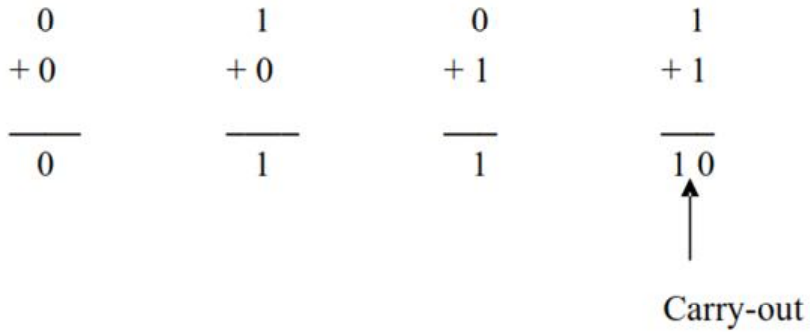


Figure 2.2 Addition of 1-bit numbers

1.12 MEMORY LOCATIONS AND ADDRESSES

Number and character operands, as well as instructions, are stored in the memory of a computer. The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1. Because a single bit represents a very small amount of information, bits are seldom handled individually. The usual approach is to deal with them in groups of fixed size. For this purpose, the memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation. Each group of n bits is referred to as a word of information, and n is called the word length. The memory of a computer can be schematically represented as a collection of words as shown in figure (a).

Modern computers have word lengths that typically range from 16 to 64 bits. If the word length of a computer is 32 bits, a single word can store a 32-bit 2's complement number or four ASCII characters, each occupying 8 bits. A unit of 8 bits is called a byte.

Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or addresses for each item location. It is customary to use numbers from 0 through 2K -1, for some suitable values of k, as the addresses of successive locations in the memory. The 2k addresses constitute the address space of the computer, and the memory can have up to 2k addressable locations. 24-bit address generates an address space of 224 (16,777,216) locations. A 32-bit address creates an address space of 232 or 4G (4 giga) locations.

BYTE Addressability

We now have three basic information quantities to deal with: the bit, byte and word. A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. The most practical assignment is to have successive addresses refer to successive byte

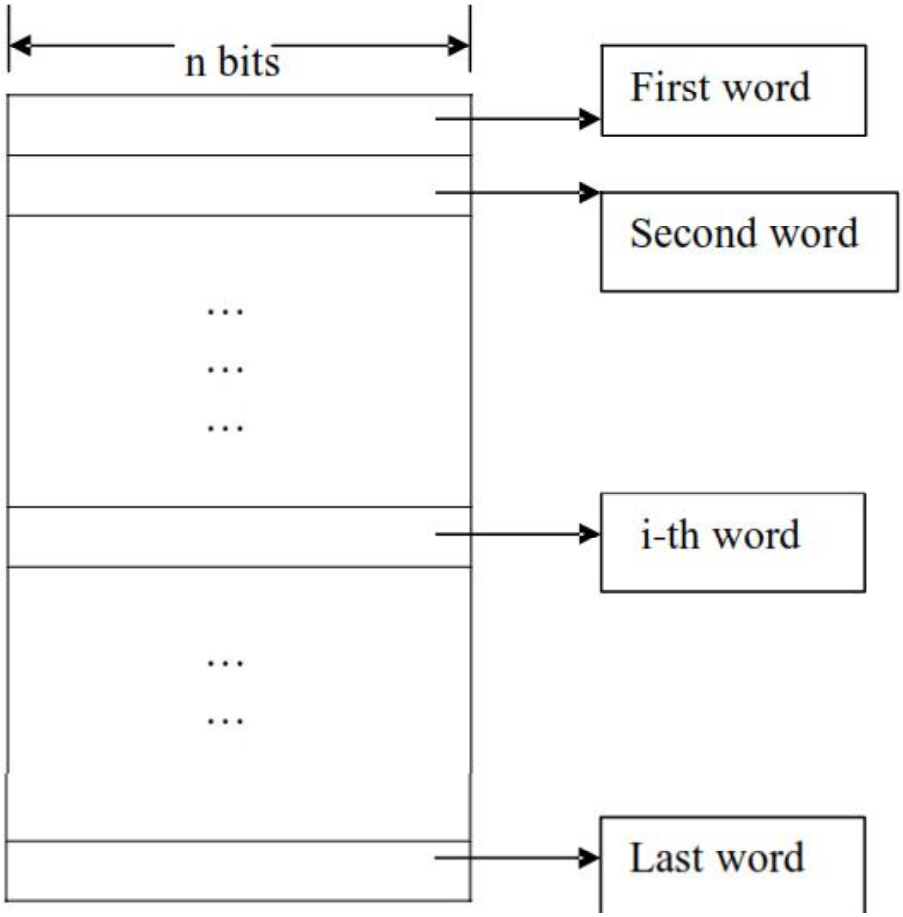
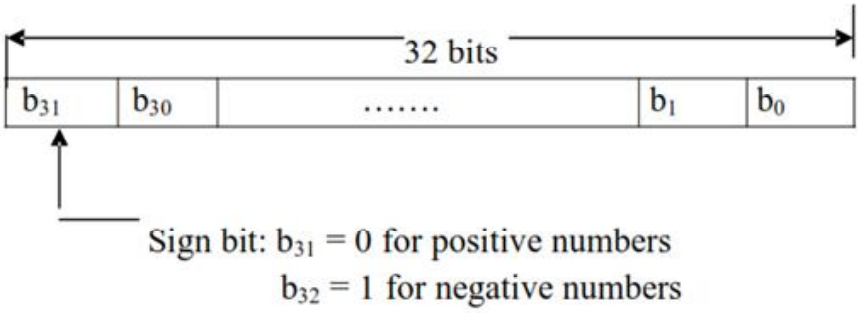


Fig a Memory words



(a) A signed integer

8 bits	8 bits	8 bits	8 bits
ASCII Character	ASCII character	ASCII character	ASCII character

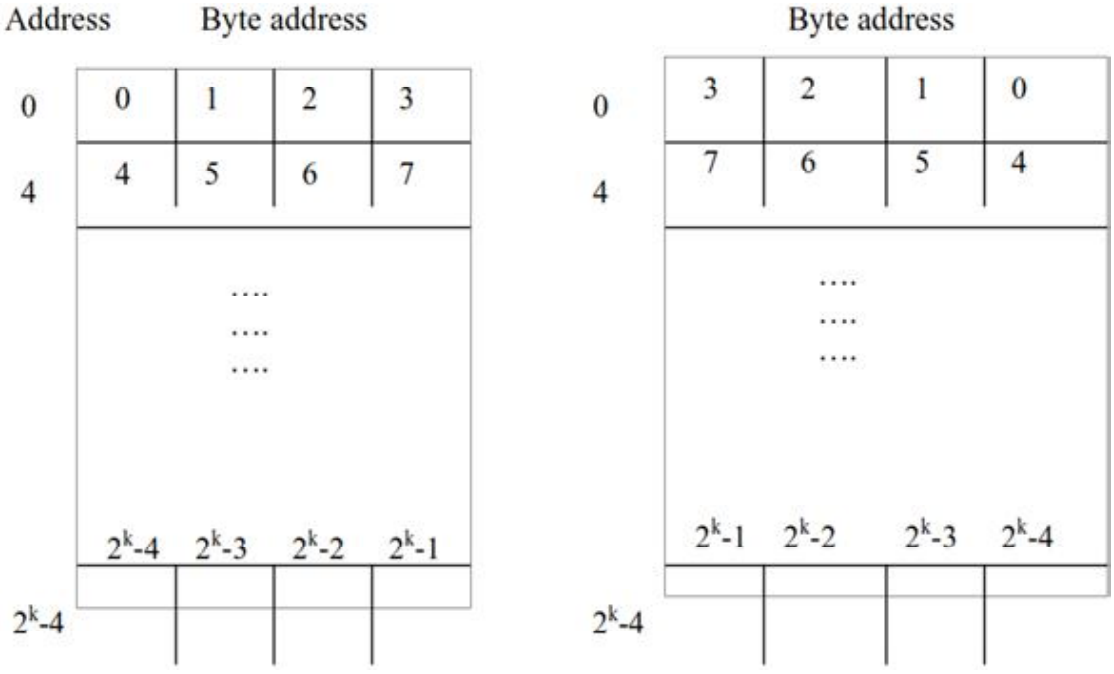
(b) Four characters

Locations in the memory. This is the assignment used in most modern computers, and is the one we will normally use in this book. The term byte-addressable memory is use for this assignment. Byte locations have addresses 0,1,2, …. Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0,4,8,…., with each word consisting of four bytes.

Big-Endian and Little-Endian Assignments

There are two ways that byte addresses can be assigned across words, as shown in fig b. The name big-endian is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word. The name little-endian is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word.

In addition to specifying the address ordering of bytes within a word, it is also necessary to specify the labeling of bits within a byte or a word. The same ordering is also used for labeling bits within a byte, that is, b7, b6, …., b , from left to right.



(a) Big-endian assignment **(b) Little-endian assiagnment**

Word Alignment

In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8, …, as shown in above fig. We say that the word locations have aligned addresses . in general, words are said to be aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word. The memory of bytes in a word is a power of 2. Hence, if the word length is 16 (2 bytes), aligned words begin at byte addresses 0,2,4,…, and for a word length of 64 (23 bytes), aligned words begin at bytes addresses 0,8,16 ….

There is no fundamental reason why words cannot begin at an arbitrary byte address. In that case, words are said to have unaligned addresses. While the most common case is to use aligned addresses, some computers allow the use of unaligned word addresses.

Accessing Numbers, Characters and Character Strings

A number usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte address.

In many applications, it is necessary to handle character strings of variable length. The beginning of the string is indicated by giving the address of the byte containing its first character. Successive byte locations contain successive characters of the string. There are two ways to indicate the length of the string. A special control character with the meaning "end of string" can be used as the last character in the string, or a separate memory word location or processor register can contain a number indicating the length of the string in bytes.

1.13 MEMORY OPERATIONS

Both program instructions and data operands are stored in the memory. To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor. Operands and results must also be moved between the memory and the processor. Thus, two basic operations involving the memory are needed, namely, Load (or Read or Fetch) and Store (or Write).

The load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Load operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

The store operation transfers an item of information from the processor to a specific memory location, destroying the former contents of that location. The processor sends the address of the desired location to the memory, together with the data to be written into that location.

An information item of either one word or one byte can be transferred between the processor and the memory in a single operation. Actually this transfer in between the CPU register & main memory.

1.14 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.

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

Another type of notation to represent machine instructions and programs. For this, we use an assembly language format. For example, an instruction that causes the transfer described above, from memory location LOC to processor register R1, is specified by the statement Move LOC, R1 he 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.

C ← [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, Source 2, 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].

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 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 R
represent a general-purpose register. The instructions
Load A, R
Store R ii, A and
Add A, R

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 Ri, Rj
Or
Add Ri, Rj, Rk

In both of these instructions, the source operands are the contents of registers Ri and Rj. In the first instruction, Rj also serves as the destination register, whereas in the second instruction, a third register, Rk, 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

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

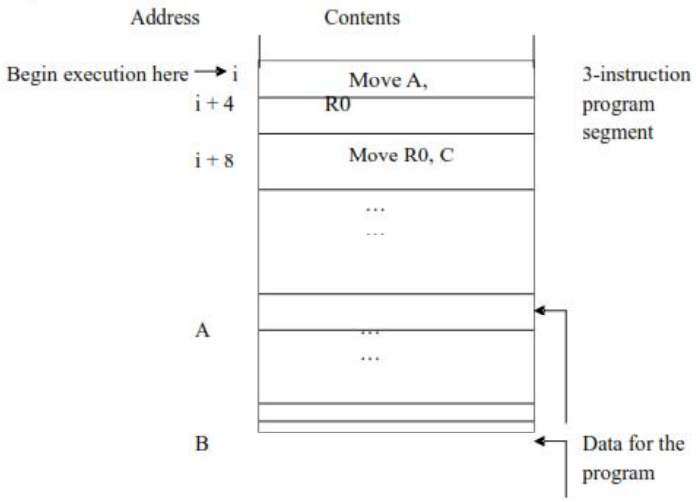
Move A, Ri
Add B, Ri
Move Ri, C

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

In the preceding discussion of instruction formats, we used to task [B]. fig 2.8 shows a possible program segment for this task as it appears in the memory of a computer. We have assumed that the computer allows one memory operand per instruction and has a number of processor registers. 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 this 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.

Executing a given instruction is a two-phase procedure. In the first phase, called instruction fetch, 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. The instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor. This often involves fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location.

Branching

Consider the task of adding a list of n numbers. Instead of using a long list of add instructions, it is possible to place a single add instruction in a program loop, as shown in fig b. 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.

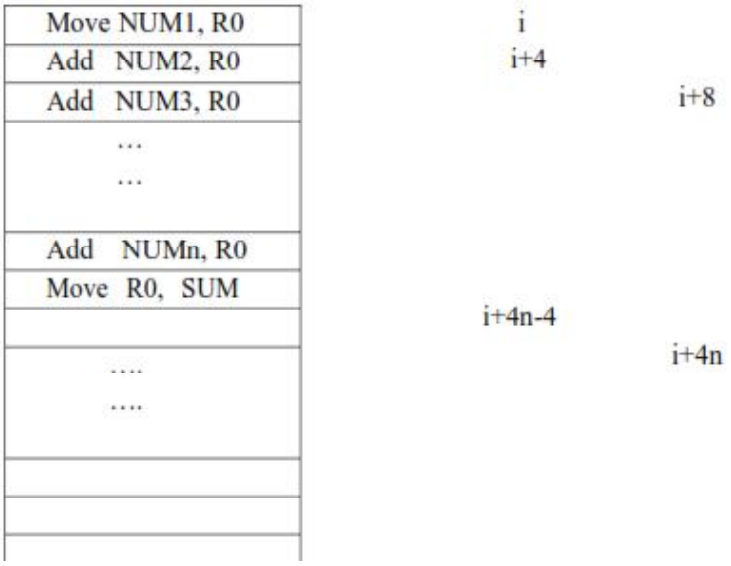


Figure a : A Straight-line program for adding n numbers

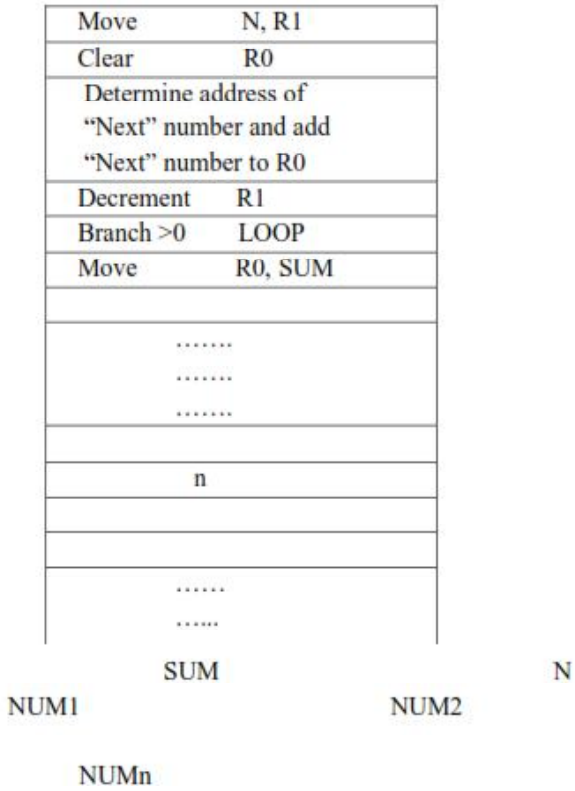


Figure b : Using a loop to add n numbers

Assume that the number of entries in the list, n, is stored in memory location N, as shown. Register R1 is used as a counter to determine the number of time the loop is executed. Hence, the contents of location N are loaded into register R1 at the beginning of the program. Then, within the body of the loop, the instruction. Decrement R1 Reduces the contents of R1 by 1 each time through the loop. This type of instruction loads a new value into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the branch target, instead of the instruction at the location that follows the branch instruction in sequential address order. A conditional branch instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in he normal way, and the next instruction in sequential address order is fetched and executed.

Branch > 0 LOOP

(branch if greater that 0) is a conditional branch instruction that causes a branch to location LOOP if the result of the immediately preceding instruction, which is the decremented value in register R1, is greater that zero. This means that the loop is repeated, as long as there are entries in the list that are yet to be added to R0. at the end of the nth pass through the loop, the Decrement instruction produces a value of zero, and hence, branching does not occur.

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(overflow) Set ot1 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 instruction Branch > 0, discussed in the previous section, is an example of a branch instruction that tests one or more of the condition flags. It causes a branch if the value tested is neither negative nor equal to zero. That is, the branch is taken if neither N nor Z is 1. The conditions are given as logic expressions involving the condition code flags.

In some computers, the condition code flags are affected automatically by instructions that perform arithmetic or logic operations. However, this is not always the case. A number of computers have two versions of an Add instruction.

Generating Memory Addresses

Let us return to fig b. The purpose of the instruction block at LOOP is to add a different number from the list during each pass through the loop. Hence, the Add instruction in the block must refer to a different address during each pass. How are the addresses to be specified ? The memory operand address cannot be given directly in a single Add instruction in the loop. Otherwise, it would need to be modified on each pass through the loop.

The instruction set of a computer typically provides a number of such methods, called addressing modes. While the details differ from one computer to another, the underlying concepts are the same

UNIT-2

Basic Structure of Computers: Computer Types, Functional Units, Basic Operational Concepts, Bus Structures, Performance - Processor Clock, Basic Performance Equation, Clock Rate, Performance Measurement, Historical Perspective.

Machine Instructions and Programs: Numbers, Arithmetic Operations and Characters, Memory Location and Addresses, Memory Operations, Instructions and Instruction Sequencing.

Unit-2

Machine Instructions and Programs

2.1 ADDRESSING MODES

In general, a program operates on data that reside in the computer's memory. These data can be organized in a variety of ways. If we want to keep track of students' names, we can write them in a list. Programmers use organizations called data structures to represent the data used in computations. These include lists, linked lists, arrays, queues, and so on.

Programs are normally written in a high-level language, which enables the programmer to use constants, local and global variables, pointers, and arrays. 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	Ri	EA = Ri
Absolute (Direct)	LOC	EA = LOC
Indirect	(Ri)	EA = [Ri]
	(LOC)	EA = [LOC]
Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri, Rj)	EA = [Ri] + [Rj]
Base with index and offset	X (Ri, Rj)	EA = [Ri] + [Rj] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(Ri)+	EA = [Ri]; Increment Ri
Autodecrement	-(Ri)	Decrement Ri; EA = [Ri]

EA = effective address
Value = a signed number

IMPLEMENTATION OF VARIABLE 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 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

Processor registers are used as temporary storage locations where the data is 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;

Immediate mode - The operand is given explicitly in the instruction.

For example, the instruction

Move 200, R0

Places the value 200 in register R0. Clearly, the Immediate mode is only used to specify the value of a source operand. Using a subscript to denote the Immediate mode is not appropriate in assembly languages. 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. Hence, we write the instruction above in the form

Move #200, R0

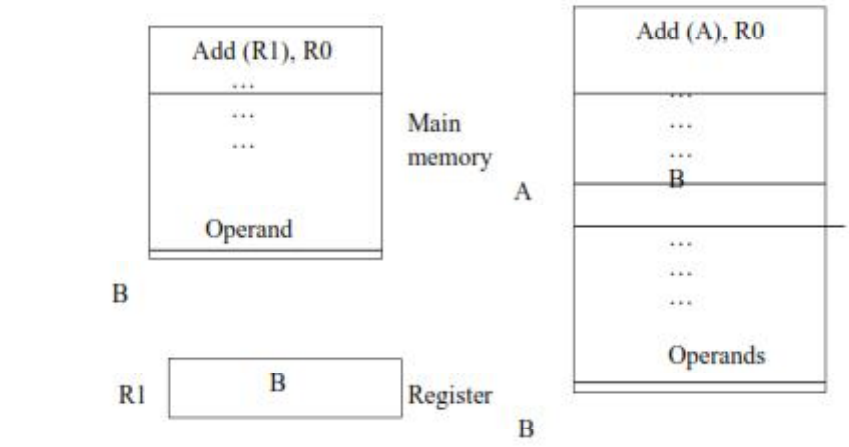
INDIRECTION AND POINTERS

In the addressing modes that follow, the instruction does not give the operand or its address explicitly, Instead, it 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

To execute the Add instruction in fig (a), the processor uses the value 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 fig (b). In this case, the processor first reads the contents of memory location A, then requests a second read operation using the value B as an address to obtain the operand.

Fig (a) Through a general-purpose register (b) Through a memory location



Address	Contents	
	Move	N, R1
	Move	#NUM, R2
	Clear	R0
LOOP	Add	(R2), R0
	Add	#4, R2
	Decrement	R1
	Branch > 0	LOOP
	Move	R0, SUM

The register or memory location that contains the address of an operand is called a pointer. Indirection and the use of pointers are important and powerful concepts in programming.

In the program shown 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 two instructions in the loop implement the unspecified instruction block starting at LOOP. The first time through the loop, the instruction **Add (R2), R0** fetches the operand at location NUM1and 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.

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

Indirect addressing through registers is used extensively. The above program shows the flexibility it provides. Also, when absolute addressing is not available, indirect addressing through registers makes it possible to access global variables by first loading the operand's address in a register.

Indexing and arrays

A different kind of flexibility for accessing operands 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 use 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 index register. We indicate the Index mode 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 + [Rj]

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 either as an explicit number or as a symbolic name representing a numerical value.

Fig a illustrates two ways of using the Index mode. In fig a, the index register, R1, 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 fig b. 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. In the most basic form of indexed addressing several variations of this basic form provide a very efficient access to memory operands in practical programming situations. For example, a second register may be used to contain the offset X, in which case we can write the Index mode as

(Ri, Rj)

Fig (a) Offset is given as a constant

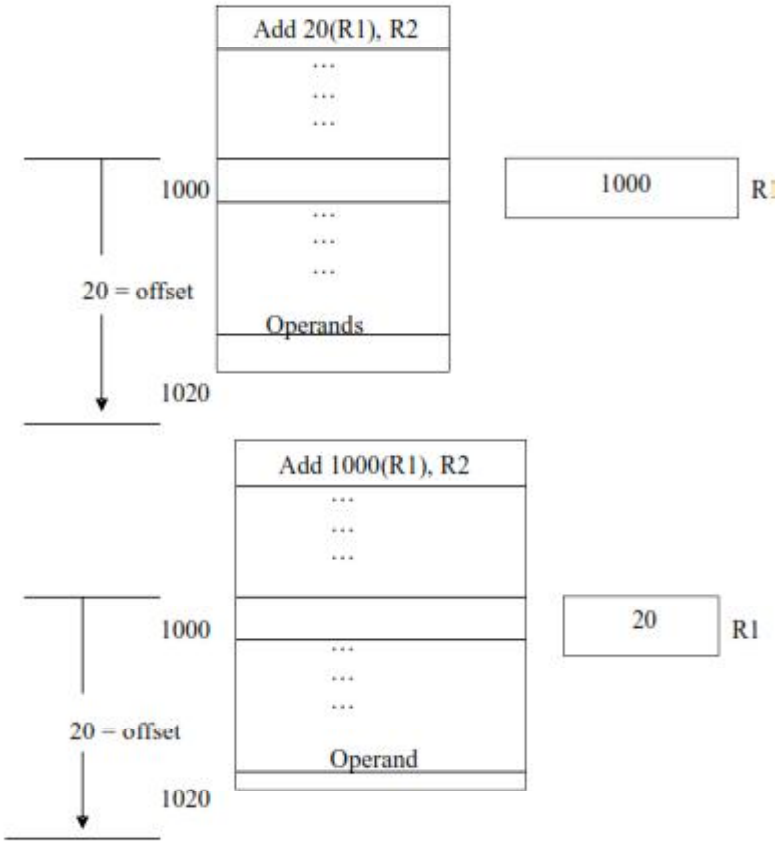


Fig (b) Offset is in the index register

```
        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
```

The effective address is the sum of the contents of registers Ri and Rj. The second register is usually called the base register. This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

Another version of the Index mode uses two registers plus a constant, which can be denoted as

X(Ri, Rj)

In this case, the effective address is the sum of the constant X and the contents of registers Ri and Rj. This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri, Rj) part of the addressing mode. In other words, this mode implements a three-dimensional array.

Relative Addressing

We have defined the Index mode using general-purpose processor registers. A useful version of this mode is obtained if the program counter, PC, is used instead of a general purpose register. 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.

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, its 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. This location can be computed by specifying it as an offset from the current value of the program counter. Since the branch target may be either before or after the branch instruction, the offset is given as a signed number.

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 to point to the next item in a list.

(Ri)+

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.

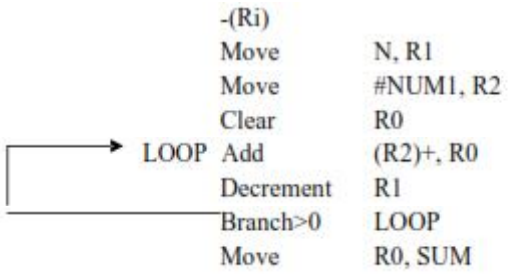


Fig c The Autoincrement addressing mode used in the program of fig 2.12

Assembly Language

Machine instructions are represented by patterns of 0s and 1s. Such patterns are awkward to deal with when discussing or preparing programs. Therefore, we use symbolic names to represent the pattern. So far, we have used normal words, such as Move, Add, Increment, and Branch, for the instruction operations to represent the corresponding binary code patterns. When writing programs for a specific computer, such words are normally replaced by acronyms called mnemonics, such as MOV, ADD, INC, and BR. Similarly, we use the notation R3 to refer to register 3, and LOC to refer to a memory location. A complete set of such symbolic names and rules for their use constitute a programming language, generally referred to as an assembly language.

Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an assembler. When the assembler program is executed, it reads the user program, analyzes it, and then generates the desired machine language program. The latter contains patterns of 0s and 1s specifying instructions that will be executed by the computer. The user program in its original alphanumeric text format is called a source program, and the assembled machine language program is called an object program.

Assembler Directives

In addition to providing a mechanism for representing instructions in a program, the assembly language allows the programmer to specify other information needed to translate the source program into the object program. We have already mentioned that we need to assign numerical values to any names used in a program. Suppose that the name SUM is used to represent the value 200. This fact may be conveyed to the assembler program through a statement such as

SUM EQU 200

This statement does not denote an instruction that will be executed when the object program is run; in fact, it will not even appear in the object program. It simply informs the assembler that the name SUM should be replaced by the value 200 wherever it appears in the program. Such statements, called assembler directives (or commands), are used by the assembler while it translates a source program into an object program.

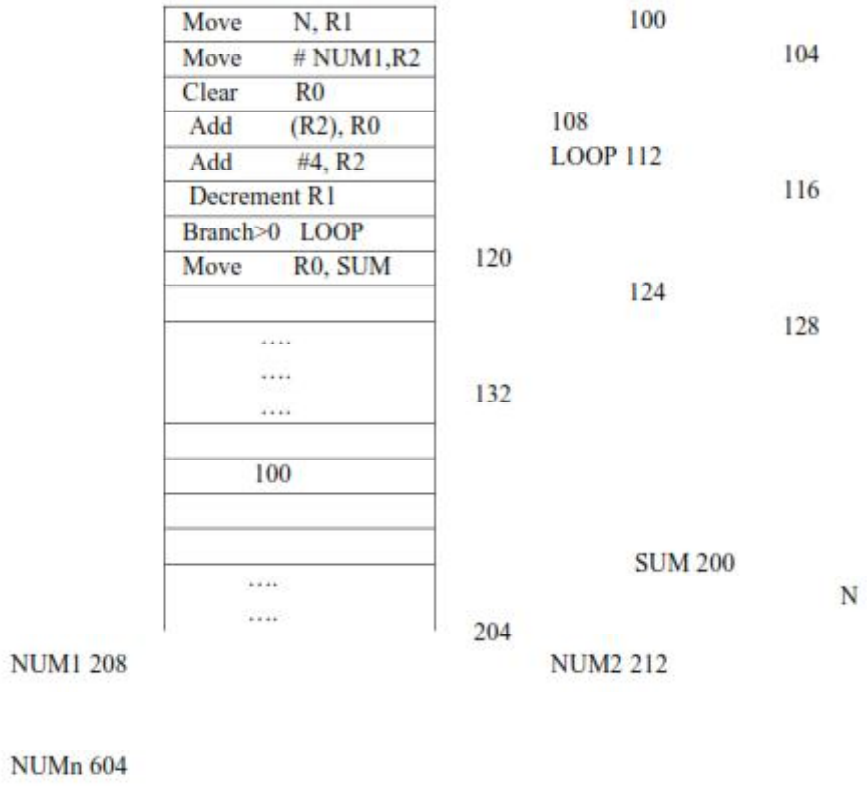


Fig 2.17 Memory arrangement for the program in fig b.

Assembly and Execution Programs

A source program written in an assembly language must be assembled into a machine language object program before it can be executed. This is done by the assembler program, which replaces all symbols denoting operations and addressing modes with the binary codes used in machine instructions, and replaces all names and labels with their actual values.

The assembler assigns addresses to instructions and data blocks, starting at the address given in the ORIGIN assembler directives. It also inserts constants that may be given in DATAWORD commands and reserves memory space as requested by RESERVE commands.

As the assembler scans through a source programs, it keeps track of all names and the numerical values that correspond to them in a symbol table. Thus, when a name appears a second time, it is replaced with its value from the table. A problem arises when a name appears as an operand before it is given a value. For example, this happens if a forward branch is required. A simple solution to this problem is to have the assembler scan through the source program twice. During the first pass, it creates a complete symbol table. At the end of this pass, all names will have

been assigned numerical values. The assembler then goes through the source program a second time and substitutes values for all names from the symbol table. Such an assembler is called a two-pass assembler.

The assembler stores the object program on a magnetic disk. The object program must be loaded into the memory of the computer before it is executed. For this to happen, another utility program called a loader must already be in the memory.

When the object program begins executing, it proceeds to completion unless there are logical errors in the program. The user must be able to find errors easily. The assembler can detect and report syntax errors. To help the user find other programming errors, the system software usually includes a debugger program. This program enables the user to stop execution of the object program at some points of interest and to examine the contents of various processor registers and memory locations.

Number Notation

When dealing with numerical values, it is often convenient to use the familiar decimal notation. Of course, these values are stored in the computer as binary numbers. In some situations, it is more convenient to specify the binary patterns directly. Most assemblers allow numerical values to be specified in different ways, using conventions that are defined by the assembly language syntax. Consider, for example, the number 93, which is represented by the 8-bit binary number 01011101. If this value is to be used an immediate operand, it can be given as a decimal number, as in the instructions.

ADD #93, R1

Or as a binary number identified by a prefix symbol such as a percent sign, as in

ADD #%01011101, R1

Binary numbers can be written more compactly as hexadecimal, or hex, numbers, in which four bits are represented by a single hex digit. In hexadecimal representation, the decimal value 93 becomes 5D. In assembly language, a hex representation is often identified by a dollar sign prefix. Thus, we would write ADD #$5D, R1 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.

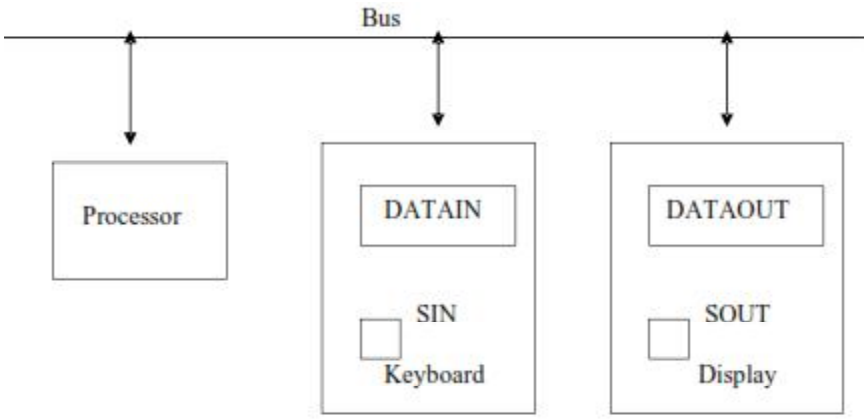


Fig a Bus connection for processor, keyboard, and display

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 stores 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 nform 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.

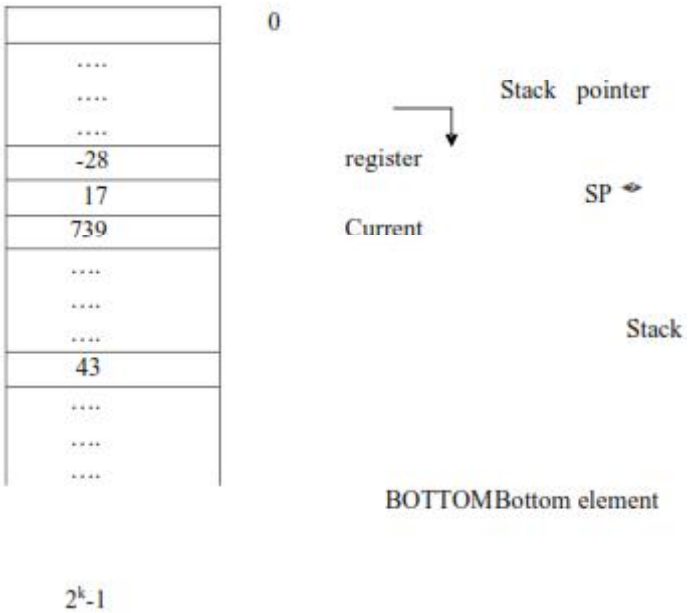


Fig b A stack of words in the memory

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.

Fig. a illustrates this procedure

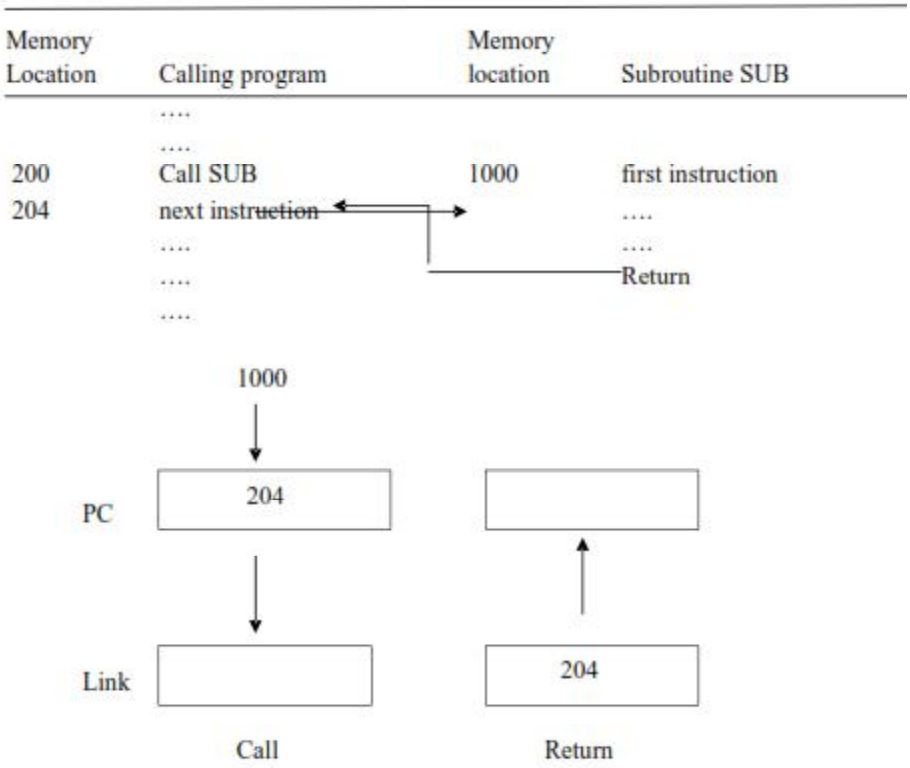


Fig b : Subroutine linkage using a ling register

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.

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.

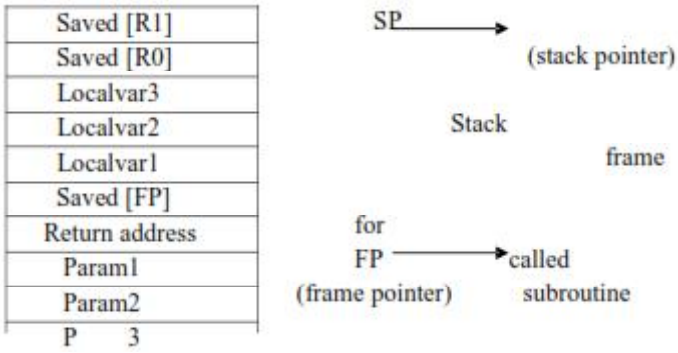


Fig b shows an example of a commonly used layout for information in a stack frame. In addition to the stack pointer SP, it is useful to have another pointer register, called the frame pointer (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used 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 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.

2.6 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

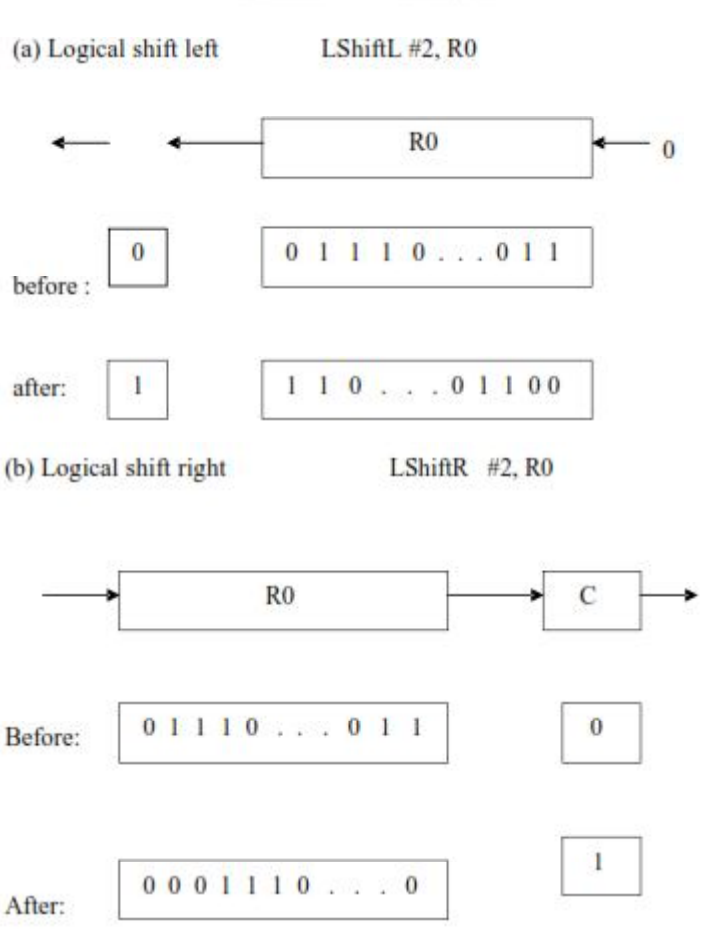
SHIFT AND ROTATE INSTRUCTIONS

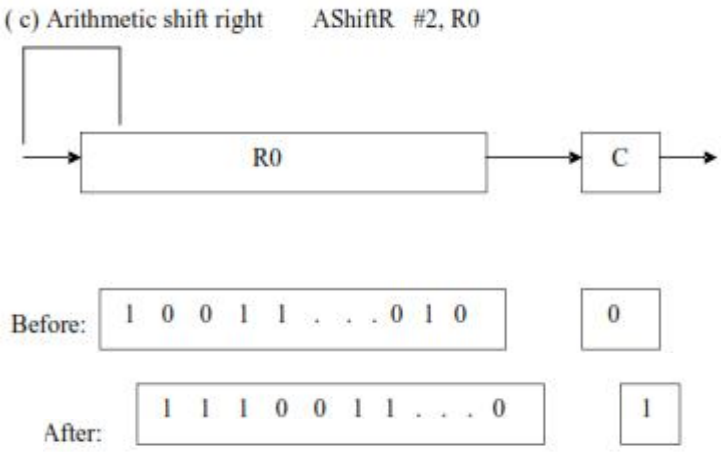
There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions. The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information. For general operands, we use 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 the local variables from the stack frame by executing the instruction. Add #12, SP

LShiftL count, dst

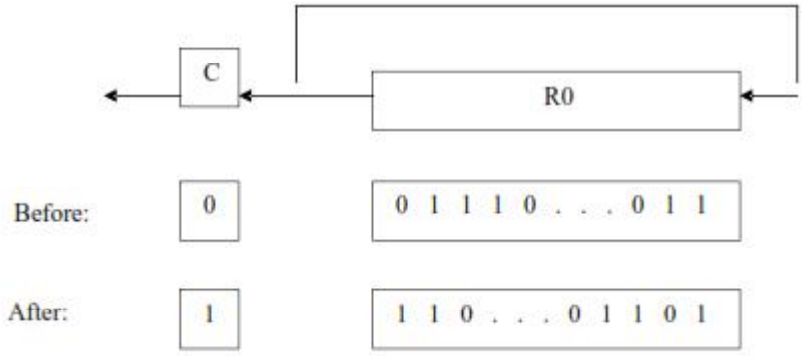




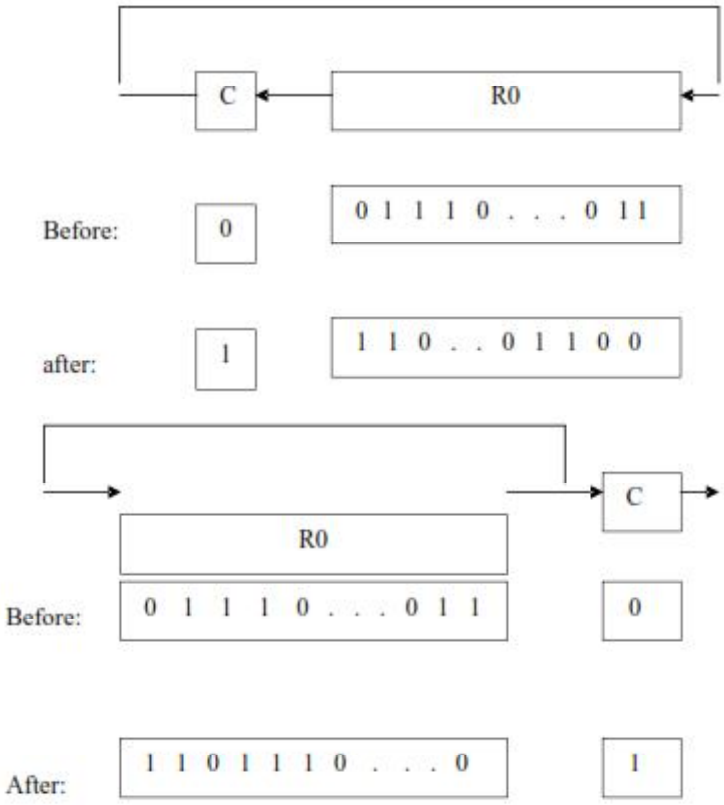
Rotate Operations

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.

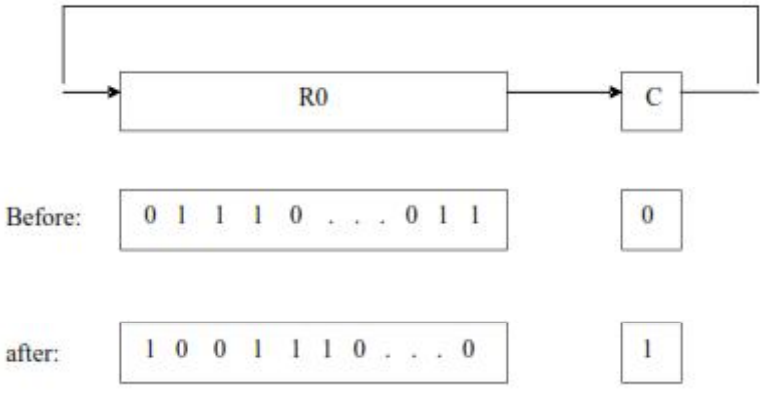
(a) Rotate left without carry RotateL #2, R0



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



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



2.7 ENCODING OF MACHINE INSTRUCTIONS

We have introduced a variety of useful instructions and addressing modes. These instructions

specify the actions that must be performed by the processor circuitry to carry out the desired tasks. We have often referred to them as machine instructions. Actually, the form in which we have presented the instructions is indicative of the form used in assembly languages, except that we tried to avoid using acronyms for the various operations, which are awkward to memorize and are likely to be specific to a particular commercial processor. To be executed in a processor, an instruction must be encoded in a compact binary pattern. Such encoded instructions are properly referred to as machine instructions. The instructions that use symbolic names and acronyms are called assembly anguage instructions, which are converted into the machine instructions using the assembler program.

We have seen instructions that perform operations such as add, subtract, move, shift, rotate, and branch. These instructions may use operands of different sizes, such as 32-bit and 8-bit numbers or 8-bit ASCII-encoded characters. The type of operation that is to be performed and the type of operands used may be specified using an encoded binary pattern referred to as the OP code for the given instruction. Suppose that 8 bits are allocated for this purpose, giving 256 possibilities for specifying different instructions.

This leaves 24 bits to specify the rest of the required information.
Let us examine some typical cases. The instruction
Add R1, R2

Has to specify the registers R1 and R2, in addition to the OP code. If the processor has 16 registers, then four bits are needed to identify each register. Additional bits are needed to indicate that the Register addressing mode is used for each operand. The instruction

Move 24(R0), R5

Requires 16 bits to denote the OP code and the two registers, and some bits to express that the source operand uses the Index addressing mode and that the index value is 24. The shift instruction

LShiftR #2, R0
And the move instruction
Move #$3A, R1

Have to indicate the immediate values 2 and #$3A, respectively, in addition to the 18 bits used to specify the OP code, the addressing modes, and the register. This limits the size of the immediate operand to what is expressible in 14 bits. Consider next the branch instruction Branch >0 LOOP

Again, 8 bits are used for the OP code, leaving 24 bits to specify the branch offset. Since the offset is a 2's-complement number, the branch target address must be within 223 bytes of the location of the branch instruction. To branch to an instruction outside this range, a different addressing mode has to be used, such as Absolute or Register Indirect. Branch instructions that use these modes are usually called Jump instructions.

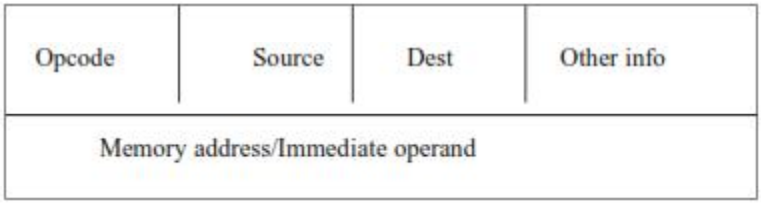
`In all these examples, the instructions can be encoded in a 32-bit word. Depicts a possible format. There is an 8-bit Op-code field and two 7-bit fields for specifying the source and destination operands. The 7-bit field identifies the addressing mode and the register involved (if any). The "Other info" field allows us to specify the additional information that may be needed, such as an index value or an immediate operand.

But, what happens if we want to specify a memory operand using the Absolute addressing mode? The instruction

Move R2, LOC

(a) One-word instruction

(b) Two-Word instruction



(c) Three-operand instruction

Requires 18 bits to denote the OP code, the addressing modes, and the register. This leaves 14 bits to express the address that corresponds to LOC, which is clearly insufficient.

And #$FF000000. R2

In which case the second word gives a full 32-bit immediate operand.

If we want to allow an instruction in which two operands can be specified using the Absolute addressing mode, for example

Move LOC1, LOC2

Then it becomes necessary to use tow additional words for the 32-bit addresses of the operands.

This approach results in instructions of variable length, dependent on the number of operands

and the type of addressing modes used. Using multiple words, we can implement quite complex instructions, closely resembling operations in high-level programming languages. The term complex instruction set computer (CISC) has been used to refer to processors that use instruction sets of this type.

The restriction that an instruction must occupy only one word has led to a style of computers that have become known as reduced instruction set computer (RISC). The RISC approach introduced other restrictions, such as that all manipulation of data must be done on operands that are already in processor registers. This restriction means that the above addition would need a two-instruction sequence

Move (R3), R1
Add R1, R2

If the Add instruction only has to specify the two registers, it will need just a portion of a 32-bit word. So, we may provide a more powerful instruction that uses three operands

Add R1, R2, R3
Which performs the operation
R3 □ [R1] + [R2]

A possible format for such an instruction in shown in fig c. Of course, the processor has to be able to deal with such three-operand instructions. In an instruction set where all arithmetic and logical operations use only register operands, the only memory references are made to load/store the operands into/from the processor registers.

RISC-type instruction sets typically have fewer and less complex instructions than CISC-type sets. We will discuss the relative merits of RISC and CISC approaches in Chapter 8, which deals with the details of processor design.

UNIT-3

Input/output Organization: Accessing I/O Devices, Interrupts - Interrupt Hardware, Enabling and Disabling Interrupts, Handling Multiple Devices, Controlling Device Requests, Exceptions, Direct Memory Access, Buses.

Unit-3

The System Memory

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

Move DATAIN, R0

Reads the data from DATAIN 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.

Most computer systems use memory-mapped I/O. some processors have special In and Out instructions to perform I/O transfers. When building a computer system based on these processors, the designer had the option of connecting I/O devices to use the special I/O address space or simply incorporating them as part of the memory address space. The I/O devices examine the low-order bits of the address bus to determine whether they should respond.

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.

I/O devices operate at speeds that are vastly different from that of the processor. When a human operator is entering characters at a keyboard, the processor is capable of executing millions of instructions between successive character entries. An instruction that reads a character from the keyboard should be executed only when a character is available in the input buffer of the keyboard interface. Also, we must make sure that an input character is read only once.

This example illustrates program-controlled I/O, in which 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. The routine executed in response to an interrupt request is called the interrupt-service routine, which is the PRINT routine in our example. Interrupts bear considerable resemblance to subroutine calls. Assume that an interrupt request arrives during execution of instruction i in figure 1.

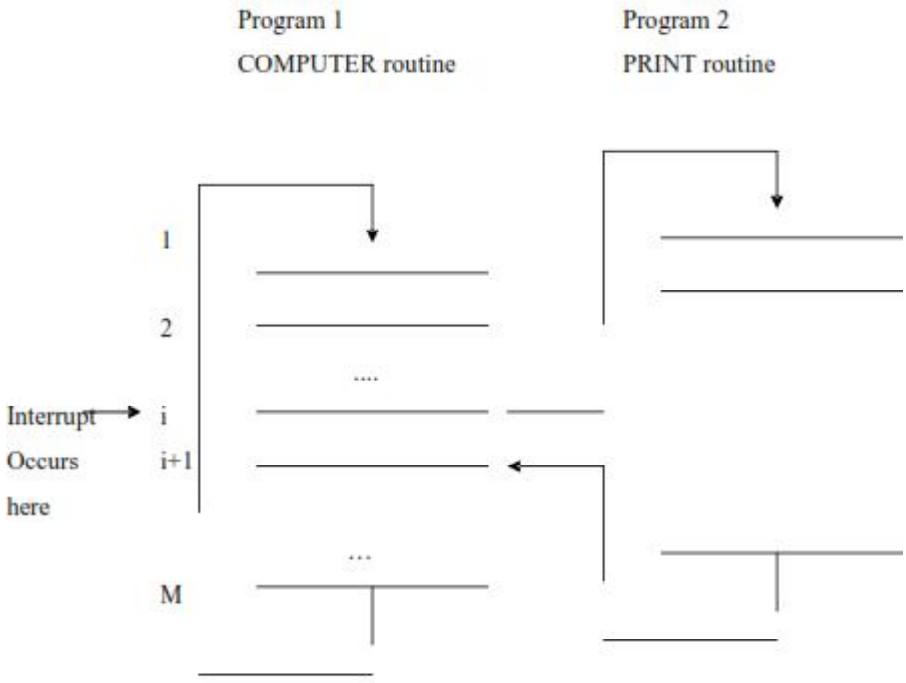


Figure1 : Transfer of control through the use of interupts

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 the temporary storage location, causing execution to resume at instruction i +1. In many processors, the return address is saved on the processor stack. We should note that as part of handling interrupts, 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. The execution of an instruction in the interrupt-service routine that accesses a status or data register in the device interface implicitly informs that device that its interrupt request has been recognized.

So far, treatment of an interrupt-service routine is very similar to that of a subroutine. An important departure from this similarity should be noted. A subroutine performs a function required by the program from which it is called. However, the interrupt-service routine may not have anything in common with the program being executed at the time the interrupt request is received. In fact, the two programs often belong to different users. Therefore, before starting execution of the interrupt-service routine, any information that may be altered during the execution of that routine must be saved. This information must be restored before execution of the interrupt program is resumed. In this way, the original program can continue execution without being affected in any way by the interruption, except for the time delay. The information that needs to be saved and restored typically includes the condition code flags and the contents of any registers used by both the interrupted program and the interrupt-service routine. 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 registers involves memory transfers that increase the total execution time, and hence represent execution overhead. Saving registers also increase 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.

3.2 INTERRUPT HARDWARE

We pointed out that an I/O device requests an interrupt by activating a bus line called interrupt-request. Most computers are likely to have several I/O devices that can request an interrupt. A single interrupt-request line may be used to serve n devices as depicted. 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

are inactive, that is, if all switches are open, the voltage on the interrupt-request line will be equal to. This is the inactive state of the line. 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

+ ……+INTR

It is customary to use the complemented form,

n

signal on the common line, because this signal is active when in the low-voltage state.

INTR

3.3 ENABLING AND DISABLING INTERRUPTS:

, to name the interrupt-request

The facilities provided in a computer must give the programmer complete control over the events that take place during program execution. 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 the envisaged by the programmer. Hence, the interruption of program execution must be carefully controlled. Let us consider in detail the specific case of a single interrupt request from one device. 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. 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 processor must guarantee that execution of the Return-from-interrupt instruction is completed before further interruption can occur. 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. 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.

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.

3.4 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 in 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 interrupts?
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? 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?

   The means by which these problems are resolved vary from one computer to another, And the approach taken is an important consideration in determining the computer's suitability for a given application. When a request is received over the common interrupt-request line, additional information is needed to identify the particular device that activated the line.

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. For example, bits KIRQ and DIRQ are the interrupt request bits for the keyboard and the display, respectively.

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, which we describe next.

VECTORED INTERRUPTS

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. The term vectored interrupts refers to all interrupt-handling schemes based on this approach.

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.

This arrangement implies that the interrupt-service routine for a given device must always start at the same location. The programmer can gain some flexibility by storing in this location an instruction that causes a branch to the appropriate routine.

INTERRUPT NESTING

Interrupts should be disabled during the execution of an interrupt-service routine, to ensure that a request from one device will not cause more than one interruption. The same arrangement is often used when several devices are involved, in which case execution of a given interrupt-service routine, once started, always continues to completion before the processor accepts an interrupt request from a second device. Interrupt-service routines are typically short, and the delay they may cause is acceptable for most simple devices.

For some devices, however, a long delay in responding to an interrupt request may lead to erroneous operation. Consider, for example, a computer that keeps track of the time of day using a real-time clock. This is a device that sends interrupt requests to the processor at regular intervals. For each of these requests, the processor executes a short interrupt-service routine to increment a set of counters in the memory that keep track of time in seconds, minutes, and so on. Proper operation requires that the delay in responding to an interrupt request from the real-time clock be small in comparison with the interval between two successive requests. To ensure that this requirement is satisfied in the presence of other interrupting devices, it may be necessary to accept

an interrupt request from the clock during the execution of an interrupt-service routine for another device.

This example suggests that 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.

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 privileged instruction.

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

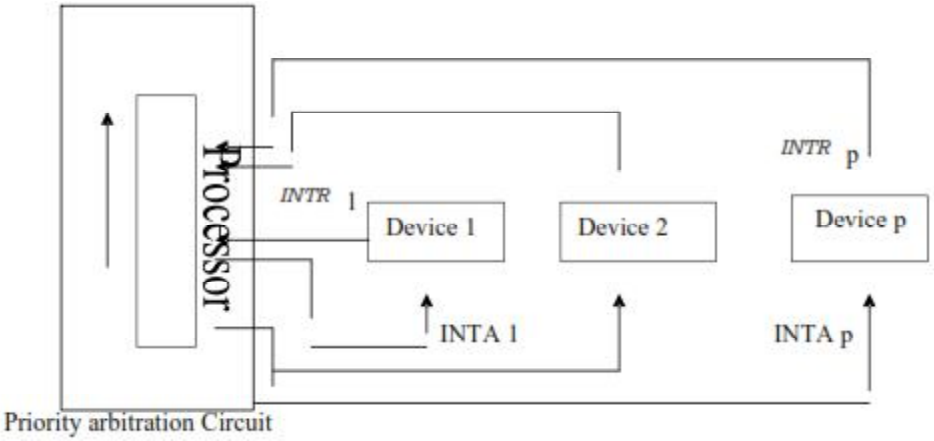
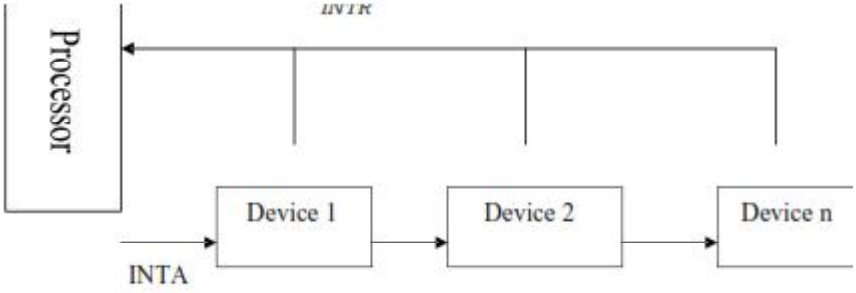


Figure 2 : Implementation of interrupt priority using individual interrupt-request and acknowledge lines

Let us now consider the problem of simultaneous arrivals of interrupt requests from two or more devices. The processor must have some means of deciding which requests to service first. Using a priority scheme such as that of figure, the solution is straightforward. The processor simply accepts the requests having the highest priority. Polling the status registers of the I/O devices is the simplest such mechanism. 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 daisyhain, as shown in figure 3a. The interrupt-request line is common to all devices.



(3.a) Daisy chain

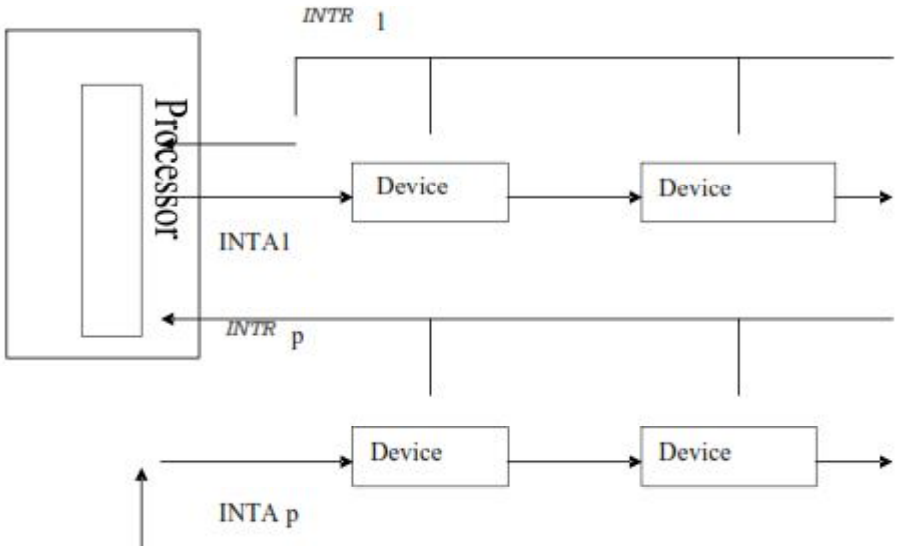


Fig. 3.b : Arrangement of Priority Groups

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.

The scheme in figure 3.a requires considerably fewer wires than the individual connections in figure 2. The main advantage of the scheme in figure 2 is that it allows the processor to accept interrupt requests from some devices but not from others, depending upon their priorities. The two schemes may be combined to produce the more general structure in figure 3b. 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.

3.5 CONTROLLING DEVICE REQUESTS

Until now, we have assumed that an I/O device interface generates an interrupt request whenever it is ready for an I/O transfer, for example whenever the SIN flag is 1. It is important to ensure that interrupt requests are generated only by those I/O devices that are being used by a given program. Idle devices must not be allowed to generate interrupt requests, even though they may be ready to participate in I/O transfer operations. Hence, we need a mechanism 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 perform this function. 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.

3.6 EXCEPTIONS

An interrupt is an event that causes the execution of one program to be suspended and the execution of another program to begin. So far, we have dealt only with interrupts caused by requests received during I/O data transfers. However, the interrupt mechanism is used in a number of other situations.

The term exception is often used to refer to any event that causes an interruption. Hence, I/O interrupts are one example of an exception. We now describe a few other kinds of exceptions.

Recovery 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 errors occur, 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. For example, the OP-code field of an instruction may not correspond to any legal instruction, or an arithmetic instruction may attempt a division by zero.

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. Recall that in the case of an I/O interrupt, the processor completes execution of the instruction in progress before accepting the interrupt. 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

Another important type of exception is used as an aid in debugging programs. 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, then the debugging program is activated again. The trace exception is disabled during the execution of the debugging program.

Breakpoint provides 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 a 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 programs, certain instructions can be executed only while the processor is in supervisor mode. These are called privileged instructions. For example, when the processor is running in the user mode, it will not execute an instruction that changes the priority level of the processor or that enables a user program to access areas in the computer memory that have been allocated to other users. An attempt to execute such an instruction will produce a privilege exceptions, causing the processor to switch to the supervisor mode and begin executing an appropriate routine in the operating system.

3.7 DIRECT MEMORY ACCESS

The discussion in the previous sections concentrates on data transfer between the processor and I/O devices. Data are transferred by executing instructions such as

Move DATAIN, R0

An instruction to transfer input or output data is executed only after the processor determines that the I/O device is ready. To do this, the processor either polls a status flag in the device interface or waits for the device to send an interrupt request. In either case, considerable overhead is incurred, because several program instructions must be executed for each data word transferred. In addition to polling the status register of the device, instructions are needed for incrementing the memory address and keeping track of the word count. When interrupts are used, there is the additional overhead associated with saving and restoring the program counter and other state information.

To transfer large blocks of data at high speed, an alternative approach is used. A special control unit may be provided to allow transfer of a block of data directly between n 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 I/O device interface. We refer to this circuit as a 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 must increment the memory address for successive words and keep track of the number of transfers.

Although a DMA controller can transfer data without intervention by the processor, its operation must be under the control of a program executed by the processor. 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.

While a DMA transfer is taking place, the program that requested the transfer cannot continue, and the processor can be used to execute another program. After the DMA transfer is completed, the processor can return to the program that requested the transfer.

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.

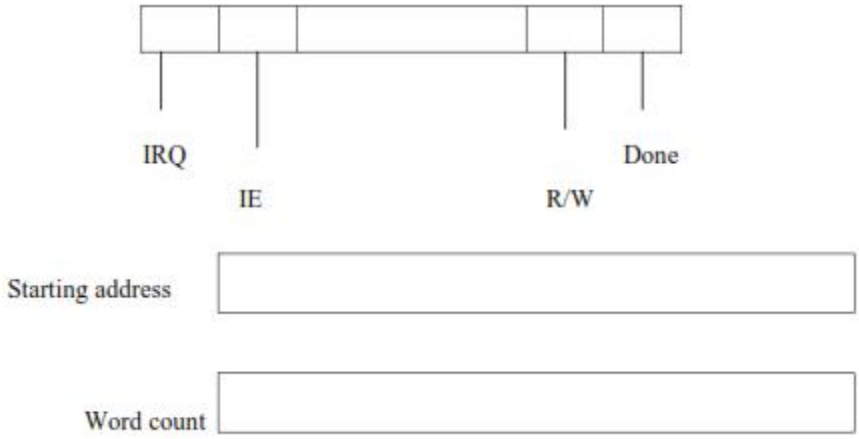


Figure 4 : Registers in DMA interface

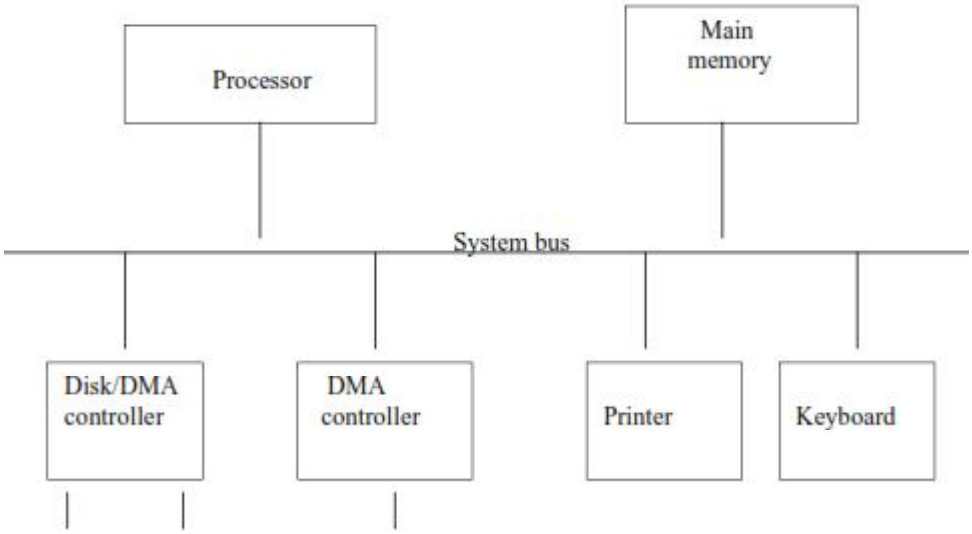


Figure 5 : Use of DMA controllers in a computer system

Figure 4 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 Status and Control Starting address and the word count. The third register contains status and control flags. The R/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. 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.

An example of a computer system is given in above figure, showing 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.

Disk
Disk

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. The DMA controller proceeds independently to implement the specified operation. When the DMA transfer is completed. This fact is recorded in the status and control register of the DMA channel byetting the Done bit. At the same time, 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 controller are interwoven. Requests 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, the 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.

Most DMA controllers incorporate a data storage buffer. In the case of the network interface in figure 5 for example, the DMA controller reads a block of data from the main memory and stores

it into its input buffer. This transfer takes place using burst mode at a speed appropriate to the memory and the computer bus. Then, the data in the buffer are transmitted over the network at the speed of the network.

A conflict may arise if both the processor and a DMA controller or two DMA controllers try to use the bus at the same time to access the main memory. To resolve these conflicts, an arbitration procedure is implemented on the bus to coordinate the activities of all devices requesting memory transfers.

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 he 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

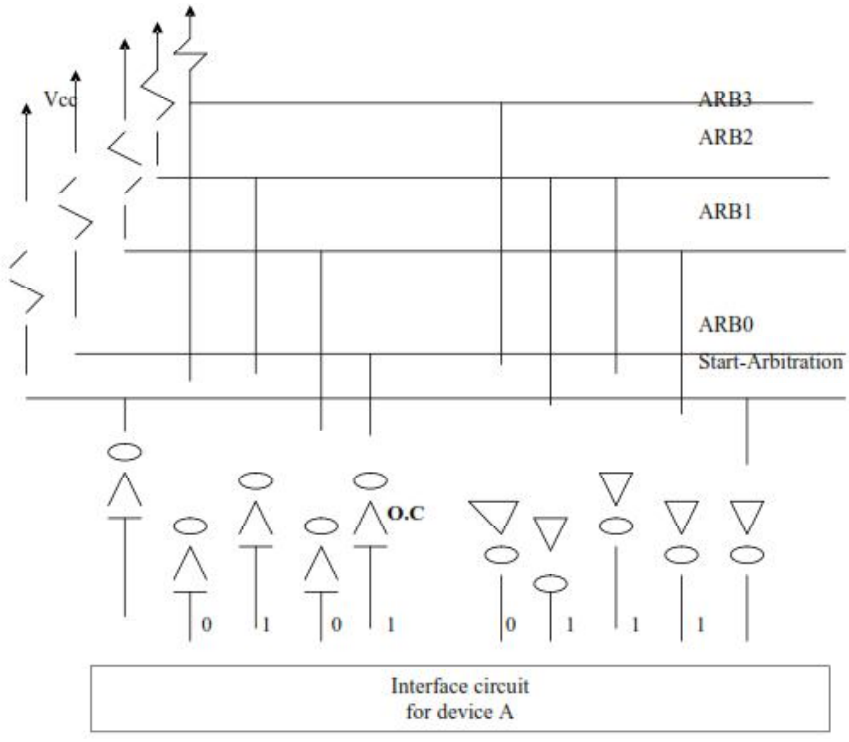


Fig 6 A distributed arbitration

The bus arbiter may be the processor or a separate unit connected to the bus. 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. 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 controller 1 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 device that it is using the bus by activating another open-controller line called Bus-Busy. 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.

Distributed arbitration means that all devices waiting to use the bus have equal responsibility in carrying out the arbitration process, without using a central arbiter. A simple method for distributed arbitration is illustrated in figure 6. Each device on the bus assigned a 4-bit identification number. When one or more devices request the bus, they assert the

Start?Arbitration

signal and place their 4-bit ID numbers on four open-

3.8 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 communication path for the transfer of data. The bus includes the lines needed to support interrupts and arbitration. In this section, we discuss the main features of the bus protocols used for transferring data. A bus protocol is the set of rules that govern the behavior of various devices connected to the bus as to when to place information on the bus, assert control signals, and so on. After describing bus protocols, we will present examples of interface circuits that use these protocols.

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. Such a scheme is illustrated in figure 7 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

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 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 = 0.

It is important that slaves take no action or place any data on the bus before t. The information on the bus is unreliable during the period t = 0, to t because signals are changing state. The addressed slave places the requested input data on the data lines at time t = 1

At the end of the clock cycle, at time t2, the master strobes the data on the data lines into its input buffer. In this context, "strobe" means to capture the values of the ata of a given instant and store them into a buffer. For data to be loaded correctly into any storage device, such as a register built with flip-flops, the data must be available at the input of that device for a period greater than the setup time of the device. Hence, the A similar procedure is followed for an output operation. The master places the output data on the data lines when it transmits the address and command information at time t, the addressed device strobes the data lines and loads the data into its data buffer.

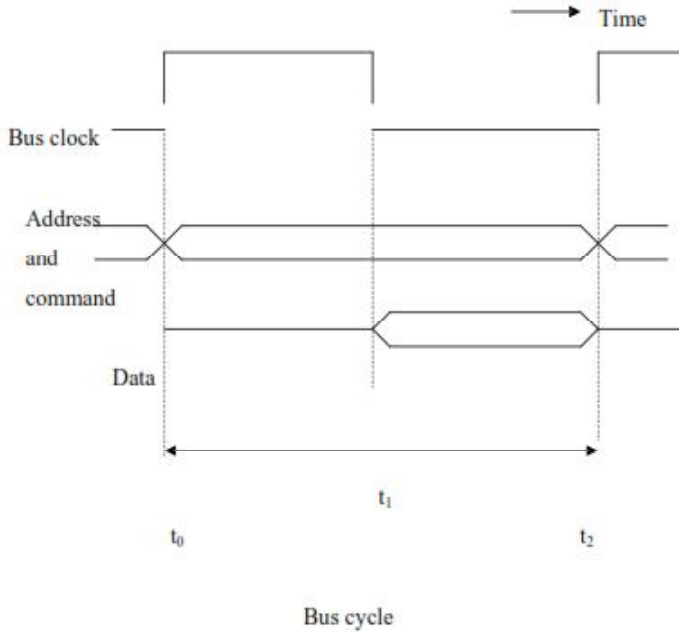


Figure 7 : Timing of an input transfer on a synchronous bus.

2

The timing diagram in figure 7 is an idealized representation of the actions that take place on the bus lines. The exact times at which signals actually change state are somewhat different from those shown because of propagation delays on bus wires and in the circuits of the devices. 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.

The master sends the address and command signals on the rising edge at the beginning of clock period 1 (t). However, these signals do not actually appear on the bus

until f

AM

0, largely due to the delay in the bus driver circuit. A while later, at t, the signals reach the slave. The slave decodes the address and at t1 sends the requested data.

Here again, the data signals do not appear on the bus until t. They travel toward the master and arrive at tMDS. At t2, the master loads the data into its input buffer. Hence the period t2-t is the setup time for the master's input buffer. The data must continue to be valid after tDM for a period equal to the hold time of that buffer. 2

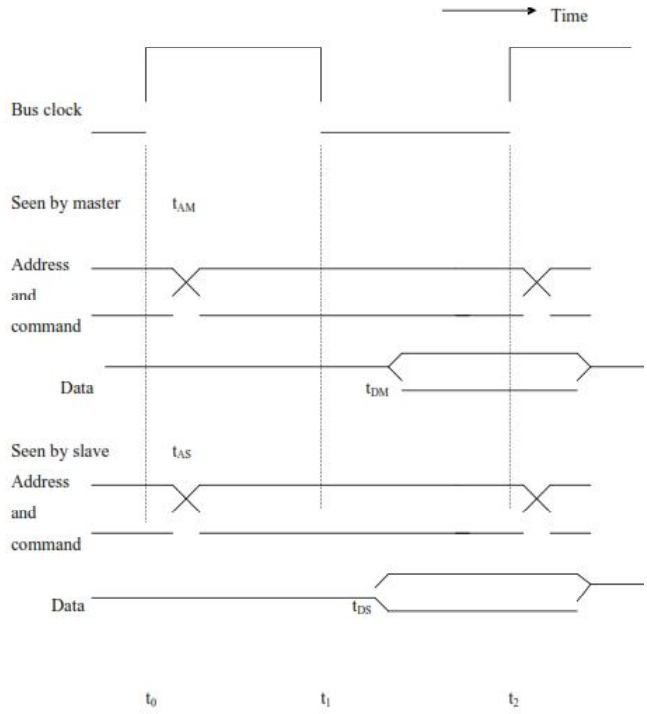


Figure 8 A detailed timing diagram for the input transfer of figure 7

Multiple-Cycle transfers

The scheme described above results in a simple design for the device interface, however, it has some limitations. Because a transfer has to be completed within one clock cycle, the clock period, t2-, must be chosen to accommodate the longest delays on the us and the lowest device interface. This forces all devices to operate at the speed of the slowest device.

Also, the processor has no way of determining whether the addressed device has actually responded. It simply assumes that, at t, the output data have been received by the I/O device or the input data are available on the data lines. If, because of a malfunction, the device does not respond, the error will not be detected.

2

To overcome these limitations, 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.

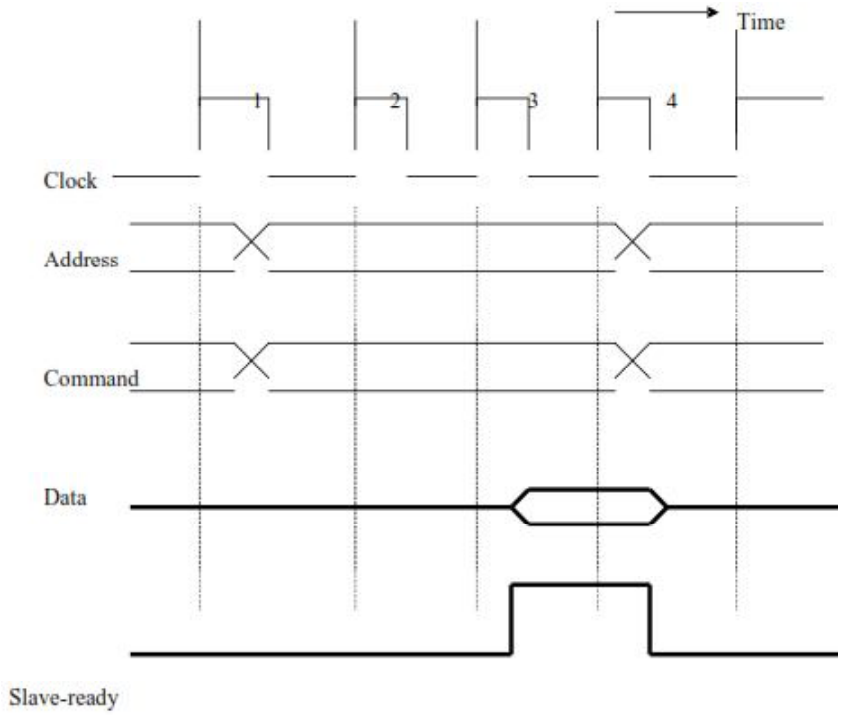


Figure 9 : An input transfer using multiple clock cycles

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. We have assumed that some delay is involved in getting the data, and hence the slave cannot respond immediately. 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 Slave-ready signal is an acknowledgment from the slave to the master, confirming that valid data have been sent. In the example in figure 9, 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, then aborts the operation. This could be the result of an incorrect address or a device malfunction.

ASYNCHRONOUS BUS

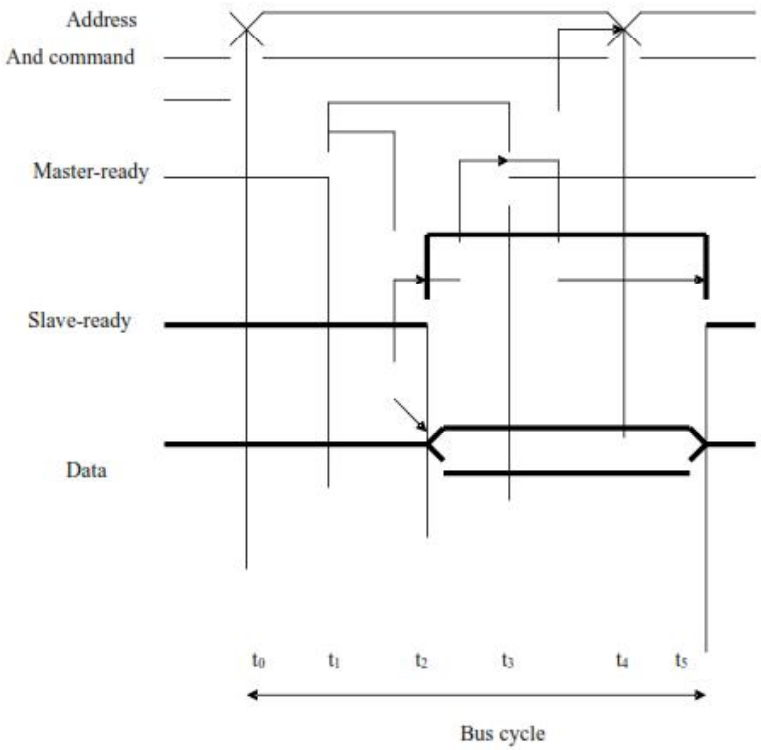
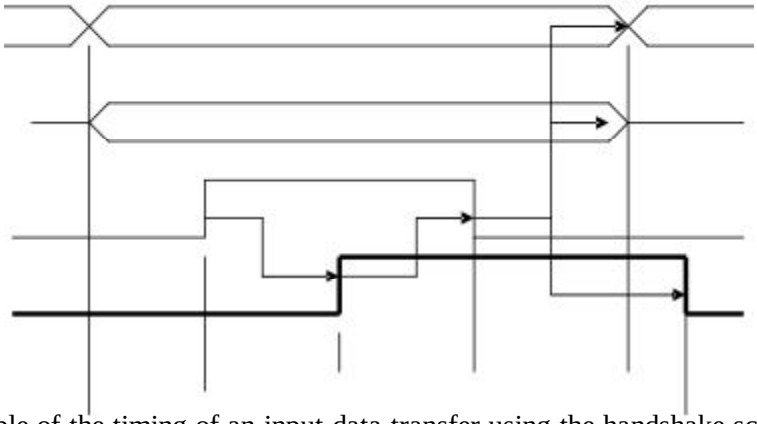


Figure 10 : Handshake control of data transfer during an input operation.

An alternative scheme for controlling data transfers on the bus is based on the use of a handshake between the master and the salve. The concept of a handshake is a generalization of

the idea of the Slave-ready signal in figure 10. The common clock is replaced by two timing control lines, Master-ready and Slave-ready. 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.

n 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 - The master places the address and command information on the bus, and all devices on the bus begin to decode this information.

t

0

- 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

1

-t

is intended to allow for any skew that may occur o the bus. Skew occurs when two signals simultaneously transmitted from one source arrive at the destination at different times. This happens

because different lines of the bus may have different propagation speeds. Thus, to guarantee that the Master-ready signal does not arrive at any device ahead of the address and command information, the delay t1-00 should be larger than the maximum possible bus skew. t - 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. t2 - The Slave-ready signal arrives at the master, indicating that the input data are available on the bus. t3 - The master removes the address and command information from the bus. The delay between t43and t is again intended to allow for bus skew. t5 - 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.

UNIT-4

Input/Output Organization contd.: Interface Circuits, Standard I/O Interfaces - PCI Bus, SCSI Bus, USB

Unit-4

Input/Output Organization Contd.

4.1 INTERFACE CIRCUITS

Parallel port

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.

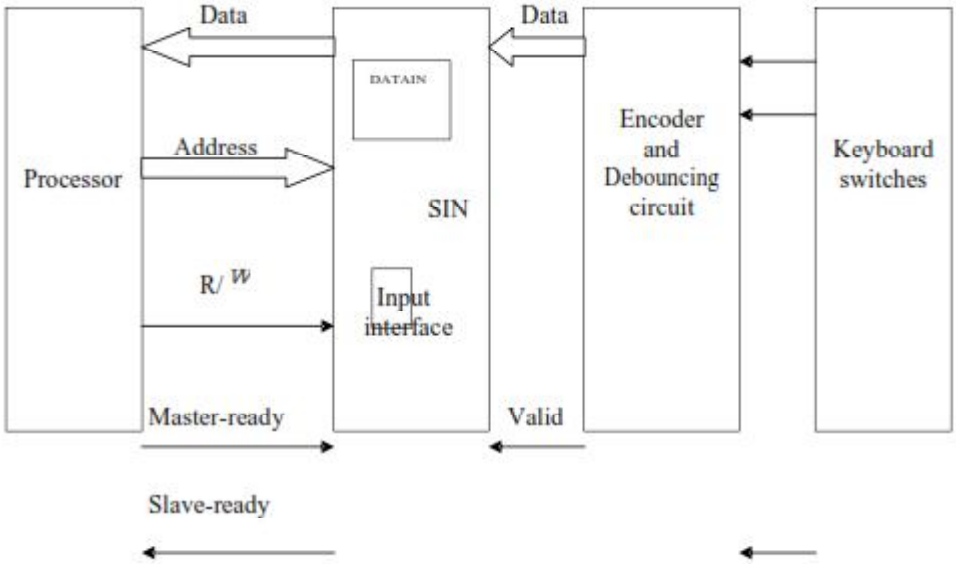


Figure 11 : Keyboard to processor connection.

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 Valid 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, as indicated in figure 11. The third control line, R/ distinguishes read and write transfers.

Figure 12 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 D0, 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.

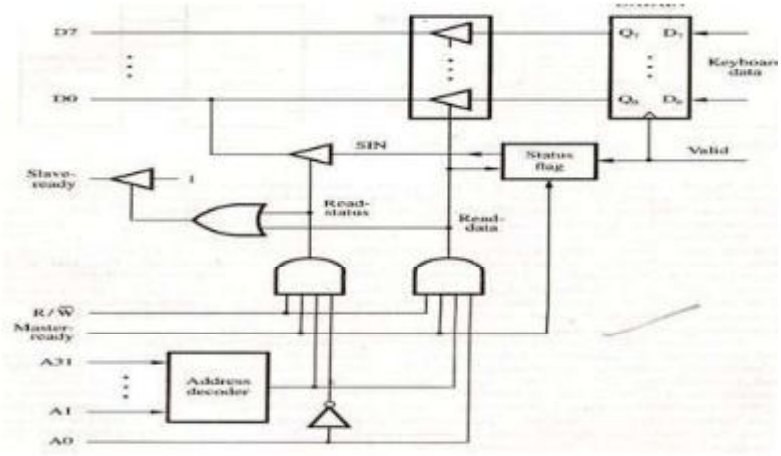
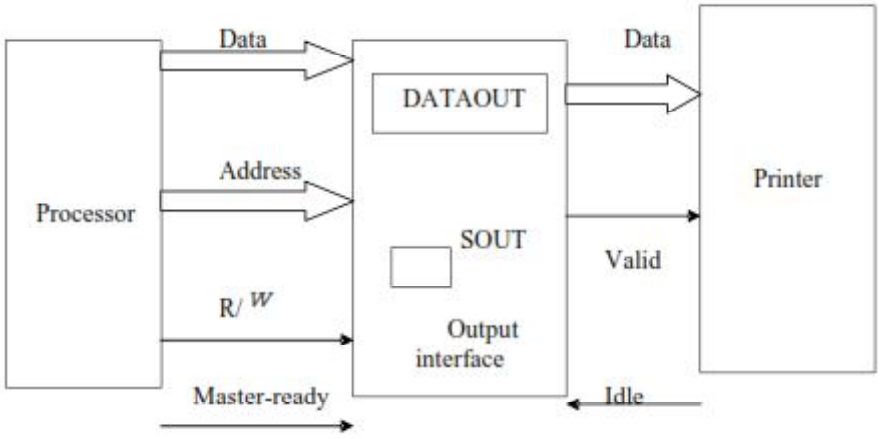


Figure 12 : Input Interface Circuit



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.

Let us now consider an output interface that can be used to connect an output device, such as a printer, to a processor, as shown in figure 13. The printer operates under control of the handshake signals Valid and Idle in a manner similar to the handshake used on the bus with the Master-ready and Slave-ready signals. 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 circuit in figure 16 has separate input and output data lines for connection to an I/O device. A more flexible parallel port is created if the data lines to I/O devices are bidirectional. Figure 17 shows a general-purpose parallel interface circuit that can be configured in a variety of ways. 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.

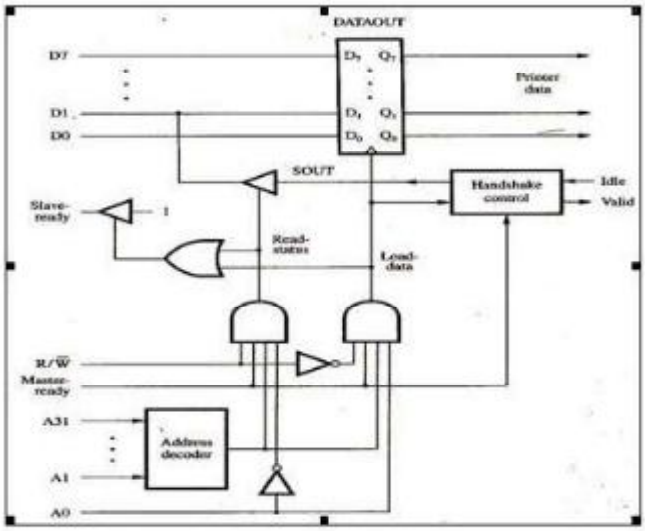


Figure 14 : Output interface circuit

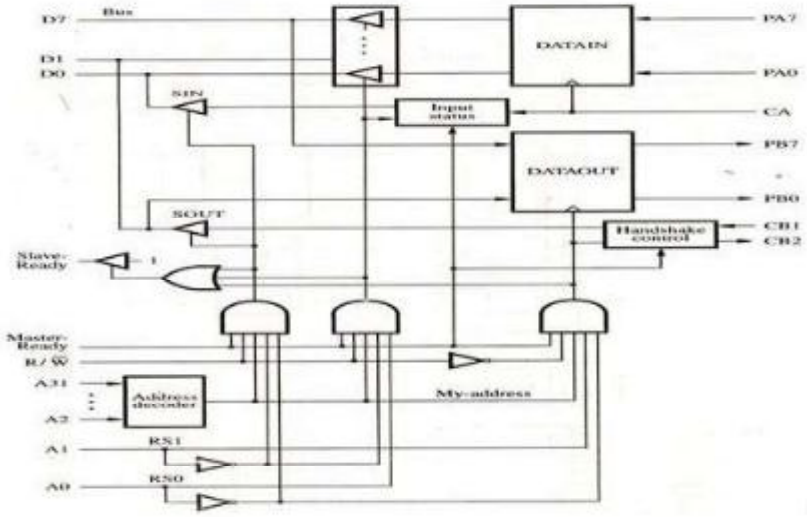
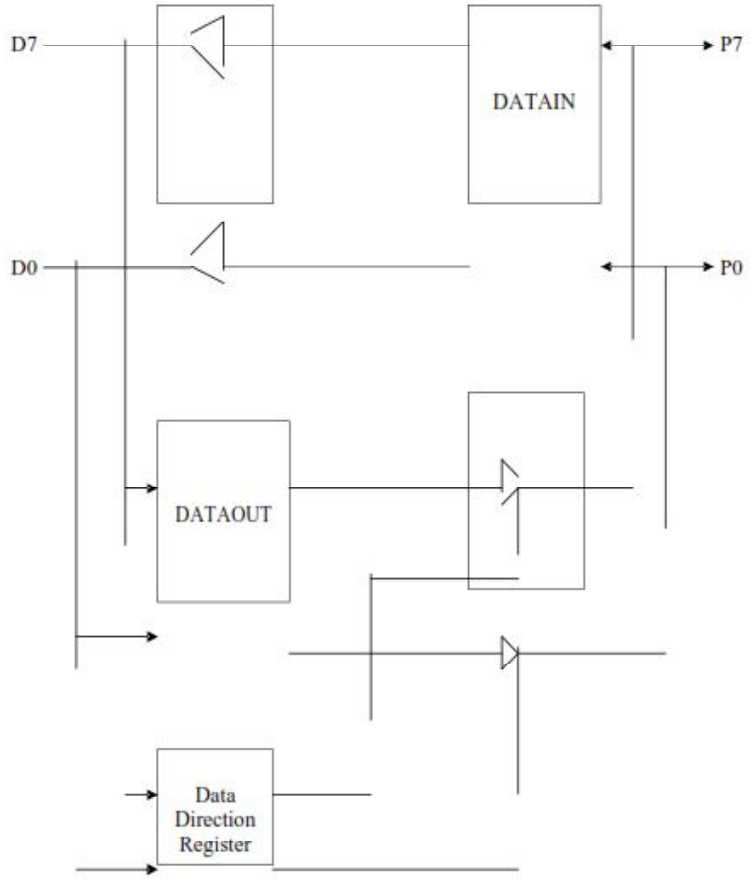


Figure 15 : Combined input/output interface output



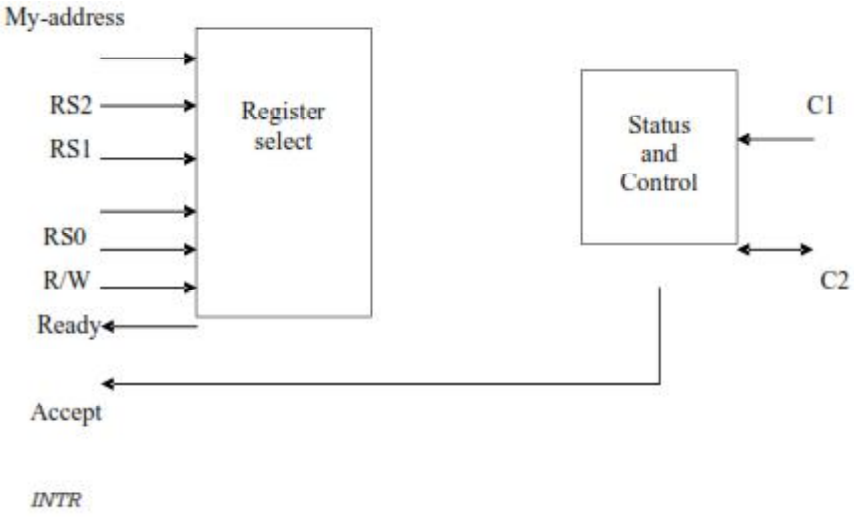


Figure 16 : A general 8-bit parallel interface

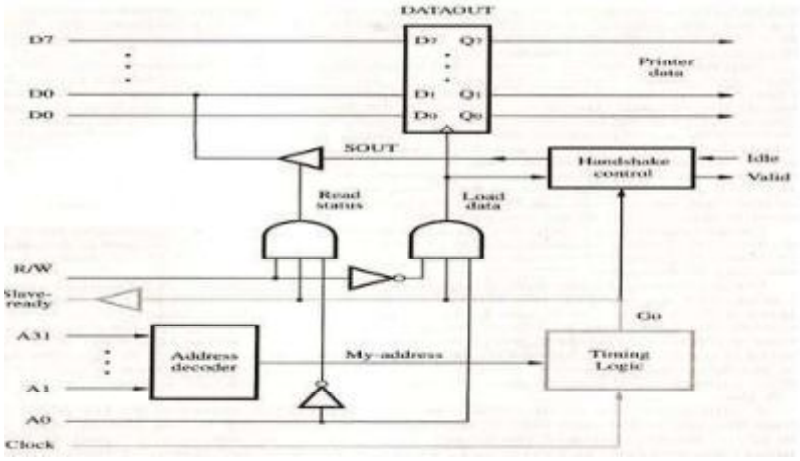


Figure 17 : A parallel port interface for the bus

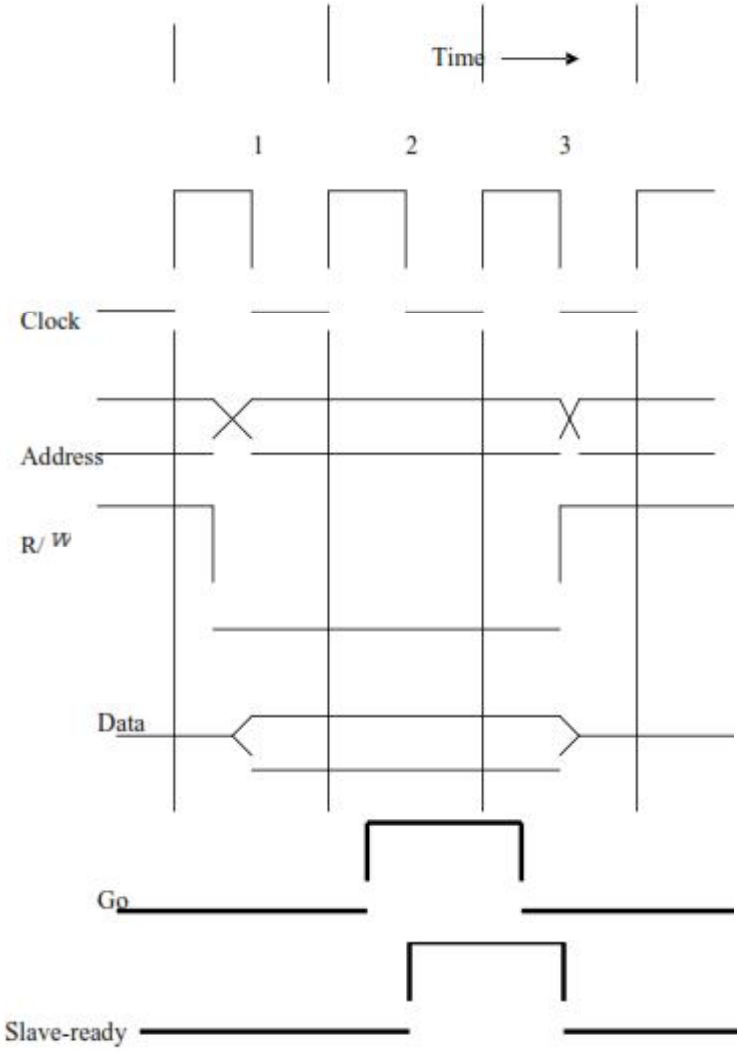


Figure 18 : State diagram for the timing logic

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 20. It includes the familiar DATAIN and DATAOUT registers. The input shift register accepts bit-serial input from the I/O 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 DATAOUT

register are loaded into the output register, from which the bits are shifted out and sent to the I/O device.

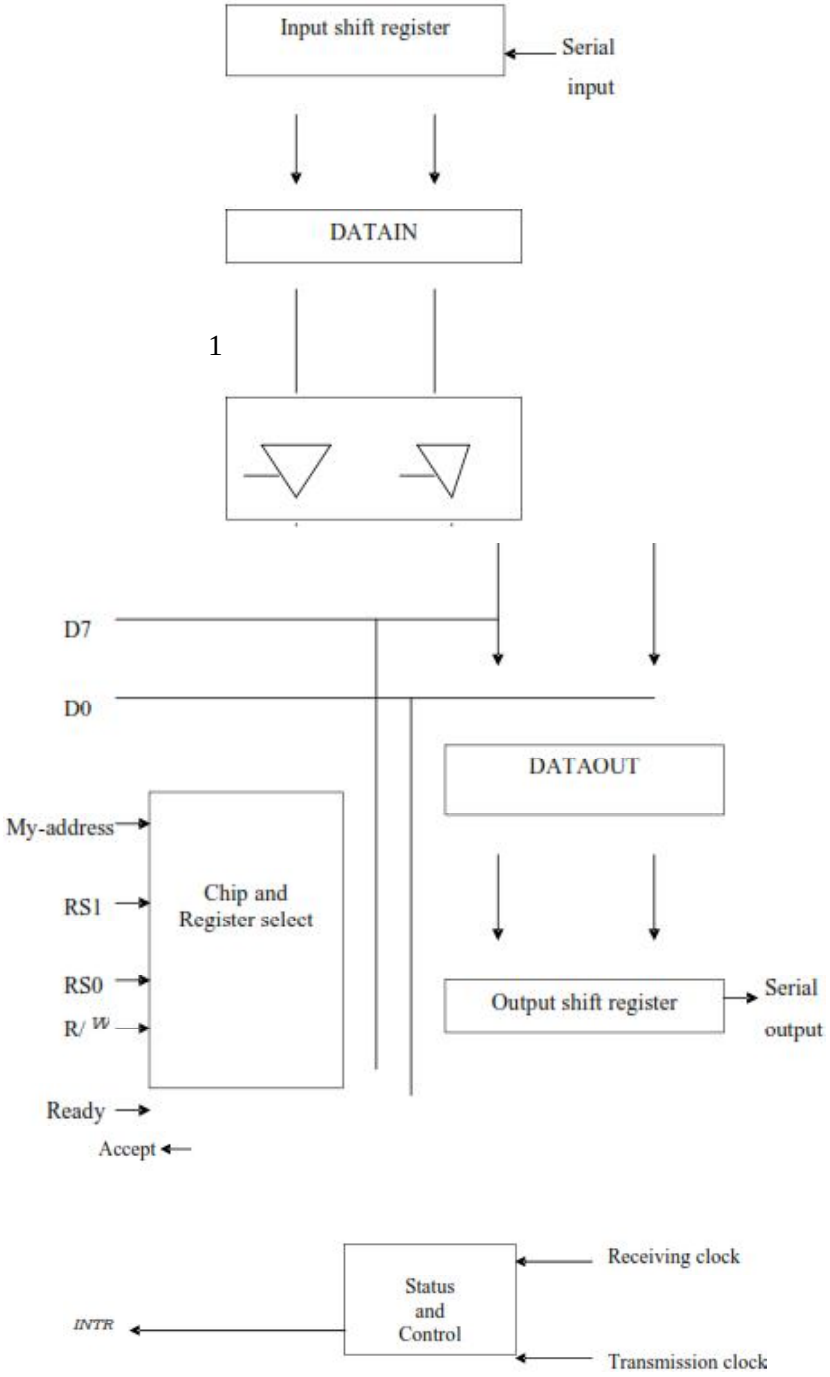


Figure 20 : A serial interface

The double buffering used in the input and output paths are important. A simpler interface could be implemented by turning DATAIN and DATAOUT into shift registers and eliminating the shift registers in figure 4.37. However, this would impose awkward restrictions on the operation of the I/O device; after receiving one character from the serial line, the device cannot start receiving the next character until the processor reads the contents of DATAIN. Thus, a pause would be needed between two characters to allow the processor to read the input data. 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. Thus, provided the processor reads the contents of DATAIN before the serial transfer of the second character is completed, the interface can receive a continuous stream of serial data. An analogous situation occurs in the output path of the interface.

Because serial interfaces play a vital role in connecting I/O devices, several widely used standards have been developed. A standard circuit that includes the features of our example in figure 20. is known as a Universal Asynchronous Receiver Transmitter (UART). It is intended for use with low-speed serial devices. Data transmission is performed using the asynchronous start-stop format. To facilitate connection to communication links, a popular standard known as RS-232-C was developed.

4.2 STANDARD I/O INTERFACES

The processor bus is the bus defied by the signals on the processor chip itself. Devices that require a very high-speed connection to the processor, such as the main memory, may be connected directly to this bus. For electrical reasons, only a few devices can be connected in this manner. The motherboard usually provides another bus that can support more devices. The two buses are interconnected by a circuit, which we will call a bridge, that translates the signals and protocols of one bus into those of the other. Devices connected to the expansion bus appear to the processor as if they were connected directly to the processor's own bus. The only difference is that the bridge circuit introduces a small delay in data transfers between the processor and those devices.

It is not possible to define a uniform standard for the processor bus. The structure of this bus is closely tied to the architecture of the processor. It is also dependent on the electrical characteristics of the processor chip, such as its clock speed. The expansion bus is not subject to these limitations, and therefore it can use a standardized signaling scheme. A number of standards have been developed. Some have evolved by default, when a particular design became commercially successful. For example, IBM developed a bus they called ISA (Industry Standard Architecture) for their personal computer known at the time as PC AT. Some standards have been developed through industrial cooperative efforts, even among competing companies driven by their common self-interest in having compatible products. In some cases, organizations such as the IEEE (Institute of Electrical and Electronics Engineers), ANSI (American National Standards Institute), or international bodies such as ISO (International Standards Organization) have blessed these standards and given them an official status.

A given computer may use more than one bus standards. A typical Pentium computer has both a PCI bus and an ISA bus, thus providing the user with a wide range of devices to choose from.

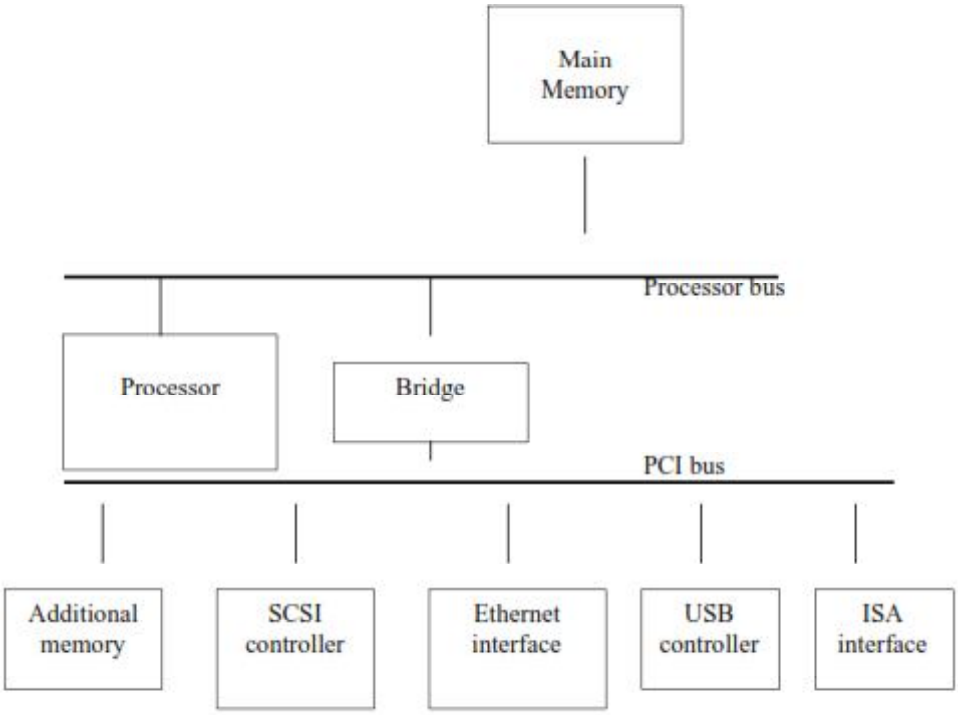


Figure 21 : Peripheral Component Interconnect (PCI) Bus

PERIPHERAL COMPONENT INTERCONNECT (PCI) BUS

The PCI bus is a good example of a system bus that grew out of the need for standardization. It supports the functions found on a processor bus bit in a standardized format that is independent of any particular processor. Devices connected to the PCI bus appear to the processor as if they were connected directly to the processor bus. They are assigned addresses in the memory address space of the processor.

The PCI follows a sequence of bus standards that were used primarily in IBM PCs. Early PCs used the 8-bit XT bus, whose signals closely mimicked those of Intel's 0x86 processors. Later, the 16-bit bus used on the PC At computers became known as the ISA bus. Its extended 32-bit version is known as the EISA bus. Other buses developed in the eighties with similar capabilities are the Microchannel used in IBM PCs and the NuBus used in Macintosh computers.

The PCI was developed as a low-cost bus that is truly processor independent. Its design anticipated a rapidly growing demand for bus bandwidth to support high-speed disks and graphic and video devices, as well as the specialized needs of multiprocessor systems. As a result, the PCI is still popular as an industry standard almost a decade after it was first introduced in 1992.

An important feature that the PCI pioneered is a plug-and-play capability for connecting I/O devices. To connect a new device, the user simply connects the device interface board to the bus. The software takes care of the rest.

Data Transfer

In today's computers, most memory transfers involve a burst of data rather than just one word. The reason is that modern processors include a cache memory. Data are transferred between the cache and the main memory in burst 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 the address. The PCI is designed primarily to support this mode of operation. A read or write operation involving a single word is simply treated as a burst of length one.

The bus supports three independent address spaces: memory, I/O, and configuration. The first two are self-explanatory. The I/O address space is intended foruse with processors, such as Pentium, that have a separate I/O address space. However, as noted , the system designer may choose to use memory-mapped I/O even when a separate

I/O address space is available. In fact, this is the approach recommended by the PCI its plug-and-play capability. A 4-bit command that accompanies the address identifies which of the three spaces is being used in a given data transfer operation.

The signaling convention on the PCI bus is similar to the one used, we assumed that the master maintains the address information on the bus until data transfer is completed. But, this is not necessary. The address is needed only long enough for the slave to be selected. The slave can store the address in its internal buffer. 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 because the number of wires on a bus is an important cost factor. This approach in used in the PCI bus.

At any given time, one device is the bus master. It has the right to initiate data transfers by issuing read and write commands. A master is called an initiator in PCI terminology. This is either a processor or a DMA controller. The addressed device that responds to read and write commands is called a target.

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. The PCI simplifies this process by

incorporating in each I/O device interface a small configuration ROM memory that stores information about that device. The configuration ROMs of all devices is accessible in the configuration address space. The PCI initialization software reads these ROMs whenever the system is powered up or reset. In each case, it 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#.

The PCI bus has gained great popularity in the PC word. It is also used in many other computers, such as SUNs, to benefit from the wide range of I/O devices for which a PCI interface is available. In the case of some processors, such as the Compaq Alpha, the PCI-processor bridge circuit is built on the processor chip itself, further simplifying system design and packaging.

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. A SCSI bus may have eight data lines, in which case it is called a narrow bus and transfers data one byte at a time. Alternatively, a wide SCSI bus has 16 data lines and transfers data 16 bits at a time. There are also several options for the electrical signaling scheme used.

Devices connected to the SCSI bus are not part of the address space of the processor in the same way as devices connected to the processor bus. 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 out the device.

To illustrate the operation of the SCSI bus, let us consider how it may be used with a disk drive. Communication with a disk drive differs substantially from communication with the main memory.

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. Clearly, the controller on the processor side, such as the SCSI controller, must be able to operate as an initiator. 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.

The processor sends a command to the SCSI controller, which causes the following sequence of event to take place:

1. The SCSI controller, acting as an initiator, contends for control of the bus.
2. 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.
4. 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.
5. 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.
6. The target transfers the contents of the data buffer to the initiator and then suspends the connection again. Data are transferred either 8 or 16 bits in parallel, depending on the width of the bus.
7. 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 transfers, the logical connection between the two controllers is terminated.
8. As the initiator controller receives the data, it stores them into the main memory using the DMA approach.
9. The SCSI controller sends as interrupt to the processor to inform it that the requested operation has been completed This scenario show that the messages exchanged over the SCSI bus are at a higher level than those exchanged over the processor bus. In this context, a "higher

level" means that the messages refer to operations that may require several steps to complete, depending on the device. Neither the processor nor the SCSI controller need be aware of the details of operation of the particular device involved in a data transfer. In the preceding example, the processor need not be involved in the disk seek operation.

UNIVERSAL SERIAL BUS (USB

The synergy between computers and communication is at the heart of today's information technology revolution. A modern computer system is likely to involve a wide variety of devices such as keyboards, microphones, cameras, speakers, and display devices. Most computers also have a wired or wireless connection to the Internet. A key requirement is such an environment is the availability of a simple, low-cost mechanism to connect these devices to the computer, and an important recent development in this regard is the introduction of the Universal Serial Bus (USB). This 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.

The USB supports two speeds of operation, called low-speed (1.5 megabits/s) and full-speed (12 megabits/s). The most recent revision of the bus specification (USB 2.0) introduced a third speed of operation, called high-speed (480 megabits/s). The USB is quickly gaining acceptance in the market place, and with the addition of the high-speed capability it may well become the interconnection method of choice for most computer devices.

The USB has been designed to meet several key objectives:

- Provides a simple, low-cost and easy to use interconnection system that overcomes the difficulties due to the limited number of I/O ports available on a computer.
- Accommodate a wide range of data transfer characteristics for I/O devices, including telephone and Internet connections.
- Enhance user convenience through a "plug-and-play" mode of operation Port Limitation:-

The parallel and serial ports described in previous section provide a general- purpose point of connection through which a variety of low-to medium-speed devices can be connected to a computer. For practical reasons, only a few such ports are provided in a typical computer.

Device Characteristics

The kinds of devices that may be connected to a computer cover a wide range of functionality. The speed, volume, and timing constraints associated with data transfers to and from such devices vary significantly. A variety of simple devices that may be attached to a computer generate data of a similar nature - low speed and asynchronous. Computer mice and the controls and manipulators used with video games are good examples.

Plug-and-Play

As computers become part of everyday life, their existence should become increasingly transparent. For example, when operating a home theater system, which includes at least one computer, the user should not find it necessary to turn the computer off or to restart the system to connect or disconnect a device. The plug-and-play feature means that a new device, such as an additional speaker, can be connected at any time while the system is operating. The system should detect the existence of this new device automatically, identify the appropriate device-driver software and any other facilities needed to service that device, and establish the appropriate addresses and logical connections to enable them to communicate. The plug-and-play requirement has many implications at all levels in the system, from the hardware to the operating system and the applications software. One of the primary objectives of the design of the USB has been to provide a plug-and-play capability.

USB Architecture

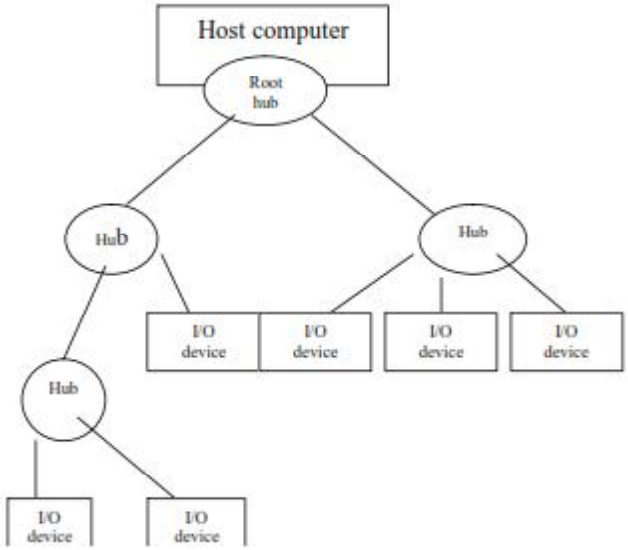


Figure 23 : Universal Serial Bus tree structure

The discussion above points to the need for an interconnection system that combines low cost, flexibility, and high data-transfer bandwidth. Also, I/O devices may be located at some distance from the computer to which they are connected. The requirement for high bandwidth would normally suggest a wide bus that carries 8, 16, or more bits in parallel. However, a large number of wires increases cost and complexity and is inconvenient to the user. Also, it is difficult to design a wide bus that carries data for a long distance because of the data skew problem discussed. The amount of skew increases with distance and limits the data that can be used. A serial transmission format has been chosen for the USB because a serial bus satisfies the low-cost and flexibility requirements.

Clock and data information are encoded together and transmitted as a single signal. Hence, there are no limitations on clock frequency or distance arising from data skew. Therefore, it is possible to provide a high data transfer bandwidth by using a high clock frequency. As pointed out earlier, the USB offers three bit rates, ranging from 1.5 to 480 megabits/s, to suit the needs of different I/O devices.

To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure shown in figure 23. 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), which are called functions in USB terminology. For consistency with the rest of the discussion in the book, we will refer to these devices as I/O devices.

The tree structure enables many devices to be connected while using only simple point-to-point serial links. Each hub has a number of ports where devices may be connected, including other hubs. 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. In this respect, the USB functions in the same way as the bus in figure 4.1. However, unlike the bus in figure 4.1, 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.

Note how the tree structure helps meet the USB's design objectives. The tree makes it possible to connect a large number of devices to a computer through a few ports (the root hub). At the same time, each I/O device is connected through a serial point-to-point connection. This is an important consideration in facilitating the plug-and-play feature, as we will see shortly.

The USB operates strictly on the basis of polling. A device may send a message only in response to a poll message from the host. Hence, upstream messages do not encounter conflicts or interfere with each other, as no two devices can send messages at the same time. This restriction allows hubs to be simple, low-cost devices. The mode of operation described above is observed for all devices operating at either low speed or full speed. However, one exception has been necessitated by the introduction of high-speed operation in USB version 2.0. Consider the situation in figure 24. Hub A is connected to the root hub by a high-speed link. This hub serves one high-speed device, C, and one low-speed device, D. Normally, a messages to device D would be sent at low speed from the root hub. At 1.5 megabits/s, even a short message takes several tens of microsends. For the duration of this message, no other data transfers can take place, thus reducing the effectiveness of the high-speed links and introducing unacceptable delays for high-speed devices. To mitigate this problem, the USB protocol requires that a message transmitted on a high-speed link is always

transmitted at high speed, even when the ultimate receiver is a low-speed device. Hence, a message at low speed to device D. The latter transfer will take a long time, during which high-speed traffic to other nodes is allowed to continue. For example, the root hub may exchange several message with device C while the low-speed message is being sent from hub A to device D. During this period, the bus is said to be split between high-speed and low-speed traffic. The message to device D is preceded and followed by special commands to hub A to start and end the split-traffic mode of operation, respectively.

PART - B

UNIT-5

Memory System: Basic Concepts, Semiconductor RAM Memories, Read Only Memories, Speed, Size, and Cost, Cache Memories - Mapping Functions, Replacement Algorithms, Performance Considerations, Virtual Memories, Secondary Storage

Unit-5

Memory System

The maximum size of the Main Memory (MM) that can be used in any computer is determined by its addressing scheme. For example, a 16-bit computer that generates 16-bit addresses is capable of addressing upto 216 =64K memory locations. If a machine generates 32-bit addresses, it can access upto 232 = 4G memory locations. This number represents the size of address space of the computer.

If the smallest addressable unit of information is a memory word, the machine is called word-addressable. If individual memory bytes are assigned distinct addresses, the computer is called byte-addressable. Most of the commercial machines are byte- addressable. For example in a byte-addressable 32-bit computer, each memory word contains 4 bytes. A possible word-address assignment would be:

Word Address Byte Address

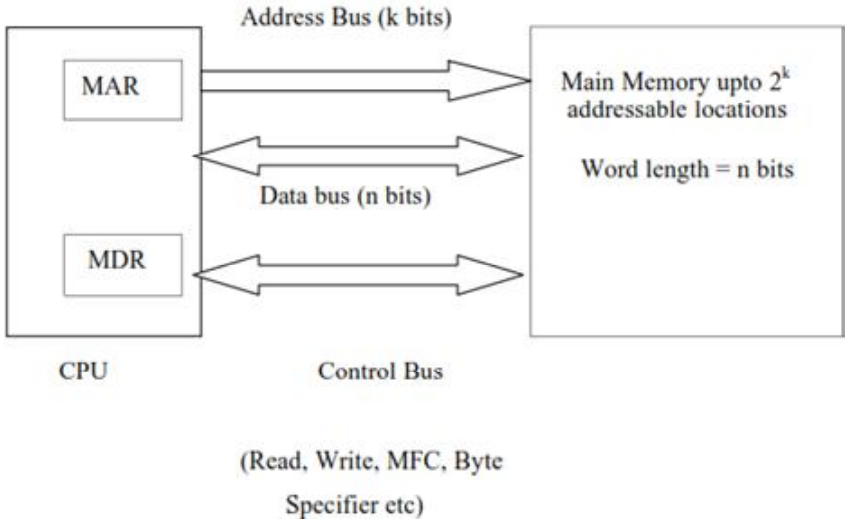
0 0 1 2 3
4 4 5 6 7
8 8 9 10 11
. …..
. …..
. …..

With the above structure a READ or WRITE may involve an entire memory word or it may involve only a byte. In the case of byte read, other bytes can also be read but ignored by the CPU. However, during a write cycle, the control circuitry of the MM must ensure that only the specified byte is altered. In this case, the higher-order 30 bits can specify the word and the lower-order 2 bits can specify the byte within the word.

CPU-MAIN MEMORY CONNECTION - A BLOCK SCHEMATIC

From the system standpoint, the Main Memory (MM) unit can be viewed as a "block box". Data transfer between CPU and MM takes place through the use of two CPU registers, usually called MAR (Memory Address Register) and MDR (Memory Data Register). If MAR is K bits long and MDR is 'n' bits long, then the MM unit may contain upto 2k addressable locations and each location will be 'n' bits wide, while the word length is equal to 'n' bits. During a "memory cycle", n bits of data may be transferred between the MM and CPU. This transfer takes place over

the processor bus, which has k address lines (address bus), n data lines (data bus) and control lines like Read, Write, Memory Function completed (MFC), Bytes specifiers etc (control bus). For a read operation, the CPU loads the address into MAR, set READ to 1 and sets other control signals if required. The data from the MM is loaded into MDR and MFC is set to 1. For a write operation, MAR, MDR are suitably loaded by the CPU, write is set to 1 and other control signals are set suitably. The MM control circuitry loads the data into appropriate locations and sets MFC to 1. This organization is shown in the following block schematic .



SOME BASIC CONCEPTS

Memory Access Times

It is a useful measure of the speed of the memory unit. It is the time that elapses between the initiation of an operation and the completion of that operation (for example, the time between READ and MFC).

Memory Cycle Time

It is an important measure of the memory system. It 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.

5.2 RANDOM ACCESS MEMORY (RAM)

A memory unit is called a Random Access Memory 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. Main memory units are of this type. This distinguishes them from serial or partly serial access storage devices such as magnetic tapes and disks which are used as the secondary storage device.

Cache Memory

The CPU of a computer can usually process instructions and data faster than they can be fetched from compatibly priced main memory unit. Thus the memory cycle time become the bottleneck in the system. One way to reduce the memory access time is to use cache memory. This is a small and fast memory that is inserted between the larger, slower main memory and the CPU. This holds the currently active segments of a program and its data. Because of the locality of address references, the CPU can, most of the time, find the relevant information in the cache memory itself (cache hit) and infrequently needs access to the main memory (cache miss) with suitable size of the cache memory, cache hit rates of over 90% are possible leading to a cost-effective increase in the performance of the system.

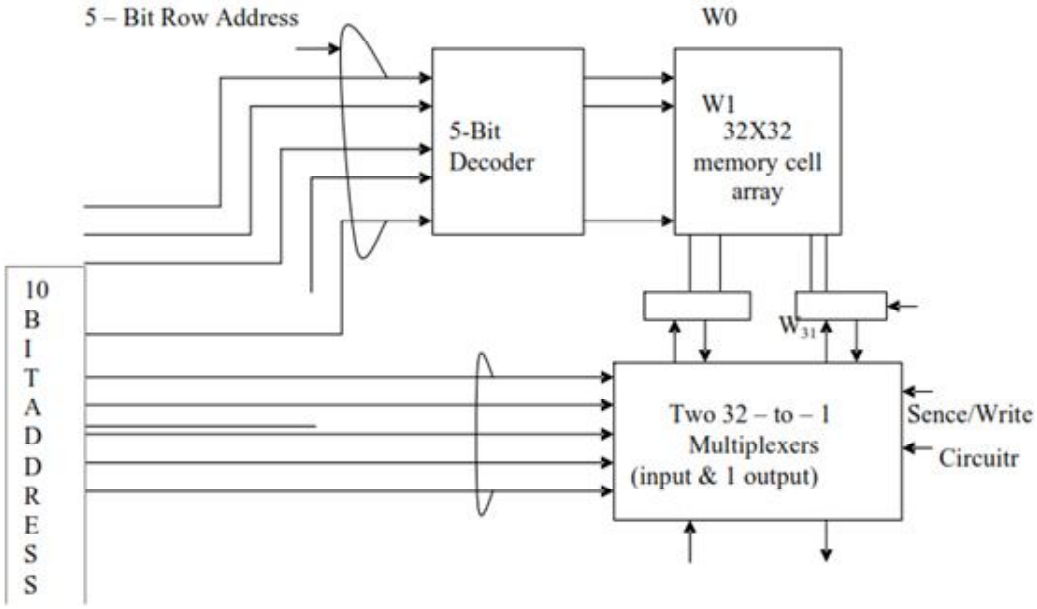
5.3 INTERNAL ORGANIZATION OF SEMICONDUCTOR MEMORY CHIPS

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

The following figure shows such an organization of a memory chip consisting of 16 words of 8 bits each, which is usually 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 bi-directional data line in order to reduce the number of pins required. One control line, the R/W (Read/Write) input is used a specify the required operation and another control line, the CS (Chip Select) input is used to select a given chip in a multichip memory system. 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:



The 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. One of these, selected by the column address, is connected to the external data lines by the input and output multiplexers. This structure can store 1024 bits, can be implemented in a 16-pin chip.

A Typical Memory Cell

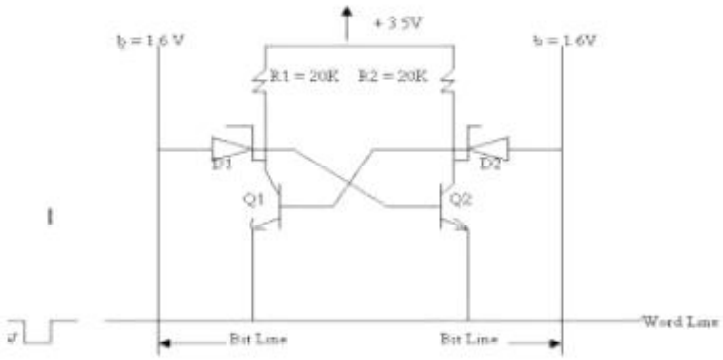
Semiconductor memories may be divided into bipolar and MOS types. They may be compared as follows:

Characteristic	Bipolar	MOS
Power Dissipation	More	Less
Bit Density	Less	More
Impedance	Lower	Higher
Speed	More	Less

Bipolar Memory Cell

Two transistor inverters connected to implement a basic flip-flop. The cell is connected to one word line and two bits lines as shown. Normally, the bit lines are kept at about 1.6V, and the word line is kept at a slightly higher voltage of about 2.5V. Under these conditions, the two diodes D1 and D2 are reverse biased. Thus, because no current flows through the diodes, the cell is isolated from the bit lines.

A typical bipolar storage cell is shown below:



Read Operation

Let us assume the Q1 on and Q2 off represents a 1 to read the contents of a given cell, the voltage on the corresponding word line is reduced from 2.5 V to approximately 0.3 V. This causes one of the diodes D1 or D2 to become forward-biased, depending on whether the transistor Q1 or Q2 is conducting. As a result, current flows from bit line b when the cell is in the 1 state and from bit line b when the cell is in the 0 state. The Sense/Write circuit at the end of each pair of bit lines monitors the current on lines b and b' and sets the output bit line accordingly.

Write Operation

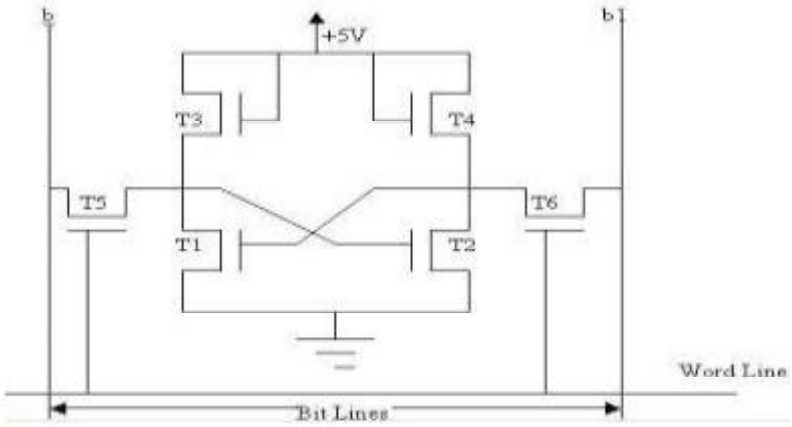
While a given row of bits is selected, that is, while the voltage on the corresponding word line is 0.3V, the cells can be individually forced to either the 1 state by applying a positive voltage of about 3V to line b' or to the 0 state by driving line b. This function is performed by the Sense/Write circuit.

MOS Memory Cell

MOS technology is used extensively in Main Memory Units. As in the case of bipolar memories, many MOS cell configurations are possible. The simplest of these is a flip-flop circuit. Two transistors T1 and T2 are connected to implement a flip-flop. Active pull-up to VCC is provided through T3 and T4. Transistors T5 and T6 act as switches that can be opened or closed under control of the word line. For a read operation, when the cell is selected, T5 or T6 is closed and the corresponding flow of current through b or b' is sensed by the sense/write circuits to set the output bit line accordingly. For a write operation, the bit is selected and a positive voltage is applied on the appropriate bit line, to store a 0 or 1. This configuration is shown below:

Bipolar as well as MOS memory cells using a flip-flop like structure to store information can maintain the information as long as current flow to the cell is maintained. Such memories are called

static memories. In contracts, Dynamic memories require not only the maintaining of a power supply, but also a periodic "refresh" to maintain the information stored in them. Dynamic memories can have very high bit densities and very lower power consumption relative to static memories and are thus generally used to realize the main memory unit.

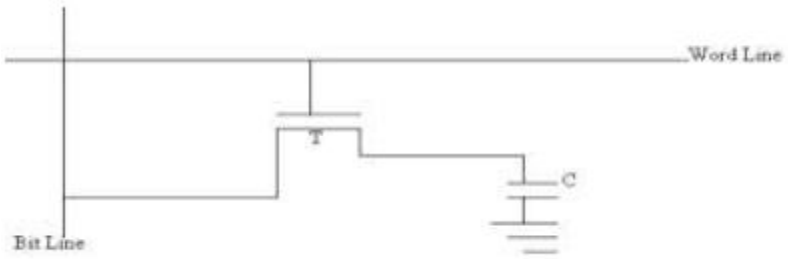


Schematic Memories Vs Dynamic Memories

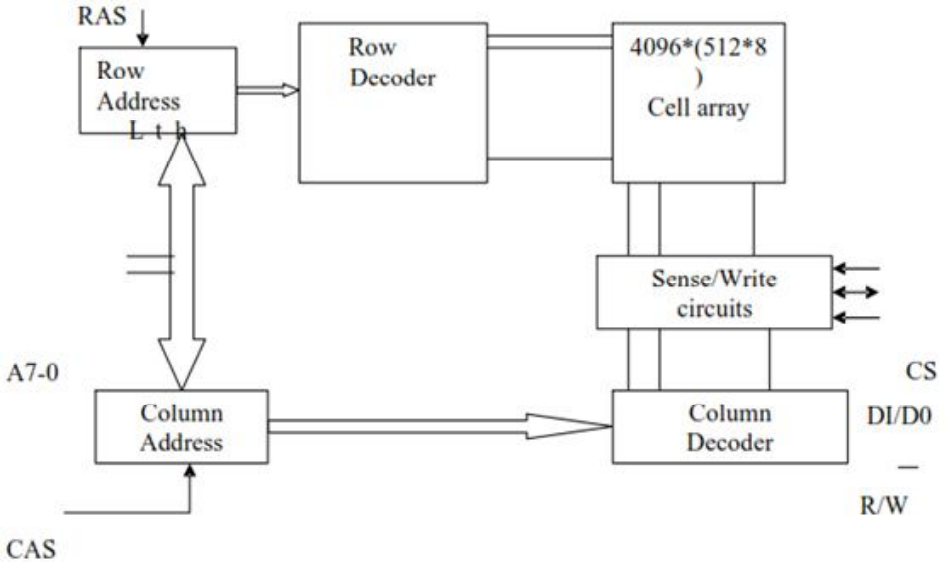
Dynamic Memories

The basic idea of dynamic memory is that information is stored in the form of a charge on the capacitor. An example of a dynamic memory cell is shown below:

When the transistor T is turned on and an appropriate voltage is applied to the bit line, information is stored in the cell, in the form of a known amount of charge stored on the capacitor. After the transistor is turned off, the capacitor begins to discharge. This is caused by the capacitor's own leakage resistance and the very small amount of current that still flows through the transistor. Hence the data is read correctly only if is read before the charge on the capacitor drops below some threshold value. During a Read operation, the bit line is placed in a high-impendance state, the transistor is turned on and a sense circuit connected to the bit line is used to determine whether the charge on the capacitor is above or below the threshold value. During such a Read, the charge on the capacitor is restored to its original value and thus the cell is refreshed with every read operation



A typical organization of a 64k x 1 dynamic memory chip is shown below:



Typical Organization of a Dynamic Memory Chip

The cells are organized in the form of a square array such that the high-and lower-order 8 bits of the 16-bit address constitute the row and column addresses of a cell, respectively. In order to reduce the number of pins needed for external connections, the row and column address are multiplexed on 8 pins. To access a cell, the row address is applied first. It is loaded into the row address latch in response to a single pulse on the Row Address Strobe (RAS) input. This selects a row of cells. Now, the column address is applied to the address pins and is loaded into the column address latch under the control of the Column Address Strobe (CAS) input and this address selects the appropriate sense/write circuit. If the R/W signal indicates a Read operation, the output of the selected circuit is transferred to the data output. Do. For a write operation, the data on the DI line is used to overwrite the cell selected.

It is important to note that the application of a row address causes all the cells on the corresponding row to be read and refreshed during both Read and Write operations. To ensure that the contents of a dynamic memory are maintained, each row of cells must be addressed periodically, typically once every two milliseconds. A Refresh circuit performs this function. Some dynamic memory chips incorportate a refresh facility the chips themselves and hence they appear as static memories to the user! such chips are often referred to as Pseudostatic.

Another feature available on many dynamic memory chips is that once the row address is loaded, successive locations can be accessed by loading only column addresses. Such block transfers can be carried out typically at a rate that is double that for transfers involving random addresses. Such a feature is useful when memory access follow a regular pattern, for example, in a graphics terminal.

Because of their high density and low cost, dynamic memories are widely used in the main memory units of computers. Commercially available chips range in size from 1k to 4M bits or more, and are available in various organizations like 64k x 1, 16k x 4, 1MB x 1 etc.

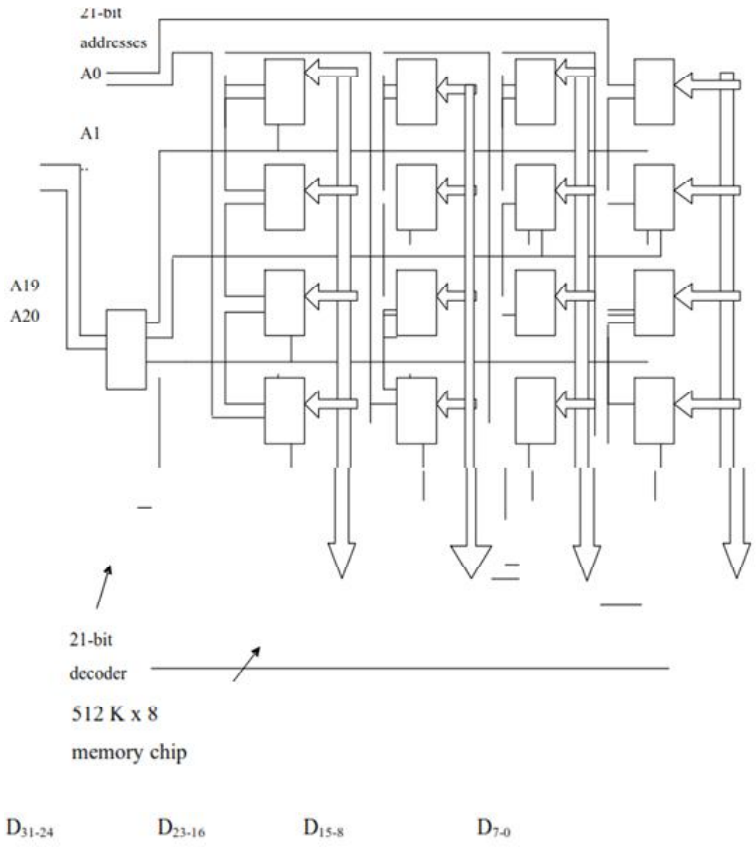
DESIGN CONSIDERATION FOR MEMORY SYSTEMS

The choice of a RAM chip for a given application depends on several factors like speed, power dissipation, size of the chip, availability of block transfer feature etc.

Bipolar memories are generally used when very fast operation is the primary requirement. High power dissipation in bipolar circuits makes it difficult to realize high bit densities.

Dynamic MOS memory is the predominant technology used in the main memories of computer, because their high bit-density makes it possible to implement large memories economically. Static MOS memory chips have higher densities and slightly longer access times compared to bipolar chips. They have lower densities than dynamic memories but are easier to use because they do not require refreshing.

Design using static Memory Chips



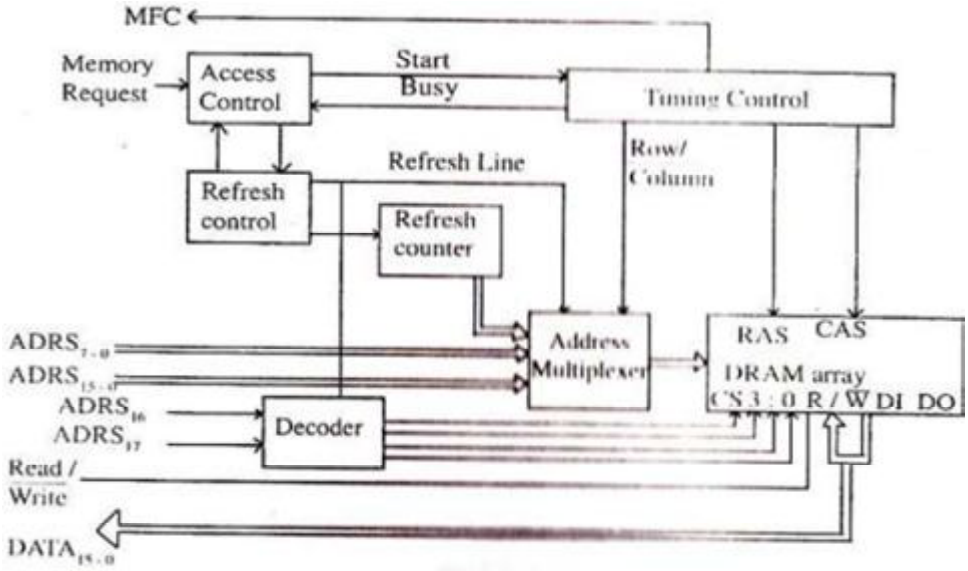
Consider the design of a memory system of 64k x 16 using 16k x 1 static memory chips. We can use a column of 4 chips to implement one bit position. Sixteen such sets provide the required 64k x 16 memories. Each chip has a control input called chip select, which should be set to 1 to enable the chip to participate in a read or write operation. When this signal is 0, the chip is electrically isolated from the rest of the system. The high-order 2 bits of the 16-bit address bus are decoded to obtain the four chip select control signals, and the remaining 14address bits are used to access specific bit locations inside each chip of the selected row.

Design using Dynamic Memory Chips

The design of a memory system using dynamic memory chips is similar to the design using static memory chips, but with the following important differences in the control circry.

- The row and column parts of the address of each chip have to be multiplexed;
- A refresh circuit is needed; and
- The timing of various steps of a memory cycle must be carefully controlled.

A memory system of 256k x 16 designed using 64k x 1 DRAM chips, is shown below;



The memory unit is assumed to be connected to an asynchronous memory bus that has 18 address lines (ADRS 17-0 request and MFC), and a Read/), 16 data lines (DATA Write), two handshake signals (Memory line to specify the operation (read to Write). 15-0 The memory chips are organized into 4 rows, with each row having 16 chips. Thus each column of the 16 columns implements one bit position. The higher order 12 bits of the address are decoded to get four chip select control signals which are used to select one of the four rows. The remaining 16 bits, after

multiplexing, are used to access specific bit 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. The DI and DO lines, together, provide D 15 - D 0 i.e. the data bus DATA . 15-0.

The operation of the control circuit can be described by considering a normal memory read cycle. The cycle begins when the CPU activates the address, the and the Memory Request lines. When the memory request line becomes Read/active, the Access control block sets the start signal to 1. The timing control block responds by setting Busy lines to 1, in order to prevent the access control block from accepting new requests until the current cycle ends. The timing control block then loads the row and column addresses into the memory chips by activating the RAS and CAS lines. During this time, it uses the Row/ADRS 15-8 , followed by the column address, ADRS

Refresh Operation

(7-0)

The Refresh control block periodically generates Refresh, requests, causing the access control block to start a memory cycle in the normal way. This block allows the refresh operation by activating the Refresh Grant line. The access control block arbitrates between Memory Access requests and Refresh requests, with priority to Refresh requests in the case of a tie to ensure the integrity of the stored data. As soon as the Refresh control block receives the Refresh Grant signal, it activates the Refresh line. This causes the address multiplexer to select the Refresh counter as the source and its contents are thus loaded into the row address latches of all memory chips when the RAS signal is activated. During this time the R/W low, causing an inadvertent write operation. One way to prevent this is to use the Refresh line to control the decoder block to deactivate all the chip select lines. The rest of the refresh cycle is the same as in a normal cycle. At the end, the Refresh control block increments the refresh counter in preparation for the next Refresh cycle.

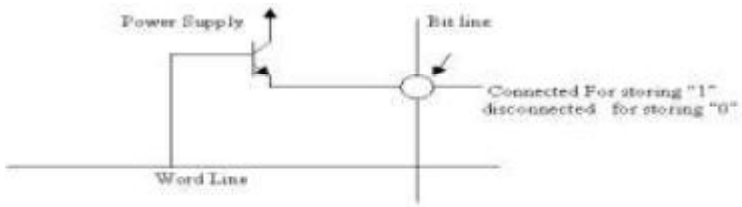
Even though the row address has 8 bits, the Refresh counter need only be 7 bits wide because of the cell organization inside the memory chips. In a 64k x 1 memory chip, the 256x256 cell array actually consists of two 128x256 arrays. The low order 7 bits of the row address select a row from both arrays and thus the row from both arrays is refreshed!

Ideally, the refresh operation should be transparent to the CPU. However, the response of the memory to a request from the CPU or from a DMA device may be delayed if a Refresh operation is in progress. Further, in the case of a tie, Refresh operation is given priority. Thus the normal access may be delayed. This delay will be more if all memory rows are refreshed before the memory is returned to normal use. A more common scheme, however, interleaves Refresh operations on successive rows with accesses from the memory bus. In either case, Refresh operations generally use less than 5% of the available memory cycles, so the time penalty caused by refreshing is very small.

The variability in the access times resulting from refresh can be easily accommodated in an asynchronous bus scheme. With synchronous buses, it may be necessary for the Refresh circuit to request bus cycles as a DMA device would!

Semi Conductor Rom Memories

Semiconductor read-only memory (ROM) units are well suited as the control store components in micro programmed processors and also as the parts of the main memory that contain fixed programs or data. The following figure shows a possible configuration for a bipolar ROM cell.



The word line is normally held at a low voltage. If a word is to be selected, the voltage of the corresponding word line is momentarily raised, which causes all transistors whose emitters are connected to their corresponding bit lines to be turned on. The current that flows from the voltage supply to the bit line can be detected by a sense circuit. The bit positions in which current is detected are read as 1s, and the remaining bits are read as O.

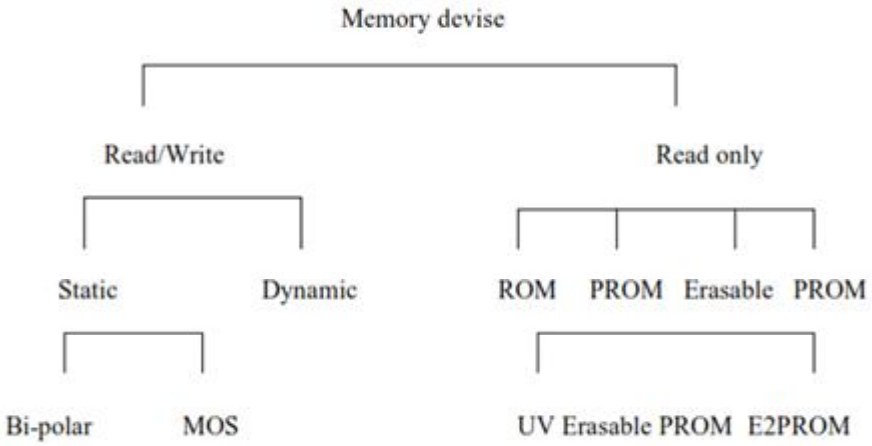
Therefore, the contents of a given word are determined by the pattern of emitter to bit-line connections similar configurations are possible in MOS technology.

Data are written into a ROM at the time of manufacture programmable ROM (PROM) devices allow the data to be loaded by the user. Programmability is achieved by connecting a fuse between the emitter and the bit line. Thus, prior to programming, the memory contains all 1s. The user can inserts O at the required locations by burning out the fuses at these locations using high-current pulses. This process is irreversible.

ROMs are attractive when high production volumes are involved. For smaller numbers, PROMs provide a faster and considerably less expensive approach. Chips which allow the stored data to be erased and new data to be loaded. Such a chip is an erasable, programmable ROM, usually called an EPROM. It provides considerable flexibility during the development phase. An EPROM cell bears considerable resemblance to the dynamic memory cell. As in the case of dynamic memory, information is stored in the form of a charge on a capacitor. The main difference is that the capacitor in an EPROM cell is very well insulated. Its rate of discharge is so low that it retains the stored information for very long periods. To write information, allowing charge to be stored on the capacitor.

The contents of EPROM cells can be erased by increasing the discharge rate of the storage capacitor by several orders of magnitude. This can be accomplished by allowing ultraviolet light

into the chip through a window provided for that purpose, or by the application of a high voltage similar to that used in a write operation. If ultraviolet light is used, all cells in the chip are erased at the same time. When electrical erasure is used, however, the process can be made selective. An electrically erasable EPROM, often referred to as EEPROM. However, the circuit must now include high voltage generation.



5.5 CACHE MEMORY

Analysis of a large number of typical programs has shown that most of their execution time is spent on a few main row lines in which a number of instructions are executed repeatedly. These instructions may constitute a simple loop, nested loops or few procedure that repeatedly call each other. The main observation is that many instructions in a few localized are as of the program are repeatedly executed and that the remainder of the program is accessed relatively infrequently. This phenomenan is referred to as locality of reference.

If the active segments of a program can be placed in a fast memory, then the total execution time can be significantly reduced, such a memory is referred as a cache memory which is in served between the CPU and the main memory as shown in fig.1

Fig.1 cache memory between main memory & cpu.

Two Level memory Hierarchy: We will adopt the terms Primary level for the smaller, faster memory and the secondary level for larger, slower memory, we will also allow cache to be a primary level with slower semiconductor memory as the corresponding secondary level. At a different point in the hierarchy, the same S.C memory could be the primary level with disk as the secondary level.

Primary and Secondary addresses

A two level hierarchy and its addressing are illustrated in fig.2. A system address is applied to the memory management unit (MMU) that handles the mapping function for the particular pair in the hierarchy. If the MMU finds the address in the Primary level, it provides Primary address, which selects the item from the Primary memory. This translation must be fast, because every time memory is accessed, the system address must be translated. The translation may fail to produce a Primary address because the requested items is not found, so that information can be retrieved from the secondary level and transferred to the Primary level.

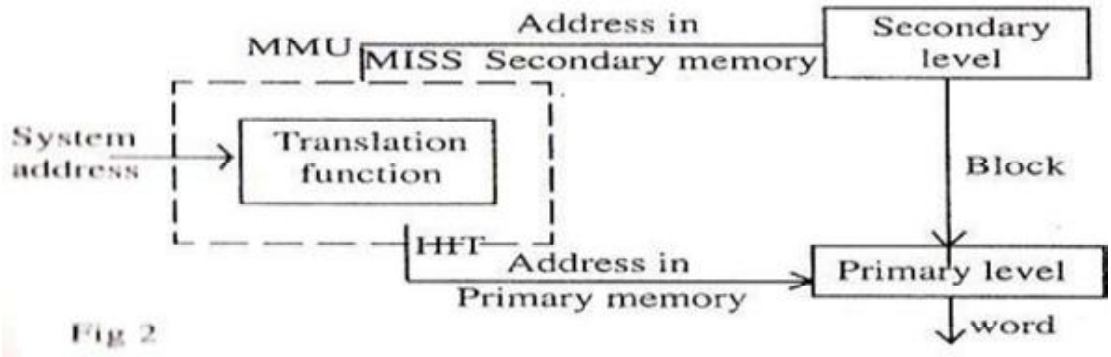


Fig 2

Hits and Misses:- Successful translation of reference into Primary address is called a hit, and failure is a miss. The hit ratio is (1-miss ratio). If t is the Primary memory access time and tsp is the secondary access time, the average access time for the two level hierarchy is

$$t_a = h\,t_p + (1\text{-}h)t_s$$

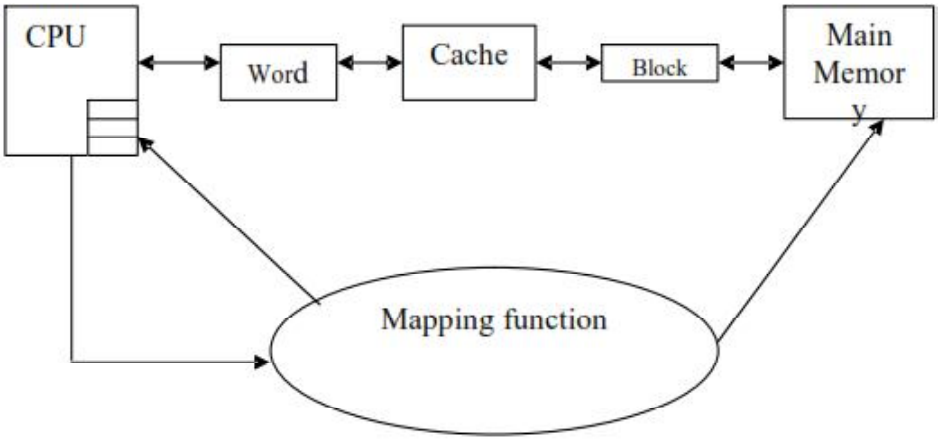
The Cache

The Mapping Function Fig. Shows a schematic view of the cache mapping function. The mapping function is responsible for all cache operations. It is implemented in hardware, because of the required high speed operation. The mapping function determines the following.

- **Placement strategies -** Where to place an incoming block in the cache
- **Replacement strategies -** Which block to replace when a cache miss occurs
- How to handle Read and Writes up as cache hit and mises

Three different types of mapping functions are in common use

1. Associative mapping cache
2. Direct mapping cache
3. Block-set-associative mapping cache



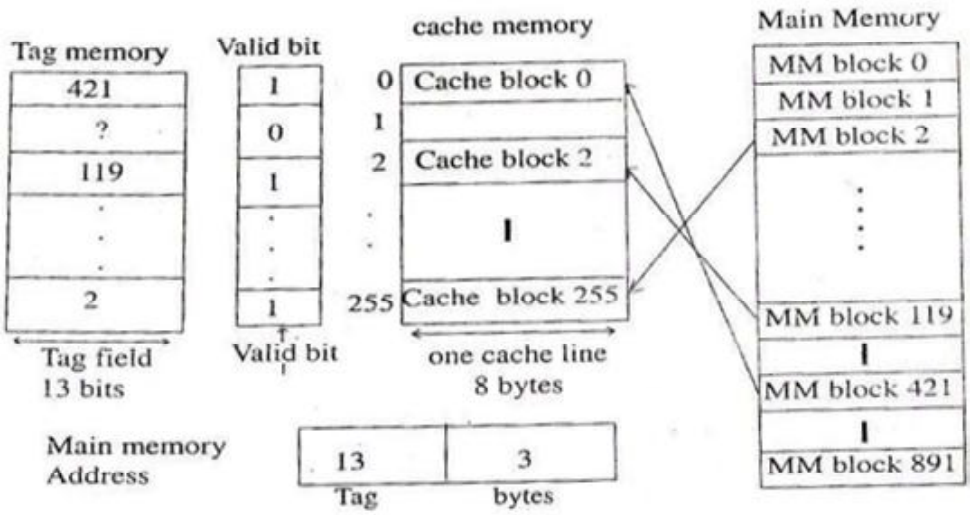
1. **Associative mapped caches:-** In this any block from main memory can be placed any where in the cache. After being placed in the cache, a given block is identified uniquely by its main memory block number, referred to as the tag, which is stored inside a separate tag memory in the cache. Regardless of the kind of cache, a given block in the cache may or may not contain valid information. For example, when the system has just been powered up add before the cache has had any blocks loaded into it, all the information there is invalid.

   The cache maintains a valid bit for each block to keep track of whether the information in the corresponding block is valid.

   Fig4. shows the various memory structures in an associative cache, The cache itself contain 256, 8byte blocks, a 256 x 13 bit tag memory for holding the tags of the blocks currently stored in the cache, and a 256 x 1 bit memory for storing the valid bits, Main memory contains 8192, 8 byte blocks. The figure indicates that main memory address references are partition into two fields, a 3 bit word field describing the location of the desired word in the cache line, and a 13 bit tag field describing the main memory block number desired. The 3 bit word field becomes essentially a "cache address" specifying where to find the word if indeed it is in the cache.

   The remaining 13 bits must be compared against every 13 bit tag in the tag memory to see if the desired word is present.

   In the fig, below, main memory block 2 has been stored in the 256 cache block and so the 256 tag entry is 2 mm block 119 has been stored in the second cache block corresponding entry in tag memory is 119 mm block 421 has been stored in cache block 0 and tag memory location 0 has been set to 421. Three valid bits have also been set, indicating valid information in these locations.



The associative cache makes the most flexible and complete use of its capacity, storing blocks wherever it needs to, but there is a penalty to be paid for this flexibility the tag memory must be searched in for each memory reference.

Fig.5 shows the mechanism of operation of the associative cache. The process begins with the main memory address being placed in the argument register of the (associative) tag memory (1) if there is a match (hit), (2) and if the ratio bit for the block is set (3), then the block is gated out of the cache (4), and the 3 bit offset field is used to select the byte corresponding to the block offset field of the main memory address (5) That byte is forwarded to the CPU, (6).

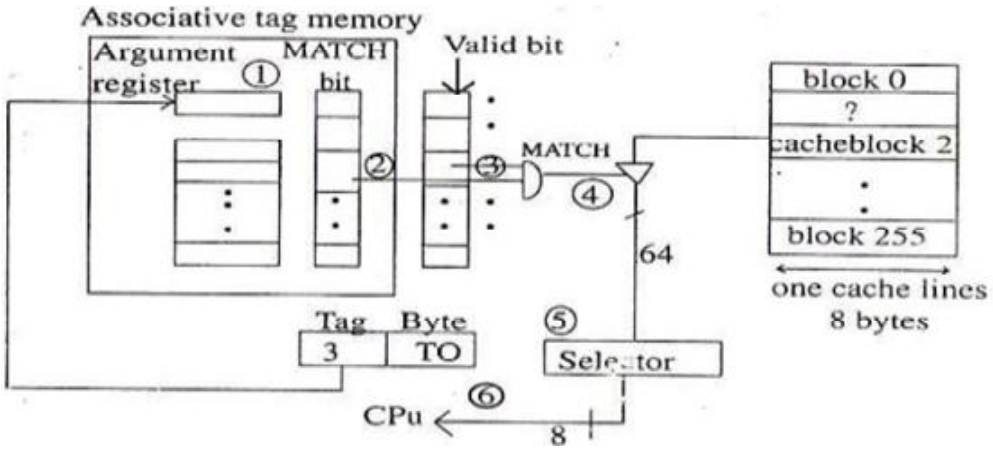


Figure 5 : Associative cache Mechanism

2. **Direct mapped caches:** In this a given main memory block can be placed in one and only one place in the cache. Fig6. Shows an example of a direct - mapped cache. For simplicity,

the example again uses a 256 block x 8 byte cache and a 16 bit main memory address. The main memory in the fig. has 256 rows x 32 columns, still fielding 256 x 32 = 8192 = 213 total blocks as before. Notice that the main memory address is partitioned which of the 256 cache locations the block will be in, if it is indeed in the cache. The tag field specifies which of the 32 blocks from main memory is actually present in the cache.

Now the cache address is composed of the group field, which specifies the address of the block location in the cache and the word field, which specifies the address of the word in the block. There is also valid bit specifying whether the information in the selected block in valid.

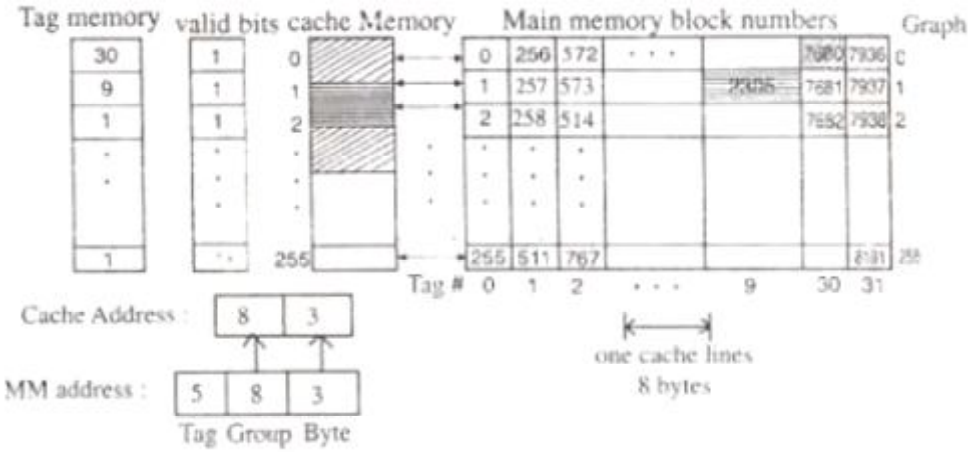


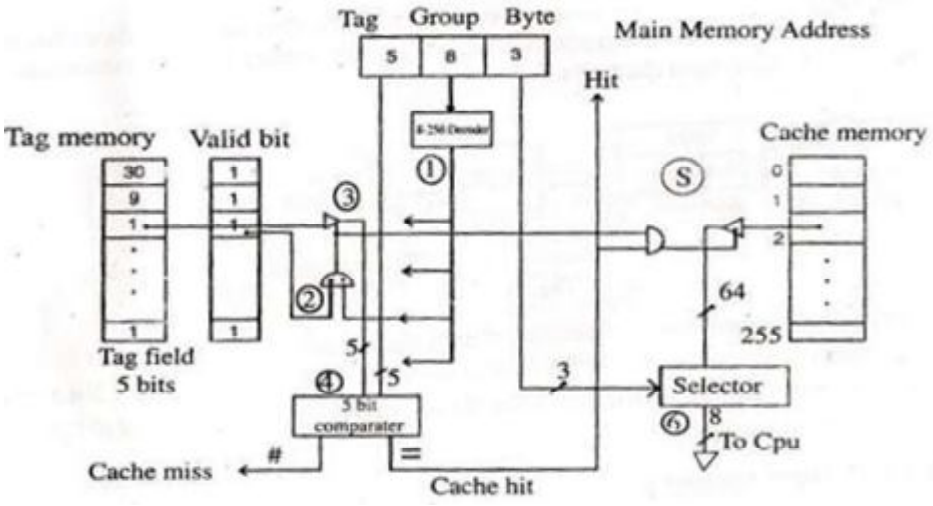
Figure 6 : Direct support cache

The fig.6 shows block 7680, from group 0 of MM placed in block location 0 of the cache and the corresponding tag set to 30. Similarly MM block 259 is in MM group 2, column 1, it is placed in block location 2 of the cache and the corresponding tag memory entry is 1.

The tasks required of the direct - mapped cache in servicing a memory request are shown in fig7.

The fig. shows the group field of the memory address being decoded 1) and used to select the tag of the one cache block location in which the block must be stored if it is the cache. If the valid bit for that block location is gated (2), then that tag is gated out, (3) and compared with the tag of the incoming memory address (4), A cache hit gates the cache block out (5) and the word field selects the specified word from the block (6), only one tag needs to be compared, resulting in considerably less hardware than in the associative memory case.

The direct mapped cache has the advantage of simplicity, but the obvious disadvantage that only a single block from a given group can be present in the cache at any given time.



3. **Block-set-Associative cache:** Block-set-Associative caches share properties of both of the previous mapping functions. It is similar to the direct-mapped cache, but now more than one block from a given group in main memory can occupy the same group in the cache. Assume the same main memory and block structure as before, but with the cache being twice as large, so that a set of two main memory blocks from the same group can occupy a given cache group.

   Fig 8 shows a 2 way set associative cache that is similar to the direct mapped cache in the previous example, but with twice as many blocks in the cache, arranged so that a set of any two blocks from each main memory group can be stored in the cache. MM is still partitioned into an 8 bit set field and a 5 bit tag field, but now there are two possible places in which a given block can reside and both must be searched associatively.

   The cache group address is the same as that of the direct matched cache, an 8 bit block location and a 3 bit word location. Fig 8 shows that the cache entries corresponding to the second group contains blocks 513 and 2304. the group field now called the set field, is again decoded and directs the search to the correct group and new only the tags in the selected group must be searched. So instead of 256 compares, the cache only needs to do 2.

For simplicity, the valid bits are not shown, but they must be present. The cache hard ware would be similar so that shown in fig 7. but there would be two simultaneous comparisons of the two blocks in the set.

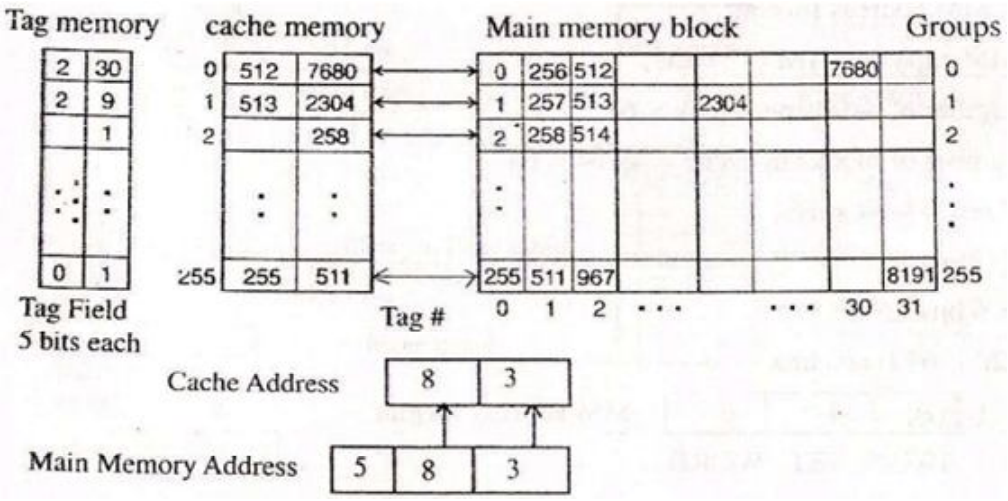


Figure 8 : Two-way set Associative cache

SOLVED PROBLEMS

1. A block set associative cache consists of a total of 64 blocks divided into 4 block sets. The MM contains 4096 blocks each containing 128 words. a) How many bits are there in MM address? b) How many bits are there in each of the TAG, SET & word fields

Solution:

Number of sets = 64/4 = 16

Set bits = 4(24 = 16)

Number of words = 128

Word bits = 7 bits (2

7 = 128)

MM capacity : 4096 x 128 (2

12 x 2

7 = 2)

a) Number of bits in memory address = 19 bits

b) 8 4 7

TAG SET WORD

TAG bits = 19 - (7+4) = 8 bits.

19

2. A computer system has a MM capacity of a total of 1M 16 bits words. It also has a 4K words cache organized in the block set associative manner, with 4 blocks per set & 64 words per block. Calculate the number of bits in each of the TAG, SET & WORD fields of MM address format.

Solution:

Capacity: 1M (2

20

= 1M)

Number of words per block = 64

Number of blocks in cache = 4k/64 = 64

Number of sets = 64/4 = 16

Set bits = 4 (2

4

= 16)

Word bits = 6 bits (2

6

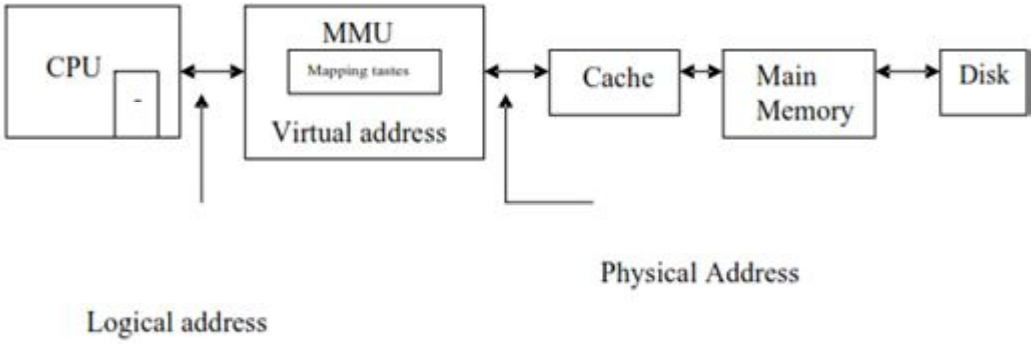
= 64)

Tag bits = 20-(6+4) = 10 bits

MM address format

VIRTUAL MEMORY

Virtual memory is the technique of using secondary storage such as disks to enter the apparent size of accessible memory beyond its actual physical size. Virtual memory is implemented by employing a memory-management unit (MMU) to translate every logical address reference into a physical address reference as shown in fig 1. The MMU is imposed between the CPU and the physical memory where it performs these translations under the control of the operating system. Each memory reference is sued by the CPU is translated from the logical address space to the physical address space. Mapping tables guide the translation, again under the control of the operating system. Virtual memory usually demand paging, which means that a Page is moved from disk into main memory only when the processor accesses a word on that page.



Virtual memory pages always have a place on the disk once they are created, but are copied to main memory only on a miss or page fault.

Advantages of Virtual memory

1. **Simplified addressing:** Each program unit can be compiled into its own memory space, beginning at address O and extending far beyond the limits of physical memory. Programs and data structures do not require address relocation at load time, nor must they be broken into fragments merely to accommodate memory limitations.

2. **Cost effective use of memory:** Less expensive disk storage can replace more expensive RAM memory, since the entire program does not need to occupy physical memory at one time.

3. **Access control:** Since each memory reference must be translated, it can be simultaneously checked for read, write and execute privileges. This allows hardward level control of access to system resources and also prevents and also prevents buggy programs or intruders from causing damage to the resources of other users or the system.

a) Memory management by segmentation:- Segmentation allows memory to be divided into segments of varying sizes depending upon requirements. Fig 2. Shows a main memory containing five segments identified by segment numbers. Each segment beings at a virtual address 0, regardless of where it is located in physical memory.

Each virtual address arriving from the CPU is added to the contents of the segment base register in the MMU to form the physical address. The virtual address may also optionally be compared to a segment limit register to trap reference beyond a specified limit. b) Memory management by paging:- Fig 3 shows a simplified mechanism for virtual address translation in a paged MMU. The process begins in a manner similar to the segmentation process. The virtual address composed of a high order page number and a low order word number is applied to MMU. The virtual page number is limit checked to be certain that the page is within the page table, and if it is, it is added to the page table base to yield the page table entry. The page table entry contains several control fields in addition to the page field. The control fields may include access control bits, a presence bit, a dirty bit and one or more use bits, typically the access control field will include bits

specifying read, write and perhaps execute permission. The presence bit indicates whether the page is currently in main memory. The use bit is set upon a read or write to the specified page, as an indication to the replaced algorithm in case a page must be replaced.

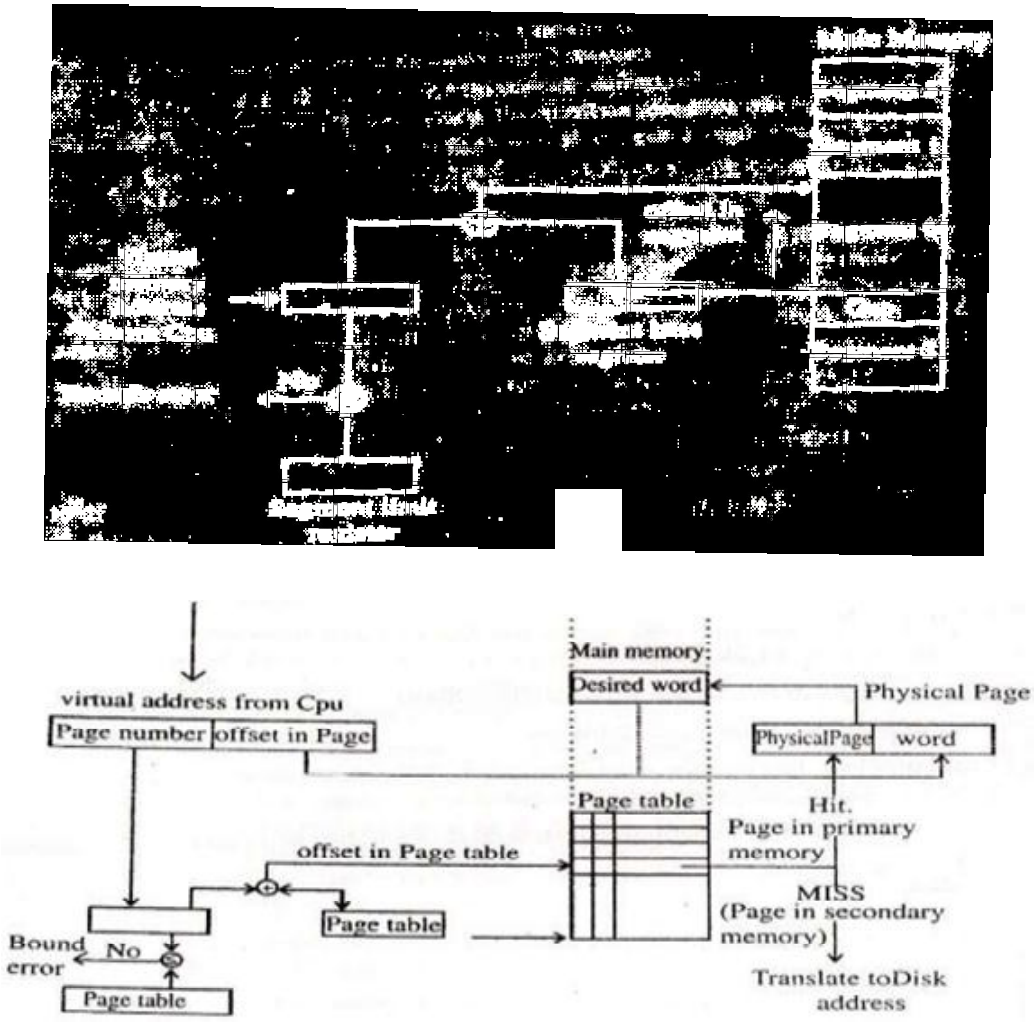


Figure 12 : Virtuel address translation in a paged MMU

If the presence bit indicates a hit, then the page field of the page table entry will contain the physical page number. If the presence bit is a miss, which is page fault, then the page field of the page table entry which contains an address is secondary memory where the page is stored. This miss condition also generates an interrupt. The interrupt service routine will initiate the page fetch from secondary memory and with also the CPU operation is a write hit, then the dirty bit is set. If the CPU operation is a write miss, then the MMU with begin a write allocate process.

PROBLEMS

1. An address space is specified by 24 bits & the corresponding memory space is 16 bits.

Solution:

a) How many words are there in address space?

b) How many words are there in memory space?

c) If a page has 2k words, how many pages & blocks are in the system?

a) Address space = 24 bits

224 = 24.220 = 16M words

b) Memory space: 16 bits

216 = 64k words

c) page consists of 2k words

Number of pages in add space = 16M/2K = 8000

Number of blocks = 64k/2k = 32 blocks

2. A virtual memory has a page size of 1k words. There are 8 pages & 4 blocks. The associative memory page table has the following entries. suspended the requesting process until the page has been bought into main memory. If

Page	block	Make a list of virtual addresses (in decimal) that will cause a page
0	3	by cpu fault of used
1	1	
4	2	
6	0	

The page numbers 2, 3, 5 & 7 are not available in associative memory page table & hence cause a page fault.

5.7 PERIPHERAL DEVICES

1. Magnetic Disk Drives: Hard disk Drive organization

The modern hard disk drive is a system in itself. It contains not only the disks that are used as the storage medium and the read write heads that access the raw data encoded on them, but

also the signal conditioning circuitry and the interface electronics that separate the system user from the details & getting bits on and off the magnetic surface. The drive has 4 platters with read/ write heads on the top and bottom of each platter. The drive rotates at a constant 3600rpm.

Platters and Read/Write Heads: - The heart of the disk drive is the stack of rotating platters that contain the encoded data, and the read and write heads that access that data. The drive contains five or more platters. There are read/write heads on the top and bottom of each platter, so information can be recorded on both surfaces. All heads move together across the platters. The platters rotate at constant speed usually 3600 rpm.

Drive Electronics

The disk drive electronics are located on a printed circuit board attached to the disk drive. After a read request, the electronics must seek out and find the block requested, stream is off of the surface, error check and correct it, assembly into bytes, store it in an on-board buffer and signal the processor that the task is complete.

To assist in the task, the drive electronics include a disk controller, a special purpose processor.

Data organization on the Disk

The drive needs to know where the data to be accessed is located on the disk. In order to provide that location information, data is organized on the disk platters by tracks and sectors. Fig 13 shows simplified view of the organization of tracks and sectors on a disk. The fig. Shows a disk with 1024 tracks, each of which has 64 sectors. The head can determine which track it is on by counting tracks from a known location and sector identities are encoded in a header written on the disk at the front of each sector.

The number of bytes per sector is fixed for a given disk drive, varying in size from 512 bytes to 2KB. All tracks with the same number, but as different surfaces, form a cylinder.

The information is recorded on the disk surface 1 bit at a time by magnetizing a small area on the track with the write head. That bit is detected by sending the direction of that magnetization as the magnetized area passes under the read head as shown in fig 14.

Fig 5. shows how a typical sector might be organized. The header usually contains both synchronization and location information. The synchronization information allows the head positioning circuitry to keep the heads centered on the track and the location information allows the disk controller to determine the sectors & identifies as the header passes, so that the data can be captured if it is read or stored, if it is a write. The 12 bytes of ECC (Error Correcting Code) information are used to detect and correct errors in the 512 byte data field. Generally the disk drive manufacturer initializes or formats the drive by writing its original track and sector information on the surfaces and checks to determine whether data can be written and read from each sector. If

any sectors are found to be bad, that is incapable of being used even with ECC, then they are marked as being defective so their use can be avoided by operating system.

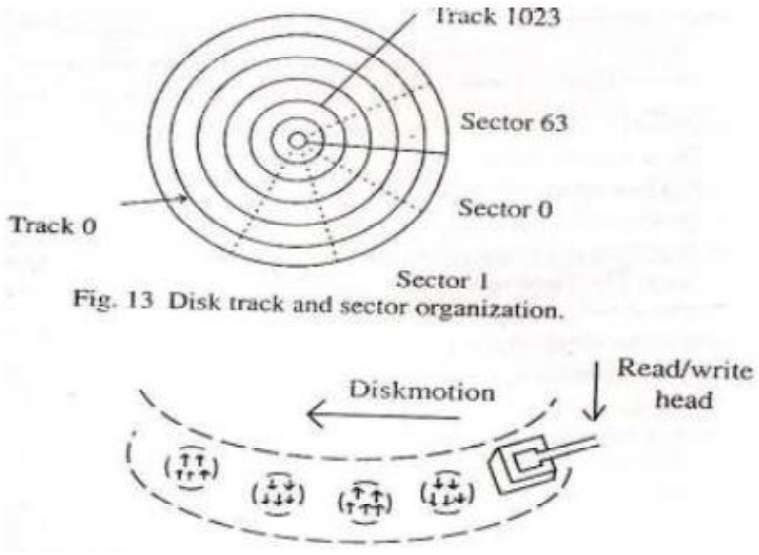


Fig. 13 Disk track and sector organization.

Figure 14 : Individual bits encoded on a disk model

The operating system interface:- The operating system specifies the track, sector and surface of the desired block. The disk controller translates that requests to a series of low level disk operations. Fig 15 shows logical block address (LBA) used to communicate requests to the drive controller.

Head #, Cylinder #, and Sector # refer to the starting address of the information and sector count specifies the number of sectors requested.

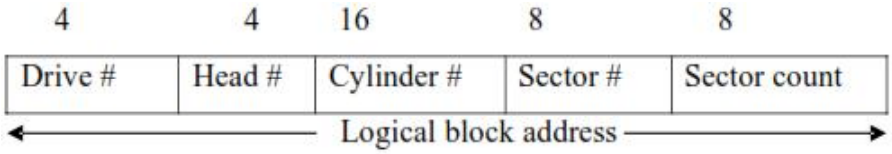


Figure 15 : Block address for disk address.

The Disk Access Process:- Let us follow the process of reading a sector from the disk

1. The OS communicates the LBA to the disk drive and issues the read command.
2. The drive seeks the correct track by moving the heads to the correct position and enabling the one on the specified surface. The read feed reads sector numbers as they travel by until the requested one is found.
3. Sector data and ECC stream into a buffer on the drive interface. ECC is done as the fly.
4. The drive communicates 'data ready' to the OS.

5. The OS reads data, either byte by byte or by issuing a DMA command.Dynamic properties are those that deal with the access time for the reading and writing of data. The calculation of data access time is not simple. It depends not only as the rotational speed of the disk, but also the location of the read/write head when it begins the access. There are several measures of data access times.

1. **Seek time:** Is the average time required to move the read/write head to the desired track. Actual seek time which depend on where the head is when the request is received and how far it has to travel, but since there is no way to know what these values will be when an access request is made, the average figure is used. Average seek time must be determined by measurement. It will depend on the physical size of the drive components and how fast the heads can be accelerated and decelerated. Seek times are generally in the range of 8-20 m sec and have not changed much in recent years.

2. **Track to track access time:** Is the time required to move the head from one track to adjoining one. This time is in the range of 1-2 m sec. 3. Rotational latency: - Is the average time required for the needed sector to pass under head once and head has been positioned once at the correct track. Since on the average the desired sector will be half way around the track from where the head is when the head first arrives at the track, rotational latency is taken to be ½ the rotation time. Current rotation speeds are from 3600 to 7200 rpm, which yield rotational latencies in the 4-8 ms range.

4. **Average Access time:** Is equal to seek time plus rotational latency.

5. **Burst rate:** Is the maximum rate at which the drive produces or accepts data once the head reaches the desired sector, It is equal to the rate at which data bits stream by the head, provided that the rest of the system can produce or accept data at that rate Burst rate (byte/ sec) = rows/sec * sector/row * bytes/sector

6. **Sustained data rate:** Is the rate at which data can be accessed over a sustained period of time.

Problems 1: A hard drive has 8 surface, with 521 tracks per surface and a constant 64 sectors per track. Sector size is 1KB. The average seek time is 8 m sec, the track to track access time is 1.5 m sec and the drive runs at 3600 rpm. Successive tracks in a cylinder can be read without head movement.

a) What is the drive capacity?

b) What is the average access time for the drive?

c) Estimate the time required to transfer a 5MB file?

d) What is the burst transfer rate?

Solution:

a) Capacity = Surfaces x tracks x sectors x sector size

= 8 x 521 x 64 x 1k = 266.752MB

b) Rotational latency : Rotation time/2 = 60/3600x2 = 8.3 m sec

Average Access time = Seek time + Rotational latency

= 8 m sec + 8.3 m sec = 16.3 m sec

c) Assume the file is stored in successive sectors and tracks starting at sector #0, track#0 of cylinder #i. A 5MB file will need 1000 blocks and occupy from cylinder #1, track #0, sector #0 to cylinder #(i+9), track #6, sector #7, we also assume the size of disk buffer is unlimited.

The disk needs 8ms, which is the seek time, to find the cylinder #i, 8.3 ms to find sector #0 and 8 x (60/3600) seconds to read all 8 tracks data of this cylinder. Then the time needed for the read to move to next adjoining track will be only 1.5 m sec.

Which is the track to track access time. Assume a rotational latency before each new track. Access time = 8+9(8.3÷8x16.6+1.5)+8.3+6x16.6+8/64x16.6 = 1406.6 m sec

d) Burst rate = rows/sec x sectors/row x bytes/sector

= 3600/60 x 64 x 1k = 3.84 MB/sec

Problem 2: A floppy disk drive has a maximum area bits density of 500 bits per mm. its innermost track is at a radius of 2cm, and its outermost track is at radius of 5cm. It rotates at 300rpm.

a) What is the total capacity of the drive?

b) What is its burst data rate in bytes per second?

Solution:

a) At the maximum area density the bit spacing equals the track spacing t, so

500 = 1/t

2

or t = 0.045 mm

The number of tracks = (50-20)/0.045 = 666

The number of bits per track = 40p/0.045 2791

Thus the unformatted or raw capacity is 666 x 2791 = 1,841,400 bits

b) Given 5 raw per second the (raw)burst rate is

Rev/second x bytes/rev = 5 x 2791/8 = 1744 bytes/sec

OPTICAL DISKS

Compact Disk (CD) Technology

The optical technology that is used for CD system is based on 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 towards a photo detector, which detects the stored binary patterns. The laser emits a coherent light beam that is sharply focused 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 brighter beam. But, if a photo detector is used to etect 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 shown in fig. 16 The bottom payer 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 lands. A thin layer of reflecting aluminium material is placed on top of a programmed disk. The aluminium is then covered by a protective acrylic. Finally the topmost layer is deposited and stamped with a label.

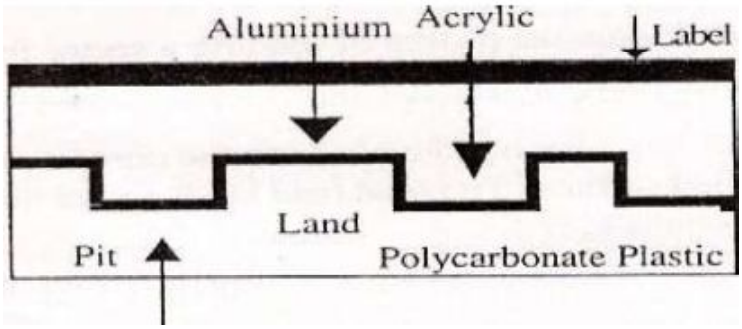


Figure 16 : Optical Disk (Cross Section)

The laser source and the Photo detector are positioned below the polycarbonate plastic. The emitted bean travels through this plastic, reflects off the aluminium layer and travels back toward photo detector.

Some important optical disks are listed below

1. CD-ROM
2. CD-RWs (CD-re writables)
3. DVD technology (Digital Versatile disk)

Problem 1 : A certain two-way associative cache has an access time of 40 nano seconds, compared to a miss time of 90 nano seconds. Without the cache, main memory access time was 90 nano seconds. Without the cache, main memory access time was 70 nano seconds. Running a set of

benchmarks with and without the cache indicated a speed up of 1.4. What is the approximate hit rate?

Solution:

Hit:tn = 40 nano seconds

Miss:tm = 90

No Cache: tn = 70

Access time with Cache ta = h.tn+(1-h)tm = tn-h(tn-tn)

Speed up S = t

n/t

= 70/90-50h = 1.4

Which gives hit ratio h = 80%

Problem 2: A 16 MB main memory has a 32 KB direct mapped Cache with 8 bytes per line.

a. How many lines are there in the Cache.

b. Show how the main memory address is Partitioned?

UNIT-6

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

Unit-6

Arithmetic

6.1 ADDITION AND SUBTRACTION OF SIGNED NUMBERS

In figure-1, the function table of a full-adder is shown; sum and carryout are the outputs for adding equally weighted bits xi and y, in two numbers X and Y. The logic expressions for these functions are also shown, 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 ci, to represent the carry-in to the i

ith
stage, which is
the same as the carryout from the (i - 1) th stage.
The logic expression for s in Figure-1 can be implemented with a 3-input XOR gate. The carryout function, c
i +1

i 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 as shown in the figure-1a.

A cascaded connection of such n full adder blocks, as shown in Figure 1b, 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, Co, into the least-significant-bit (LSB) position [Ist 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-1c.

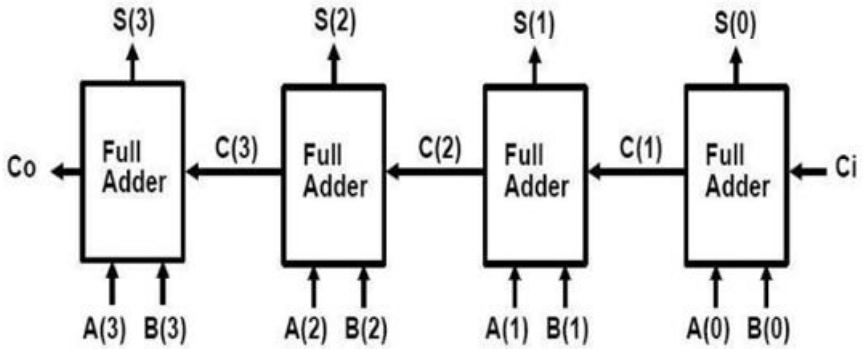


Figure 2 : 4 Bit parallel sector

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.

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

n

at the least-significant-bit (LSB) position to outputs c

n

and s

at the most-significant-bit (MSB) position.

Using the logic implementation indicated in Figure-1, c

n-1

n-1

is available in 2

gate delays, and s

n-1

is one XOR gate delay later. The final carry-out, c

is available after

2n gate delays. Therefore, if a ripple-carry adder is used to implement the addition/subtraction unit shown in Figure-3, all sum bits are available in 2n gate delays, including the delay through the XOR gates on the Y input. Using the implementation cn □ cn-1

for overflow, this indicator is available after 2nn+2 gate delays. In summary, in a parallel adder an nth stage adder can not 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

(carry-out) of stage ith 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

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

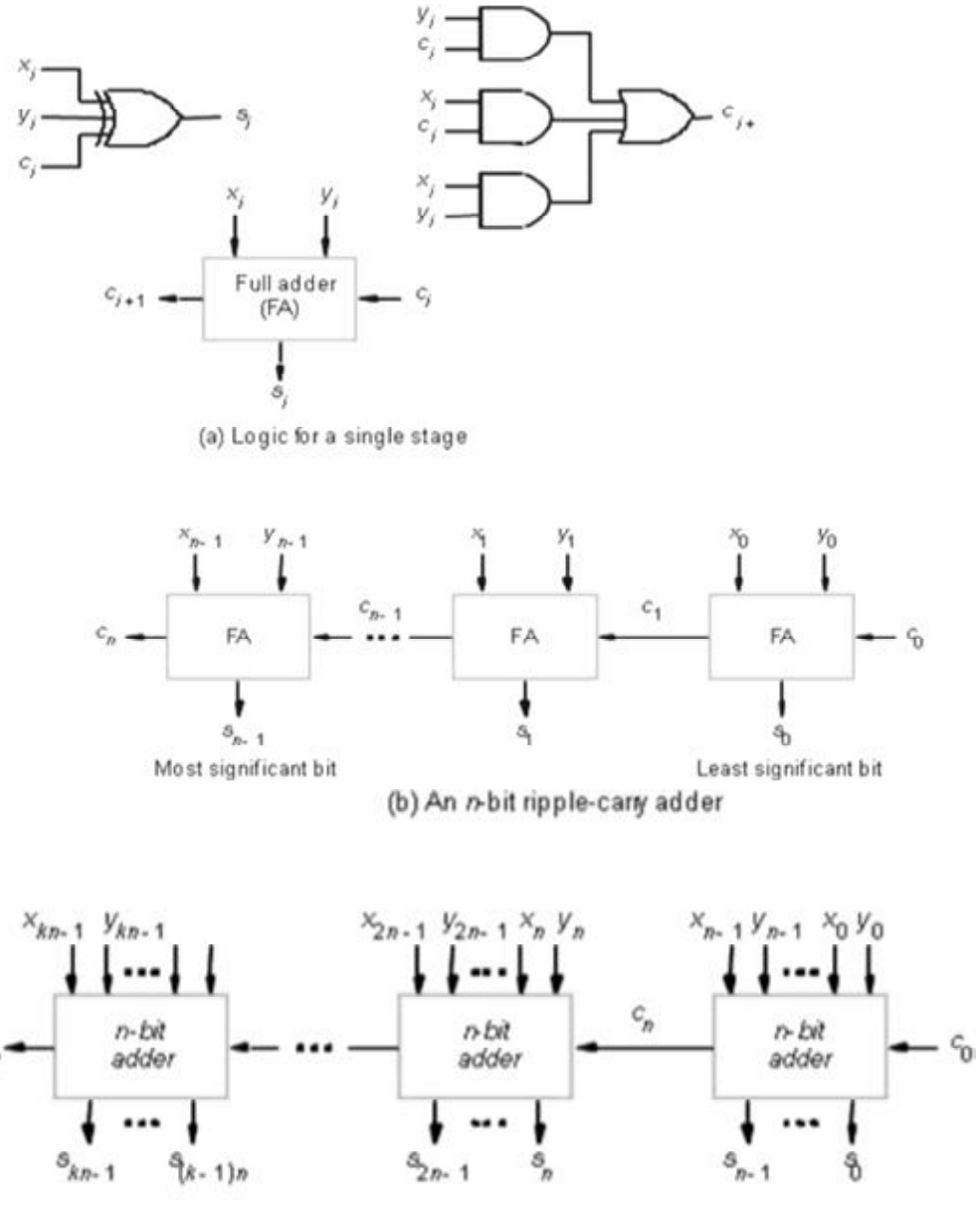


Figure 1 : Addition of binary vectors

ORGANIZATION 10CS46

The above expressions G and P are called carry generate and propagate functions for stage i. If the generate function for stage i is equal to 1, then c = 1, independent of the input carry, c I This occurs when both xi and y ii+1 are 1. The propagate function means that an input carry will produce an output carry when either xi or y or both equal to 1.

Now, using Gi & Pi functions we can decide carry for ith stage even before its previous stages have completed their addition operations. All Gi nd P ii functions can be formed independently and in parallel in only one gate delay after the Xi and Yi inputs are applied to an n-bit adder. Each bit stage contains an AND gate to form Gi an OR gate to form P and a three-input XOR gate to form s . However, a much simpler circuit can be derived by considering the propagate function as $P_i = x_i$, y_i which differs from $P_i = x_i + y$ only when $x_i = y_i = 1$, where $G_i = 1$ (so it does not mater whether Pi is 0 or 1). Then, the basic diagram in figure 5 can be used in each bit stage to predict carry ahead of any stage completing its addition.

Consider the C_{i-1} expression.

$$c_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}c_{i-1}$$

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;

$$c_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \ldots\ldots P_i\ P_{i-1}\ \ldots\ldots\ P_iG_0 + P_iP_{i-1}\ \ldots\ldots\ldots\ P_0G_0$$

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 signals, followed by two gate delays in the AND-OR circuit (SOP expression) for c

i

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.

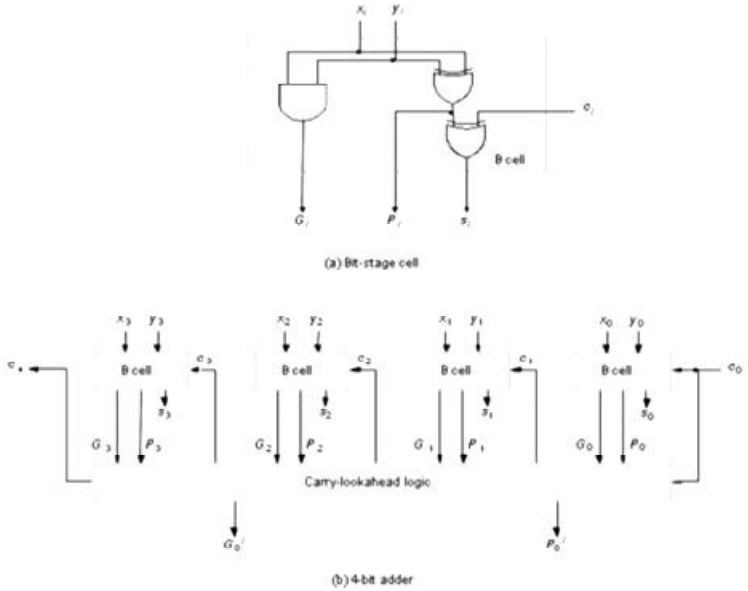


Figure 5 : 4 bit carry look ahead adder

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

$c_1 = G_0 + P_0c_0$: i = 0

$c_2 = G_0 + P_0G_0 + P_0\ P_0c_0$: i = 0

$c_3 = G_0 + P_0G_1 + P_0\ P_1G_0 + P_2P_1P_0c_0$: i = 0

$c_4 = G_0 + P_0G_2 + P_0\ P_1G_1 + P_1P_2P_3G_0 + P_1P_2P_3P_0c_0$: i = 0

The complete 4-bit adder is shown in figure -5b whre the B cell indciates G_i, P_i & S_i generator. The carries are implemented in the block labeled carry look-aed logic. An adder implemented in the form is called a carry look ahead adder. Delay through the adder is 3 gate delays for all carry bits and 4 gate delates for all sum bits. In comparison noe that a 4-bit ripple-carry adder requires 7 gate delays for S3(2n–1) and 8 gate declays (2n) for c_n.

If we try to extend the carry look- ahead 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 c.

For c

(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 6. Eight 4-bit carry look-ahead adders can be connected as in Figure-2 to form a 32-bit adder.

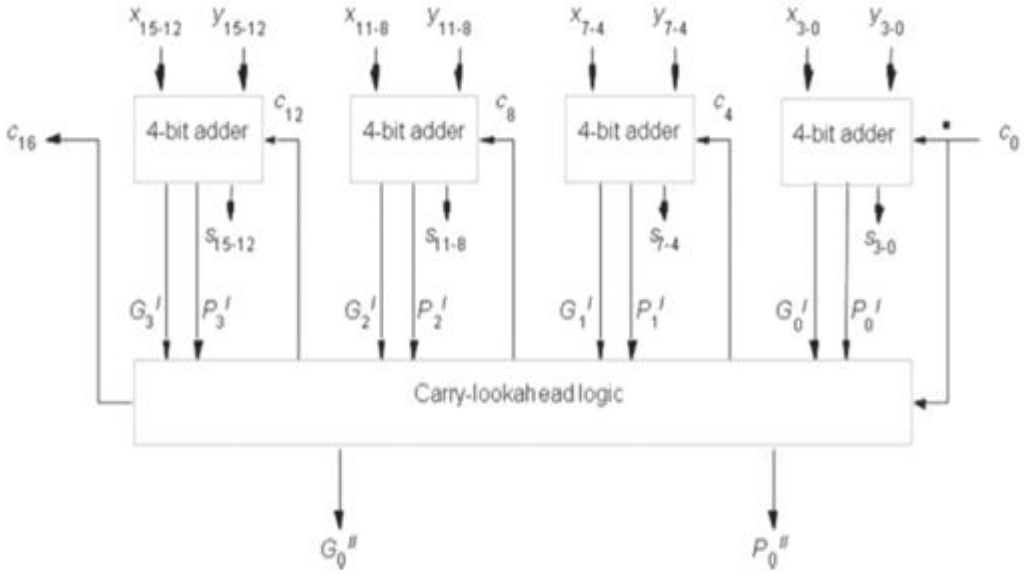


Figure 6 : 16 bit carry-look ahead adder

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 7. 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, q

i. For i in the range of 0 to 3, if q

= 1, add the multiplicand

(appropriately shifted) to the incoming partial product, PPi, to generate the outgoing partial product, PP(i+ 1) & if q

i

i

= 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.

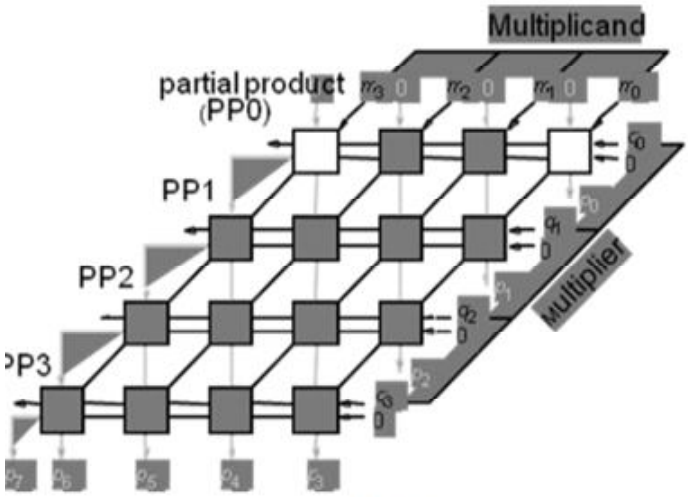


FIG-7a

P7, P6, P5,…,P0 – product.

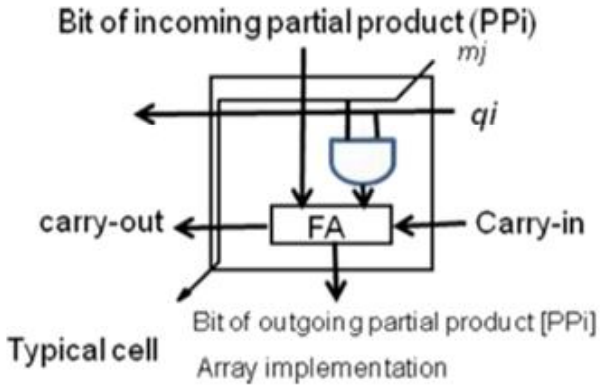
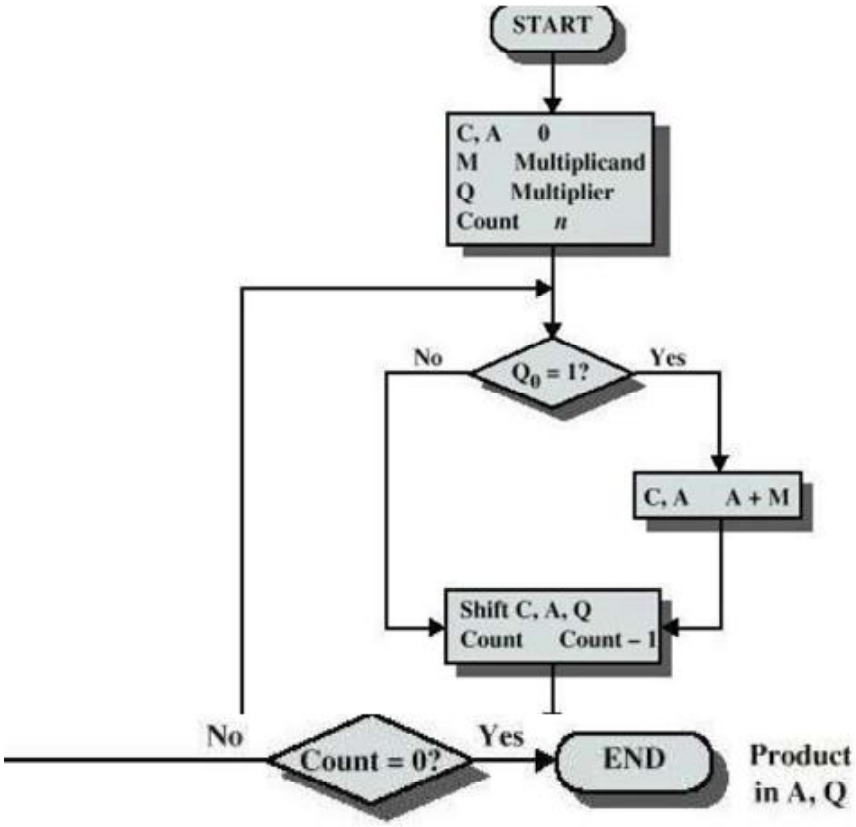


Figure 7b



SHIFT & ADD method flow chart depicts the multiplicatin 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.

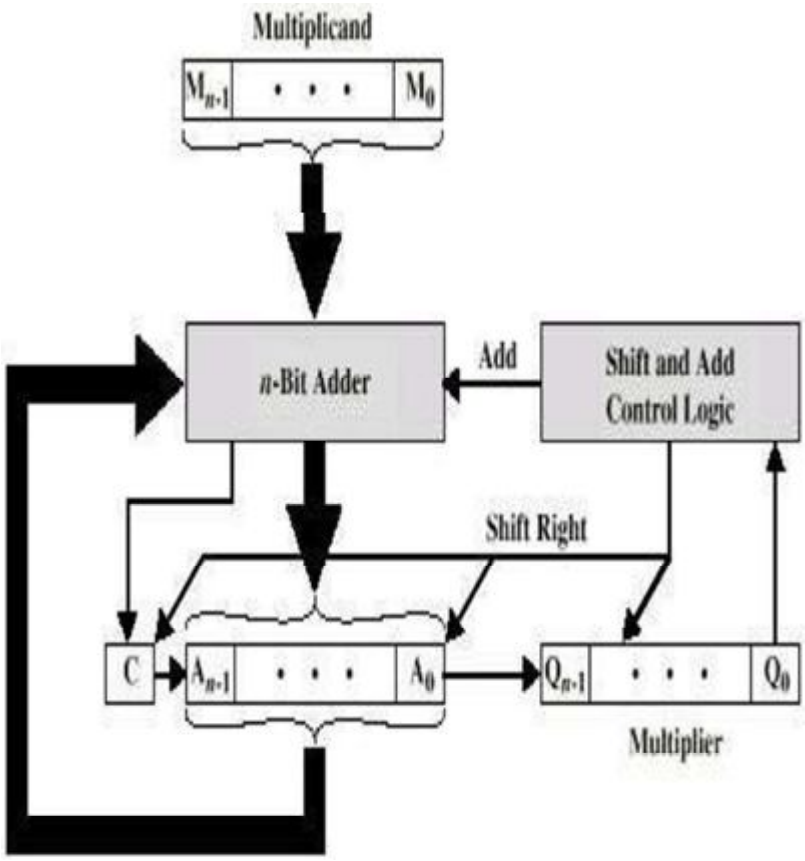


Figure 8a : Block Diagram

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 q 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 position 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.

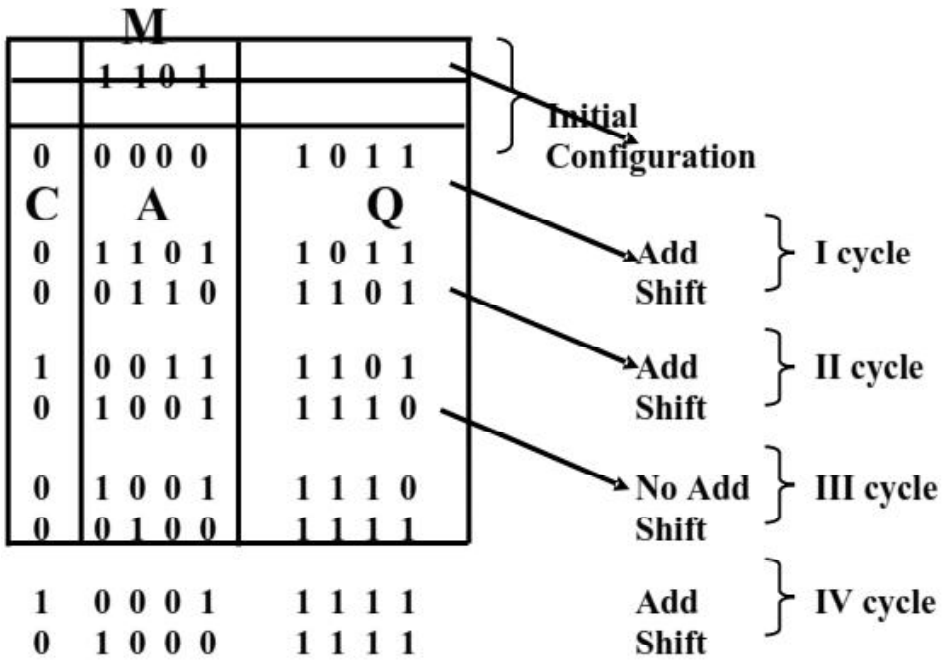


Figure 8b : Product

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 9, 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.

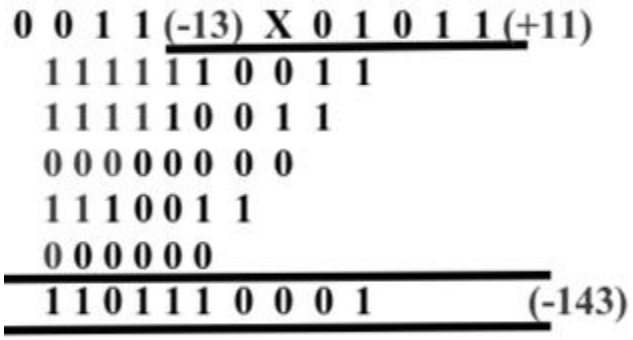


Figure 9

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.

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;

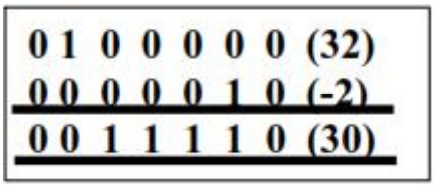


Figure 10

This susggests that the product can be generated by addigng 2’ times the multiplicand to the 2’s-complement of 2’ times the multiplicand. For convenience, we can describe the sequence of required operations by recoding the preceding multiplier as 0 + 1000 – 10. In geneal, in the Booth-Scheme -1, times the shifted multiplicand is selected from 1 to 0, as the multiplier is scanned from right to left.

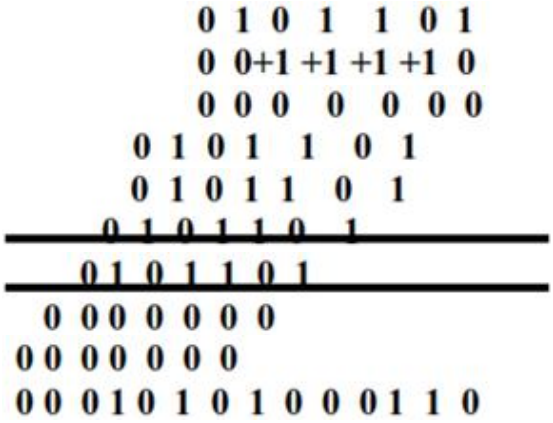


Figure 10 (a) : NormalMultiplication

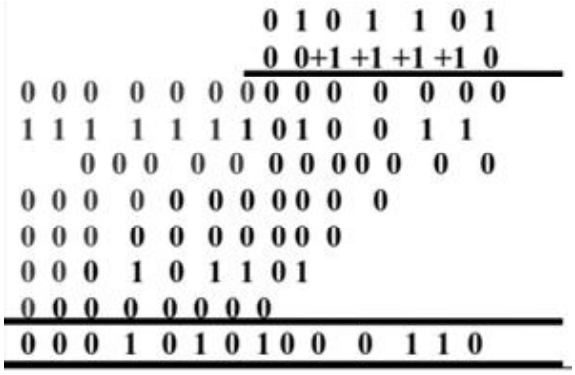


Figure 10 (b) : Booth Multiplication

Figure 10 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 11a 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 11a. 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.

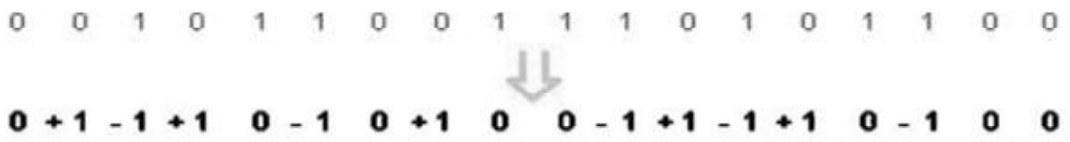


Figure 11a : Booth recoding of a multiplie

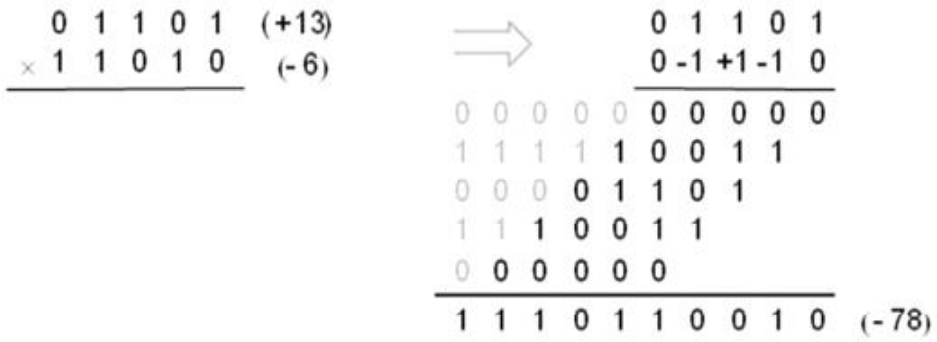


Figure 11b : Booth Multiplication with a negative multiplier

6.5 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 14a shows an example of bit-pair recoding of themultiplier in Figure 11, and Figure 14b shows a table of the multiplicand election decisions for all possibilities. The multiplication operation in figure 11a is shown in Figure 15. It is clear from the example that the bit pair recoding method requires only n/2 summands as against n summands in Booth's algorithm.

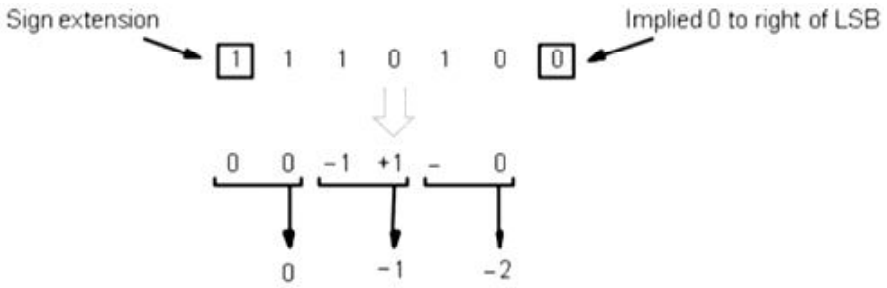
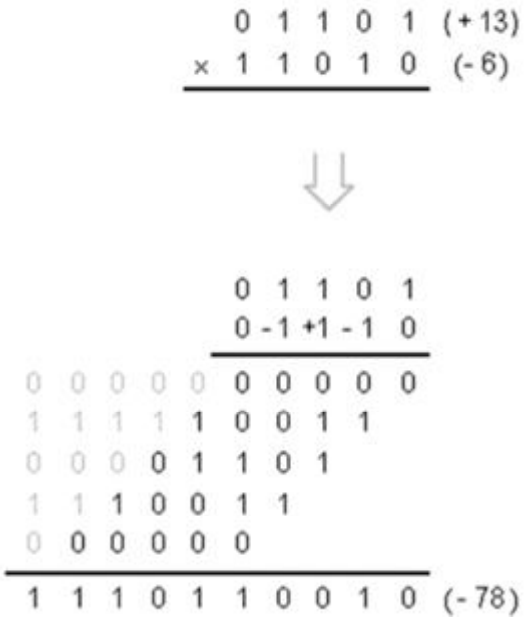


Figure 14 (a) : Example of bit-pair recoding derived from Booth recoding

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		$+1 \times M$
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	$0 \times M$

Figure 14 (b) : Table of multiplicand and selection decisions



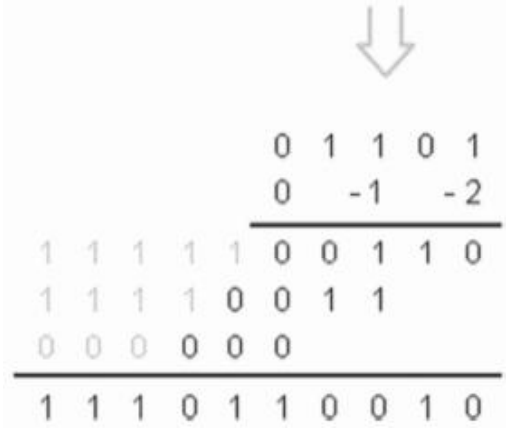


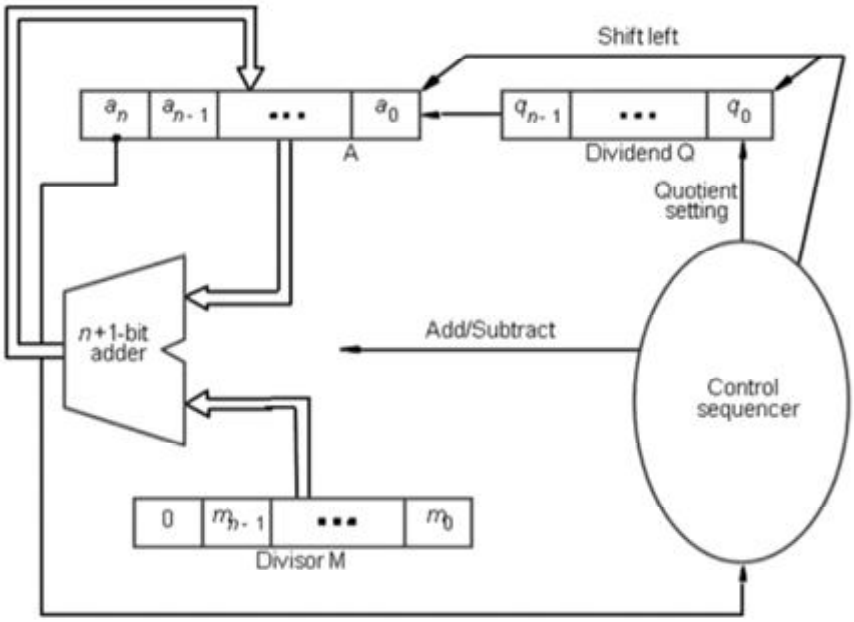
Figure 15 : Multiplication requiring n/2 summands

6.6 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 16 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 16 : Longhand division examples



Circuit arrangement for binary division.

Figure 17 : Binary Division

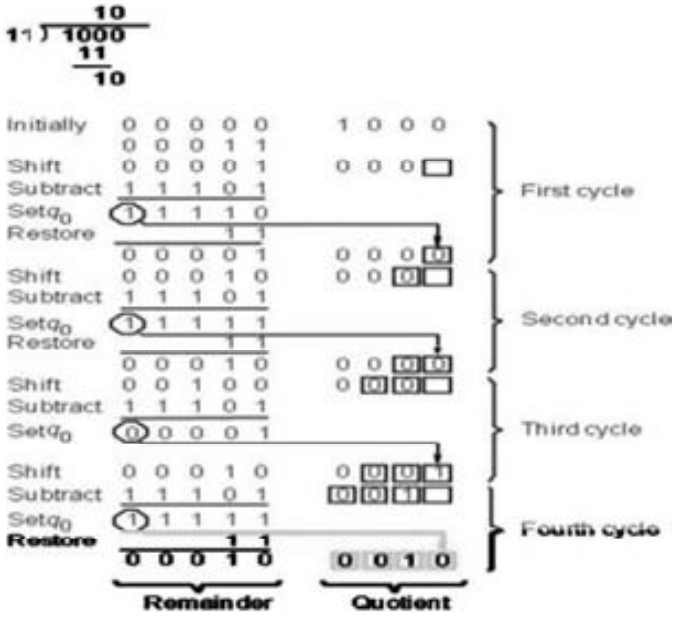


Figure 18 : Restoring Division

Restoring Division

Figure 17 shows a logic circuit arrangement that implements restoring division. Note its similarity to the structure for multiplication that was shown in Figure 8. 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.

Do the following n times:

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 to 0 and add M back to A (that is, restore A); otherwise, set q

   0
   0
   to 1.

Figure 18 shows a 4-bit example as it would be processed by the circuit in Figure 17.

No restoring 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 bit is appropriately set to 0 or 1 after the correct operation has been performed. We can summarize this in the following algorithm for no restoring division.

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 q
   0
   to 1; otherwise, set q
   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 17 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 19 shows how the division example in Figure 18 is executed by the no restoring-division algorithm. There are no simple algorithms for directly performing division on signed operands that are comparable to the algorithms for signed multiplication. In division, the operands can be preprocessed to transform them into positive values. After using one of the algorithms

just discussed, the results are transformed to the correct signed values, as necessary.

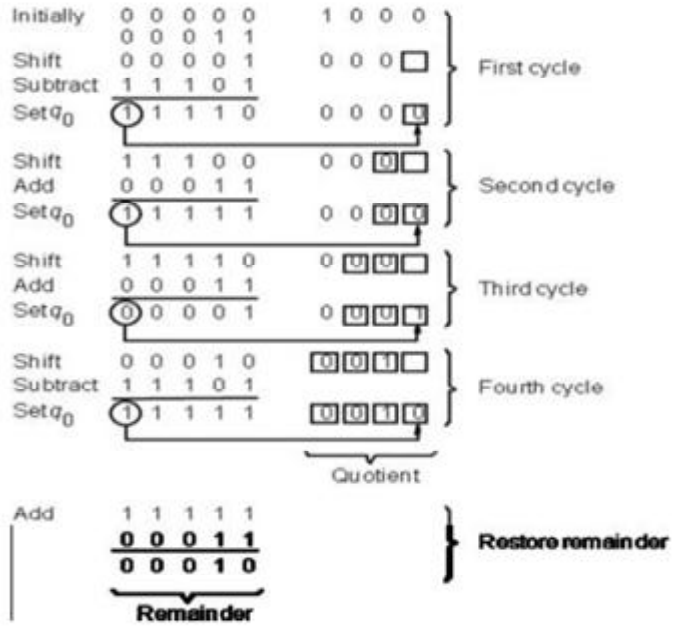


Figure 19 : Non-restroing Division

6.7 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 any where else resulting in real numbers. In the 2's-complement system, the signed value F, represented by the n-bit binary fraction

B = b0.b - 1b-2 …..b-) is given by
F(B) = -bo x 2° + b-1(n-1 x 2-1 +b-x2-2 + ... + b-(n-X) x 2-{n~l) where the range of F is
-1 = F = 1 -2-(n-1)2

Consider the range of values representable in a 32-bit, signed, fixed-point format. Interpreted as integers, the value range is approximately 0 to ±2.15 x 10. 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 x 10
23
mole
-1
) or Planck's constant (6.6254 x 10 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 x 10

23
-
1.0341 x 10
2
, -7.3000 x 10
-14
-27
, 6.6254 -10
, 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.

UNIT-7

Basic Processing Unit: Some Fundamental Concepts, Execution of a Complete Instruction, Multiple Bus Organization, Hard-wired Control, Microprogrammed Control

Unit-7

Basic Processing Unit

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. Figure 1 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.

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:

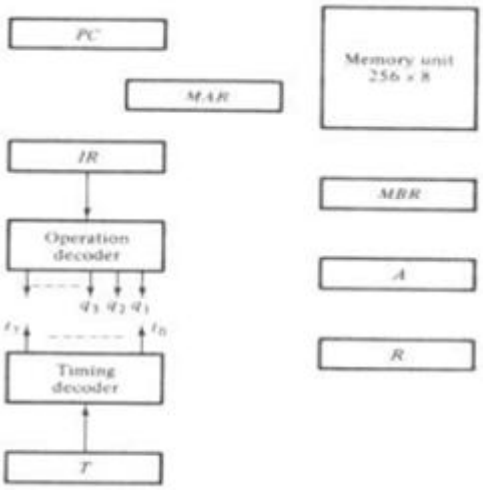


Figure 1

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 [(PC)]

2. Assuming that the memory is byte addressable, increment the contents of the PC by 2, that is, PC [PC] + 2

3. Decode the instruction to understand the operation & generate the control signals necessary to carry out the operation. 4. Carry out the actions specified by the instruction in the IR.

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 decoding phase; and step 4 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 2. 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.

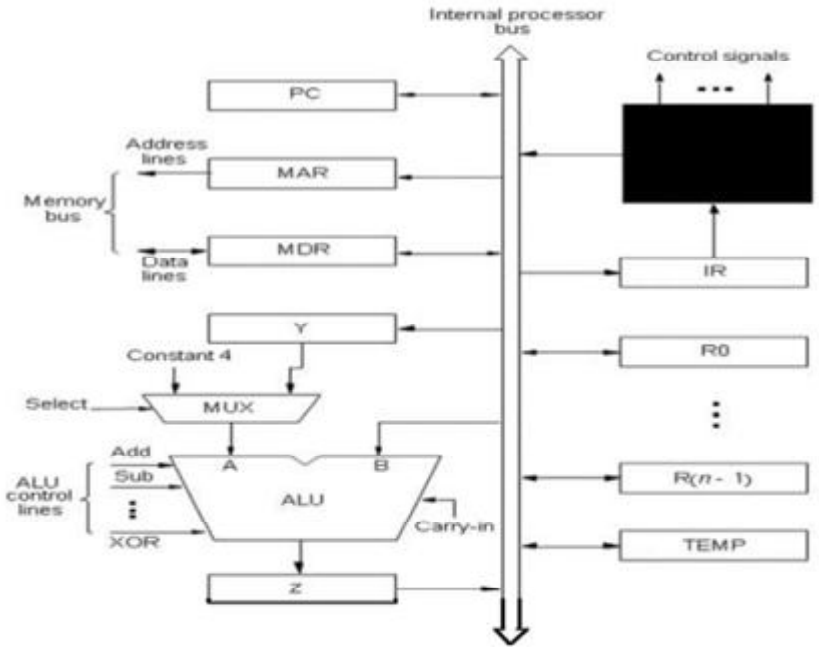


Figure 2

The number and use of the processor registers RO 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 2, 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

We now consider in detail how each of these operations is implemented, using the simple processor model in Figure 2.

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 3. 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
is set to 1, the
data on the bus are loaded into Ri. Similarly, when Ri
out
in
, is set to 1, the contents of
register Ri
out
are placed on the bus. While Ri
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
   out
   out
   by setting Rl
   , tc 1. This places the contents
   of R1 on the processor bus.
2. Enable the input of register R4 by setting R4
   out
   to 1. This loads data from the

processor bus into register R4.

in

All operations and data transfers within the processor take place within time periods 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

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

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

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

, 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

, = 1, the gate drives the

bus to 0 or 1, depending on the value of Q.

7.2 EXECUTION OF A COMPLETE INSTRUCTION

out

Let us now put together the sequence of elementary operations required to execute one 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**

The listing shown in figure 7 above indicates the sequence of control steps required to perform these operations for the single-bus architecture of Figure 2. Instruction execution proceeds as follows. In step 1, 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.

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	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Figure 8

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. 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-IRou Add, Z, 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.

7.3 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 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) described in Chapter 5. The register file in Figure 9 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 in Figure 2.

A second feature in Figure 9 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.

The control sequence for executing this instruction is given in Figure 10. 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.

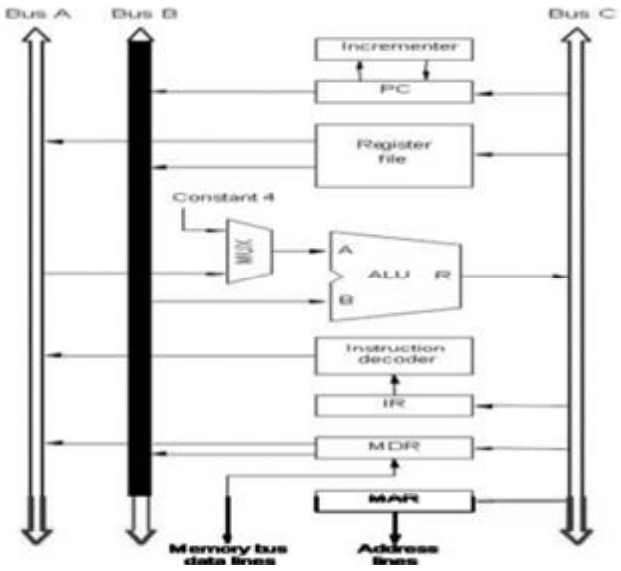


Fig 9

Consider the three-operand instruction

Add R4.R5.R6

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 10

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.

7.4 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. We discuss each of these techniques in detail, starting with hardwired

control in this section. Consider the sequence of control signals given in Figure 7. Each step in this sequence is completed in one clock period. A counter may be used to keep track of the control steps, as shown in Figure 11. Each state, or count, of this counter corresponds to one control step. The required control signals are determined by the following information:

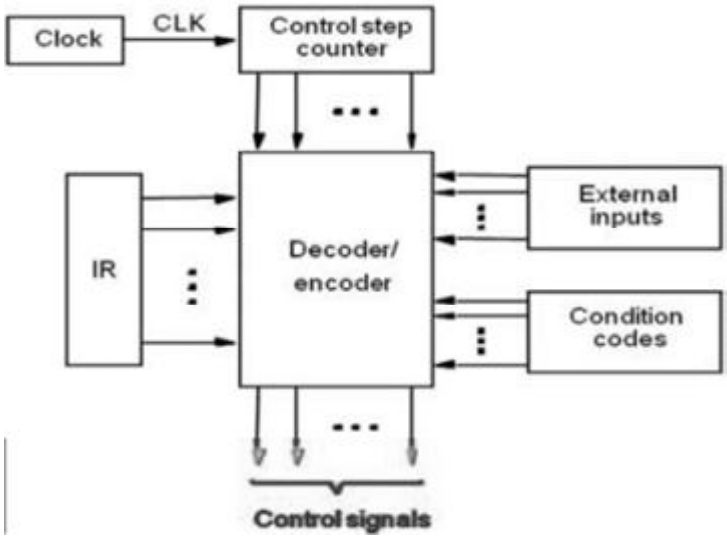


Figure 11

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 11 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 12. 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
is set to 1, and all other lines

re set to 0. (For design details of decoders, refer to Appendix A.) The input signals to the encoder block in Figure 12 are combined to generate the individual control signals Y,
PC
out
, Add, End, and so on. An example of how the encoder generates the Z
control signal for the processor organization in Figure 2 is given in Figure 13. This circuit implements the logic function
Z

in
=T
1
+T
6
- ADD + T
-BR+---
This signal is asserted during time slot Ti for all instructions, during T
4
for an Add
instruction, during T4 for an unconditional branch instruction, and so on. The logic
function for Z
is derived from the control sequences in Figures 7 and 8. As another example, Figure 14 gives a circuit that generates the End control signal from the logic function
in
End = T7 o ADD + T5 o BR + (T5 o N + T4 o N) o BRN + o o o

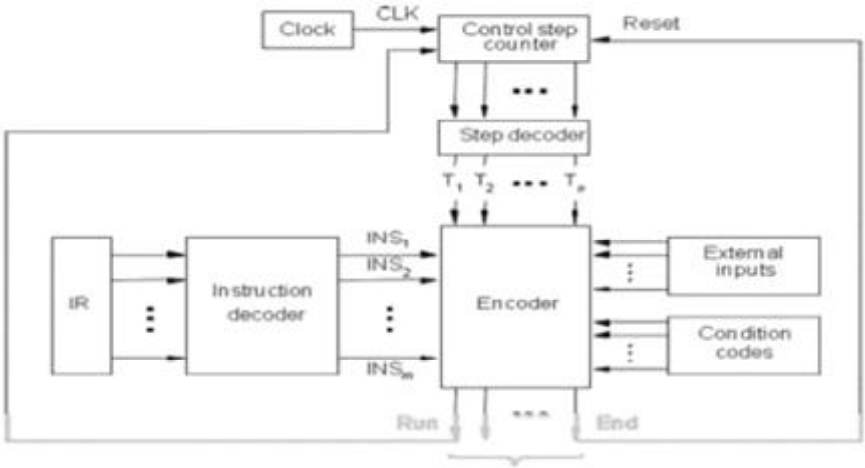


Figure 12

The End signal starts a new instruction fetch cycle by resetting the control step counter to its starting value. Figure 12 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.

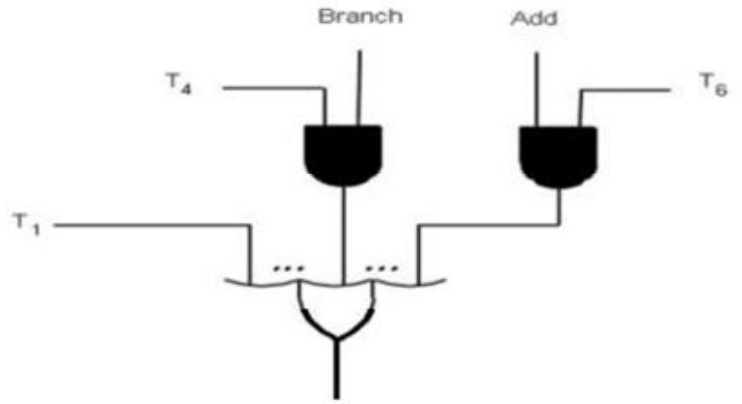


Figure 13a

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.

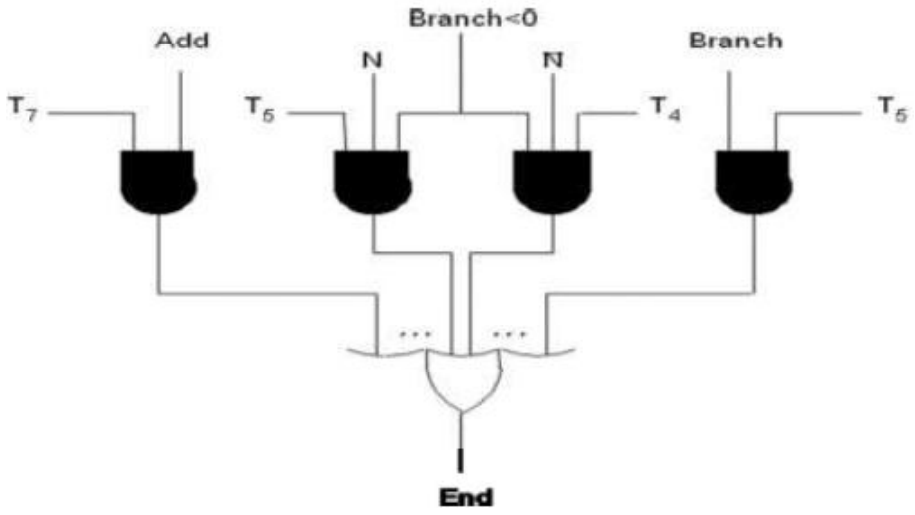
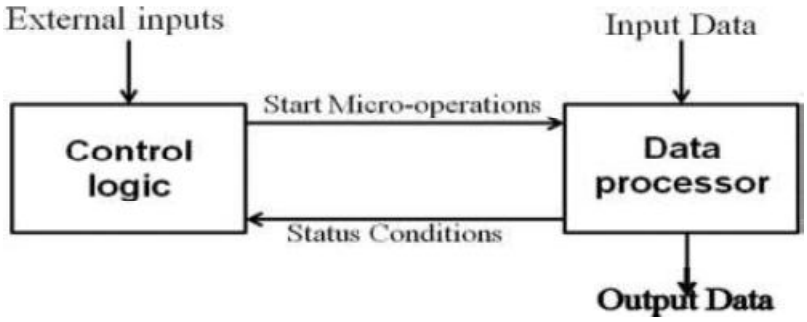


Figure 13b

7.5 MICROPROGRAMMED CONTROL

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



The control signals required inside the processor can be generated using a control step counter and a decoder/ encoder circuit. Now we discuss an alternative scheme, called micro programmed control, in which control signals are generated by a program similar to machine language programs.

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 15

First, we introduce some common terms. A control word (CW) is a word whose individual bits represent the various control signals in Figure 12. Each of the control steps in the control sequence of an instruction defines a unique combination of Is and Os in the CW. The CWs corresponding to the 7 steps of Figure 6 are shown in Figure 15. We have assumed that Select Y is represented by Select = 0 and Select4 by Select = 1. A sequence of CWs corresponding to the control sequence of a machine instruction constitutes the micro routine for that instruction, and the individual control words in this micro routine 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. This suggests organizing the control unit as shown in Figure 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.

One important function of the control unit cannot be implemented by the simple organization in Figure 16. This is the situation that arises when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action. In the case of hardwired control, this situation is handled by including an appropriate logic function, in the encoder circuitry. In micro programmed control, an alternative approach 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.

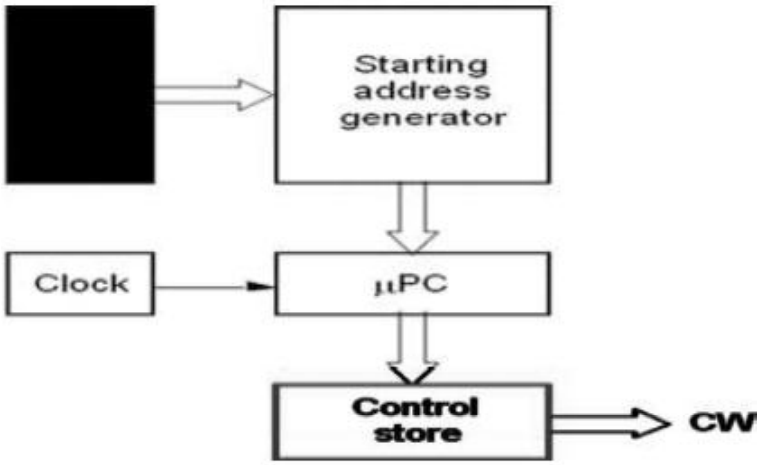


Figure 16

The instruction Branch <0 may now be implemented by a micro routine such as that shown in Figure 17. After loading this instruction into IR, a branch microinstruction transfers control to the corresponding micro routine, which is assumed to start at location 25 in the control store. This address is the output of staring address generator block 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 0 to fetch a new machine instruction. Otherwise the microinstruction at location 27 loads this address into the PC.

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 17

To support micro program branching, the organization of the control unit should be modified as shown in Figure 18. The starting address generator block of Figure 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.

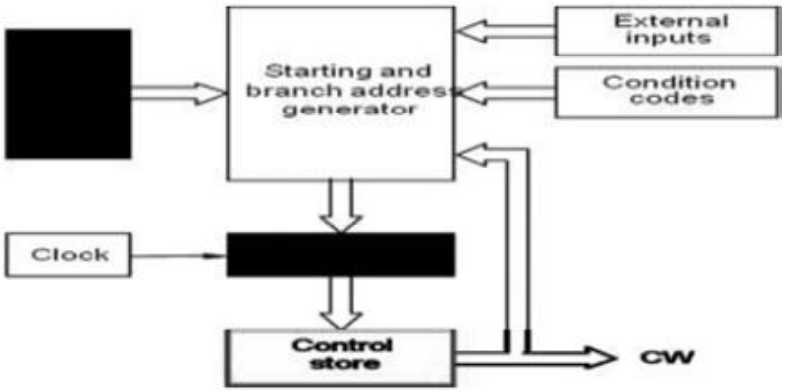


Figure 18

Microinstructions

Having described a scheme for sequencing microinstructions, we now take a closer look at the format of individual microinstructions. A straightforward way to structure Microinstruction is to assign one bit position to each control signal, as in Figure 15.

However, this scheme has one serious drawback - assigning individual bits to each control signal results in long microinstructions because the number of required signals is usually large. Moreover, only a few bits are set to 1 (to be used for active gating) in any given microinstruction, which means the available bit space is poorly used. Consider again the simple processor of Figure 2, and assume that it contains only four general-purpose registers, R0, Rl, R2, and R3. Some of the connections in this processor are permanently enabled, such as the output of the IR to the decoding circuits and both inputs to the ALU. The remaining connections to various registers require a total of 20 gating signals. Additional control signals not shown in the figure are also needed, including the

Read, Write, Select, WMFC, and End signals. Finally, we must specify the function to be performed by the ALU. Let us assume that 16 functions are provided, including Add, Subtract, AND, and XOR. These functions depend on the particular ALU used and do not necessarily have a one-to-one correspondence with the machine instruction OP codes. In total, 42 control signals are needed. If we use the simple encoding scheme described earlier, 42 bits would be needed in each microinstruction. Fortunately, the length of the microinstructions can be reduced easily. Most signals are not needed simultaneously, and many signals are mutually exclusive. For example, only one function of the ALU can be activated at a time. The source for a data transfer must be unique because it is not possible to gate the contents of two different registers onto the bus at the same time. Read and Write signals to the memory cannot be active simultaneously. This suggests that signals can be grouped so that all mutually exclusive signals are placed in the same group. Thus, at most one micro operation per group is specified in any microinstruction. Then it is possible to use a binary coding scheme to represent the signals within a group. For example, four bits suffice to represent the 16 available functions in the ALU. Register output control signals can be placed in a group consisting of PC

out
, MDR
out
, Z
out
, Offset
out
, R0
,
R3

out

, and TEMP

. Any one of these can be selected by a unique 4-bit code.

Further natural groupings can be made for the remaining signals. Figure 19

out

shows an example of a partial format for the microinstructions, in which each group occupies a field large enough to contain the required codes. Most fields must include one inactive code for the case in which no action is required. For example, the all-zero pattern in Fl indicates that none of the registers that may be specified in this field should have its contents placed on the bus. An inactive code is not needed in all fields. For example, F4 contains 4 bits that specify one of the 16 operations performed in the ALU. Since no spare code is included, the ALU is active during the execution of every microinstruction.

However, its activity is monitored by the rest of the machine through register Z, which is loaded only when the

Z

signal is activated.

Grouping control signals into fields requires a little more hardware because

in

decoding circuits must be used to decode the bit patterns of each field into individual control signals. The cost of this additional hardware is more than offset by the reduced number of bits in each microinstruction, which results in a smaller control store. In Figure 19, only 20 bits are needed to store the patterns for the 42 signals.

Figure 19

So far we have considered grouping and encoding only mutually exclusive control signals. We can extend this idea by enumerating the patterns of required signals in all possible microinstructions. Each meaningful combination of active control signals can then be assigned a distinct code that represents the microinstruction. Such full encoding is likely to further reduce the length of micro words but also to increase the complexity of the required decoder circuits.

Highly encoded schemes that use compact codes to specify only a small number of control functions in each microinstruction are referred to as a vertical organization. On the other hand, the minimally encoded scheme of Figure 15, in which many resources can be controlled with a single microinstruction, is called a horizontal organization. The horizontal approach is useful when a higher operating speed is desired and when the machine structure allows parallel use of resources. The vertical approach results in considerably slower operating speeds because more microinstructions are needed to perform the desired control functions. Although fewer bits are required for each microinstruction, this does not imply that the total number of bits in the control store is smaller. The significant factor is that less hardware is needed to handle the execution of microinstructions. Horizontal and vertical organizations represent the two organizational extremes in micro programmed control. Many intermediate schemes are also possible, in which the degree of encoding is a design parameter. The layout in Figure 19 is a horizontal organization because it groups only mutually exclusive micro operations in the same fields. As a result, it does not limit in any way the processor's ability to perform various micro operations in parallel.

Although we have considered only a subset of all the possible control signals, this subset is representative of actual requirements. We have omitted some details that are not essential for understanding the principles of operation.

UNIT-8

Multicores, Multiprocessors, and Clusters: Performance, The Power Wall, The Switch from Uniprocessors to Multiprocessors, Amdahl's Law, Shared Memory Multiprocessors, Clusters and other Message Passing Multiprocessors, Hardware Multithreading, SISD, IMD, SIMD, SPMD, and Vector.

Unit-8

Multicores, Multiprocessors, and Clusters

8.1 PERFORMANCE

Computer performance is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used. Depending on the context, good computer performance may involve one or more of the following:

- Short response time for a given piece of work
- High throughput (rate of processing work)
- Low utilization of computing resource(s)
- High availability of the computing system or application
- Fast (or highly compact) data compression and decompression
- High bandwidth / short data transmission time

Technical performance metrics

There are a wide variety of technical performance metrics that indirectly affect overall computer performance. Because there are too many programs to test a CPU's speed on all of them, benchmarks were developed. The most famous benchmarks are the SPECint and SPECfpbenchmarks developed by Standard Performance Evaluation Corporation and the ConsumerMark benchmark developed by the Embedded Microprocessor Benchmark Consortium EEMBC.

Some important measurements include:

- **Instructions per second -** Most consumers pick a computer architecture (normally Intel IA32 architecture) to be able to run a large base of pre-existing, pre-compiled software. Being relatively uninformed on computer benchmarks, some of them pick a particular CPU based on operating frequency (see megahertz myth). oFLOPS - The number of floating-point operations per second is often important in selecting computers for scientific computations.
- **Performance per watt - System designers** building parallel computers, such as Google, pick CPUs based on their speed per watt of power, because the cost of powering the CPUoutweighs the cost of the CPU itself. [2][3]

- Some system designers building parallel computers pick CPUs based on the speed per dollar.
- System designers building real-time computing systems want to guarantee worst-case response. That is easier to do when the CPU has low interrupt latency and when it has deterministic response. (DSP[disambiguation needed][clarification needed])
- Computer programmers who program directly in assembly language want a CPU to support a full-featured instruction set.
- **Low power -** For systems with limited power sources (e.g. solar, batteries, human power).
- **Small size or low weight -** for portable embedded systems, systems for spacecraft.
- **Environmental impact -** Minimizing environmental impact of computers during manufacturing and recycling as well as during use. Reducing waste, reducing hazardous materials. (see Green computing).
- **Giga-updates per second -** a measure of how frequently the RAM can be updated Occasionally a CPU designer can find a way to make a CPU with better overall performance by improving one of these technical performance metrics without sacrificing any other (relevant) technical performance metric-for example, building the CPU out of better, faster transistors. However, sometimes pushing one technical performance metric to an extreme leads to a CPU with worse overall performance, because other important technical performance metrics were sacrificed to get one impressive-looking number-for example, the megahertz myth.

8.2 THE POWER WALL

A multi-core processor is a single computing component with two or more independent actual central processing units (called "cores"), which are the units that read and execute program instructions. [1] The instructions are ordinary CPU instructions such as add, move data, and branch, but the multiple cores can run multiple instructions at the same time, increasing overall speed for programs amenable to parallel computing. [2] Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP), or onto multiple dies in a single chip package.

Processors were originally developed with only one core. A dual-core processor has two cores (e.g. AMD Phenom II X2, Intel Core Duo), a quad-core processor contains four cores (e.g. AMD Phenom II X4, Intel's quad-core processors, see i3, i5, and i7 atIntel Core), a hexa-core processor contains six cores (e.g. AMD Phenom II X6, Intel Core i7 Extreme Edition 980X), an octa-core processor contains eight cores (e.g. Intel Xeon E7-2820, AMD FX-8150). A multi-core processor implements multiprocessing in a single physical package. Designers may couple cores in a multi-core device tightly or loosely. For example, cores may or may not share caches, and they may implement message passing or shared memory inter-core communication methods. Common

network topologies to interconnect cores include bus, ring, two-dimensional mesh, and crossbar. Homogeneous multi-core systems include only identical cores, heterogeneous multi-core systems have cores that are not identical. Just as with single-processor systems, cores in multi-core systems may implement architectures such as superscalar, VLIW, vector processing, SIMD, or multithreading.

Multi-core processors are widely used across many application domains including general-purpose, embedded, network, digital signal processing (DSP), and graphics.

The improvement in performance gained by the use of a multi-core processor depends very much on the software algorithms used and their implementation. In parallel simultaneously on multiple cores; this effect is described by Amdahl's law. In the best case, so-called embarrassingly parallel problems may realize speedup factors near the number of cores, or even more if the problem is split up enough to fit within each core's cache(s), avoiding use of much slower main system memory. Most applications, however, are not accelerated so much unless programmers invest a prohibitive amount of effort in re-factoring the whole problem.

[3] The parallelization of software is a significant ongoing topic of research.

8.3 THE SWITCH FROM UNIPROCESSORS TO MULTIPROCESSORS:

The steps are:

- In Control Panel, click System, choose the Hardware tab, then click the Device Manager button.
- Select the Computer node and expand it.
- Double-click the object listed there (on my system, it is called Standard PC), then choose the Driver tab. Click the Update Driver button.
- On the Upgrade Device Driver Wizard, click the Next button, then select Display a known list of drivers for this device so that I can choose a specific driver. Click the Next button.
- On the Select Device Driver page, select Show all hardware of this device class.
- Select the HAL that matches your new configuration, multiprocessor or uniprocessor. Click the Next button. Check that the wizard is showing the configuration you want, then click the Next button to complete the wizard. particular, possible gains are limited by the fraction of the software that can be run in.

8.4 AMDAHL'S LAW

Amdahl's law, also known as Amdahl's argument is named after computer architect Gene Amdahl, and is used to find the maximum expected improvement to an overall system when only part of the system is improved. It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors. It was presented at the AFIPS Spring Joint Computer

Conference in 1967. The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. For example, if a program needs 20 hours using a single processor core, and a particular portion of 1 hour cannot be parallelized, while the remaining promising portion of 19 hours (95%) can be parallelized, then regardless of how many processors we devote to a parallelized execution of this program, the minimum execution time cannot be less than that critical 1hour. Hence the speedup is limited up to 20×, as the diagram illustrates.

Amdahl's law is a model for the relationship between the expected speedup of parallelized implementations of an algorithm relative to the serial algorithm, under the assumption that the problem size remains the same when parallelized. For example, if for a given problem size a parallelized implementation of an algorithm can run 12% of the algorithm's operations arbitrarily quickly (while the remaining 88% of the operations are not parallelizable), Amdahl's law states that the maximum speedup of the parallelized version is 1/(1 - 0.12) = 1.136 times as fast as the non-parallelized implementation. More technically, the law is concerned with the speedup achievable from an improvement to a computation that affects a proportion P of that computation where the improvement has a speedup of S. (For example, if 30% of the computation may be the subject of a speed up, P will be 0.3; if the improvement makes the portion affected twice as fast, S will be 2.) Amdahl's law states that the overall speedup of applying the improvement will be: To see how this formula was derived, assume that the running time of the old computation was 1, for some unit of time. The running time of the new computation will be the length of time the unimproved fraction takes (which is 1 - P), plus the length of time the improved fraction takes. The length of time for the improved part of the computation is the length of the improved part's former running time divided by the speedup, making the length of time of the improved part P/S. The final speedup is computed by dividing the old running time by the new running time, which is what the above formula does.

Here's another example. We are given a sequential task which is split into four consecutive parts: P1, P2, P3 and P4 with the percentages of runtime being 11%, 18%, 23% and 48% respectively. Then we are told that P1 is not sped up, so S1 = 1, while P2 is sped up 5×, P3 is sped up 20×, and P4 is sped up 1.6×. By using the formula P1/S1 + P2/S2 + P3/S3 + P4/S4, we find the new sequential running time is: or a little less than 1/2 the original running time. Using the formula (P1/S1 + P2/S2 + P3/S3 + P4/S4)-1, the overall speed boost is 1 / 0.4575 = 2.186, or a little more than double the original speed. Notice how the 20× and 5× speedup don't have much effect on the overall speed when P1 (11%) is not sped up, and P4 (48%) is sped up only 1.6 times.

8.5 SHARED MEMORY MULTIPROCESSORS SHARED MEMORY MULTIPROCESSOR

Shared memory multiprocessors have multiple CPUs all of which share the same address space. This means there is only one memory accessed by all CPUs on an equal basis. Shared memory systems can be either SIMD or MIMD. Single-CPU vectorprocessors can be regarded

as an example of the former, while the multi-CPU models of these machines are examples of the latter. The main problem with shared memory systems is the connection of the CPUs to each other and to the memory. Various interconnection alternatives have been used, including crossbar, Ω-network, and central databus. The IBM SP2, the Meiko CS-2, and the Cenju-3 use the Ω-network.\cite{van-der-Steen-overview}

8.6 CLUSTERS AND OTHER MESSAGE PASSING MULTIPROCESSORS

What is MPI?

1. MPI stands for Message Passing Interface and its standard is set by the Message Passing Interface Forum
2. It is a library of subroutines/functions, NOT a language
3. MPI subroutines are callable from Fortran and C
4. Programmer writes Fortran/C code with appropriate MPI library calls, compiles with Fortran/C compiler, then links with Message Passing library
5. For large problems that demand better turn-around time (and access to more memory)
6. For Fortran "dusty deck", often it would be very time-consuming to rewrite code to take advantage of parallelism. Even in the case of SMP, as are the SGI PowerChallengeArray and Origin2000, automatic parallelizer might not be able to detect parallelism.
7. For distributed memory machines, such as a cluster of Unix work stations or a cluster of NT/Linux PCs.
8. Maximize portability; works on distributed and shared memory architectures.

Preliminaries of MPI Message Passing

- In a user code, wherever MPI library calls occur, the following header file must be included: #include "mpi.h" for C code or include "mpif.h" for Fortran code

  These files contain definitions of constants, prototypes, etc. which are neccessary to compile a program that contains MPI library calls
- MPI is initiated by a call to MPI_Init. This MPI routine must be called before any other MPI routines and it must only be called once in the program.
- MPI processing ends with a call to MPI_Finalize.
- Essentially the only difference between MPI subroutines (for Fortran programs) and MPI functions (for C programs) is the error reporting flag. In fortran, it is returned as the last member of the subroutine's argument list. In C, the integer error flag is returned through the function value. Consequently, MPI fortran routines always contain one additional variable in the argument list than the C counterpart.

- C's MPI function names start with "MPI_" and followed by a character string with the leading character in upper case letter while the rest in lower case letters. Fortran subroutines bear the same names but are case-insensitive.

8.7 HARDWARE MULTITHREADING

Multithreading computer central processing units have hardware support to efficiently execute multiple threads. These are distinguished from multiprocessing systems (such as multi-core systems) in that the threads have to share the resources of a single core: the computing units, the CPU caches and the translation lookaside buffer(TLB). Where multiprocessing systems include multiple complete processing units, multithreading aims to increase utilization of a single core by using thread-level as well as instruction-level parallelism. As the two techniques are complementary, they are sometimes combined in systems with multiple multithreading CPUs and in CPUs with multiple multithreading cores.

Overview

The multithreading paradigm has become more popular as efforts to further exploit instruction level parallelism have stalled since the late-1990s. This allowed the concept of throughput computing to re-emerge to prominence from the more specialized field of transaction processing:

- Even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multi-tasking among multiple threads or programs.
- Techniques that would allow speedup of the overall system throughput of all tasks would be a meaningful performance gain.

The two major techniques for throughput computing are multiprocessing and multithreading.

Some advantages include

oIf a thread gets a lot of cache misses, the other thread(s) can continue, taking advantage of the unused computing resources, which thus can lead to faster overall execution, as these resources would have been idle if only a single thread was executed. oIf a thread cannot use all the computing resources of the CPU (because instructions depend on each other's result), running another thread can avoid leaving these idle. oIf several threads work on the same set of data, they can actually share their cache, leading to better cache usage or synchronization on its values.

Some criticisms of multithreading include

- Multiple threads can interfere with each other when sharing hardware resources such as caches or translation lookaside buffers (TLBs).
- Execution times of a single thread are not improved but can be degraded, even when only one thread is executing. This is due to slower frequencies and/or additional pipeline stages that are necessary to accommodate thread-switching hardware.

- Hardware support for multithreading is more visible to software, thus requiring more changes to both application programs and operating systems than multiprocessing. The mileage thus varies; Intel claims up to 30 percent improvement with its HyperThreading technology, while a synthetic program just performing a loop of non-optimized dependent floating-point operations actually gains a 100 percent speed improvement when run in parallel. On the other hand, hand-tuned assembly languageprograms using MMX or Altivec extensions and performing data pre-fetches (as a good video encoder might), do not suffer from cache misses or idle computing resources. Such programs therefore do not benefit from hardware multithreading and can indeed see degraded performance due to contention for shared resources.

  Hardware techniques used to support multithreading often parallel the software techniques used for computer multitasking of computer programs.

- Thread scheduling is also a major problem in multithreading.

Concept

The simplest type of multi-threading occurs when one thread runs until it is blocked by an event that normally would create a long latency stall. Such a stall might be a cache-miss that has to access off-chip memory, which might take hundreds of CPU cycles for the data to return. Instead of waiting for the stall to resolve, a threaded processor would switch execution to another thread that was ready to run. Only when the data for the previous thread had arrived, would the previous thread be placed back on the list of ready-to-run threads.

For example:

1. Cycle i : instruction j from thread A is issued
2. Cycle i+1: instruction j+1 from thread A is issued
3. Cycle i+2: instruction j+2 from thread A is issued, load instruction which misses in all caches
4. Cycle i+3: thread scheduler invoked, switches to thread B
5. Cycle i+4: instruction k from thread B is issued
6. Cycle i+5: instruction k+1 from thread B is issued

Conceptually, it is similar to cooperative multi-tasking used in real-time operating systems in which tasks voluntarily give up execution time when they need to wait upon some type of the event.

Terminology

This type of multi threading is known as Block or Cooperative or Coarse-grained multithreading.

Hardware cost

The goal of multi-threading hardware support is to allow quick switching between a blocked thread and another thread ready to run. To achieve this goal, the hardware cost is to replicate the program visible registers as well as some processor control registers (such as the program counter). Switching from one thread to another thread means the hardware switches from using one register set to another. Such additional hardware has these benefits:

o The thread switch can be done in one CPU cycle.

o It appears to each thread that it is executing alone and not sharing any hardware resources with any other threads[dubious - discuss]. This minimizes the amount of software changes needed within the application as well as the operating system to support multithreading.

In order to switch efficiently between active threads, each active thread needs to have its own register set. For example, to quickly switch between two threads, the register hardware needs to be instantiated twice.

Examples

- Many families of microcontrollers and embedded processors have multiple register banks to allow quick context switching for interrupts. Such schemes can be considered a type of block multithreading among the user program thread and the interrupt threads

1. Cycle i+1: an instruction from thread B is issued
2. Cycle i+2: an instruction from thread C is issued

The purpose of interleaved multithreading is to remove all data dependency stalls from the execution pipeline. Since one thread is relatively independent from other threads, there's less chance of one instruction in one pipe stage needing an output from an older instruction in the pipeline. Conceptually, it is similar to pre-emptive multi-tasking used in operating systems. One can make the analogy that the time-slice given to each active thread is one CPU cycle.

Hardware costs

In addition to the hardware costs discussed for interleaved multithreading, SMT has the additional cost of each pipeline stage tracking the Thread ID of each instruction being processed. Again, shared resources such as caches and TLBs have to be sized for the large number of active threads being processed.

Examples

- DEC (later Compaq) EV8 (not completed)
- Intel Hyper-Threading
- IBM POWER5

- Sun Microsystems UltraSPARC T2
- MIPS MT
- CRAY XMT

8.8 SISD, IMD, SIMD, SPMD, AND VECTOR

Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Thus, such machines exploit data level parallelism. SIMD is particularly applicable to common tasks like adjusting the contrast in a digital image or adjusting the volume of digital audio. Most modern CPU designs include SIMD instructions in order to improve the performance of multimedia use.

In computing, SPMD (single program, multiple data) is a technique employed to achieve parallelism; it is a subcategory of MIMD. Tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster. SPMD is the most common style of parallel programming.[1] It is also a prerequisite for research concepts such as active messages and distributed shared memory.

SPMD vs SIMD

In SPMD, multiple autonomous processors simultaneously execute the same program at independent points, rather than in the lockstep that SIMD imposes on different data. With SPMD, tasks can be executed on general purpose CPUs; SIMD requires vector processors to manipulate data streams. Note that the two are not mutually exclusive.

Distributed memory

SPMD usually refers to message passing programming on distributed memory computer architectures. A distributed memory computer consists of a collection of independent computers, called nodes. Each node starts its own program and communicates with other nodes by sending and receiving messages, calling send/receive routines for that purpose. Barrier synchronization may also be implemented by messages. The messages can be sent by a number of communication mechanisms, such as TCP/IP over Ethernet, or specialized high-speed interconnects such as Myrinet and Supercomputer Interconnect. Serial sections of the program are implemented by identical computation on all nodes rather than computing the result on one node and sending it to the others. Nowadays, the programmer is isolated from the details of the message passing by standard interfaces, such as PVM and MPI.

Distributed memory is the programming style used on parallel supercomputers from homegrown Beowulf clusters to the largest clusters on the Teragrid.

Shared memory

On a shared memory machine (a computer with several CPUs that access the same memory space), messages can be sent by depositing their contents in a shared memory area. This is often the most efficient way to program shared memory computers with large number of processors, especially on NUMA machines, where memory is local to processors and accessing memory of another processor takes longer. SPMD on a shared memory machine is usually implemented by standard (heavyweight) processes.

Unlike SPMD, shared memory multiprocessing, also called symmetric multiprocessing or SMP, presents the programmer with a common memory space and the possibility to parallelize execution by having the program take different paths on different processors. The program starts executing on one processor and the execution splits in a parallel region, which is started when parallel directives are encountered. In a parallel region, the processors execute a single program on different data. A typical example is the parallel DO loop, where different processors work on separate parts of the arrays involved in the loop. At the end of the loop, execution is synchronized, only one processor continues, and the others wait. The current standard interface for shared memory multiprocessing is OpenMP. It usually implemented by lightweight processes, called threads.

Combination of levels of parallelism

Current computers allow exploiting of many parallel modes at the same time for maximum combined effect. A distributed memory program using MPI may run on a collection of nodes. Each node may be a shared memory computer and execute in parallel on multiple CPUs using OpenMP. Within each CPU, SIMD vector instructions (usually generated automatically by the compiler) and superscalar instruction execution (usually handled transparently by the CPU itself), such as pipeliningand the use of multiple parallel functional units, are used for maximum single CPU speed. In computing, MIMD (multiple instruction, multiple data) is a technique employed to achieve parallelism. Machines using MIMD have a number of processors that function asynchronously and independently. At any time, different processors may be executing different instructions on different pieces of data. MIMD architectures may be used in a number of application areas such as computer-aided design/computer-aided manufacturing, simulation, modeling, and as communication switches. MIMD machines can be of either shared memory or distributed memory categories. These classifications are based on how MIMD processors access memory. Shared memory machines may be of the bus-based, extended, or hierarchical type. Distributed memory machines may have hypercube or mesh interconnection schemes.

Bus-based

MIMD machines with shared memory have processors which share a common, central memory. In the simplest form, all processors are attached to a bus which connects them to memory.