LECTURENOTES

COMPUTER ORGANIZATION AND ARCHITYECTURE

B.Tech, 4THSemester,CSE

Prepared by:

MRS SANJUKTA URMA

Lecturer in Computer Science & Engineering

Vikash Institute of Technology Bargarh

(Approved by AICTE, New Delhi & Affiliated to BPUT, Odisha)

BarahagudaCanalChowk,Bargarh,Odisha-768040

www.vitbargarh.ac.in

DISCLAIMER

- This document does not claim any originality and cannot be used as a substitute for prescribed textbooks.
- The information presented here is merely a collection by Mrs. SANJUKTA URMA with the inputs of students for their respective teaching assignments as an additional tool for the teaching- learning process.
- Various sources as mentioned at the reference of the document as well as freely available materials from internet were consulted for preparing this document.
- Further, this document is not intended to be used for commercial purpose and the authors are not accountable for any issues, legal or otherwise, arising out of use of this document.
- The author makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and specifically disclaims any implied warranties of merchantability or fitness for a particular purpose.

****

COURSECONTENT

COMPUTER ORGANIZATION AND ARCHITYECTURE

B.Tech, 4THSemester, CSE

➢ Module-I: FUNDAMENTAL BLOCK OF A COMPUTER

Functional blocks of a computer: CPU, memory, input-output subsystems, control Unit, Overview of Computer Architecture and Organization: Fundamentals of computer architecture, Organization of Von Neumann machine, Basic operation concepts, Performance and Historical perspective, Instruction set architecture of a CPU-registers, instruction execution cycle, RTL interpretation of instructions, addressing modes, instruction set.

➢ Module-II: DATA REPRESENTATION

Data representation: signed number representation, fixed and floating point representations, character representation. Computer arithmetic - integer addition and subtraction, ripple carry adder, carry look-ahead adder, etc. multiplication - shift and add, Booth multiplier, carry save multiplier, etc. Division restoring and non-restoring techniques, floating point arithmetic.

➢ Module-III: CPU CONTROL UNIT DESIGN

CPU control unit design: hardwired and micro-programmed design approaches, Memory system design: semiconductor memory technologies, Memory Organization- Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory.

Peripheral devices and their characteristics: I/O subsystems, I/O device interface, I/O transfers-program controlled, Asynchronous data transfer, Modes of Transfer, interrupt driven and DMA,

Privileged and non-privileged instructions, software interrupts and exceptions. Programs and Processes-role of interrupts in process state transitions, I/O device interfaces - SCII, USB

➢

➢ Module-IVREDUCE INSTRUCTION SET COMPUTER

Reduced Instruction Set Computer: CISC Characteristics, RISC Characteristics.

Pipeline and Vector Processing: Pipelining: Basic concepts of pipelining, throughput and speedup, pipeline hazards. Vector Processing, Array Processor.

Multi Processors: Characteristics of Multiprocessors, Interconnection Structures, Inter-processor arbitration, Inter-processor communication and synchronization, Cores, and Hyper-Threading ,Cache Coherence.

➢ Module-V: MEMORY ORGANIZATION

Memory organization: Memory interleaving, concept of hierarchical memory organization, cache memory, cache size vs. block size, mapping functions, replacement algorithms, write policies.

****

REFERENCES

COMPUTER ORGANIZATION AND ARCHITYECTURE

B.Tech,4TH Semester,CSE

Books:

- 1. "Computer Organization and Embedded Systems", 6th Edition by CarlHamacher, McGraw Hill Higher Education.
- 2."Computer Organization and Architecture: Designing for Performance", 10th Edition by William Stallings, Pearson Education.
- 3."Computer Architecture and Organization", 3rd Edition by John P. Hayes, WCB/McGraw-Hill
- 4."Computer Organization and Design: The Hardware/Software Interface", 5th Edition by David A. Patterson and John L. Hennessy, Elsevier.

MODULE-1

Functional blocks of a computer:

A computer consists of five functionally independent main parts:

1. input
2. memory
3. arithmetic and logic
4. output
5. control unit

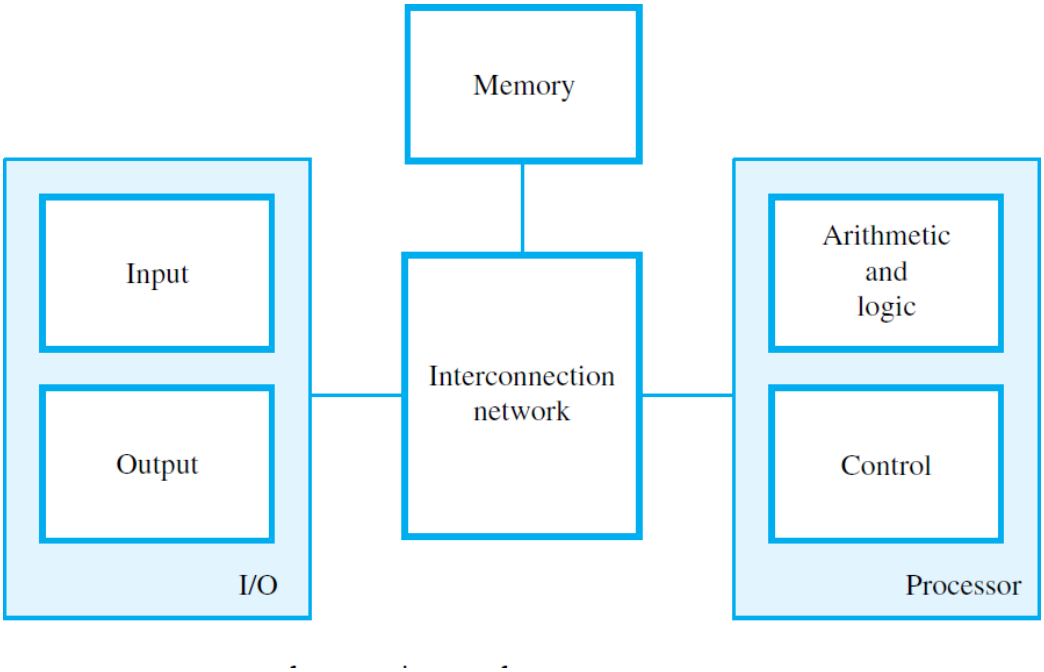


Fig:Basic functional units of a computer.

1. The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines.
2. The information received is stored in the computer's memory, either for later use or to be processed immediately by the arithmetic and logic unit.
3. The processing steps are specified by a program that is also stored in the memory.
4. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit.
5. An interconnection network provides the means for the functional units to exchange information and coordinate their actions.

A program is a list of instructions which performs a task. Programs are stored in the memory.The processor fetches the program instructions from the memory, one after another, and perform the desired operations. The computer is controlled by the stored program, except for possible external interruption by an operator or by I/O devices. The instructions and data handled by a computer is encoded as a string of binary bits.

- **Input Unit**: Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor. Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Similarly, cameras can be used to capture video input.

E.g.: touchpad, mouse, joystick

- **Memory Unit:**

The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

Primary Memory: Primary memory, also called main memory, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed. The memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. The memory is organized so that one word can be stored or retrieved in one basic operation. The number of bits in each word is referred to as the word length of the computer, typically 16, 32, or 64 bits. To provide easy access to any word in the memory, a distinct address is associated with each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations. A particular word is accessed by specifying its address and issuing a control command to the memory that starts the storage or retrieval process. Instructions and data can be written into or read from the memory under the control of the processor. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a random-access memory (RAM). The time required to access one word is called the memory access time. This time is independent of the location of the word being accessed.

Cache Memory: As an adjunct to the main memory, a smaller, faster RAM unit, called a cache, is used to hold sections of a program that are currently being executed, along with any associated data. The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip. The purpose of the cache is to facilitate high instruction execution rates. At the start of program execution, the cache is empty. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache. When the execution of an

instruction requires data located in the main memory, the data are fetched and copies are also placed in the cache.

Secondary Storage: Although primary memory is essential, it tends to be expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent secondary storage is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently. Access times for secondary storage are longer than for primary memory. A wide selection of secondary storage devices is available, including magnetic disks, optical disks (DVD and CD), and flash memory devices.

Arithmetic and Logic Unit: Most computer operations are executed in the arithmetic and logic unit (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication, division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU. For example, if two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use. When operands are brought into the processor, they are stored in high-speed storage elements called registers. Each register can store one word of data.

Output Unit: The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. A familiar example of such a device is a printer. Some units, such as graphic displays, provide both an output function, showing text and graphics, and an input function, through touch screen capability.

Control Unit: The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the responsibility of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states. Control circuits are responsible for generating the timing signals that govern the transfers and determine when a given action is to take place. In practice, much of the control circuitry is physically distributed throughout the computer. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

Basic Operational Concepts

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

Load R2, LOC

This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2.The original contents of location LOC are preserved, whereas those of register R2 are overwritten. Execution of this instruction requires several steps.

1. . First, the instruction is fetched from the memory into the processor.
2. Next, the operation to be performed is determined by the control unit.
3. The operand at LOC is then fetched from the memory into the processor.
4. Finally, the operand is stored in register R2.

Let us consider another example

Add R4, R2, R3

This instruction adds the contents of registers R2 and R3, then places their sum into register R4. The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum. After completing the desired operations, the results are in processor registers. They can be transferred to the memory using instructions such as

Store R4, LOC

This instruction copies the operand in register R4 to memory location LOC. The original contents of location LOC are overwritten, but those of R4 are preserved. For Load and Store instructions, transfers between the memory and the processor are initiated by sending the address of the desired memory location and asserting the appropriate control signals. The data are then transferred to or from the memory. Figure 1.1 shows how the memory and the processor can be connected.

In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes. The instruction register (IR) holds the instruction that is currently being executed. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction.

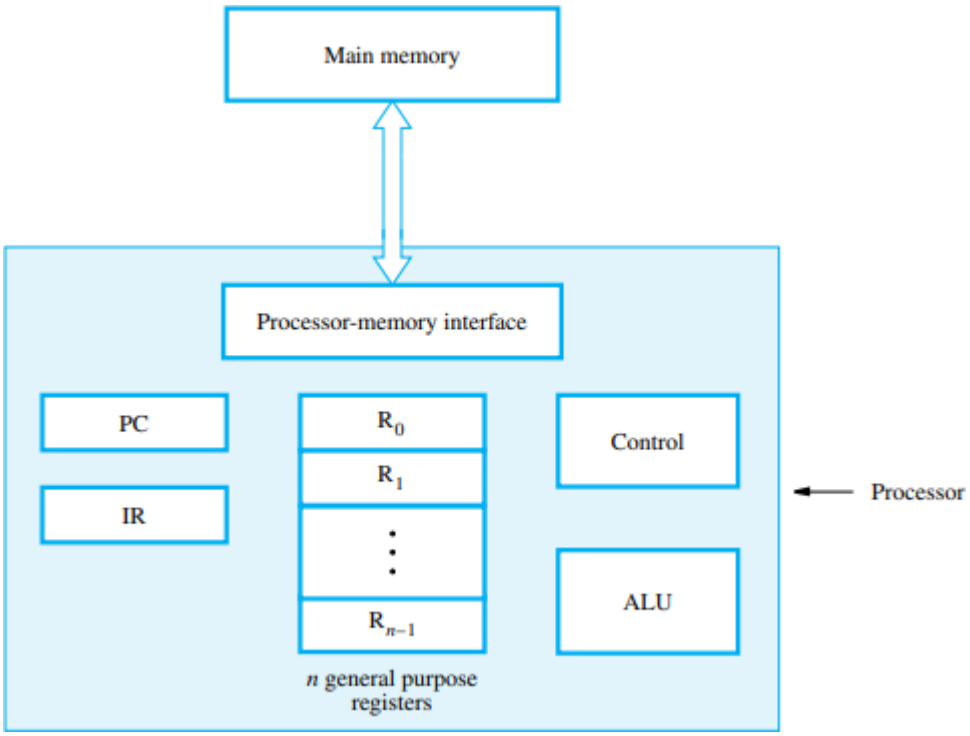


Fig 1.1: ***Connection between the processor and the main memory***

PC: The program counter (PC) is another specialized register. Contains the memory address of the next instruction to be fetched and executed. During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed.

General purpose Registers: There are also general-purpose registers R0 through Rn−1, often called processor registers. They serve a variety of functions, including holding operands that have been loaded from the memory for processing.

Processor Memory Interface: The processor-memory interface is a circuit which manages the transfer of data between the main memory and the processor. If a word is to be read from the memory, the interface sends the address of that word to the memory along with a Read control signal. The interface waits for the word to be retrieved, then transfers it to the appropriate processor register.If a word is to be written into memory, the interface transfers both the address and the word to the memory along with a Write control signal.

Following are typical operating steps:

1) A program must be in the main memory in order for it to be executed. It is often transferred there from secondary storage
2) Execution of the program begins when the PC is set to point to the first instruction of the program.
3) The contents of the PC are transferred to the memory along with a Read control signal. When the

addressed word (in this case, the first instruction of the program) has been fetched from the

memory it is loaded into register IR. At this point, the instruction is ready to be decoded and executed.

4) If an operand that resides in the memory is required for an instruction, it is fetched by sending its address to the memory and initiating a Read operation. When the operand has been fetched from the memory, it is transferred to a processor register "**R**".
5) After operands have been fetched in this way, the ALU can perform a desired arithmetic operation, such as Add, on the values in processor registers. The result is sent to a processor register.
6) If the result is to be written into the memory with a Store instruction, it is transferred from the processor register to the memory, along with the address of the location where the result is to be stored, and then a Write operation is initiated.

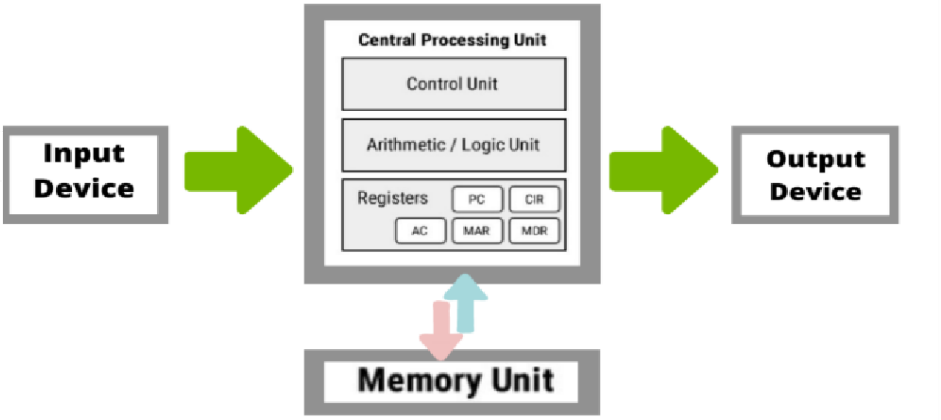
At some point during the execution of each instruction, the contents of the PC are incremented so that the PC points to the next instruction to be executed. Thus, as soon as the execution of the current instruction is completed, the processor is ready to fetch a new instruction.

Normal execution of a program may be preempted if some device requires urgent service. For example, a monitoring device in a computer-controlled industrial process may detect a dangerous condition. In order to respond immediately, execution of the current program must be suspended. To cause this, the device raises an interrupt signal, which is a request for service by the processor. The processor provides the requested service by executing a program called an interrupt-service routine. When the interrupt-service routine is completed, the state of the processor is restored from the memory so that the interrupted program may continue.

Von Neumann Architecture:

The Von-Neumann Architecture or Von-Neumann model is also known as **"Princeton Architecture"**. This architecture was published by the Mathematician **John Von Neumann** in **1945**.

Von Neumann architecture is the design upon which many general purpose computers are based. This architecture implemented the stored program concept in which the data and instructions are stored in the same memory. This architecture consists of a CPU(ALU, Registers, Control Unit), Memory and I/O unit.


Central Processing Unit
Control Unit
Arithmetic / Logic Unit
Registers
PC
CIR
AC
MAR
MDR
Input Device
Output Device
Memory Unit

Following are the components of Von Neumann Architecture:

1. CPU(Central processing unit)
    - CU(Control Unit)
    - ALU(Arithmetic and logic unit)
    - Registers
        - ✓ PC(Program Counter)
        - ✓ IR(Instruction Register)
        - ✓ AC(Accumulator)
        - ✓ MAR(Memory Address Register)
        - ✓ MDR(Memory Data Register)
2. BUSES
3. I/o Devices
4. Memory Unit

1. **CPU:** CPU acts as the brain of the computer and is responsible for the execution of instructions.
2. **Control Unit:** A control unit (CU) handles all processor control signals. It directs all input that is used to connect computer components and transfer data between them. There are three types of BUSES

a) **Data Bus:** It carries data among the memory unit, the I/O devices, and the processor.
b) **Address Bus:** It carries the address of data (not the actual data) between memory and processor.
c) **Control Bus:** It carries control commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer.

3. **I/o Devices**: Program or data is read into main memory from the *input device* or secondary storage under the control of CPU input instruction. *Output devices* are used to output the information from a computer. If some results are evaluated by CPU and it is stored in the computer, then with the help of output devices, we can present them to the user.

4. **Memory:** A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the storage. The memory stores binary information in groups of bits called words. The internal structure of a memory unit is specified by the number of words it contains and the number of bits in each word (2^Mx N, eg: 128KB).

There are two types of Primary Memory:

1) RAM: VOLATILE MEMORY or temporary Memory(to store the program in execution)

2) ROM: NON-VOLATILE MEMORY or permanent Memory(to store the booting program)

a) and output flow, fetches code for instructions, and controls how data moves around the system.

b) **Arithmetic and Logic Unit (ALU) :**
The arithmetic logic unit is that part of the CPU that handles all the calculations the CPU may need, e.g. Addition, Subtraction, Comparisons. It performs Logical Operations, Bit Shifting Operations, and Arithmetic operations.

c) **Registers**: A processor based on von Neumann architecture has five special registers which it uses for processing:

Instruction set architecture of a CPU: Architecture of 8086:

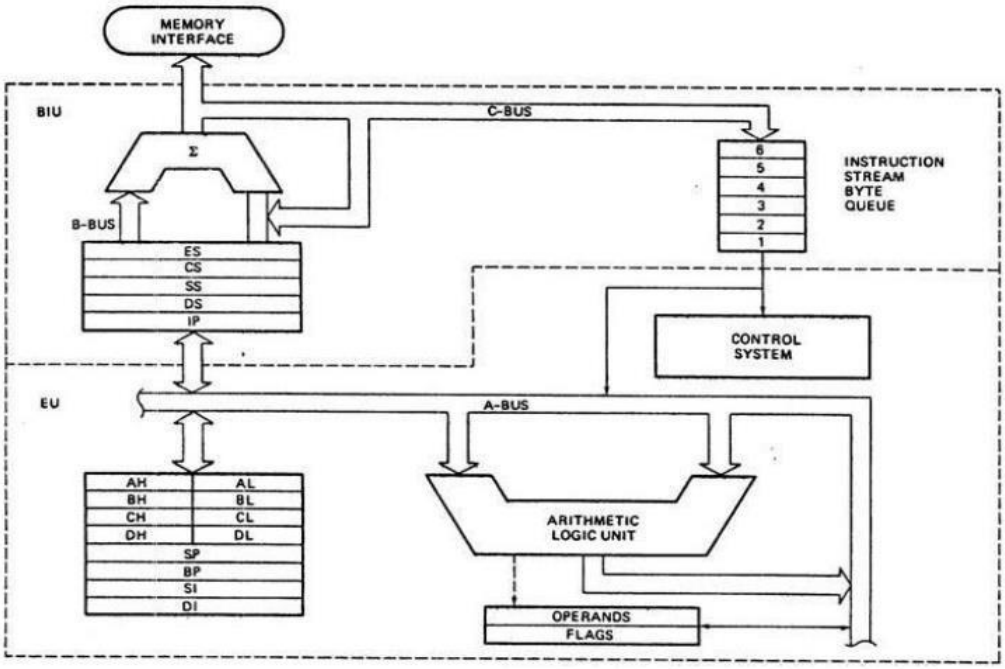


Fig: 8086 internal Architecture

The architecture of 8086 supports a 16-bit ALU , a set of 16-bit registers, and provides segmented memory addressing capability, fetched instruction queue for overlapped fetching and execution.

- Architecture of 8086 is pipeline type of architecture.
- The architecture of 8086 is divided into two functional parts i.e.,

i. **Execution unit (EU)**

ii. **Bus interface unit (BIU)**

These two units work asynchronously.

- Functional division of architecture speeds up the processing, since BIU and EU operate parallelly and independently i.e., EU executes the instructions and BIU fetches another instruction from the memory simultaneously.
- As the whole architecture is divided into two independent functional parts and both the subsystem's operations can be overlapped, hence the architecture is PIPELINING type of architecture.

EXECUTION UNIT

- The execution unit informs the BIU of the processor regarding from where to fetch the instructions from and then executes these instructions.
- The execution unit consists of the following:
    - General purpose registers
    - Stack pointer
    - Base pointer
    - Index registers
    - ALU
    - Flag register(FLAGS/ PSW)
    - Instruction decoder
    - Timing and control unit

Functions of EU

- Tells BIU regarding from where to fetch instructions or to read data.
- Receives opcode of an instruction from the queue.
- decodes the instructions.
- Executes the instruction.

Functions of various parts of EU

- Control circuitry: Directs internal operations.
- Instruction Decoder: Translates instructions fetched from memory into series of actions.
- ALU: Performs arithmetic and logical operations.
- FLAGS: Reflects the status of program.
- General purpose registers: Used to store Temporary data.
- Index and Pointer registers: Specifies/ informs about offset of operand

BUS INTERFACE UNIT

- The BIU handles transfer of data and address between the processor and memory/ I/O devices by computing address (Physical/ Effective address) and send the computed address to memory / I/O and fetches instruction codes then stores them in FIFO register set called Queue register.

- The BIU consists of the following:
    - ❖ Segment Registers
    - ❖ Instruction pointer
    - ❖ 6-Byte instruction Queue Register

Functions of BIU

- Handles transfer of data and address between processor and memory / I/O devices.
- Compute physical address and send it to memory interfaces.
- Fetches instruction codes and stores it in Queue
- Reads/Writes data from/to memory/ I/O devices
- Functions of various parts of BIU **Segment registers :** Used to hold the starting address of the segmer registers.
- **Queue register:** Used to store pre fetched instructions and inputs it to EU.
- **Instruction Pointer:** Used to point to the next instruction to be executed by EU.
- While the EU is decoding an instruction or executing an instruction which does not require use of the buses, the BIU fetches up to six instruction bytes that will be following the present instruction from memory and stores them in the queue register simultaneously.

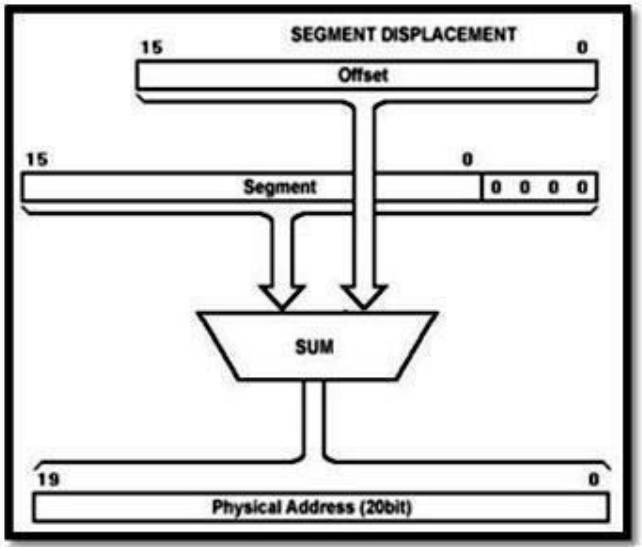
Logical and Physical Address

- Addresses within a segment can range from address 00000h to address 0FFFFh. This corresponds to the 64K-bytelength of the segment. An address within a segment is called an offset or logical address.
- A logical address gives the displacement from the base address of the segment to the desired location within it, as opposed to its "real" address, which maps directly anywhere into the 1 MByte memory space. This "real" address is called the physical address.

Difference between the physical and the logical address:

The physical address is 20 bits long and corresponds to the actual binary code output by the BIU on the address bus lines. The logical address is an offset from location 0 of a given segment

Segment address ⟶ 1005H
Offset address ⟶ 5555H
Segment address ⟶ 1005H ⟶ 0001 0000 0000 0101
Shifted by 4 bit positions ⟶ 0001 0000 0000 0101 0000
+
Offset address ⟶ 0101 0101 0101 0101
Physical address ⟶ 0001 0101 0101 1010 0101
1 5 5 A 5



- .

physical address while addressing memory. The segment address value is to be taken from an appropriate gment register depending upon whether code, data or stack are to be accessed, while the offset may be the ontent of IP, BX, SI, DI, SP, BP or an immediate 16-bit value, depending upon the addressing mode.

In case of 8085, once the opcode is fetched and decoded, the external bus remains free for some time hile the processor internally executes the instruction. This time slot is utilised in 8086 to achieve the over pped fetch and execution cycles. While the fetched instruction is executed internally, the external bus is sed to fetch the machine code of the next instruction and arrange it in a queue known as predecoded in ruction byte queue. It is a 6 bytes long, first-in first-out structure. The instructions from the queue are taker r decoding sequentially. Once a byte is decoded, the queue is rearranged by pushing it out and the queue atus is checked for the possibility of the next opcode fetch cycle. While the opcode is fetched by the Bus terface Unit (BIU), the Execution Unit (EU) executes the previously decoded instruction concurrently he BIU along with the Execution Unit (EU) thus forms a pipeline. The bus interface unit, thus manages the mplete interface of execution unit with memory and I/O devices, of course, under the control of the timing nd control unit.

The execution unit contains the register set of 8086 except segment registers and IP. It has a 16-bit ALU, able to perform arithmetic and logic operations. The 16-bit flag register reflects the results of execution by the ALU. The decoding unit decodes the opcode bytes issued from the instruction byte queue. The timing and control unit derives the necessary control signals to execute the instruction opcode received from the queue, depending upon the information made available by the decoding circuit. The execution unit may pass the results to the bus interface unit for storing them in memory.

Flag register of 8086

There are total 9 flags in 8086 and the flag register is divided into two types:

(a) Status Flags – There are 6 flags in 8086 microprocessor which become set(1) or reset(0) depending upon condition after either 8-bit or 16-bit operation. These flags are conditional/status flags.

The 6 status flags are:

(b) Sign Flag (S): This flag is set when the result of any computation is negative.
(c) Zero Flag (Z): This flag is set when the result of any computation or comparison performed is zero.
(d) Auxiliary Cary Flag (AC): This flag is set when there is acarry from the lower nibble.
(e) Parity Flag (P): This flag is set when the lower byte of the result contains even number of 1's .
(f) Overflow Flag: This flag will be set (1) if the result of a signed operation is too large to fit in the number of bits available to represent it, otherwise reset (0).(eg:50+32= 82)
(g) Carry Flag (CY)): This flag is set when there is a carry out of the MSB in case of addition or a borrow in case of subtraction.

Control Flags – The control flags enable or disable certain operations of the microprocessor.There are 3 control flags in 8086 microprocessor and these are:

Directional Flag (D) – This flag is specifically used by string manipulation instructions string instructions. If this flag is 0,the string is processed beginning from the lowest address to the highest address. If this flag is 1,the string is processed beginning from the highest address to the lowest address.

Interrupt Flag: If interrupt flag is set (1), the microprocessor will recognize interrupt requests from the peripherals.

If interrupt flag is reset (0), the microprocessor will not recognize any interrupt requests and will ignore them.

Trap Flag (T) –Setting trap flag puts the microprocessor into single step mode for debugging.

INSTRUCTION SET ARCHITECTURE OF CPU

Register Transfer Language:

- A digital computer system exhibits an interconnection of digital modules such as registers, decoders, arithmetic elements, and Control logic. These digital modules are interconnected with some common data and control paths to form a complete digital system. Digital modules are best defined by the registers and the operations that are performed on the data stored in them.
- The **operations performed on the data stored in registers are called Micro-operations**. A microoperation is an elementary operation performed on the information stored in oneor more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examplesof microoperations are shift, count, clear, and load.
- The **Register Transfer Language** is the **symbolic representation of notations used to specify the sequence of micro-operations**.

In a computer system, **data transfer takes place between processor registers and memory and between processor registers and input-output systems**. These data transfer can be represented by standard notations given below:

- Notations R0, R1, R2..., and so on represent processor registers.
- The addresses of memory locations are represented by names such as LOC, PLACE, MEM, etc.
- Input-output registers are represented by names such as DATA IN, DATA OUT and so on.
- The content of register or memory location is denoted by placing square brackets around the name of the register or memory location.

Register Transfer:

Computer registers are denoted by capital letters (sometimes followed by numerals) to denote the function of the register. The register that holds an address for the memory unit is usually called a **memory address register** and is denoted by **MAR**. Other registers are PC (for program counter), IR (for instruction register, and R1 (for processor register). An n-bit register is sequence of n-flipflops numbered from 0 through n-1, starting from 0 in the rightmost position and increasing the numbers toward the left.

The most common way to represent a register is by a rectangular box with the name of the register inside, as shown in the figure below. The individual bits can be distinguished as shown in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H(for high byte). The name of the 16-bit register is PC. The symbol PC (0-7) or PC (L) refers to the low-order byte and PC(8-15) or PC(H) to the high-order byte.

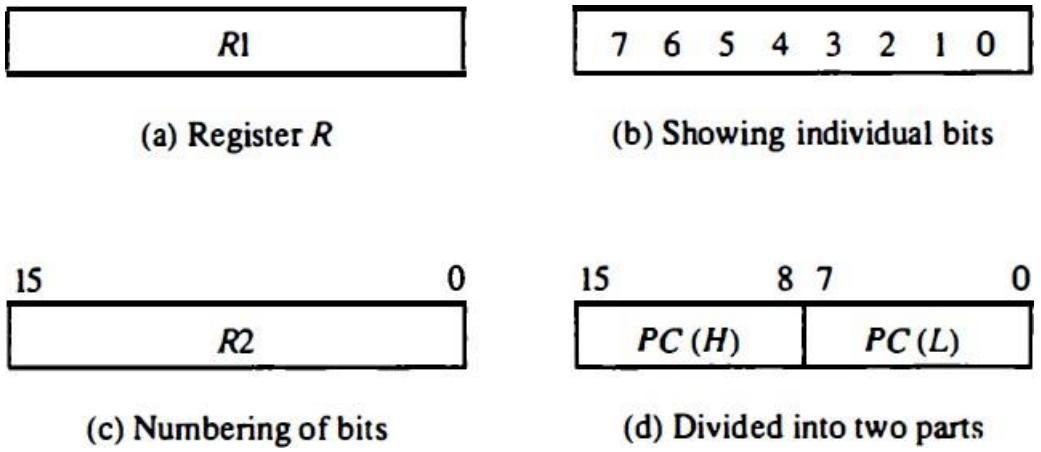


Fig: Block diagram of registers

Information transfer from one register to another is designated in symbolic form by means of a replacement operator as shown below, which denotes a transfer of the contents of register R1 into register R2.Contents of R2 are replaced by the contents of R1.By definition, the content of the source register R1 does not change after the transfer. Register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register.

R2 <--R1

Sometimes, we may want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement

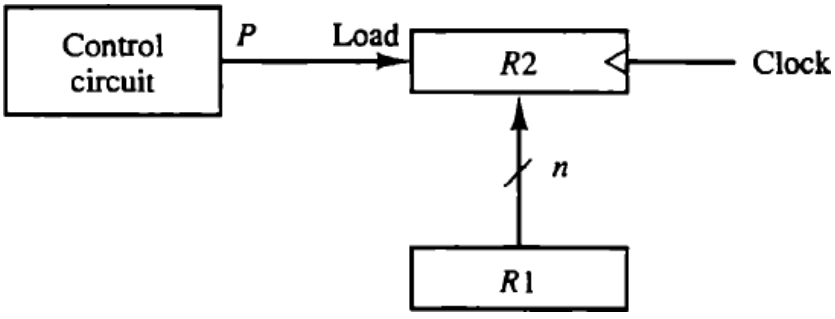
If (P = 1) then (R2 <--R1)

where P is a control signal generated in the control section. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows

P: R2 <--R1

The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if P = 1.

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure below shows the block diagram that depicts the transfer from R1 to R2. The n outputs of register R1 are connected to the n inputs of register R2. The letter n will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register R2 has a load input that inactivated by the control variable P. It is assumed that the control variable is synchronized with the same clock as the one applied to the register.



In the timing diagram below, P is activated in the control section by the rising edgeof a clock pulse at time t . The next positive transition of the clock at time t + 1finds the load input active and the data inputs of R2 are then loaded into theregister in parallel. P may go back to 0 at time t + 1; otherwise, the transferwill occur with every clock pulse transition while P remains active.

Note:Even though the control condition such as P becomes active just after time t,the actual transfer does not occur until the register is triggered by the nextpositive transition of the clock at time t + 1 .

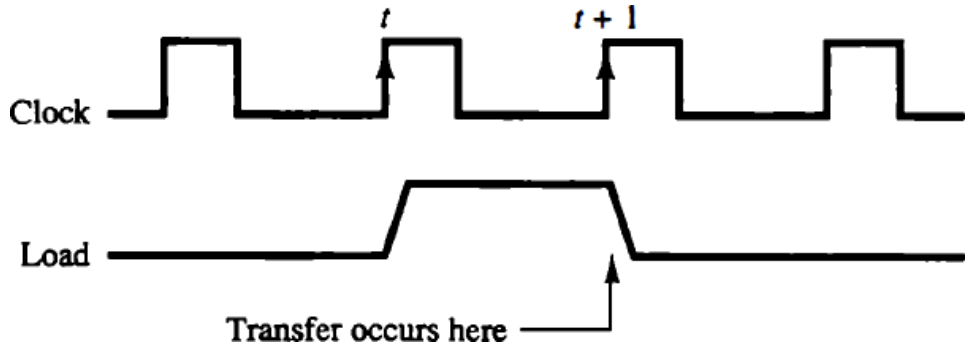


Fig: Timing Diagram

Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time.

The statement

T: R2 <- R1, R1 <- R2

It denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T = 1.

The basic symbols of the register transfer notation are given below:

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*MAR*, *R2*
Parentheses ( )	Denotes a part of a register	*R2*(0–7), *R2*(*L*)
Arrow ←	Denotes transfer of information	*R2* ← *R1*
Comma ,	Separates two microoperations	*R2* ← *R1*, *R1* ← *R2*

Fig: Basic symbols of register Transfer

Memory Transfer:

The transfer of information from a memory word to the outside environment is called a read operation. The transfer of new information to be stored into the memory is called a write operation. A memory word will be symbolized by the letter M.

The particular memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter M.

Memory Read: Consider a memory unit that receives the address from a register, called the address register, symbolized by AR. The data are transferred to another register, called the data register, symbolized by DR. The read operation can be stated as follows:

Read: DR <- M [AR]

This causes a transfer of information into DR from the memory word M selected by the address in AR.

Memory Write: The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data is in register R1 and the address in AR. The write operation can be stated symbolically as follows:

Write: M[AR] ← R1

This causes the transfer of information from R1 into the memory word M selected by the address in AR.

Instruction cycle:

In the basic computer each instruction cycle consists of the following phases:

1. Fetch an instruction from memory.

2. Decode the instruction.

3. Read the effective address from memory if the instruction has an indirect address.

4. Execute the instruction.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

FETCH AND DECODE: Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal To. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T0, T1, T2, and so on.

The Micro-operations for the fetch and decode phases can be specified by the following register transfer statements:

T_0: AR ← PC

The address from PC to AR during the clock transition associated with timing signal T0.

T_1: IR ←M[AR], PC ←PC + 1

The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T1. At the same time, PC is incremented by one to prepare it for the address of the next instruction in the program

T_2: D0, …, D7 ← Decode IR(12-14),AR ← IR(0-11), I← IR(l5)

At time T2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR.

Decoding: The timing signal that is active after the decoding is T3. During time T3, the control unit determines the type of instruction that was just read from memory. Decoder output D7, is equal to 1 if the operation code is equal to binary 111. If D7 = 1, the instruction must be a register-reference or input-output type. If D7 = 0, the operation code must be one of the other seven values 000 through 110,specifying a memory-reference instruction. Control then inspects the value of the first bit of the

instruction, which is now available in flip-flop I. If D7 = 0 and I = 1, we have a

Memory reference instruction with an indirect address. The micro operation for the indirect address condition can be symbolized by the register transfer statement:

AR ← M [AR]

$D_7' IT_3$: $AR \leftarrow M[AR]$
$D_7' I'T_3$: Nothing
$D_7 I'T_3$: Execute a register-reference instruction
$D_7 IT_3$: Execute an input–output instruction

When a **memory-reference instruction** with I = 0 is encountered, it is not necessary to do anything since the effective address is already in AR. However, the sequence counter SC must be incremented so that the execution of the memory-reference instruction can be continued with timing variable T4.After the instruction is executed ,SC is cleared to 0 and control returns to the fetch phase with T0 = 1 .

Register-reference instructions are recognized by the control when 07 = 1 and i= 0.The 12 bits available in IR(0-11) are transferred to AR during time T2.These instructions are executed with the clock transition associated with timing variable T3.The execution of a register-reference instruction is completed at time T3.The sequence counter SC is cleared to 0 and the control goes back to fetch the next instruction with timing signal T0.

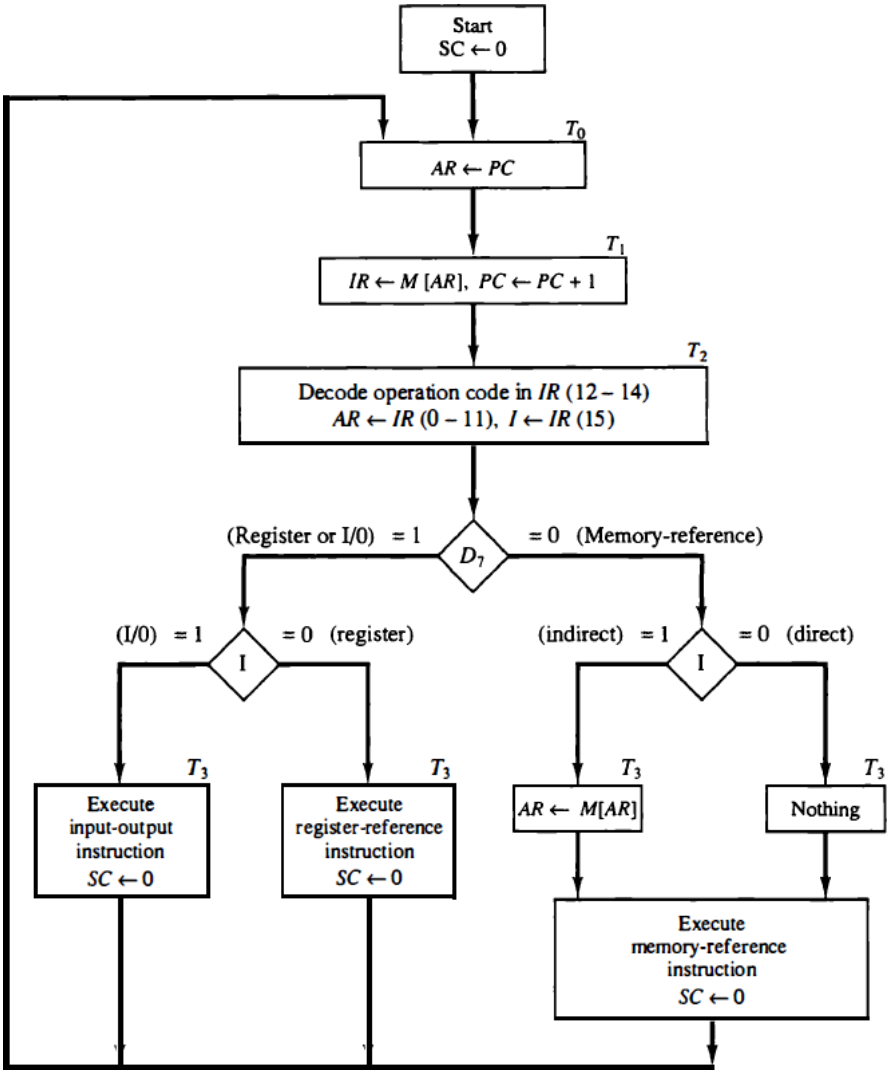


Fig:Flowchart for instruction cycle

Addressing Modes

ADDRESSING MODES OF 8086:

Addressing modes is the manner in which operands are given in an instruction. The addressing modes of 8086 are as follows:

1) **IMMEDIATE ADDRESSING MODE:** In this mode the operand is specified in the instruction itself. Instructions are longer but the operands are easily identified.
   E.g.: MOV CL, 12H ; **Moves 12 immediately into CL register**
   MOV BX, 1234H ; **Moves 1234 immediately into BX register**
2) **REGISTER ADDRESSING MODE**: In this mode operands are specified using registers. Instructions are shorter but operands cant be identified by looking at the instruction.
   E.g.: MOV AX,BX
   ADD BX,CX
3) **DIRECT ADDRESSING MODE:** In this mode address of the operand is directly specified in the instruction.
   Eg:MOV CL, [2000H] ; CL Register gets data from memory location 2000H
   CL ←[2000H]
   MOV [3000H], DL ; Memory location 3000H gets data from DL Register
   [3000H] ← DL
4) **INDIRECT ADDRESSING MODE:** In Indirect Addressing modes, address is given by a register. The register can be incremented in a loop to access a series of locations. There are various sub-types of Indirect addressing mode.
   REGISTER INDIRECT ADDRESSING MODE
   This is the most basic form of indirect addressing mode. Here address is simply given by a register.
   E.g.: MOV CL, [BX] ; CL gets data from a memory location pointed by BX
   CL ← [BX]. If BX = 2000H, CL ← [2000H]
   E.g.: MOV [BX], CL ; CL is stored at a memory location pointed by BX
   [BX] ←CL. If BX = 2000H, [2000H] ← CL.

REGISTER RELATIVE ADDRESSING MODE : Here address is given by a register plus a numeric displacement.

E.g.: MOV CL, [BX + 03H] ; CL gets data from a location BX + 03H

CL ← [BX+03H]. If BX = 2000H, then CL ← [2003H]

E.g.: MOV [BX + 03H], CL ; CL is stored at location BX + 03H

[BX+03H] ←CL. If BX = 2000H, then [2003H] ← CL.

BASE INDEXED ADDRESSING MODE Here address is given by a sum of two registers. This is typically useful in accessing an array or a look up table. One register acts as the base of the array holding its starting address and the other acts as an index indicating the element to be accessed.

E.g.: MOV CL, [BX + SI] ; CL gets data from a location BX + SI ; CL ← [BX+SI]. ;

If BX = 2000H, SI = 1000H, then CL ç [3000H] E.g.: MOV [BX + SI], CL ; CL is stored at location BX + SI ; [BX+SI] ← CL. ; If BX = 2000H, SI = 1000H, then [3000H] ← CL.

BASE RELATIVE PLUS INDEX ADDRESSING MODE Here address is given by a sum of base register plus index register plus a numeric displacement.

E.g.: MOV CL, [BX+SI+03H] ; CL gets data from a location BX + SI + 03H ;

CL ←[BX+SI+03H]. ;

If BX = 2000H, SI = 1000H, then CL ← [3003H]

E.g.: MOV [BX+SI+03H], CL ; CL is stored at location BX + SI + 03H ;

[BX+SI+03H] ←CL. ;

If BX = 2000H, SI = 1000H, then [3003H] ← CL.

IMPLIED ADDRESSING MODE: In this addressing mode, the operand is not specified at all, as it is an implied operand. Some instructions operate only on a particular register. In such cases, specifying the register becomes unnecessary as it becomes implied.

E.g.: STC ; Sets the Carry flag.; This instruction can only operate on the Carry Flag.

E.g.: CMC ; Complements the Carry flag.; This instruction can only operate on the Carry Flag

Instruction Set:

Most computer instructions can be classified into three categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

1) Data Transfer Instructions: Data transfer instructions move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves. Table below gives a list of eight data transfer instructions used in many computers.

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Accompanying each instruction is a mnemonic symbol. Different computers use different mnemonics for the same instruction name.

The **load instruction** has been used mostly to designate a transfer from memory to a processor register, usually an accumulator. The **store instruction** designates a transfer from a processor register into memory. The **move instruction** has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words. The **exchange instruction** swaps information between two registers or a register and a memory word. The **input and output instructions** transfer data among processor registers and input or output terminals. The **push and pop** instructions transfer data between processor registers and a memory stack.

2) Data Manipulation Instructions: The data manipulation instructions in a typical computer are usually divided into three basic types:

- ✓ Arithmetic instructions

- ✓ Logical and bit manipulation instructions

- ✓ Shift instructions

Arithmetic instructions: The four basic arithmetic operations are addition, subtraction, multiplication, and division. Most computers provide instructions for all four operations. Some small computers have only addition and possibly subtraction instructions.

A list of typical arithmetic instructions is given in Table given below:

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

The **increment instruction** adds 1 to the value stored in a register or memory word. The **decrement instruction** subtracts 1 from a value stored in a register or memory word. The add, subtract, multiply, and divide instructions may be available for different types of data. The data type assumed ·to be in processor registers during the execution of these arithmetic operations is included in the definition of the operation code. An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data.

The mnemonics for three add instructions that specify different data types are shown below:

ADDI Add two binary integer numbers

ADDF Add two floating-point numbers

ADDD Add two decimal numbers in BCD

The instruction "**add with carry**" performs the addition on two operands plus the value of the carry from the previous computation. Similarly, the "**subtract with borrow**" instruction subtracts two words and a borrow which may have resulted from a previous subtract operation. The **negate instruction** forms the 2' s complement of number, effectively reversing the sign of an integer when represented in the signed- 2's complement form.

Logical and Bit Manipulation Instructions: Logical instructions **perform binary operations on strings of bits** stored in registers. They are useful for manipulating individual bits or a group of bits that represent binary-coded information. The **logical instructions consider each bit of the operand separately** and treat it as a Boolean variable. By proper application of the logical instructions, it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in registers or memory words.

Some logical and bit manipulation instructions are shown in the figure below:

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

The clear instruction causes the specified operand to be replaced by D's.The complement instruction produces the 1's complement by inverting all the bits of the operand. The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands. Although they perform Boolean operations, when used in computer instructions, the logical instructions should be considered as performing bit manipulation operations. There are three-bit manipulation operations possible: a selected bit can be cleared to 0, or can be set to 1, or can be complemented. The three logical instructions are usually applied to do just that.

Shift Instructions: Shifts are operations in which the bits of a word are moved to the left or right. Shift instructions may specify logical shifts, arithmetic shifts, or rotate-type operations. In either case the shift maybe to the right or to the left. Table below lists four types of shift instructions:

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

The **logical shift** inserts 0to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left.

The **arithmetic shift-right** instruction must preserve the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. This is a shift-right operation with the end bit remaining the same. The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-left instruction.

The rotate instructions produce a circular shift. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end. The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated. Thus, a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shifts the entire register to the left.

Program Control Instructions: Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer a program control type of instruction, when executed, may change the address value in the program counter and cause the flow of control to be altered. In other words, program control instructions specify conditions for altering the content of the program counter.

Some program control instructions are listed in Table below:

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

Branch and jump instructions are used interchangeably to mean the same thing, but sometimes they are used to denote different addressing modes. Branch instruction is written as **BR ADR**, where ADR is a symbolic name for an address. Branch and jump instructions may be conditional or unconditional. An unconditional branch instruction causes a branch to the specified address without any conditions. The conditional branch instruction specifies a condition such as branch if positive or branch if zero. If the condition is met, the program counter is loaded with the branch address and the next instruction is taken. from this address. If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.

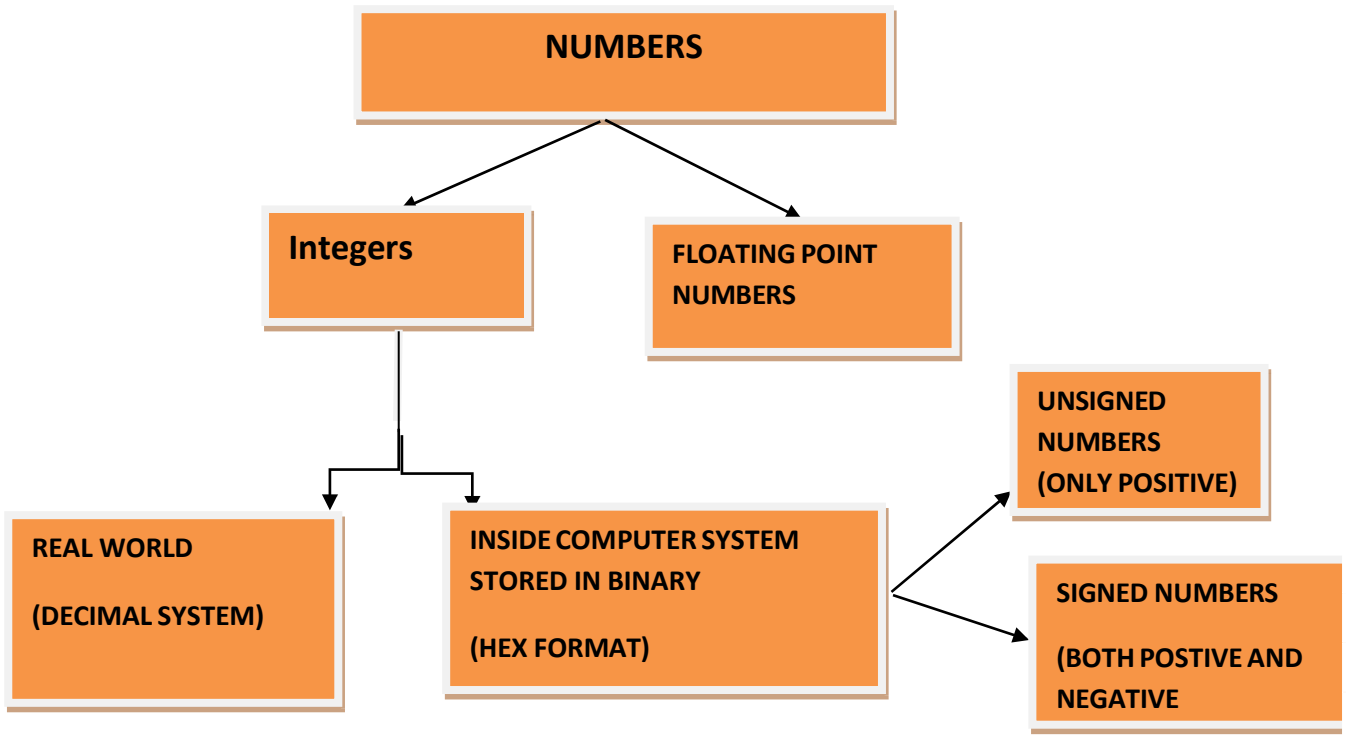
The **skip instruction** does not need an address field and is therefore a zero-address instruction. A conditional skip instruction will skip the next instruction if the condition is met. If the condition is not met, control proceeds with the next instruction in sequence.

The **call and return** instructions are used in conjunction with subroutines.

The **compare instruction** performs a subtraction between two operands, but the result of the operation is not retained. However, certain status bit conditions are set as a result of the operation. Similarly, the **test instruction** performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands. (***Note:*** The compare and test instructions do not change the program sequence directly. They are listed in Table because of their application in setting conditions for subsequent conditional branch instructions)

- **Program counter (PC)** holds the memory address of the next instruction to be fetched from primary storage.
- The **Memory Address Register (MAR)** holds the address of the current instruction that is to be fetched from memory, or the address in memory to which data is to be transferred.
- The **Memory Data Register (MDR)** holds the contents found at the address held in MAR or data which is to be transferred to the primary storage.
- The **Current Instruction Register (CIR)** holds the instruction that is currently being decoded and executed.
- The **Accumulator** is a special purpose Register and is used by the ALU to hold the data being processed and the results of calculations.

NUMBER REPRESENTATION:



UNSIGNED INTEGERS

These are binary numbers that are always assumed to be positive. Here all available bits of the number are used to represent the magnitude of the number. No bits are used to indicate its sign, hence they are called unsigned numbers.

E.g.: Roll Numbers, Memory addresses etc

SIGNED INTEGERS

These are binary numbers that can be either positive or negative. The MSB of the number indicates whether it is positive or negative. If **MSB is 0 then the number is Positive**. If **MSB is 1 then the number is Negative**. Negative numbers are always stored in 2's complement form.

Three systems are used for representing such numbers:

- **Signed magnitude**
- **1's-complement**
- **2's-complement**

In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers.Positive values have identical representations in all systems, but negative values have different representations.

In the **signed magnitude system**, negative values are represented by changing the most significant bit from 0 to 1.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.The same operation, bit complementing, is done to convert a negative number to the corresponding positive value.

B	Values represented		
$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	− 0	− 7	− 8
1 0 0 1	− 1	− 6	− 7
1 0 1 0	− 2	− 5	− 6
1 0 1 1	− 3	− 4	− 5
1 1 0 0	− 4	− 3	− 4
1 1 0 1	− 5	− 2	− 3
1 1 1 0	− 6	− 1	− 2
1 1 1 1	− 7	− 0	− 1

Fig: Binary signed number Representations

Two's complement gives a unique representation for zero. Any other system gives a separate representation for +0 and for -0. This is absurd. In two's complement system, -(x) is stored as two's complement of (x). Applying the same rule for 0, -(0) should be stored as two's complement of 0. 0 is stored as 000. So –(0) should be stored as two's complement of 000, which again is 000. Hence two's complement gives a unique representation for 0.**It produces an additional number on the negative side.** As two's complement system produces a unique combination for 0, it has a spare combination '1000' in the above case, and can be used to represent –(8).

3 Bit Integer	
2^3 = 8 therefore 8 combinations	
Unsigned	Signed
0 ... 7	-4 ... -1 0 1 ... 3

4 Bit Integer	
2^4 = 16 therefore 16 combinations	
Unsigned	Signed
0 ... 15	-8 ... -1 0 1 ... 7

Fixed and Floating point Representations:

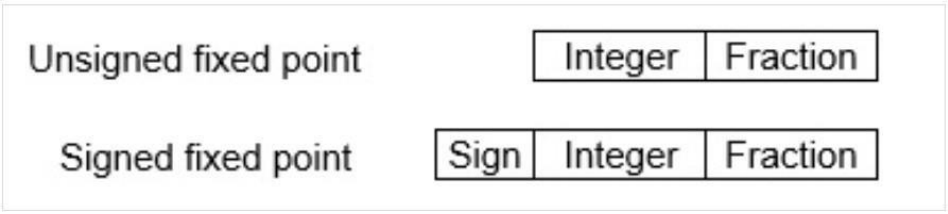
There are two major approaches to store real numbers (i.e., numbers with fractional component) in modern computing. These are

(i) Fixed Point Notation and

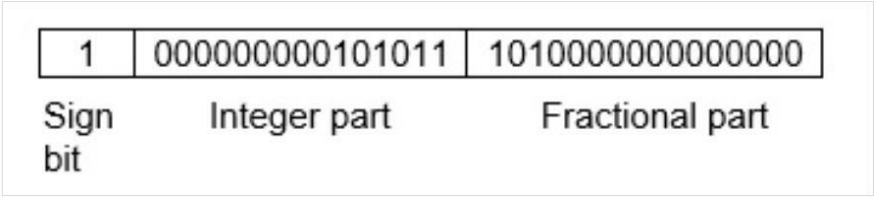
(ii) Floating Point Notation.

Fixed Point Notation: In **fixed point notation**, there are a fixed number of digits after the decimal point, whereas **floating point number** allows for a varying number of digits after the decimal point.

This representation has fixed number of bits for integer part and for fractional part. For example, if given fixed-point representation is IIII.FFFF, then you can store minimum value is 0000.0001 and maximum value is 9999.9999. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.



Assume number is using 32-bit format which reserve 1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part. Then, -43.625 is represented as following:



Where, 0 is used to represent + and 1 is used to represent -. 000000000101011 is 15-bit binary value for decimal 43 and 1010000000000000 is 16-bit binary value for fractional 0.625.

The advantage of using a fixed-point representation is performance and disadvantage is relatively limited range of values that they can represent. So, it is usually inadequate for numerical analysis as it does not allow enough numbers and accuracy. A number whose representation exceeds 32 bits would have to be stored inexactly.

Floating Point Representation:

In some numbers, which have a fractional part, the position of the decimal point is not fixed as the number of bits before (or after) the decimal point may vary. **E.g.: 0010.01001, 0.0001101, -1001001.01** etc. the position of the decimal point is not fixed, instead it **"floats"** in the number. Such numbers are called Floating Point Numbers. Floating Point Numbers are stored in a "Normalized" form.

NORMALIZATION OF A FLOATING POINTNUMBER:

Normalization is the process of shifting the point, left or right, so that there is only one non-zero digit to the left of the point.

01010.01 $(-1)^{\boldsymbol{0}} \times 1.\boldsymbol{01001} \times 2^3$

11111.01 $(-1)^{\boldsymbol{0}} \times 1.\boldsymbol{111101} \times 2^4$

-10.01 $(-1)^{\boldsymbol{1}} \times 1.\boldsymbol{001} \times 2^1$

A normalized form of a number is:

$$\mathbf{-1^s\ x1.MX2^E}$$

Where: S = Sign, M = Mantissa and E = Exponent.

As Normalized numbers are of the 1.M format, the "1" is not actually stored, it is instead assumed. Also the Exponent is stored in the biased form by adding an appropriate bias value to it so that -ve exponents can be easily represented.

Advantages of Normalization.

1. Storing all numbers in a standard for makes **calculations easier** and **faster**.
2. By **not storing** the **1** (of 1.M format) for a number, considerable **storage space** is **saved**.
3. The **exponent** is **biased** so there is **no need** for **storing** its **sign bit** (as the biased exponent cannot be -ve).

SHORT REAL FORMAT / SINGLE PRECISION FORMAT / IEEE 754: 32 BIT FORMAT:

S	Biased Exponent	Mantissa
(1)	(8) Bias value = 127	(23 bits)

1. **32 bits** are used to store the **number**.
2. **23 bits** are used for the **Mantissa**.
3. **8 bits** are used for the Biased **Exponent**.
4. **1 bit** used for the **Sign** of the number.
5. The **Bias** value is $(\mathbf{127})_{10}$.

Range: $\pm 1 \times 10^{-38}$ to $\pm 3 \times 10^{38}$

LONG REAL FORMAT / DOUBLE PRECISION FORMAT / IEEE 754: 64 BIT FORMAT

1. **64 bits** are used to store the **number**.
2. **52 bits** are used for the **Mantissa**.
3. **11 bits** are used for the Biased **Exponent**.
4. **1 bit** used for the **Sign** of the number.
5. The **Bias** value is $(\mathbf{1023})_{10}$.
6. The range is $+\mathbf{10^{-308}}$ **to** $+\mathbf{10^{308}}$approximately.

s	Biased Exponent	Mantissa
1 bit	11-bits (Bias value:1023)	52-bits

Extreme cases of floating point numbers:

Floating point numbers are represented in IEEE formats. Consider IEEE 754 32-bit format also called Single Precision format or Short real format.

Overflow:

For a value, 1.0 the normalized form will be

$(-1)^0 \times 1.0 \times 2^0$

Here the True Exponent is 0.

```
If:   TE = 0,     BE = 127    Representation = 0111 1111
If:   TE = 1,     BE = 128    Representation = 1000 0000
If:   TE = 2,     BE = 129    Representation = 1000 0000
...
If:   TE = 127,   BE = 254    Representation = 1111 1110
If:   TE = 128,   BE = 255    Representation = 1111 1111
If:   TE = 129,   BE = 255    Representation = 1111 1111
If:   TE = 130,   BE = 255    Representation = 1111 1111
```

This is because the 8-bit biased exponent cannot hold a value more than 255.Hence, all cases where the TE = 128 or more, the ***BE will be represented as 1111 1111.This indicates as exception (error) called***

OVERFLOW. The number is called NaN (Not a Number).It is identified by Exponent being all 1s (1111

1111).Here, the Mantissa can be anything! The **Extreme case of NaN is Infinity**. It is also an OVERFLOW and hence the Exponent will be 1111 1111.To differentiate Infinity from NaN, the Mantissa in infinity is 0000 0000.Hence **Infinity is identified as Exponent all 1s and Mantissa all 0s.**

Suppose the number is 0.1.It will be normalized as

$(-1)^0 \times 1.0 \times 2^{-1}$

The true exponent here is -1.

```
If:  TE = -1,    BE = 126    Representation = 0111 1110
If:  TE = -2,    BE = 125    Representation = 0111 1101
…
If:  TE = -126,  BE = 1      Representation = 0000 0001
If:  TE = -127,  BE = 0      Representation = 0000 0000
If:  TE = -128,  BE = 0      Representation = 0000 0000
If:  TE = -129,  BE = 0      Representation = 0000 0000
```

Underflow: All cases where the TE = -127 or less, the BE will be represented as 0000 0000.This indicates as exception (error) called UNDERFLOW.

The number is called De-Normal Number. It is identified by Exponent being all 0s (0000 0000).Here, the Mantissa can be anything. The **Extreme case of De-Normal Number is Zero.**

It is also an UNDERFLOW and hence the Exponent will be 0000 0000.To differentiate Zero from De-Normal Number, the Mantissa in Zero is 0000 0000.Hence **Zero is identified as Exponent all 0s and Mantissa all 0s**.This means Zero is represented as all 0s.

<u>Example:</u> Convert 2A3BH into Short Real format.

Soln: Converting the number into binary we get:

0010 1010 0011 1011

Normalizing the number we get:

$(-1)^0$**x** 1.**0101000111011** x 2^{13}

Here S = 0; M = 0101000111011; True Exponent = 13.

Bias value for Short Real format is 127:

Biased Exponent (BE) = True Exponent + Bias

= 13 + 127

= 140.

Converting the Biased exponent into binary we get:

Biased Exponent (BE) = (1000 1100)

Representing in the required format we get:

0	**10001100**	**010100011101100…**

S Biased Exp Mantissa

(1) (8) (23)

Computer Arithmetic

Integer Addition:

Addition of Unsigned Integers: Addition of 1-bit numbers is illustrated below. The sum of 1 and 1 is the 2-bit vector 10, which represents the value 2. We say that the sum is 0 and the carry-out is 1. In order to add multiple-bit numbers, We add bit pairs starting from the low-order (right) end of the bit vectors, propagating carries toward the high-order (left) end. The carry-out from a bit pair becomes the carry-in to the next bit pair to the left. The carry-in must be added to a bit pair in generating the sum and carry-out at that position. For example, if both bits of a pair are 1 and the carry-in is 1, then the sum is 1 and the carry-out is 1, which represents the value 3.

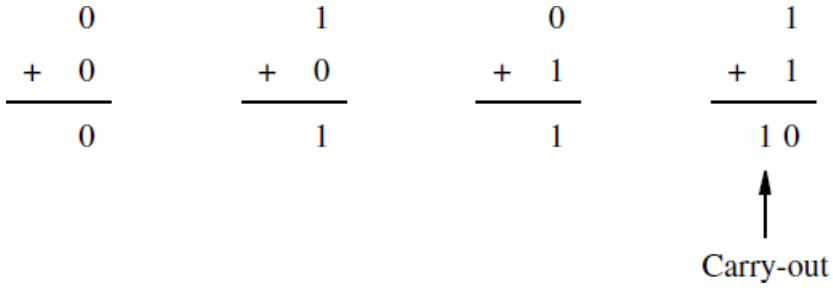


Fig: Addition of 1-bit Numbers

Addition and Subtraction of Signed Integers:

- To add two numbers, add their n-bit representations, ignoring the carry-out bit fromthe most significant bit (MSB) position. The sum will be the algebraically correct value in2's-complement representation if the actual result is in the range$-(2^{n-1})$ through$+2^{n-1}-1$.
- To subtract two numbers X and Y, that is, to perform X − Y , form the 2's-complement of Y , then add it to X using the add rule. Again, the result will be the algebraically correct value in 2's-complement representation if the actual result is in the range $-(2^{n-1})$ through$+2^{n-1}$.

X-Y = X+(-Y) = X+(2'S Complement of Y)

Example: **To perform 7-3 using 2's complement addition**

```
    0 1 1 1
+   1 1 0 1
-------------
1   0 1 0 0
↑
Carry-out
```

If we ignore the carry-out from the fourth bit position in this addition, we obtain the correct answer.

Few more examples:

(a)	0 0 1 0	(+2)	(b)	0 1 0 0	(+4)
	+ 0 0 1 1	(+3)		+ 1 0 1 0	(–6)
	0 1 0 1	(+5)		1 1 1 0	(–2)
(c)	1 0 1 1	(–5)	(d)	0 1 1 1	(+7)
	+ 1 1 1 0	(–2)		+ 1 1 0 1	(–3)
	1 0 0 1	(–7)		0 1 0 0	(+4)
(e)	1 1 0 1	(–3)	⇒	1 1 0 1	
	– 1 0 0 1	(–7)		+ 0 1 1 1	
				0 1 0 0	(+4)

Sign Extension: We often need to represent a value given in a certain number of bits by using a larger number of bits. For a positive number, this is achieved by adding 0s to the left. For a negative number in 2's-complement representation, the leftmost bit, which indicates the sign of the number, is a 1. A longer number with the same value is obtained by replicating the sign bit to the left as many times as needed.

Overflow in Integer Arithmetic: Using 2's-complement representation, n bits can represent values in the range $-(2^{n-1})$ through$+2^{n-1}$.For example, the range of numbers that can be represented by 4 bits is −8through +7.When the actual result of an arithmetic operation is outside the represent able range, an arithmetic overflow has occurred.

Introduction to adder circuits:

ONE BIT ADDITION: FULL ADDER

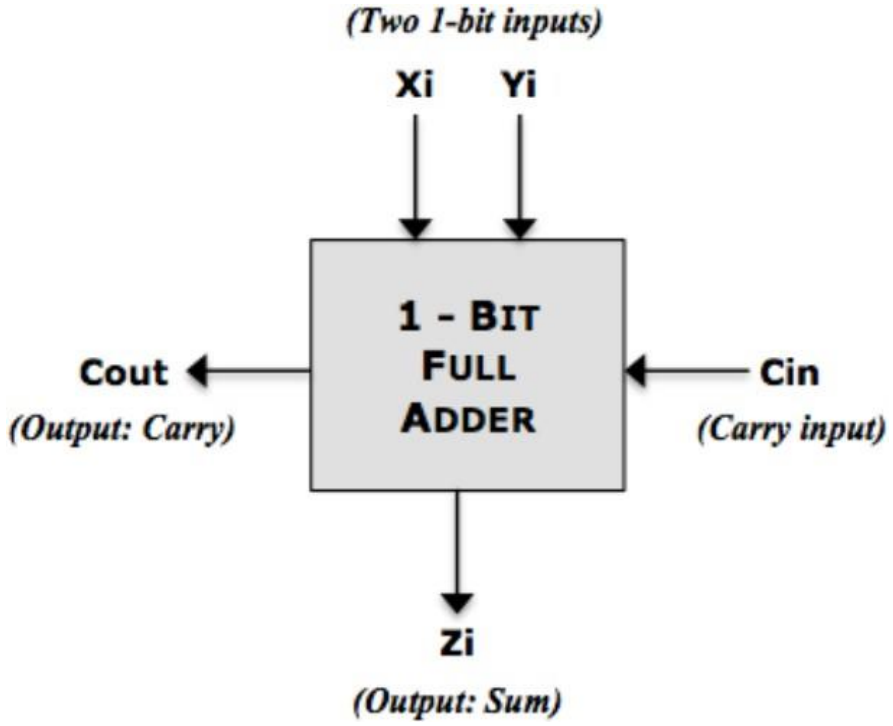
1) It is a 1-bit adder circuit.

2) It adds two 1-bit inputs Xi & Yi, along with a Carry Input Cin.

3) It produces a sum Zi and a Carry output Cout.

4) As it considers a carry input, it can be used in combination to add large numbers.

5) Hence it is called a Full Adder.



Inputs bits: Xi and Yi.
Input Carry: Cin

Output (Sum): Zi
Output (Carry): Cout

Formula for Sum (Zi)

$$Zi = Xi \oplus Yi \oplus Cin$$
$$\therefore Zi = Xi \cdot Yi \cdot Cin + Xi \cdot Yi \cdot \overline{Cin} + Xi \cdot \overline{Yi} \cdot Cin + \overline{Xi} \cdot Yi \cdot Cin$$

Formula for Carry (Cout)

$$Cout = Xi \cdot Yi + Xi \cdot Cin + Yi \cdot Cin$$

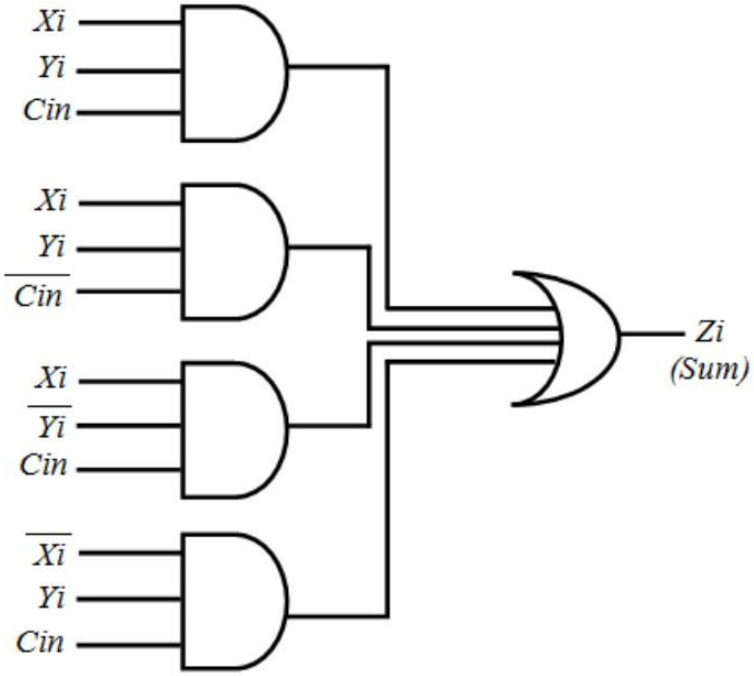


Fig: Circuit for Sum

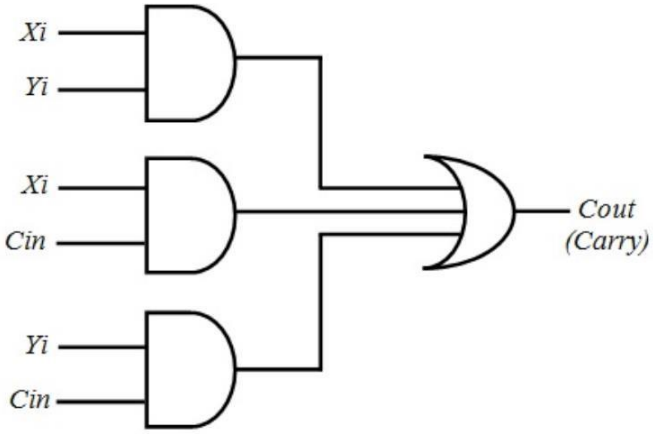


Fig: Circuit for carry

RIPPLE CARRY ADDER(For Multiple bit addition):

1) A Full Adder can add two "1-bit" numbers with a Carry input.
2) It produces a "1-bit" Sum and a Carry output.
3) Combining many of these Full Adders, we can add multiple bits.
4) One such method is called Serial Adder.
5) Here, bits are added one-by-one from Least significant bit(LSB) onwards.
6) The carries are connected in a chain through the full adders. The Carry of each stage is propagated (Rippled) into the next stage.
7) Hence, these adders are also called Ripple Carry Adders.

Advantage: They are very easy to construct.

Drawback: As addition happens bit-by-bit, they are slow.

8) Number of cycles needed for the addition is equal to the number of bits to be added.

Inputs:

Assume X and Y are two "4-bit" numbers to be added, along with a Carry input CIN.

X = X0 X1 X2 X3 (X0 is the MSB … X3 is the LSB)

Y = Y0 Y1 Y2 Y3 (Y0 is the MSB … Y3 is the LSB)

CIN = Carry Input

Outputs:

Assume Z to be a "4-bit" output, and COUT to be the output Carry

Z = Z0 Z1 Z2 Z3 (Z0 is the MSB … Z3 is the LSB)(Here Z represents the sum)

COUT = Carry Output

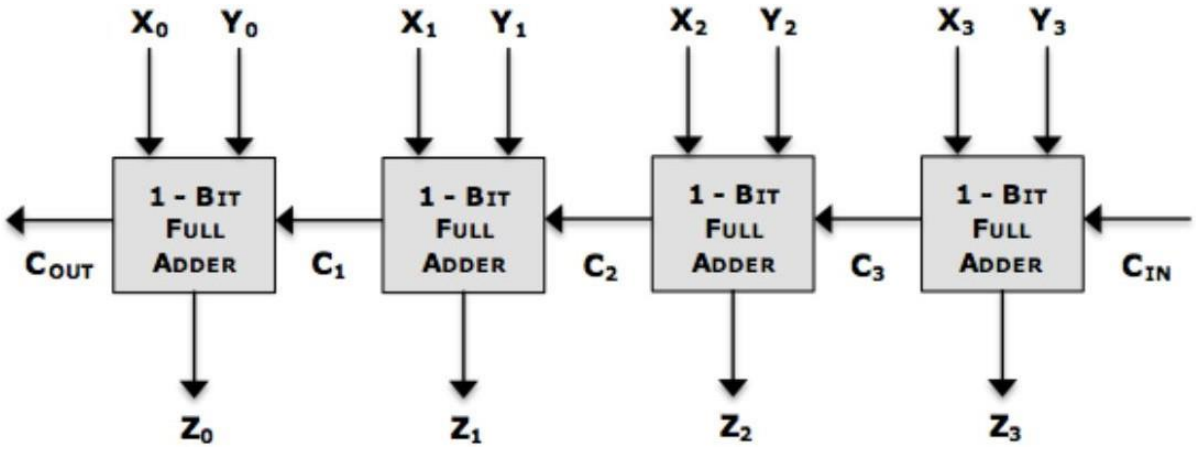


Fig:4-bit Ripple Carry Adder

<u>Carry Look ahead Adder(For multiple bit Addition):</u>

1) This is also called as parallel adder. It is used to add multiple bits simultaneously.

2) While adding multiple bits, the main issue is that of the intermediate carries.

3) In Serial Adders, we therefore added the bits one-by-one.

4) This allowed the carry at any stage to propagate to the next stage.

5) But this also made the process very slow.

6) If we "PREDICT" the intermediate carries, then all bits can be added simultaneously.

7) This is done by the Carry Look Ahead Generator Circuit.

8) Once all carries are determined beforehand, then all bits can be added simultaneously.

Advantage: This makes the addition process extremely fast.

Drawback: Circuit is complex.

Inputs:

Assume X and Y are two "4-bit" numbers to be added, along with a Carry input CIN.

X = X0 X1 X2 X3 (X0 is the MSB … X3 is the LSB); Y = Y0 Y1 Y2 Y3 & CIN = Carry Input

Outputs:

Assume Z to be a "4-bit" output, and C_{OUT} to be the output Carry

Z = Z0 Z1 Z2 Z3 & C_{OUT} = Carry Output

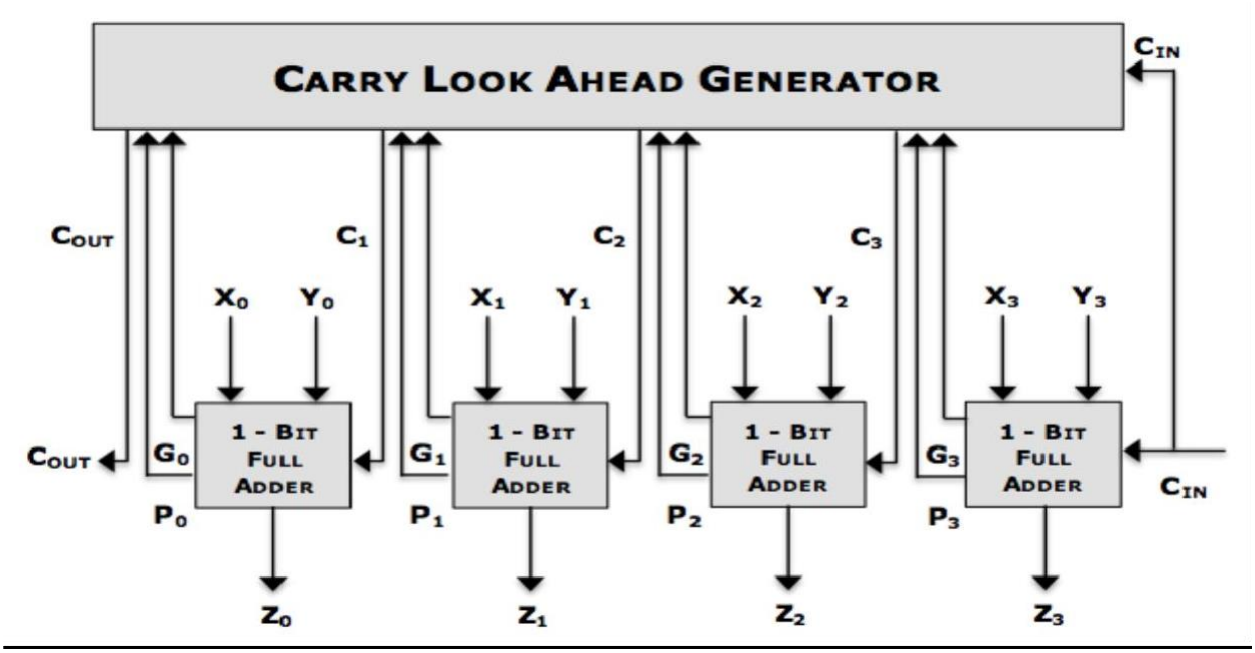


Fig: Circuit for Carry Look ahead Adder

We can "Predict" (Look Ahead) all the intermediate carries in the following manner: The carry at any stage can be calculated as:

$$C_i = X_i.Y_i + X_i.C_{in} + Y_i.C_{in}$$
$$C_i = X_i.Y_i + C_{in}(X_i + Y_i)$$

This implies $C_i = G_i + P_i.C_{IN}$

Here $G_i = X_i.Y_i$ ... (Generate)
And $P_i = X_i + Y_i$ ... (Propagate)

We need to predict the Carries: C3, C2, C1 and C0

C3 = G3 + P3CIN (I)

C2 = G2 + P2C3

Substituting the value of C3, we get:

C2 = G2 + P2G3 + P2P3CIN (II)

C1 = G1 + P1C2

Substituting the value of C2, we get:

C1 = G1 + P1G2 + P1P2G3 + P1P2P3CIN (III)

C0 = G0 + P0C1

Substituting the value of C1, we get:

C0 = G0 + P0G1 + P0P1G2 + P0P1P2G3 + P0P1P2P3CIN (IV)

From the above four equations, it is clear that the values of all the four Carries (C3, C2, C1, C0) can be determined beforehand even without doing the respective additions. To do this we need the values of all G's (Xi.Yi) and all P's (Xi+Yi) and the original carry input CIN. This is done by the Carry Look Ahead Generator Circuit.

Cycle 1: g_1, p_1, g_2, p_2, g_3, p_3, g_0, $p_0$are given to the carry look ahead generator.

Cycle 2: Input carries are given to the adders by the carry generator.

Cycle 3: Results are produced.

Total number of cycles required :3

Multiplication:

1) Shift and Add: This method is used to multiply two unsigned numbers. When we multiply two "N-bit" numbers, the answer is "2 x N" bits. Three registers A, Q and M, are used for this process. Q contains the Multiplier and M contains the Multiplicand. A (Accumulator) is initialized with 0. At the end of the operation, the Result will be stored in (A & Q) combined. The process involves addition and shifting. That is why it is called shift and add method.

Algorithm:

The **number of steps** required is equal to the **number of bits in the multiplier**.

1) At each step, **examine** the current **multiplier bit** starting from the **LSB**.

2) If the current **multiplier bit is "1"**, then the **Partial-Product** is the **Multiplicand** itself.

3) If the current **multiplier bit is "0"**, then the **Partial-Product** is the **Zero**.

4) At each step, **ADD the Partial-Product to the Accumulator**.

5) Now **Right-Shift the Result** produced so far (**A & Q combined**).

Repeat steps 1 to 5 for **all bits** of the multiplier.

The **final answer** will be in **A & Q** combined.

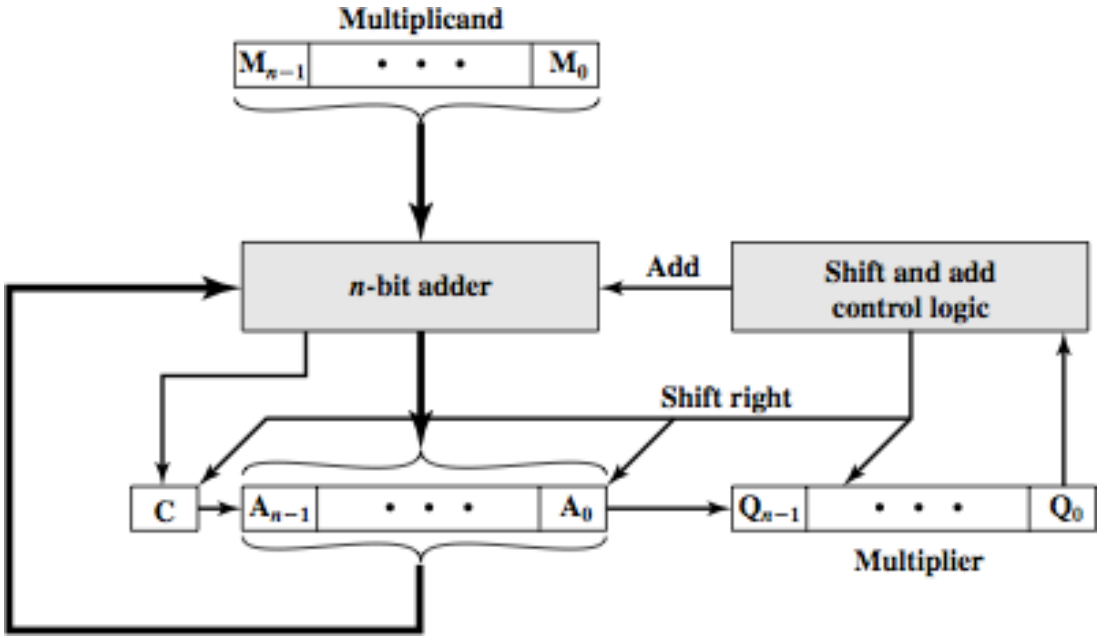


Fig: Shift and Add Multiplication

Example: Let us consider 7X6

```
                    0  1  1  1   … Multiplicand (7)
              X     0  1  1  0   … Multiplier (6)
-------------------------------------------------
                    0  0  0  0   … Partial-Product
                 0  1  1  1  X              ”
              0  1  1  1  X  X              ”
+  0  0  0  0  X  X  X              ”
-------------------------------------------------
   0  1  0  1  0  1  0   … Result (42)
-------------------------------------------------
```

Step	**C** **Carry**	**A** **Accumulator**	**Q** **Multiplier**	**M** **Multiplicand**	**Explanation**
	0	0000	0110	0111	*Initial Value*
1	**0** **0**	0000 0000	0110 0011		*Current Multiplier bit is “0” so ADD “0” to Accumulator and Right-Shift*

2	0 0	0111 0011	0011 1001		*Current Multiplier bit is "1" so ADD Multiplicand to Accumulator and Right-Shift*
3	0 0	1010 0101	1001 0100		*Current Multiplier bit is "1" so ADD Multiplicand to Accumulator and Right-Shift*
4	0 0	0101 ***0010***	0100 ***1010***		*Current Multiplier bit is "0" so ADD "0" to Accumulator and Right-Shift*

2) Booth Multiplier(For signed Multiplication):

Booth's Algorithm is used to **multiply two SIGNED numbers**. When we multiply two **"N-bit"** numbers, the answer is "**2 x N**" bits. Three registers A, Q and M, are used for this process.**Q** contains the **Multiplier** and **M** contains the **Multiplicand**. **A** (**Accumulator**) is initialized with 0.At the end of the operation, the **Result** will be stored in (**A & Q**) combined. The process involves **addition, subtraction** and **shifting**.

Algorithm:

The **number of steps** required is equal to the **number of bits in the multiplier**.

At the beginning, consider an **imaginary "0" beyond LSB of Multiplier**

1) At each step, **examine two adjacent Multiplier bits** from **Right to Left**.

2) If the transition is from **"0 to 1"** then **Subtract M** from **A** and **Right-Shift** (A & Q) combined.

3) If the transition is from **"1 to 0"** then **ADD M** to **A** and **Right-Shift**.

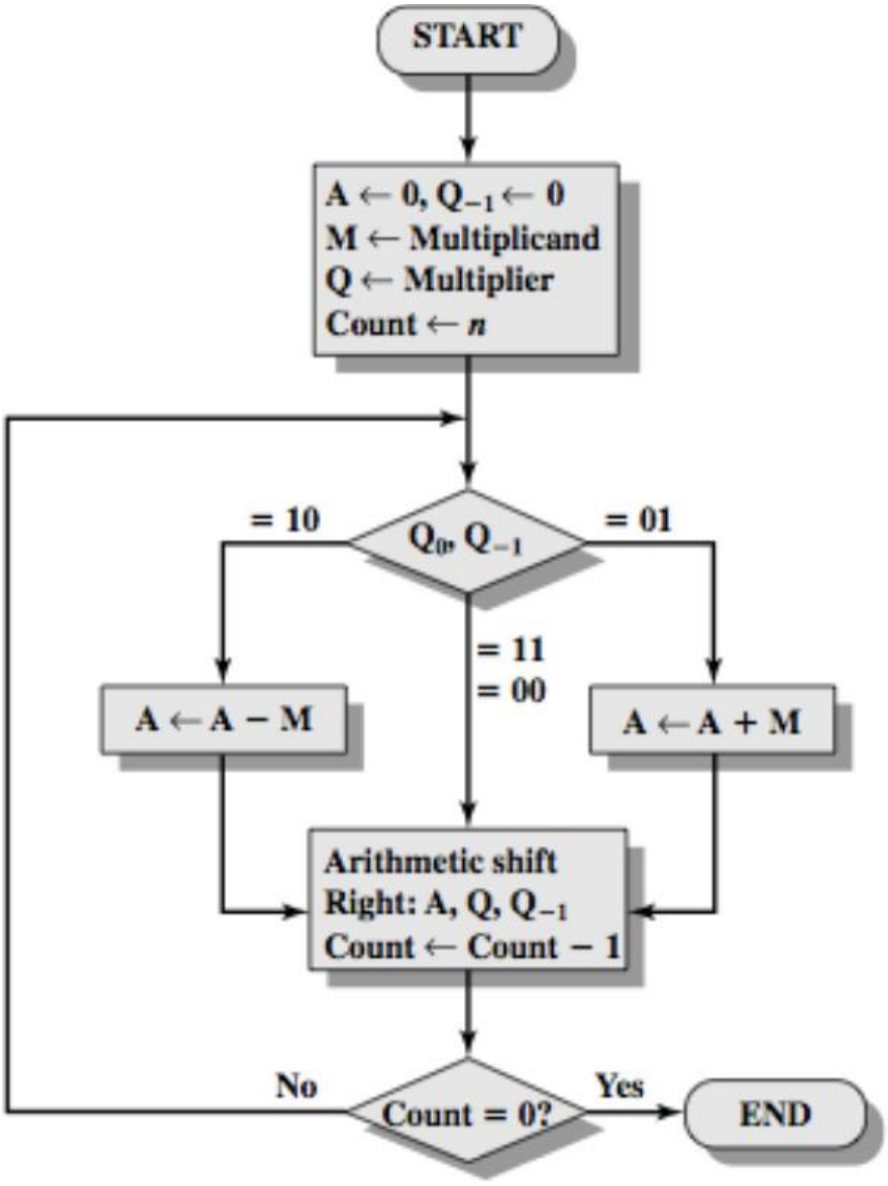
4) If the transition is from **"0 to 0"** then **simply Right-Shift**.

5) If the transition is from **"1 to 1"** then **simply Right-Shift**.

Repeat steps 1 to 5 for **all bits** of the multiplier.

The **final answer** will be in **A & Q** combined.

Flowchart for Booth's Algorithm:



Example: -9x10=-90

Multiplicand (M): **-9 = 10111** **9 = 01001**. (Two's Complement Form)

Multiplier (Q): **10 = 01010**. **-10 = 10110** (Two's Complement Form)

step	**A** **Accumulator**	**Q** **Multiplier**	**Q(-1)**	**M** **Multiplicand**
Initial	**00000**	**01010**	**0**	**10111**
1) (0 ç 0) No Add or Sub Right-Shift	**00000** **00000**	**01010** **00101**	**0** **0**	

2) (1 ç 0)	**01001**	**00101**	**0**	
Perform **(A - M)** Right-Shift	**00100**	**10010**	**1**	
3) (0 ç 1) Perform **(A + M)** Right-Shift	**11011** **11101**	**10010** **11001**	**1** **0**	
4) (1 ç 0) Perform **(A - M)** Right-Shift	**00110** **00011**	**11001** **01100**	**0** **1**	
5) (0 ç 1) Perform **(A + M)** Right-Shift	**11010** **11101**	**01100** **00110**	**1** **0**	

Restoring and Non-Restoring Division:

Non Restoring Division:

1) Let Q register hold the divided, M register holds the divisor and A register is 0.
2) On completion of the algorithm, Q will get the quotient and A will get the remainder.

Algorithm:

The number of steps required is equal to the number of bits in the Dividend.
1) At each step, left shift the dividend by 1 position.
2) Subtract the divisor from A (perform A - M).
3) If the result is positive then the step is said to be "Successful". In this case quotient bit will be "1" and Restoration is NOT Required. The Next Step will also be Subtraction.
4) If the result is negative then the step is said to be "Unsuccessful". In this case quotient bit will be "0". Here Restoration is NOT Performed. Instead, the next step will be ADDITION in place of subtraction. As restoration is not performed, the method is called Non-Restoring Division.
Repeat steps 1 to 4 for all bits of the Dividend.

Example: (7) / (5)
Dividend (Q) = 7
Divisor (M) = 5
Accumulator (A) = 0

7 = 0111 **5** = 0101

-7 = 1001-**5** = 1011

	Accumulator A(0)	Dividend Q(7)	Divisor M(5)
Initial Values	**0000**	**0111**	**0101**
Step 1: Left shift A-M Unsuccessful(-ve) Next step: Add	0000 +1011 **1011**	**111_** **1110**	
Step 2: Left shift A+M Unsuccessful(-ve) Next step: Add	0111 +0101 **1100**	**110_** **1100**	
Step 3: Left shift A+M Unsuccessful(-ve) Next step: Add	1001 +0101 **1110**	**100_** **1000**	
Step 4: Left shift A+M successful(+ve)	1101 +0101 **0010**	**000_** **0001**	
	Remainder:2	**Quotient:1**	

RESTORING DIVISION (For unsigned Numbers)

1) Let Q register hold the divided, M register holds the divisor and A register is 0.

2) On completion of the algorithm, Q will get the quotient and A will get the remainder.

Algorithm:

The number of steps required is equal to the number of bits in the Dividend.

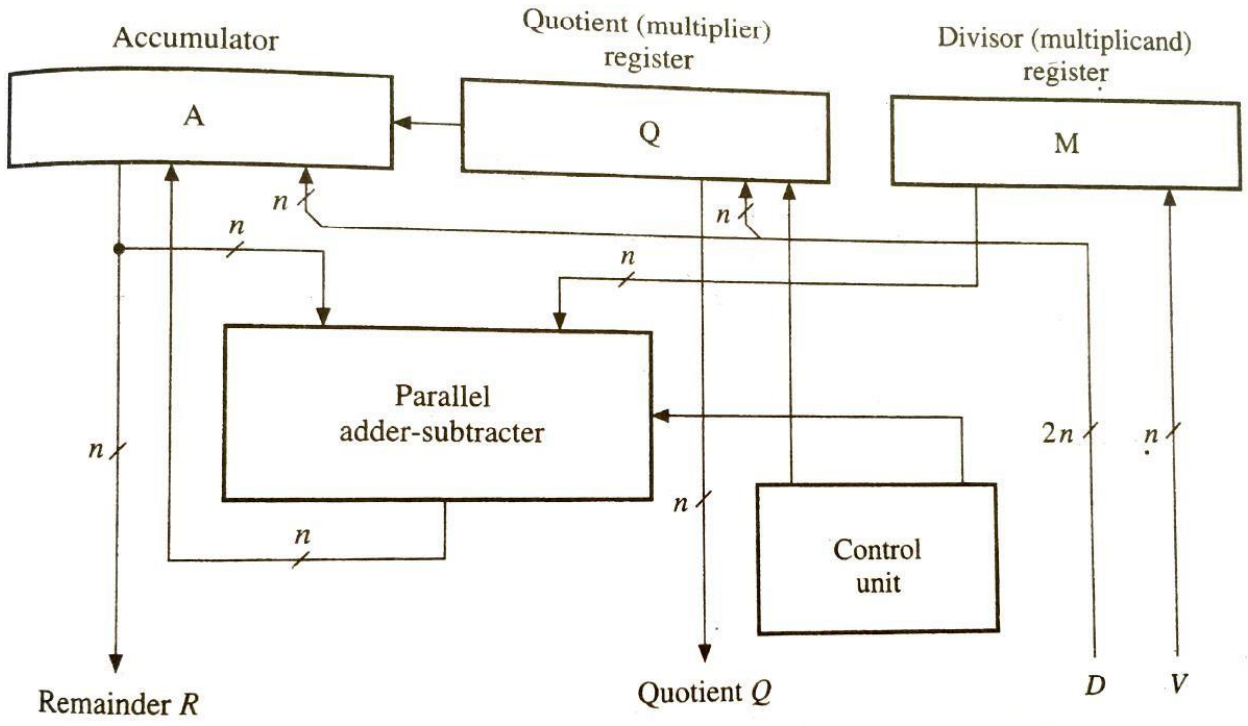
1) At each step, left shift the dividend by 1 position.

2) Subtract the divisor from A (perform A - M).

3) If the result is positive then the step is said to be "Successful". In this case quotient bit will be "1" and Restoration is NOT Required.

4) If the result is negative then the step is said to be "Unsuccessful". In this case quotient bit will be "0".Here Restoration is performed by adding back the divisor.

Hence the method is called Restoring Division. Repeat steps 1 to 4 for all bits of the Dividend.



Example: (6) / (4)

Dividend (Q) = 6

Divisor (M) = 4

Accumulator (A) = 0

6 = 0110 4 = 0100

-6 = 1010 -4 = 1100

	Accumulator A(0)	**Dividend Q(6)**	**Divisor M(4)**
Initial Values	0000	0110	0100
Step 1: Left shift A-M Unsuccessful(-ve) Restoration:	0000 + 1100 **1100** 0000	110_ 1100	
Step 2: Left shift A-M Unsuccessful(-ve) Restoration:	0001 +1100 1101 0001	100_ 1000	
Step 3: Left shift A-M Unsuccessful(-ve) Restoration:	0011 +1100 1111 0011	000_ 0000	

Step 3: Left shift A-M Successful(+ve) No Restoration	0110 +1100 0010	000_ 0001	
	Remainder(2)	Quotient(1)	

RESTORING DIVISION FOR SIGNED NUMBERS:

1) Let M register hold the divisor, Q register hold the divided.

2) A register should be the signed extension of Q.

3) On completion of the algorithm, Q will get the quotient and A will get the remainder.

Algorithm:

The number of steps required is equal to the number of bits in the Dividend.

2) At each step, left shift the dividend by 1 position.

3) If Sign of A and M is the same then Subtract the divisor from A (perform A - M),

Else Add M to A

4) After the operation, If Sign of A remains the same or the dividend (in A and Q) becomes zero, then the step is said to be "Successful". In this case quotient bit will be "1" and Restoration is NOT Required.

5) If Sign of A changes, then the step is said to be "Unsuccessful". In this case quotient bit will be "0".Here Restoration is Performed. Hence, the method is called Restoring Division. Repeat steps 1 to 4 for all bits of the Dividend.

Note: The result of this algorithm is such that, the quotient will always be positive and the remainder will get the same sign as the dividend.

Example: (-19) / (7)

19 = 010011 7 = 000111

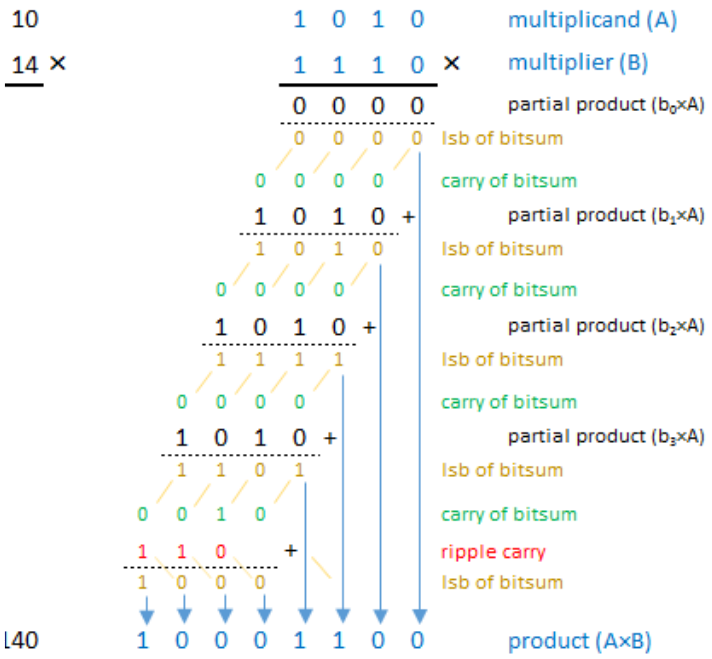
-19 = 101101 -7 = 111001

	Accumulator A(Sign Extension)	**Dividend Q(-19)**	**Divisor M(7)**
Initial Values	111111	101101	000111
Step 1: Left-shift Sign(A,M) Different: A+M Sign changes: Unsuccessful Restore	111111 + 000111 000110 111111	01101_ 011010	

Step 2: Left-shift	111110	11010_	
Sign(A,M) Different: A+M	+ 000111		
Sign changes: Unsuccessful	000101		
Restore	111110	110100	
Step 3: Left-shift	111101	10100_	
Sign(A,M) Different: A+M	+ 000111		
Sign changes: Unsuccessful	000100		
Restore	111101	101000	
Step 4: Left-shift	111011	01000_	
Sign(A,M) Different: A+M	+ 000111		
Sign changes: Unsuccessful	000010		
Restore	111011	010000	
Step 5: Left-shift	110110	10000_	
Sign(A,M) Different: A+M	+ 000111		
Sign still same: Successful	111101		
Restoration not required	111101	100001	
Step 6: Left-shift	111011	00001_	
Sign(A,M) Different: A+M	+ 000111		
Sign changes: Unsuccessful	000010		
Restore	111011	000010	
	Remainder(-5)	Quotient(2)	

Carry-save Array Multiplier

Important advance in improving the speed of multipliers, pioneered by Wallace, is the use of carry save lers (CSA). Even though the building block is still the multiplying adder (ma), the topology of prevents a ple carry by ensuring that, wherever possible, the carry-out signal propagates downward and not sideways. ıstration below gives an example of this multiplication process.



Inside a carry-save array multiplier

Again, the building block is the multiplying adder (ma) as describe on the previous page. However, th topology is so that the carry-out from one adder is not connected to the carry-in of the next adder. Henc preventing a ripple carry. The circuit diagram below shows the connections between these blocks.

4-bit carry-save array multiplier

The observant reader might notice that ma0x can be replaced with simple AND gates, ma4x can b replaced by adders. Also the block ma43 is not needed. More interesting, the the ripple adder in the las row, can be replace with the faster carry look ahead adder.

Similar to the carry-propagate array multiplier, using Verilog HDL we can generate instances of ma block based on the word length of the multiplicand and multiplier (N). To describe the circuit in Verilog HDI we need to derive the rules that govern the connections between the blocks.

Start by numbering the output ports based on their location in the matrix. For this circuit, we have th output signals *sum* (s) and *carry-out* (c). E.g. c_13 identifies the carry-out signal for the block in row 1 an column 3. Next, we express the input signals as a function of the output signal names s and c and do th same for the product itself as shown in the table below.

output ports

so	3	2	1	jj=0
ii=0	s02	s01	s00	p0
1	s12	s11	s10	p1
2	s22	s21	s20	p2
3	s32	s31	s30	p3
4	p7	p6	p5	p4

co	3	2	1	jj=0
ii=0	c03	c02	c01	c00
1	c13	c12	c11	c10
2	c23	c22	c21	c20
3	c33	c32	c31	c30
4	p8	c42	c41	c40

input ports

x	3	2	1	jj=0
ii=0	a3	a2	a1	a0
1	a3	a2	a1	a0
2	a3	a2	a1	a0
3	a3	a2	a1	a0
4	c42	c41	c40	0

y	3	2	1	jj=0
ii=0	b0	b0	b0	b0
1	b1	b1	b1	b1
2	b2	b2	b2	b2
3	b3	b3	b3	b3
4	1	1	1	1

si	3	2	1	jj=0
ii=0	0	0	0	0
1	0	s02	s01	s00
2	0	s12	s11	s10
3	0	s22	s21	s20
4	0	s32	s31	s30

ci	3	2	1	jj=0
ii=0	0	0	0	0
1	c03	c02	c01	c00
2	c13	c12	c11	c10
3	c23	c22	c21	c20
4	c33	c32	c31	c30

›duct output

p	6	5	4	3	2	1	0	0
	s43	s42	s41	s40	s30	s20	s10	s00

ıction for output signals 'so' and 'co' and output signals 'x', 'y', 'si', 'ci' and 'p'

ed on this table, we can now express the interconnects using Verilog HDL using `?:` expressions.

```
generate genvar ii, jj;
    for ( ii = 0; ii &lt;;= N; ii = ii + 1) begin: gen_ii
        for ( jj = 0; jj &lt;; N; jj = jj + 1) begin: gen_jj
            math_multiplier_ma_block ma(
                .x ( ii &lt;; N ? a[jj] : (jj &gt; 0) ? c[N][jj-1] :
1'b0 ),
                .y ( ii &lt;; N ? b[ii] : 1'b1 ),
                .si ( ii &gt; 0 jj &lt;; N - 1 ? s[ii-1][jj+1] : 1'b0
),
                .ci ( ii &gt; 0 ? c[ii-1][jj] : 1'b0 ),
                .so ( s[ii][jj] ),
                .co ( c[ii][jj] ) );
            if ( ii == N ) assign p[N+jj] = s[N][jj];
        end
        assign p[ii] = s[ii][0];
    end
endgenerate
```

Results

The propagation delay tpd depends size N and the value of operands. For a given size N, the maximu propagation delay occurs when the low order bit cause a carry/sum that propagate to the highest order bi This worst-case propagation delay is linear with 2N, this makes this carry-save multiplier is about *33 faster* as the ripple-carry multiplier. Note that the average propagation delay is about half of this.he pos map Timing Analysis tool shows the worst-case propagation delays for the Terasic Altera Cyclone IV DE Nano. The exact value depends on the model and speed grade of the FPGA, the silicon itself, voltage an the die temperature.

N	*Timing Analysis*			*Measured*
	slow 85°C	*slow 0°C*	*fast 0°C*	*actual*
4-bits	9.0 ns	8.0 ns	5.6 ns	

8-bits	18.7 ns	16.8 ns	11.4 ns	
16-bits	30.9 ns	27.6 ns	18.3 ns	
27-bits	46.8 ns	41.9 ns	27.7 ns	
32-bits	57.9 ns	51.6 ns	34.3 ns	

MODULE-3

CPU CONTROL UNIT DESIGN

- **Hardwired CU :**

In Hardwired CU, control signals are produced by hardware. There are three types of Hardwired Control Units

1) STATE TABLE METHOD
2) DELAY ELEMENT METHOD
3) SEQUENCE COUNTER METHOD

STATE TABLE METHOD:

1) It is the most basic type of hardwired control unit.
2) Here the behavior of the control unit is represented in the form of a table called the state table.
3) The rows represent the T-states and the columns indicate the instructions.
4) Each intersection indicates the control signal to be produced, in the corresponding T-state of every instruction.
4) A circuit is then constructed based on every column of this table, for each instruction.

T-STATES	INSTRUCTIONS			
	I_1	I_2	...	I_N
T_1	$Z_{1,1}$	$Z_{1,2}$	...	$Z_{1,N}$
T_2	$Z_{2,1}$	$Z_{2,2}$	...	$Z_{2,N}$
...	...	...	...	...
T_M	$Z_{M,1}$	$Z_{M,2}$	...	$Z_{M,N}$

$Z_{1,1}$: Control Signal to be produced in T-state (T_1) of Instruction (I_1)

ADVANTAGE: It is the simplest method and is ideally suited for very small instruction sets.

DRAWBACK: As the number of instructions increase, the circuit becomes bigger and hence more complicated. As a tabular approach is used, instead of a logical approach (flowchart), there are duplications of many circuit elements in various instructions.

Delay Element Method:

1) Here the behavior of the control unit is represented in the form of a flowchart.

2) Each step in the flowchart represents a control signal to be produced.

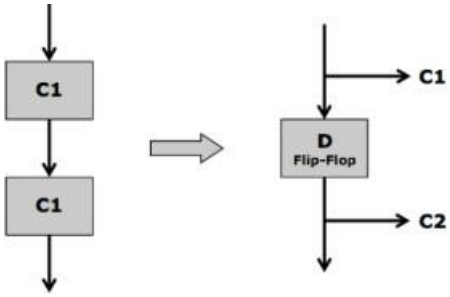
3) Once all steps of a particular instruction, are performed, the complete instruction gets executed.

4) Control signals perform Micro-Operations, which require one T-states each.

5) Hence between every two steps of the flowchart, there must be a delay element.

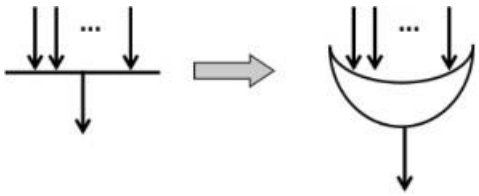
6) The delay must be exactly of one T-state. This delay is achieved by D Flip-Flops.

7) These D Flip-Flops are inserted between every two consecutive control signals.

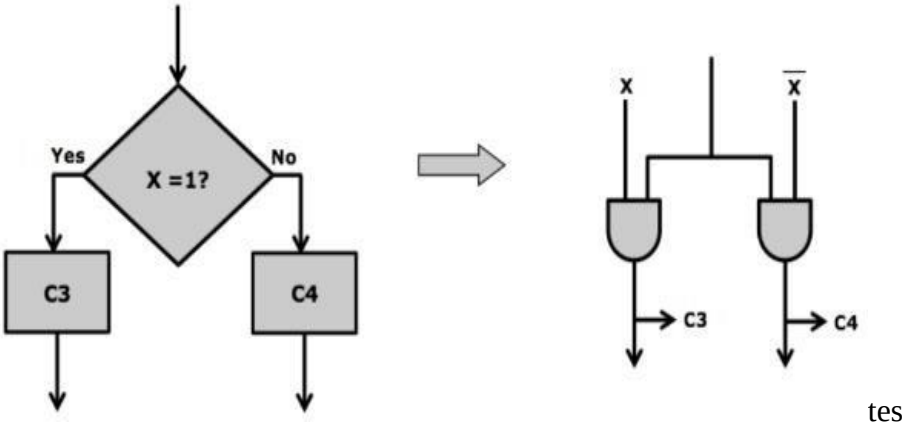


8) Of all D Flip-Flops only one will be active at a time. So the method is also called “One Hot Method”.

9) In a multiple entry point, to combine two or more paths, we use an OR gate.



10) A decision box is replaced by a set of two complementing AND ga



tes

11) A multiple entry point is substituted by an OR gate.

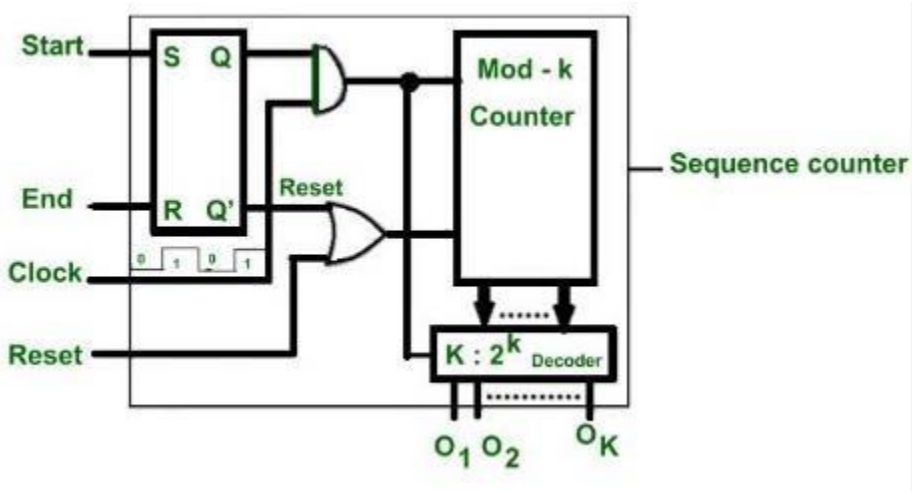
ADVANTAGE:

As the method has a logical approach, it can reduce the circuit complexity. This is done by re-utilizing common elements between various instructions.

DRAWBACK:

As the no of instructions increase, the number of D Flip-Flops increase, so the cost increases. Moreover, only one of those D Flip-Flops are actually active at a time.

SEQUENCE COUNTER METHOD:



1) This is the most popular form of hardwired control unit. The goal of this circuit is to provide triggers to different parts of the circuit after gaps of 1-Tstate.

2) It follows the same logical approach of a flowchart, like the Delay element method, but does not use all those unnecessary D Flip-Flops because at any point of time only one delay element is active and a complex circuitry would involve many delay elements which is very inefficient. The D-Flip-flops are replaced by trigger points which are activated after gaps of one T-state.

Following are the steps involved in designing a CU using Sequence Counter Method.

1) First a flowchart is made representing the behavior of a control unit.

2) It is then converted into a circuit using the same principle of AND & OR gates.

3) We need a delay of 1 T-state (one clock cycle) between every two consecutive control signals.

4) That is achieved by the above circuit.

5) If there are “k” number of distinct steps producing control signals, we employ a “mod k” and “k”output decoder.

6) The counter will start counting at the beginning of the instruction.

7) The “clock” input via an AND gate ensures each count will be generated after 1 T-state.

8) The count is given to the decoder which triggers the generation of “k” control signals, each aftera delay of 1 T-state.

9) When the instruction ends, the counter is reset so that next time, it begins from the first count.

ADVANTAGE:

Avoids the use of too many D Flip-Flops.

GENERAL DRAWBACKS OF A HARDWIRED CONTROL UNIT

1) Since they are based on hardware, as the instruction set increases, the circuit becomes more and more complex. For modern processors having hundreds of instructions, it is virtually impossible to create Hardwired Control Units.

2) Such large circuits are very difficult to debug.

3) As the processor gets upgraded, the entire Control Unit has to be redesigned, due to the rigid nature of hardware design.

Micro programmed CU

WILKES’ DESIGN FOR A MICROPROGRAMMED CONTROL UNIT:

1) Micro programmed Control Unit produces control signals by software, using micro-instructions

2) A program is a set of instructions.

3) An instruction requires a set of Micro-Operations.

4) Micro-Operations are performed by control signals.

5) Instead of generating these control signals by hardware, we use micro-instructions. This means every instruction requires a set of micro-instructions This is called its micro-program.

6) Micro programs for all instructions are stored in a small memory called “Control Memory”. The Control memory is present inside the processor.

7) Consider an Instruction that is fetched from the main memory into the Instruction Register (IR).

8) The processor uses its unique “opcode” to identify the address of the first micro-instruction. That address is loaded into CMAR (Control Memory Address Register). CMAR passes the address to the decoder.

9) The decoder identifies the corresponding micro-instruction from the Control Memory.

10) A micro-instruction has two fields: a control filed and an address field.

Control field: Indicates the control signals to be generated.

Address field: Indicates the address of the next micro-instruction.

11) This address is further loaded into CMAR to fetch the next micro-instruction.

12) For a conditional micro-instruction, there are two address fields. This is because, the address of the next micro-instruction depends on the condition. The condition (true or false) is decided by the appropriate control flag.

13) The control memory is usually implemented using FLASH ROM as it is writable yet non-volatile.

ADVANTAGES

1) The biggest advantage is flexibility.

2) Any change in the control unit can be performed by simply changing the micro-instruction.

3) This makes modifications and up gradation of the Control Unit very easy.

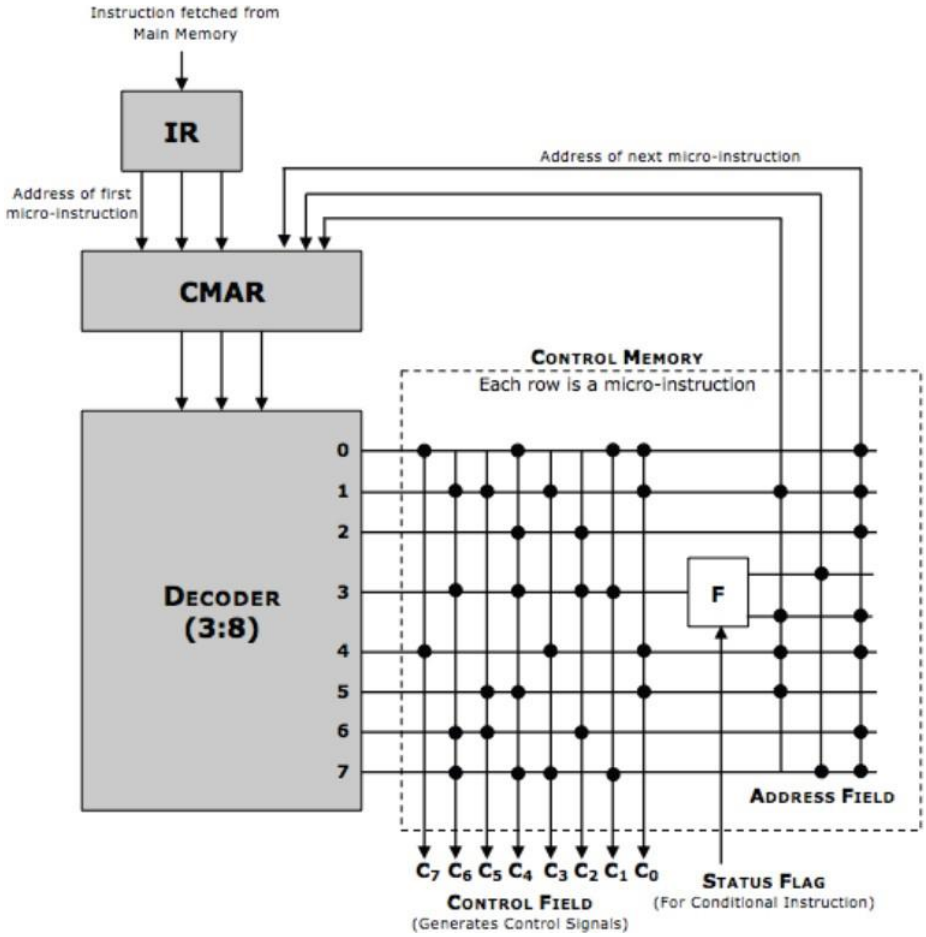
4) Moreover, software can be much easily debugged as compared to a large Hardwired Control Unit.

DRAWBACKS

1) Control memory has to be present inside the processor, increasing its size.

2) This also increases the cost of the processor.

3) The address field in every micro-instruction adds more space to the control memory. This can be easily avoided by proper micro-instruction sequencing.



TYPICAL MICROPROGRAMMED CONTROL UNIT

1) Microprogrammed Control Unit produces control signals by software, using micro-instructions.

2) A program is a set of instructions.

3) An instruction requires a set of Micro-Operations.

4) Micro-Operations are performed by control signals.

5) Here, these control signals are generated using micro-instructions.

6) This means every instruction requires a set of micro-instructions

7) This is called its micro-program.

8) Micro programs for all instructions are stored in a small memory called “Control Memory”.

9) The Control memory is present inside the processor.

10) Consider an Instruction that is fetched from the main memory into the Instruction Register (IR).

11) The processor uses its unique “opcode” to identify the address of the first micro-instruction.

12) That address is loaded into CMAR (Control Memory Address Register) also called μIR.

13) This address is decoded to identify the corresponding μ-instruction from the Control Memory.

14) There is a big improvement over Wilkes’ design, to reduce the size of micro-instructions.

15) Most micro-instructions will only have a Control field.

16) The Control field Indicates the control signals to be generated.

17) Most micro-instructions will not have an address field.

18) Instead, μPC will simply get incremented after every micro-instruction.

19) This is as long as the μ-program is executed sequentially.

20) If there is a branch μ-instruction only then there will be an address filed.

21) If the branch is unconditional, the branch address will be directly loaded into CMAR.

22) For Conditional branches, the Branch condition will check the appropriate flag.

23) This is done using a MUX which has all flag inputs.

24) If the condition is true, then the MUX will inform CMAR to load the branch address.

25) If the condition is false CMAR will simply get incremented.

26) The control memory is usually implemented using FLASH ROM as it is writable yet non-volatile.

ADVANTAGES

1) The biggest advantage is flexibility.

2) Any change in the control unit can be performed by simply changing the micro-instruction.

3) This makes modifications and up gradation of the Control Unit very easy.

4) Moreover, software can be much easily debugged as compared to a large Hardwired Control Unit.

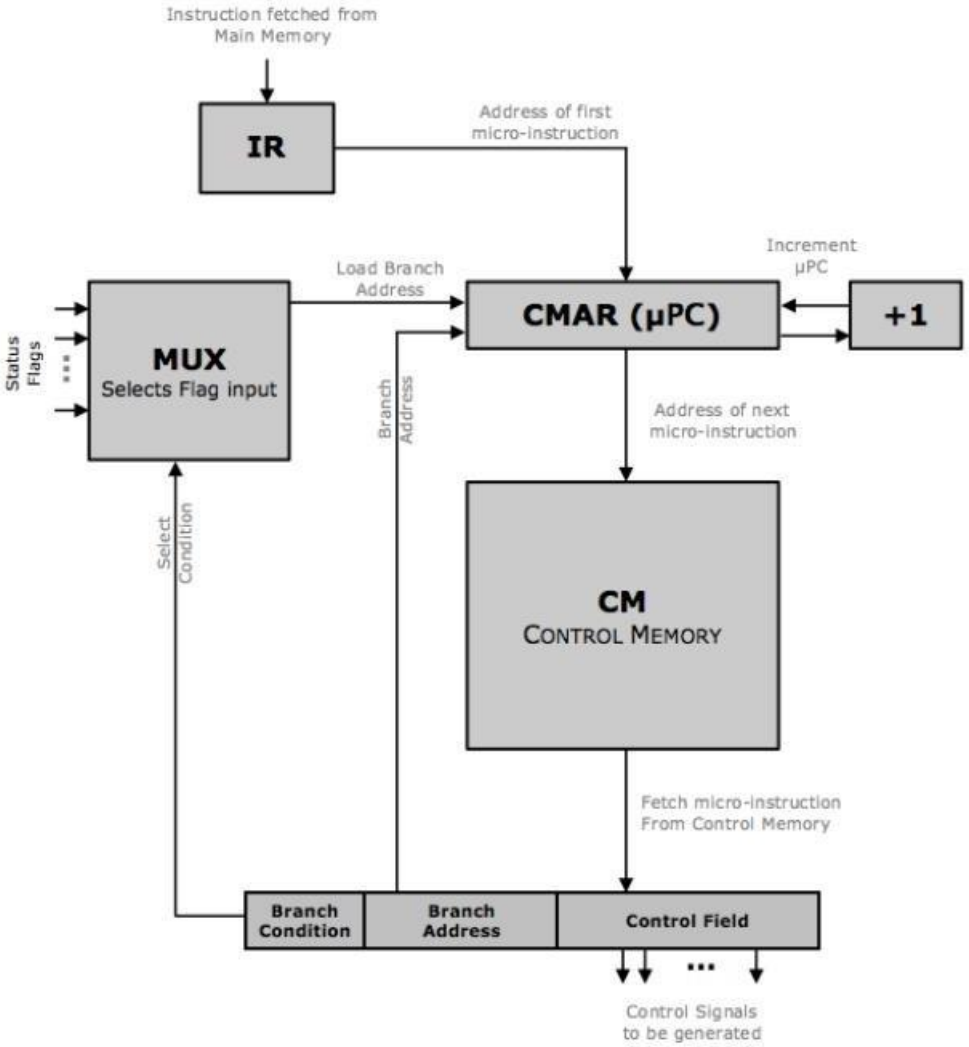
5) Since most micro-instructions are executed sequentially, they don’t need for an address field.

6) This significantly reduces the size of micro-instructions, and hence the Control Memory.

DRAWBACKS

1) Control memory has to be present inside the processor, increasing its size.

2) This also increases the cost of the processor.



Memory System Design

Memory system design: Semiconductor memory technologies, memory organization.

Memory organization: Memory interleaving, concept of hierarchical memory organization, Cache memory, mapping functions, Replacement algorithms, write policies, Virtual Memory Management

Semiconductor Memory Technologies:

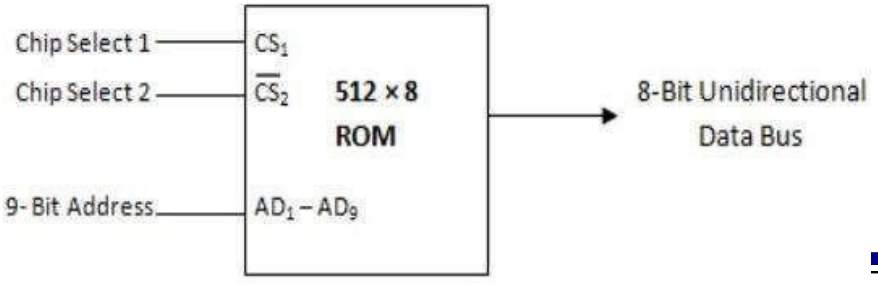
Semiconductor random-access memories (RAMs) are available in a wide range of speeds. Their cycle times range from 100 ns to less than 10 ns. Semiconductor memory is used in any electronics assembly that uses computer processing technology. The use of semiconductor memory has grown, and the size of these memory cards has increased as the need for larger and larger amounts of storage is needed.

There are two main types or categories that can be used for semiconductor technology.

RAM - Random Access Memory: As the names suggest, the RAM or random access memory is a form of semiconductor memory technology that is used for reading and writing data in any order - in other words as it is required by the processor. It is used for such applications as the computer or processor memory where variables and other stored and are required on a random basis. Data is stored and read many times to and from this type of memory.

Block Diagram Representing 128 x 8 RAM
(Random Access Memory)

ROM - Read Only Memory: A ROM is a form of semiconductor memory technology used where the data is written once and then not changed. In view of this it is used where data needs to be stored permanently, even when the power is removed - many memory technologies lose the data once the power is removed. As a result, this type of semiconductor memory technology is widely used for storing programs and data that must survive when a computer or processor is powered down. For example the BIOS of a computer will be stored in ROM. As the name implies, data cannot be easily written to ROM Depending on the technology used in the ROM, writing the data into the ROM initially may require special hardware. Although it is often possible to change the data, this gain requires special hardware to erase the data ready for new data to be written in.



The different memory types or memory technologies are detailed below:

DRAM: Dynamic RAM is a form of random access memory. DRAM uses a capacitor to store each bit of data, and the level of charge on each capacitor determines whether that bit is a logical 1 or 0. However these capacitors do not hold their charge indefinitely, and therefore the data needs to b refreshed periodically. As a result of this dynamic refreshing it gains its name of being a dynamic RAM DRAM is the form of semiconductor memory that is often used in equipment including persona computers and workstations where it forms the main RAM for the computer.

EEPROM: This is an Electrically Erasable Programmable Read Only Memory. Data can be written tc it and it can be erased using an electrical voltage. This is typically applied to an erase pin on the chip. Like other types of PROM, EEPROM retains the contents of the memory even when the power is turnec off. Also like other types of ROM, EEPROM is not as fast as RAM.

EPROM: This is an Erasable Programmable Read Only Memory. This form of semiconductor memory can be programmed and then erased at a later time. This is normally achieved by exposing the silicon to ultraviolet light. To enable this to happen there is a circular window in the package of the EPROM to enable the light to reach the silicon of the chip. When the PROM is in use, this window is normally covered by a label, especially when the data may need to be preserved for an extended period. The PROM stores its data as a charge on a capacitor. There is a charge storage capacitor for each cell and this can be read repeatedly as required. However it is found that after many years the charge may leak away and the data may be lost. Nevertheless, this type of semiconductor memory used to be widely used in applications where a form of ROM was required, but where the data needed to be changed periodically, as in a development environment, or where quantities were low.

FLASH MEMORY: Flash memory may be considered as a development of EEPROM technology Data can be written to it and it can be erased, although only in blocks, but data can be read on ar individual cell basis. To erase and re-programme areas of the chip, programming voltages at levels tha are available within electronic equipment are used. It is also non-volatile, and this makes it particularly useful. As a result Flash memory is widely used in many applications including memory cards for digita cameras, mobile phones, computer memory sticks and many other applications.

F-RAM: Ferroelectric RAM is a random-access memory technology that has many similarities to the standard DRAM technology. The major difference is that it incorporates a ferroelectric layer instead of the more usual dielectric layer and this provides its non-volatile capability. As it offers a non-volatile capability, F-RAM is a direct competitor to Flash.

MRAM: This is Magneto-resistive RAM, or Magnetic RAM. It is a non-volatile RAM memory technology that uses magnetic charges to store data instead of electric charges. Unlike technologies including DRAM, which require a constant flow of electricity to maintain the integrity of the data, MRAM retains data even when the power is removed. An additional advantage is that it only requires low power for active operation. As a result this technology could become a major player in the electronics industry now that production processes have been developed to enable it to be produced.

P-RAM / PCM: This type of semiconductor memory is known as Phase change Random Access Memory, P-RAM or just Phase Change memory, PCM. It is based around a phenomenon where a form of chalcogenid glass changes is state or phase between an amorphous state (high resistance) and a polycrystalline state (low resistance). It is possible to detect the state of an individual cell and hence use this for data storage. Currently this type of memory has not been widely commercialized, but it is expected to be a competitor for flash memory.

PROM: This stands for Programmable Read Only Memory. It is a semiconductor memory which can only have data written to it once - the data written to it is permanent. These memories are bought in a blank format and they are programmed using a special PROM programmer. Typically a PROM will consist of an array of fusible links some of which are "blown" during the programming process to provide the required data pattern.

SDRAM: Synchronous DRAM. This form of semiconductor memory can run at faster speeds than conventional DRAM. It is synchronized to the clock of the processor and is capable of keeping two sets of memory addresses open simultaneously. By transferring data alternately from one set of addresses, and then the other, SDRAM cuts down on the delays associated with non-synchronous RAM, which must close one address bank before opening the next.

SRAM: Static Random Access Memory. This form of semiconductor memory gains its name from th fact that, unlike DRAM, the data does not need to be refreshed dynamically. It is able to support faste read and write times than DRAM (typically 10 ns against 60 ns for DRAM), and in addition its cycl time is much shorter because it does not need to pause between accesses. However it consumes mor power, is less dense and more expensive than DRAM. As a result of this it is normally used for caches while DRAM is used as the main semiconductor memory technology.

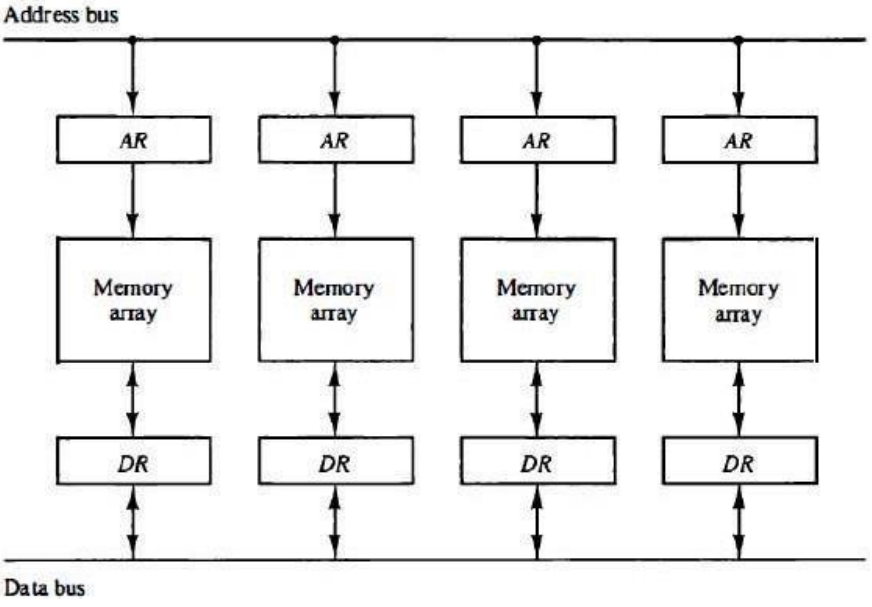
MEMORY ORGANIZATION

Memory Interleaving:

Pipeline and vector processors often require simultaneous access to memory from two or mor sources. An instruction pipeline may require the fetching of an instruction and an operand at the sam time from two different segments.

Similarly, an arithmetic pipeline usually requires two or more operands to enter the pipeline a the same time. Instead of using two memory buses for simultaneous access, the memory can b partitioned into a number of modules connected to a common memory address and data buses. A memory module is a memory array together with its own address and data registers. Figure 9-13 shows memory unit with four modules. Each memory array has its own address register AR and data registe DR.

Figure 9-13 Multiple module memory organization.



The address registers receive information from a common address bus and the data register communicate with a bidirectional data bus. The two least significant bits of the address can be used to distinguish between the four modules. The modular system permits one module to initiate a memory access while other modules are in the process of reading or writing a word and each module can honor a memory request independent of the state of the other modules.

The advantage of a modular memory is that it allows the use of a technique called interleaving In an interleaved memory, different sets of addresses are assigned to different memory modules. For example, in a two-module memory system, the even addresses may be in one module and the odd addresses in the other.

Concept of Hierarchical Memory Organization

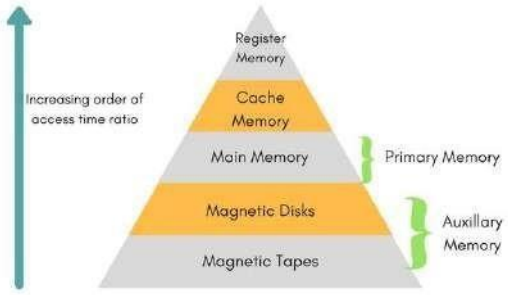
This Memory Hierarchy Design is divided into 2 main types:

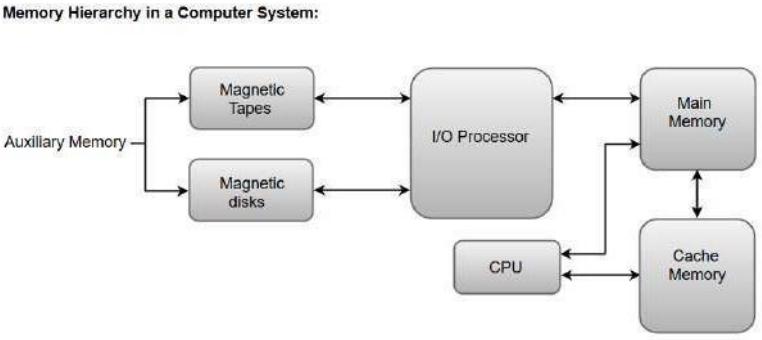
External Memory or Secondary Memory

Comprising of Magnetic Disk, Optical Disk, Magnetic Tape i.e. peripheral storage devices which are accessible by the processor via I/O Module.

Internal Memory or Primary Memory

Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by the processor.





Characteristics of Memory Hierarchy

Capacity:

It is the global volume of information the memory can store. As we move from top to bottom in the Hierarchy, the capacity increases.

Access Time:

It is the time interval between the read/write request and the availability of the data. As we move from top to bottom in the Hierarchy, the access time increases.

Performance:

Earlier when the computer system was designed without Memory Hierarchy design, the speed gap increases between the CPU registers and Main Memory due to large difference in access time. This results in lower performance of the system and thus, enhancement was required. This enhancement was made in the form of Memory Hierarchy Design because of which the performance of the system increases. One of the most significant ways to increase system performance is minimizing how far down the memory hierarchy one has to go to manipulate data.

Cost per bit:

As we move from bottom to top in the Hierarchy, the cost per bit increases i.e. Internal Memory is costlier than External Memory.

Cache Memories:

The cache is a small and very fast memory, interposed between the processor and the main memory. Its purpose is to make the main memory appear to the processor to be much faster than it actually is. The effectiveness of this approach is based on a property of computer programs called locality of reference.

Analysis of programs shows that most of their execution time is spent in routines in which many instructions are executed repeatedly. These instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other.

The cache memory can store a reasonable number of blocks at any given time, but this number i small compared to the total number of blocks in the main memory. The correspondence between th main memory blocks and those in the cache is specified by a mapping function.

When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the cache's *replacement algorithm*.

Cache Hits

The processor does not need to know explicitly about the existence of the cache. It simply issues Read and Write requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache.

If it does, the Read or Write operation is performed on the appropriate cache location. In this case, a *read* or *write hit* is said to have occurred.

Cache Misses

A Read operation for a word that is not in the cache constitutes a *Read miss*. It causes the block of words containing the requested word to be copied from the main memory into the cache.

Cache Mapping:

There are three different types of mapping used for the purpose of cache memory which are as follows: Direct mapping, Associative mapping, and Set-Associative mapping. These are explained as following below.

Direct mapping

The simplest way to determine cache locations in which to store memory blocks is the *direct mapping* technique. In this technique, block *j* of the main memory maps onto block *j* modulo 128 of th cache, as depicted in Figure 8.16. Thus, whenever one of the main memory blocks 0, 128, 256, . . . i loaded into the cache, it is stored in cache block 0. Blocks 1, 129, 257, . . . are stored in cache block 1 and so on. Since more than one memory block is mapped onto a given cache block position, contentio may arise for that position even when the cache is not full.

For example, instructions of a program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed, both of these blocks must be transferred to the block-position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block.

With direct mapping, the replacement algorithm is trivial. Placement of a block in the cache is determined by its memory address. The memory address can be divided into three fields, as shown in Figure 8.16. The low-order 4 bits select one of 16 words in a block.

When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache.

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

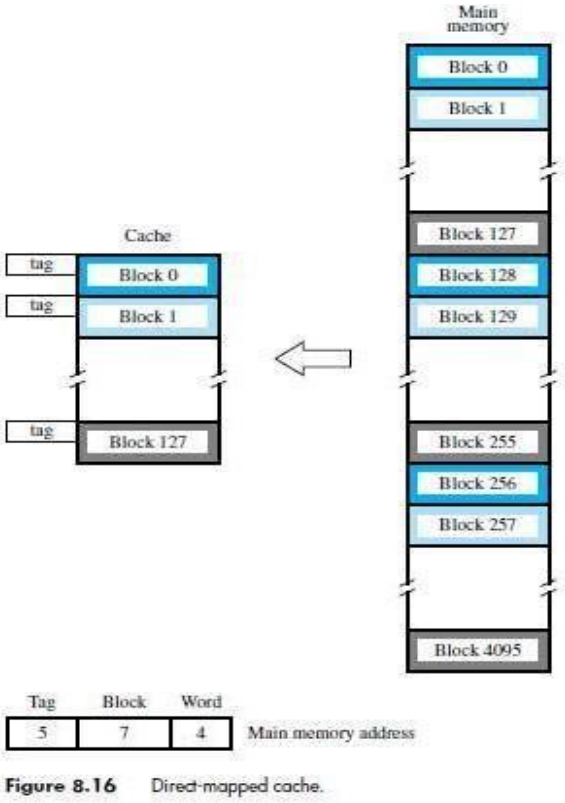


Figure 8.16 Direct-mapped cache.

Associative Mapping

In Associative mapping method, in which a main memory block can be placed into any cach block position. In this case, 12 tag bits are required to identify a memory block when it is resident in th cache. The tag bits of an address received from the processor are compared to the tag bits of each bloc of the cache to see if the desired block is present. This is called the *associative-mapping* technique.

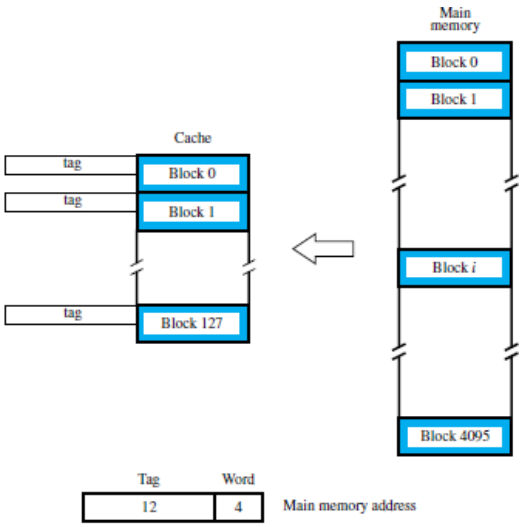


Figure 8.17 Associative-mapped cache.

It gives complete freedom in choosing the cache location in which to place the memory block, resulting in a more efficient use of the space in the cache. When a new block is brought into the cache, i replaces (ejects) an existing block only if the cache is full. In this case, we need an algorithm to selec the block to be replaced.

To avoid a long delay, the tags must be searched in parallel. A search of this kind is called an *associative search*.

Set-Associative Mapping

Another approach is to use a combination of the direct- and associative-mapping techniques. The blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement.

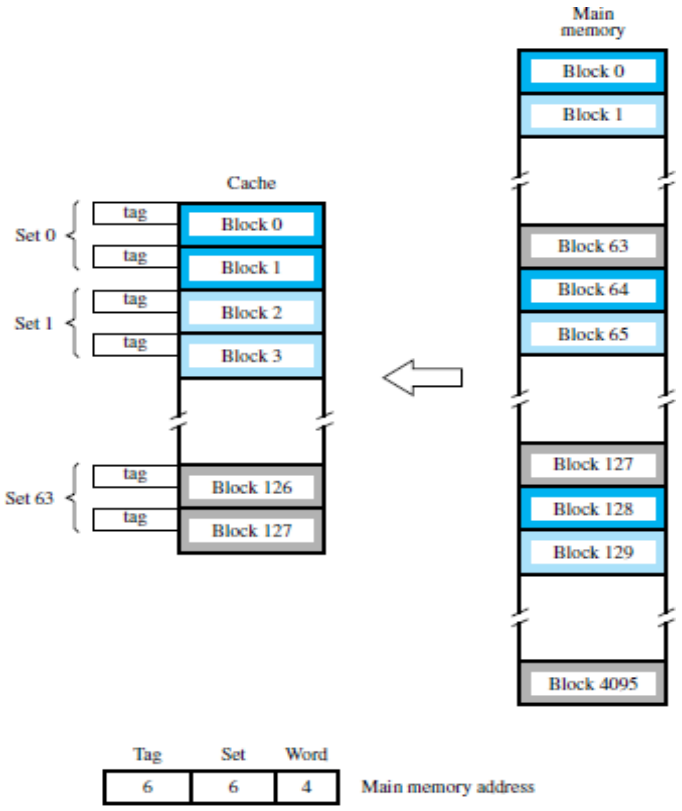


Figure 8.18 Set-associative-mapped cache with two blocks per set.

At the same time, the hardware cost is reduced by decreasing the size of the associative search. An example of this *set-associative-mapping* technique is shown in Figure 8.18 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, . . . , 4032 map into cache set 0, and they can occupy either of the two block positions within this set.

Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

The number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer. For the main memory and cache sizes in Figure 8.18, four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully-associative technique with 12 tag bits. The other extreme of one block per set is the direct-mapping.

Replacement Algorithms

In a direct-mapped cache, the position of each block is predetermined by its address; hence, the replacement strategy is trivial. In associative and set-associative caches there exists some flexibility.

When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite.

This is an important issue, because the decision can be a strong determining factor in system performance. In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. But, it is not easy to determine which blocks are about to be referenced.

The property of locality of reference in programs gives a clue to a reasonable strategy. Because program execution usually stays in localized areas for reasonable periods of time, there is a high probability that the blocks that have been referenced recently will be referenced again soon. Therefore, when a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the *least recently used* (LRU) block, and the technique is called the *LRU replacement algorithm*.

The LRU algorithm has been used extensively. Although it performs well for many access patterns, it can lead to poor performance in some cases.

Write Policies

The write operation is proceeding in 2 ways.

- Write-through protocol
- Write-back protocol

Write-through protocol:

Here the cache location and the main memory locations are updated simultaneously.

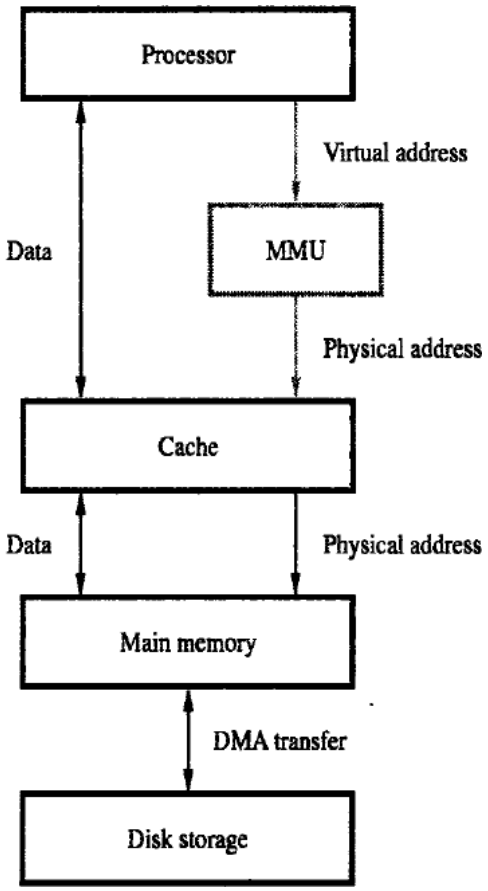
Write-back protocol:

- This technique is to update only the cache location and to mark it as withassociated flag bit called dirty/modified bit.
- The word in the main memory will be updated later, when the block containing thismarked word is to be removed from the cache to make room for a new block.
- To overcome the read miss Load –through / Early restart protocol is used.

Virtual Memory Management /Paging:

In most modern computer systems, the physical main memory is not as large as the address space spanned by an address issued by the processor. For example, a processor that issues 32-bit addresses has an addressable space of 4G bytes. The size of the main memory in a typical computer ranges from a few hundred megabytes to 1G bytes. When a program does not completely fit into the main memory, the parts of it not currently being executed are stored on secondary storage devices, such as magnetic disks. Of course, all parts of a program that are eventually executed are first brought into the

Fig: Virtual Memory Organization



main memory. When a new segment of a program is to be moved into a full memory, it must replace another segment already in the memory. In modern computers, the operating system moves programs and data automatically between the main memory and secondary storage. Thus, the application programmer does not need to be aware of limitations imposed by the available main memory.

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

Figure 5.26 shows a typical organization that implements virtual memory. A special hardware unit, called the *Memory Management Unit* (MMU), translates virtual

addresses into physical addresses. When the desired data (or instructions) are in the main memory, these data are fetched as described in our presentation of the cache mechanism. If the data are not in the main memory, the MMU causes the operating system to bring the data into the memory from the disk. Transfer of data between the disk and the main memory is performed using the DMA scheme discussed in

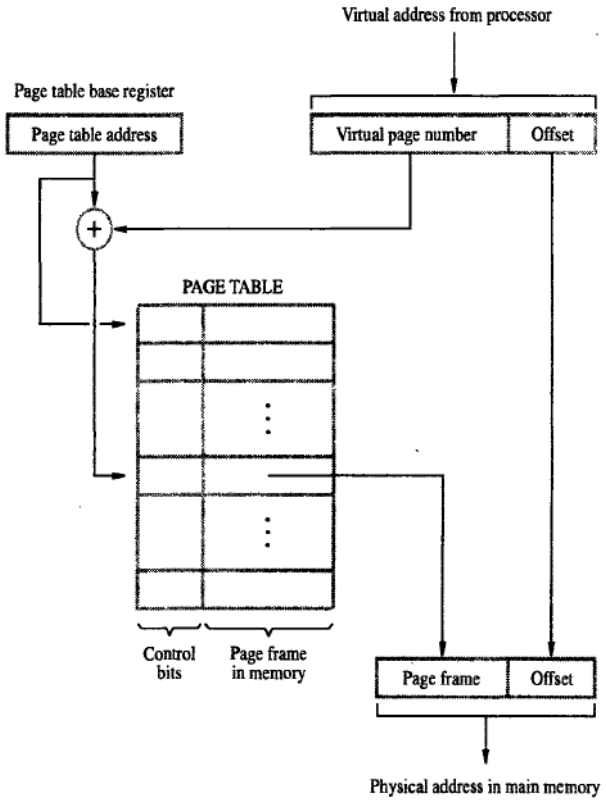


Fig: Virtual Memory Address Translation

Internal Memory Organization:

Memory cells are usually organized in the form of an array, in which each cell is capable of storing one bit of information. Apossible organization is illustrated in Figure below:

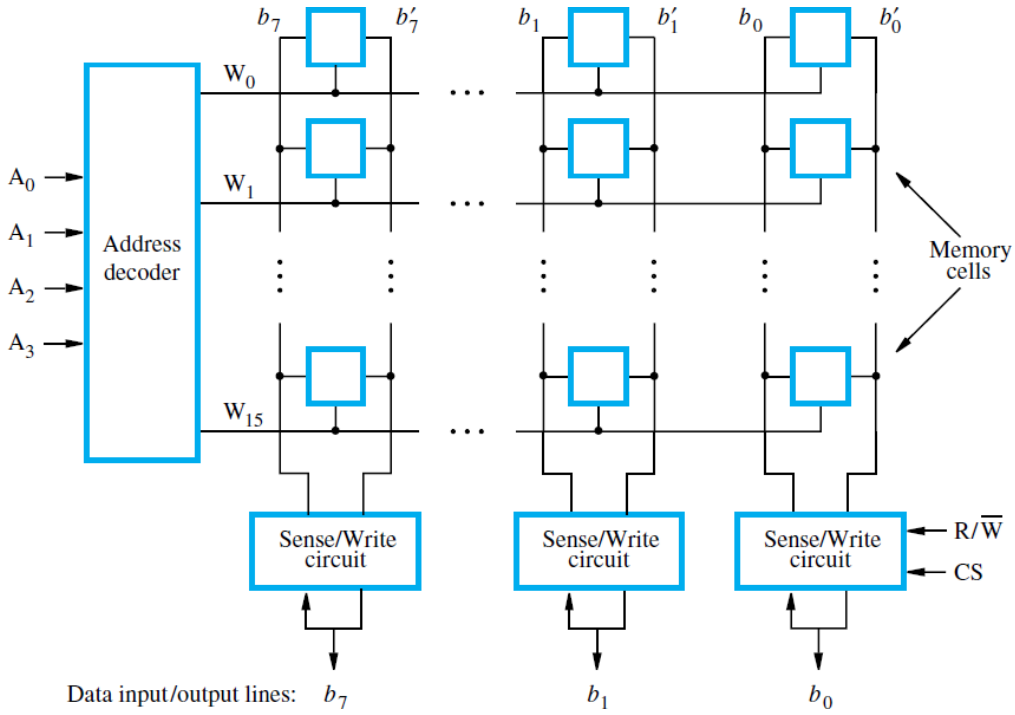


Fig: Organization of bit cells in a memory chip.

Each row of cells constitutes a memory word, and **all cells of a row are connected to a common line referred to as the word line**, which is driven by the address decoder on the chip. The **cells in each column are connected to a Sense/Write circuit by two bit lines**, and the Sense/Write circuits are connected to the data input/output lines of the chip. **During a Read** operation, these **circuits sense, or read, the information stored in the cells selected by a word line** and place this information on the output data lines. **During a Write operation, the Sense/Write circuits receive input data and store them in the cells of the selected word**. Above figures an example of a very small memory circuit consisting of 16 words of 8 bits each. This is referred to as a 16 × 8 organization. The data input and the data output of each Sense/Write circuit are connected to a single bidirectional data line that can be connected to the data lines of a computer. Two control lines, R/W and CS, are provided. The R/W(Read/Write) input specifies the required operation, and the CS (Chip Select) input selects given chip in a multichip memory system. **The above memory circuit stores 128 bits and requires 14 external connections for address, data, and control lines.** It also needs two lines for power supply and ground connections.

Consider now a slightly larger memory circuit, one that has 1K (1024) memory cells. This **circuit can be organized as a 128 × 8 memory, requiring a total of 19 external connections**. Alternatively, the same number of cells can be organized into a 1K×1 format. In this case, a 10-bit address is needed, but there is only one data line, resulting in 15 external connections. Figure below shows such an organization. The required 10-bit address is divided into two groups of 5 bits each to form the row and column addresses for the cell array. Arrow address selects a row of 32 cells, all of which are accessed in parallel. But, only one of these cells is connected to the external data line, based on the column address. For example, a 1Giga-bitchip may have a 256M × 4 organization, in which case a 28-bit address is needed and 4bits are transferred to or from the chip.

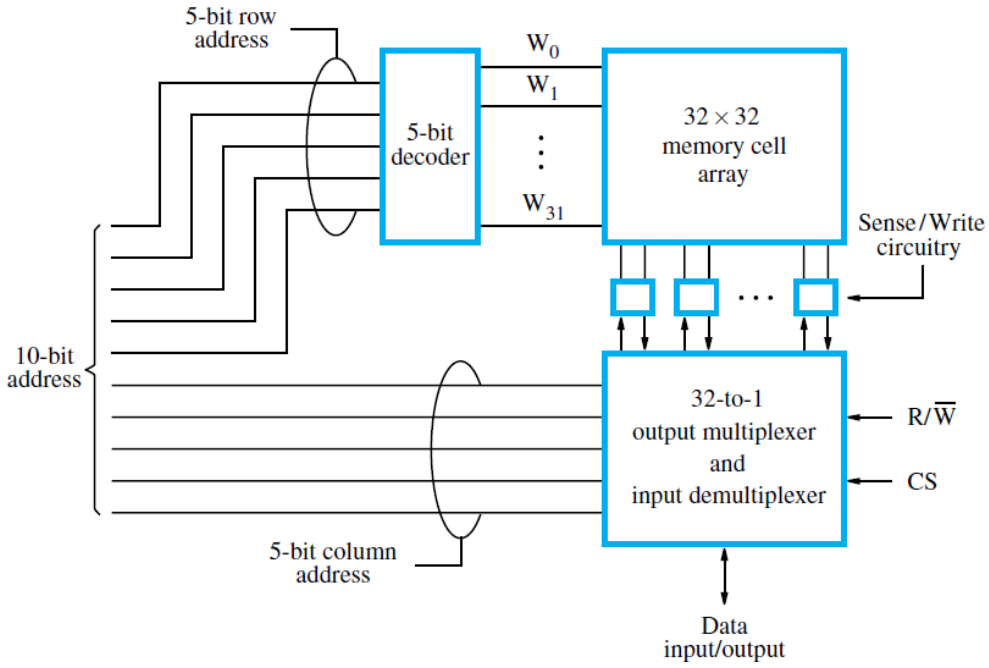


Fig: Organization of a 1K × 1 memory chip.

Memory Hierarchy

- The total memory capacity of a computer can be visualized as being a hierarchy of components.

The memory hierarchy system consists of all storage devices employed in a computer system

from the slow but high-capacity secondary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high-speed processing logic.

- The purpose of any memory device is to store programs and data. Several types of memory devices are used in the computer forming a Memory Hierarchy. Each plays a specific role contributing to the speed, cost effectiveness, portability etc.

Main Memory

The memory unit that communicates directly with the CPU is called the main memory. It comprises of RAM and ROM, both are Semi-Conductor memories. (chip memories). ROM is non-volatile. It is used is storing permanent information like the BIOS program. It is typically of 2 MB - 4 MB in size.RAM is writable and hence is used for day-to-day operations. Every file that we access from secondary memory, is first loaded into RAM. The main memory occupies
a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an UO processor. To provide large amount of working space RAM is typically 4 GB - 8 GB.

Secondary Memory (Auxiliary Memory):

Devices that provide backup storage are called auxiliary memory. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. They are used for storing system programs, large data files, and other backup information. Only pro-grams and data currently needed by the processor reside in main memory. When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. The main purpose of Secondary Memory is to increase the storage capacity, at low cost. Its biggest component is the Hard Disk. It is writeable as well as non-volatile. Typical size of a HD is 1 TB. Disk memories are much slower than chip memories but are also much cheaper.

Cache Memory:

The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory. A technique used to compensate for the mismatch in operating speeds is to employ an extremely fast, small cache between the CPU and main memory whose access time Is dose to processor logic dock cycle time. It is the fastest form of memory as it uses SRAM (Static RAM).The Main Memory uses DRAM (Dynamic RAM).SRAM uses flip-flops and hence is much faster than DRAM which uses capacitors. But SRAM is also very expensive as compared to DRAM. Hence only the current portion of the file we need to access is copied from Main Memory (DRAM)to Cache memory (SRAM), to be directly accessed by the processor. This gives maximum performance and yet keeps the cost low. While the VO processor manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU .Typical size of Cache is around 2 MB – 8MB. If code and data are in the same cache then it is unified cache else its called split cache. Depending upon the location of cache, it is of three types: L1, L2 and L3.L1 cache is present inside the processor and is a split cache typically 4-8 KB.L2 is present on the same die as the processor and is a unified cache typically 1 MB.L3 is present outside the processor. It is also unified and is typically of 2-8 MB.

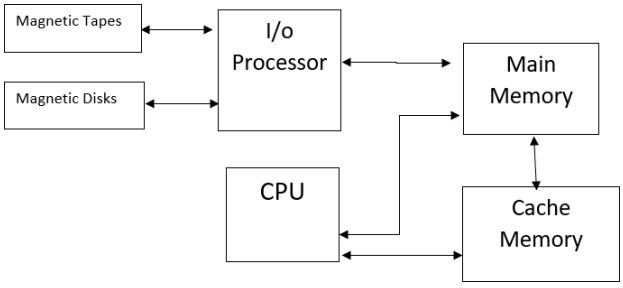


Fig: Memory Hierarchy in a computer system

The reason for having two or three levels of memory hierarchy is economics. As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer. The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed. Thus as the memory access speed increases, so does its relative cost. The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system.

Memory Interleaving

Pipeline and vector processors often require simultaneous access to memory from two or more sources. An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments. Similarly, an arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time. Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses. A memory module is a memory array together with its own address and data registers. Figure below shows a memory unit with four modules.

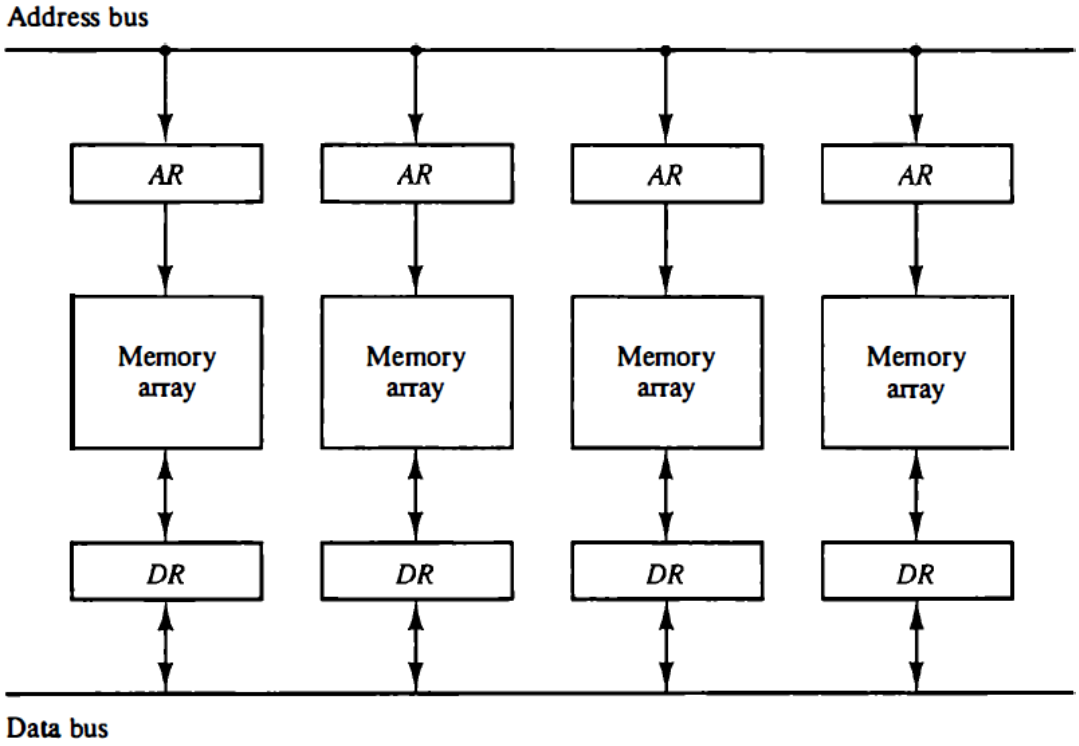


Fig: Multiple module memory organization.

Each memory array has its own address register AR and data register DR . The address registers receive information from a common address bus and the data registers communicate with a bidirectional data bus. The two least significant bits of the address can be used to distinguish between the four modules. The modular system permits one module to initiate a memory access while other modules are in the process of reading or writing a word and each module can honor a memory request independent of the state of the other modules. The advantage of a modular memory is that it allows the use of a technique called interleaving. In an interleaved memory, different sets of addresses are assigned to different memory modules. For example, in a two-module memory system, the even addresses may be in one module and the odd addresses in the other. When the number of modules is a power of 2, the least significant bits of the address select a memory module and the remaining bits designate the specific location to be accessed within the selected module. A modular memory is useful in systems with pipeline and vector processing. A vector processor that uses an n-way interleaved memory can fetch operands

from n different modules. By staggering the memory access, the effective memory cycle time can be reduced by

a factor close to the number of modules. A CPU with instruction pipeline can take advantage of multiple memory modules so that each segment in the pipeline can access memory independent of memory access from other segments.

Cache Memory:

The cache is a small and very fast memory, interposed between the processor and the main memory. Its purpose is to make the main memory appear to the processor to be much faster than it actually is.

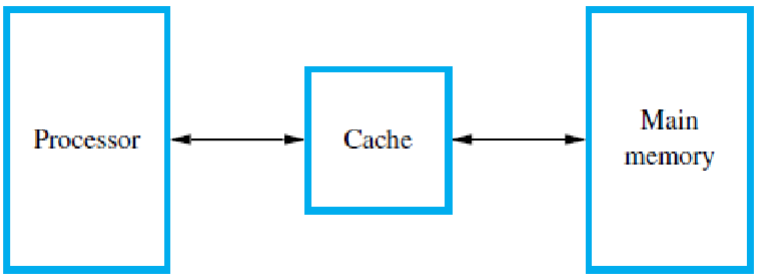


Fig: Use of Cache Memory

operation of a cache memory is very simple. The memory control circuitry is designed to take advantage of the property of locality of reference. Temporal locality suggests that whenever an information item, instruction or data, is first needed, this item should be brought into the cache, because it is likely to be needed again soon. Spatial locality suggests that instead of fetching just one item from the main memory to the cache, it is useful to fetch several items that are located at adjacent addresses as well. The termcache block refers to a set of contiguous address locations of some size. Another term that is often used to refer to a cache block is a cache line**.**

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

Cache Hits:

The processor does not need to know explicitly about the existence of the cache. It simply issues Read andWrite requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache. If it does, the Read or Write operation is performed on the appropriate cache location. In this case, a read or write hit is said to have occurred. The main memory is not involved when there is a cache hit in a Read operation. For a Write operation, the system can proceed in one of two ways. In the first technique, called the write-through protocol, both the cache location and the main memory location are updated. The second technique is to update only the cache location and to mark the block containing it with an associated flag bit, often called the dirty or modified bit. The main memory location of the word is updated later, when the block containing this marked word is removed from the cache to make room fora new block. This technique is known as the write-back, or copy-back, protocol.

The write-through protocol is simpler than the write-back protocol, but it results in unnecessary Write operations in the main memory when a given cache word is updated several times during its cache residency. The write-back protocol also involves unnecessary Write operations, because all words of the block are eventually written back, even if only a single word has been changed while the block was in the cache. The write-back protocol is used most often, to take advantage of the high speed with which data blocks can be transferred to memory chips.

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

When a Write miss occurs in a computer that uses the write-through protocol, the information is written directly into the main memory. For the write-back protocol, the block containing the addressed word is first brought into the cache, and then the desired word in the cache is overwritten with the new information.

Peripheral Devices and Their Characteristics

I/o Device Interface:

Introduction: Input-output interface ***provides a method for transferring information between internal storage and external I/0 devices.*** **Peripherals**(I/O Devices) connected to a computer ***need special communication links for interfacing them with the central processing unit.*** The purpose of the communication link is to resolve the differences that exist between the central computer and each peripheral.

The major differences are:

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a ***conversion of signal values*** may be required.

2. The ***data transfer rate*** of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.

3. ***Data codes and formats*** in peripherals ***differ*** from the word format in the CPU and memory.

4. The ***operating modes of peripherals*** are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include ***special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers.*** These components are called ***interface units*** because they interface ***between the processor bus and the peripheral device***. In addition, ***each device may have its own controller*** that supervises the operations of the particular mechanism in the peripheral.

l/O Bus and Interface Modules:

A typical communication link between the processor and several peripherals is shown below:

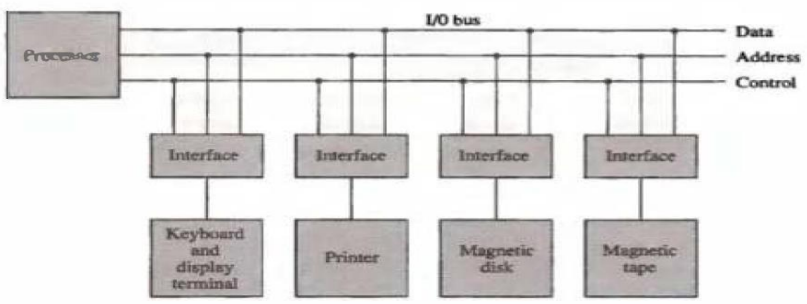


Fig: Connection of I/O bus to input-output devices.

The I/0 bus consists of data lines, address lines, and control lines. ***Each peripheral device has associated with it an interface unit.***

Each ***interface decodes the address and control received from the I/O bus***, interprets them for the peripheral, and ***provides signals for the peripheral controller***. It also ***synchronizes the data flow and supervises the transfer*** between peripheral and processor.

Each ***peripheral has its own controller that*** operates the particular electromechanical device. For example, the ***printer controller controls the paper motion, the print timing, and the selection of printing characters***. A controller may be housed separately or may be physically integrated with the peripheral.

The I/O bus from the processor is attached to all peripheral interfaces. ***To communicate with a particular device***, the processor places a device address on the address lines. ***Each interface attached to the I/0 bus contains an address decoder that monitors the address lines***. ***When the interface detects its own address, it activates the path between the bus lines and the device that it controls***. All peripherals whose address does not correspond to the address in the bus are disabled by their interface.

At the same time that the address is made available in the address lines, the ***processor provides a function code in the control lines. The interface selected responds to the function code and proceeds to execute*** it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit. There are **four types of commands that an interface may receive**. They are classified as control, status, data output, and data input.

- A ***control command*** is issued to ***activate the peripheral*** and to inform it what to do.
- A ***status command*** is used to test various status conditions in the interface and the peripheral. For Example, the computer may wish to check the status of the peripheral before a transfer is initiated.
- A ***data output command*** causes the interface to respond by transferring ***data from the bus into one of its registers.***
- The ***data input command*** is the opposite of the data output. In this case the ***interface receives an item of data from the peripheral and places it in its buffer register.*** The processor checks if data are available by means of a status command and then issues a data input command. ***The interface places the data on the data lines, where they are accepted by the processor.***

There are ***three ways that computer buses can be used to communicate with memory and I/O:***

1. Use two separate buses, one for memory and the other for I/O. (This method uses a separate I/O Processor alongside CPU to provide an independent pathway for the transfer of information between external devices and internal memory.)

2. Use one common bus for both memory and I/O but have separate control lines for each.

3. Use one common bus for memory and I/O with common control lines.

Isolated I/O: Many computers use one common bus to transfer information between memory or I/0 and the CPU. The distinction between a memory transfer and I/0 transfer is made through ***separate read and write lines***.

The ***CPU specifies whether the address on the address lines*** is for a memory word or for an interface register by ***enabling one of two possible read or write lines***. ***The I/0 read and I/0 write control lines are enabled during an I/0 transfer. The memory read and memory write control lines are enabled during a memory transfer***. This configuration isolates all I/0 interface addresses from the addresses assigned to memory and is referred to as the isolated I/0 method for assigning addresses in a common bus. In the isolated I/0 configuration, the CPU has distinct ***input and output instructions***, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/0 read (for input) or I/0 write (for output) control line. This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word.

Memory-mapped I/O: **Memory mapped I/O** uses the same address space for both memory and I/O. This is the case in computers that ***employ only one set of read and write signals and do not distinguish between memory and I/O addresses***. This configuration is referred to as memory-mapped I/O. In a memory-mapped l/O organization ***there are no specific input or output instructions.*** The ***CPU can manipulate l/O data residing in interface registers with the same instructions that are used to manipulate memory words***. Each interface is organized as a set of registers that respond to read and write requests in the normal address space.

Computers with memory-mapped l/O can ***use memory-type instructions to access l/0 data***. It allows the computer to use the same instructions for either input-output transfers or for memory transfers. The ***advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from l/O registers.*** In a typical computer, there are more memory-reference instructions than l/O instructions. With memory-mapped l/O all instructions that refer to memory are also available for l/O.

DATA TRANSFER MODES:

The transfer of data between two units may be done in parallel or serial. In ***parallel data transmission, each bit of the message has its own path*** and the total message is transmitted at the same time. This means that an n-bit message must be transmitted through n separate conductor paths. In serial data transmission, each bit in the message is sent in sequence one at a time. ***This method requires the use of one pair of conductors or one conductor and a common ground.*** Parallel transmission is faster but requires many wires. It is **used for short distances and where speed is important**. Serial transmission is slower but is less expensive since it requires only one pair of conductors.

Serial transmission can be **synchronous or asynchronous**.

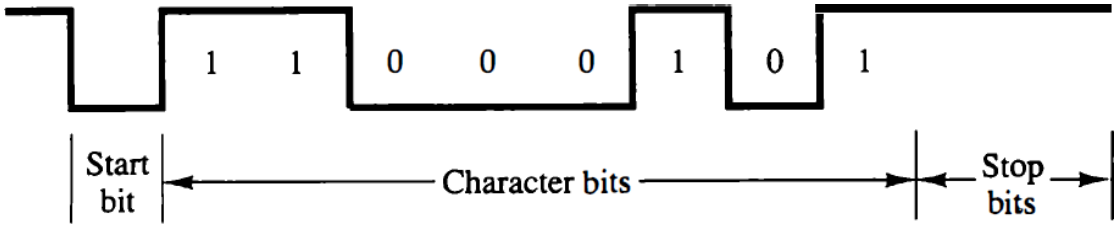
Synchronous Data Transfer: In synchronous transmission, the two units (***Sending Unit and Receiving unit) share a common clock frequency*** and bits are transmitted continuously at the rate dictated by the clock pulses. In long distant serial transmission, each unit is driven by a separate clock of the same frequency. *Synchronization signals are transmitted periodically between the two units to keep their clocks in step with each other.*

In asynchronous transmission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted. This is in contrast to synchronous transmission, ***where bits must be transmitted continuously to keep the clock frequency in both units synchronized with each other.***

Eg:Any two units of a digital system are designed independently, such as CPU and I/O interface. If the registers in the I/O interface share a common clock with CPU registers, then transfer between the two units is said to be synchronous.

Asynchronous Serial Transfer: A serial asynchronous data transmission technique used in many interactive terminals ***employs special bits that are inserted at both ends of the character code***. With this technique, each character consists of three parts: a start bit, the character bits, and stop bits. The convention is that the transmitter rests at the 1-state when no characters are transmitted. The first bit, called the ***start bit, is always a 0 and is used to indicate the beginning of a character.*** The last bit called the stop bit is always a 1.

An example of this format is shown below:



A transmitted character can be detected by the receiver from knowledge of the transmission rules:

1 . When a character is not being sent, the line is kept in the 1-state.

2. The initiation of a character transmission is detected from the start bit, which is always 0.

3. The character bits always follow the start bit.

4. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.

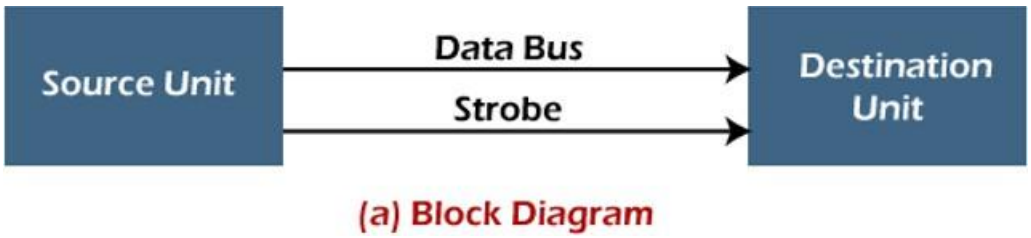
Asynchronous way of data transfer can be achieved using two methods:

1) Strobe control

2) Handshaking

Strobe Control Method: The Strobe Control method of asynchronous data transfer employs a single control line to time each transfer. This control line is also known as a strobe, and it may be achieved either by source or destination, depending on which initiate the transfer.

Source initiated strobe: In the below block diagram, strobe is initiated by source, and as shown in the timing diagram, the source unit first places the data on the data bus.



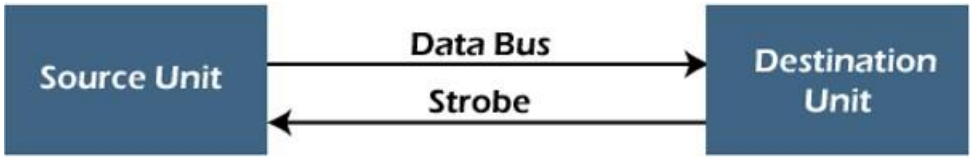
(a) Block Diagram

(b) Timing Diagram

After a brief delay, the source activates a strobe pulse. The information on the data bus and strobe control signal remains in the active state for a sufficient time to allow the destination unit to receive the data. **The destination unit uses a falling edge of strobe control to transfer the contents of a data bus to one of its internal registers.** The source removes the data from the data bus after it disables its strobe pulse. Thus, new valid data will be available only after the strobe is enabled again.

Example: The strobe may be a memory-write control signal from the CPU to a memory unit.

Destination initiated strobe: In the below block diagram, the strobe initiated by destination, and in the timing diagram, the destination unit first activates the strobe pulse, informing the source to provide the data.



(a) Block Diagram

(b) Timing Diagram

The falling edge of the strobe pulse can use again to trigger a destination register. The destination unit then disables the strobe. Finally, and source removes the data from the data bus after a determined time interval.

Example: the strobe may be a memory read control from the memory unit to CPU.

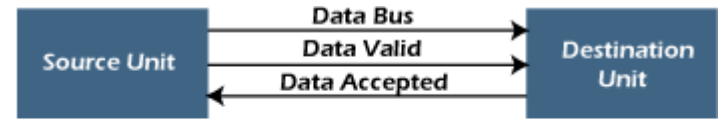
Handshaking Method: The strobe method has the disadvantage that the source unit that initiates the transfer has no way of knowing whether the destination has received the data that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has placed data on the bus.

So this problem is solved by the handshaking method. The handshaking method introduces a second control signal line.

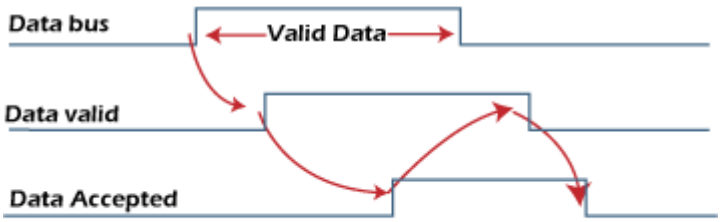
In this method, one control line is in the same direction as the data flow in the bus from the source to the destination. The source unit uses it to inform the destination unit whether there are valid data in the bus.

The other control line is in the other direction from the destination to the source. This is because the destination unit uses it to inform the source whether it can accept data. And in it also, the sequence of control depends on the unit that initiates the transfer. So it means the sequence of control depends on whether the transfer is initiated by source and destination.

Source initiated handshaking: In the below block diagram, two handshaking lines are "data valid", which is generated by the source unit, and "data accepted", generated by the destination unit.



(a) Block Diagram



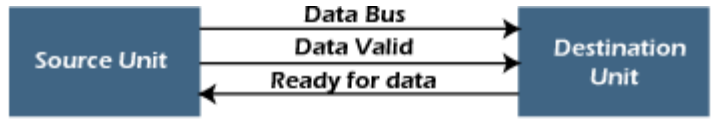
(b) Timing Diagram

The timing diagram shows the timing relationship of the exchange of signals between the two units. The source initiates a transfer by placing data on the bus and enabling its data valid signal. The destination unit then activates the data accepted signal after it accepts the data from the bus.

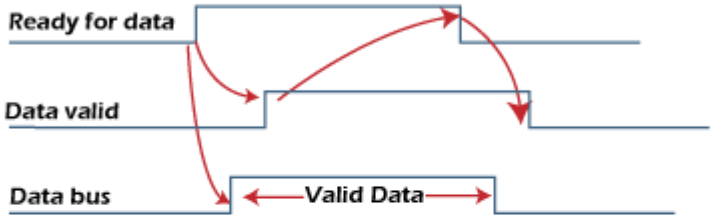
The source unit then disables its valid data signal, which invalidates the data on the bus. After this, the destination unit disables its data accepted signal, and the system goes into its initial state. The source unit does not send the next data item until after the destination unit shows

readiness to accept new data by disabling the data accepted signal. This sequence of events described in its sequence diagram, which shows the above sequence in which the system is present at any given time.

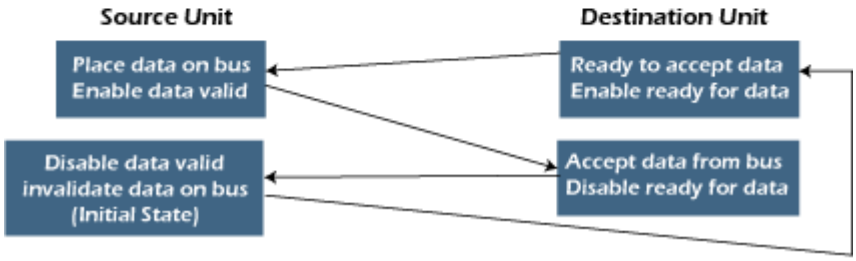
Destination initiated handshaking: In the below block diagram, the two handshaking lines are "**data valid**", generated by the source unit, and "**ready for data**" generated by the destination unit.



(a) Block Diagram



(b) Timing Diagram



(c) Sequence Diagram (Sequence of events)

I/O Transfers:

Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/0 instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit. Data transfer between the central computer and I/0 devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; others transfer the data directly to and from the memory unit. Data transfer to and from peripherals may be handled in one of three possible modes:

1) **Program Controlled I/O**
2) **Interrupt-initiated I/0**
3) **Direct memory access (DMA)**

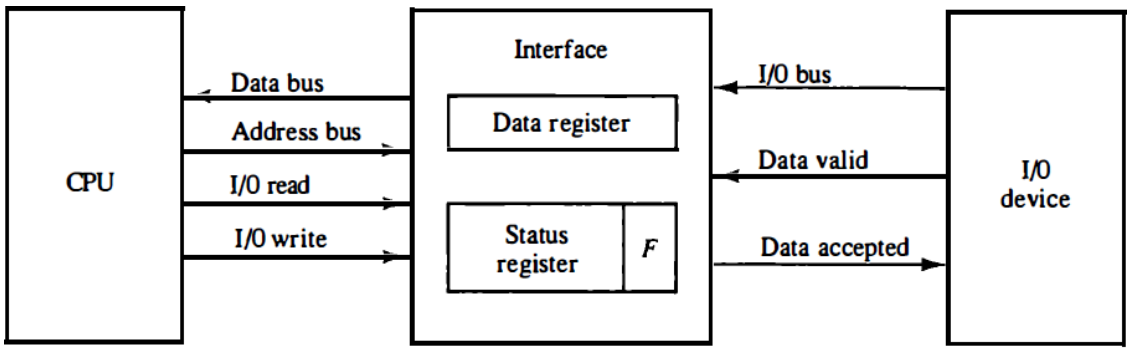
1. ***Program Controlled I/O:*** Programmed I/0 operations are the result of I/0 instructions written in the computer program. ***CPU executes a program that transfers data between I/O device and memory.*** Each data item transfer is initiated by an instruction in the program. **Usually, the transfer is to and from a CPU register and peripheral.** ***Other***

instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. In the programmed I/0 method, the CPU stays in a program loop until the I/0 unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly.

Example of Programmed I/0:

In the programmed I/0 method, the I/0 device does not have direct access to memory. A transfer from an I/0 device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU and a store instruction to transfer the data from the CPU to memory. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.

An example of data transfer from an I/O device through an interface into the CPU is shown in Fig below:



F = Flag bit

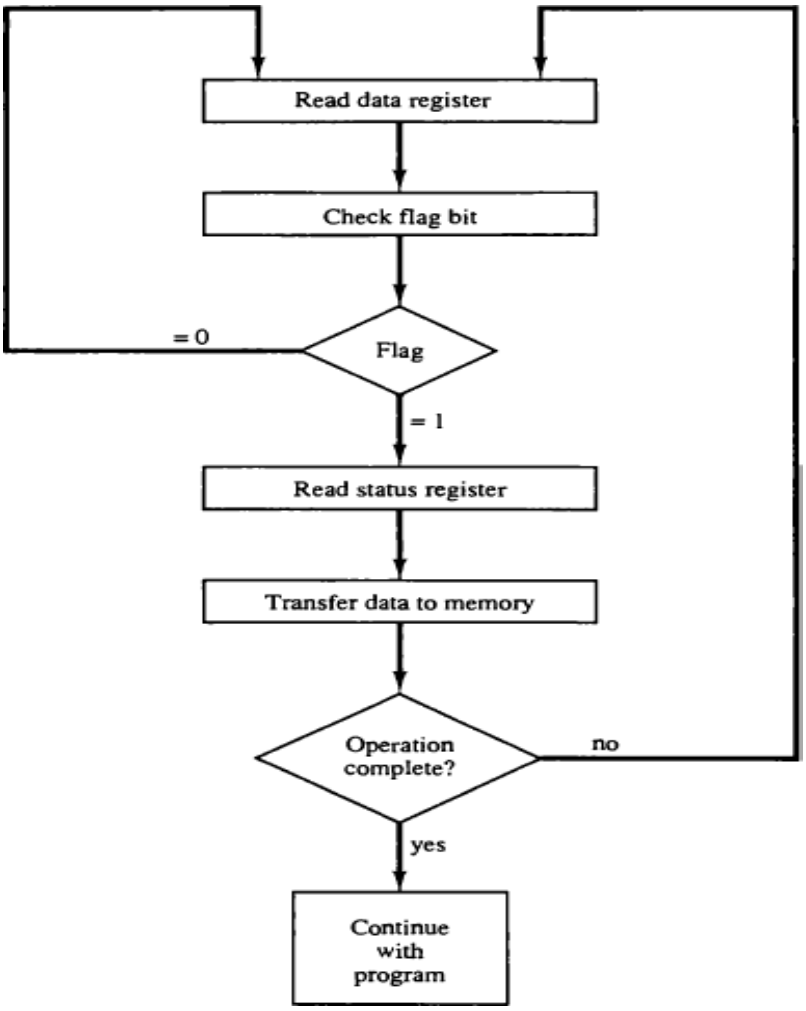
The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a bit in the status register that we will refer to as an F or "flag" bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface.

A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the VO device. This is done by reading the status register into a CPU register and checking the value of the flag bit. If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte. A flowchart of the program that must be written for the CPU is shown in Fig below. It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

1 Read the status register.

2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.

3. Read the data register.



Each byte is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words from an I/O device and store them in a memory buffer. The programmed VO method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously.

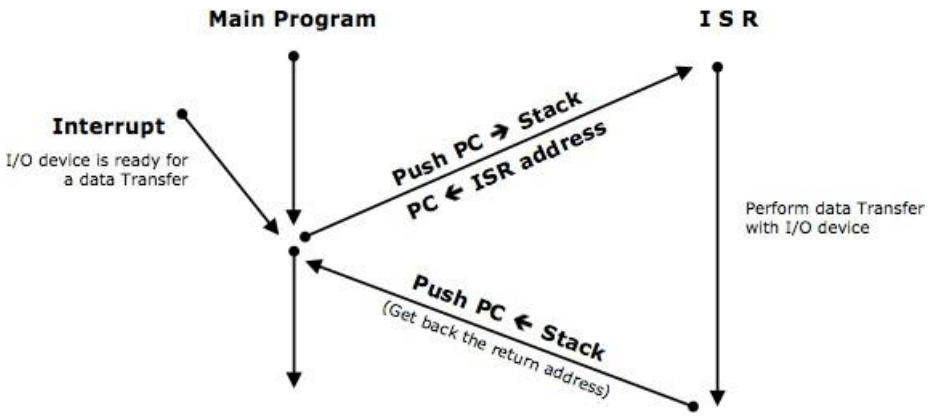
2) INTERRUPT DRIVEN I/O:

1) In interrupt driven I/O, the transfer is not initiated by the processor.
2) Instead, **an I/O device which wants to perform a data transfer with the processor, must give an interrupt to the processor.**
3) An interrupt is a condition that makes the processor execute an ISR (Interrupt Service Routine).
4) In the ISR, processor will perform data Transfer with the I/O device.
5) This relieves the processor from periodically checking the status of every I/O device thereby saves as
lot of time of the processor.
6) **The processor is free to carry on its own operations.**
7) Whenever a device wants to transfer data, it will interrupt the processor.
8) This is how many I/O devices Transfer data with the processor.

9) E.g.:: **Keyboard.** Instead of the processor checking all the time, whether a key is pressed, the
keyboard interrupts the processor as an when we press a key. In the ISR of the keyboard, which is a
part of the keyboard driver software, the processor will read the data from the keyboard.
10) **Hence interrupt driven I/O is much better than Polled I/O (Programmed I/O).**

INTERRUPT HANDLING MECHANISM

1) When an interrupt occurs, processor, firstly, finishes the current instruction.
2) It then suspends the current program and executes an ISR.
3) To do so, it Pushes the value of PC (address of next instruction), into the stack.
4) Now it loads the ISR address into PC and proceeds to execute the ISR.
5) At the end of the ISR, it POPs the return address from the stack and loads it back into PC.
6) This is how the processor return to the very next instruction in the program.



	INTERRUPT DRIVEN I/O	POLLING (PROGRAMMED I/O)
1	**I/O device interrupts the processor whenever it wants to perform a data Transfer.**	**Processor periodically checks (polls) the status of every I/O device to know if it wants to perform a data Transfer.**
2	**Processor is free** to carry on its own operations, hence **saves system time**.	**Processor is busy** in constantly checking all I/O devices, hence **system time wasted**.
3	**Additional hardware required** to handle interrupts. E.g.:: 8259 Programmable interrupt controller.	Additional hardware **not required**.
4	Increases **cost and complexity** of the system.	System is **cheaper and less complex**.
5	Interrupt **priority** has to be managed through software or through hardware.	No such issue.
6	Interrupt **vector addresses** (ISR Addresses) need to be stored in an Interrupt Vector table - **IVT**.	No such issue.

3) DMA BASED I/O

DMA means **transferring data directly between memory and I/O.**

DMA transfers are **very fast** as compared to Processor based transfers due to two reasons.

1. They are **hardware based** so no time is wasted in fetching and decoding instructions.
2. Transfers are **directly between memory and I/O** without data going via the Processor.

To Perform a DMA transfer we need a **DMA Controller like 8237/ 8257.**

It is capable of taking control of the buses from the Processor.

The process is performed as follows.

1) By Default **Processor is the bus master**.
2) The DMA transfer parameters first initialized by the processor.
3) **Processor programs two registers inside the DMAC called CAR and CWCR** giving the starting address and the number of bytes to be transferred.
4) DMAC now ensures that the I/O device is ready for the transfer by checking the DREQ signal.
5) **If DREQ=1, then DMAC gives HOLD signal** to the Processor requesting control of the system bus.
6) **Processor releases control of the bus** after finishing the current machine (bus) cycle.
7) Processor **gives HLDA** informing DMAC that it is now the bus master.
8) **DMAC issues DACK#** (by default active low, but can be changed) to I/O device indicating that the transfer is about to begin.
9) Now DMAC **transfers one byte in one cycle**.
10) After every byte is transferred the **Address register and Count register are decremented by 1**.
11) This repeats till Count reaches **"0"** also called **Terminal Count**.
12) Now the **transfer is complete**.
13) DMAC **returns the system bus to Processor by making HOLD = 0**.
14) Processor once again **becomes bus master**.

Advantage of DMA

DMA transfers are very fast.

Drawback of DMA

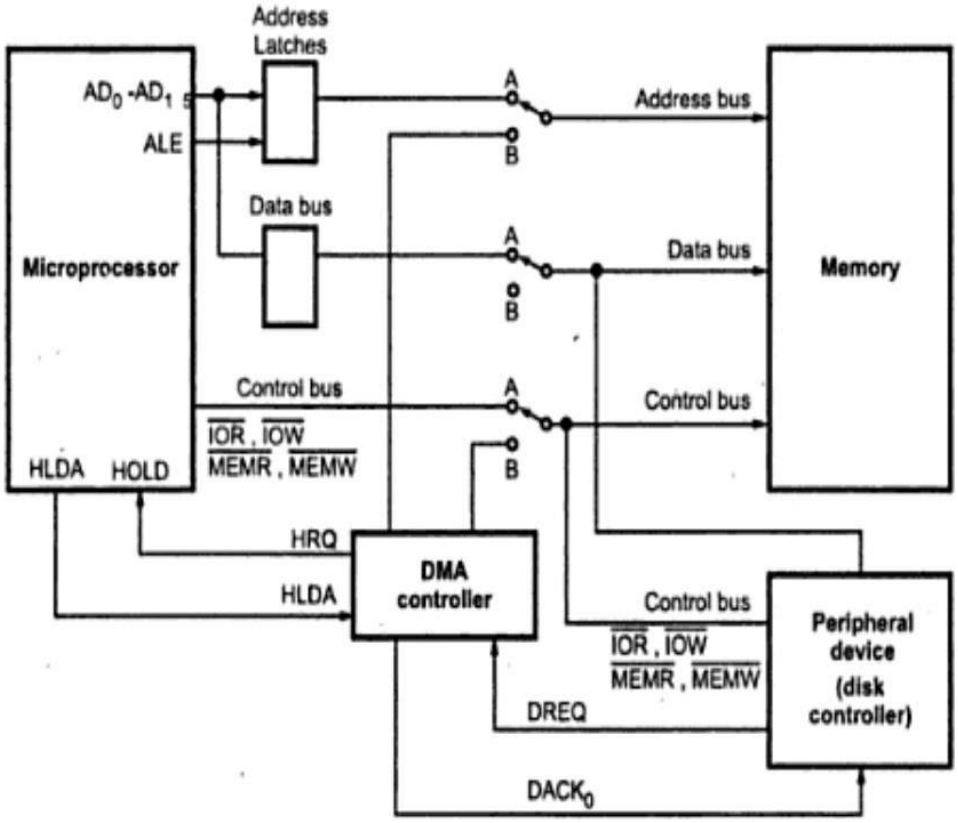
DMAC becomes the bus master. Hence during DMA cycles, the processor cannot perform any operations as the bus is already being used for DMA. The processor remains in HOLD state.

Difference between Interrupt Request and DMA request

When an interrupt occurs, the processor has to suspend the current program, execute the ISR and then **return to the next instruction** of the main program. Hence it is necessary that the **processor completes the current instruction** before servicing an interrupt request.

When a DMA request occurs, the processor has to simply relinquish (give away) control of the system bus and enter hold state. When ever it gets back the bus it can **resume from where it had left**. Hence the processor **need not finish the current instruction** before servicing a DMA request. It simply has to **finish the current machine cycle**. **Hence Instruction cycles are Interrupt Breakpoints and Machine cycles are DMA breakpoints.**

DMA Operation



TYPES / METHODS / TECHNIQUES OF DMA TRANSFERS

There are **four modes of data transfer**:

1) BLOCK TRANSFER MODE / BURST MODE.

In this mode, the DMAC is programmed to **transfer ALL THE BYTES** in one complete DMA operation. After a byte is transferred, the **CAR and CWCR are adjusted** accordingly. The **system bus** is **returned** to the **processor, ONLY after all the bytes are transferred.** It is the **fastest** form of DMA but keeps the **processor inactive** for a long time.

2) SINGLE BYTE TRANSFER MODE/ CYCLE STEALING.

Once the DMAC becomes the bus master, it will transfer only **ONE BYTE** and return the bus to the processor. As soon as the processor performs one bus cycle, DMAC will once again take the bus back from the processor. Hence both **DMAC and processor** are **constantly stealing bus cycles** from each other. It is the **most popular** method of DMA, because it keeps the **processor active in the background**. After a byte is transferred, the **CAR** and **CWCR** are **adjusted** accordingly.

3) DEMAND TRANSFER MODE.

It is very **similar to Block Transfer**, **except** that the **DREQ must remain active throughout the DMA operation**. **If** during the operation **DREQ goes low**, the **DMA operation is stopped** and the **busses** are **returned** to the **processor.**

In the meantime, the **processor** can **continue** with its own operations. **Once DREQ goes high again**, the **DMA operation continues** from where it had stopped. This means, the transfer happens on demand from the I/O device, hence the name.

4) HIDDEN MODE / TRANSPARENT MODE.

In this mode, **once the processor programs all parameters inside the DMAC, the DMAC** does not request the processor for the control of the bus. Instead, it observes the processor. **It waits for the processor to enter idle state**. Once the processor is idle, the DMAC will take control of the bus and perform the Transfer. So, the Transfer is **totally transparent to the processor** or hidden from the processor. Hence the name.

Interrupts and Exceptions:

Interrupt

The term Interrupt is usually reserved for hardware interrupts. They are program control interruptions caused by external hardware events. Here, external means external to the CPU. Hardware interrupts usually come from many different sources such as timer chip, peripheral devices (keyboards, mouse, etc.), I/O ports (serial, parallel, etc.), disk drives, CMOS clock, expansion cards (sound card, video card, etc). That means hardware interrupts almost never occur due to some event related to the executing program.

Example –

An event like a key press on the keyboard by the user, or an internal hardware timer timing out can raise this kind of interrupt and can inform the CPU that a certain device needs some attention. In a situation like that the CPU will stop whatever it was doing (i.e. pauses the current program), provides the service required by the device and will get back to the normal program. When hardware interrupts occur and the CPU starts the ISR, other hardware interrupts are disabled (e.g. in 80×86 machines). If you need other hardware interrupts to occur while the ISR is running, you need to do that explicitly by clearing the interrupt flag (with sti instruction). In 80×86 machines, clearing the interrupt flag will only affect hardware interrupts.

Exception

Exception is a software interrupt, which can be identified as a special handler routine. An exception occurs due to an "exceptional" condition that occurs during program execution.

Example –

Division by zero, execution of an illegal opcode or memory related fault could cause exceptions. Whenever an exception is raised, the CPU temporarily suspends the program it was executing and starts the ISR. ISR will contain what to do with the exception. It may correct the problem or if it is not possible it may abort the program gracefully by printing a suitable error message. Although a specific instruction does not cause an exception, an exception will always be caused by an instruction. For example, the division by zero error can only occur during the execution of the division instruction.

Exceptions and interrupts are unexpected events which will disrupt the normal flow of execution of instruction (that is currently executing by processor). An exception is an unexpected event from within the processor. Interrupt is an unexpected event from outside the processor. Whenever an exception or interrupt occurs, the hardware starts executing the code that performs an action in response to the exception. This action may involve killing a process, outputting an error message, communicating with an external device, or horribly crashing the entire computer system by initiating a "Blue Screen of Death" and halting the CPU. The instructions responsible for this action reside in the operating system kernel, and the code that performs this action is called the interrupt handler code. handler code is an operating system subroutine. Then, After the handler code is executed, it may be possible to continue execution after the instruction where the execution or interrupt occurred.

Whenever an exception or interrupt occurs, execution transitions from user mode to kernel mode where the exception or interrupt is handled. The following steps must be taken to handle an exception or interrupts:
While entering the kernel, the context (values of all CPU registers) of the currently executing process must first be saved to memory. The kernel is now ready to handle the exception/interrupt.

1) Determine the cause of the exception/interrupt.
2) Handle the exception/interrupt.

When the exception/interrupt have been handled the kernel performs the following steps:

1) Select a process to restore and resume.
2) Restore the context of the selected process.
3) Resume execution of the selected process.

At any point in time, the values of all the registers in the CPU defines the context of the CPU. Another name used for CPU context is CPU state.

Types of interrupts:

1. VECTORED AND NON-VECTORED INTERRUPTS

A key element in interrupt handling is the ISR address.
If an interrupt has a fixed ISR address, it is called a Vectored interrupt.
Such interrupts are **executed faster** as the ISR address is known to the processor.
But such interrupts are **rigid**. Since they have a fixed ISR address they can serve only **one device**. They cannot accept interrupts from multiple devices. So they cannot expand the interrupt structure.
E.g: NMI interrupt of 8086 (Has a fixed vector number i.e. 2)
If an interrupt does not have a fixed ISR address, it is called a Non-Vectored interrupt. Such interrupts are **executed slower**. The ISR address is **obtained from the** interrupting **device**, usually an interrupt controller like **8259**. But such interrupts are **flexible**. Since they don't have a fixed ISR address they **can accept interrupts from multiple devices**. So they can be used to **expand the interrupt structure**.
E.g: INTR interrupt of 8086 (Can service any vector number from 0… 255)

2. MASKABLE AND NON MASKABLE INTERRUPTS

Masking an interrupt means disabling it. A Mask able interrupt is an interrupt that can be disabled. If disabled, whenever this interrupt occurs, the processor will ignore it and simply continue the main program. Such interrupts are generally used to handle low priority, non-critical events like keyboard presses which can be easily disabled (Keypad can be locked)

E.g.:: INTR interrupt of 8086 (is disabled when Interrupt Flag is 0)

A Non-Mask able interrupt is an interrupt that cannot be disabled. Whenever this interrupt occurs, the processor will have to service it. Such interrupts are generally used to handle high priority, critical events like over-heating of the mother board, power failure etc.

E.g.:: NMI interrupt of 8086 (can never be disabled)

3. SOFTWARE AND HARDWARE INTERRUPTS

This is based on how the interrupt occurs.
If an interrupt is caused by **writing an instruction**, it is called a **software interrupt(Exception)**. Software interrupts are predictable events and are given by the programmer.

E.g.:: INT n instruction of 8086 (n can be anything between 0… 255)
If an interrupt is caused by **a signal on an external pin**, it is called a **hardware interrupt**. Hardware interrupts are un-predictable events and are given by external devices.
E.g.:: NMI and INTR pins of 8086

I/O Device Interfaces:

Universal Serial Bus (USB):

The Universal Serial Bus (USB) [1] is the most widely used interconnection standard. A large variety of devices are available with a USB connector, including mice, memory keys, disk drives, printers, cameras, and many more. The commercial success of the USB is due to its simplicity and low cost. The original USB specification supports two speeds of operation, called low-speed (1.5 Megabits/s) and full-speed (12 Megabits/s). Later, USB 2, called High-Speed USB, was introduced. It enables data transfers at speeds up to 480 Megabits/s. As I/O devices continued to evolve with even higher speed requirements, USB 3 (called Superspeed) was developed. It supports data transfer rates up to 5 Gigabits/s.

The USB has been designed to meet several key objectives:

• Provide a simple, low-cost, and easy to use interconnection system

• Accommodate a wide range of I/O devices and bit rates, including Internet connections, and audio and video applications

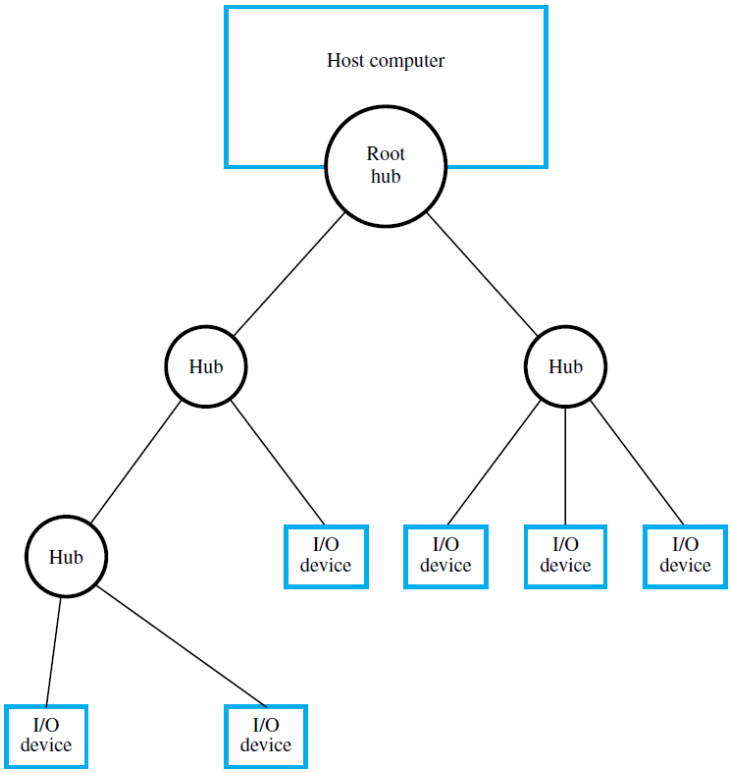
• Enhance user convenience through a “plug-and-play” mode of operation

Plug-and-Play

When an I/O device is connected to a computer, the operating system needs some information about it. It needs to know what type of device it is so that it can use the appropriate device driver. It also needs to know the addresses of the registers in the device’s interface to be able to communicate with it. The USB standard defines both the USB hardware and the software that communicates with it. Its plug-and-play feature means that when a new device is connected, the system detects its existence automatically. The software determines the kind of device and how to communicate with it, as well as any special requirements it might have. As a result, the user simply plugs in a USB device and begins to use it, without having to get involved in any of these details. The USB is also hot-pluggable, which means a device can be plugged into or removed from a USB port while power is turned on.

USB Architecture:

The USB uses point-to-point connections and a serial transmission format. When multiple devices are connected, they are arranged in a tree structure as shown in Figure below:



Each node of the tree has a device called a hub, which acts as an intermediate transfer point between the host computer 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: a mouse, a keyboard, a printer, an Internet connection, a camera, or a speaker.

If I/O devices are allowed to send messages at any time, two messages may reach the hub at the same time and interfere with each other. For this reason, 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 processor. Hence, no two devices can send messages at the same time. This restriction allows hubs to be simple, low-cost devices. Each device on the USB, whether it is a hub or an I/O device, is assigned a 7-bit address. The root hub of the USB, which is attached to the processor, appears as a single device. The host software communicates with individual devices by sending information to the root hub, which it forwards to the appropriate device in the USB tree.

When a device is first connected to a hub, or when it is powered on, it has the address 0. Periodically, the host polls each hub to collect status information and learn about new devices that may have been added or disconnected. When the host is informed that a new device has been connected, it reads the information in a special memory in the device's USB interface to learn about the device's capabilities. It then assigns the device a unique USB address and writes that address in one of the device's interface registers. It is this initial connection procedure that gives the USB its plug-and-play capability.

Isochronous Traffic on USB

An important feature of the USB is its ability to support the transfer of isochronous data in a simple manner. Isochronous data need to be transferred

at precisely timed regular intervals. To accommodate this type of traffic, the root hub transmits a uniquely recognizable sequence of bits over the USB tree every millisecond. This sequence of bits, called a Start of Frame character, acts as a marker indicating the beginning of isochronous data, which are transmitted after this character. Thus, digitized audio and video signals can be transferred in a regular and precisely timed manner.

Electrical Characteristics:

USB connections consist of four wires, of which two carry power, +5 V and Ground, and two carry data. Thus, I/O devices that do not have large power requirements can be powered directly from the USB. This obviates the need for a separate power supply for simple devices such as a memory key or a mouse.
Two methods are used to send data over a USB cable. When sending data at low speed, a high voltage relative to Ground is transmitted on one of the two data wires to represent a 0 and on the other to represent a 1. The Ground wire carries the return current in both cases. Such a scheme in which a signal is injected on a wire relative to ground is referred to as single-ended transmission.
The speed at which data can be sent on any cable is limited by the amount of electrical noise present. The term noise refers to any signal that interferes with the desired data signal and hence could cause errors. Single-ended transmission is highly susceptible to noise. The voltage on the ground wire is common to all the devices connected to the computer. Signals sent by one device can cause small variations in the voltage on the ground wire, and hence can interfere with signals sent by another device. Interference can also be caused by one wire picking up noise from nearby wires. The High-Speed USB uses an alternative arrangement known as differential signaling. The data signal is injected between two data wires twisted together. The ground wire is not involved. The receiver senses the voltage difference between the two signal wires directly,
without reference to ground. This arrangement is very effective in reducing the noise seen by the receiver, because any noise injected on one of the two wires of the twisted pair is also injected on the other. Since the receiver is sensitive only to the voltage difference between the two wires, the noise component is cancelled out. The ground wire acts as a shield for the data on the twisted pair against interference from nearby wires. Differential signaling allows much lower voltages and much higher speeds to be used compared to single-ended signaling.

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). The SCSI bus may be used to connect a variety of devices to a computer. It is particularly well-suited for use with disk drives. It is often found in installations such as institutional databases or email systems where many disks drives are used.
In the original specifications of the SCSI standard, devices 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 of up to 5 Megabytes/s. The standard has undergone many revisions, and its data transfer capability has increased rapidly. SCSI-2 and SCSI-3 have been defined, and each has several options. Data are transferred either 8 bits or 16 bits in parallel, using clock speeds of up to 80 MHz. There are also several options for the electrical signaling scheme used. The bus may use single-ended transmission, where each signal uses one wire, with a common ground return for all signals. In another option, differential signaling is used, with a pair of wires for each signal. wires for each signal.

Data Transfer

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 or to the PCI bus. A SCSI bus may be connected directly to the processor bus, or more likel y to another standard I/O bus such as PCI, through a SCSI controller. Data and commands are transferred in the form of multi-byte messages called packets. To send commands or data to a device, the processor assembles the information in the memory then instructs the SCSI controller to transfer it to the device.

Similarly, when data are read from a device, the controller transfers the data to the memory and then informs the processor by raising an interrupt.

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. Data are stored on a disk in blocks called sectors, where each sector may contain several hundred bytes. When a data file is written on a disk, it is not always stored in contiguous sectors. Some sectors may already contain previously stored information; others may be defective and must be skipped. Hence, a Read or Write request may result in accessing several disk sectors that are not necessarily contiguous. Because of the constraints of the mechanical motion of the disk, there is a long delay, on the order of several milliseconds, before reaching the first sector to or from which data are to be transferred. Then, a burst of data are transferred at high speed. Another delay may ensue to reach the next sector, followed by a burst of data. A single Read or Write request may involve several such bursts. The SCSI protocol is designed to facilitate this mode of operation. Let us examine a complete Read operation as an example. The following is a simplified high-level description, ignoring details and signaling conventions. Assume that the processor wishes to read a block of data from a disk drive and that these data are stored in two disk sectors that are not contiguous. The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

1. The SCSI controller contends for control of the SCSI bus.
2. When it wins the arbitration process, the SCSI controller sends a command to the disk controller, specifying the required Read operation.
3. The disk controller cannot start to transfer data immediately. It must first move the read head of the disk to the required sector. Hence, it sends a message to the SCSI controller indicating that it will temporarily suspend the connection between them. The SCSI bus is now free to be used by other devices.
4. The disk controller sends a command to the disk drive to move the read head to the first sector involved in the requested Read operation. It reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data, it requests control of the bus. After it wins arbitration, it re-establishes the connection with the SCSI controller, sends the contents of the data buffer, then suspends the connection again.
5. The process is repeated to read and transfer the contents of the second disk sector.
6. The SCSI controller transfers the requested data to the main memory and sends an interrupt to the processor indicating that the data are now available.

This scenario shows that the messages exchanged over the SCSI bus are at a higher level than those exchanged over the processor bus. Messages refer to more complex 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 the disk's operation and how it moves from one sector to the next. The SCSI bus standard defines a wide range of control messages that can be used to handle different types of I/O devices. Messages are also defined to deal with various error or failure conditions that might arise during device operation or data transfer.

MODULE-4

RISC vs CISC Architecture

RISC stands for Reduced Instruction set Computer and **CISC** stands for Complex Instruction Set Computer .RISC and CISC are the two ideologies behind making the processor.

	RISC	CISC
1	**Instructions of a fixed size**	Instructions of **variable size**
2	Most instructions take **same time to fetch**.	Instructions have **different fetching times**.
3	Instruction set **simple and small**.	Instruction set **large and complex**.
4	**Less addressing modes** as most operations are register based.	**Complex addressing modes** as most operations are memory based.
5	**Compiler design is simple**	**Compiler design is complex**
6	**Total size of program is large** as many instructions are required to perform a task as instructions are simple.	**Total size of program is small** as few instructions are required to perform a task as instructions are complex & more powerful.
7	Instructions use a **fixed number of operands.**	Instructions have **variable** number of operands.
8	Ideal for processors performing a **dedicated operation**.	Ideal for processors performing a **verity of operations**.
9	Since instructions are simple, they can be decoded by a **hardwired control unit**.	Since instructions are complex, they require a **Micro-programmed Control Unit**.
10	Execution speed is **faster** as most operations are register based.	Execution speed is **slower** as most operations are memory based.
11	As No. of cycles per instruction is fixed, it gives a **better degree of pipelining**	Since number of cycles per instruction varies, **pipelining has more bubbles** or stalls.
12	E.g.:: **ARM7**, **PIC 18** Microcontrollers.	E.g.: Intel **8085, 8086** Microprocessors.

Pipelining and Parallel Processors

Basic Concepts of Pipelining :

Introduction:

1. Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.
2. A pipeline can be visualized as a collection of processing segments through which binary information flows.
3. Each segment performs partial processing dictated by the way the task is partitioned.
4. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.
5. It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time. The overlapping of computation is made possible by associating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

Pipeline organization is demonstrated by means of a simple example.

Suppose that we want to perform the combined multiply and add operations with a stream of numbers. Ai* Bi + Ci for i = 1, 2, 3, . . . , 7 Each sub operation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig below:

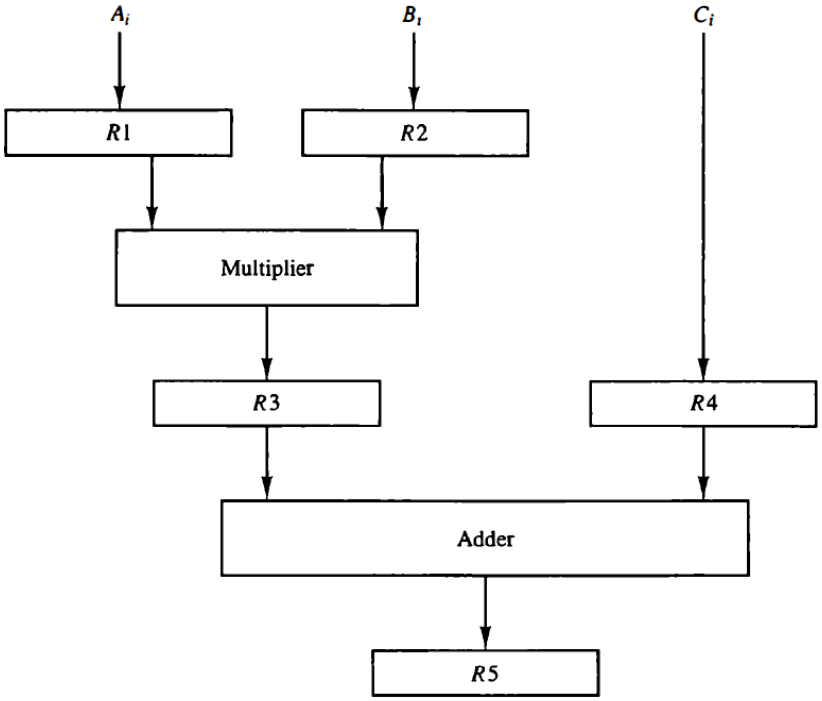
The sub operations performed in each segment of the pipeline are as follows:

R 1 <--Ai, R2 <--Bi

R3 <--R 1 * R2, R4 <--C,

R5 <--R3 + R4

- Input A, and B,
- Multiply and input C,
- Add C; to product



Example of pipeline processing.

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Table: Contents of Registers in pipeline

The five registers are loaded with new data every clock pulse. The first clock pulse transfers A1 and B1 into R 1 and R2. The second dock pulse transfers the product of R 1 and R2 into

R3 and C1 into R4. The same clock pulse transfers A2 and B2 into R 1 and R2. The third clock pulse operates on all three segments simultaneously. It places A, and B, into R1 and R2, transfers the product of R1 and R2 into R3, transfers C, into R4, and places the sum of R3 and R4 into RS. It takes three clock pulses to fill up the pipe and retrieve the first output from RS. From there on, each dock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

Instruction Pipelining:

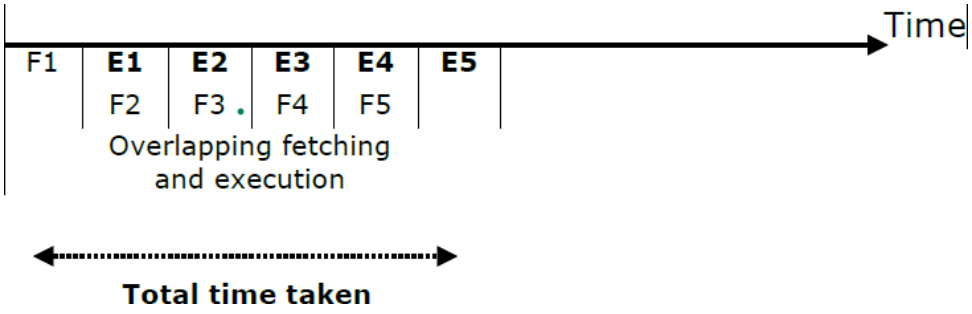
Instruction Pipelining is an implementation technique in which ***multiple instructions are overlapped in execution.*** An instruction requires several steps which mainly involve fetching, decoding and execution.

If these steps are performed one after the other, they will take a long time.

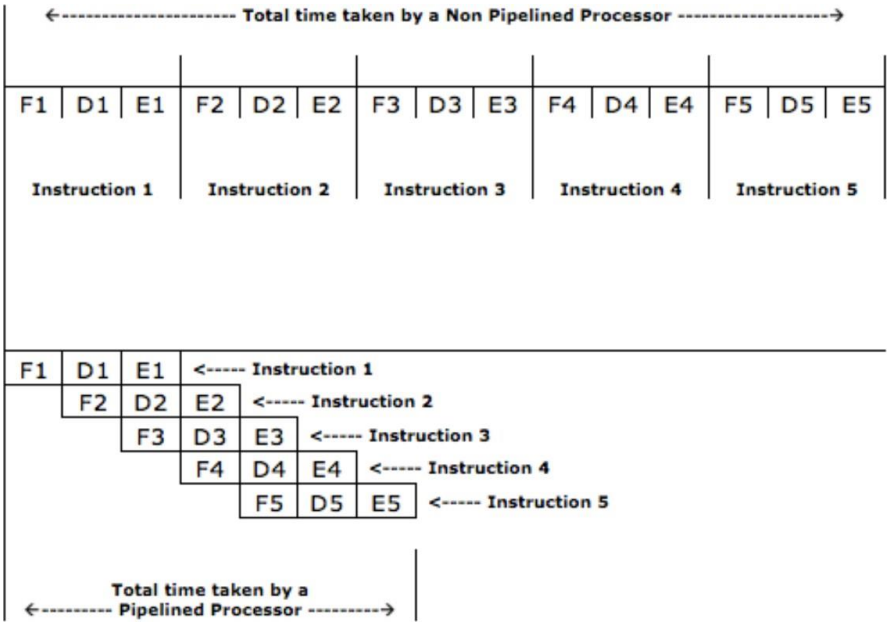
As processors became faster, several of these steps started to get overlapped, resulting in faster processing. This is done by a mechanism called pipelining.

2 STAGE PIPELINING - 8086

Here the instruction process in divided into two stages of fetching and execution. Fetching of the next instruction takes place while the current instruction is being executed. Hence two instructions are being processed at any point of time.



3 STAGE PIPELINING –ARM 7



Consider the case where a k-segment pipeline with a clock cycle time t_p ,is used to execute n tasks. The first task T1 requires a time equal to Kt_p, to complete its operation since there are k segments in the pipe. The remaining n - 1 tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to (n - 1) tp . Therefore, to complete n tasks using a k-segment pipeline requires k + (n - 1) clock cycles.

Next consider a non-pipeline unit that performs the same operation and
takes a time equal to t_n. to complete each task. The total time required for n tasks is nt_n. The speedup of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

ADVANTAGE OF PIPELINING

The advantage of pipelining is that it increases the performance. As shown by the various examples above, deeper the pipelining, more is the level of parallelism, and hence the processor becomes much faster.

DRAWBACKS/ HAZARDS OF PIPELINING

There are various hazards of pipelining, which **cause a dip** in the performance of the processor. These hazards become even **more prominent** as the **number of pipeline stages increase**. They may occur
due to the following reasons.

1) DATA HAZARD/ DATA DEPENDENCY HAZARD

Data Hazard is caused when **the result (destination) of one instruction becomes the operand (source) of the next instruction**.
Consider two instructions I1 and I2 (I1 being the first).
Assume I1: INC [4000H]
Assume I2: MOV BL , [4000H]
Clearly in I2, BL should get the incremented value of location [4000H].
But this can only happen once I1 has completely finished execution and also written back the result at [4000H].
In a multistage pipeline, I2 may reach execution stage before I1 has finished storing the result at location [4000H], and hence get a wrong value of data.
This is called **data dependency hazard**.
It is solved by inserting NOP (No operation) instructions between such data dependent instructions.

Because of the data hazard, there will be a delay in the pipeline. The data hazards are basically of three types:

1. RAW
2. WAR
3. WAW

To understand these hazards, we will assume we have two instructions I1 and I2, in such a way that I2 follows :

RAW:

RAW hazard can be referred to as 'Read after Write'. It is also known as Flow/True data dependency. If the later instruction tries to read on operand before earlier instruction writes it, in this case, the RAW hazards will occur. The condition to detect the RAW hazard is when O_n (Output of nth instruction) and I_{n+1}(Input of $n+1^{th}$ instruction) both have a minimum one common operand.

I1: add R1, R2, R3

I2: sub R5, R1, R4

WAR

WAR can be referred to as 'Write after Read'. It is also known as Anti-Data dependency. If the later instruction tries to write an operand before the earlier instruction reads it, in this case, the WAR hazards will occur. The condition to detect the WAR hazard is when I_n and O_{n+1} both have a minimum one common operand.

add R1, **R2,** R3
sub **R2,** R5, R4

In a reasonable (in-order) pipeline, the WAR hazard is very uncommon or impossible.

WAW

WAW can be referred to as 'Write after Write'. It is also known as Output Data dependency. If the later instruction tries to write on operand before earlier instruction writes it, in this case, the WAW hazards will occur. The condition to detect the WAW hazard is when O_n and O_{n+1} both have a minimum one common operand.

add **R1,** R2, R3
sub **R1,** R2, R4

2) CONTROL HAZARD/ CODE HAZARD

Pipelining assumes that the program will always flow in a sequential manner.
Hence, it performs various stages of the forthcoming instructions before-hand, while the current instruction is still being executed. While programs are sequential most of the times, it is not true always.
Sometimes, branches do occur in programs.

In such an event, all the forthcoming instructions that have been fetched/ decoded etc have to be flushed/ discarded, and the process has to start all over again, from the branch address. This causes pipeline bubbles, which simply means time of the processor is wasted. Consider the following set of instructions:

Start:
JMP Down
INC BL
MOV CL, DL
ADD AL, BL
…
…
…
Down: DEC CH

JMP Down is a branch instruction.

After this instruction, program should jump to the location "Down" and continue with DEC CH
instruction.

But, in a multistage pipeline processor, the sequentially next instructions after JMP Down have already been fetched and decoded. These instructions will now have to be discarded and fetching will begin all over again from DEC CH. This will keep several units of the architecture idle for some time. This is called a pipeline bubble. The **problem of branching is solved** in higher processors by a method called "**Branch Prediction Algorithm**". It was introduced by **Pentium** processor. It relies on the **previous history** of the instruction as most programs are repetitive in nature. It then **makes a prediction** whether branch will be **taken or not** and hence puts the correct instructions in the pipelines.

3) STRUCTURAL HAZARD

Structural hazards are caused by **physical constraints in the architecture like the buses**. Even in the most basic form of pipelining, we want to execute one instruction and fetch the next one. Now as long as execution only involves registers, pipelining is possible. But **if execution requires to read/ write data from the memory, then it will make use of the buses, which means fetching cannot take place at the same time**. So the fetching unit will have to wait and hence a pipeline bubble is caused. This problem is solved in complex Harvard architecture processors, which use separate memories and separate buses for programs and data. This means fetching and execution can actually happen at the same time without any interference with each other.

E.g.: PIC 18 Microcontroller.

Introduction to Parallel Processors:

- A multiprocessor system is an interconnection of two or more CPUs with memory and input-output equipment. The term "processor" In multiprocessor can mean either a central processing unit (CPU) or an input-output processor (lOP).
- However, a system with a single CPU and one or more lOPs is usually not included in the definition of a multiprocessor system unless the lOP has computational facilities comparable to a CPU.
- Multiprocessors are classified as **multiple instruction stream, multiple data stream (MIMD) systems.**

- A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem.
- The fact that microprocessors take **very little physical space and are very inexpensive** brings about the feasibility of interconnecting a large number of microprocessors into one composite system.
- **Very-large-scale integrated circuit technology** has reduced the cost of computer components to such a low level that the concept of applying multiple processors to meet system performance requirements has become an attractive design possibility.
- Multiprocessing **improves the reliability** of the system so that a failure or error in one part has a limited effect on the rest of the system. If a fault causes one processor to fail, a second processor can be assigned to perform the functions of the disabled processor. The system as a whole can continue to function correctly with perhaps some loss in efficiency.
- A multiprocessor system derives its high performance from the fact that computations can proceed in parallel in one of two ways.
  1. Multiple independent jobs can be made to operate in parallel.
  2. A single job can be partitioned into multiple parallel tasks.
- An overall function can be partitioned into a number of tasks that each processor can handle individually. System tasks may be allocated to special purpose processors whose design is optimized to perform certain types of processing efficiently.
- Example is a computer where one processor performs highspeed floating-point mathematical computations and another takes care of routine data-processing tasks.
- Multiprocessors are classified by the way their memory is organized.
  1. A multiprocessor system with common shared memory is classified as a shared memory or tightly coupled multiprocessor. This does not preclude each processor from having its own local memory. In fact, most commercial **tightly coupled multiprocessors** provide a cache memory with each CPU. In addition, there is a global common memory that all CPUs can access. Information can therefore be shared among the CPUs by placing it in the common global memory.
  2. An alternative model of microprocessor is the distributed-memory or **loosely coupled system**. Each processor element in a loosely coupled system has its own private local memory. The processors are tied together by a switching scheme designed to route information from one processor to another through a message- passing scheme. The processors relay program and data to other processors in packets. A packet consists of an address, the data content, and some error detection code. The packets are addressed to a specific processor or taken by the first available processor, depending on the communication system used.

Shared Memory Multiprocessors:

- A **multiprocessor system consists of a number of processors capable of simultaneously executing independent tasks.** A task may encompass a few

instructions for one pass through a loop, or thousands of instructions executed in a subroutine.

- In a shared-memory multiprocessor**, all processors have access to the same memory**. Tasks running in different processors can access shared variables in the memory using the same addresses. The size of the shared memory is likely to be large.
- Implementing a **large memory in a single module would create a bottleneck when many processors make requests to access the memory simultaneously**. This **problem is alleviated by distributing the memory across multiple modules** so that simultaneous requests from different processors are more likely to access different memory modules, depending on the addresses of those requests.
- **An interconnection network enables any processor to access any module that is a part of the shared memory**. When memory modules are kept physically separate from the processors, all requests to access memory must pass through the network. Below Figure shows such an arrangement.
- A **system which has the same network latency for all accesses from the processors to the memory modules is called a Uniform Memory Access** (UMA) multiprocessor.

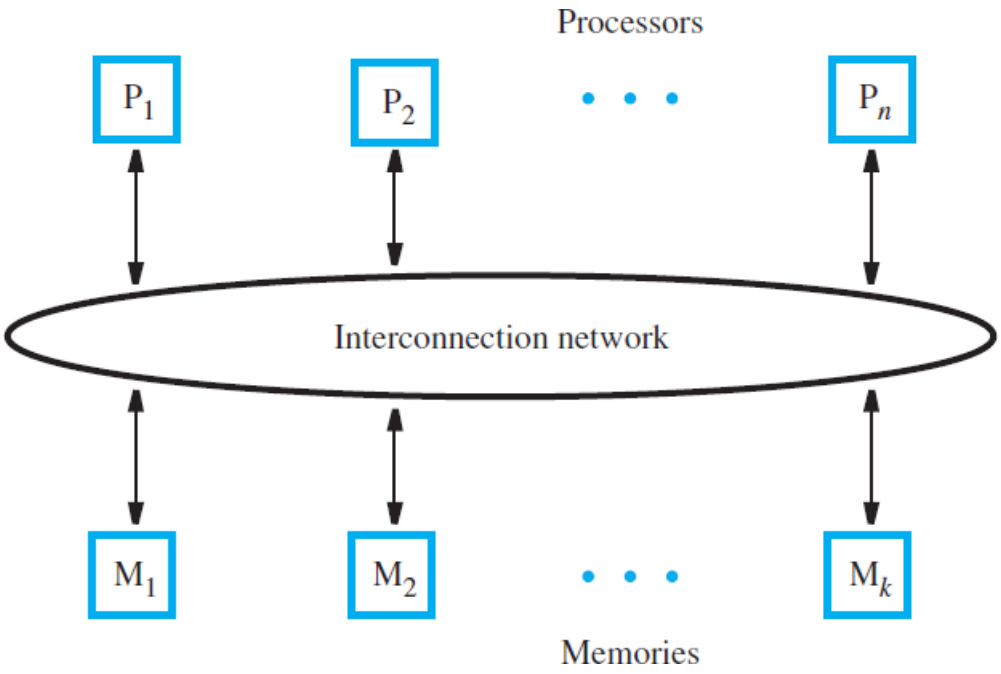


Fig: A UMA multiprocessor.

- For better performance, **it is desirable to place a memory module close to each processor.** The result is a collection of nodes, each consisting of a processor and a memory module. The nodes are then connected to the network, as shown in Figure below:

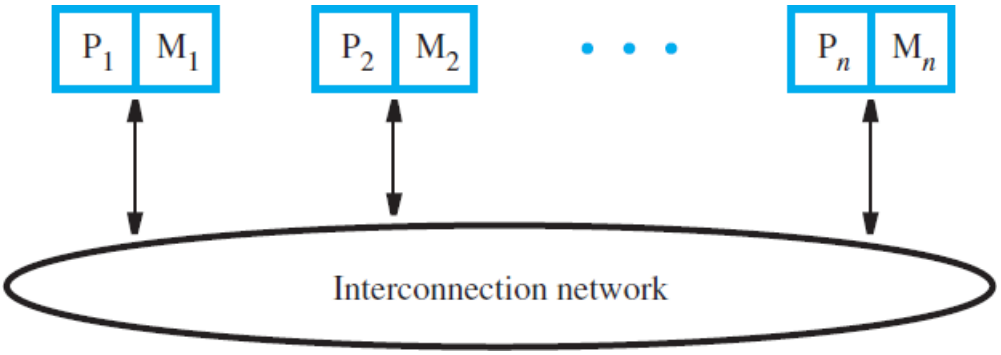


Fig: A NUMA multiprocessor.

- The network latency is avoided when a processor makes a request to access its local memory. However, a request to access a remote memory module must pass through the network. **Because of the difference in latencies for accessing local and remote portions of the shared memory, systems of this type are called Non-Uniform Memory Access (NUMA) multiprocessors**.

Interconnection Networks:

- The interconnection network must allow information transfer between any pair of nodes in the system. The network may also be used to broadcast information from one node to many other nodes. The traffic in the network consists of requests (such as read and write)
  and data transfers.
- The suitability of a particular network is judged in terms of **cost, bandwidth, effective throughput, and ease of implementation. The term bandwidth refers to the capacity of a transmission link to transfer data and is expressed in bits or bytes per second.** The effective throughput is the actual rate of data transfer. **This rate is less than the available bandwidth because a given link must also carry control information that coordinates the transfer of data.**
- **Information transfer through the network usually takes place in the form of packets of fixed length and specified format.** For example, a read request is likely to be a single packet sent from a processor to a memory module. The packet contains the node identifiers for the source and destination, the address of the location to be read, and a command field that indicates what type of read operation is required. A write request that writes one word in a memory module is also likely to be a single packet that includes the data to be written. On the other hand, a read response may involve an entire cache block requiring several packets for the data transfer.
- Ideally**, a complete packet would be handled in parallel in one clock cycle at any node or switch in the network.** This implies having wide links, comprising many wires. However, **to reduce cost and complexity, the links are often considerably narrower**. In such cases, a packet must be divided into smaller pieces, each of which can be transmitted in one clock cycle.
- The following are few of the interconnection networks that are commonly used in multiprocessors:

Bus:

A bus is a set of lines (wires) that provide a single shared path for information transfer. Buses are most commonly used in UMA multiprocessors to connect a

number of processors to several shared-memory modules. **Arbitration is necessary to ensure that only one of many possible requesters is granted use of the bus at any time.**

The bus is suitable for a relatively small number of processors because of the contention for access to the bus when many processors are connected. A simple bus does not allow a new request to appear on the bus until the response for the current request has been provided. However, if the response latency is high, there may be considerable idle time on the bus. Higher performance can be achieved by using a **split-transaction bus, in which a request and its corresponding response are treated as separate events. Other transfers may take place between them.**

Ring:

A ring network is formed with point-to-point connections between nodes, as shown in Figure below:

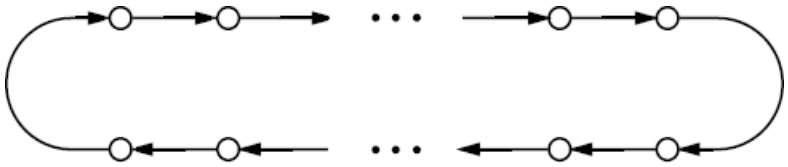
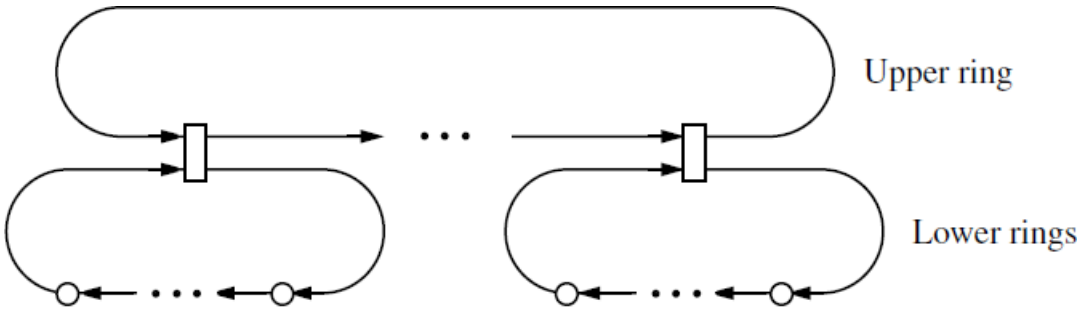


Fig: Simple Ring

A long single ring results in high average latency for communication between any two nodes. This high latency can be mitigated in two different ways. **A second ring can be added to connect the nodes in the opposite direction.** The resulting bidirectional ring halves the average latency and doubles the bandwidth. However, handling of communications is more complex.

Another approach is to use a hierarchy of rings. A two-level hierarchy is shown in Figure below: The upper-level ring connects the lower-level rings. The average latency for communication between any two nodes on lower-level rings is reduced with this arrangement. Transfers between nodes on the same lower-level ring need not traverse the

upper-level ring. Transfers between nodes on different lower-level rings include a traversal on part of the upper-level ring.



(b) Hierarchy of rings

Crossbar:

A crossbar is a network that provides a direct link between any pair of units connected to the network. It is typically used in UMA multiprocessors to connect

processors to memory modules. It enables many simultaneous transfers if the same destination is not the target of multiple requests. Below figure shows a crossbar that comprises a collection of switches. For n processors and k memories, n × k switches are needed.

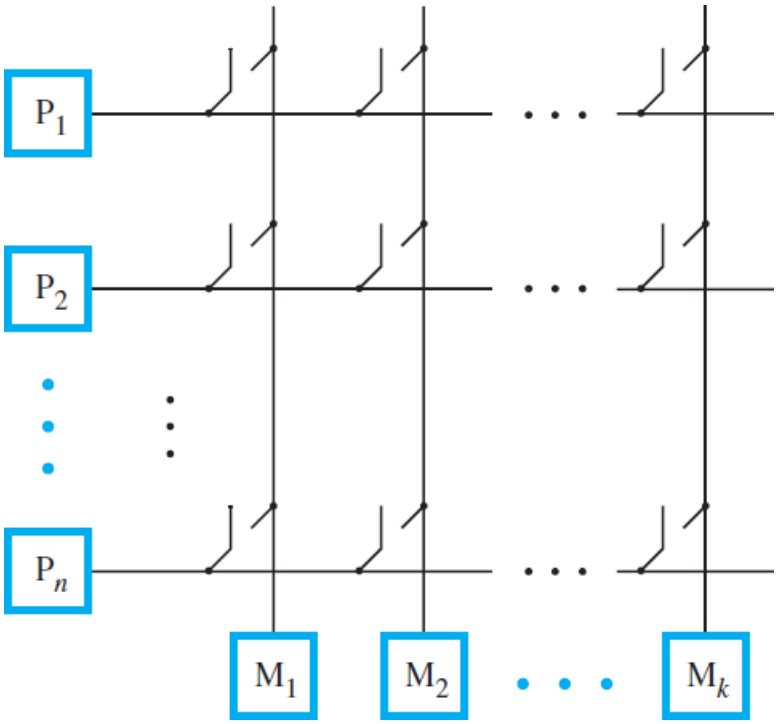


Fig: Crossbar Interconnection Network

Mesh:

A natural way of connecting a large number of nodes is with a two-dimensional mesh, as shown in Figure below:

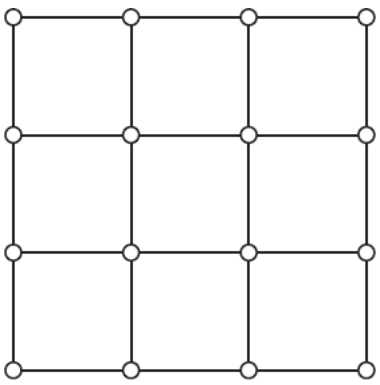


Fig : A two-dimensional mesh network.

Each internal node of the mesh has four connections, one to each of its horizontal and vertical neighbours. Nodes on the boundaries and corners of the mesh have fewer neighbours and hence fewer connections. To reduce latency for communication between nodes that would otherwise be far apart in the mesh, wrap around connections may be introduced between nodes at opposite boundaries of the mesh. **A network with**

such connections is called a torus. All nodes in a torus have four connections. Average latency is reduced, but the implementation complexity for routing requests and responses through a torus is somewhat higher than in the case of a simple mesh.

Cache Coherence:

1. A shared-memory multiprocessor is easy to program. *Each variable in a program has a unique address location in the memory, which can be accessed by any processor.* However, *each processor has its own cache*. Therefore, it is *necessary to deal with the possibility that copies of shared data may reside in several caches*.
2. When any processor writes to a shared variable in its own cache, all other caches that contain a copy of that variable will then have the old, incorrect value. They must be informed of the change so that they can either update their copy to the new value or invalidate it. This is the issue of maintaining *cache coherence, which requires having a consistent view of shared data in multiple caches*.
3. The write-through approach changes the data in both the cache and the main memory. The write-back approach changes the data only in the cache; the main memory copy is updated when a modified data block in the cache has to be replaced. Similar approaches can be used to address cache coherence in a multiprocessor system.

Write Through Protocol:

A write-through protocol can be implemented in one of two ways:

1) First version is based on updating the values in other caches. When a processor writes a new value to a block of data in its cache, the new value is also written into the memory module containing the block being modified. Since copies of this block may exist in other caches, these copies must be updated to reflect the change caused by the Write operation. The simplest way of doing this is to broadcast the written data to the caches of all processors in the system. As each processor receives the broadcast data, it updates the contents of the affected cache block if this block is present in its cache.

2) The second version of the write-through protocol is based on invalidation of copies. When a processor writes a new value into its cache, this value is also sent to the appropriate location in memory, and all copies in other caches are invalidated. Again, broadcasting can be used to send the invalidation requests throughout the system.

Write-Back protocol:

- ✓ Maintaining coherence with the write-back protocol **is based on the concept of ownership of a block of data in the memory.** Initially, the memory is the owner of all blocks, and the memory retains ownership of any block that is read by a processor to place a copy in its cache.
- ✓ If some processor wants to write to a block in its cache, it must first become the exclusive owner of this block. To do so, **all copies in other caches must first be invalidated with a broadcast request**. The new owner of the block may then modify the contents at will without having to take any other action.
- ✓ **Read: When another processor wishes to read a block that has been modified**, **the request for the block must be forwarded to the current owner.** The data

are then sent to the requesting processor by the current owner**. The data are also sent to the appropriate memory module, which reacquires ownership and updates the contents of the block in the memory.** The cache of the processor that was the previous owner retains a copy of the block. **Subsequent requests from other processors to read the same block are serviced by the memory module containing the block.**

✓ When another processor wishes to write to a block that has been modified, the current owner sends the data to the requesting processor. **It also transfers ownership of the block to the requesting processor and invalidates its cached copy**. Since the block is being modified by the new owner, the contents of the block in the memory are not updated. The next request for the same block is serviced by the new owner.

✓ The ***write-back protocol has the advantage of creating less traffic than the write-through protocol***. This is because a processor is likely to perform several writes to a cache block

before this block is needed by another processor. With the write-back protocol, these writes are performed only in the cache, once ownership is acquired with an invalidation request.

Snoopy Caches:

✓ In multiprocessors that connect a modest number of processors to the memory modules using a single bus, ***cache***

coherence can be realized using a scheme known as **snooping.**

✓ In a single-bus system, **all transactions between processors and memory modules occur via requests and responses on the bus.** Suppose that each processor cache has a controller circuit that observes, or snoops, all transactions on the bus.

✓ Below are some scenarios for the write-back protocol and how cache coherence is enforced**:**

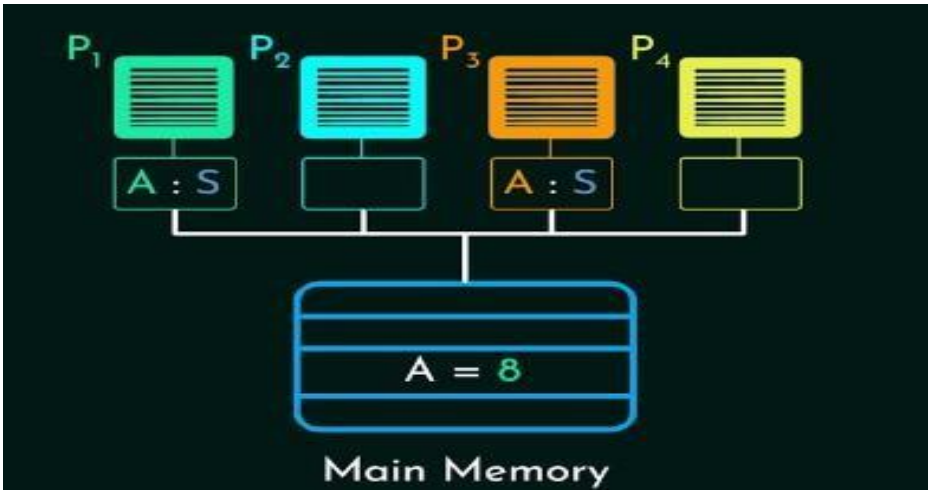
➢ Consider a processor that has previously read a copy of a block from the memory into its cache. Before writing to this block for the first time, **the processor must broadcast an invalidation request to all other caches, whose controllers accept the request and invalidate any copies of the same block.** This action causes the requesting processor to become the new owner of the block. The processor may then write to the block and mark it as being modified. No further broadcasts are needed from the same processor to write to the modified block in its cache.

➢ Now, if another processor broadcasts a read request on the bus for the same block, the memory must not respond because it is not the current owner of the block. The processor owning the requested block snoops the read request on the bus. **Because it holds a modified copy of the requested block in its cache, it asserts a special signal on the bus to prevent the memory from responding**. The owner then broadcasts a copy of the block on the bus, and marks its copy as clean (unmodified). The data response on the bus is accepted by the cache of the processor that issued the read request. **The data response is also accepted by the memory to update its copy of the block. In this case, the memory reacquires ownership of the block, and the block is said to be in a shared state because copies of it are in the caches of two processors**. Coherence is maintained because the two cached copies and the copy of the block in the memory contain the same data. **Subsequent requests from any processor are serviced by the memory**.

➢ Consider now the situation in which **two processors have copies of the same block in their respective caches**, and **both processors attempt to write to the same cache block at the same time**. Since the block is in the shared state, the memory is the owner of the block. Hence, **both processors request the use of the bus to broadcast an invalidation messag**e. One of the processors is granted the use of the bus first. **That processor broadcasts**

its invalidation request and becomes the new owner of the block. Through snooping, the copy of the block in the cache of the other processor is invalidated. When the other processor is later granted the use of the bus, it broadcasts a **read-exclusive request. This request combines a read request and an invalidation** request for the same block. The controller for the first processor snoops the read-exclusive request, provides a data response on the bus, and invalidates the copy in its cache. **Ownership of the block is therefore transferred to the second processor making the request**. The memory is not updated because the block is being modified again. Since the requests from the two processors are handled sequentially, cache coherence is maintained at all times.

- **The scheme just described is based on the ability of cache controllers to observe the activity on the bus and take appropriate actions**. Such schemes are called **snoopy-cache techniques**.



Snooping based protocol

Directory-Based Cache Coherence:

The concept of snoopy caches is easy to implement in single-bus systems. **Large shared memory multiprocessors use interconnection networks such as rings and meshe**s. In such systems, **broadcasting every single request to the caches of all processors is inefficient**. A scalable, but more complex, solution to this problem **uses directories** in each memory module **to indicate which nodes may have copies of a given block in the shared state. If a block is modified, the directory identifies the node that is the current owner.** Each request from a processor must be sent first to the memory module containing the relevant block. **The directory information for that block is used to determine the action that is taken. A read request is forwarded to the current owner, if the block is modified. In the case of a write request for a block that is shared, individual invalidations are sent only to nodes that may have copies of the block in question**. The cost and complexity of the directory-based approach for enforcing cache coherence limits its use to large systems. Small multiprocessors, including current multicore chips, typically use snooping.

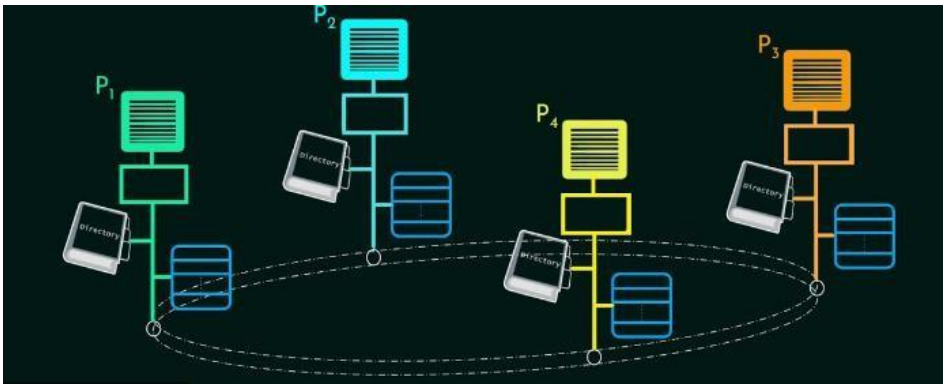
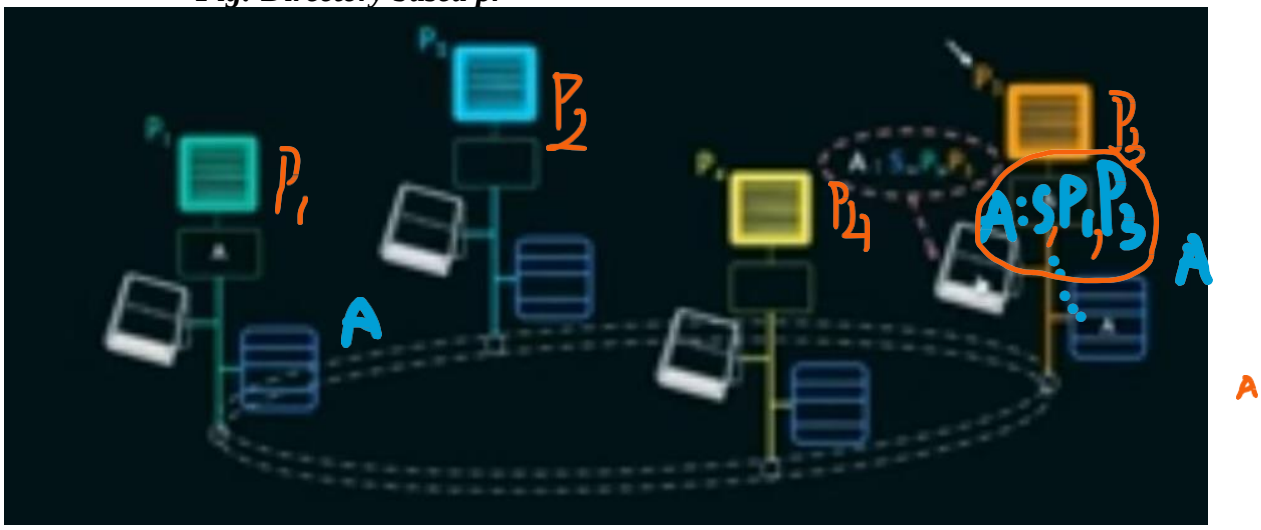


Fig: Directory based pr



MODULE-5

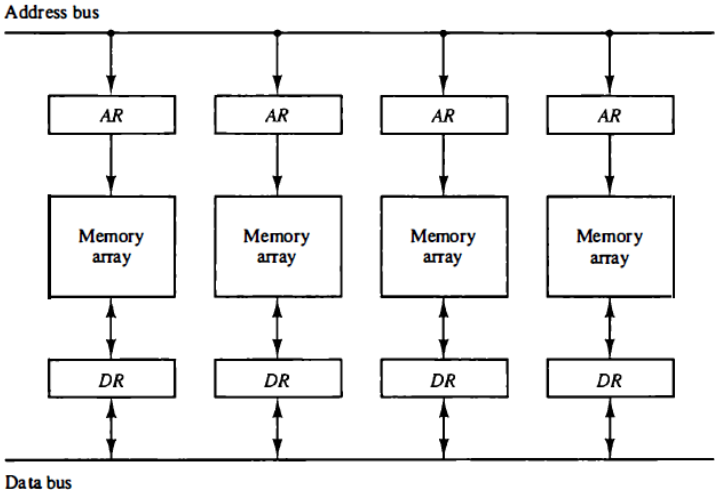
MEMORY ORGANIZATION

Memory Interleaving:

Pipeline and vector processors often require simultaneous access to memory from two or more sources. An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments.

Similarly, an arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time. Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses. A memory module is a memory array together with its own address and data registers. Figure 9-13 shows a memory unit with four modules. Each memory array has its own address register AR and data register DR.

Figure 9-13 Multiple module memory organization.



✓ The address registers receive information from a common address bus and the data registers communicate with a bidirectional data bus. The two least significant bits of the address can be used to distinguish between the four modules. The modular system permits one module to initiate a memory access while other modules are in the process of reading or writing a word and each module can honor a memory request independent of the state of the other modules.

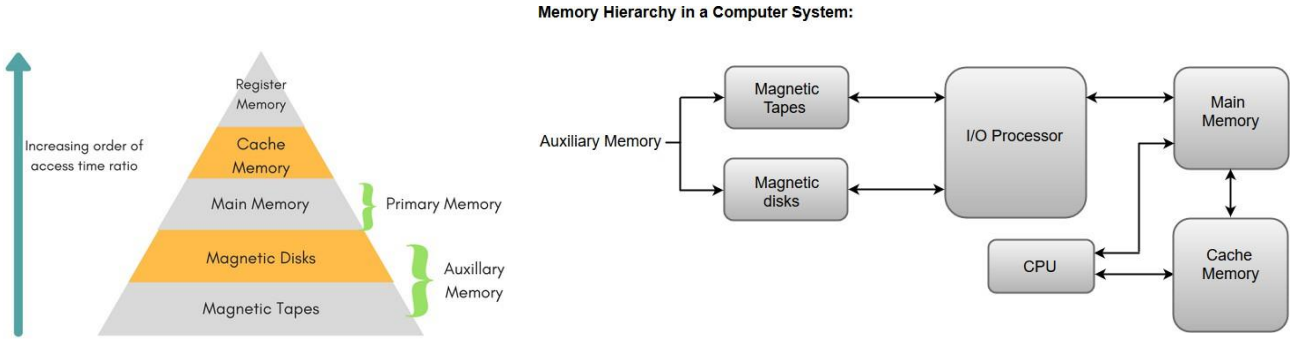
✓ The advantage of a modular memory is that it allows the use of a technique called interleaving. In an interleaved memory, different sets of addresses are assigned to different memory modules. For example, in a two-module memory system, the even addresses may be in one module and the odd addresses in the other.

✓ Concept of Hierarchical Memory Organization
✓ This Memory Hierarchy Design is divided into 2 main types:
✓ External Memory or Secondary Memory

✓ Comprising of Magnetic Disk, Optical Disk, Magnetic Tape i.e. peripheral storage devices which are accessible by the processor via I/O Module.

✓ Internal Memory or Primary Memory

✓ Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by the processor.



✓ Characteristics of Memory Hierarchy Capacity:

✓ It is the global volume of information the memory can store. As we move from top to bottom in the Hierarchy, the capacity increases.

✓ Access Time:

✓ It is the time interval between the read/write request and the availability of the data. As we move from top to bottom in the Hierarchy, the access time increases.

✓ Performance:

✓ Earlier when the computer system was designed without Memory Hierarchy design, the speed gap increases between the CPU registers and Main Memory due to large difference in access time. This results in lower performance of the system and thus, enhancement was required. This enhancement was made in the form of Memory Hierarchy Design because of which the performance of the system increases. One of the most significant ways to increase system performance is minimizing how far down the memory hierarchy one has to go to manipulate data.

✓ Cost per bit:

✓ As we move from bottom to top in the Hierarchy, the cost per bit increases i.e. Internal Memory is costlier than External Memory.

✓

✓ Cache Memories:

✓ The cache is a small and very fast memory, interposed between the processor and the main memory. Its purpose is to make the main memory appear to the processor to be much faster than it actually is. The effectiveness of this approach is based on a property of computer programs called locality of reference.

✓ Analysis of programs shows that most of their execution time is spent in routines in which many instructions are executed repeatedly. These instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other.

✓ The cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

✓ When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection

of rules for making this decision constitutes the cache‟s replacement algorithm.

- ✓
- ✓ Cache Hits
- ✓ The processor does not need to know explicitly about the existence of the cache. It simply issues Read andWrite requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache.
- ✓ If it does, the Read orWrite operation is performed on the appropriate cache location. In this case, a read
- ✓ or write hit is said to have occurred.
- ✓
- ✓ Cache Misses
- ✓ A Read operation for a word that is not in the cache constitutes a Read miss. It causes the block of words containing the requested word to be copied from the main memory into the cache.
- ✓
- ✓ Cache Mapping:
- ✓ There are three different types of mapping used for the purpose of cache memory which are as follows: Direct mapping, Associative mapping, and Set-Associative mapping. These are explained as following below.

CACHE SIZE VS BLOCK SIZE

Cache Size:

- Represents the total capacity of the cache memory, which is a small, fast memory used to store frequently accessed data for quicker retrieval.
- A larger cache size can store more data, potentially leading to faster access times and improved system performance.
- Cache size is measured in bytes, kilobytes (KB), megabytes (MB), or gigabytes (GB).
- Examples of cache sizes include 32KB, 512KB, 1MB, or 8MB.
  Block Size (Cache Line Size):
- Determines the amount of data that is transferred between the cache and main memory in a single operation.
- When the CPU needs data that is not in the cache (a "cache miss"), the entire block containing that data is fetched from main memory and stored in the cache.
- Larger block sizes can lead to better spatial locality (more data is likely to be accessed near the requested data), but also mean that more data is fetched on a cache miss, potentially wasting space if only a small portion of the block is needed.
- Smaller block sizes can lead to more cache misses, but also mean that only the necessary data is fetched, potentially saving space and improving performance.
- Common block sizes in modern CPU architectures are 64 bytes.

Cache Mapping:

There are three different types of mapping used for the purpose of cache memory which are as follows: Direct mapping, Associative mapping, and Set-Associative mapping. These are explained as following below.

Direct mapping

The simplest way to determine cache locations in which to store memory blocks is the direct- mapping technique. In this technique, block j of the main memory maps onto block j modulo 128 of the cache, as depicted in Figure 8.16. Thus, whenever one of the main memory

blocks 0, 128, 256, . . . is loaded into the cache, it is stored in cache block 0. Blocks 1, 129, 257, . . . are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full.

For example, instructions of a program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed, both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block.

With direct mapping, the replacement algorithm is trivial. Placement of a block in the cache is determined by its memory address. The memory address can be divided into three fields, as shown in Figure 8.16. The low-order 4 bits select one of 16 words in a block.

When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache.

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

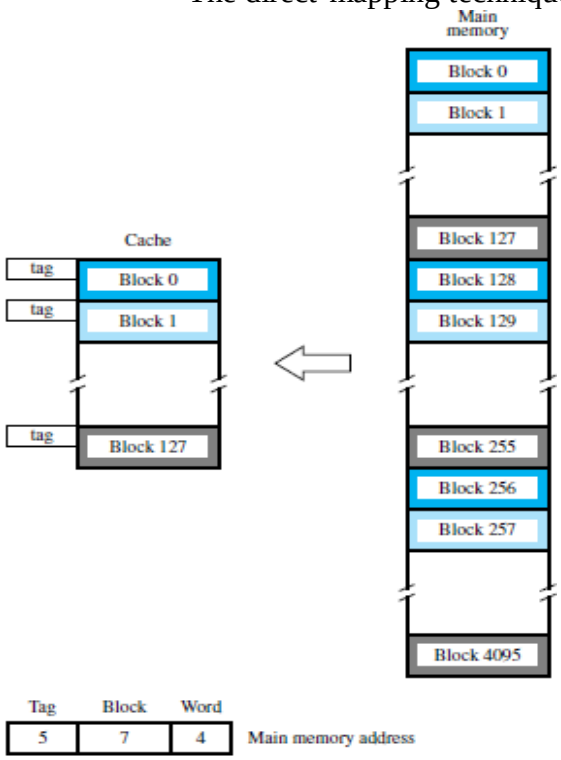


Figure 8.16 Direct-mapped cache.

Associative Mapping

In Associative mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the *associative-mapping* technique.

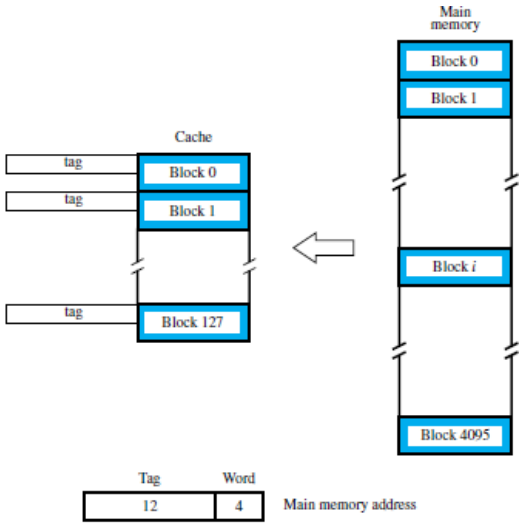


Figure 8.17 Associative-mapped cache.

It gives complete freedom in choosing the cache location in which to place the memory block, resulting in a more efficient use of the space in the cache. When a new block is brought into the cache, it replaces (ejects) an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced.

o avoid a long delay, the tags must be searched in parallel. A search of this kind is called an *associative search.*

Set-Associative Mapping

Another approach is to use a combination of the direct- and associative-mapping techniques. The blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement.

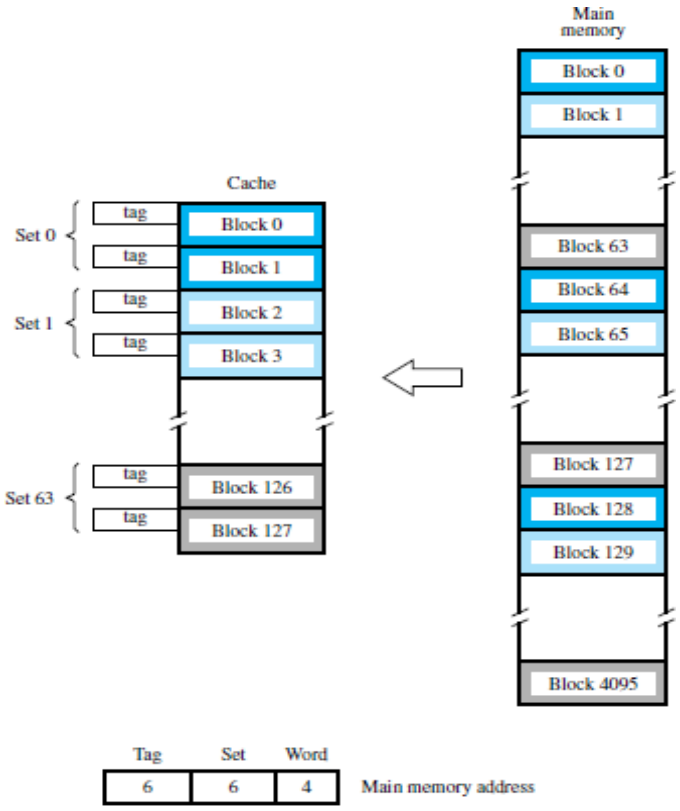


Figure 8.18 Set-associative-mapped cache with two blocks per set.

At the same time, the hardware cost is reduced by decreasing the size of the associative search.An example of this set-associative-mapping technique is shown in Figure 8.18 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, . . . , 4032 map into cache set 0, and they can occupy either of the two block positions within this set.

Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

The number of blocks per set is a parameter that can be selected to suit the requirements of a articular computer. For the main memory and cache sizes in Figure 8.18, four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully-associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping.

Replacement Algorithms

In a direct-mapped cache, the position of each block is predetermined by its address; hence, the replacement strategy is trivial. In associative and set-associative caches there exists some flexibility.

When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite.

This is an important issue, because the decision can be a strong determining factor in system performance. In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. But, it is not easy to determine which blocks are about to be referenced.

The property of locality of reference in programs gives a clue to a reasonable strategy. Because program execution usually stays in localized areas for reasonable periods of time, there is a high probability that the blocks that have been referenced recently will be referenced again soon. Therefore, when a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the *least recently used* (LRU) block, and the technique is called the *LRU replacement algorithm*.

The LRU algorithm has been used extensively. Although it performs well for many access patterns, it can lead to poor performance in some cases.

Write Policies

The write operation is proceeding in 2 ways.

- Write-through protocol
- Write-back protocol

Write-through protocol:

Here the cache location and the main memory locations are updated simultaneously.

Write-back protocol:

- This technique is to update only the cache location and to mark it as with associated flag bit called dirty/modified bit.
- The word in the main memory will be updated later, when the block containing this marked word is to be removed from the cache to make room for a new block.
- To overcome the read miss Load –through / Early restart protocol is used.