

COMPUTER ORGANIZATION

UNIT I

Flip-flops – Registers and shift registers – counters – decoders – Multiplexers – PLDs – sequential circuits. Basic Structure of Computers. Functional UNITS – Basic operational concepts – Bus structures – performance – Multiprocessors and Multi computers – Historical Perspective.

UNIT II

Addressing Methods and Machine Program Sequencing: 1. Basic Concepts: –Memory locations and address, Main Memory operations, Instructions and Instruction Sequencing Addressing Modes.

UNIT III

Input / Output organization: Accessing I/O Devices – Interrupts – Direct Memory Access-I/O Hardware Standard I/O Interface.

UNIT IV

Memory System Concepts: – Semiconductor RAM Memories - Read only memories – Cache Memories – Performance Considerations –Virtual Memories: - Memory Management Requirements, Arithmetic: - Addition and subtraction of sign members – Design of fast adders – Multiplication of positive members – Signed operand multiplication – Fast multiplication – Integer division – Floating point numbers and operations.

UNIT V

Basic Processing UNIT: Concepts – execution of a complete instruction – Multiple – Bus organization – Hardware control – Micro Programmed Control. Pipelining: Concepts – Data hazards – Instruction hazards – Influence on Instruction sets - data path and control constructions.

Text Book: 1. Hamacher C, Vranesic Z, and Zaky S. Computer Organization, 5th edition, Mc Graw – Hill, 2002. 2. 1. Stallings W, Computer Organization and Architecture, 6th edition. Parson Education, 2003.

Reference Books: 1. Yarbrough JM, Digital Logic – Applications and Design, Thomas Learning, 1997

UNIT-I

Flip-flops:

The flip-flop is a circuit that maintains a state until directed by input to change the state. A basic flip-flop can be constructed using four-NAND or four-NOR gates. A flip-flop is popularly known as the basic digital memory circuit. It has its two states as logic 1(High) and logic 0 (Low) states.

- A flip-flop is a sequential circuit which consists of a single binary state of information or data.
- The digital circuit is a flip-flop which has two outputs and are of opposite states. It is also known as a Bistable Multivibrator.

Types of Flip-Flops

Given Below are the Types of Flip-Flop

- SR Flip Flop
- JK Flip Flop
- D Flip Flop
- T Flip Flop

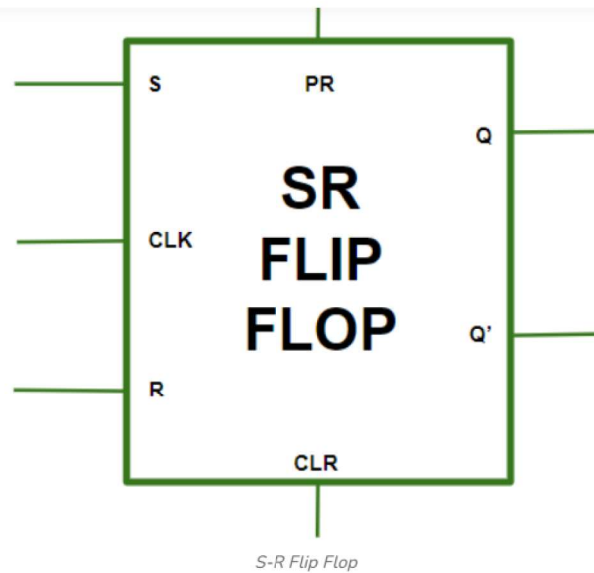
Logic diagrams and truth tables of the different types of flip-flops are as follows:

S-R Flip Flop:

In the flip flop, with the help of preset and clear when the power is switched ON, the states of the circuit keeps on changing, that is it is uncertain. It may come to set($Q=1$) or reset($Q'=0$) state. In many applications, it is desired to initially set or reset the flip flop that is the initial state of the flip flop that needs to be assigned. This thing is accomplished by the preset(PR) and the clear(CLR).

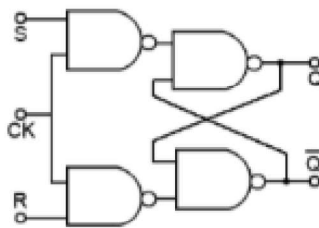
Block Diagram of S-R Flip Flop

Given Below is the Block Diagram of S-R Flip Flop



Circuit Diagram and Truth Table of S-R Flip Flop

Given Below is the Diagram of S-R Flip Flop with its Truth Table



TRUTH TABLE

S	R	Q_n	Q_{n-1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	-
1	1	1	-

Operations of S-R Flip Flop

Given Below is the Operations of S-R Flip Flop

- **Case 1($PR=CLR=1$):** The asynchronous inputs are inactive and the flip flop responds freely to the S,R and the CLK inputs in the normal way.
- **Case 2($PR=0$ and $CLR=1$):** This is used when the Q is set to 1.

- **Case 3(PR=1 and CLR=0):** This is used when the Q' is set to 1.
- **Case 4(PR=CLR=0):** This is an invalid state.

Characteristics Equation for SR Flip Flop

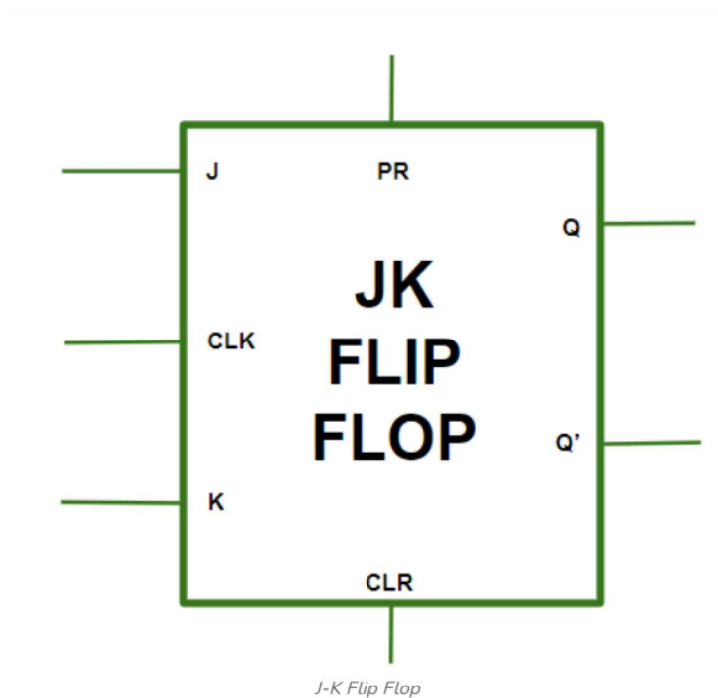
$$Q_{N+1} = QNR' + SR'$$

J-K Flip Flop

In JK flip flops, The basic structure of the flip flop which consists of Clock (CLK), Clear (CLR), Preset (PR).

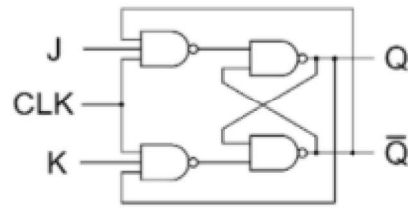
Block Diagram of J-K Flip Flop

Given Below is Block Diagram of J-K Flip Flop



Circuit Diagram and Truth Table of J-K Flip Flop

Given Below is the Diagram of J-K Flip Flop with its Truth Table



TRUTH TABLE

J	K	Q_n	Q_{n+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Operations of J-K Flip Flop

Given Below is the Operations of J-K Flip Flop

- **Case 1 (PR=CLR=0):** This condition is in its invalid state.
- **Case 2 (PR=0 and CLR=1):** The PR is activated which means the output in the Q is set to 1. Therefore, the flip flop is in the set state.
- **Case 3 (PR=1 and CLR=0):** The CLR is activated which means the output in the Q' is set to 1. Therefore, the flip flop is in the reset state.
- **Case 4 (PR=CLR=1):** In this condition the flip flop works in its normal way whereas the PR and CLR gets deactivated.

Race Around Condition in J-K Flip Flop

When the J and K both are set to 1, the input remains high for a longer duration of time, then the output keeps on toggling. Toggle means that switching in the output instantly i.e. $Q=0$, $Q'=1$ will immediately change to $Q=1$ and $Q'=0$ and this continuation keeps on changing. This change in output leads to race around condition.

Characteristics Equation for JK Flip Flop

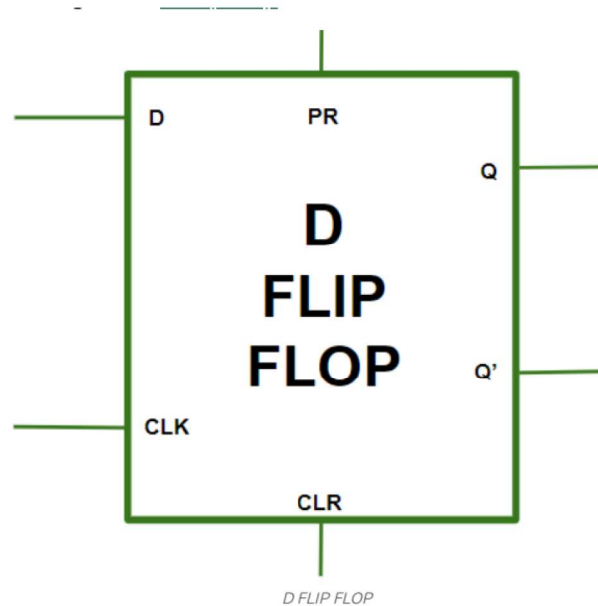
$$Q_{n+1} = JQ'N + K'QN$$

D Flip Flop

The D Flip Flop Consists a single data input(D), a clock input(CLK),and two outputs: Q and Q' (the complement of Q).

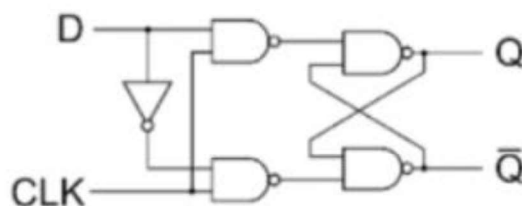
Block Diagram of D Flip Flop

Given Below is the Block Diagram of D Flip Flop



Circuit Diagram and Truth Table of D Flip Flop

Given Below is the Diagram of D Flip Flop with its Truth Table



Q	D	Q(t+1)
0	0	0
0	1	1
1	0	0
1	1	1

Operation of the D Flip-Flop

Given Below is the operation of D Flip-Flop

- **Case 1 (PR=CLR=0):** This condition represents an invalid state where both PR(present) and CLR(clear) inputs are inactive.
- **Case 2 (PR=0 and CLR=1):** This state is set state in which PR is inactive (0) and CLR is active(1) and the output Q is set to 1.
- **Case 3 (PR=1 and CLR=0):** This state is reset state in which PR is active (1) and CLR is inactive (0) and the complementary output Q' is set to 1.
- **Case 4 (PR=CLR=1):** In this state the flip flop behaves as normal, both PR and CLR inputs are active(1).

Characteristics Equation for D Flip Flop

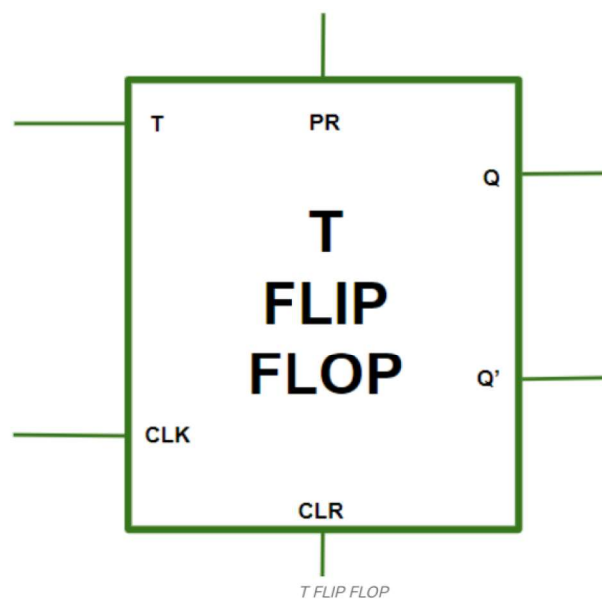
$$Q_{N+1} = D$$

T Flip Flop

The T Flip Flop consists of data input (T), a clock input (CLK), and two outputs: Q and Q' (the complement of Q).

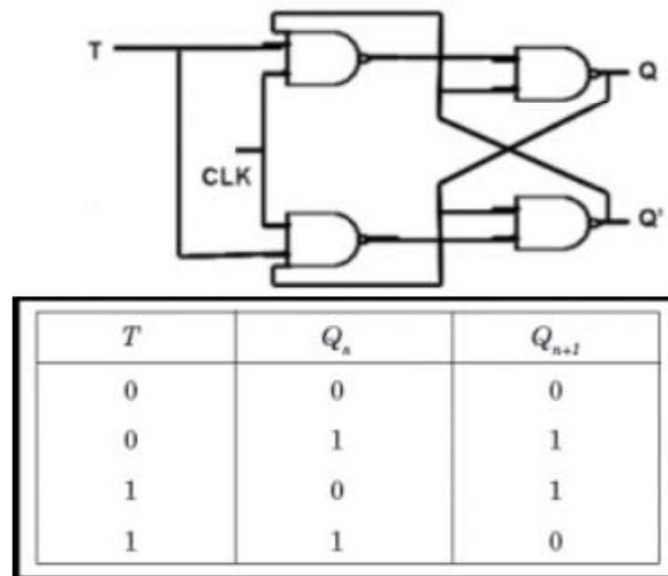
Block Diagram of T Flip Flop

Given Below is the Block Diagram of T Flip Flop



Circuit Diagram and Truth Table of T Flip Flop

Given Below is the Circuit Diagram and Truth Table of T Flip Flop



Operation of the T Flip-Flop

Given Below is the Operation of T Flip-Flop

- **Case 1 (T=0):** In this condition the flip-flop remains in its current state regardless of clock input, Also the Output Q will remain unchanged until the value of T will not change.
- **Case 2 (T=1):** In this condition the flip flop will change when T input is 1, At each rising or falling edge of the clock signal the output Q will be in complementary state.

Characteristics Equation for T Flip Flop

$$Q_{n+1} = Q'NT + QNT' = QN \text{ XOR } T$$

Applications of Flip-Flops

These are the various types of flip-flops being used in digital electronic circuits and the applications of Flip-flops are as specified below.

- **Counters:** The Flip Flop are used in the [Counter](#) Circuits for Counting pulse or events.
- **Frequency Dividers:** The Flip Flop are used in [Frequency Dividers](#) to divide the frequency of a input signal by a specific factor.
- **Shift Registers:** The [Shift registers](#) consist of interconnected flip-flops that shift data serially.
- **Storage Registers:** The Storage Register uses Flip Flop to store data in binary information.
- **Bounce elimination switch:** The Flip Flop are used in Bounce elimination switch to eliminate the contact bounce.

- **Data storage:** The Flip Flop are used in the Data Storage to store binary data temporarily or permanently.
- **Data transfer:** The Flip Flops are used for data transfer in different [electronic parts](#).
- **Latch:** The Latches are the [Sequential circuit](#) which uses Flip Flop for temporary storage of data
- **Registers:** The [Registers](#) are mode from the array of flip flop which are used to store data temporarily.
- **Memory:** The Flip Flops are the main components in the [memory unit](#) for data storage.

Registers and shift registers:

Registers: A register is a tiny, fast storage memory within the central processing unit (CPU) or the [arithmetic logic unit](#) (ALU) of a computer. Registers are utilized for a variety of functions in handling and controlling instructions and data and play an important role in the operation of a computer's CPU.

Types of Registers

There are different types of registers listed below as per the different parameters :

- Accumulator Register
- Program Counter (PC) Register
- General-Purpose Registers
- Instruction Register (IR)
- Memory Address Register (MAR)
- Memory Data Register (MDR)
- Stack Pointer (SP)
- Floating-Point Registers
- Index Register (IR)
- Memory Buffer Register (MBR)
- Data Register (DR)

Accumulator Register

The [accumulator](#) acts as a central point for arithmetic and logical operations within the CPU. It fetches data from memory and stores intermediate results during calculations. Arithmetic operations such as addition, subtraction, multiplication, and division often take place in the accumulator. The final result may be stored in the accumulator or transferred to other registers or memory locations.

Program Counter (PC) Register

The [program counter](#) is a special register that keeps track of the memory address of the next instruction to be fetched and executed. As the CPU executes each instruction in sequence the program counter is updated to indicate the next instruction's address in memory. This process continues until the program's execution is complete.

General-Purpose Registers

General-purpose registers are versatile because they can hold data and memory addresses. They are used for various calculations and data manipulation tasks during program execution. General-purpose registers are essential for performing arithmetic and logical operations on data stored in the CPU.

Instruction Register (IR)

The [instruction register](#) holds the currently fetched instruction from memory. It allows the CPU to decode and execute the instruction based on its opcode and operands.

Memory Address Register (MAR)

The [memory address register](#) stores the memory address of data or instructions to be accessed or written in memory. It plays a crucial role in memory operations by indicating the location of the data or instruction the CPU needs to access.

Memory Data Register (MDR)

The Memory Data Register holds the actual data fetched from or written to memory. When the CPU retrieves data from memory, it is temporarily stored in the MDR before being processed further.

Stack Pointer (SP)

The [stack pointer](#) is used in stack-based memory operations. It keeps track of the top of the stack, which is a region of memory used for temporary storage of data and return addresses during function calls.

Floating-Point Registers

The Floating-point registers are specialized for handling floating-point numbers and performing floating-point arithmetic operations. These registers can store and manipulate floating-point numbers with higher precision.

Index Register (IR)

The Index Register is a kind of useful instrument in the computer that tracks specific locations of data. It is like a small helper that makes it easy for the computer to get to where memory is stored quickly and process it.

Memory Buffer Register (MBR)

The Memory Buffer Register can be viewed as a temporary store in the computer where data takes time off when it is being transferred from one place to another by the CPU. This register can be seen as some kind of traffic police man, ensuring smooth movements of information between CPU and memory without any bumps.

Data Register (DR)

The Data Register is an example of a fast storage place in the computer where information rests while all these processes take place on the same machine. It's a location where transient amounts of data are kept within the system for temporary usage like pit stops within then performing calculations or tasks or others.

Applications of Registers

- **Arithmetic and Logic Operations:** The Registers are extensively used during arithmetic and logic operations in the CPU. They temporarily store operands, intermediate results, and flags facilitating quick and efficient calculations.
- **Instruction Execution:** The Registers play a crucial role in the execution of machine instructions. They hold instructions and data needed for immediate processing ensuring smooth program execution.
- **CPU Pipelining:** The [pipelining](#) is employed to increase performance by overlapping the execution of multiple instructions. Registers help in holding intermediate results and data between different stages of the pipeline.
- **Context Switching:** the CPU switches between different processes or threads. it needs to save and restore the context of each process. Registers are used to store the CPU's state including the program counter stack pointer and general-purpose registers during [context switching](#).

- **Memory Addressing:** The memory address register (MAR) and memory data register (MDR) are involved in memory operations. MAR holds the memory address being accessed. while MDR temporarily stores data fetched from or written to memory.
- **Input/Output Operations:** The Registers are used to buffer data during input/output operations. Input data is stored in input registers and output data is temporarily held in output registers before being processed further.
- **Floating-Point Operations:** To Specialized [floating-point](#) registers are used for handling floating-point arithmetic operations. These registers can store and manipulate floating-point numbers with higher precision.
- **Control Unit Operations:** The Registers play a critical role in the control unit of the [CPU](#). They hold control signals and flags that determine the control flow and sequencing of the instructions during program execution.

Advantages and Disadvantages of Registers

Advantages and Disadvantages of Registers are given below :

Advantages of Registers

- **Speed:** The Registers offer fast access times due to their proximity to the CPU, enhancing overall system performance.
- **Data Processing Efficiency:** They enable quick data manipulation, reducing the need to access slower main memory frequently.

Disadvantages of Registers

- **Limited Capacity:** The Registers have a small size, restricting the amount of data they can hold at a time.
- **Cost:** The Registers are made from flip-flops and require more hardware, contributing to the overall cost of the processor.

shift register:

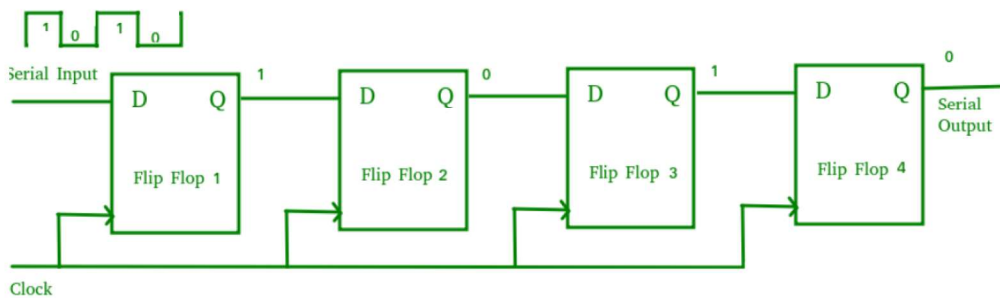
A shift register is a group of flip-flops connected in series to store and shift multiple bits of data either left or right with clock pulses.

- The registers which will shift the bits to the left are called “Shift left registers”.
- The registers which will shift the bits to the right are called “Shift right registers”.

Serial-In Serial-Out Shift Register (SISO)

A Serial-In Serial-Out (SISO) shift register accepts data one bit at a time through a single input line and outputs data serially, using a series of flip-flops all triggered by the same clock. The main use of a SISO is to act as a delay element.

- Data enters and leaves one bit at a time in a serial sequence.
- Typically made of multiple D flip-flops connected in series and synchronised by a common clock.

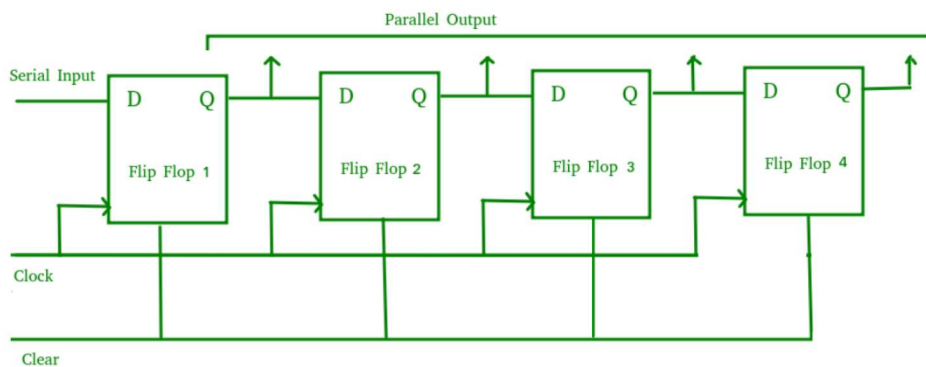


Serial-In Serial-Out Shift Register (SISO)

Serial-In Parallel-Out Shift Register (SIPO)

A Serial-In Parallel-Out (SIPO) shift register accepts data serially through a single input line and produces all bits simultaneously as a parallel output, using multiple synchronized flip-flops. The main use of the SIPO register is to convert serial data into parallel data.

- Data is input one bit at a time and outputs all stored bits in parallel.
- Consists of connected D flip-flops with a common clock and a clear (CLR) signal to reset all flip-flops.

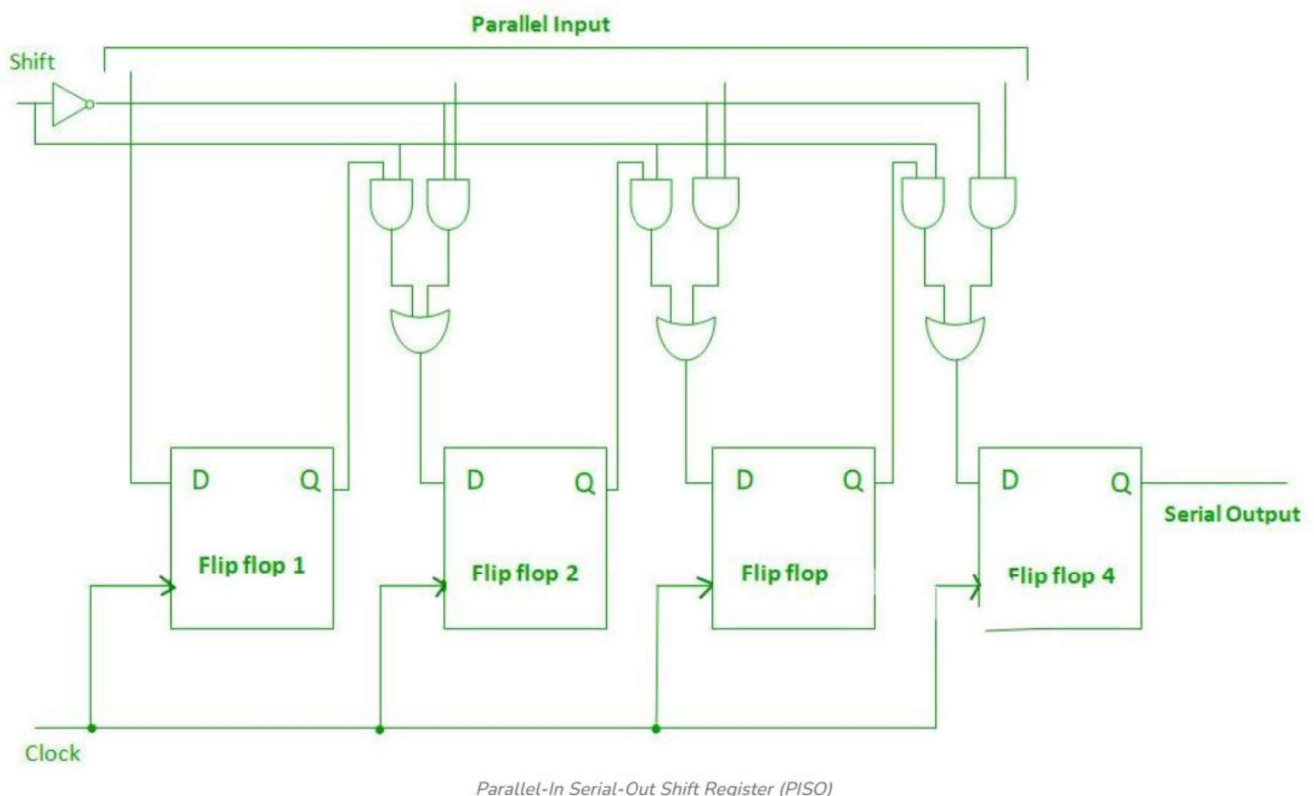


Serial-In Parallel-Out shift Register (SIPO)

Parallel-In Serial-Out Shift Register (PISO)

A Parallel-In Serial-Out (PISO) shift register accepts parallel data simultaneously into each flip-flop and shifts it out serially through a single output line using synchronized flip-flops and multiplexers. It is used to convert parallel data to serial data.

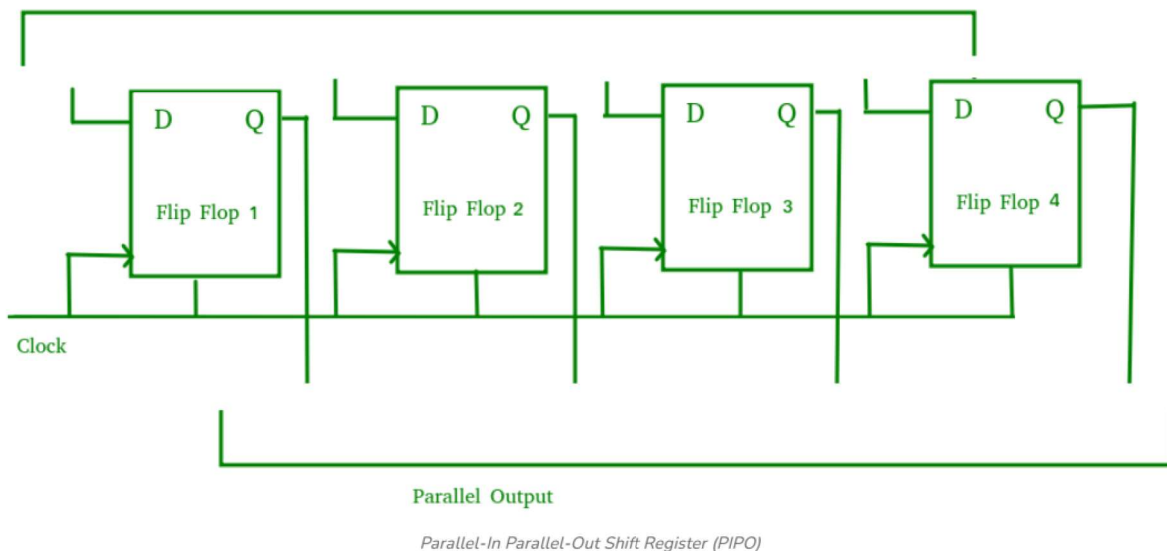
- Data is loaded in parallel and then shifted out one bit at a time serially.
- Consists of D flip-flops with inputs controlled by multiplexers selecting between parallel data and previous flip-flop output, all clocked together.



Parallel-In Parallel-Out Shift Register (PIPO)

A Parallel-In Parallel-Out (PIPO) shift register accepts and outputs data simultaneously across all flip-flops without any serial shifting, using synchronized control signals. It is used as a temporary storage device and also acts as a delay element.

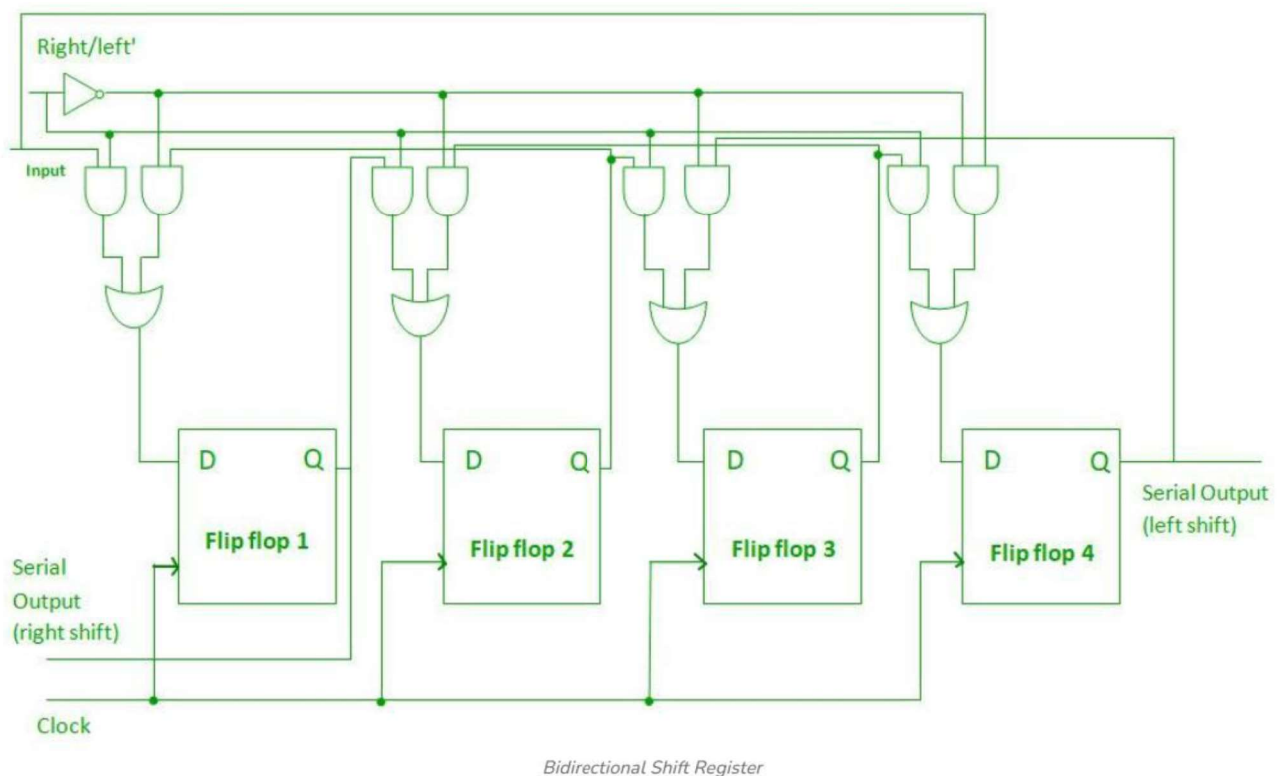
- Data is loaded and retrieved in parallel from each flip-flop at the same time.
- Includes D flip-flops with no interconnections, controlled by common clock and clear (CLR) signals.



Bidirectional Shift Register

A Bidirectional Shift Register can shift binary data either to the left or right, enabling multiplication or division by 2 based on the selected mode.

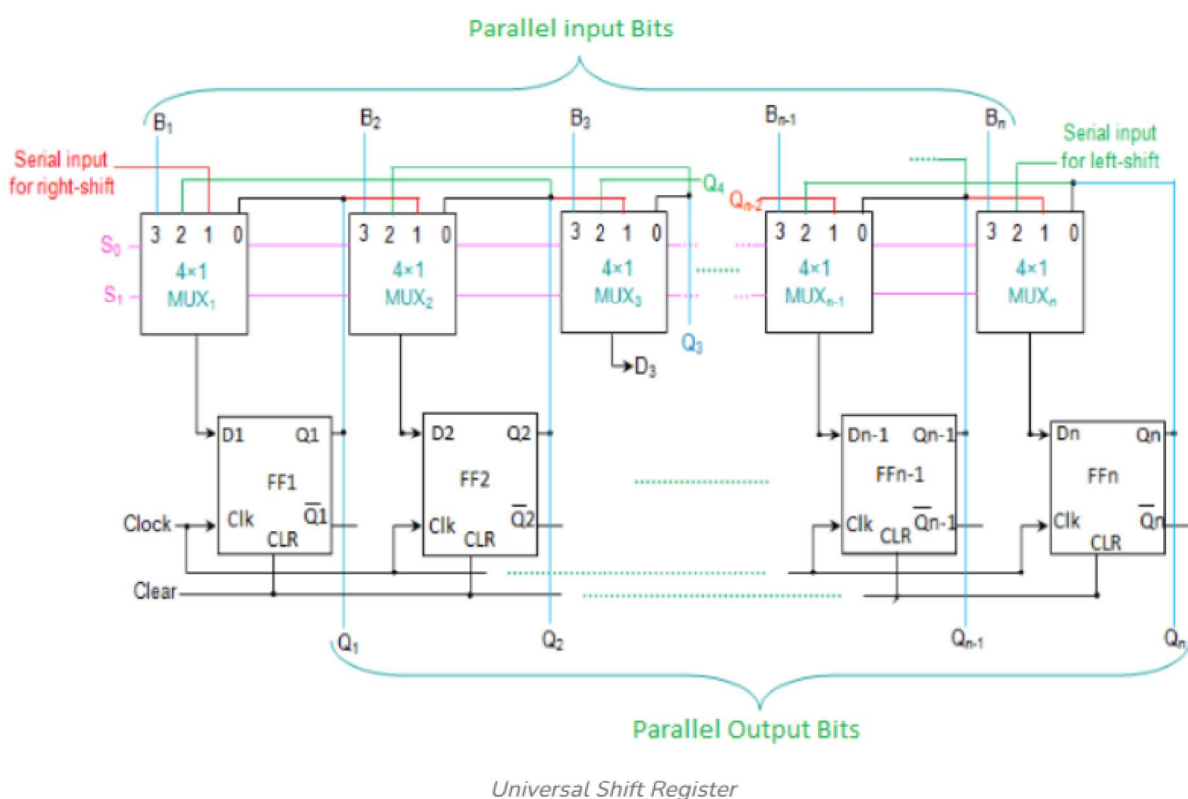
- Mode 1 (high) shifts data right (division by 2); Mode 0 (low) shifts data left (multiplication by 2).
- Composed of four D flip-flops with inputs at both ends and gates controlled by mode signal to enable directional shifting.



Universal Shift Register

A Universal Shift Register is an N-bit register that supports both left and right shifting as well as parallel loading, using N flip-flops and N multiplexers with shared select lines to control the data flow.

- It combines the features of bidirectional and unidirectional shift registers, enabling versatile data manipulation (shift left, shift right, parallel load, and hold).
- Each flip-flop is connected to a multiplexer, and all multiplexers are controlled by common select lines to determine the register's operation mode.



Shift Register Counter

Shift Register Counters are the shift registers in which the outputs are connected back to the inputs in order to produce particular sequences. There are basically two types:

- Ring Counter
- Johnson Counter

Ring Counter

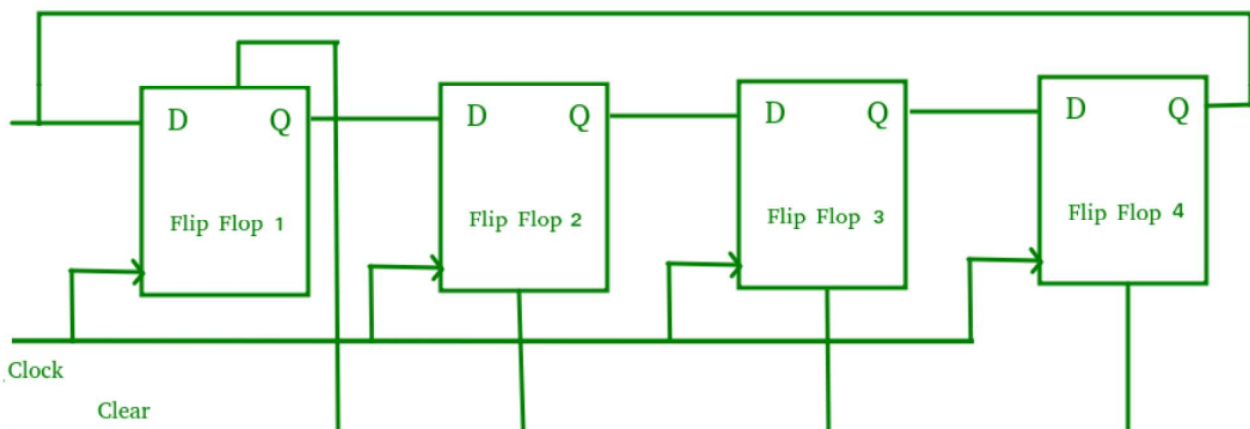
A ring counter is basically a shift register counter in which the output of the first flip-flop is connected to the next flip-flop and so on and the output of the last flip-flop is again fed back to the input of the

first flip-flop. The data pattern within the shift register will circulate as long as clock pulses are applied.

Clock Pulse	Q1	Q2	Q3	Q4
0	1	0	0	1
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1

Ring Counter Truth Table

The circuit consists of four D flip-flops which are connected. The data pattern will repeat after every four clock pulses as shown in the truth table. A Ring counter is generally used because it is self-decoding.



Ring Counter

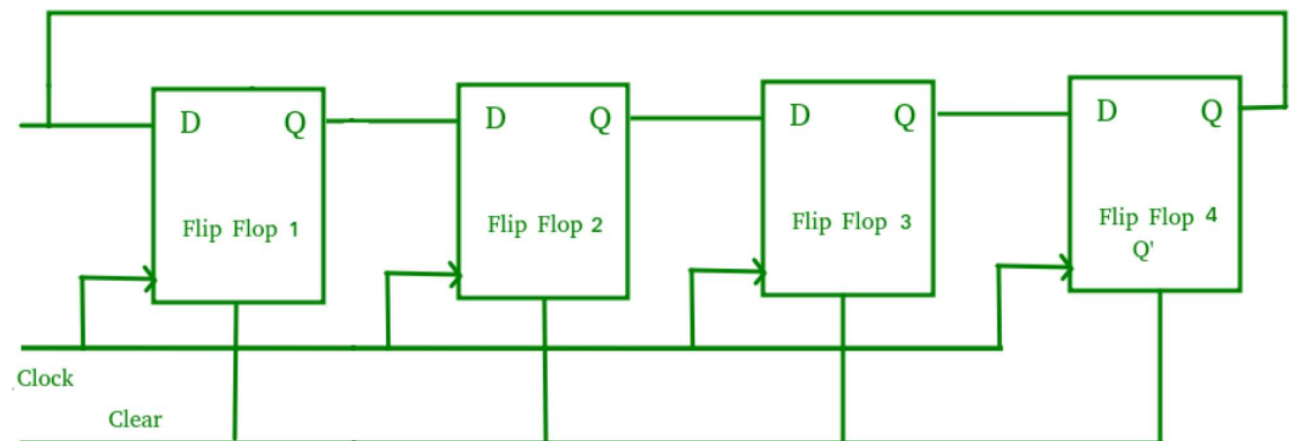
Johnson Counter

A Johnson counter is basically a shift register counter in which the output of the first flip flop is connected to the next flip flop and so on and the inverted output of the last flip flop is again fed back to the input of the first flip flop. They are also known as twisted ring counters.

Clock Pulse	Q1	Q2	Q3	Q4
0	0	0	0	1
1	0	0	0	0
2	1	0	0	0
3	1	1	0	0
4	1	1	1	0
5	1	1	1	1
6	0	1	1	1
7	0	0	1	1

Johnson Counter Truth Table

An n-stage Johnson counter yields a count sequence of $2n$ different states, thus also known as a mod- $2n$ counter. The circuit consists of four D flip-flops which are connected, the data pattern will repeat every eight clock pulses. It only needs n number of flip-flops to generate a sequence of $2n$ states.



Johnson Counter

Decoders:

A decoder does the opposite job of an encoder. It is a combinational circuit that converts n lines of input into 2

n

lines of output. Let's take an example of 3-to-8 line decoder.

Truth Table -

X	Y	Z	D0	D1	D2	D3	D4	D5	D6	D7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Implementation -

D0 is high when $X = 0$, $Y = 0$ and $Z = 0$. Hence,

$$D0 = X' Y' Z'$$

Similarly,

$$D1 = X' Y' Z$$

$$D2 = X' Y Z'$$

$$D3 = X' Y Z$$

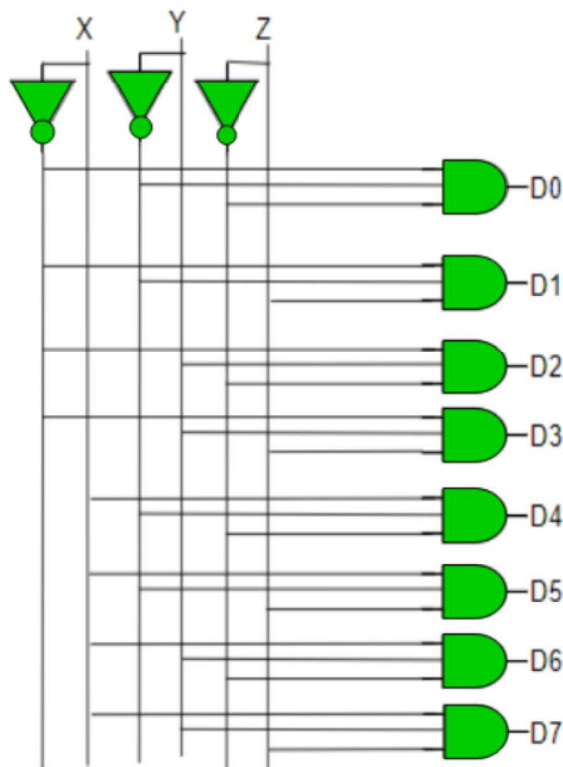
$$D4 = X Y' Z'$$

$$D5 = X Y' Z$$

$$D6 = X Y Z'$$

$$D7 = X Y Z$$

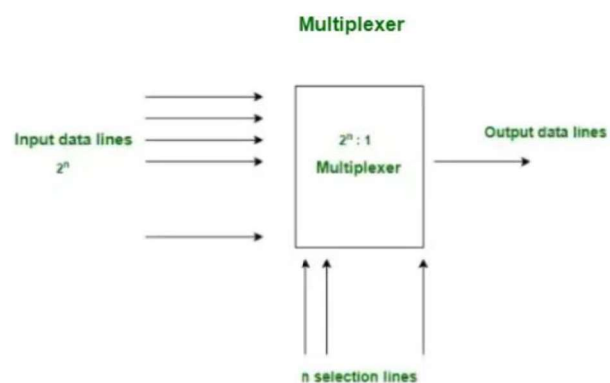
Hence,



Multiplexers:

A multiplexer is a combinational circuit that has many data inputs and a single output, depending on control or select inputs. For N input lines, $\log_2(N)$ selection lines are required, or equivalently, for 2^n input lines, n selection lines are needed.

- Multiplexers are also known as "N-to-1 selectors," parallel-to-serial converters, many-to-one circuits, and universal logic circuits.
- They are mainly used to increase the amount of data that can be sent over a network within a certain amount of time and bandwidth.



Types of Mux

The Mux can be of different types based on input but in this article, we will go through two major types of mux, which are

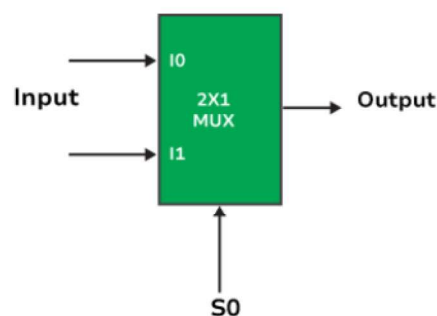
- **2x1 Mux**
- **4x1 Mux**

2x1 Multiplexer

The 2x1 is a fundamental circuit which is also known 2-to-1 multiplexer that are used to choose one [signal](#) from two inputs and transmits it to the output. The 2x1 mux has two input lines, one output line, and a single selection line. It has various applications in digital systems such as in microprocessor it is used to select between two different data sources or between two different instructions.

Block Diagram of 2:1 Multiplexer with Truth Table

Given Below is the Block Diagram and Truth Table of 2:1 Mux. In this Block Diagram where I_0 and I_1 are the input lines, Y is the output line and S_0 is a single select line.



Truth Table

S_0	I_0	I_1	Y
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

Block Diagram of 2:1 Multiplexer with Truth Table

The output of the 2x1 Mux will depend on the selection line S_0 ,

- When S is 0(low), the I_0 is selected

- when S0 is 1(High), I1 is selected

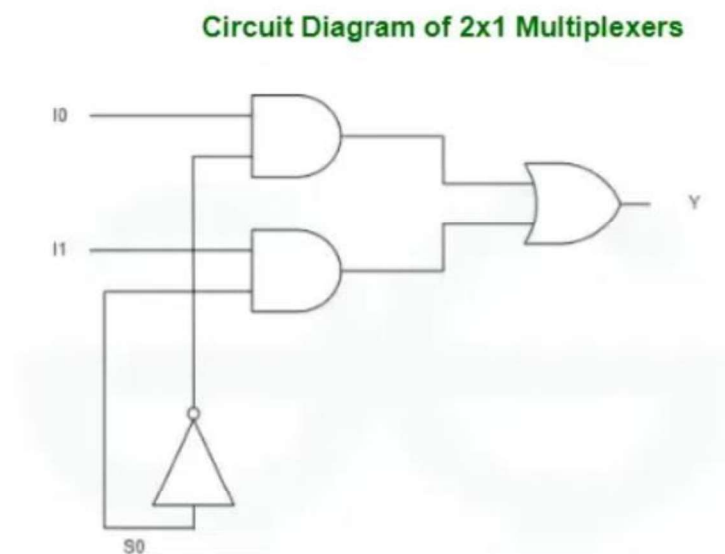
Logical Expression of 2x1 Mux

Using the Truth Table ,the Logical Expression for Mux can be determined as

$$Y = \bar{S}_0 \cdot I_0 + S_0 \cdot I_1$$

Circuit Diagram of 2x1 Multiplexers

Using truth table the [circuit](#) diagram can be given as

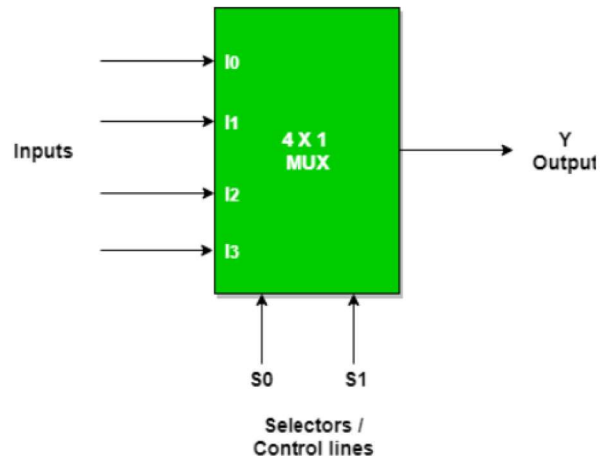


4x1 Multiplexer

The 4x1 Multiplexer which is also known as the 4-to-1 multiplexer. It is a multiplexer that has 4 inputs and a single output. The Output is selected as one of the 4 inputs which is based on the selection inputs. The number of the Selection lines will depend on the number of the input which is determined by the equation $\log_2 n$,In 4x1 Mux the selection lines can be determined as $\log_4 = 2$,slo two selections are needed.

Block Diagram of 4x1 Multiplexer

In the Given Block Diagram I0, I1, I2, and I3 are the 4 inputs and Y is the Single output which is based on Select lines S0 and S1.



The output of the multiplexer is determined by the binary value of the selection lines

- When $S_1S_0=00$, the input I_0 is selected.
- When $S_1S_0=01$, the input I_1 is selected.
- When $S_1S_0=10$, the input I_2 is selected.
- When $S_1S_0=11$, the input I_3 is selected.

Truth Table of 4×1 Multiplexer

Given Below is the [Truth Table](#) of 4x1 Multiplexer

Truth Table

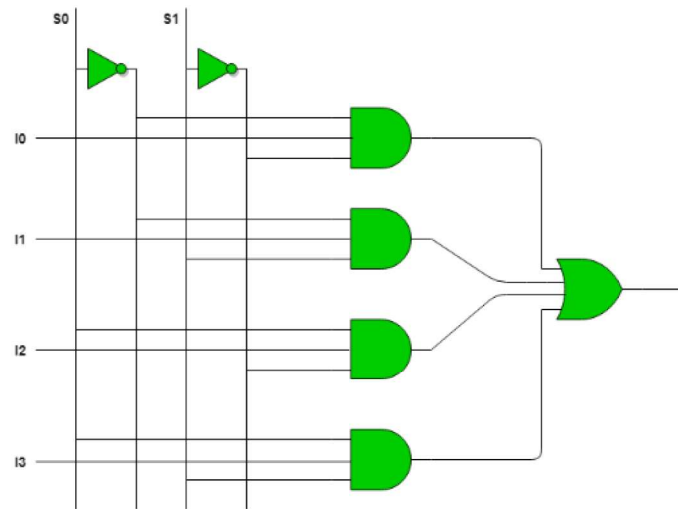
S_0	S_1	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

So, final equation,

$$Y = S_0'.S_1'.I_0 + S_0'.S_1.I_1 + S_0.S_1'.I_2 + S_0.S_1.I_3$$

Circuit Diagram of 4x1 Multiplexers

Using truth table the circuit diagram can be given as



Multiplexer can act as universal combinational circuit. All the standard logic gates can be implemented with multiplexers.

Applications of MUX

- **Data Routing** : The Mux is used for data routing in the digital system where they select one of the several data lines and re-route it the output.
- **Data Selection** : The Mux is used for data selection where they select data source according to the select lines.
- **Analog-to-Digital Conversion** : The Mux are used in [ADC](#) to select different analog input channels.
- **Address Decoding** : The Mux are used in [Microprocessors](#) or memory for address decoding.
- **Logic Function Implementation** : Mux can be used to implement various logic functions.

PLDs:

An IC that contains large numbers of gates, flip-flops, etc. that can be configured by the user to perform different functions is called a Programmable Logic Device (PLD).

The internal logic gates and/or connections of PLDs can be changed/configured by a programming process.

One of the simplest programming technologies is to use fuses. In the original state of the device, all the fuses are intact.

Programming the device involves blowing those fuses along the paths that must be removed in order to obtain the particular configuration of the desired logic function.

PLDs are typically built with an array of AND gates (AND-array) and an array of OR gates (OR-array).

Advantages of PLDs:

Problems of using standard ICs:

Problems of using standard ICs in logic design are that they require hundreds or thousands of these ICs, considerable amount of circuit board space, a great deal of time and cost in inserting, soldering, and testing. Also require keeping a significant inventory of ICs.

Advantages of using PLDs:

Advantages of using PLDs are less board space, faster, lower power requirements (i.e., smaller power supplies), less costly assembly processes, higher reliability (fewer ICs and circuit connections means easier troubleshooting), and availability of design software.

There are three fundamental types of standard PLDs: PROM, PAL, and PLA.

A fourth type of PLD, which is discussed later, is the Complex Programmable Logic Device (CPLD), e.g., Field Programmable Gate Array (FPGA).

A typical PLD may have hundreds to millions of gates.

In order to show the internal logic diagram for such technologies in a concise form, it is necessary to have special symbols for array logic.

The PLA (Programmable Logic Array):

In PLAs, instead of using a decoder as in PROMs, a number (k) of AND gates is used where $k < 2^n$, (n is the number of inputs).

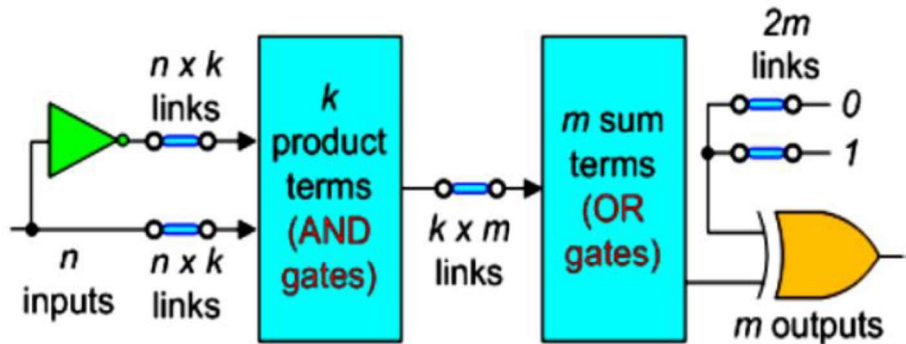
Each of the AND gates can be programmed to generate a product term of the input variables and does not generate all the minterms as in the ROM.

The AND and OR gates inside the PLA are initially fabricated with the links (fuses) among them.

The specific Boolean functions are implemented in sum of products form by opening appropriate

links and leaving the desired connections.

A block diagram of the PLA is shown in the figure. It consists of **n** inputs, **m** outputs, and **k** product terms.



The product terms constitute a group of **k** AND gates each of $2n$ inputs.

Links are inserted between all **n** inputs and their complement values to each of the AND gates.

Links are also provided between the outputs of the AND gates and the inputs of the OR gates.

Since PLA has **m**-outputs, the number of OR gates is **m**.

The output of each OR gate goes to an XOR gate, where the other input has two sets of links, one connected to logic 0 and other to logic 1. It allows the output function to be generated either in the true form or in the complement form.

The output is inverted when the XOR input is connected to 1 (since $X \oplus 1 = X/$). The output does not change when the XOR input is connected to 0 (since $X \oplus 0 = X$).

Thus, the total number of programmable links is $2n \times k + k \times m + 2m$.

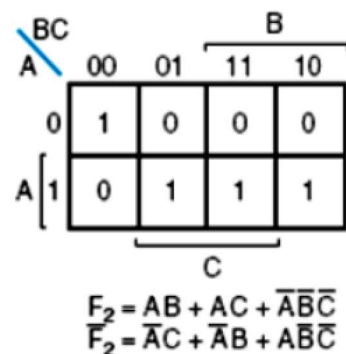
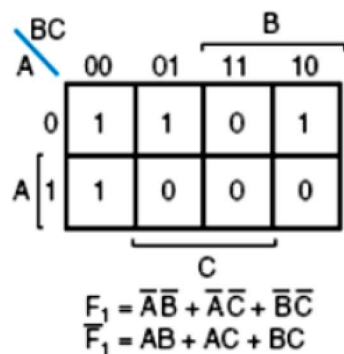
The size of the PLA is specified by the number of inputs (**n**), the number of product terms (**k**), and the number of outputs (**m**), (the number of sum terms is equal to the number of outputs).

Example:

Implement the combinational circuit having the shown truth table, using PLA.

A	B	C	F ₁	F ₂
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

Each product term in the expression requires an AND gate. To minimize the cost, it is necessary to simplify the function to a minimum number of product terms.



Designing using a PLA, a careful investigation must be taken in order to reduce the distinct product terms. Both the true and complement forms of each function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions.

The combination that gives a minimum number of product terms is:

$$F_1' = AB + AC + BC \text{ or } F_1 = (AB + AC + BC)'$$

$$F_2 = AB + AC + A'B'C'$$

This gives only 4 distinct product terms: AB, AC, BC, and A'B'C'.

So the PLA table will be as follows:

PLA programming table					
	Product term	Inputs A B C	Outputs		
			(C) F ₁	(T) F ₂	
AB	1	1 1 –	1	1	
AC	2	1 – 1	1	1	
BC	3	– 1 1	1	–	
$\overline{A}\overline{B}\overline{C}$	4	0 0 0	–	1	

or each product term, the inputs are marked with 1, 0, or – (dash). If a variable in the product term appears in its normal form (unprimed), the corresponding input variable is marked with a 1.

A 1 in the Inputs column specifies a path from the corresponding input to the input of the AND gate that forms the product term.

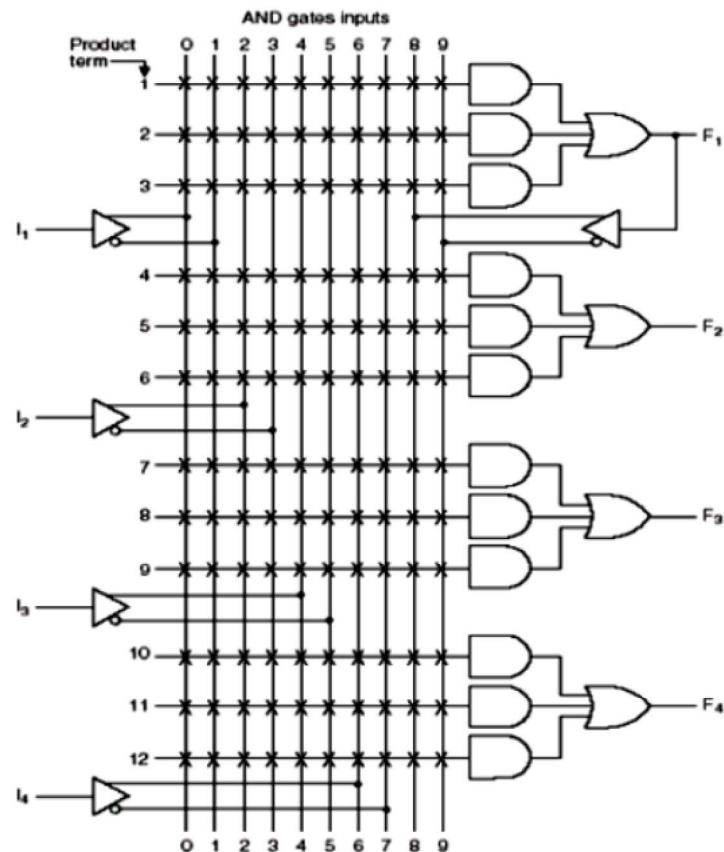
A 0 in the Inputs column specifies a path from the corresponding complemented input to the input of the AND gate. A dash specifies no connection.

The appropriate fuses are blown and the ones left intact form the desired paths. It is assumed that the open terminals in the AND gate behave like a 1 input.

The PAL (Programmable Array Logic):

The PAL device is a PLD with a fixed OR array and a programmable AND array.

As only AND gates are programmable, the PAL device is easier to program but it is not as flexible as the PLA.



The device shown in the figure has 4 inputs and 4 outputs. Each input has a buffer-inverter gate, and each output is generated by a fixed OR gate.

The device has 4 sections, each composed of a 3-wide AND-OR array, meaning that there are 3 programmable AND gates in each section.

Each AND gate has 10 programmable input connections indicating by 10 vertical lines intersecting each horizontal line. The horizontal line symbolizes the multiple input configuration of an AND gate.

One of the outputs **F1** is connected to a buffer-inverter gate and is fed back into the inputs of the AND gates through programmed connections.

Designing using a PAL device, the Boolean functions must be simplified to fit into each section.

The number of product terms in each section is fixed and if the number of terms in the function is too large, it may be necessary to use two or more sections to implement one Boolean function.

Example:

Implement the following Boolean functions using the PAL device as shown above:

$$W(A, B, C, D) = \Sigma m(2, 12, 13)$$

$$X(A, B, C, D) = \Sigma m(7, 8, 9, 10, 11, 12, 13, 14, 15)$$

$$Y(A, B, C, D) = \Sigma m(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$$

$$Z(A, B, C, D) = \Sigma m(1, 2, 8, 12, 13)$$

Simplifying the 4 functions to a minimum number of terms results in the following Boolean functions:

$$W = ABC' + A'B'CD'$$

$$X = A + BCD$$

$$Y = A'B + CD + B'D'$$

$$Z = ABC' + A'B'CD + AC'D' + A'B'C'D = W + AC'D' + A'B'C'D$$

Note that the function for Z has four product terms. The logical sum of two of these terms is equal to W. Thus, by using W, it is possible to reduce the number of terms for Z from four to three, so that the function can fit into the given PAL device.

The PAL programming table is similar to the table used for the PLA, except that only the inputs of the AND gates need to be programmed.

Product term	AND Inputs					Outputs
	A	B	C	D	W	
1	1	1	0	—	—	$W = \overline{ABC} + \overline{A}BCD$
2	0	0	1	0	—	
3	—	—	—	—	—	
4	1	—	—	—	—	$X = A + BCD$
5	—	1	1	1	—	
6	—	—	—	—	—	
7	0	1	—	—	—	$Y = \overline{A}B + CD + \overline{B}D$
8	—	—	1	1	—	
9	—	0	—	0	—	
10	—	—	—	—	1	$Z = W + \overline{A}CD + \overline{A}B\overline{C}D$
11	1	—	0	0	—	
12	0	0	0	1	—	

The figure shows the connection map for the PAL device, as specified in the programming table.

Sequential Circuits:

Sequential circuits are digital circuits that store and use previous state information to determine their next state. They are commonly used in digital systems to implement state machines, timers, counters, and memory elements and are essential components in digital systems design.

- Sequential circuits are commonly used in digital systems to implement state machines, timers, counters, and memory elements.
- The memory elements in sequential circuits can be implemented using [flip-flops](#), which are circuits that store binary values and maintain their state even when the inputs change.
- Unlike combinational circuits, which only depend on the current input values to produce outputs, sequential circuits depend on both the current inputs and the previous state stored in memory elements.

Sequential circuit is a [combinational logic circuit](#) that consists of inputs variable (X), logic gates (Computational circuit), and output variable (Z).



Figure: Combinational Circuits

A combinational circuit produces an output based on input variables only, but a **sequential circuit** produces an output based on **current input and previous output variables**. That means sequential circuits include memory elements that are capable of storing binary information. That binary information defines the state of the sequential circuit at that time. A [latch](#) capable of storing one bit of information.

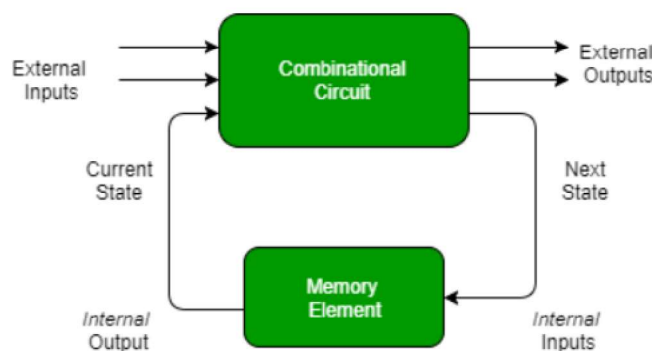


Figure: Sequential Circuit

As shown in the figure, there are two types of input to the combinational logic :

1. External inputs which are not controlled by the circuit.
2. Internal inputs, which are a function of a previous output state.

Secondary inputs are state variables produced by the storage elements, whereas secondary outputs are excitations for the storage elements.

Types of Sequential Circuits

There are two types of sequential circuits

Asynchronous Sequential Circuit

These circuits **do not use a clock signal** but uses the pulses of the inputs. These circuits are **faster** than synchronous sequential circuits because there is clock pulse and change their state immediately when there is a change in the input signal. We use asynchronous sequential circuits when speed of operation is important and **independent** of internal clock pulse.

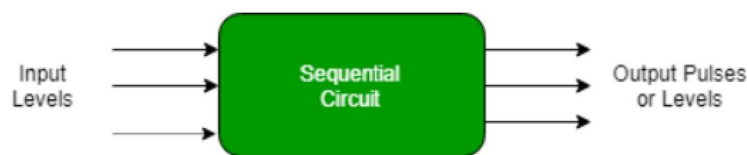


Figure: Asynchronous Sequential Circuit

But these circuits are more **difficult** to design and their output is **uncertain**.

Synchronous Sequential Circuit

These circuits **uses clock signal** and level inputs (or pulsed) (with restrictions on pulse width and circuit propagation). The output pulse is the same duration as the clock pulse for the clocked sequential circuits. Since they wait for the next clock pulse to arrive to perform the next operation, so these circuits are bit **slower** compared to asynchronous. Level output changes state at the start of an input pulse and remains in that until the next input or clock pulse.

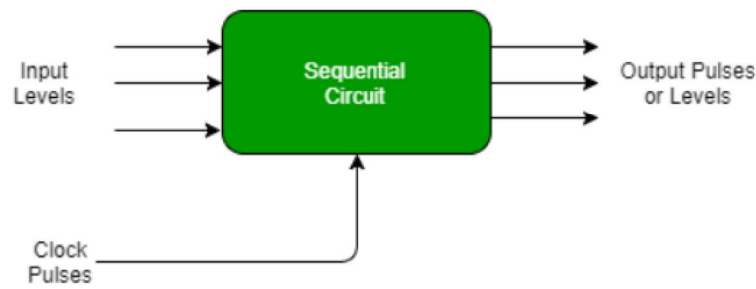


Figure: Synchronous Sequential Circuit

We use synchronous sequential circuit in synchronous counters, flip flops, and in the design of MOORE-MEALY state management machines. We use sequential circuits to design Counters, Registers, RAM, MOORE/MEALY Machine and other state retaining machines.

Clock Signal and Triggering

Clock signal is a kind of control signal that allows the elements of synchronous circuits to be in phase or phenomena that occur in circuits. It is derived from the square wave that has a high and a low level, it helps in measuring the sequential changes in the circuit states. The clock signal also makes a pulse simultaneously on all the circuit parts that are needed for the proper work of synchronous sequential circuits.

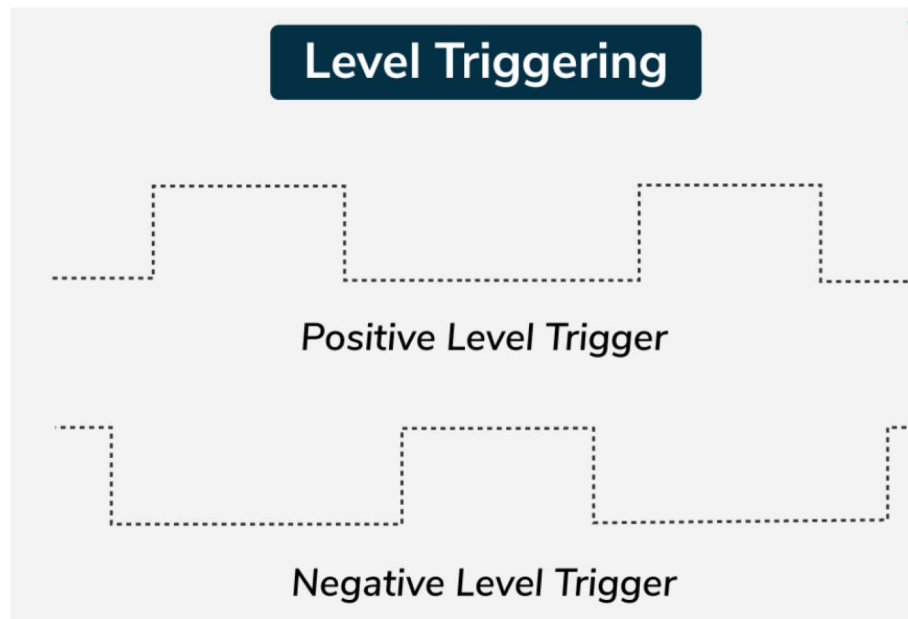
Types of Triggering

In Sequential circuits, triggering denotes the way, in terms of which the state changes take place. There are two main types of triggering

Level Triggering

Level triggering happens when the change of state is from the level of the clock signal is high or low. The circuit depends on the level of the clock signal rather than the rising or the falling edge of it. There are two types of level triggering:

1. **Positive Level Triggering:** The circuit changes state when it is high time in the clock cycle i.e. when the clock signal is high.
2. **Negative Level Triggering:** The circuit changes state when the clock signal is in the low state.



Applications

Sequential circuits find application in virtually almost every digital system today because of their capacity to handle state information. Some common applications include:

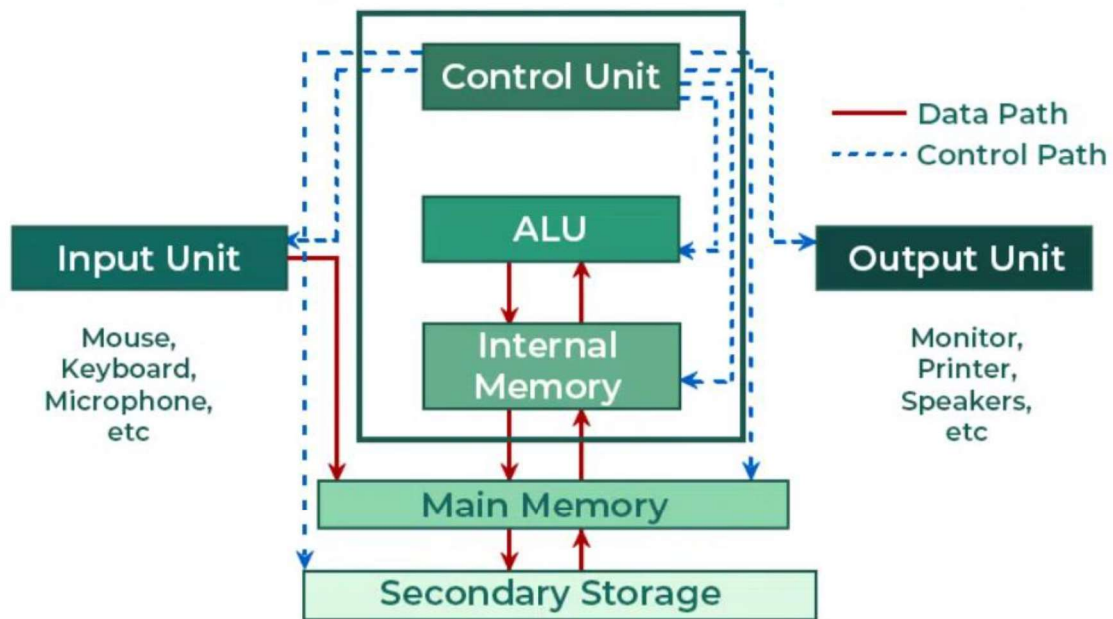
- Counters: Appearing in commonly in digital clocks, frequency counters, and event counters.
- Registers: Found in microprocessors and digital systems as a storage medium, a transfer medium and a medium for manipulating data.
- Memory Elements: Used in [RAM](#) and other storage devices to keep data in a temporary hold.
- State Machines: Made use in [control systems](#) , communication processes, and different digital devices for state control.
- Timers: It is applied in time measurement, delay production, and scheduling functions in digital circuits.

Basic Structure of Computers

Functional units:

The functional components of a digital computer include the Input Unit, which takes in data; the CPU, which processes data with its Control Unit (CU), Arithmetic Logic Unit (ALU), and Registers; the Memory Unit, which stores data temporarily (RAM) or permanently (HDD/SSD); the Output Unit, which displays results; and the Bus System, which connects and transfers data between components. These parts work together to execute tasks and provide results.

Components of computer



Functional Components of Computer

The **functional components of a computer** are the key parts that work together to process and manage data. These include the **Input Unit** for receiving data, the **CPU** for processing it, the **Memory Unit** for storing information, the **Output Unit** for displaying results, and the **Bus System** that connects all parts. These components help the computer perform tasks efficiently.

1. Input Unit

- **Purpose:** Captures data and instructions from users or external sources.
- **Function:** Converts user input into binary signals that the computer can process.
- **Common Devices (2025):**
 - Keyboard, Mouse, Touchscreens
 - Scanners, Sensors, Stylus pens
 - Voice Assistants (e.g., Siri, Alexa)
 - Biometric devices (face/fingerprint recognition)
 - Iot-based inputs from smart devices

2. Central Processing Unit (CPU) – The Brain of the Computer

The **CPU** executes instructions and controls all internal operations. In 2025, CPUs will often have multiple **cores** and **threads** to handle parallel processing efficiently.

Components of CPU:

a. Arithmetic Logic Unit (ALU)

- Performs arithmetic operations (add, subtract, multiply, divide).
- Handles logical operations (comparison, decision-making).
- Supports AI/ML tasks using built-in vector/matrix operations (in modern CPUs).

b. Control Unit (CU)

- Directs the operations of all computer parts.
- Decodes instructions and coordinates data flow.
- Sends control signals to memory and I/O devices.

c. Registers

- **High-speed memory locations** within the CPU.
- Temporarily store instructions, addresses, and intermediate data.
- Examples: Accumulator, Instruction Register, Program Counter, Address Register.
- **Modern CPUs** include 64-bit or even 128-bit registers for faster processing.

3. Memory / Storage Unit

The **memory unit** holds data and instructions before, during, and after processing.

a. Primary Memory (Main Memory):

- **RAM (Random Access Memory):** Temporarily stores data during execution.
 - Types in 2025: DDR5, LPDDR5X, and emerging **MRAM**.
- **ROM (Read-Only Memory):** Stores boot-up instructions and firmware.
- **Cache Memory:** Ultra-fast memory between CPU and RAM (L1, L2, L3 levels).

b. Secondary Storage:

- Used for long-term data storage.
- **Examples:** SSDs (NVMe drives), HDDs, flash drives, and cloud storage.
- **Modern Trend:** Use of **Cloud Integration** and **hybrid storage models**.

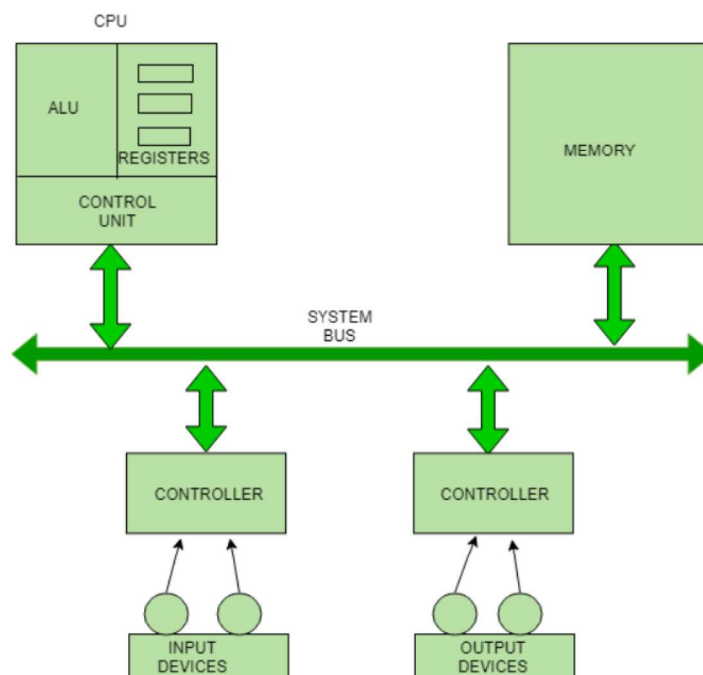
4. Output Unit

- **Purpose:** Converts processed data (binary) into a form users can understand.
- **Examples:**
 - Visual: Monitors (LED, OLED, 4K/8K displays)
 - Print: Printers (Inkjet, Laser, 3D Printers)
 - Audio: Speakers, Headphones
 - Haptic: Vibration feedback devices
- **Emerging Tech:** AR/VR headsets, voice-based output, Braille displays for accessibility

Interconnection between Functional Components

A computer consists of an input unit that takes input, a CPU that processes the input and an output unit that produces output. All these devices communicate with each other through a common bus. A bus is a transmission path, made of a set of conducting wires over which data or information in the form of electric signals is passed from one component to another in a computer. The bus can be of three types – **Address bus, Data bus and Control Bus.**

The following figure shows the connection of various functional components:



The address bus carries the address location of the data or instruction. The data bus carries data from one component to another, and the control bus carries the control signals. The system bus is the common communication path that carries signals to/from the CPU, main memory and input/output devices. The input/output devices communicate with the system bus through the controller circuit, which helps in managing various input/output devices attached to the computer.

Basic operational concepts:

A computer is an electronic device that can receive, store, process, and output data. In simple terms, it's a machine that helps us perform all sorts of tasks, from basic calculations to complex activities like playing games, watching videos, or even running artificial intelligence programs.

Computers have two main parts: Hardware and Software

- Hardware refers to the physical parts of the computer, like the central processing unit (CPU), memory, storage devices, input/output devices (like keyboards and screens), and other peripherals (extra devices connected to the computer).
- Software refers to the programs and applications that tell the computer what to do. This includes things like the operating system (such as Windows or macOS) and the apps you use for schoolwork, games, and more.

Data and Information

Data and information are related concepts, but they have different meanings. Data refers to raw facts and figures that are unorganized and have no meaning on their own. Information, on the other hand, is data that has been processed, organized, and given context to make it meaningful and useful.

- Data can be numbers, words, images, or sounds. For example, a list of sales figures is just data. On its own, it doesn't tell us much. To make it useful, we need to process and analyze it.
- Information is what we get after processing data. It's organized in a way that helps us understand something or answer a question. For example, turning the sales numbers into a graph or chart shows trends over time and gives us useful information about the company's performance.

In simple terms, data is just the raw facts, while information is what you get after processing and organizing the data to make it meaningful and helpful for making decisions.

Classification of Computers

Computers can be classified based on various factors, including their size, functionality, purpose, and processing capabilities. Here's a breakdown of the classification of computers:

- Based on Size and Power
- Based on Purpose
- Based on Data Handling
- Based on Architecture
- Based on Functionality
- Based on Usage

Please refer [Classification of Computers](#) for more details.

Peripheral Devices

Peripheral devices are external hardware components that are connected to a computer to expand its functionality. These devices can be categorized into input devices, output devices, and storage devices.

Here's an overview of the main peripheral devices:

1. Input Devices

Input devices allow users to provide data and commands to the computer.

- Keyboard: Used to type text and enter commands into the computer.
- Mouse: A pointing device used to move the cursor and interact with objects on the screen.
- Scanner: Converts physical documents and images into digital form.
- Microphone: Captures audio and converts it into digital data for the computer to process.
- Digital Camera/Webcam: Captures video or images, often for video conferencing or photography.
- Touchpad: A touch-sensitive surface used on laptops to move the cursor and interact with the computer.

2. Output Devices

Output devices display or produce the results of a computer's processing.

- Monitor: A screen that displays visual output from the computer.

- **Printer:** A device that produces a physical copy (paper printout) of documents, images, or graphics.
- **Speakers:** Convert digital audio data into sound so users can hear music, videos, or other audio output.
- **Headphones:** Provide personal audio output by converting digital sound signals into audible sound.
- **Projector:** Projects computer content (images, videos) onto a larger screen or surface.

3. Storage Devices

Storage devices are used to store data for the computer.

- **Hard Disk Drive (HDD):** A traditional storage device that stores large amounts of data on rotating disks.
- **Solid-State Drive (SSD):** A faster and more reliable storage device that uses flash memory.
- **USB Flash Drive:** A portable storage device that connects via a USB port to transfer and store data.
- **Optical Disc Drives:** Devices like CD/DVD/Blu-ray drives that read and write data on optical discs.
- **Memory Card:** Small storage devices, like SD cards or microSD cards, used in cameras, smartphones, and other portable devices.

4. Communication Devices

These devices help a computer connect with other devices or networks.

- **Modem:** A device that allows a computer to connect to the internet over a phone or cable line.
- **Router:** A networking device that connects multiple devices to the internet or a local network.
- **Bluetooth Adapter:** A device that allows a computer to communicate wirelessly with Bluetooth-enabled devices.

Types of Software

Here are the main **types of software**

1. System Software

- **Operating Systems (OS):** Manages computer hardware and software resources (e.g., Windows, macOS, Linux).
- **Utility Software:** Helps maintain and optimize the system (e.g., antivirus software, disk management tools).
- **Device Drivers:** Enable hardware components to work with the operating system (e.g., printer drivers, graphics card drivers).

2. Application Software

- **Productivity Software:** Used for creating documents, presentations, and managing data (e.g., Microsoft Office, Google Docs).
- **Multimedia Software:** Used for creating, editing, and playing audio, video, and images (e.g., Adobe Photoshop, VLC Media Player).
- **Web Browsers:** Software used to access and navigate the internet (e.g., Google Chrome, Mozilla Firefox).
- **Games:** Entertainment software designed for gaming (e.g., Fortnite, Minecraft).
- **Educational Software:** Used for learning or teaching purposes (e.g., Khan Academy, Duolingo).

3. Development Software

- **Programming Languages:** Tools used for writing software programs (e.g., Python, Java, C++).
- **Integrated Development Environments (IDEs):** Provide a platform for developers to write and test code (e.g., Visual Studio, Eclipse).
- **Database Management Software (DBMS):** Manages data storage and retrieval (e.g., MySQL, Oracle).

4. Firmware

- **Embedded Software:** Special software stored in hardware devices to control them (e.g., software in washing machines, cameras, or smart thermostats).

Basic Computer Operation

1. **Booting up:** This is the process of starting up the computer by loading the operating system (OS) into memory. The computer runs a series of checks to ensure all hardware is working correctly before loading the OS.
2. **Logging in:** After booting up, you will need to log in to access your user account. This requires entering your username and password.
3. **Running programs:** Once you have logged in, you can run programs on your computer. Programs can include web browsers, office applications, media players, and more.
4. **Accessing data:** You can access data stored on your computer or other connected devices, such as external hard drives or cloud storage services. This can include files, documents, photos, and other media.
5. **Connecting to the internet:** You can connect to the internet to access websites, download files, and communicate with others online. This typically involves using a web browser to access websites and other online services.
6. **Communicating with other devices:** You can communicate with other devices connected to your computer, such as printers, scanners, or other peripherals. This allows you to print documents, scan images, and perform other tasks.
7. **Saving and backing up data:** It's important to save and back up your data regularly to avoid data loss. You can save data to your local hard drive, external hard drives, or cloud storage services.
8. **Shutting down:** When you're finished using your computer, you should shut it down properly. This involves closing all programs and files, saving any changes, and shutting down the OS.
9. **Troubleshooting:** If you experience issues with your computer, you may need to troubleshoot the problem. This can involve diagnosing hardware or software issues, performing updates, or reinstalling drivers.
10. **Security:** It's important to keep your computer secure by using antivirus software, firewalls, and other security measures. This helps protect your data and prevent unauthorized access to your system.

Issues of Basic Computers and Computer Operations

some common issues that can arise with basic computers and computer operations:

1. **Slow performance:** Computers can become slow and unresponsive due to a variety of factors, such as lack of storage space, outdated hardware, or malware.
2. **Malware and viruses:** Malware and viruses can infect computers and cause a range of problems, including data loss, system crashes, and identity theft.
3. **Hardware failures:** Computer hardware components can fail over time, leading to issues like system crashes, data loss, and display problems.
4. **Software glitches:** Software applications can sometimes malfunction or crash, leading to issues like data loss or unresponsive programs.
5. **Driver problems:** Drivers are software components that enable hardware devices to communicate with the operating system. If drivers become outdated or corrupted, it can lead to issues like hardware failures or system crashes.
6. **Compatibility issues:** Sometimes software applications or hardware components may not be compatible with each other, leading to issues like system crashes or display problems.
7. **Internet connectivity issues:** Problems with internet connectivity can lead to issues like slow loading web pages, interrupted downloads, and difficulty accessing online services.
8. **User error:** Users can sometimes make mistakes or accidentally delete files or programs, leading to issues like data loss or unresponsive programs.

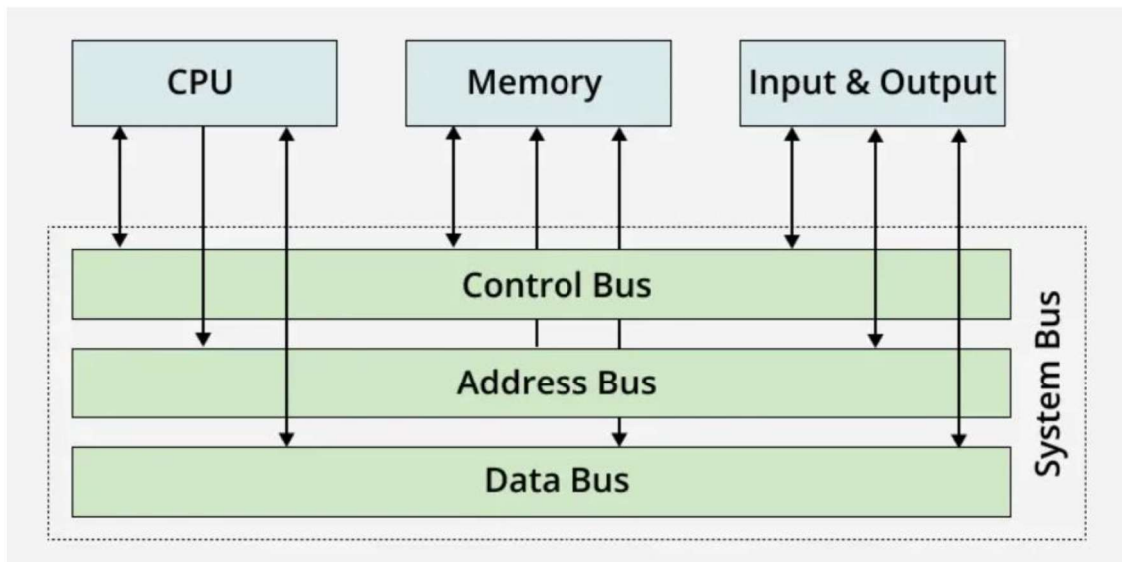
Bus structures:

A computer bus is a communication system used to transfer data between components within a computer or between different computers. It plays an important role in minimizing the number of connections needed by centralizing communication over shared pathways.

- It consists of physical connections like wires, circuits, or cables.
- Components like the CPU, memory, and input/output (I/O) devices are connected through a bus.
- It simplifies data transfer and improves efficiency.

Types Of Buses

There are three main types of buses in a computer system, which are discussed below:



1. Address Bus

A collection of wires used to identify particular location in main memory is called [Address Bus](#). Or in other words, the information used to describe the memory locations travels along the address bus.

- The address bus transports memory addresses which the processor wants to access in order to read or write data..
- The address bus is unidirectional.
- The size of address bus determines how many unique memory locations can be addressed.

Example:

- *A system with 4-bit address bus can address $2^4 = 16$ Bytes of memory.*
- *A system with 16-bit address bus can address $2^{16} = 64$ KB of memory*
- *A system with 20-bit address bus can address $2^{20} = 1$ MB of memory.*

2. Data Bus

A collection of wires through which data is transmitted from one part of a computer to another is called Data Bus. It can be thought of as a highway on which data travels within a computer.

- The main objective of data bus is transfer of the data between microprocessor to input/output devices or memory.
- The data bus transfers instructions coming from or going to the processor.
- The data bus is bidirectional because the data can flow in either direction from CPU to memory(or input/output device) or from memory to the CPU.

- The size (width) of bus determines how much data can be transmitted at one time.

Example:

- *A 16-bit bus can transmit 16 bits of data at a time.*
- *32-bit bus can transmit 32 bits at a time.*

3. Control Bus

The connections that carry control information between the CPU and other devices within the computer is called Control Bus. The control bus transports orders and synchronization signal coming from the control unit and travelling to all other hardware components

- The main objective of control bus is all signals controller carried from processor to other hardware device.
- The Control bus is bidirectional because the data can flow in either direction from CPU to memory(or input/output device) or from memory to the CPU.
- It also transmits response signals from the hardware.

Performance:

In computer organization, performance refers to the speed and efficiency at which a computer system can execute tasks and process data. A high-performing computer system is one that can perform tasks quickly and efficiently while minimizing the amount of time and resources required to complete these tasks.

Here are several factors that can impact the performance of a computer system, including:

- **Processor speed:** The speed of the processor, measured in GHz (gigahertz), determines how quickly the computer can execute instructions and process data.
- **Memory:** The amount and speed of the memory, including RAM (random access memory) and cache memory, can impact how quickly data can be accessed and processed by the computer.
- **Storage:** The speed and capacity of the storage devices, including hard drives and solid-state drives (SSDs), can impact the speed at which data can be stored and retrieved.
- **I/O devices:** The speed and efficiency of input/output devices, such as [keyboards](#) , mice, and displays, can impact the overall performance of the system.

- **Software optimization:** The efficiency of the software running on the system, including operating systems and applications, can impact how quickly tasks can be completed.

Improving the performance of a computer system typically involves optimizing one or more of these factors to reduce the time and resources required to complete tasks. This can involve upgrading hardware components, optimizing software, and using specialized performance-tuning tools to identify and address bottlenecks in the system.

Computer performance is the amount of work accomplished by a computer system. The word performance in computer performance means "How well is the computer doing the work it is supposed to do?". It basically depends on the response time, throughput, and execution time of a computer system. **Response time** is the time from the start to completion of a task. This also includes:

- Operating system overhead.
- Waiting for I/O and other processes
- Accessing disk and memory
- Time spent executing on the CPU or execution time.

Throughput is the total amount of work done in a given time. **CPU execution time** is the total time a CPU spends computing on a given task. It also excludes time for I/O or running other programs. This is also referred to as simply CPU time. Performance is determined by execution time as performance is inversely proportional to execution time.

Performance = (1 / Execution time)

And,

(Performance of A / Performance of B)

= (Execution Time of B / Execution Time of A)

If given that Processor A is faster than processor B, that means execution time of A is less than that of execution time of B. Therefore, performance of A is greater than that of performance of B. **Example** - Machine A runs a program in 100 seconds, Machine B runs the same program in 125 seconds

(Performance of A / Performance of B)

= (Execution Time of B / Execution Time of A)

= 125 / 100 = 1.25

That means machine A is 1.25 times faster than Machine B. And, the time to execute a given program can be computed as:

Execution time = CPU clock cycles x clock cycle time

Since clock cycle time and clock rate are reciprocals, so,

Execution time = CPU clock cycles / clock rate

The number of CPU clock cycles can be determined by,

CPU clock cycles

= (No. of instructions / Program) x (Clock cycles / Instruction)

= Instruction Count x CPI

Which gives,

Execution time

= Instruction Count x CPI x clock cycle time

= Instruction Count x CPI / clock rate

Units for CPU Execution Time

CPU time	=	Seconds	=	Instructions	x	Cycles	x	Seconds
		Program		Program		Instruction		Cycle

How to Improve Performance?

To improve performance you can either:

- Decrease the CPI (clock cycles per instruction) by using new Hardware.
- Decrease the clock time or Increase clock rate by reducing propagation delays or by use pipelining.
- Decrease the number of required cycles or improve ISA or Compiler.

Execution Upgrade Procedures

The Exhibition upgrade on computer chip execution time is worked with by the accompanying variables in a significant manner.

- 1.Internal Engineering of the computer chip
- 2.Instruction Arrangement of the central processor
- 3.Memory Speed and transmission capacity
- 4.Percentage utilization of the registers in execution (note: Registers are something like multiple

times quicker than memory).

Further, the accompanying highlights of a framework likewise improve the general presentation:

Compositional Augmentations (Register set/GPRs/Register Document)

- 1.Special guidelines and tending to modes
- 2.Status register contents
- 3.Program control stack
- 4.Pipelining
- 5.Multiple degrees of Store Memory
- 6.Use of co-processors or particular equipment for Drifting Point tasks, Vector handling, Media handling.
- 7.Virtual Memory and Memory the executives Unit execution.
- 8.System Transport execution.
- 9.Super Scalar Handling

Uses and Benefitsof Performance of Computer

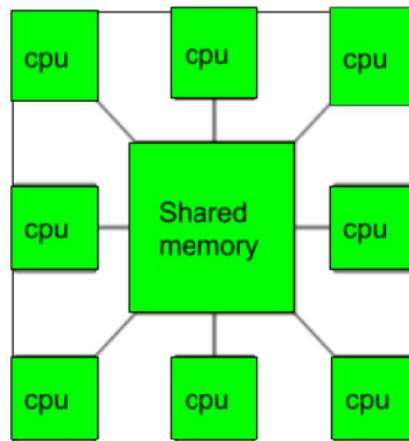
Some of the key uses and benefits of a high-performing computer system include:

- **Increased productivity:** A high-performing computer can help increase productivity by reducing the time required to complete tasks, allowing users to complete more work in less time.
- **Improved user experience:** A fast and efficient computer system can provide a better user experience, with smoother operation and fewer delays or interruptions.
- **Faster data processing:** A high-performing computer can process data more quickly, enabling faster access to critical information and insights.
- **Enhanced gaming and multimedia performance:** High-performance computers are better suited for gaming and multimedia applications, providing smoother and more immersive experiences.
- **Better efficiency and cost savings:** By optimizing the performance of a computer system, it is possible to reduce the time and resources required to complete tasks, leading to better efficiency and cost savings.

Multiprocessors and Multi computers:

1. Multiprocessor: A Multiprocessor is a computer system with two or more central processing units (CPUs) share full access to a common RAM. The main objective of using a multiprocessor is to boost

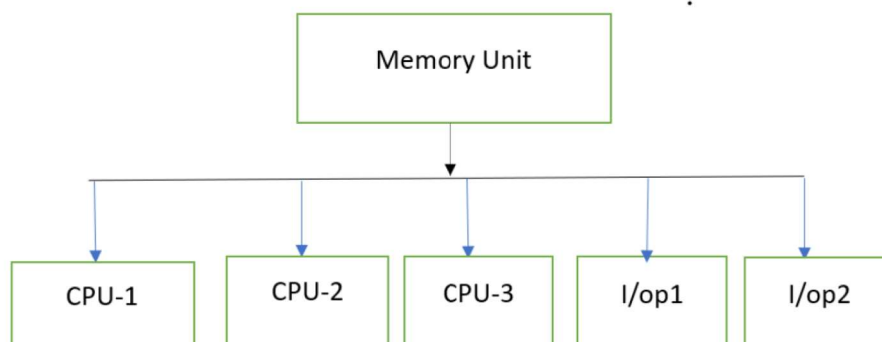
the system's execution speed, with other objectives being fault tolerance and application matching. There are two types of multiprocessors, one is called shared memory multiprocessor and another is distributed memory multiprocessor. In shared memory multiprocessors, all the CPUs share the common memory but in a distributed memory multiprocessor, every CPU has its own private memory.



The interconnection among two or more processor and shared memory is done with three methods

- 1) Time shared common bus
- 2) Multiport memories
- 3) Crossbar switch network

1) Time shared common bus



Illustrate time shared common bus

As the name itself indicates, in this method, there contains a single shared bus through which all processor & memory unit can be communicated.

Consider CPU-1 is interacting with memory unit using common shared bus; in that case, all other processors must be idle as we have only one bus to communicate.

Advantage:

- Simple to implement.
- Due to single common bus cost to implement is very less.

Disadvantage:

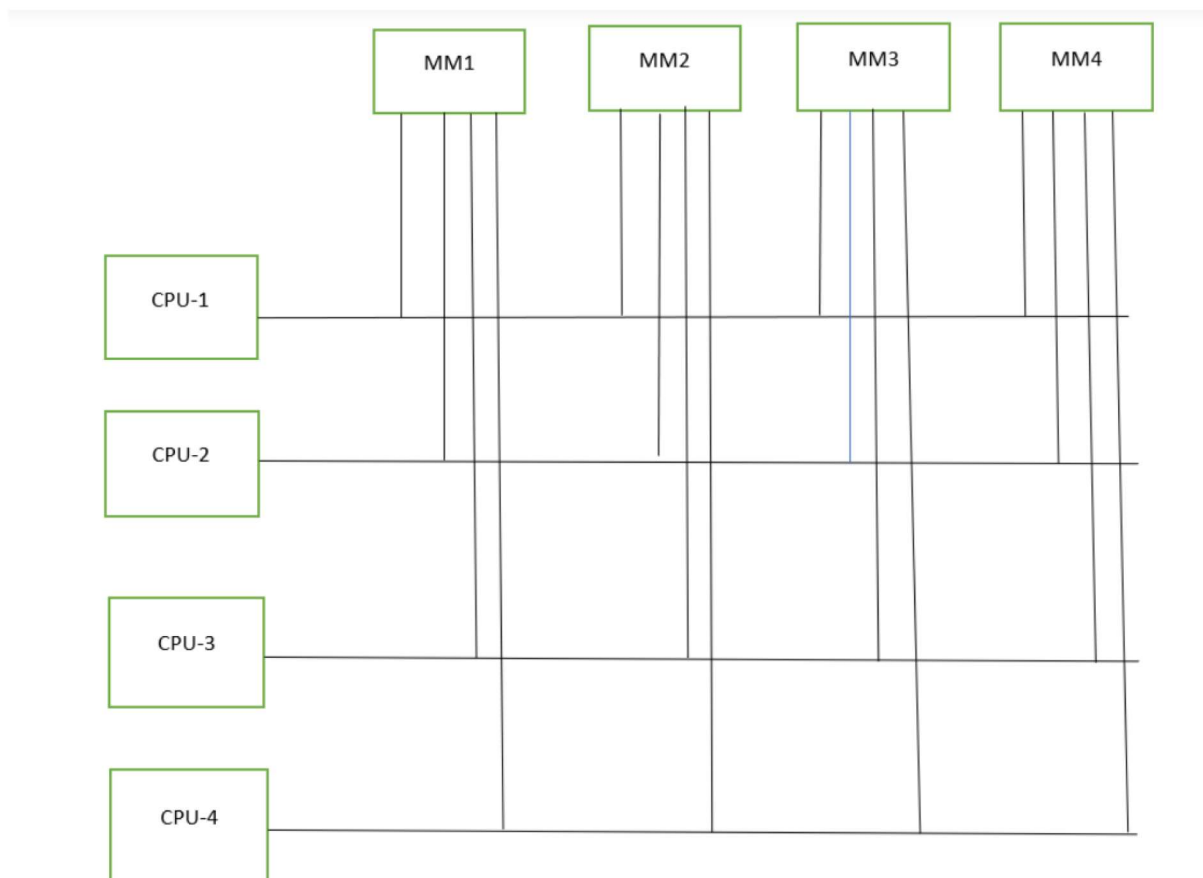
- **Data transfer rate is slow.**

2)Multiport memories

Unlike in the shared common bus method, hence it contains separate bus for each processor to communicate with the memory module.

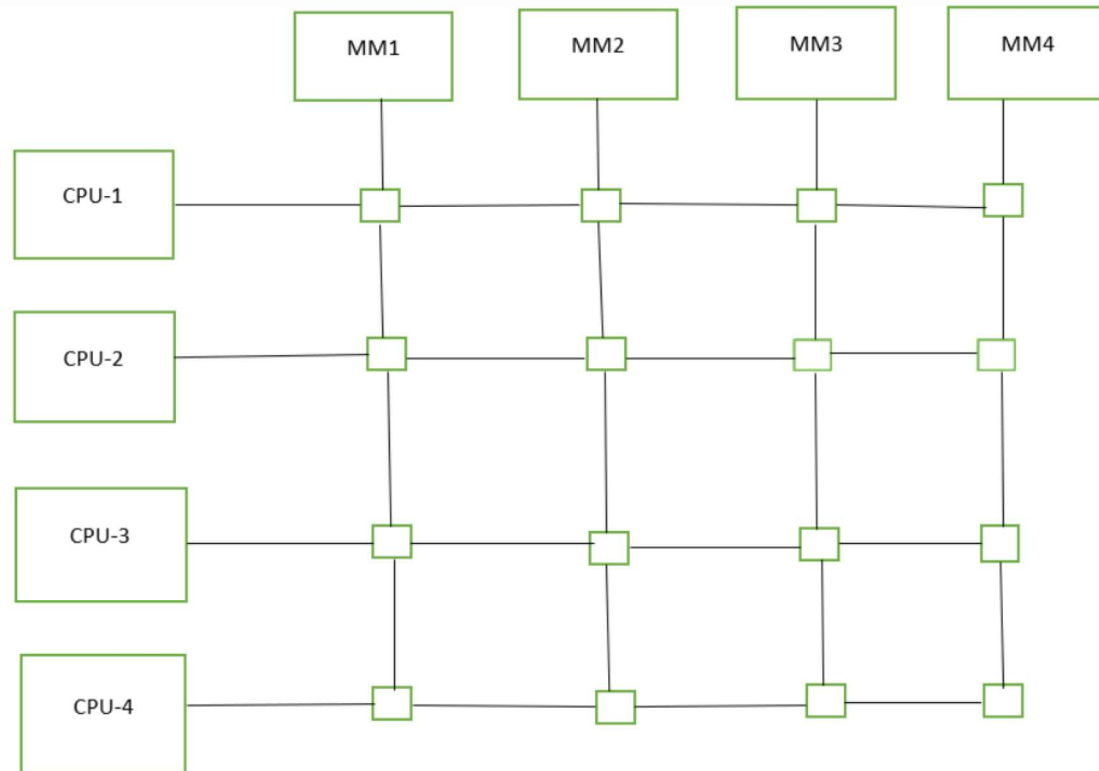
Suppose CPU-1 wants to interact with memory module 1 then port mm1 is enabled. Similarly CPU-4 wants to

to interact with memory module 4 then port mm4 is enabled. Hence all the process can be communicated parallelly. If more than one CPU request for same time memory module, priority will be given in the order of CPU-1,CPU-2,CPU-3,CPU-4.



3)Crossbar switch network

Here instead multiport unlike in multiport memories, a switch will be installed between memory unit and CPU. Switch is responsible for whether to pass the request to a particular memory module or not based on the request made for.



Illustrate cross-bar switch network

Advantage:

- High data through rate.

Disadvantage:

- **Complex to implement as more switches involved.**
- Costlier to implement.

Applications of Multiprocessor -

1. As a uniprocessor, such as single instruction, single data stream (SISD).
2. As a multiprocessor, such as single instruction, multiple data stream (SIMD), which is usually used for vector processing.
3. Multiple series of instructions in a single perspective, such as multiple instruction, single data stream (MISD), which is used for describing hyper-threading or pipelined processors.

4. Inside a single system for executing multiple, individual series of instructions in multiple perspectives, such as multiple instruction, multiple data stream (MIMD).

Benefits of using a Multiprocessor -

- Enhanced performance.
- Multiple applications.
- Multi-tasking inside an application.
- High throughput and responsiveness.
- Hardware sharing among CPUs.

Advantages:

Improved performance: Multiprocessor systems can execute tasks faster than single-processor systems, as the workload can be distributed across multiple processors.

Better scalability: Multiprocessor systems can be scaled more easily than single-processor systems, as additional processors can be added to the system to handle increased workloads.

Increased reliability: Multiprocessor systems can continue to operate even if one processor fails, as the remaining processors can continue to execute tasks.

Reduced cost: Multiprocessor systems can be more cost-effective than building multiple single-processor systems to handle the same workload.

Enhanced parallelism: Multiprocessor systems allow for greater parallelism, as different processors can execute different tasks simultaneously.

Disadvantages:

Increased complexity: Multiprocessor systems are more complex than single-processor systems, and they require additional hardware, software, and management resources.

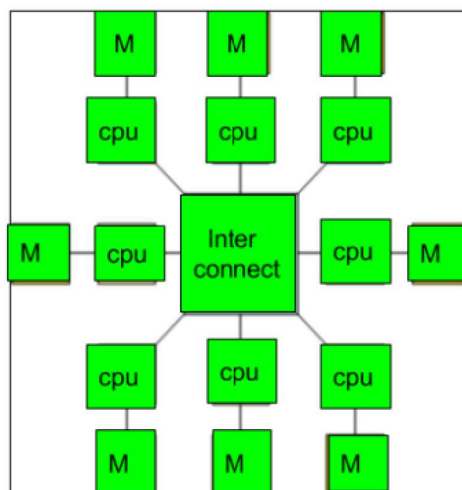
Higher power consumption: Multiprocessor systems require more power to operate than single-processor systems, which can increase the cost of operating and maintaining the system.

Difficult programming: Developing software that can effectively utilize multiple processors can be challenging, and it requires specialized programming skills.

Synchronization issues: Multiprocessor systems require synchronization between processors to ensure that tasks are executed correctly and efficiently, which can add complexity and overhead to the system.

Limited performance gains: Not all applications can benefit from multiprocessor systems, and some applications may only see limited performance gains when running on a multiprocessor system.

2. Multicomputer: A [multicomputer system](#) is a computer system with multiple processors that are connected together to solve a problem. Each processor has its own memory and it is accessible by that particular processor and those processors can communicate with each other via an interconnection network.



As the multicomputer is capable of messages passing between the processors, it is possible to divide the task between the processors to complete the task. Hence, a multicomputer can be used for distributed computing. It is cost effective and easier to build a multicomputer than a multiprocessor. **Difference between multiprocessor and Multicomputer:**

1. Multiprocessor is a system with two or more central processing units (CPUs) that is capable of performing multiple tasks where as a multicomputer is a system with multiple processors that are attached via an interconnection network to perform a computation task.
2. A multiprocessor system is a single computer that operates with multiple CPUs where as a multicomputer system is a cluster of computers that operate as a singular computer.
3. Construction of multicomputer is easier and cost effective than a multiprocessor.
4. In multiprocessor system, program tends to be easier where as in multicomputer system, program tends to be more difficult.

5. Multiprocessor supports parallel computing, Multicomputer supports distributed computing.

Advantages:

Improved performance: Multicomputer systems can execute tasks faster than single-computer systems, as the workload can be distributed across multiple computers.

Better scalability: Multicomputer systems can be scaled more easily than single-computer systems, as additional computers can be added to the system to handle increased workloads.

Increased reliability: Multicomputer systems can continue to operate even if one computer fails, as the remaining computers can continue to execute tasks.

Reduced cost: Multicomputer systems can be more cost-effective than building a single large computer system to handle the same workload.

Enhanced parallelism: Multicomputer systems allow for greater parallelism, as different computers can execute different tasks simultaneously.

Disadvantages:

Increased complexity: Multicomputer systems are more complex than single-computer systems, and they require additional hardware, software, and management resources.

Higher power consumption: Multicomputer systems require more power to operate than single-computer systems, which can increase the cost of operating and maintaining the system.

Difficult programming: Developing software that can effectively utilize multiple computers can be challenging, and it requires specialized programming skills.

Synchronization issues: Multicomputer systems require synchronization between computers to ensure that tasks are executed correctly and efficiently, which can add complexity and overhead to the system.

Network latency: Multicomputer systems rely on a network to communicate between computers, and network latency can impact system performance.

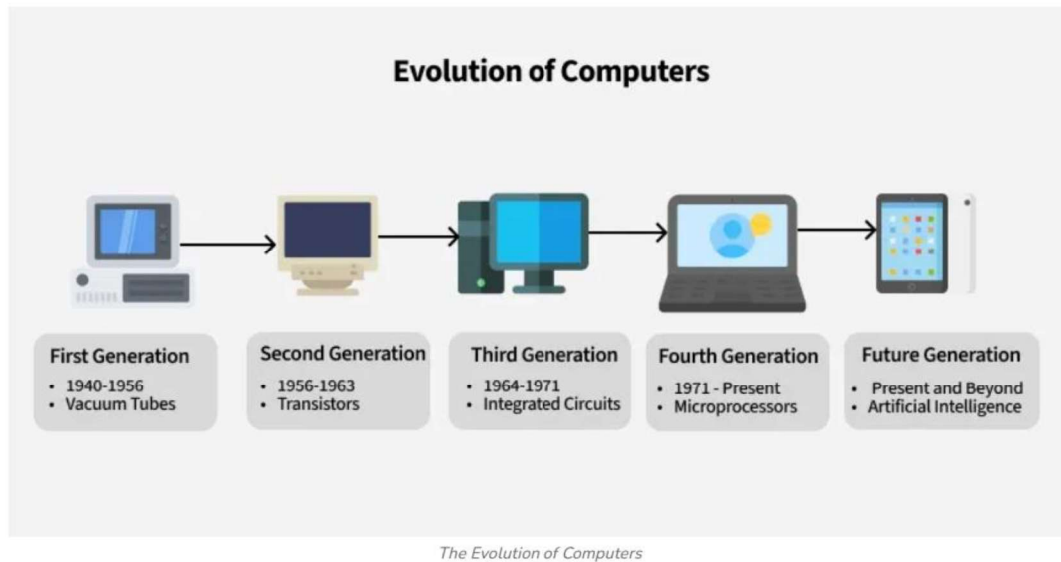
Historical Perspective:

Before the invention of computers, people relied on simple tools like sticks, stones, and bones to keep track of numbers and perform basic calculations. As technology progressed and human understanding grew, more advanced devices were developed, such as the abacus and Napier's Bones. While these early tools served as basic computational devices, they were limited in their ability to handle complex calculations.

Below, we take a look at some of the most significant computing devices throughout history, tracing their evolution from the earliest forms to the most advanced technologies that followed.

The Evolution of Computers

The history of computers spans thousands of years, from early counting devices to the powerful systems we use today. Here's an overview of the key milestones in the evolution of computers:



1. Early Counting Devices (Pre-Computer Era)

The Abacus (c. 4000 BCE)

The abacus, created by the Chinese, is often regarded as the first computing device. It consisted of beads strung on rods and was used to perform simple arithmetic operations like addition and subtraction. Over time, different versions of the abacus spread across Asia, becoming an essential tool for calculations.

Napier's Bones (1617)

Invented by John Napier, **Napier's Bones** were a set of ivory rods engraved with numbers, designed to assist with multiplication and division. This invention also introduced the concept of the decimal point, a crucial development in simplifying calculations.

2. Mechanical Calculators (17th-19th Century)

Pascaline (1642-1644)

French mathematician **Blaise Pascal** developed the **Pascaline**, the first mechanical calculator capable of performing addition and subtraction. It used gears and wheels to calculate, and its purpose was to help Pascal's father, a tax collector, with his work.

Stepped Reckoner (1673)

German philosopher and mathematician **Gottfried Wilhelm Leibniz** improved Pascal's design, developing the **Stepped Reckoner**. It was capable of performing addition, subtraction, multiplication, and division, and it used fluted drums instead of gears.

Difference Engine (1820s)

Charles Babbage, often called the "Father of Modern Computing," designed the **Difference Engine**, a mechanical device meant to calculate polynomial functions. Though it was never fully built during his lifetime, it demonstrated the potential for automatic computation.

Analytical Engine (1830s)

Babbage also developed the **Analytical Engine**, a more advanced version of the Difference Engine. It was the first design for a general-purpose mechanical computer. It included a control unit, memory, and an input/output system using punch cards. Although it was never constructed, its principles anticipated modern computers.

3. The Rise of Electronic Computing (1930s-1940s)

Tabulating Machine (1890)

Herman Hollerith, an American statistician invented this machine in the year 1890. Tabulating Machine was a mechanical tabulator that was based on punch cards. It was capable of tabulating statistics and record or sort data or information. This machine was used by U.S. Census in the year 1890. Hollerith's Tabulating Machine Company was started by Hollerith and this company later became International Business Machine (IBM) in the year 1924.

Differential Analyzer (1930s)

Differential Analyzer was the first electronic computer introduced in the year 1930 in the United States. It was basically an analog device that was invented by Vannevar Bush. This machine consists of vacuum tubes to switch electrical signals to perform calculations. It was capable of doing 25 calculations in a few minutes.

Mark I

In the year 1937, major changes began in the history of computers when Howard Aiken planned to develop a machine that could perform large calculations or calculations involving large numbers. In the year 1944, Mark I computer was built as a partnership between IBM and Harvard. It was also the first programmable digital computer marking a new era in the computer world.

4. The Era of Transistors (1950s-1960s)

Transistor Computers (1950s)

In the 1950s, the invention of the **transistor** revolutionized computing. Transistors were smaller, more reliable, and energy-efficient compared to vacuum tubes. They played a key role in making computers more compact and affordable.

UNIVAC I (1951)

The **Universal Automatic Computer I (UNIVAC I)**, developed by **Eckert and Mauchly**, was the first commercially successful computer. It was used for scientific and business applications and demonstrated the potential of electronic computing.

5. The Rise of Integrated Circuits (1960s-1970s)

Integrated Circuits (1960s)

The introduction of **Integrated Circuits (ICs)** allowed multiple transistors to be placed on a single chip, which dramatically reduced the size and cost of computers while improving their performance.

IBM System/360 (1964)

The **IBM System/360** was a family of mainframe computers that utilized integrated circuits, setting a new standard for computing in business, government, and academia. It became one of the first systems to offer compatibility across different machines.

Minicomputers and Microcomputers

With the development of the **microprocessor**, the size of computers shrank even further, leading to the creation of affordable **minicomputers** like the **PDP-8** and **PDP-11**. These smaller systems paved the way for the personal computer revolution.

6. The Personal Computer Revolution (1970s-1980s)

Apple II (1977)

The **Apple II**, developed by **Steve Jobs** and **Steve Wozniak**, was one of the first successful personal computers. It used a microprocessor and could run basic software applications like word processors and games.

IBM PC (1981)

The introduction of the **IBM PC** in 1981 standardized the personal computer market, offering a system that could be easily upgraded and compatible with a wide variety of software. It played a major role in the spread of personal computing.

The Macintosh (1984)

Apple's **Macintosh** introduced the concept of the **graphical user interface (GUI)**, making computers more user-friendly and accessible to a broader audience.

7. The Internet and Networking (1990s-Present)

The World Wide Web (1990s)

The invention of the **World Wide Web** by **Tim Berners-Lee** revolutionized the way people used computers. It made information accessible globally and led to the creation of web browsers like **Netscape Navigator** and **Internet Explorer**.

Cloud Computing (2000s-Present)

Cloud computing allows **have been** users to store and access data remotely via the internet, making it easier to scale computing resources. Services like **Google Drive**, **Dropbox**, and **Amazon Web Services (AWS)** transformed how businesses and individuals manage data.

8. The Modern Day and the Future of Computing

Artificial Intelligence (AI):

AI is rapidly becoming a cornerstone of modern computing. **Machine learning** and **deep learning** algorithms enable computers to make decisions, recognize patterns, and even understand human language, leading to advancements in everything from virtual assistants to autonomous vehicles.

Quantum Computing (Emerging):

Quantum computing promises to revolutionize fields like cryptography and materials science by solving problems that are beyond the reach of classical computers. Though still in its early stages, quantum computers could one day solve complex problems exponentially faster than traditional systems.

The Internet of Things (IoT):

The **Internet of Things (IoT)** is allowed **fifth-generation**, allowing them to collect and share data. From smart homes to wearable tech, IoT devices are transforming the way we interact with the world around us.

Generations of Computers

First Generation Computers

In the period of the year 1940-1956, it was referred to as the period of the first generation of computers. These machines are slow, huge, and expensive. In this generation of computers, vacuum tubes were used as the basic components of CPU and memory. Also, they were mainly dependent on the batch operating systems and punch cards. Magnetic tape and paper tape were used as output and input devices. For example **The**, etc.

Second Generation Computers

The **were** period of the year, 1957-1963 was referred to as the period of the second generation of computers. It was the time of the transistor computers. In the second generation of computers, transistors (which were cheap in cost) are used. Transistors are also compact and consume less power. Transistor computers are faster than first-generation computers. For primary memory, magnetic cores were used, and for secondary memory magnetic disc and tapes for storage purposes. In second-generation computers, COBOL and FORTRAN are used as Assembly language and programming languages, and Batch processing and multiprogramming operating systems are allowed in these computers.

For example **IBM 1620, IBM 7094, CDC 1604, CDC 3600**, etc.

Third Generation Computers

In the third generation of computers, integrated circuits (ICs) were used instead of transistors(in the second generation). A single IC consists of many transistors which increased the power of a computer and also reduced the cost. The third generation computers are more reliable, efficient, and smaller in size. It used remote processing, time-sharing, and multiprogramming as operating systems. FORTRON-II TO IV, COBOL, and PASCAL PL/1 were used which are high-level programming languages.

For example **IBM-360 series, Honeywell-6000 series, IBM-370/168**, etc.

Fourth Generation Computers

The period of 1971-1980 was mainly the time of fourth generation computers. It used VLSI(Very Large Scale Integrated) circuits. VLSI is a chip containing millions of transistors and other circuit elements and because of these chips, the computers of this generation are more compact, powerful, fast, and affordable(low in cost). Real-time, time-sharing and distributed operating system are used

by these computers. C and C++ are used as the programming languages in this generation of computers.

For example **STAR 1000, PDP 11, CRAY-1, CRAY-X-MP**, etc.

Fifth Generation Computers

From 1980 - to till date these computers have been **Internet** used. The ULSI (Ultra Large Scale Integration) technology is used in fifth-generation computers instead of the VLSI technology of fourth-generation computers. Microprocessor chips with ten million electronic components are used in these computers. Parallel processing hardware and AI (Artificial Intelligence) software are also used in fifth-generation computers. The programming languages like C, C++, Java, .Net, etc. are used.

For example **Desktop, Laptop, NoteBook, UltraBook**, etc.

UNIT-II

Addressing Methods and Machine Program Sequencing

Addressing Methods:

Addressing modes are techniques used by the CPU to identify the location of the operand(s) needed for executing an instruction. They provide rules for interpreting the address field in an instruction, helping the CPU fetch operands correctly.

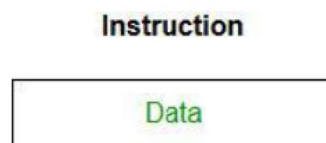
- Opcode – Tells the CPU what operation to perform (e.g., ADD, MOV).
- Operands – The data or addresses on which the operation is performed.



Addressing Modes Types

Implicit (Implied) Addressing

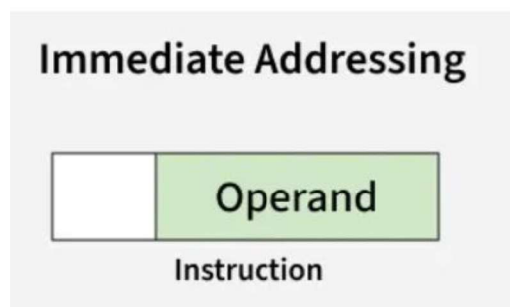
The instruction does not mention the operand directly. The CPU knows what to use from the instruction itself, usually a special register like the accumulator or the stack.



It is used for special instructions or control commands like CLA, PUSH, and RET, where the operand is automatically known from the instruction itself

Immediate Addressing

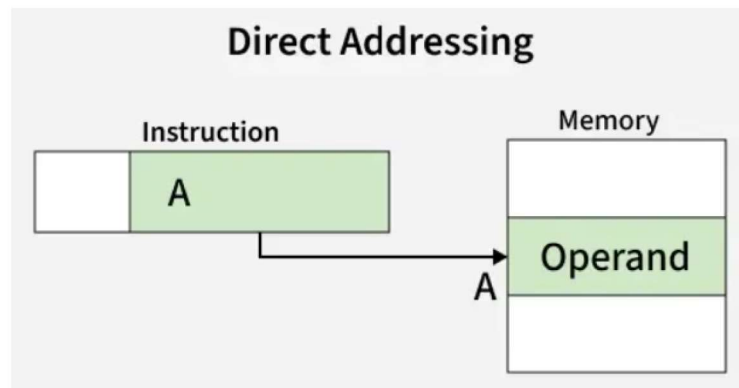
The operand is the part of the instruction itself. It is used when the value is known while writing the program.



Example: MOV R1, #5 moves the value 5 into register R1, where #5 is the immediate value.

Direct Addressing

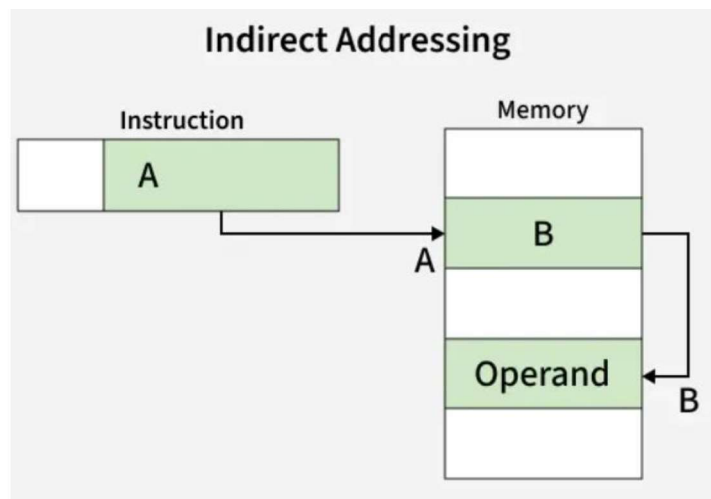
The instruction contains the memory address of the operand. The CPU accesses the data directly from that address.



Example: LOAD R1, 1000 loads data from memory address 1000 into register R1.

Indirect Addressing

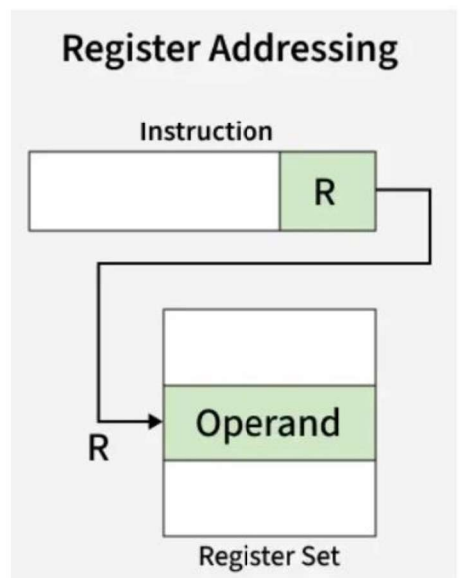
The instruction contains the address of a register or memory location that holds the actual address of the operand. The CPU first fetches this address, then accesses the operand.



Example: LOAD R1, (R2) loads data from the memory location whose address is in register R2.

Register Addressing

The operand is located in a CPU register specified by the instruction.



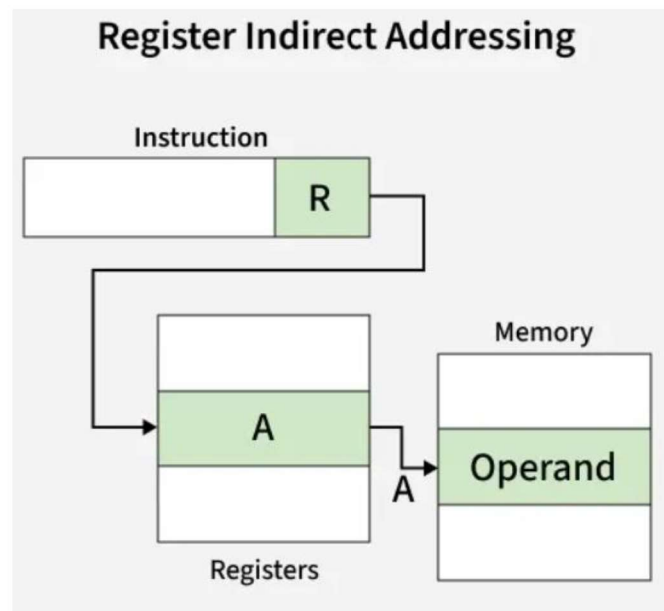
Step:

- The instruction specifies a register (R).
- The CPU takes operand directly from register R.

Example: MOV A, B operates between registers A and B.

Register Indirect Addressing

The register specified in the instruction contains the memory address of the operand.



Step:

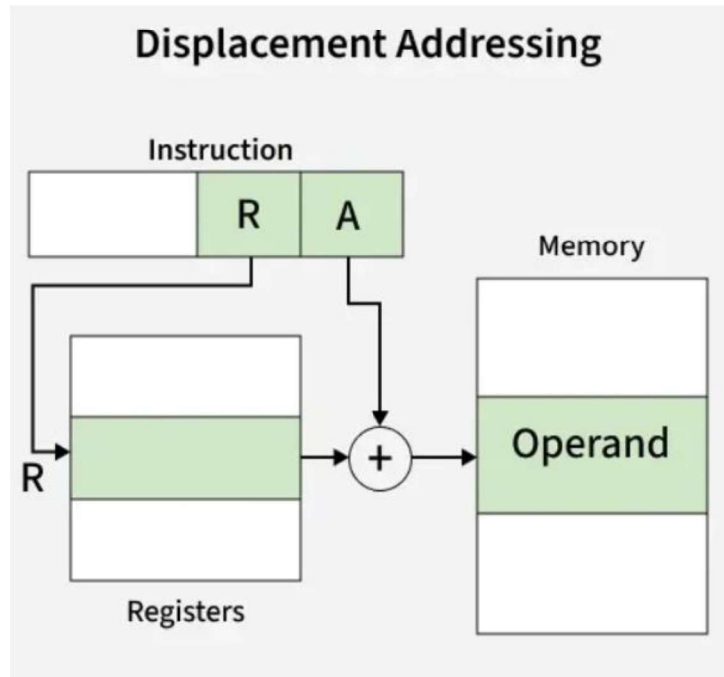
- The instruction specifies a register.
- This register holds the address (A).

- The CPU fetches the operand from memory location A.

Example: MOV A, [R1] uses content of R1 as memory address.

Displacement Addressing (Indexed, Base-Register, Relative)

The operand's effective address is calculated by adding a constant value (displacement) to the contents of one or more registers.



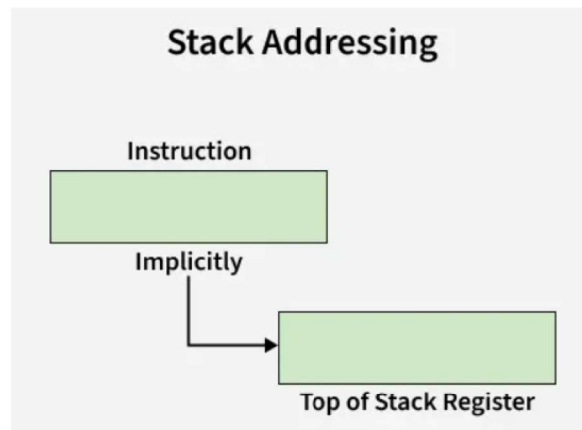
Step:

- The instruction provides a base register (R) and an address part (A).
- CPU adds the value of R and A to get the effective operand address.
- Operand is fetched from the calculated address in memory.

Example: Used for arrays, accessing an element at a position relative to a base.

Stack Addressing

The operand is implicitly taken from the top of the stack, without being mentioned in the instruction.



Step:

- Operation is performed using the value at the stack's top (implied by instruction).
- No need for explicit operand field; CPU refers to stack pointer register by default.

Example: POP and PUSH operations.

Machine Program Sequencing:

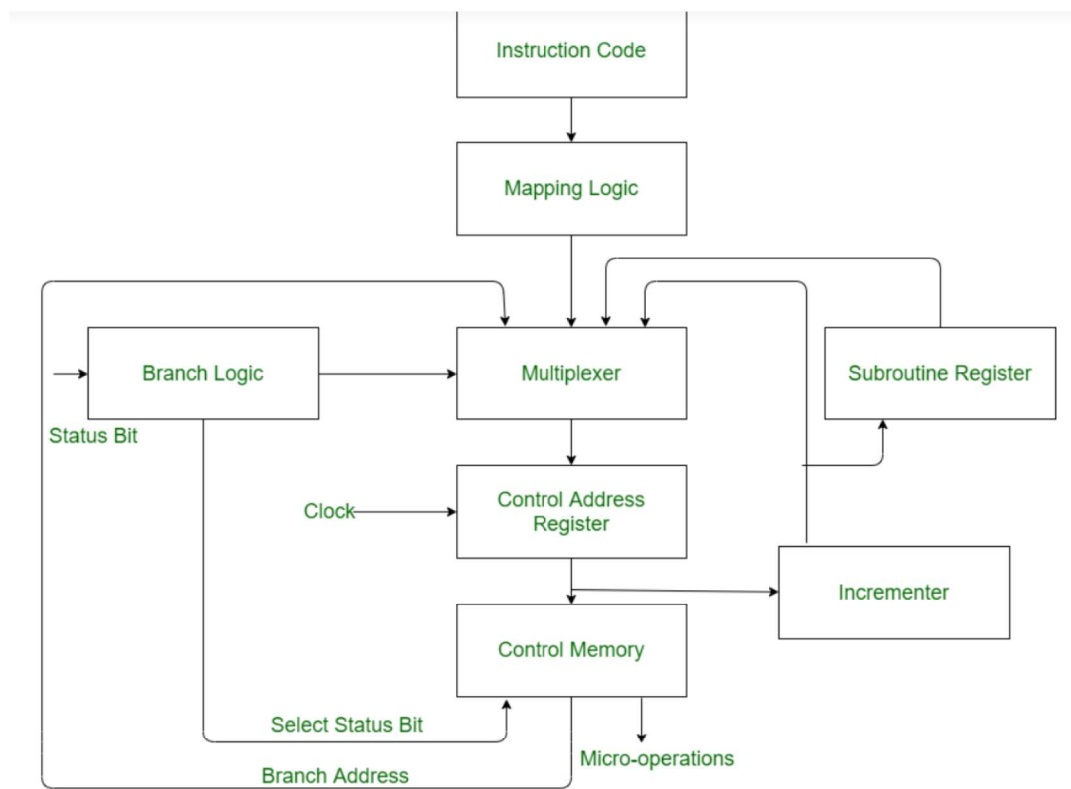
The address is used by a microprogram sequencer to decide which microinstruction has to be performed next. Microprogram sequencing is the name of the total procedure. The addresses needed to step through a control store's microprogram are created by a sequencer, also known as a microsequencer.

The control unit's main job is to communicate with the computer's memory, input/output device, and [ALU](#) on how to respond to or carry out a set of instructions. Processing is not done within the Control unit itself. It only directs and manages the task. It serves as the computer's supervisor, regulating all operations including retrieving instructions from [main memory](#) and putting them to use.

Micro Instructions Sequencer

Micro Instructions Sequencer is a combination of all hardware for selecting the next micro-instruction address. The micro-instruction in control memory contains a set of bits to initiate micro-operations in computer registers and other bits to specify the method by which the address is obtained.

Implementation of Micro Instructions Sequencer



- **Control Address Register(CAR)** : Control address register receives the address from four different paths. For receiving the addresses from four different paths, Multiplexer is used.
- **Multiplexer** : Multiplexer is a combinational circuit which contains many data inputs and single data output depending on control or select inputs.
- **Branching** : Branching is achieved by specifying the branch address in one of the fields of the micro instruction. Conditional branching is obtained by using part of the micro-instruction to select a specific status bit in order to determine its condition.
- **Mapping Logic** : An external address is transferred into control memory via a mapping logic circuit.
- **Incrementer** : Incrementer increments the content of the control address register by one, to select the next micro-instruction in sequence.
- **Subroutine Register (SBR)** : The return address for a subroutine is stored in a special register called Subroutine Register whose value is then used when the micro-program wishes to return from the subroutine.
- **Control Memory** : Control memory is a type of memory which contains addressable storage registers. Data is temporarily stored in control memory. Control memory can be accessed quicker than main memory.

Design of Micro-programmed Controller Unit

A set of micro-instructions make up microprogramming. Three micro-operation phases are primarily present in each micro-instruction. F1, F2, F3, respectively. We also have the condition, branch, and address fields in addition to these three.

A 3X8 decoder decodes the F1, F2, and F3 fields because they are all 3 bits in size. We receive 8 outputs from the F1 micro-operation phase. One of these eight outputs is still unutilized.

Like the F1 procedure, the F2 operation produces a total of 7 outputs.

Only six operations can be carried out in the F3 micro-operation phase (one is left unfinished, and the other is saved for later use).

Therefore, we can complete 20 micro-operations in total (7+7+6).

Out of 20 operations, the diagram above shows 5 micro-operations.

The ALU is attached to F2 output number 3. It applies the AND operation to the contents of the data register and accumulator, then transfers the outcome to the accumulator. This is how it is written: ACAC AND DR

And we give the matching accumulator a clock pulse. The ability to load it onto the accumulator is enabled.

Output 1 of F1 adds data register and accumulator using an ALU, then sends the outcome to the accumulator.

The DRTAR operation, or transfer of the contents of the data register to the address register, is carried out by output 5 of F1. However, the data register's size is 16 bits, while the the rightmost significant 12 bits of the data register are moved to the address register since the address register only has a capacity of 12 bits.

Transferring the contents of the programme counter to the address register is what output 6 of F1 does.

Memory locations and address:

Computer stores data, instructions in the memory. The data can be in the form of numbers or characters. The memory consists of many millions of storage cells.

Each cell is capable of storing 1-bit information having value 0 or 1.

- A single bit represents a very small amount of information. For this reason, the memory is organized so that a group of n-bits can be stored or retrieved in a single, basic operation.
- Each group of n bits is referred to as a word of information, and n is called the word length.
- Fig. 6.4.1 shows the computer memory in the form of collection of words. In most of the memories word length is 8-bit, i.e., byte.
- Looking at the figure, we can say that memory is a array of M number of n-bit words. Each word of memory has an unique address. The address is used to identify a particular word in the memory or memory location. Thus the number and address lines provided by the processor decides the addressing capability of the processor.
- For example, if the processor has 16 address lines it can address up to 2^{16} (65536) memory locations.
- Table 6.7.1 shows the relation between memory capacity and the address lines.

Memory capacity	Address lines required
1K = 1024 memory locations	10
2K = 2048 memory locations	11
4K = 4096 memory locations	12
8K = 8192 memory locations	13
16K = 16384 memory locations	14
32K = 32768 memory locations	15
64K = 65536 memory locations	16

Byte Addressability

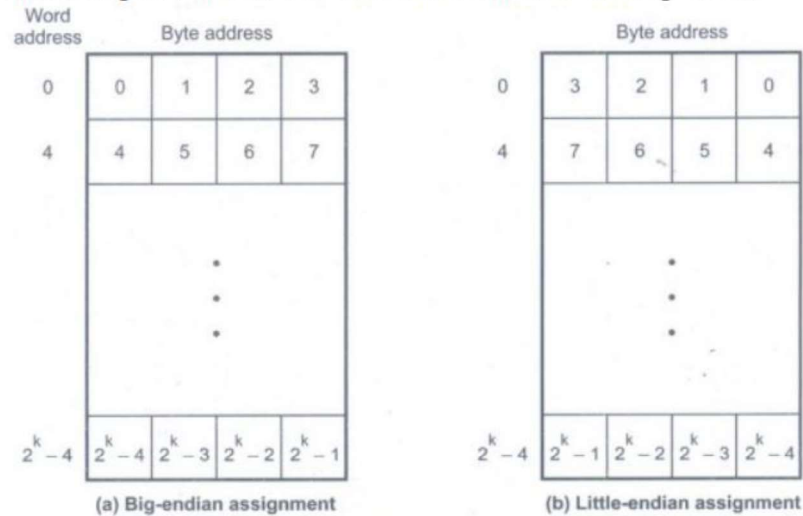
- As mentioned earlier, the word length of 8-bit is known as byte. However the word length of memory ranges from 16 to 64 bits are simply called word.
- Each byte location of the word has a unique address. That is each successive byte location in the memory is addressed with byte 0, byte 1, byte 2 and so on. Thus if the word length of the processor is 32 bits, successive words are located at addresses 0, 4, 8,..., with each word consisting of four bytes.

Big-Endian and Little-Endian Assignments

- There are two ways that byte addresses can be assigned across words : big-endian and little-endian.
- When lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word, addressing is called big endian.

- When the lower byte addresses are used for the less significant bytes (the rightmost byte) of the word, addressing is called little-endian.

addressing is called little-endian. This is illustrated in Fig. 9.7.12.



Interpret the storage as numbers in the manner indicated below and find their decimal values in each case.

- Big-endian storage of 2 hex words of 4-digits each.
- Big-endian storage of 2 BCD words of 4 -digits each.
- Little-endian storage, in ASCII, of a 4-digit signed hex word.
- Little endian storage, in ASCII, of a 4-digit BCD word.

Solution: We know that, big endian storage uses lower byte address for more-significant bytes and little-endian storage uses lower byte address for less-significant bytes. Therefore, in big-endian storage given two words are: 3834H, 3239H and in little-endian storage given two words are: 3932H, 3438H.

- Big-endian storage of 2 hex words of 4-digit each are: 3834H, 3239H

$$3834H = 3 \times 16^3 + 8 \times 16^2 + 3 \times 16^1 + 4 \times 16^0 = 14388 \text{ decimal}$$

$$3239H = 3 \times 16^3 + 2 \times 16^2 + 3 \times 16^1 + 9 \times 16^0 = 12857 \text{ decimal}$$

- Big-endian storage of 2 BCD words of 4-digit each are: 3834, 3239.

$$3834 \text{ (BCD)} = 3834 \text{ decimal}$$

$$3239 \text{ (BCD)} = 3239 \text{ decimal}$$

- Little-endian storage in ASCII, of 4-digit signed hex word is : 9248H.

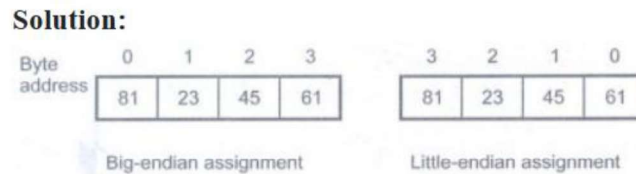
Note: We subtract 30H from ASCII code to get the digit value in hex.

$$9248H = 9 \times 16^3 + 2 \times 16^2 + 4 \times 16^1 + 8 \times 16^0 = 37448 \text{ decimal}$$

iv) Little-endian storage, in ASCII, of a 4-digit BCD word is : 9248

$$9248 \text{ BCD} = 9248 \text{ decimal}$$

Example 6.7.2 Represent the number 81234561 in 32-bit Big-endian and little-endian memory organization.



Word Alignment

- Word length is typically a multiple of 8, common CPU word length being 8, 16, 32 and 64-bits.
- When word lengths are more than 8-bits we have to take special care while accessing words. In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8,
- In general, words are said to be aligned in memory if they begin at the natural word boundaries. The addresses of such words are called aligned addresses.
- For 16-bit word length aligned addresses are 0, 2, 4, and so on. If the words begin at an arbitrary byte address, the address is called unaligned address. Most of the times we use aligned addresses.

Example 6.7.3 What is word alignment of a machine (microprocessor based system)? Explain. What are the consecutive addresses of aligned words for 16, 32 and 64 bit word lengths of machines? Give two consecutive addresses for each case.

Solution : Consecutive addresses

For 16 bit word lengths: 0, 2, 4, 6, ...

For 32-bit word lengths: 0, 4, 8, 12, ...

For 64-bit word lengths: 0, 8, 16, 24, ...

Accessing Numbers, Characters and Character Strings

- We have seen that the number is stored in the memory as a word. Thus it can be accessed by specifying its word address. On the other hand, the character is stored in the memory as a byte and it is accessed by specifying its byte address.

The character string is stored using successive memory locations in the memory. Since the character strings are of variable lengths we have to specify two things about the strings: 1. Starting address of the string 2. Length of the string.

- There are two ways to indicate the length of the string. A special control character with the meaning 'end of string' can be used as the last character in the string, or a separate number is used to specify the length of the string.
- Thus the string can be accessed by simply specifying the address of the first byte of the character (starting address) or by specifying the starting address and the number indicating the length of the string which can be in the separate word location or in the processor register.

Example 6.7.4 Consider a computer that has a byte addressable memory organized in 32 bit words according to the big Endian scheme. A program reads ASCII characters entered at a keyboard and store them in successive byte location starting at location 1000. Show the contents of the two memory words at location 1000 and 1004 after the name "Johnson" has been entered. (ASCII codes J = 4A H, o = 6F H, h = 68 H, n = 6E H, S = 73 H)

Solution: Byte contents in hex, starting at location 1000, will be 4A, 6F, 68, 6E, 73, 6F, 6E. The two words at 1000 and 1004 will be 4A6F686E and 736F6EXX. Byte 1007 (shown as XX) is unchanged.

Main Memory operations:

We know that both program instructions and data operands are stored in the memory. To execute an instruction processor has to fetch instruction and read the operand or operands if necessary from the memory. After execution of instruction processor may store the result or modified operand back in the memory. Thus, there are two basic operations required in the memory access: 1. Load (Read or fetch) 2. Store (write).

Load operation: In the load operation the contents of the specified memory location are read by the processor. For load operation processor sends the address of the memory location whose contents are to be read and generates the appropriate read signal to indicate that it is a read operation. The memory identifies the addressed memory location and sends the contents of that memory location to the processor. This operation is illustrated in Fig. 6.8.1 (a).

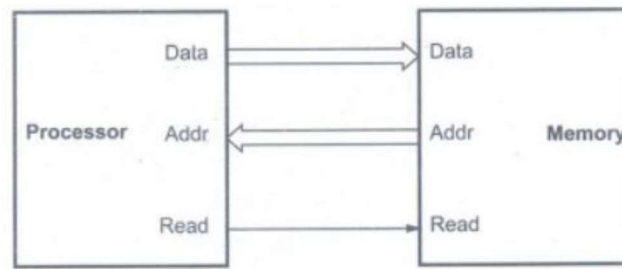


Fig. 6.8.1 (a) Load operation

Store operation: In the store operation the data from the processor is stored in the specified memory location. For store operation processor sends the address of the memory location where the data is to be written on the address lines, the data to be written on the data lines and generates the appropriate write signal to indicate the store operation. The memory identifies the addressed memory location and writes the data sent by the processor at that location destroying the former contents of that location. This operation is illustrated in Fig. 6.8.1 (b).

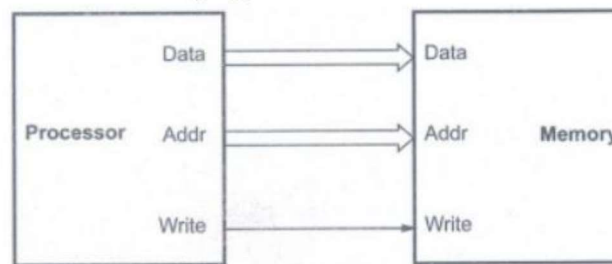


Fig. 6.8.1 (b) Store operation

The data items or instructions which are transferred between memory and processor may be either byte or word. They can be transferred using a single operation. Mainly this transfer takes place between processor registers and the memory locations.

We know that processor has small number of built-in registers. Each register is capable of holding a word data. These registers are either the source or the destination of a transfer to or from the memory.

Instructions and Instruction Sequencing

A computer has a set of instructions that allows the user to formulate any data-processing task. To carry out tasks, regardless of whether a computer has 100 instructions or 300 instructions, its instructions must be capable of performing following basic operations:

Data transfers between the memory and the processor registers.

- Arithmetic and logic operations on data.
- Program sequencing and control.

- I/O control.

Register Transfer Notation

- We have seen that in a computer system data transfer takes place between processor registers and memory and between processor registers and I/O system. These data transfers can be represented by standard notations given below :
- Processor registers are represented by notations R0, R1, R2,... and so on.
- The addresses of the memory locations are represented by names such as LOC, PLACE, MEM, etc.
- I/O registers are represented by names such as DATAIN, DATAOUT and so on.
- The contents of register or memory location are denoted by placing square brackets around the name of the register or memory location.

Let us see following examples for clear understanding.

Example: $R2 \leftarrow [LOC]$

- This expression states that the contents of memory location LOC are transferred into the processor register R2.

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

- This expression states that the contents of processor registers R1 and R2 are added and the result is stored into the processor register R3.

Example: $[LOC] \leftarrow [R1] - [R2]$

- This expression states that the contents of the processor register R2 is subtracted from processor register and the result is stored into the memory location LOC.
- The notations explained above are commonly known as Register Transfer Notations (RTN). In these notations, the data represented by the right-hand side of the expression is transferred to the location specified by the left hand side of the expression, overwriting the old contents of that location.

Assembly Language Notations

- Assembly language notations are the another type of notations used to represent machine instructions and programs. These notations use assembly language formats. However, register names, names of memory locations are same as that of register notations.

- Let us see some examples.

Example: MOVE R2, R1

- This expression states that the contents of processor register R2 are transferred to processor register R1. Thus the contents register R2 remain unchanged but contents of register R1 are overwritten.

Example: ADD R1, R2, R3

- This expression states that the contents of processor registers R1 and R2 are added and the result is stored in the register R3.
- It is important to note that the above expressions written in the assembly language notations has three fields: Operation, source and destination having their positions from left to right. This order is followed by many computer. But there are many computers in which the order of source and destination operands is reversed.

Basic Instruction Types

- Each instruction of the CPU contain specific information fields, which are required to execute it. These information fields of instructions are called elements of instruction. These are:

- **Operation code :** The operation code field in the instruction specifies the operation to be performed. The operation is specified by binary code, hence the name operation code or simply opcode.
- **Source / Destination operand:** The source/destination operand field directly specifies the source/destination operand for the instruction.
- **Source operand address:** We know that the operation specified by the instruction may require one or more operands. The source operand may be in the CPU register or in the memory. Many times the instruction specifies the address of the source operand so that operand(s) can be accessed and operated by the CPU according to the instruction.
- **Destination operand address:** The operation executed by the CPU may produce result. Most of the times the result is stored in one of the operand. Such operand is known as destination operand. The instruction which produce result specifies the destination operand address.

Next instruction address: The next instruction address tells the CPU from where to fetch the next instruction after completion of execution of current instruction. For JUMP and BRANCH

instructions the address of the next instruction is specified within the instruction. However, for other instructions, the next instruction to be fetched immediately follows the current instruction.

- Computers may have instructions of several different lengths containing varying number of addresses. According to address reference there are three address, two address, one address and zero address reference instructions. Let us see examples of each of them.

Three address instructions : The three address instruction can be represented symbolically as

ADD A, B, C

where A, B, C are the variables. These variable names are assigned to distinct locations in the memory. In this instruction operands A and B are called source operands and operand C is called destination operand and ADD is the operation to be performed on the operands. Thus the general instruction format for three address instruction is Operation Source 1, Source 2, Destination

The number of bits required to represent such instruction include:

1. Bits required to specify the three memory addresses of the three operands. If n -bits are required to specify one memory address, $3n$ bits are required to specify three memory addresses.
2. Bits required to specify the operation.

Two address instructions : The two address instruction can be symbolically as

ADD A, B

This instruction adds the contents of variables A and B and stores the sum in variable B destroying its previous contents. Here, operand A is source operand ; however operand B serves as both source and destination operand. The general instruction format for two address instruction is

Operation Source, Destination

To represent this instruction less number of bits are required as compared to three address instruction. The number of bits required to represent two address instruction include:

1. Bits required to specify the two memory addresses of the two operands, i.e. $2n$ bits.
2. Bits required to specify the operation.

One address instruction : The one address instruction can be represented symbolically as

ADD B

This instruction adds the contents of variable A into the processor register called accumulator and stores the sum back into the accumulator destroying the previous contents of the accumulator. In this instruction the second operand is assumed implicitly in a unique location accumulator. The general instruction format for one address instruction is

Operation Source

Few more examples of one address instructions are :

LOAD A: This instruction copies the contents of memory location A into the accumulator.

STORE B: This instruction copies the contents of accumulator into memory location B.

In one address instruction, it is important to note that the operand specified in the instruction can be either source operand or destination operand depending on the instruction. For example, in LOAD A instruction, the operand specified in the instruction is a source operand whereas the operand specified in the STORE B instruction is a destination operand. Similarly, in one address instruction the implied operand (accumulator) can be either source or destination depending on the instruction.

Zero address instructions: In these instructions, the locations of all operands are defined implicitly. Such instructions are found in machines that store operands in a structure called a pushdown stack.

From above discussion we can easily understand that the instruction with only one address will require less number of bits to represent it, and instruction with three addresses will require more number of bits to represent it. Therefore, to access entire instruction from the memory, the instruction with three addresses requires more memory accesses while instruction with one address requires less memory accesses. The speed of instruction execution is mainly depend on how much memory accesses it requires for the execution. If memory accesses are more, more time is required to execute the instruction. Therefore, the execution time for three address instructions is more than the execution time for one address instructions.

To have a less execution time we have to use instructions with minimum memory accesses. For this instead of referring the operands from memory it is advised to refer operands from processor registers. When machine level language programs are generated by compilers from high-level languages, the intelligent compilers see that the maximum references to the operands lie in the processor registers.

Example 6.9.1 Write a program to evaluate the arithmetic statement $Y = (A+B) * (C+D)$ using three-address, two-address, one-address and zero-address instructions.

Solution :

Using three address instructions

ADD A, B, R1 ; $R1 \leftarrow M[A] + M[B]$

ADD C, D, R2 ; $R2 \leftarrow M[C] + M[D]$

MUL R1, R2, Y ; $M[Y] \leftarrow R1 * R2$

Using two address instructions

MOV A, R1 ; $R1 \leftarrow M[A]$

ADD B, R1 ; $R1 \leftarrow R1 + M[B]$

MOV C, R2 ; $R2 \leftarrow M[C]$

ADD D, R2 ; $R2 \leftarrow R2 + M[D]$

MUL R2, R1 ; $R1 \leftarrow R1 * R2$

MOV R1, Y ; $[Y] \leftarrow R1$

Using one address instruction

LOAD A ; $AC \leftarrow M[A]$

ADD B ; $AC \leftarrow AC + M[B]$

STORE T ; $M[T] \leftarrow AC$

LOAD C ; $AC \leftarrow M[C]$

ADD D ; $AC \leftarrow AC + M[D]$

MUL T ; $AC \leftarrow AC * M[T]$

STORE Y ; $M[Y] \leftarrow AC$

Using zero address instructions

PUSH A ; $TOS \leftarrow A$

PUSH B ; $TOS \leftarrow B$

ADD ; $TOS \leftarrow (A+B)$

```

PUSH   C       ;   TOS←C
PUSH   D       ;   TOS←D
ADD                      ;   TOS←(C+D)
MUL                      ;   TOS←(C+D)*(A+B)
POP     Y       ;   M[Y]←TOS

```

Instruction Execution and Straight Line Sequencing

- We have seen that instruction consists of opcode or opcode and operand/s or opcode and operand address.
- Every processor has some basic types of instructions such as data transfer instructions, arithmetic instructions, logical instructions, branch instructions and so on.
- To perform a particular task on the computer, it is programmers job to select and write appropriate instructions one after the other, i.e. programmer has to write instructions in a proper sequence. This job of programmer is known as instruction sequencing.
- The instructions written in a proper sequence to execute a particular task is called program.
- Processor executes a program with the help of Program Counter (PC). PC holds the address of the instruction to be executed next.
- To begin execution of a program, the address of its first instruction is placed into the PC. Then, the processor control circuits use the information (address of memory) in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called straight-line sequencing.
- During the execution of instruction, the PC is incremented by the length of the current instruction in execution. For example, if currently executing instruction length is 3 bytes, the PC is incremented by 3 so that it points to the instruction to be executed next.

Branching

- Every time it is not possible to store a program in the consecutive memory locations. After execution of decision making instruction we have to follow one of the two program sequences.
- In such cases we can not use straight-line sequencing. Here, we have to use branch instructions to transfer the program control from one straight-line sequence to another straight-line sequence of instruction, as shown in following program.

- For example, see the program given for operation $|A - B|$. In this program, we have to check whether $A > B$ or $B > A$ and accordingly we have to perform operation

$A - B$ or $B - A$.

MOV NUM1, R0 ; Get the number 1 into R0

MOV NUM2, R1 ; Get the number 2 into R1

CMP R0, R1 ; NUM1- NUM2

JB NEXT ; If NUM1 < Num2

; Jump to another program sequence

SUB R0, R1 ; NUM2← NUM2- NUM1

MOV R2, R1 ; Store the result in R2

:

:

SUB R1, R0 ; NUM2← NUM2- NUM1

MOV R2, R0 ; Store the result in R2

- In the above program we have used JB NEXT instruction to transfer the program control to the instruction SUB R1, R0 if NUM1 is less than NUM2. Thus we have decided to branch the program control after checking the condition. Such branch instructions are called conditional branch instructions.

- In branch instructions the new address called target address or branch target is loaded into PC and instruction is fetched from the new address, instead of the instruction at the location that follows the branch instruction in sequential address order.

- The conditional branch instructions are used for program looping. In looping, the program is instructed to execute certain set of instructions repeatedly to execute a particular task number of times. For example, to add ten numbers stored in the consecutive memory locations we have to perform addition ten times.

- The program loop is the basic structure which forces the processor to repeat a sequence of instructions. Loops have four sections.

1. Initialization section.

2. Processing section.

3. Loop control section.

4. Result section.

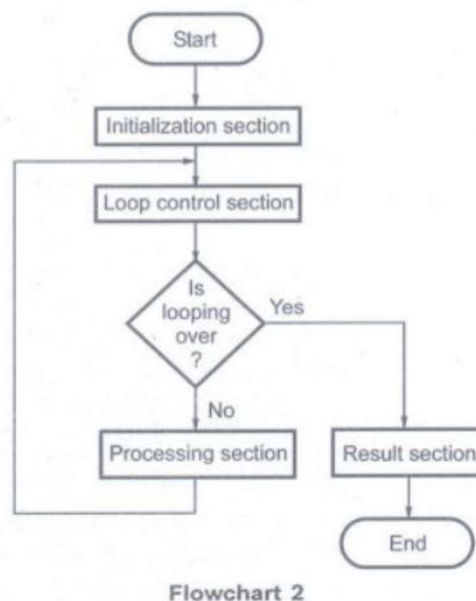
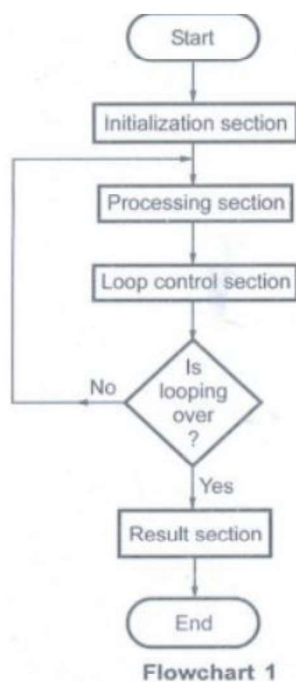
1. The initialization section establishes the starting values of

- loop counters for counting how many times loop is executed,
- address registers for indexing which give pointers to memory locations and
- other variables.

2. The actual data manipulation occurs in the processing section. This is the section which does the work.

3. The loop control section updates counters, indices (pointers) for the next iteration.

4. The result section analyzes and stores the results.



Note: The processor executes initialization section and result section only once, while it may execute processing section and loop control section many times. Thus, the execution time of the loop will be mainly dependent on the execution time of the processing section and loop control section. The flowchart 1 shows typical program loop. The processing section in this flowchart is always executed at least once. If you interchange the position of the processing and loop control section then it is possible that the processing section may not be executed at all, if necessary. Refer flowchart 2.

Conditional Codes

- The condition code flags are used to store the results of certain condition when certain operations are performed during execution of the program. The condition code flags are stored in the status registers. The status register is also referred to as flag register.
- ALU operations and certain register operations may set or reset one or more bits in the status register.
- Status bits lead to a new set of microprocessor instructions. These instructions permit the execution of a program to change flow on the basis of the condition of bits in the status register. So the condition bits in the status register can be used to take logical decisions within the program. Some of the common condition code flags are :

1. Carry/Borrow: The carry bit is set when the summation of two 8-bit numbers is greater than 1111 1111 (FFH). A borrow is generated when a large number is subtracted from a smaller number.

2. Zero: The zero bit is set when the contents of register are zero after any operation. This happens not only when you decrement the register, but also when any arithmetic or logical operation causes the contents of register to be zero.

3. Negative or sign: In 2's complement arithmetic, the most significant bit is a sign bit. If this bit is logic 1, the number is negative number, otherwise a positive number. The negative bit or sign bit is set when any arithmetic or logical operation gives a negative result.

4. Auxiliary carry: The auxiliary carry bit of status register is set when an addition in the first 4-bits causes a carry into the fifth bit. This is often referred as half carry or intermediate carry. This is used in the BCD arithmetic.

5. Overflow Flag: In 2's complement arithmetic, most significant bit is used to represent sign and remaining bits are used to represent magnitude of a number (see Fig. 6.9.1). This flag is set if the result of a signed operation is too large to fit in the number of bits available (7-bits for 8-bit number) to represent it.

For example, if you add the 8-bit signed number 01110110 (+118 decimal) and the 8-bit signed number 00110110 (+ 54 decimal). The result will be 10101100 (+ 172 decimal), which is the correct binary result, but in this case it is too large to fit in the 7-bits allowed for the magnitude in an 8-bit signed number. The overflow flag will be set after this operation to indicate that the result of the addition has overflowed into the sign bit.

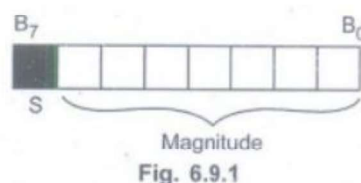


Fig. 6.9.1

6. Parity: When the result of an operation leaves the indicated register with an even number of 1's, the parity bit is set.

Generating Memory Addresses

- The address of the memory can be specified directly within the instruction. For example, `MOV [2000H], R1`. In this instruction the memory address is fixed; it can not be dynamically changed in the program itself. There are some situations where we need to change the memory address dynamically.
- Let us see the example program. In this program the contents from the array of data are added to get the total sum of all array elements.
- We know for this we have to repeat the add instruction number of times equal to the array elements and each time we have to add a number from a new successive memory location.
- Every time the address of memory is different. So to change the address of memory each time when we enter the loop, an address variable is used. Such addressing is called indirect addressing. For example, `ADD R1, [R2]`.
- Here, the contents of R2 register are used as an address of memory location. By incrementing the contents of register R2 it is possible to change the memory address each time we enter the loop.
- **Note:** The instruction used in the program given below specifies first destination operand and then the source operand.

`MOV R2, Array_start` ; Load the starting address of the array

`MOV R0, Count` ; Initialize the counter

`MOV R1, 0` ; Result = 0

`ADD R1, [R2]` ; $\text{Result} \leftarrow \text{Result} + \text{array element}$

`INC R2` ; increment memory pointer

`DEC R0` ; Decrement count

`JNZ BACK` ; if count 0, repeat

UNIT-III

Input / Output organization

Introduction :

- A general purpose computer should have the ability to exchange information with a wide range of devices in varying environments.
- Computers can communicate with other computers over the Internet and access information around the globe.
- They are an integral part of home appliances, manufacturing equipment, transportation systems, banking and point-of-sale terminals.

Accessing I/O Devices:

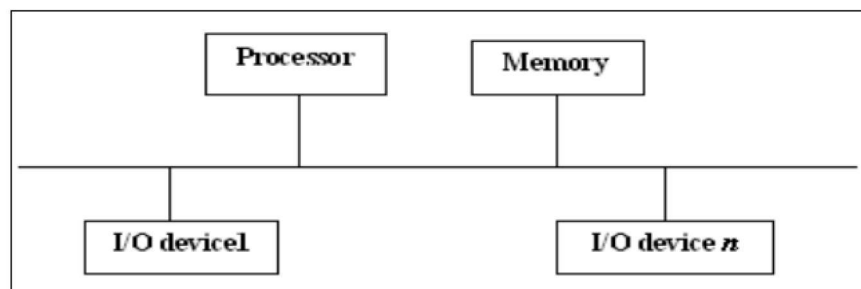


Figure: A single-bus structure

The I/O devices are connected by using a single bus which enables data transaction between each device as shown in figure.

- Bus consists of 3 sets of lines used to carry address, data and control signals.
- Each I/O device is assigned a unique set of address. When the processor places a particular address on the address lines, the device that recognizes this address responds to the commands issued on the control lines.
- The processor requests either a read or a write operation which is transferred over the data lines. When I/O devices and the memory share the same address space, the arrangement is called memory-mapped I/O. There are two ways to deal with I/O devices as shown above figure. Memory mapped I/O The Memory and I/O devices are sharing information by a common address-space. Any data-transfer instruction (like Move, Load) can be used to exchange information.

For example, Move DATAIN, R0;

- This instruction reads data from DATAIN (address of input-buffer associated with Keyboard) & stores them into processor-register R0. Move R0, DATAOUT
- This instruction sends the contents of register R0 to location DATAOUT, which may be the output data buffer of a display unit or a printer.

I/O mapped I/O :

In I/O mapped I/O, the memory and I/O address-spaces are different. Special instructions are used for data transfer such as IN and OUT.

- The I/O devices use special I/O address space or memory address space.
- The I/O devices examine the low-order bits of the address bus to determine whether they should respond.
- Advantage of separate I/O space: I/O devices deal with fewer address-lines.

The hardware required to connect an I/O device to the bus as shown in the figure.

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

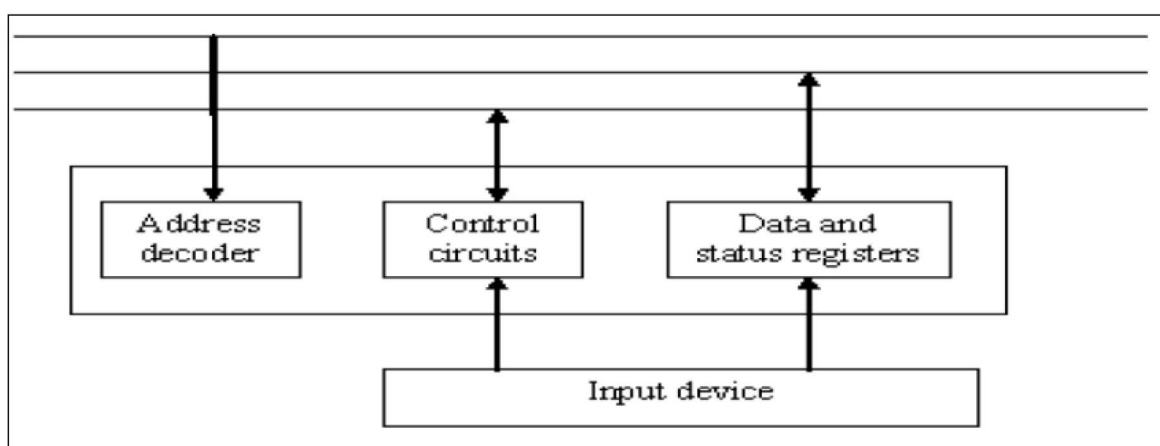


Figure: I/O interface for an input device

Program Controlled I/O :

- Processor repeatedly checks a status-flag to achieve required synchronization between processor & input/output device. (We say that the processor polls the device).

Main drawback: The processor wastes its time in checking the status of the device before actual data transfer takes place.

Mechanisms Used For Interfacing I/O Operations:

Interrupt I/O:

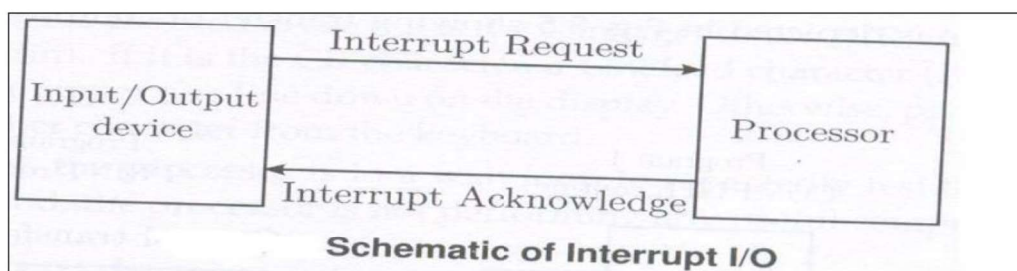
- Synchronization is achieved by having I/O device send a special signal over bus whenever it is ready for a data transfer operation.

Direct Memory Access (DMA) :

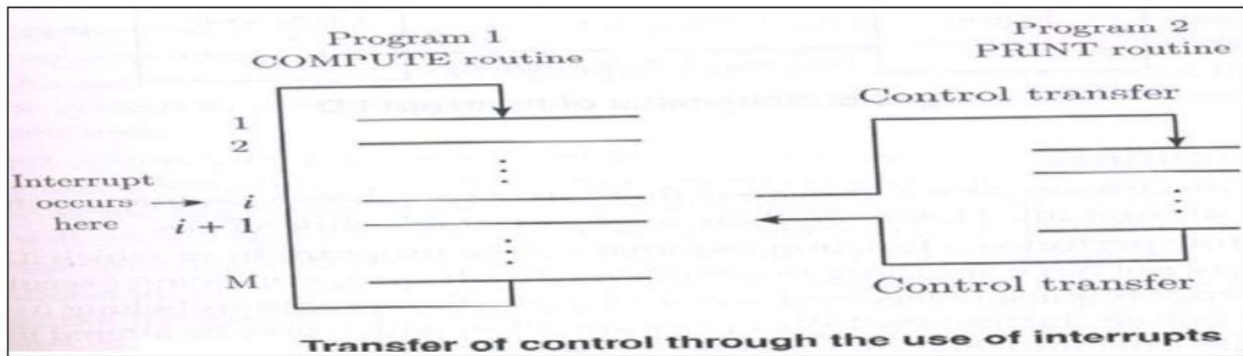
- This involves having the device-interface transfer data directly to or from the memory without continuous involvement by the processor. This technique is used for high-speed I/O devices.

Interrupts:

- There are many situations where other tasks can be performed while waiting for an I/O device to become ready. A hardware signal called an Interrupt will alert the processor when an I/O device becomes ready. It can do so by sending a hardware signal called an interrupt to the processor.



- For example, consider, COMPUTE and PRINT routines. The routine executed in response to an interrupt request is called interrupt-service routine.
- Transfer of control through the use of interrupts happens. The processor must inform the device that its request has been recognized by sending interrupt-acknowledge signal.



- One must therefore know the difference between Interrupt Vs Subroutine. Interrupt latency is concerned with saving information in registers will increase the delay between the time an interrupt request is received and the start of execution of the interrupt-service routine.
- Processor is executing the instruction located at address i when an interrupt occurs.
- Routine executed in response to an interrupt request is called the interrupt-service routine.
- When an interrupt occurs, control must be transferred to the interrupt service routine.
- But before transferring control, the current contents of the PC ($i+1$), must be saved in a known location.
- This will enable the return-from-interrupt instruction to resume execution at $i+1$.
- Return address, or the contents of the PC are usually stored on the processor stack.

Interrupt latency:

Delay between the time an interrupt request is received and the start of execution of the interrupt service routine.

Interrupt Hardware:-

- An I/O device requests an interrupt by activating a bus-line called interrupt-request (IR).
- A single interrupt request line may be used to serve n devices as depicted. All devices are connected to the line via switches to ground. To request an interrupt, a device closes its associated switch. Thus, if all interrupt-request signals $INTR_1$ to $INTR_n$ are inactive, that is, if all switches are open, the voltage on the interrupt-request line will be equal to V_{dd} .
- This is the inactive state of the line. Since the closing of one or more switches will cause the line voltage to drop to 0, causing the interrupt request signal, $INTR$ received by the processor to go to 1.

- The value of INTR is the logical OR of the requests from individual devices, that is, $INTR = INTR_1 + \dots + INTR_n$. It is customary to use the complemented form, $\overline{INTR}$, to name the interrupt-request signal on the common line, because this signal is active when in the low-voltage state.

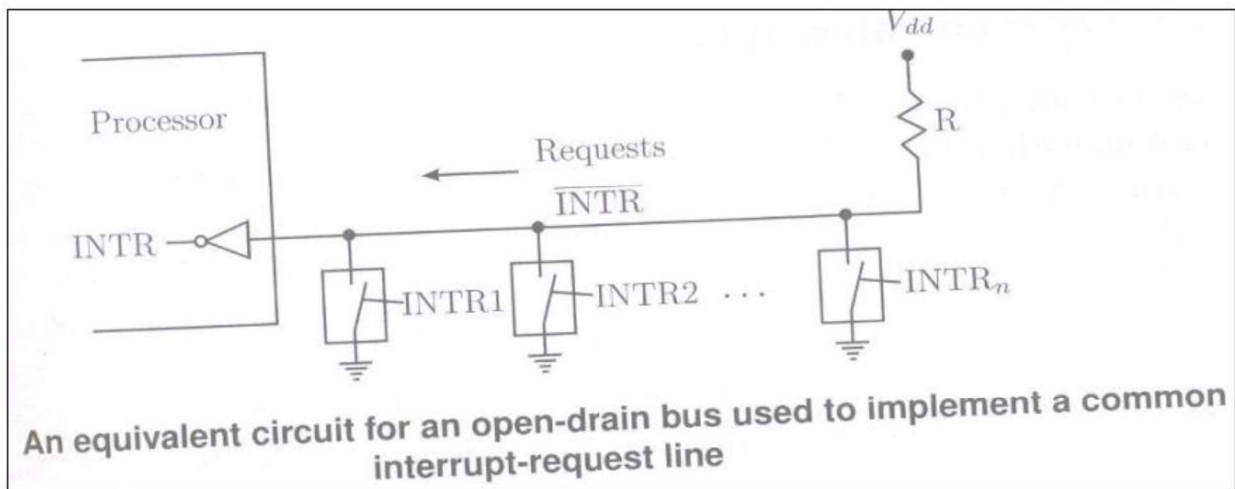


Figure: An equivalent circuit for an open-drain bus used to implement a common interrupt-request line

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

Enabling and Disabling Interrupts:

- The facilities provided in a computer must give the programmer complete control over the events that take place during program execution. The arrival of an interrupt request from an external device causes the processor to suspend the execution of one program and start the execution of another.
- Because interrupts can arrive at any time, they may alter the sequence of events from the envisaged by the programmer. Hence, the interruption of program execution must be carefully controlled.
- Let us consider in detail the specific case of a single interrupt request from one device. When a device activates the interrupt-request signal, it keeps this signal activated until it learns that the processor has accepted its request.
- This means that the interrupt-request signal will be active during execution of the interrupt service routine, perhaps until an instruction is reached that accesses the device.

- This activated signal, if not deactivated, may lead to successive interruptions, causing the system to enter into an infinite loop. To prevent the system from entering into an infinite-loop because of interrupt, there are 3 possibilities:

First Option

- The first possibility is to have the processor hardware ignore the interrupt-request line until the execution of the first instruction of the interrupt-service routine has been completed.
- Then, by using an Interrupt-disable instruction as the first instruction in the interrupt-service routine, the programmer can ensure that no further interruptions will occur until an Interruptenable instruction is executed.
- Typically, the Interrupt-enable instruction will be the last instruction in the interrupt-service routine before the Return-from-interrupt instruction. The processor must guarantee that execution of the Return-from-interrupt instruction is completed before further interruption can occur.

Second Option

- The second option is to have the processor automatically disable interrupts before starting the execution of the ISR.
- After saving the contents of the PC and the processor status register (PS) on the stack, the processor performs the equivalent of executing an Interrupt-disable instruction. It is often the case that one bit in the PS register, called Interrupt-enable, indicates whether interrupts are enabled.

Third option

- In the third option, the processor has a special interrupt-request line for which the interrupthandling circuit responds only to the leading edge of the signal. Such a line is said to be edgetriggered. Processor will receive only one request hence there will not be multiple interruption.

Summary:

Sequence of events involved in handling an interrupt-request from a single device is as follows:

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed.
3. Interrupts are disabled by changing the control bits in the PS (except in the case of edge-triggered interrupts).

4. The device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.
5. The action requested by the interrupt is performed by the interrupt-service routine.
6. Interrupts are enabled and execution of the interrupted program is resumed.

Handling Multiple Devices :

- An I/O device requests a service by activating a bus line called interrupt request. There may be more than one I/O devices in a system which request service. Since these devices operate independently, there is no specific order in which they generate interrupts. In another case, several devices may send request at the same time.

When several devices requests interrupt at the same time,

it raises some questions. They are

- How can the processor recognize the device requesting an interrupt?
- How can the processor obtain the starting address of the appropriate routine?
- Should a device be allowed to interrupt the processor while another interrupt is being serviced?
- How should two or more simultaneous interrupt requests be handled?
- Information needed to determine whether a device is requesting an interrupt is available in its status-register. When a device raises an interrupt-request, it sets IRQ bit to 1 in its status-register.
- KIRQ and DIRQ are the interrupt-request bits for keyboard & display. Simplest way to identify interrupting device is to have ISR poll all I/O devices connected to bus.
- The first device encountered with its IRQ bit set is the device that should be serviced. After servicing this device, next requests may be serviced.

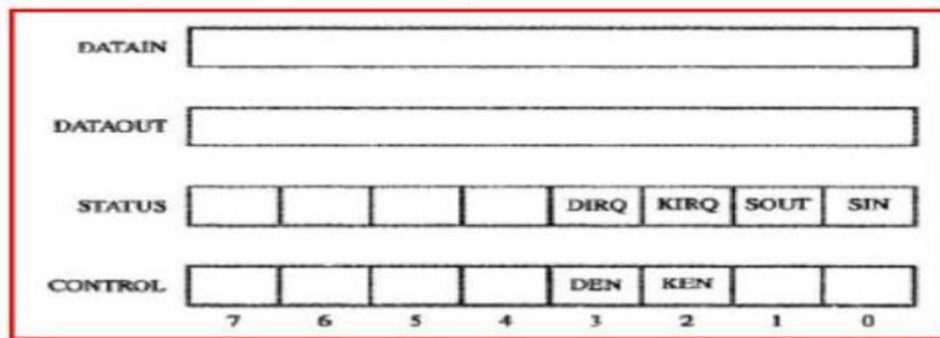


Figure 5: Registers in keyboard and display interfaces

WAITK	Move	#LINE,R0	Initialize memory pointer.
	TestBit	#0,STATUS	Test SIN.
	Branch=0	WAITK	Wait for character to be entered.
	Move	DATAIN,R1	Read character.
WAITD	TestBit	#1,STATUS	Test SOUT.
	Branch=0	WAITD	Wait for display to become ready.
	Move	R1,DATAOUT	Send character to display.
	Move	R1,(R0)+	Store character and advance pointer.
	Compare	#\$0D,R1	Check if Carriage Return.
	Branch≠0	WAITK	If not, get another character.
	Move	#\$0A,DATAOUT	Otherwise, send Line Feed.
	Call	PROCESS	Call a subroutine to process the the input line.

Figure 6: A program that reads one line from the keyboard, stores it in memory buffer, and echoes it back to the display

Vectored Interrupts:

- To reduce the polling process time, a device requesting an interrupt may identify itself directly to the processor. Now, the processor can immediately start executing the corresponding ISR. Such interrupts are called as vectored interrupts. A device requesting an interrupt can identify itself by sending a special code to the processor over the bus.
- This enables the processor to identify individual devices even if they share a single interruptrequest line. The code supplied by the device may represent the starting address of the interruptservice routine for that device.
- The code length is typically in the range of 4 to 8 bits. The remainder of the address is supplied by the processor based on the area in its memory where the addresses for interrupt-service routines are located.
- The address stored at the location pointed to by interrupting-device is called the interrupt-vector. The processor reads this address, called the interrupt vector, and loads it into PC then executes appropriate ISR. Interrupting-device must wait to put data on bus only when processor is ready to receive it.
- When processor is ready to receive interrupt-vector code, it activates INTA line. I/O device responds by sending its interrupt-vector code & turning off the INTR signal.

Interrupt Nesting :

- Computer keeps track of the time of the day using real-time clock. Device sends IR to processor at regular intervals to update time in seconds, minutes.
- For some device, a long delay in responding to an interrupt request may cause error in the operation of computer. Such interrupts are acknowledged and serviced even though processor is executing an interrupt service routine for another device system of interrupts that allows an interrupt service routine to be implemented is known as nested interrupts.

Multiple Priority Scheme:

- In multiple level priority schemes, we assign a priority level to the processor that can be changed under program control.
- The priority level of the processor is the priority of the program that is currently being executed.
- The processor accepts interrupts only from devices that have priorities higher than its own.
- At the time of execution of an ISR for some device is started, the priority of the processor is raised to that of the device.
- The action disables interrupts from devices at the same level of priority or lower.

Privileged Instruction:

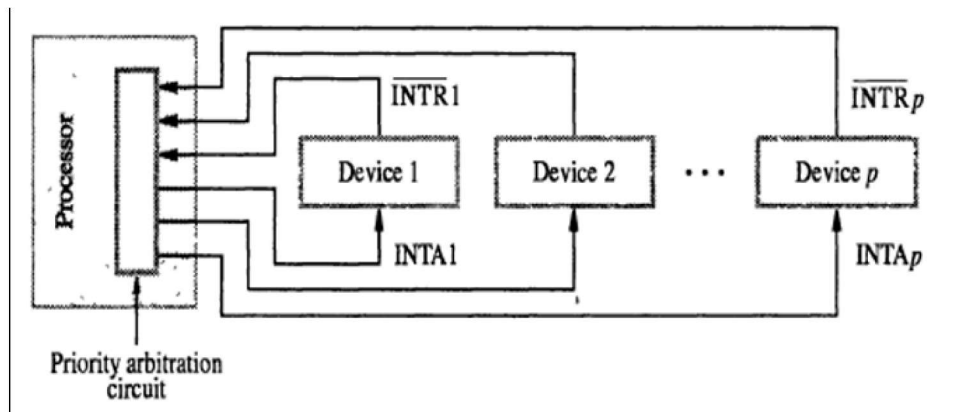
- The processor priority is usually encoded in a few bits of the Processor Status word. It can also be changed by program instruction & then it is writing into PS. These instructions are called privileged instruction.
- This can be executed only when the processor is in supervisor mode. The processor is in supervisor mode only when executing OS routines. It switches to the user mode before beginning to execute application program.

Privileged Exception:

- Processor will have privileged instructions to run in supervisor mode.

User program cannot accidentally or intentionally change the priority of the processor & disrupts the system operation. An attempt to execute a privileged instruction while in user mode, leads to a special type of interrupt called the privileged exception.

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



7: implementation of interrupt priority using individual interrupt-request and acknowledge lines

Simultaneous Requests:

Daisy Chain:

- The interrupt request line INTR is common to all devices. The interrupt acknowledge line INTA is connected in a daisy chain fashion such that INTA signal propagates serially through the devices.
- When several devices raise an interrupt request, the INTR is activated & the processor responds by setting INTA line to 1. This signal is received by device. Device1 passes the signal on to device2 only if it does not require any service. If device1 has a pending request for interrupt blocks that INTA signal & proceeds to put its identification code on the data lines. Therefore, the device that is electrically closest to the processor has the highest priority.

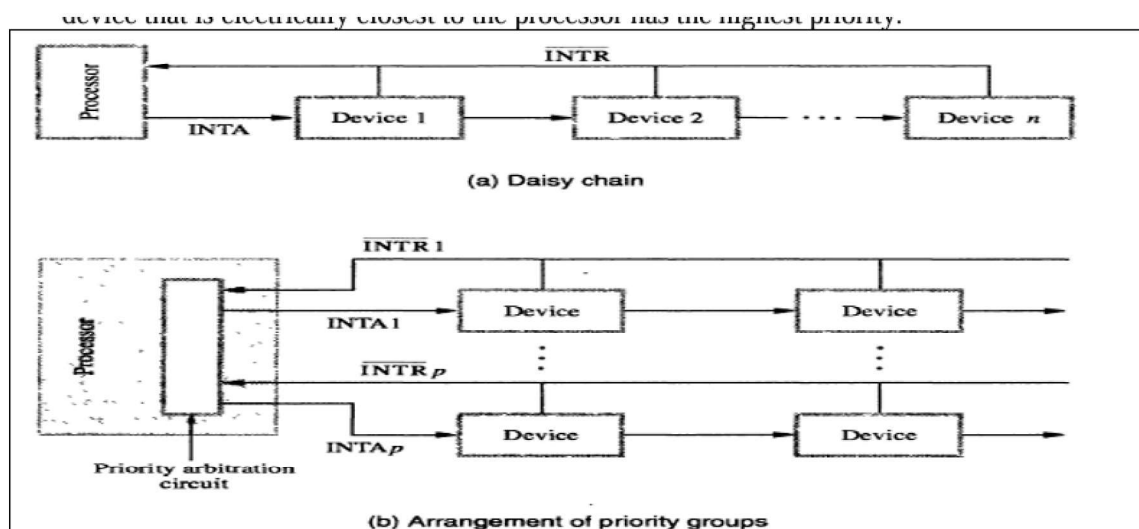


Figure 8: Interrupt priority schemes

Arrangement of Priority Groups:

Here the devices are organized in groups & each group is connected at a different priority level. Within a group, devices are connected in a daisy chain.

Controlling Device Requests

- The control needed is usually provided in the form of an interrupt-enable bit in the device's interface circuit.
- KEN = Keyboard Interrupt Enable
- DEN = Display Interrupt Enable
- KIRQ / DIRQ = Keyboard / Display unit requesting an interrupt.

There are two mechanisms for controlling interrupt requests.

- At the devices end, an interrupt enable bit in a control register determines whether the device is allowed to generate an interrupt requests.
- At the processor end, either an interrupt enable bit in the PS (Processor Status) or a priority structure determines whether a given interrupt requests will be accepted.

Initiating the Interrupt Process:

1. Load the starting address of ISR in location INTVEC (vectored interrupt).
2. Load the address LINE in a memory location PNTR. The ISR will use this location as a pointer to store the input characters in the memory.
3. Enable the keyboard interrupts by setting bit 2 in register CONTROL to 1.
4. Enable interrupts in the processor by setting to 1, the IE bit in the processor status register PS.

Exception of ISR:

1. Read the input characters from the keyboard input data register. This will cause the interface circuits to remove its interrupt requests.
2. Store the characters in a memory location pointed to by PNTR & increment PNTR.
3. When the end of line is reached, disable keyboard interrupt & inform program main.
4. Return from interrupt.

Exceptions

An interrupt is an event that causes the execution of one program to be suspended and the execution of another program to begin. The Exception is used to refer to any event that causes an interruption.

Recovery from Errors:

- Computers have error-checking code in Main Memory, which allows detection of errors in the stored data. If an error occurs, the control hardware detects it informs the processor by raising an interrupt.
- The processor also interrupts the program, if it detects an error or an unusual condition while executing the instance (i.e.) it suspends the program being executed and starts an execution service routine. This routine takes appropriate action to recover from the error.

Debugging:

System software has a program called debugger, which helps to find errors in a program. The debugger uses exceptions to provide two important facilities. They are Trace and Breakpoint.

Trace Mode:

- When processor is in trace mode, an exception occurs after execution of every instance using the debugging program as the exception service routine. The debugging program examine the contents of registers, memory location etc.
- On return from the debugging program the next instance in the program being debugged is executed. The trace exception is disabled during the execution of the debugging program.

Break point:

- Here the program being debugged is interrupted only at specific points selected by the user. An instance called the Trap (or) software interrupt is usually provided for this purpose.
- While debugging the user may interrupt the program execution after instance „I“. When the program is executed and reaches that point it examine the memory and register contents.

Privileged Exception:

- To protect the OS of a computer from being corrupted by user program certain instance can be executed only when the processor is in supervisor mode.
- These are called privileged exceptions. When the processor is in user mode, it will not execute instance (i.e.) when the processor is in supervisor mode, it will execute instance.

Direct Memory Access (DMA) :

- A special control unit may be provided to allow the transfer of large block of data at high speed directly between the external device and main memory, without continuous intervention by the processor. This approach is called direct memory access (DMA). DMA transfers are performed by a control circuit called the DMA Controller.

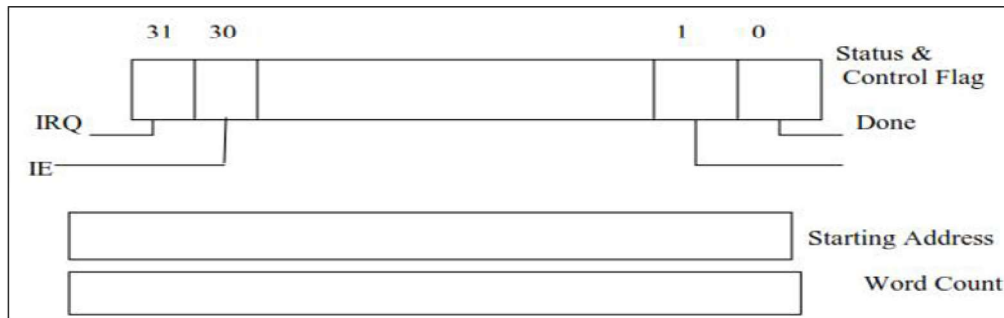


Figure 9: Registers in a DMA Interface

To initiate the transfer of a block of words, the processor sends, are Starting address, Number of words in the block and Direction of transfer.

- When a block of data is transferred, the DMA controller increment the memory address for successive words and keep track of number of words and it also informs the processor by raising an interrupt signal.
- While DMA control is taking place, the program requested the transfer cannot continue and the processor can be used to execute another program. After DMA transfer is completed, the processor returns to the program that requested the transfer.

The above figure shows an example of the DMA controller registers that are accessed by the processor to initiate transfer operations. Two registers are used for storing the Starting address and the word count. The third register contains status and control flags. The R/W bit determines the direction of the transfer.

When

- R/W =1, DMA controller read data from memory to I/O device.
- R/W =0, DMA controller perform write operation.
- Done Flag=1, the controller has completed transferring a block of data and is ready to receive another command.

- IE=1, it causes the controller to raise an interrupt (interrupt Enabled) after it has completed transferring the block of data.
- IRQ=1, it indicates that the controller has requested an interrupt.

A DMA controller connects a high speed network to the computer bus. The disk controller two disks, also has DMA capability and it provides two DMA channels. To start a DMA transfer of a block of data from main memory to one of the disks, the program writes the address and the word count information into the registers of the corresponding channel of the disk controller. When DMA transfer is completed, it will be recorded in status and control registers of the DMA channel (i.e.) done bit=IRQ=IE=1.

Cycle Stealing:

- Memory accesses by the processor and the DMA controller are interwoven. Requests by DMA devices for using the bus are always given higher priority than processor requests.
- Among different DMA devices, top priority is given to high-speed peripherals such as a disk, a high-speed network interface, or a graphics display device.
- Since the processor originates most memory access cycles, the DMA controller can be said to “steal” memory cycles from the processor. Hence, the interweaving technique is usually called cycle stealing.

Burst Mode:

- The DMA controller may be given exclusive access to the main memory to transfer a block of data without interruption. This is known as Burst/Block Mode.

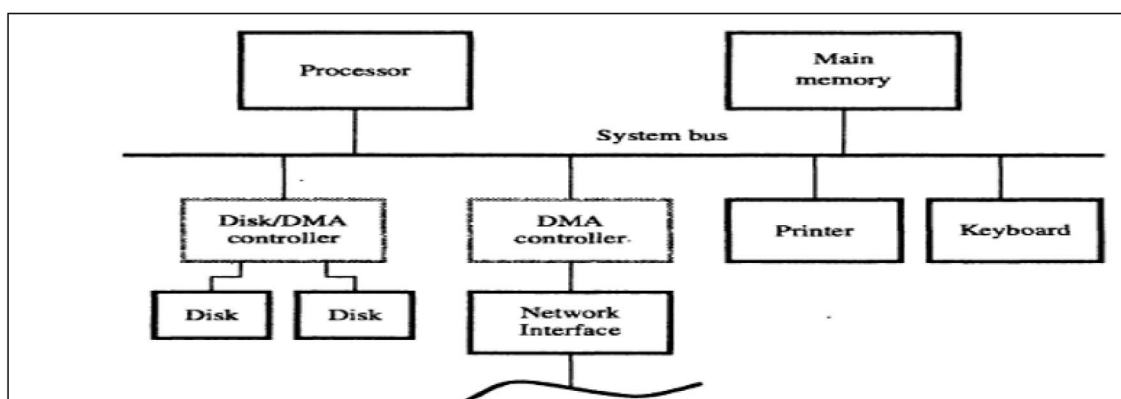


Figure 10: Use of DMA controllers in a computer system

Bus Arbitration

Bus Master:

The device that is allowed to initiate data transfers on the bus at any given time is called the bus master.

Bus Arbitration:

It is the process by which the next device to become the bus master is selected and the bus mastership is transferred to it.

There are 2 approaches to bus arbitration.

They are,

1. Centralized arbitration (A single bus arbiter performs arbitration)
2. Distributed arbitration (all devices participate in the selection of next bus master).

Centralized Arbitration:

- Here the processor is the bus master and it may grant bus mastership to one of its DMA controller. A DMA controller indicates that it needs to become the bus master by activating the Bus Request line (BR) which is an open drain line. The signal on BR is the logical OR of the bus request from all devices connected to it.

- When BR is activated the processor activates the Bus Grant Signal (BG1) and indicated the DMA controller that they may use the bus when it becomes free. This signal is connected to all devices using a daisy chain arrangement. If DMA requests the bus, it blocks the propagation of Grant Signal to other devices and it indicates to all devices that it is using the bus by activating open collector line, Bus Busy (BBSY).

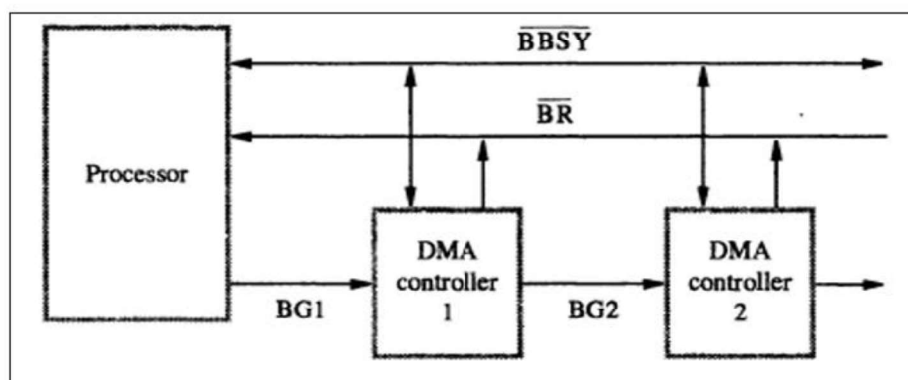


Figure 11: A simple arrangement for bus arbitration using a daisy chain

The timing diagram shows the sequence of events for the devices connected to the processor is shown. DMA Controller 2 requests and acquires bus mastership and later releases the bus. During its tenure as bus master, it may perform one or more data transfer operations, depending on whether it is operating in the cycle stealing or block mode. After it releases the bus, the processor resumes bus mastership.

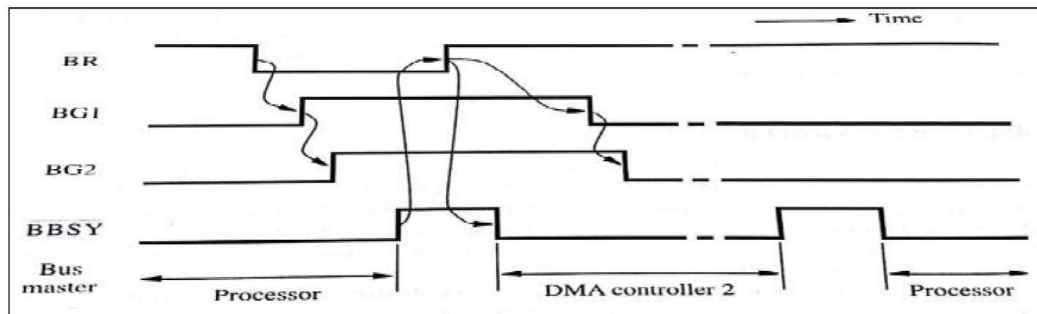


Figure 12: Sequence of signals during transfer of bus mastership for the devices in figure 11.

Distributed Arbitration:

- Distributed arbitration means that all devices waiting to use the bus have equal responsibility in carrying out the arbitration process, without using a central arbiter.
- Each device on the bus is assigned a 4 bit id. When one or more devices request the bus, they assert the Start-Arbitration signal and place their 4 bit ID number on four open collector lines, ARB0 to ARB3.
- A winner is selected as a result of the interaction among the signals transmitted over these lines. The net outcome is that the code on the four lines represents the request that has the highest ID number. The drivers are of open collector type.
- Hence, if the input to one driver is equal to 1, the input to another driver connected to the same bus line is equal to "0" the bus will be in low-voltage state.

Example:

- Assume two devices A and B have their ID 5 (0101), 6(0110) and their code is 0111. Each device compares the pattern on the arbitration line to its own ID, starting from most significant bit (MSB).
- If it detects a difference at any bit position, it disables the drivers at that bit position and for all lower-order bits. It does this by placing "0" at the input of these drivers. In our example, device "A" detects a difference in line ARB1.

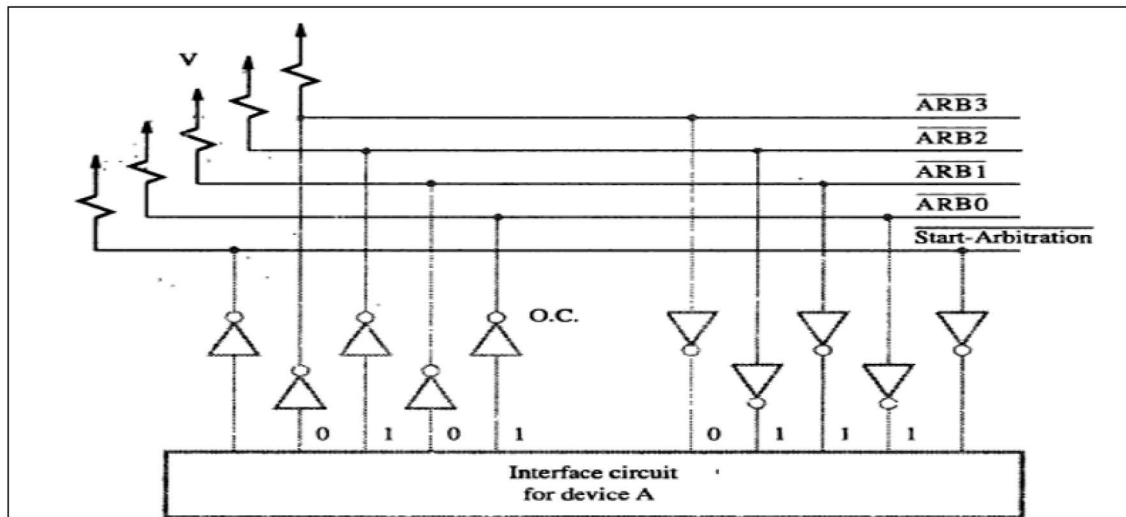


Figure 13: A distributed arbitration scheme

- Hence it disables the drivers on lines ARB1 and ARB0. This causes the pattern on the arbitration line to change to 0110 which means that “B” has won the contention.
- It has the advantage of offering higher reliability, because operation of the bus is not dependent on any single device.

I/O Hardware:

Introduction:

The I/O subsystem of a computer provides an efficient mode of communication between the central system and the outside environment. It handles all the input-output operations of the computer system.

Peripheral Devices

Input or output devices that are connected to computer are called peripheral devices. These devices are designed to read information into or out of the memory unit upon command from the CPU and are considered to be the part of computer system. These devices are also called peripherals.

For example: Keyboards, display units and printers are common peripheral devices.

There are three types of peripherals:

1. Input peripherals : Allows user input, from the outside world to the computer.

Example: Keyboard, Mouse etc.

2. Output peripherals: Allows information output, from the computer to the outside world.

Example: Printer, Monitor etc

3. Input-Output peripherals: Allows both input(from outside world to computer) as well as, output(from computer to the outside world).

Example: Touch screen etc.

Interfaces

Interface is a shared boundary between two separate components of the computer system which can be used to attach two or more components to the system for communication purposes.

There are two types of interface:

1. CPU Interface
2. I/O Interface

Let's understand the I/O Interface in details,

Input-Output Interface:

Peripherals connected to a computer need special communication links for interfacing with CPU. In computer system, there are special hardware components between the CPU and peripherals to control or manage the input-output transfers. These components are called input-output interface units because they provide communication links between processor bus and peripherals. They provide a method for transferring information between internal system and input-output devices.

Asynchronous Data Transfer:

We know that, the internal operations in individual unit of digital system are synchronized by means of clock pulse, means clock pulse is given to all registers within a unit, and all data transfer among internal registers occur simultaneously during occurrence of clock pulse. Now, suppose any two units of digital system are designed independently such as CPU and I/O interface.

And if the registers in the interface(I/O interface) share a common clock with CPU registers, then transfer between the two units is said to be synchronous. But in most cases, the internal timing in each unit is independent from each other in such a way that each uses its own private clock for its internal registers. In that case, the two units are said to be asynchronous to each other, and if data transfer occur between them this data transfer is said to be Asynchronous Data Transfer.

But, the Asynchronous Data Transfer between two independent units requires that control signals be transmitted between the communicating units so that the time can be indicated at which they send data.

This asynchronous way of data transfer can be achieved by two methods:

1. One way is by means of strobe pulse which is supplied by one of the units to other unit. When transfer has to occur. This method is known as “Strobe Control”.

2. Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another signal to acknowledge receipt of the data. This method of data transfer between two independent units is said to be “Handshaking”.

The strobe pulse and handshaking method of asynchronous data transfer are not restricted to I/O transfer. In fact, they are used extensively on numerous occasions requiring transfer of data between two independent units. So, here we consider the transmitting unit as source and receiving unit as destination.

As an example: The CPU, is the source during an output or write transfer and is the destination unit during input or read transfer.

And thus, the sequence of control during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination.

So, while discussing each way of data transfer asynchronously we see the sequence of control in both terms when it is initiated by source or when it is initiated by destination. In this way, each way of data transfer, can be further divided into parts, source initiated and destination initiated.

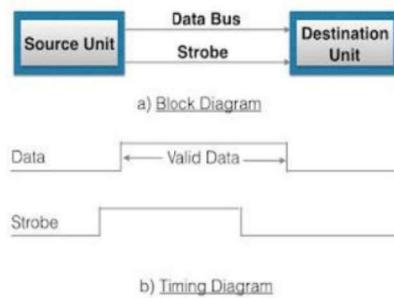
We can also specify, asynchronous transfer between two independent units by means of a timing diagram that shows the timing relationship that exists between the control and the data buses.

Now, we will discuss each method of asynchronous data transfer in detail one by one.

1. Strobe Control:

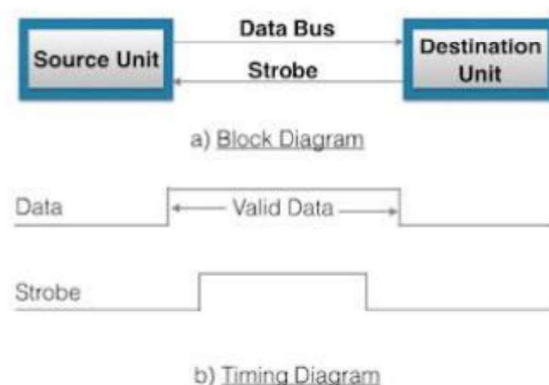
The Strobe Control method of asynchronous data transfer employs a single control line to time each transfer. This control line is also known as strobe and it may be achieved either by source or destination, depending on which initiates transfer.

Source initiated strobe for data transfer: The block diagram and timing diagram of strobe initiated by source unit is shown in figure below:



In block diagram we see that strobe is initiated by source, and as shown in timing diagram, the source unit first places the data on the data bus. After a brief delay to ensure that the data settle to a steady value, the source activates a strobe pulse. The information on data bus and strobe control signal remain in the active state for a sufficient period of time to allow the destination unit to receive the data. Actually, the destination unit, uses a falling edge of strobe control to transfer the contents of data bus to one of its internal registers. The source removes the data from the data bus after it disables its strobe pulse. New valid data will be available only after the strobe is enabled again.

Destination-initiated strobe for data transfer: The block diagram and timing diagram of strobe initiated by destination is shown in figure below:



In block diagram, we see that, the strobe initiated by destination, and as shown in timing diagram, the destination unit first activates the strobe pulse, informing the source to provide the data. The source unit responds by placing the requested binary information on the data bus. The data must be valid and remain in the bus long enough for the destination unit to accept it. The falling edge of strobe pulse can be used again to trigger a destination register. The destination unit then disables the strobe. And source removes the data from data bus after a per determine time interval.

Now, actually in computer, in the first case means in strobe initiated by source - the strobe may be a memory-write control signal from the CPU to a memory unit. The source, CPU, places the word on the data bus and informs the memory unit, which is the destination, that this is a write operation.

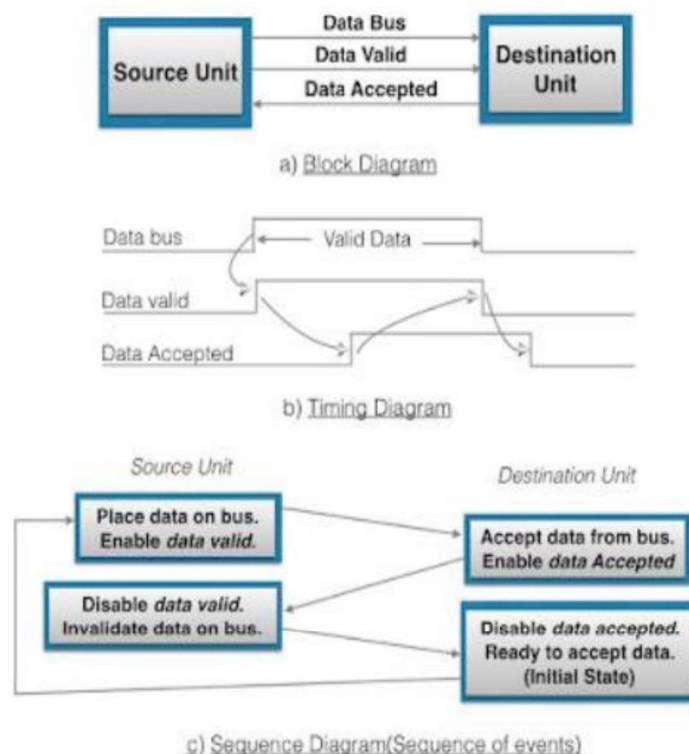
And in the second case i.e, in the strobe initiated by destination - the strobe may be a memory read control from the CPU to a memory unit. The destination, the CPU, initiates the read operation to inform the memory, which is a source unit, to place selected word into the data bus.

2. Handshaking: The disadvantage of strobe method is that source unit that initiates the transfer has no way of knowing whether the destination has actually 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 actually placed data on the bus. This problem can be solved by handshaking method.

Hand shaking method introduce a second control signal line that provides a replay to the unit that initiates the transfer.

In it, one control line is in the same direction as the data flow in the bus from the source to destination. It is used by source unit to inform the destination unit whether there are valid data in the bus. The other control line is in the other direction from destination to the source. It is used by the destination unit to inform the source whether it can accept data. And in it also, sequence of control depends on unit that initiate transfer. Means sequence of control depends whether transfer is initiated by source and destination. Sequence of control in both of them are described below:

Source initiated Handshaking: The source initiated transfer using handshaking lines is shown in figure below:

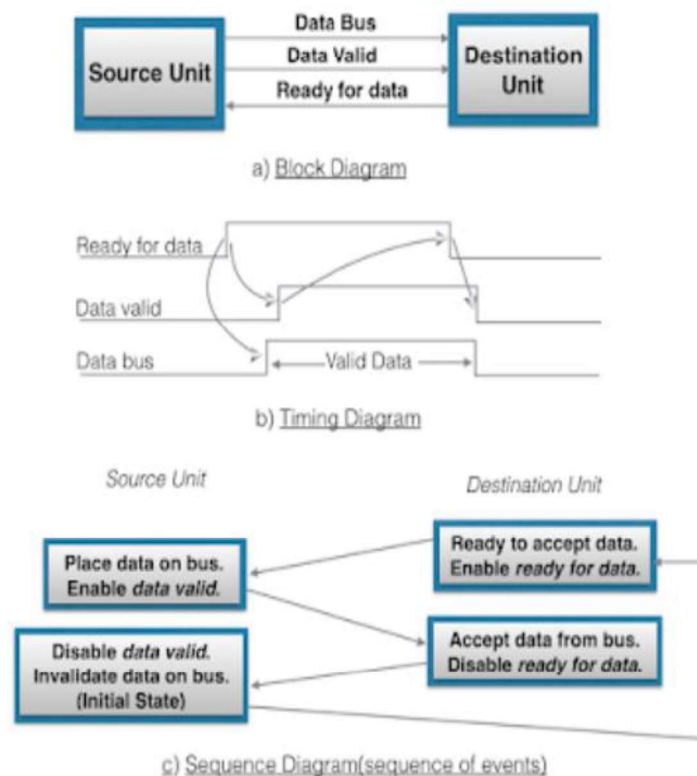


In its block diagram, we see that two handshaking lines are "data valid", which is generated by the source unit, and "data accepted", generated by the destination unit.

The timing diagram shows the timing relationship of exchange of signals between the two units. Means as shown in its timing diagram, the source initiates a transfer by placing data on the bus and enabling its data valid signal. The data accepted signal is then activated by destination unit after it accepts the data from the bus. The source unit then disable its data valid signal which invalidates the data on the bus. After this, the destination unit disables its data accepted signal and the system goes into initial state. The source unit does not send the next data item until after the destination unit shows its 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: The destination initiated transfer using handshaking lines is shown in figure below:



In its block diagram, we see that the two handshaking lines are "data valid", generated by the source unit, and "ready for data" generated by destination unit. Note that the name of signal data accepted generated by destination unit has been changed to ready for data to reflect its new meaning.

In it, transfer is initiated by destination, so source unit does not place data on data bus until it receives ready for data signal from destination unit. After that, hand shaking process is same as that of source initiated.

The sequence of events in it are shown in its sequence diagram and timing relationship between signals is shown in its timing diagram.

Thus, here we can say that, sequence of events in both cases would be identical. If we consider ready for data signal as the complement of data accept. Means, the only difference between source and destination initiated transfer is in their choice of initial state.

Modes of I/O Data Transfer

Data transfer between the central unit and I/O devices can be handled in generally three types of modes which are given below:

1. Programmed I/O
2. Interrupt Initiated I/O
3. Direct Memory Access

Programmed I/O

Programmed I/O instructions are the result of I/O instructions written in computer program. Each data item transfer is initiated by the instruction in the program. Usually the program controls data transfer to and from CPU and peripheral. Transferring data under programmed I/O requires constant monitoring of the peripherals by the CPU.

Interrupt Initiated I/O

In the programmed I/O method the CPU stays in the program loop until the I/O unit indicates that it is ready for data transfer. This is time consuming process because it keeps the processor busy needlessly.

This problem can be overcome by using interrupt initiated I/O. In this when the interface determines that the peripheral is ready for data transfer, it generates an interrupt. After receiving the interrupt signal, the CPU stops the task which it is processing and service the I/O transfer and then returns back to its previous processing task.

Direct Memory Access

Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This technique is known as DMA.

In this, the interface transfer data to and from the memory through memory bus. A DMA controller manages to transfer data between peripherals and memory unit.

Many hardware systems use DMA such as disk drive controllers, graphic cards, network cards and sound cards etc. It is also used for intra chip data transfer in multicore processors. In DMA, CPU would initiate the transfer, do other operations while the transfer is in progress and receive an interrupt from the DMA controller when the transfer has been completed.

Standard I/O interfaces:

Consider a computer system using different interface standards. Let us look in to Processor bus and Peripheral Component Interconnect (PCI) bus. These two buses are interconnected by a circuit called bridge. It is a bridge between processor bus and PCI bus. An example of a computer system using different interface standards is shown in figure 4.38.

The three major standard I/O interfaces discussed here are:

- PCI (Peripheral Component Interconnect)
- SCSI (Small Computer System Interface)
- USB (Universal Serial Bus)

Peripheral Component Interconnect (PCI) :

Bus The topics discussed under PCI are: Data Transfer, Use of a PCI bus in a computer system, A read operation on the PCI bus, Device configuration and Other electrical characteristics. Use of a PCI bus in a computer system is shown in figure 4.39 as a representation.

Host, main memory and PCI bridge are connected to disk, printer and Ethernet interface through PCI bus. At any given time, one device is the bus master. It has the right to initiate data transfers by issuing read and write commands. A master is called an initiator in PCI terminology. This is either processor or DMA controller. The addressed device that responds to read and write commands is called a target. A complete transfer operation on the bus, involving an address and a burst of data, is called a transaction. Device configuration is also discussed.

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

→ The PCI follows a sequence of bus standards that were used primarily in IBM PCs. Early PCs used the 8-bit XT bus, whose signals closely mimicked those of Intel's 80x86 processors. Later, the 16-bit bus used on the PC At computers became known as the ISA bus. Its extended 32-bit version is known as the EISA bus.

→ Other buses developed in the eighties with similar capabilities are the Microchannel used in IBM PCs and the NuBus used in Macintosh computers.

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

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

Data Transfer

→ In today's computers, most memory transfers involve a burst of data rather than just one word. The reason is that modern processors include a cache memory. Data are transferred between the cache and the main memory in burst of several words each.

→ The words involved in such a transfer are stored at successive memory locations. When the processor (actually the cache controller) specifies an address and requests a read operation from the main memory, the memory responds by sending a sequence of data words starting at that address. Similarly, during a write operation, the processor sends a memory address followed by a sequence of data words, to be written in successive memory locations starting at the address.

→ The PCI is designed primarily to support this mode of operation. A read or write operation involving a single word is simply treated as a burst of length one.

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

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

→ The signaling convention on the PCI bus is similar to the one used, we assumed that the master maintains the address information on the bus until data transfer is completed. But, this is not necessary. The address is needed only long enough for the slave to be selected. The slave can store the address in its internal buffer. Thus, the address is needed on the bus for one clock cycle only, freeing the address lines to be used for

sending data in subsequent clock cycles. The result is a significant cost reduction because the number of wires on a bus is an important cost factor. This approach is used in the PCI bus.

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

→ The addressed device that responds to read and write commands is called a target.

Device Configuration:

→ When an I/O device is connected to a computer, several actions are needed to configure both the device and the software that communicates with it.

→ The PCI simplifies this process by incorporating in each I/O device interface a small configuration ROM memory that stores information about that device. The configuration ROMs of all devices is accessible in the configuration address space.

→ The PCI initialization software reads these ROMs whenever the system is powered up or reset. In each case, it determines whether the device is a printer, a keyboard, an Ethernet interface, or a disk controller. It can further learn about various device options and characteristics. Devices are assigned addresses during the initialization process. This means that during the bus configuration operation, devices cannot be accessed based on their address, as they have not yet been assigned one. Hence, the configuration address space uses a different mechanism. Each device has an input signal called Initialization Device Select, IDSEL#.

→ The PCI bus has gained great popularity in the PC world. It is also used in many other computers, such as SUNs, to benefit from the wide range of I/O devices for which a PCI interface is available.

→ In the case of some processors, such as the Compaq Alpha, the PCI-processor bridge circuit is built on the processor chip itself, further simplifying system design and packaging.

SCSI Bus :

It is a standard bus defined by the American National Standards Institute (ANSI). A controller connected to a SCSI bus is an initiator or a target. The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

- The SCSI controller contends for control of the bus (initiator).
- When the initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.
- The target starts an output operation. The initiator sends a command specifying the required read operation.
- The target sends a message to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
- The acronym SCSI stands for Small Computer System Interface. It refers to a standard bus defined by the American National Standards Institute (ANSI) under the designation X3.131 .
- In the original specifications of the standard, devices such as disks are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates up to 5 megabytes/s.
- The SCSI bus standard has undergone many revisions, and its data transfer capability has increased very rapidly, almost doubling every two years. SCSI-2 and SCSI-3 have been defined, and each has several options.
- A SCSI bus may have eight data lines, in which case it is called a narrow bus and transfers data one byte at a time.
- Alternatively, a wide SCSI bus has 16 data lines and transfers data 16 bits at a time.
- There are also several options for the electrical signaling scheme used.
- Devices connected to the SCSI bus are not part of the address space of the processor in the same way as devices connected to the processor bus.
- The SCSI bus is connected to the processor bus through a SCSI controller. This controller uses DMA to transfer data packets from the main memory to the device, or vice versa.
- A packet may contain a block of data, commands from the processor to the device, or status information about the device.
- Communication with a disk drive differs substantially from communication with the main memory.

- A controller connected to a SCSI bus is one of two types
 - an initiator or a target.
- An initiator has the ability to select a particular target and to send commands specifying the operations to be performed.
- Clearly, the controller on the processor side, such as the SCSI controller, must be able to operate as an initiator.
- The disk controller operates as a target. It carries out the commands it receives from the initiator.
- The initiator establishes a logical connection with the intended target. Once this connection has been established, it can be suspended and restored as needed to transfer commands and bursts of data.
- While a particular connection is suspended, other device can use the bus to transfer information.
- This ability to overlap data transfer requests is one of the key features of the SCSI bus that leads to its high performance.
- Data transfers on the SCSI bus are always controlled by the target controller. To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it. Then the controller starts a data transfer operation to receive a command from the initiator. The processor sends a command to the SCSI controller, which causes the following sequence of event to take place:
 - The SCSI controller, acting as an initiator, contends for control of the bus.
 - When the initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.
 - The target starts an output operation (from initiator to target); in response to this, the initiator sends a command specifying the required read operation.
 - The target, realizing that it first needs to perform a disk seek operation, sends a message to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
 - The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read operation. Then, it reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data to the initiator, the target requests control

of the bus. After it wins arbitration, it reselects the initiator controller, thus restoring the suspended connection.

→ The target transfers the contents of the data buffer to the initiator and then suspends the connection again. Data are transferred either 8 or 16 bits in parallel, depending on the width of the bus.

→ The target controller sends a command to the disk drive to perform another seek operation. Then, it transfers the contents of the second disk sector to the initiator as before. At the end of this transfers, the logical connection between the two controllers is terminated.

→ As the initiator controller receives the data, it stores them into the main memory using the DMA approach.

→ The SCSI controller sends an interrupt to the processor to inform it that the requested operation has been completed.

→ This scenario shows that the messages exchanged over the SCSI bus are at a higher level than those exchanged over the processor bus. In this context, a “higher level” means that the messages refer to operations that may require several steps to complete, depending on the device. Neither the processor nor the SCSI controller need be aware of the details of operation of the particular device involved in a data transfer. In the preceding example, the processor need not be involved in the disk seek operation.

- The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read operation.

- The target transfers the contents of the data buffer to the initiator and then suspends the connection again.

- The target controller sends a command to the disk drive to perform another seek operation.

- As the initiator controller receives the data, it stores them into the main memory using the DMA approach.

- The SCSI controller sends an interrupt to the processor to inform it that the requested operation has been completed. The bus signals, arbitration, selection, information transfer and reselection are the topics discussed in addition to the above.

Universal Serial Bus (USB) :

The USB has been designed to meet several key objectives such as:

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

Port Limitation

Here to add new ports, a user must open the computer box to gain access to the internal expansion bus and install a new interface card. The user may also need to know how to configure the device and the software. And also it is to make it possible to add many devices to a computer system at any time, without opening the computer box.

Device Characteristics

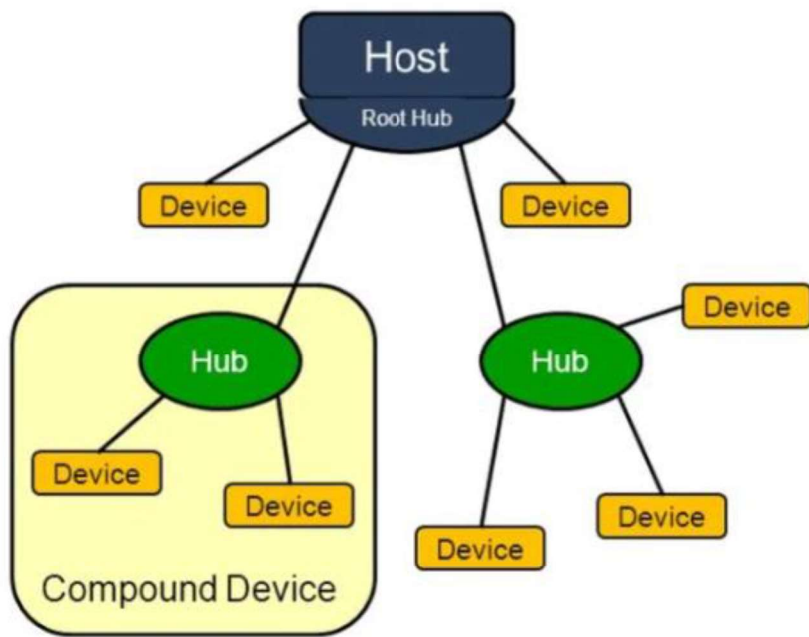
The kinds of devices that may be connected to a computer cover a wide range of functionality - speed, volume and timing constraints. A variety of simple devices attached to a computer generate data in different asynchronous mode. A signal must be sampled quickly enough to track its highest-frequency components.

Plug-and-play

Whenever a device is introduced, do not turn the computer off/restart to connect/disconnect a device. The system should detect the existence of this new device automatically, identify the appropriate device-driver software and any other facilities needed to service that device, and establish the appropriate addresses and logical connections to enable them to communicate.

USB architecture

To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure. Each node has a device called a hub. Root hub, functions, split bus operations – high speed (HS) and Full/Low speed (F/LS).



UNIT-IV

Memory System Concepts:

The maximum size of the memory that can be used in any computer is determined by the addressing scheme. For example, a computer that generates 16-bit addresses is capable of addressing up to $2^{16} = 64K$ memory locations. Similarly, machines with 32-bit addresses can utilize a memory up to $2^{32} = 4G$ locations. The number of locations represents the size of the address space of the computer. Most modern computers are byte-addressable. The memory is usually designed to store and retrieve data in word-length quantities. Data transfer between the memory and the processor takes place through the use of two processor registers, usually called MAR (memory address register) and MDR (memory data register). If MAR is k bits long and MDR is n bits long, then the memory unit may contain up to 2^k addressable locations. During a memory cycle, n bits of data are transferred between the memory and the processor. This transfer takes place over the processor bus, which has k address lines and n data lines. The bus also includes the control lines Read/Write' (R/W') and Memory Function Completed (MFC) for coordinating data transfers. The connection between the processor and the memory

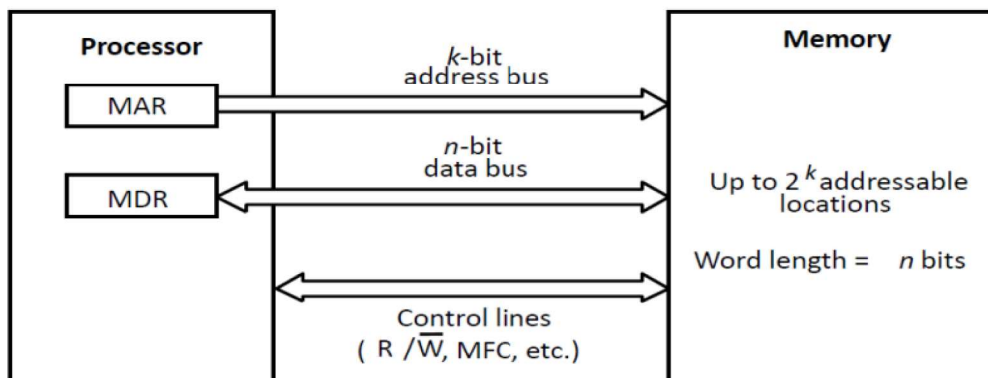


Figure 5.1 Connection of the memory to the processor.

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

The processor writes data into a memory location by loading the address of this location into MAR and loading the data into MDR. The R/W' line is set to 0. If read or write operations involve consecutive address locations in the main memory, then a "block transfer" operation can be performed.

Memory accesses may be synchronized using a clock, or they may be controlled using special signals that control transfers on the bus.

Memory access time: Time that elapses between the initiation of an operation and the completion of that operation.

Memory cycle time: Minimum time delay required between the initiations of two successive memory operations.

Random Access Memory (RAM): A memory unit in which any location can be accessed for a Read or Write operation in some fixed amount of time that is independent of the location's address.

Cache Memory: Is a small, fast memory that is inserted between the larger, slower main memory and the processor. It holds the currently active segments of a program and their data. It reduces memory access time.

Virtual memory: Data is stored in physical memory locations that have addresses different from those specified by the program. An address generated by the processor is referred to as a virtual or logical address. The virtual address space is mapped onto the physical memory where data are actually stored. The mapping function is implemented by a special memory control circuit, often called the memory management unit. Virtual memory is used to increase the apparent size of the physical memory.

SEMI CONDUCTOR RAM MEMORIES:

Internal Organization of Memory Chips: Each memory cell can hold one bit of information. Memory cells are organized in the form of an array. One row is one memory word. All cells of a row are connected to a common line, known as the word line. Word line is connected to the address decoder. The cells in each column are connected to a Sense/Write circuit by two bit lines. The Sense/Write circuits are connected to the data input/output lines of the chip.

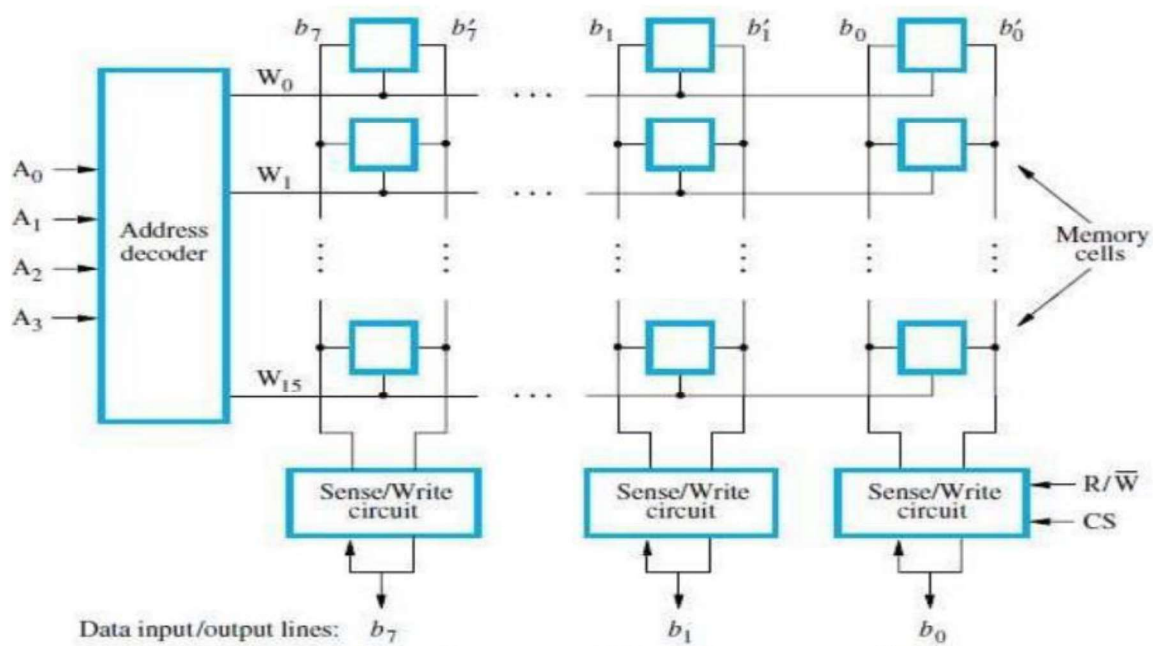
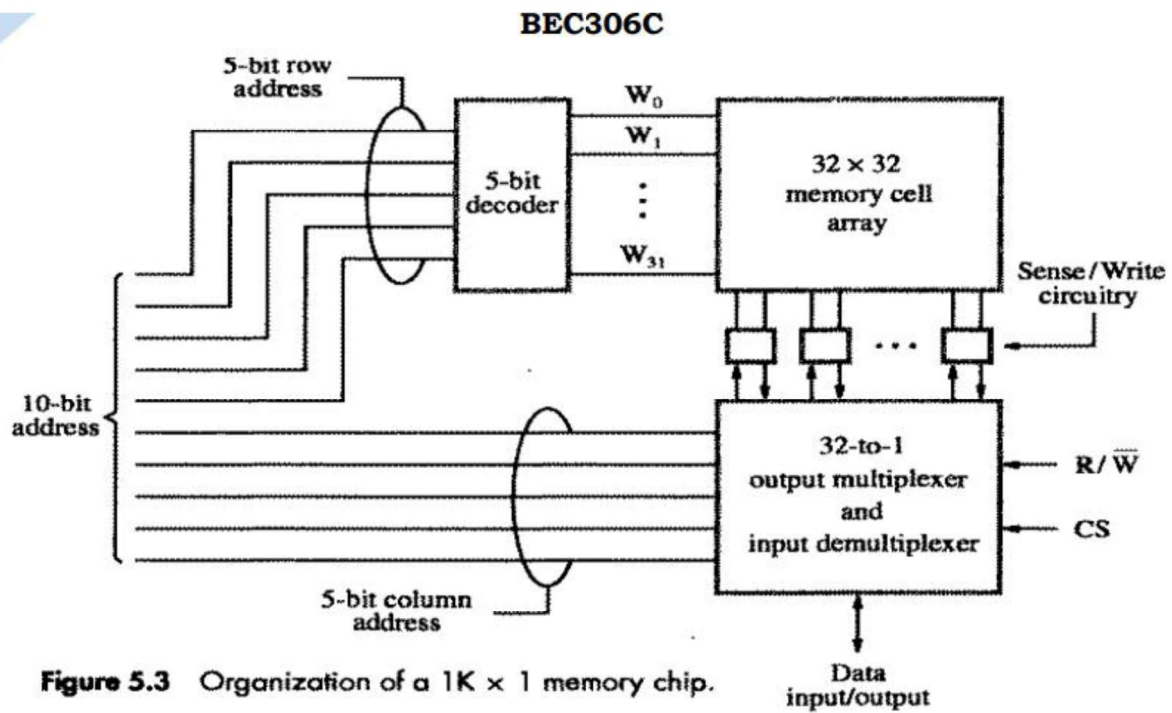


Figure 8.2 Organization of bit cells in a memory chip.

Figure 8.2 is 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, $R/\bar{W}$ and CS, are provided. The $R/\bar{W}$ (Read/Write') input specifies the required operation, and the CS (Chip Select) input selects a given chip in a multichip memory system.

The memory circuit in Figure 8.2 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. If the circuit 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 $1K \times 1$ format. In this case, a 10 bit address is needed, but there is only one data line, resulting in 15 external connections.

Figure 5.3 shows such an organization. The required 10-bit address is divided into two groups of 5 bits each to form the row & column addresses for the cell array. A row address selects a row of 32 cells, all of which are accessed in parallel. But, only one of these cells is connected to external data line, based on column address.



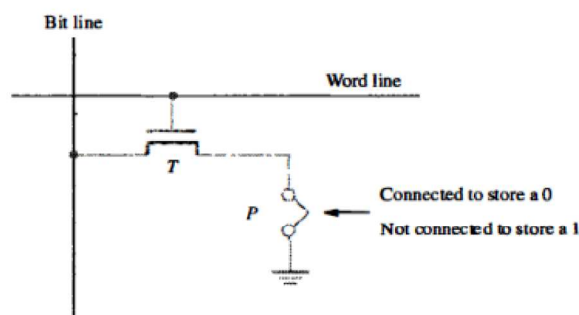
READ ONLY MEMORIES (ROMs) :

Introduction:

Both SRAM and DRAM chips are volatile: i.e. they lose the contents when the power is turned off. Many applications need memory devices to retain the stored information if power is turned off. For example, when the computer is turned on, the operating system must be loaded from the disk into the memory. This requires non-volatile memory. Non-volatile memory is used extensively in embedded systems. Normal operation involves only reading of data. This type of memory is called Read only memory (ROM).

Read Only Memory (ROM):

Figure 5.12 shows a possible configuration for a ROM cell. A logic value 0 is stored in the cell if the transistor is connected to ground at point P; otherwise, a 1 is stored. The bit line is connected through a resistor to the power supply



To read the state of the cell, the word line is activated. Thus, the transistor switch is closed and the voltage on the bit line drops to near zero if there is a connection between the transistor and ground. If there is no connection to ground, the bit line remains at the high voltage, indicating a 1. A sense circuit at the end of the bit line generates the proper output value. Data are written into a ROM when it is manufactured.

Programmable Read Only Memory (PROM): Allows the data to be loaded by a user. Programmability is achieved by inserting a fuse at point P in Figure 5.12. Before it is programmed, the memory contains all 0s. The user can insert 1's at the required locations by burning out the fuses at these locations using high current pulses. This process is irreversible. PROMs provide flexibility & convenience. PROMs provide a faster & less expensive approach because they can be programmed directly by user.

Erasable Programmable Read Only Memory (EPROM): Allows the stored data to be erased and new data to be loaded. It provides considerable flexibility during the development phase of digital systems. They can be used in place of ROMs while software is being developed. Memory changes and updates can be easily made. An EPROM cell has a structure similar to the ROM cell in Figure 5.12. The connection to ground is always made at point P and a special transistor is used, which has the ability to function either as a normal transistor or as a disabled transistor that is always turned off.

Advantage: Their contents can be erased and reprogrammed. Erasure is done by exposing the chip to ultraviolet (UV) light. EPROM chips are mounted in packages that have transparent windows.

Disadvantage: Chip must be physically removed from the circuit for reprogramming and that its entire contents are erased by the ultraviolet light.

Electrically Erasable Programmable Read Only Memory (EEPROM): They can be both programmed and erased electrically. They do not have to be removed for erasure. It is possible to erase the cell contents selectively. The only disadvantage of EEPROMs is that different voltages are needed for erasing, writing, and reading the stored data.

Flash Memory: A flash cell is based on a single transistor controlled by trapped charge, just like an EEPROM cell. In a flash device, it is possible to read the contents of a single cell, but it is only possible to write an entire block of cells. Flash devices have greater density, which leads to higher capacity and a lower cost per bit. They require a single power supply voltage, and consume less power in their operation.

Applications: Used in portable equipment that is battery driven. Hand held computers, cell phones, digital cameras, MP3 music players, etc. Single flash chips do not provide sufficient storage capacity

for the applications mentioned above. Larger memory modules consisting of a number of chips are needed. There are two popular choices for the implementation of larger memory modules: Flash cards & Flash drives.

Flash Cards: Flash chips are mounted on a small card. Such flash cards have a standard interface that makes them usable in a variety of products. A card is simply plugged into a conveniently accessible slot. Flash cards come in a variety of memory sizes. Typical sizes are 8, 32, and 64 Mbytes. A minute of music can be stored in about 1M byte of memory, using the MP3 encoding format. Hence, a 64 MB flash card can store an hour of music.

Flash Drives: Flash drives are designed to fully emulate the hard disks. The storage capacity of flash drives is significantly lower.

Advantages:

- i) Flash drives are solid state electronic devices.
- ii) They have shorter seek and access times, which results in faster response.
- iii) They have lower power consumption, which makes them attractive for battery driven applications, and they are also insensitive to vibration.

Disadvantages:

Smaller capacity & higher cost per bit, compared to hard disk drives. Flash memory will deteriorate after it has been written a number of times (Typically one million times).

CACHE MEMORIES:

Processor is much faster than the main memory. As a result, the processor has to spend much of its time waiting while instructions and data are being fetched from the main memory. This is the major obstacle towards achieving good performance. The speed of the main memory cannot be increased beyond a certain point. Cache memory is an architectural arrangement which makes the main memory appear faster to the processor than it really is. Cache memory is based on the property of computer programs known as “locality of reference”.

Many instructions in localized areas of the program are executed repeatedly during some time period, and the remainder of the program is accessed relatively infrequently. This is called “locality of reference”. There are two types:

- Temporal locality of reference: Recently executed instruction is likely to be executed again very soon.
- Spatial locality of reference: Instructions with addresses close to a recently executed instruction are likely to be executed soon.

If the active segments of a program can be placed in a fast cache memory, then the total execution time can be reduced significantly. The memory control circuitry is designed to take advantage of the property of locality of reference. The temporal aspect of the locality of reference suggests that whenever an information is first needed, this item should be brought into the cache. The spatial aspect suggests that instead of fetching just one item from the main memory to the cache, it is useful to fetch several items that reside at adjacent addresses as well. Block refers to a set of contiguous address locations of some size. Cache line refers to a cache block.

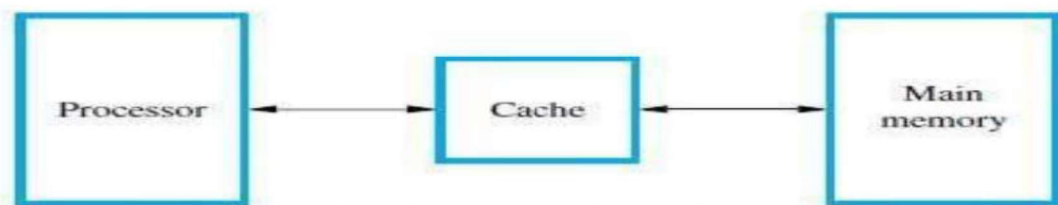


Figure 8.15 Use of a cache memory.

Operation of Cache Memories:

Processor issues a Read request, a block of words is transferred from the main memory to the cache, one word at a time. At any given time, only some blocks in the main memory are held in the cache which is determined by a “mapping function”. When the cache is full, and a block of words needs to be transferred from the main memory, some block of words in the cache must be replaced, which is determined by a “replacement algorithm”.

Cache Hit:

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 hit or write hit is said to have occurred.

In a Read operation, the main memory is not involved. Data is obtained from the cache. For a Write operation, the system can proceed in two ways. In first scheme, write through protocol, the cache location and the main memory location are updated simultaneously. In second scheme, write back or

copy back protocol, only the cache location is updated and it is marked as updated with an associated flag bit, called the dirty or modified bit. The main memory location of the word is updated later.

Cache Miss:

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

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

VIRTUAL MEMORIES :

Virtual memory is an architectural solution to increase the effective size of the memory system. 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. 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. All parts of a program that are eventually executed are first brought into the 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 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. If a virtual address refers to a part of the program or data space that is currently in the physical memory, then the contents of the appropriate location in the main memory are accessed immediately. If the referenced address is not in the main memory, its contents must be brought into a suitable location in the memory before they can be used.

Figure 8.24 shows a typical implementation of 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. 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.

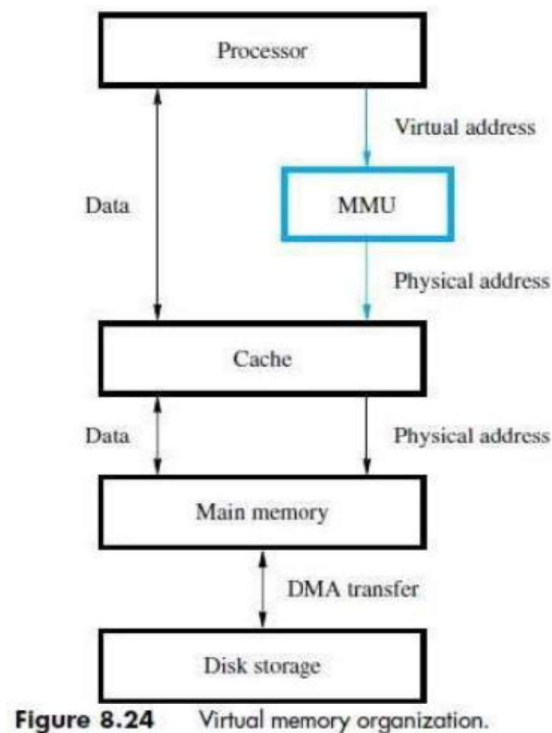


Figure 8.24 Virtual memory organization.

PERFORMANCE CONSIDERATION:

Two Key factors in the commercial success are the performance and cost where the best possible performance is at low cost. A common measure of success is called the Price Performance ratio.

Performance depends on how fast the machine instructions are brought to the processor and how fast they are executed. To achieve parallelism (i.e., both the slow and fast units are accessed in the same manner) interleaving is used.

Interleaving:

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

Two methods of address layout are indicated in Figure 3.5. In the first case, memory address generated by the processor is decoded as shown in part (a) of the figure. The high-order k bits name one of n modules and the low-order m bits name a particular word in that module. When consecutive locations are accessed, only one module is involved. At the same time, devices with DMA ability may be accessing information in other modules.

In the second case, as shown in part (b) of the figure, which is called memory interleaving. The low-order k bits of the memory address select a module, and the high-order m bits name a location within the module. Thus, any component of the system that generates requests for access to consecutive memory locations can keep several modules busy at any one time which results in both faster access to a block of data and higher average utilization of the memory system as a whole.

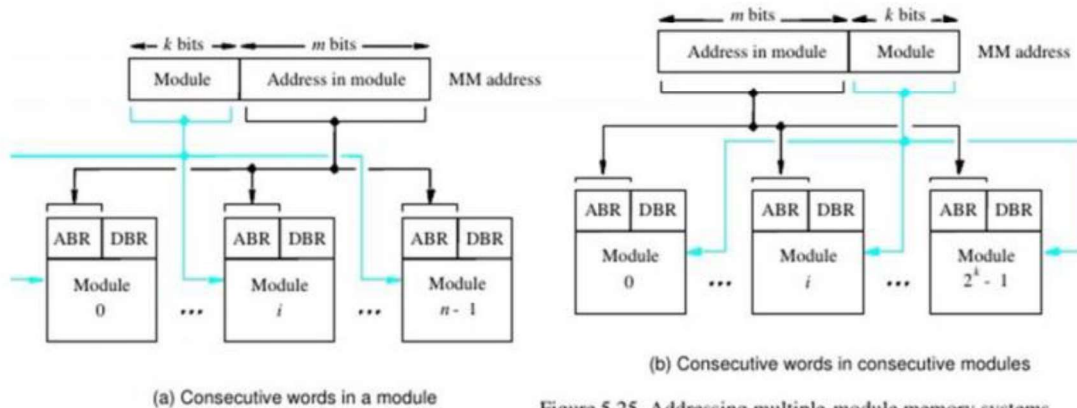


Figure 5.25. Addressing multiple-module memory systems.

HIT RATE AND MISS PENALTY:

An excellent indicator of the effectiveness of a particular implementation of the memory hierarchy is the success rate in accessing information at various levels of the hierarchy. Recall that a successful access to data in a cache is called a hit. The number of hits stated as a fraction of all attempted accesses is called the hit rate, and the miss rate is the number of misses stated as a fraction of attempted accesses. Ideally, the entire memory hierarchy would appear to the CPU as a single memory unit that has the access time of a cache on the CPU chip and the size of a magnetic disk. How close we get to this ideal depends largely on the hit rate at different levels of the hierarchy. High hit rates, over 0.9, are essential for high performance computers. Performance is adversely affected by the actions that must be taken after a miss. The extra time needed to bring the desired information into the cache is called the Miss penalty. This penalty is ultimately reflected in the time that the CPU is stalled because the required instructions or data are not available for execution. In general, the miss penalty is the time needed to bring a block of data from a slower unit in the memory hierarchy to a faster unit. The miss penalty is reduced if efficient mechanisms for transferring data between the various units of the hierarchy are implemented. The previous section shows how an interleaved memory can reduce the miss penalty substantially. Consider now the impact of the cache on the overall performance of the computer. Let h be the hit rate, M the miss penalty, that is, the time to access information in the main memory, and C the time to access information in the cache. The average access time experienced by the CPU is $hC + (1 - h)M$.

CACHES ON PROCESSING CHIPS:

When information is transferred between different chips, considerable delays are introduced in driver and receiver gates on the chips. Thus, from the speed point of view, the optimal place for a cache is on the CPU chip. Unfortunately, space on the CPU chip is needed for many other functions: this limits the size of the cache that can be accommodated. All high performance processor chips include some form of a cache. Some manufacturers have chosen to implement two separate caches, one for instructions and another for data, as in the 68040 and PowerPC 604 processors. Others have implemented a single cache for both instructions and data, as in the PowerPC 601 processor. A combined cache for instructions and data is likely to have a somewhat better hit rate, because it offers greater flexibility in mapping new information into the cache. However, if separate caches are used, it is possible to access both caches at the same time, which leads to increased parallelism and, hence, better performance. The disadvantage of separate caches is that the increased parallelism comes at the expense of more complex circuitry. Since the size of a cache on the CPU chip is limited by space constraints, a good strategy for designing a high performance system is to use such a cache as a primary cache. An external secondary cache, constructed with SRAM chips, is

When added to provide the desired capacity. If both primary and secondary caches are used, the primary cache should be designed to allow very fast access by the CPU, because its access time will have a large effect on the clock rate of the CPU. A cache cannot be accessed at the same speed as a register file, because the cache is much bigger and hence more complex. A practical way to speed up access in the cache is to access more than one word simultaneously and then let the CPU use them one at a time. The secondary cache can be considerably slower, but it should be much larger to ensure a high hit rate. Its speed is less critical, because it only affects the miss penalty of the primary cache. A workstation computer may include a primary cache with the capacity of tens of kilobytes and a secondary cache of several megabytes.

OTHER ENHANCEMENTS:

In addition to the main design issues just discussed, several other possibilities exist for enhancing performance. We discuss three of them in this section.

Write Buffer :

When the write-through protocol is used, each write operation results in writing a new value into the main memory. If the CPU must wait for the memory function to be completed, as we have assumed until now, then the CPU is slowed down by all write requests. Yet the CPU typically does not immediately depend on the result of a write operation, so it is not necessary for the CPU to wait for the write request to be completed. To improve performance, a write buffer can be included for temporary storage of write requests. The CPU places each write request into this buffer and continues

execution of the next instruction. The write requests stored in the write buffer are sent to the main memory whenever the memory is not responding to read requests. Note that it is important that the read requests be serviced immediately, because the CPU usually cannot proceed without the data that is to be read from the memory. Hence, these requests are given priority over write requests. The write buffer may hold a number of write requests. Thus, it is possible that a subsequent read request may refer to data that are still in the write buffer. To ensure correct operation, the addresses of data to be read

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

Pre-fetching:

In the previous discussion of the cache mechanism, we assumed that new data are brought into the cache when they are first needed. A read miss occurs, and the desired data are loaded from the main memory. The CPU has to pause until the new data arrive, which is the effect of the miss penalty. To avoid stalling the CPU, it is possible to prefetch the data into the cache before they are needed. The simplest way to do this is through software. A special prefetch instruction may be provided in the instruction set of the processor. Executing this instruction causes the addressed data to be loaded into the cache, as in the case of a read miss. However, the processor does not wait for the referenced data. A prefetch instruction is inserted in a program to cause the data to be loaded in the cache by the time they are needed in the program. The hope is that prefetching will take place while the CPU is busy executing instructions that do not result in a read miss, thus allowing accesses to the main memory to be overlapped with computation in the CPU. Prefetch instructions can be inserted into a program either by the programmer or by the compiler. It is obviously preferable to have the compiler insert these instructions, which can be done with good success for many applications. Note that software prefetching entails a certain overhead; because inclusion of prefetch instructions increases the length of programs. Moreover, some prefetches may load into the cache data that will not be used by the

instruction. This can happen if the prefetched data are ejected from the cache by a read miss involving other data. However, the overall effect of software prefetching on performance is positive, and many processors (including the PowerPC) have machine instructions to support this feature. Prefetching can also be done through hardware. This involves adding circuitry that attempts to discover a pattern in memory references, and then prefetches data according to this pattern.

Lockup-Free:

Cache the software prefetching scheme just discussed does not work well if it interferes significantly with the normal execution of instructions. This is the case if the action of prefetching stops other accesses to the cache until the prefetch is completed. A cache of this type is said to be locked while it services a miss. We can solve this problem by modifying the basic cache structure to allow the CPU to access the cache while a miss is being serviced. In fact, it is desirable that more than one outstanding miss can be supported. A cache that can support multiple outstanding misses is called lockup-free. Since it can service only one miss at a time, it must include circuitry that keeps track of all outstanding misses. This may be done with special registers that hold the pertinent information about these misses. Lockup-free caches were first used in the early 1980s in the Cyber series of computers manufactured by Control Data Company.

We have used software prefetching as an obvious motivation for a cache that is not locked by a read miss. A much more important reason is that, in a processor that uses a pipelined organization, which overlaps the execution of several instructions, a read miss caused by one instruction could stall the execution of other instructions. A lockup-free cache reduces the likelihood of such stalling.

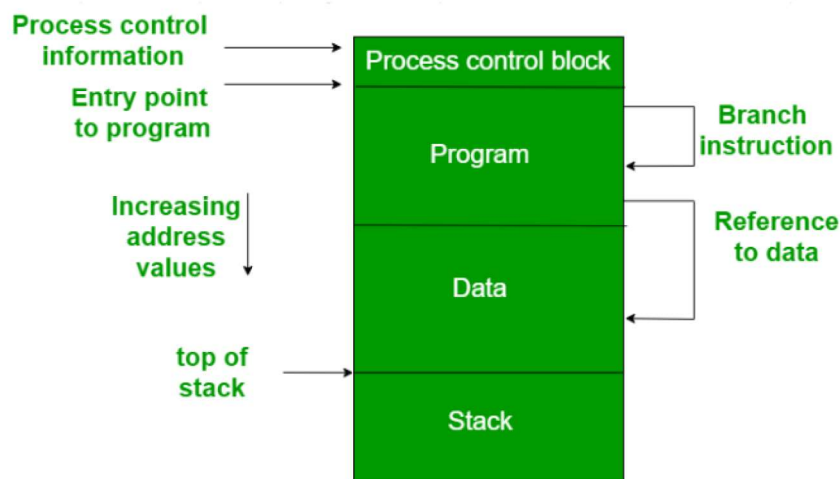
Memory Management Requirements:

Memory management keeps track of the status of each memory location, whether it is allocated or free. It allocates the memory dynamically to the programs at their request and frees it for reuse when it is no longer needed. Memory management meant to satisfy some requirements that we should keep in mind. These Requirements of memory management are:

Relocation :

The available memory is generally shared among a number of processes in a multiprogramming system, so it is not possible to know in advance which other programs will be resident in main memory at the time of execution of this program. Swapping the active processes in and out of the main memory enables the operating system to have a larger pool of ready-to-execute process.

When a program gets swapped out to a disk memory, then it is not always possible that when it is swapped back into main memory then it occupies the previous memory location, since the location may still be occupied by another process. We may need to **relocate** the process to a different area of memory. Thus there is a possibility that program may be moved in main memory due to swapping.



The figure depicts a process image. The process image is occupying a continuous region of main memory. The operating system will need to know many things including the location of process control information, the execution stack, and the code entry. Within a program, there are memory references in various instructions and these are called logical addresses. After loading of the program into main memory, the processor and the operating system must be able to translate logical addresses into physical addresses. Branch instructions contain the address of the next instruction to be executed. Data reference instructions contain the address of byte or word of data referenced.

Protection

There is always a danger when we have multiple programs at the same time as one program may write to the address space of another program. So every process must be protected against unwanted interference when other process tries to write in a process whether accidental or incidental. Between relocation and protection requirement a trade-off occurs as the satisfaction of relocation requirement increases the difficulty of satisfying the protection requirement. Prediction of the location of a program in main memory is not possible, that's why it is impossible to check the absolute address at compile time to assure protection. Most of the programming language allows the dynamic calculation of address at run time. The memory protection requirement must be satisfied by the processor rather than the operating system because the operating system can hardly

control a process when it occupies the processor. Thus it is possible to check the validity of memory references.

Sharing

A protection mechanism must have to allow several processes to access the same portion of main memory. Allowing each processes access to the same copy of the program rather than have their own separate copy has an advantage. For example, multiple processes may use the same system file and it is natural to load one copy of the file in main memory and let it shared by those processes. It is the task of Memory management to allow controlled access to the shared areas of memory without compromising the protection. Mechanisms are used to support relocation supported sharing capabilities.

Logical organization

Main memory is organized as linear or it can be a one-dimensional address space which consists of a sequence of bytes or words. Most of the programs can be organized into modules, some of those are unmodifiable (read-only, execute only) and some of those contain data that can be modified. To effectively deal with a user program, the operating system and computer hardware must support a basic module to provide the required protection and sharing. It has the following advantages:

Modules are written and compiled independently and all the references from one module to another module are resolved by the system at run time.

Different modules are provided with different degrees of protection.

There are mechanisms by which modules can be shared among processes. Sharing can be provided on a module level that lets the user specify the sharing that is desired.

Physical organization - The structure of computer memory has two levels referred to as main memory and secondary memory. Main memory is relatively very fast and costly as compared to the secondary memory. Main memory is volatile. Thus secondary memory is provided for storage of data on a long-term basis while the main memory holds currently used programs. The major system concern between main memory and secondary memory is the flow of information and it is impractical for programmers to understand this for two reasons: The programmer may engage in a practice known as overlaying when the main memory available for a program and its data may be insufficient. It allows different modules to be assigned to the same region of memory. One disadvantage is that it is time-consuming for the programmer.

In a multiprogramming environment, the programmer does not know how much space will be available at the time of coding and where that space will be located inside the memory.

Arithmetic: - Addition and subtraction of sign members:

Data is manipulated by using the arithmetic instructions in digital computers. Data is manipulated to produce results necessary to give solution for the computation problems. The Addition, subtraction, multiplication and division are the four basic arithmetic operations. If we want then we can derive other operations by using these four operations.

To execute arithmetic operations there is a separate section called arithmetic processing unit in central processing unit. The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation.

If we want to solve a problem then we use a sequence of well-defined steps. These steps are collectively called algorithm. To solve various problems we give algorithms.

In order to solve the computational problems, arithmetic instructions are used in digital computers that manipulate data. These instructions perform arithmetic calculations.

And these instructions perform a great activity in processing data in a digital computer. As we already stated that with the four basic arithmetic operations addition, subtraction, multiplication and division, it is possible to derive other arithmetic operations and solve scientific problems by means of numerical analysis methods.

A processor has an arithmetic processor(as a sub part of it) that executes arithmetic operations. The data type, assumed to reside in processor, registers during the execution of an arithmetic instruction. Negative numbers may be in a signed magnitude or signed complement representation. There are three ways of representing negative fixed point - binary numbers signed magnitude, signed 1's complement or signed 2's complement. Most computers use the signed magnitude representation for the mantissa.

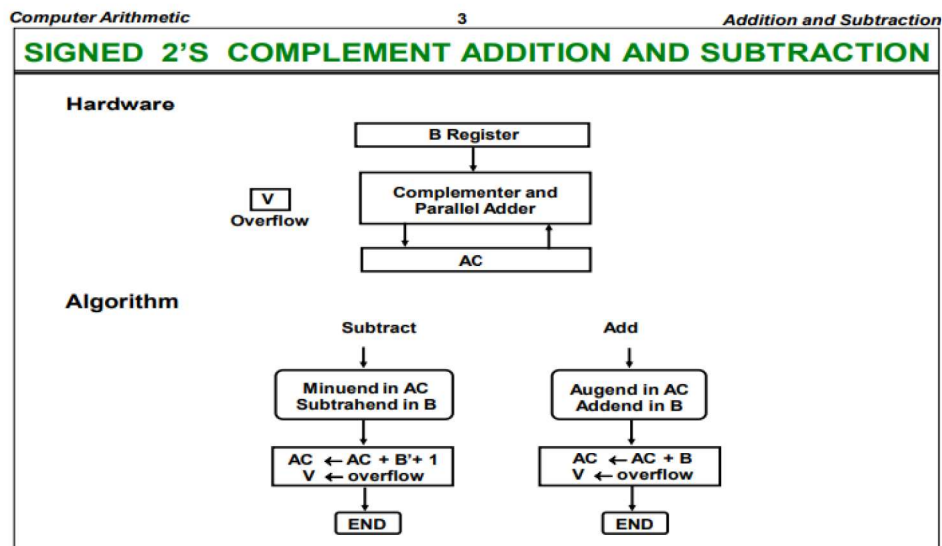
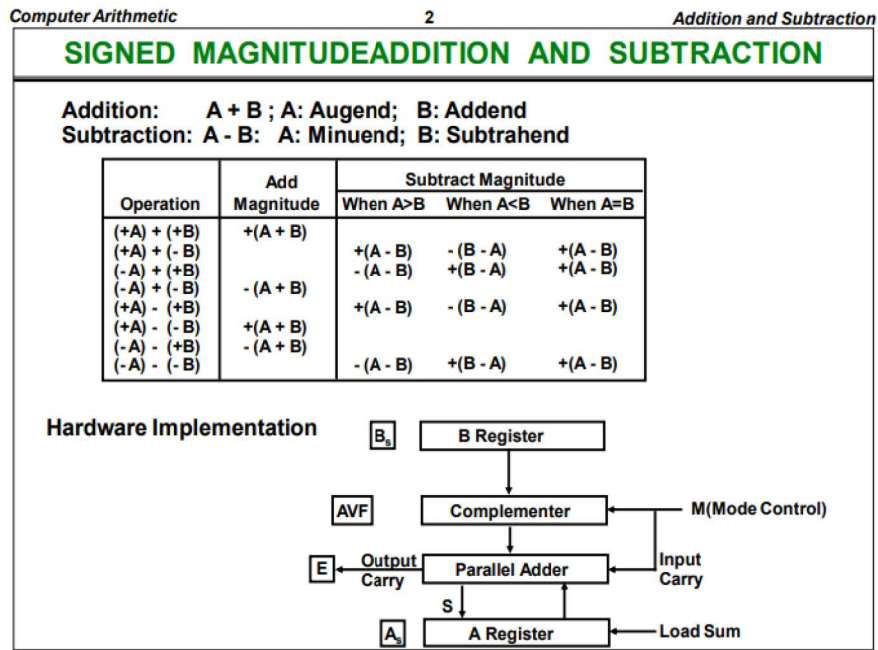
Addition and Subtraction :

Addition and Subtraction with Signed –Magnitude Data We designate the magnitude of the two numbers by A and B. Where the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table 4.1. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to

present a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words parentheses should be used for the subtraction algorithm)

Addition and Subtraction of Signed-Magnitude Numbers



Algorithm:

The flowchart is shown in Figure 7.1. The two signs A, and B, are compared by an exclusive-OR gate.

If the output of the gate is 0 the signs are identical; If it is 1, the signs are different.

For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added.

The magnitudes are added with a microoperation $E \leftarrow A + B$, where E is a register that combines E and A . The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF .

The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complemented B . No overflow can occur if the numbers are subtracted so AVF is cleared to 0.

1 in E indicates that $A \geq B$ and the number in A is the correct result. If this number is zero, the sign of A must be made positive to avoid a negative zero.

0 in E indicates that $A < B$. For this case it is necessary to take the 2's complement of the value in A . The operation can be done with one microoperation $A \leftarrow A' + 1$.

However, we assume that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two microoperations.

In other paths of the flowchart, the sign of the result is the same as the sign of A , so no change in A is required. However, when $A < B$, the sign of the result is the complement of the original sign of A . It is then necessary to complement A , to obtain the correct sign.

The final result is found in register A and its sign in A_s . The value in AVF provides an overflow indication. The final value of E is immaterial.

Figure 7.2 shows a block diagram of the hardware for implementing the addition and subtraction operations.

It consists of registers A and B and sign flip-flops A_s and B_s . Subtraction is done by adding A to the 2's complement of B .

The output carry is transferred to flip-flop E , where it can be checked to determine the relative magnitudes of two numbers.

The add-overflow flip-flop AVF holds the overflow bit when A and B are added.

The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

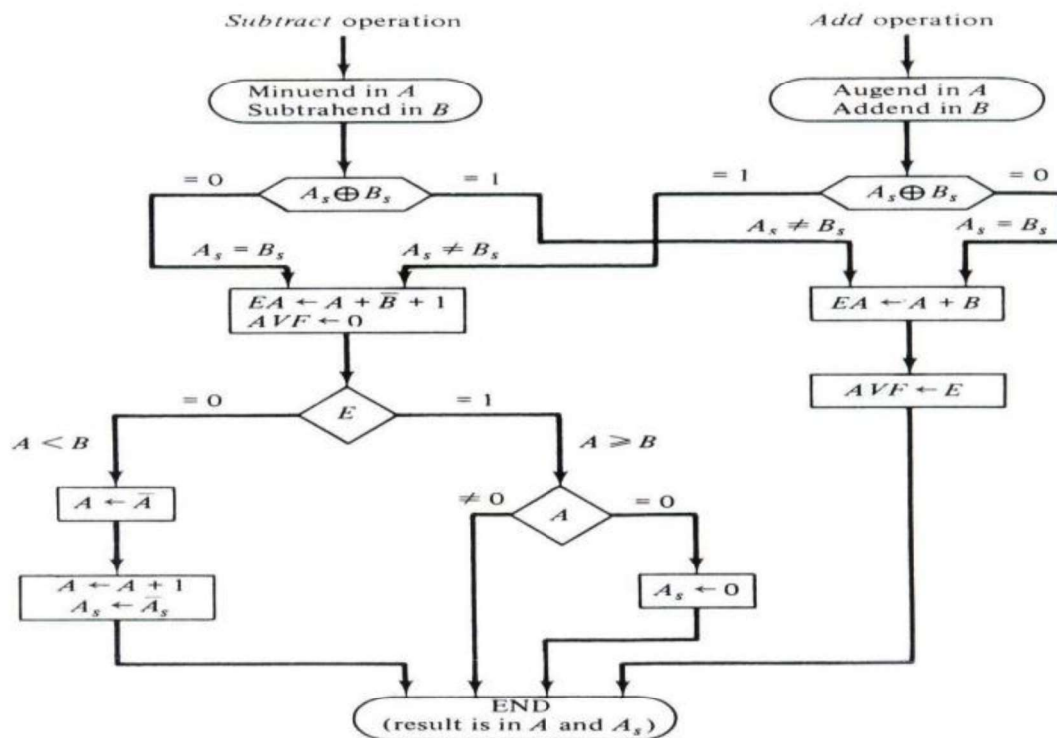


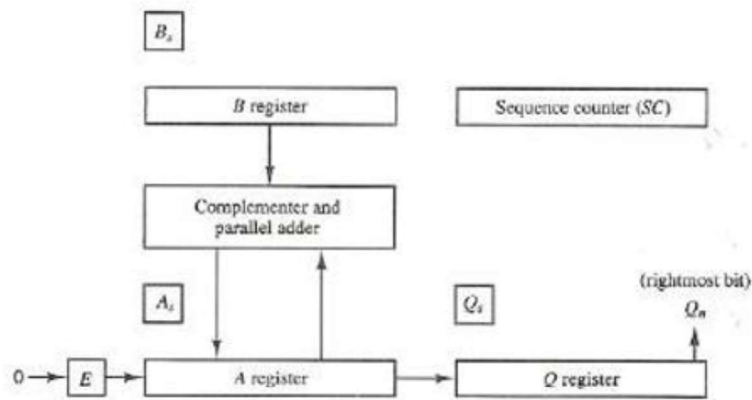
Figure 10-2 Flowchart for add and subtract operations.

Multiplication:

Hardware Implementation and Algorithm:

Generally, the multiplication of two final point binary number in signed magnitude representation is performed by a process of successive shift and ADD operation. The process consists of looking at the successive bits of the multiplier (least significant bit first). If the multiplier is 1, then the multiplicand is copied down otherwise, 0's are copied. The numbers copied down in successive lines are shifted one position to the left and finally, all the numbers are added to get the product.

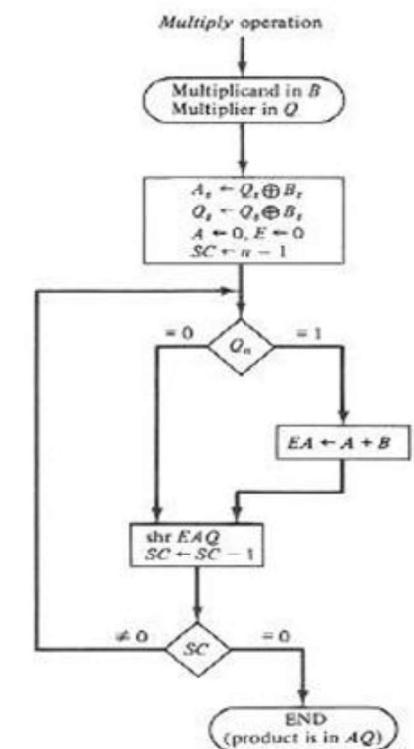
But, in digital computers, an adder for the summation (Σ) of only two binary numbers are used and the partial product is accumulated in register. Similarly, instead of shifting the multiplicand to the left, the partial product is shifted to the right. The hardware for the multiplication of signed magnitude data is shown in the figure below.



Hardware for multiply operation

Initially, the multiplier is stored q register and the multiplicand in the B register. A register is used to store the partial product and the sequence counter (SC) is set to a number equal to the number of bits in the multiplier. The sum of A and B form the partial product and both shifted to the right using a statement “Shr EAQ” as shown in the hardware algorithm. The flip flops As, Bs & Qs store the sign of A, B & Q respectively. A binary ‘0’ inserted into the flip-flop E during the shift right.

Hardware Algorithm



flowchart for multiply algorithm

Example: Multiply 23 by 19 using multiply algorithm.

multiplier	E	A	Q	SC
Initially,	0	00000	10011	101(5)
Iteration1(Qn=1), add B first partial product shrEAQ,	0	00000 +10111 10111		
	0	01011	11001	100(4)
Iteration2(Qn=1) Add B Second partial product shrEAQ,	1	01011 +10111 00010	11001	
	0	10001	01100	011(3)
Iteration3(Qn=0) shrEAQ,	0	01000	10110	010(2)
Iteration4(Qn=0) shrEAQ,	0	00100	01011	001(1)
Iteration5(Qn=1) Add B Fifth partial product shrEAQ,	0	00100 +10111 11011	01011	
	0	01101	10101	000
FinalProductinAQ	0110110101			

The final product is in register A & Q. therefore, the product is 0110110101.

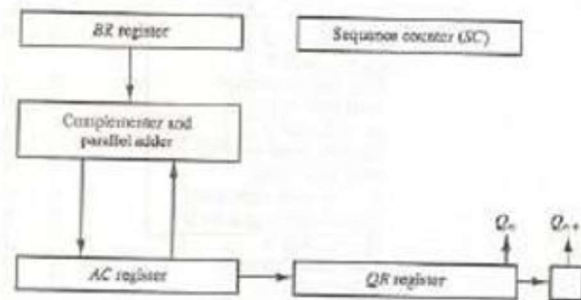
Booth Algorithm:

The algorithm that is used to multiply binary integers in signed 2's complement form is called booth multiplication algorithm. It works on the principle that the string 0's in the multiplier doesn't need addition but just the shifting and the string of 1's from bit weight 2^k to 2^m can be treated as $2^{k+1} - 2^m$ (Example, $+14 = 001110 = 23 - 21 = 14$). The product can be obtained by shifting the binary multiplication to the left and subtraction the multiplier shifted left once.

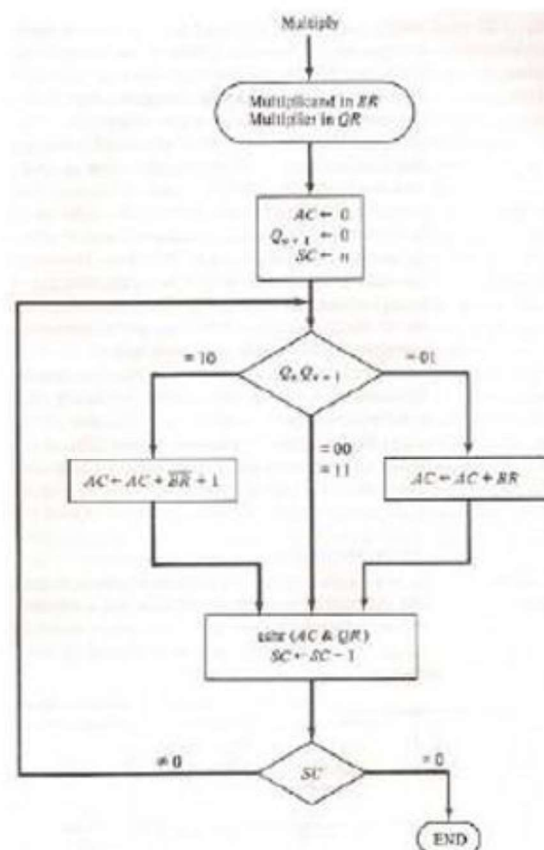
According to booth algorithm, the rule for multiplication of binary integers in signed 2's complement form are:

- The multiplicand is subtracted from the partial product of the first least significant bit is 1 in a string of 1's in the multiplier.
- The multiplicand is added to the partial product if the first least significant bit is 0 (provided that there was a previous 1) in a string of 0's in the multiplier.
- The partial product doesn't change when the multiplier bit is identical to the previous multiplier bit.

This algorithm is used for both the positive and negative numbers in signed 2's complement form. The hardware implementation of this algorithm is in figure below:



The flowchart for booth multiplication algorithm is given below:



flowchart for booth multiplication algorithm

Numerical Example: Booth algorithm

BR=10111(Multiplicand)

QR=10011(Multiplier)

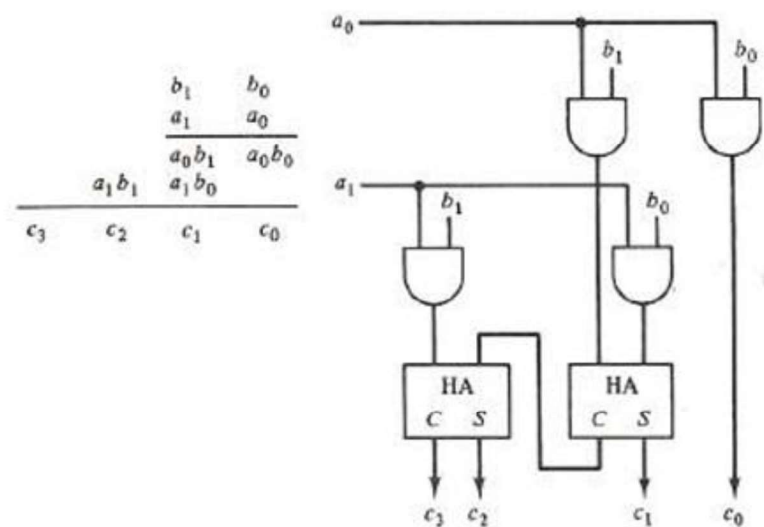
Array Multiplier :

The multiplication algorithm first check the bits of the multiplier one at time and form partial product. This is a sequential process that requires a sequence of add and shift micro operation. This method is

complicated and time consuming. The multiplication of 2 binary numbers can also be done with one microoperation by using combinational circuit that provides the product all at once.

Example.

Consider that the multiplicand bits are b_1 and b_0 and the multiplier bits are a_1 and a_0 . The partial product is $c_3c_2c_1c_0$. The multiplication two bits a_0 and a_1 produces a binary 1 if both the bits are 1, otherwise it produces a binary 0. This is identical to the AND operation and can be implemented with the AND gates as shown in figure.



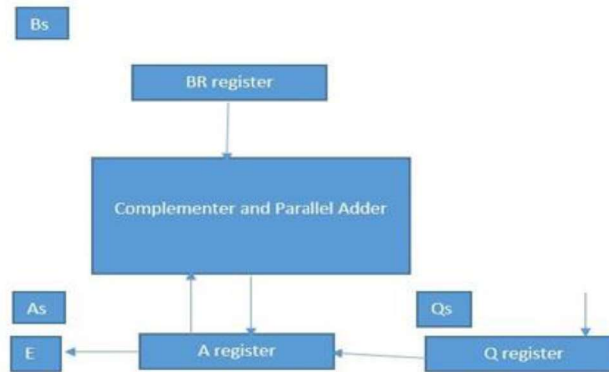
2-bit by 2-bit array multiplier

Division Algorithm :

The division of two fixed point signed numbers can be done by a process of successive compare shift and subtraction. When it is implemented in digital computers, instead of shifting the divisor to the right, the dividend or the partial remainder is shifted to the left. The subtraction can be obtained by adding the number A to the 2's complement of number B. The information about the relative magnitudes of the information about the relative magnitudes of numbers can be obtained from the end carry,

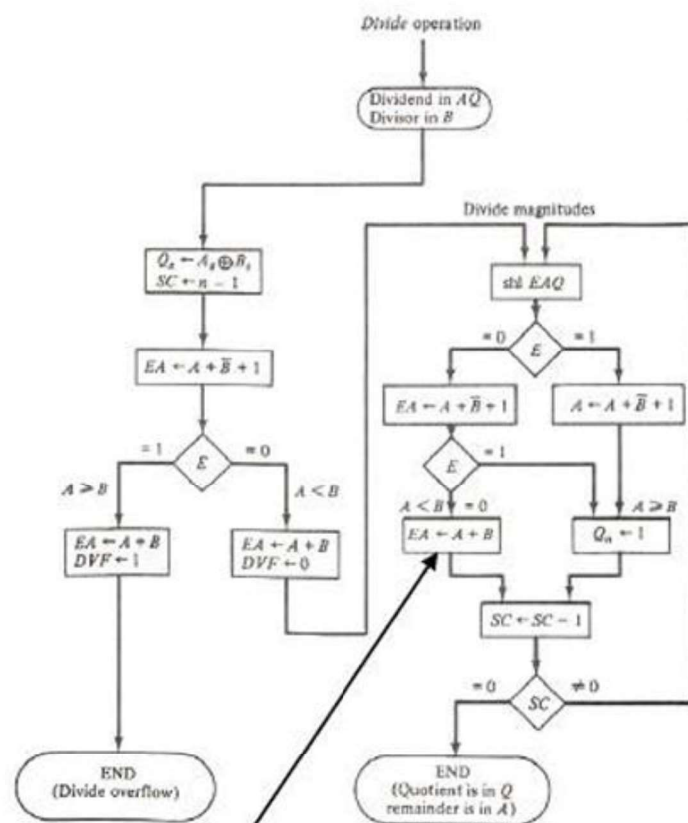
Hardware Implementation

The hardware implementation for the division signed numbers is shown id the figure.



Division Algorithm :

The divisor is stored in register B and a double length dividend is stored in register A and Q. the dividend is shifted to the left and the divider is subtracted by adding twice complement of the value. If $E = 1$, then $A \geq B$. In this case, a quotient bit 1 is inserted into Q_n and the partial remainder is shifted to the left to repeat the process. If $E = 0$, then $A > B$. In this case, the quotient bit Q_n remains zero and the value of B is added to restore the partial remainder in A to the previous value. The partial remainder is shifted to the left and approach continues until the sequence counter reaches to 0. The registers E, A & Q are shifted to the left with 0 inserted into Q_n and the previous value of E is lost as shown in the flow chart for division algorithm.



flowchart for division algorithm

This algorithm can be explained with the help of an example.

Consider that the divisor is 10001 and the dividend is 01110.

Divisor $B = 10001$,		$\overline{B} + 1 = 01111$		
	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A :		00110		
Quotient in Q :			11010	

binary division with digital hardware

Restoring method:

Method described above is restoring method in which partial remainder is restored by adding the divisor to the negative result. Other methods:

Comparison method: A and B are compared prior to subtraction. Then if $A \geq B$, B is subtracted from A. if $A < B$ nothing is done. The partial remainder is then shifted left and numbers are compared again. Comparison inspects end carry out of the parallel adder before transferring to E.

Non-restoring method:

In contrast to restoring method, when $A - B$ is negative, B is not added to restore A but instead, negative difference is shifted left and then B is added. How is it possible? Let's argue:

- In flowchart for restoring method, when $A < B$, we restore A by operation $A - B + B$. Next time in a loop, this number is shifted left (multiplied by 2) and B subtracted again, which gives: $2(A - B + B) - B = 2A - B$.
- In Non-restoring method, we leave $A - B$ as it is. Next time around the loop, the number is shifted left and B is added: $2(A - B) + B = 2A - B$ (same as above).

Divide Overflow :

The division algorithm may produce a quotient overflow called dividend overflow. The overflow can occur if the number of bits in the quotient are more than the storage capacity of the register. The overflow flip-flop DVF is set to 1 if the overflow occurs.

The division overflow can occur if the value of the half most significant bits of the dividend is equal to or greater than the value of the divisor. Similarly, the overflow can occur if the dividend is divided by a 0. The overflow may cause an error in the result or sometimes it may stop the operation. When the overflow stops the operation of the system, then it is called divide stop.

Arithmetic Operations on Floating-Point Numbers :

The rules apply to the single-precision IEEE standard format. These rules specify only the major steps needed to perform the four operations. Intermediate results for both mantissas and exponents might require more than 24 and 8 bits, respectively & overflow or an underflow may occur. These and other aspects of the operations must be carefully considered in designing an arithmetic unit that meets the standard. If their exponents differ, the mantissas of floating-point numbers must be shifted with respect to each other before they are added or subtracted. Consider a decimal example in which we wish to add 2.9400×10^1 to 4.3100×10^1 . We rewrite 2.9400×10^1 as 0.294×10^2 and then perform addition of the mantissas to get 4.3394×10^1 . The rule for addition and subtraction can be stated as follows:

Add/Subtract Rule :

The steps in addition (FA) or subtraction (FS) of floating-point numbers (s_1, e_1, f_1) and (s_2, e_2, f_2) are as follows.

1. Unpack sign, exponent, and fraction fields. Handle special operands such as zero, infinity, or NaN(not a number).
2. Shift the significand of the number with the smaller exponent right by bits.
3. Set the result exponent e_r to $\max(e_1, e_2)$.

4. If the instruction is FA and $s1 = s2$ or if the instruction is FS and $s1 \neq s2$ then add the significands; otherwise subtract them.
5. Count the number z of leading zeros. A carry can make $z = -1$. Shift the result significand left z bits or right 1 bit if $z = -1$.
6. Round the result significand, and shift right and adjust z if there is rounding overflow, which is a carry-out of the leftmost digit upon rounding.
7. Adjust the result exponent by $er = er - z$, check for overflow or underflow, and pack the result sign, biased exponent, and fraction bits into the result word.

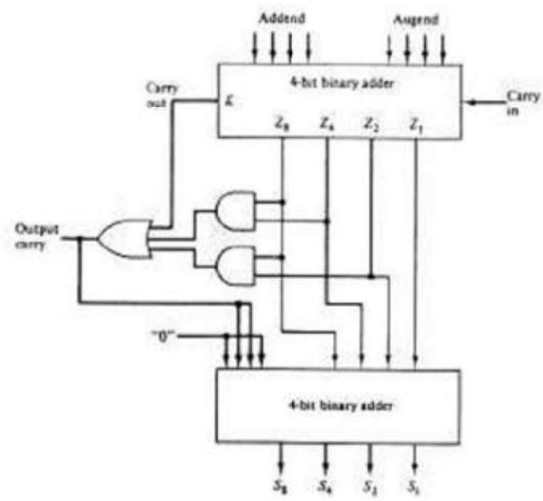
Operands	Alignment	Normalize and round
6.144×10^2	0.06144×10^4	1.003644×10^5
$+ 9.975 \times 10^4$	$+ 9.975 \times 10^4$	$+ .0005 \times 10^5$
-----	10.03644×10^4	$\frac{1.004 \times 10^5}{}$

Operands	Alignment	Normalize and round
1.076×10^{-7}	1.076×10^{-7}	7.7300×10^{-9}
$- 9.987 \times 10^{-8}$	$- 0.9987 \times 10^{-7}$	$+ .0005 \times 10^{-9}$
-----	0.0773×10^{-7}	$\frac{7.730 \times 10^{-9}}{}$

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

BCD Adder:

BCD adder A 4-bit binary adder that is capable of adding two 4-bit words having a BCD (binary-coded decimal) format. The result of the addition is a BCD-format 4-bit output word, representing the decimal sum of the addend and augend, and a carry that is generated if this sum exceeds a decimal value of 9. Decimal addition is thus possible using these devices.



UNIT-V

Basic Processing Unit:

Concepts:

To execute a program, the processor fetches one instruction at a time and performs the operations specified. Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.

The processor keeps track of the address of the memory location containing the next instruction to be fetched using the program counter, PC. After fetching an instruction, the contents of the PC are updated to point to the next instruction in the sequence. A branch instruction may load a different value into the PC. Another key register in the processor is the instruction register, IR.

Suppose that each instruction comprises 4 bytes, and that it is stored in one memory word. To execute an instruction, the processor has to perform the following three *steps:

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are the instruction to be executed; hence they are loaded into the IR. In register transfer notation, the required action is $IR \leftarrow [PC]$
2. Increment the PC to point to the next instruction. Assuming that the memory is byte addressable, the PC is incremented by 4; that is $PC \leftarrow [PC] + 4$
3. Carry out the operation specified by the instruction in the IR.

Fetching an instruction and loading it into the IR is usually referred to as the instruction fetch phase. Performing the operation specified in the instruction constitutes the instruction execution phase.

Execution of a complete instruction:

Consider the instruction $Add(R3), R1$ which adds the contents of a memory location pointed to by R3 to register R 1.

Executing this instruction requires the following actions:

1. Fetch the instruction.
2. Fetch the first operand (the contents of the memory location pointed to by R3).
3. Perform the addition. 4. Load the result into R1.

Instruction execution proceeds as follows.

Step 1: The instruction fetch operation is initiated by loading the contents of the PC into the MAR and sending a Read request to the memory. The Select signal is set to Select4, which causes the multiplexer MUX to select the constant 4. This value is added to the operand at input B, which is the contents of the PC, and the result is stored in register Z.

Step 2: The updated value is moved from register Z back into the PC, while waiting for the memory to respond.

Step 3: The word fetched from the memory is loaded into the IR. (Steps 1 through 3 constitute the instruction fetch phase, which is the same for all instructions.)

Step 4: The instruction decoding circuit interprets the contents of the IR. This enables the control circuitry to activate the control signals for steps 4 through 7, which constitute the execution phase. The contents of register R3 are transferred to the MAR in step 4, and a memory read operation is initiated.

Step 5: the contents of R1 are transferred to register Y, to prepare for the addition operation.

Step 6: When the Read operation is completed, the memory operand is available in register MDR, and the addition operation is performed. The contents of MDR are gated to the bus, and thus also to the B input of the ALU, and register Y is selected as the second input to the ALU by choosing SelectY.

Step 7: The sum is stored in register Z, and then transferred to R1. The End signal causes a new instruction fetch cycle to begin by returning to step 1.

This discussion accounts for all control signals in Figure 11 except Yin in step 2. There is no need to copy the updated contents of PC into register Y when executing the Add instruction.

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

Figure 11

But, in Branch instructions the updated value of the PC is needed to compute the Branch target address.

To speed up the execution of Branch instructions, this value is copied into register Y in step 2. Since step 2 is part of the fetch phase, the same action will be performed for all instructions. This does not cause any harm because register Y is not used for any other purpose at that time.

Branch Instruction :

A branch instruction replaces the contents of the PC with the branch target address. This address is usually obtained by adding an offset X, which is given in the branch instruction, to the updated value of the PC. Figure 12 gives a control sequence that implements an unconditional branch instruction. Processing starts, as usual, with the fetch phase. This phase ends when the instruction is loaded into the IR in step 3.

The offset value is extracted from the IR by the instruction decoding circuit, which will also perform sign extension if required. Since the value of the updated PC is already available in register Y, the offset X is gated onto the bus in step 4, and an addition operation is performed. The result, which is the branch target address, is loaded into the PC in step 5.

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

For example: if the branch instruction is at location 2000 and if the branch target address is 2050, the value of X must be 46. The PC is incremented during the fetch phase before knowing the type of the instruction being executed. Thus, when the branch address is computed in step 4, the PC value uses the updated value, which points to the instruction following the branch instruction in the memory.

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

Figure 12

Multiple Bus organization:

We used the simple single-bus structure to illustrate the basic ideas. The resulting control sequences are quite long because only one data item can be transferred over the bus in a clock cycle.

To reduce the number of steps needed, most commercial processors provide multiple internal paths that enable several transfers to take place in parallel. Figure depicts a three-bus structure used to connect the registers and the ALU of a processor.

The register file is said to have three ports. There are two outputs, allowing the contents of two different registers to be accessed simultaneously and have their contents placed on buses A and B. The third port allows the data on bus C to be loaded into a third register during the same clock cycle.

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

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

Consider the three-operand instruction

Add R4,R5,R6

The control sequence for executing this instruction is given as below

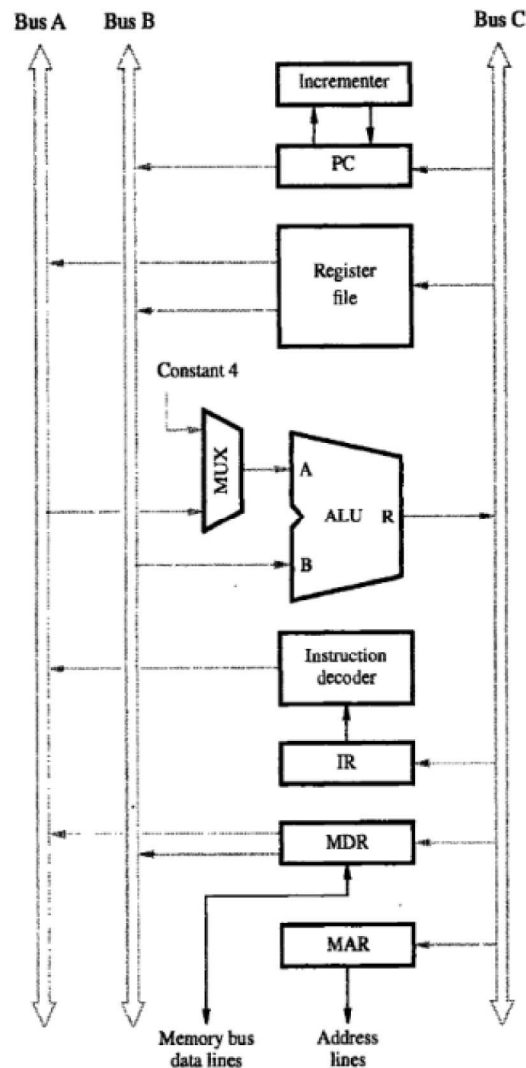
Step	Action
1	$PC_{out}, R=B, MAR_{in}, Read, IncPC$
2	WMFC
3	$MDR_{outB}, R=B, IR_{in}$
4	$R4_{outA}, R5_{outB}, SelectA, Add, R6_{in}, End$

Step 1: the contents of the PC are passed through the ALU, using the $R=B$ control signal, and loaded into the MAR to start a memory read operation. At the same time the PC is incremented by 4. Note that the value loaded into MAR is the original contents of the PC. The incremented value is loaded into the PC at the end of the clock cycle and will not affect the contents of MAR.

Step 2: the processor waits for MFC and loads the data received into MDR.

Step 3: Transfers the data received in MDR to IR. Step 4: The execution phase of the instruction requires only one control step to complete.

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



Hardware control :

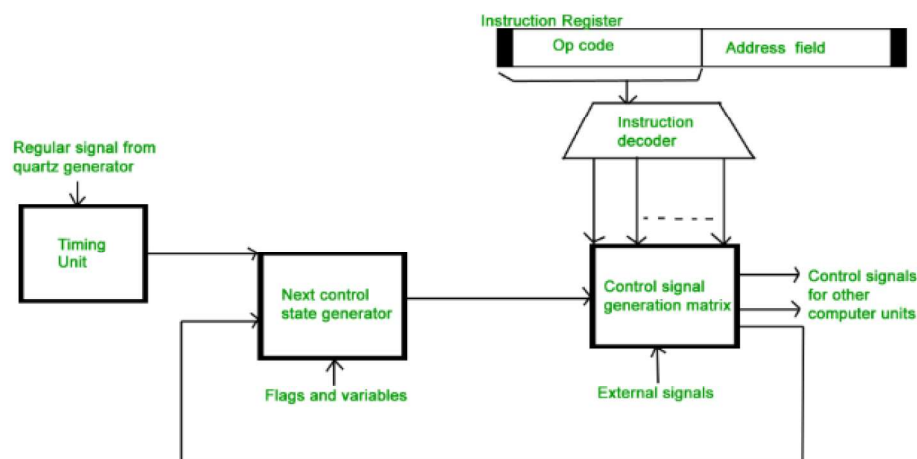
The control unit is the part of the CPU responsible for directing operations. It manages the flow of instructions and data within the processor, ensuring that tasks are carried out in the correct order. There are two main types: hardwired and microprogrammed.

- Hardwired units: Fast but less flexible (because they rely on fixed circuits).
- Microprogrammed units: Flexible but slower (because they use microcode stored in memory).

Hardwired Control Unit

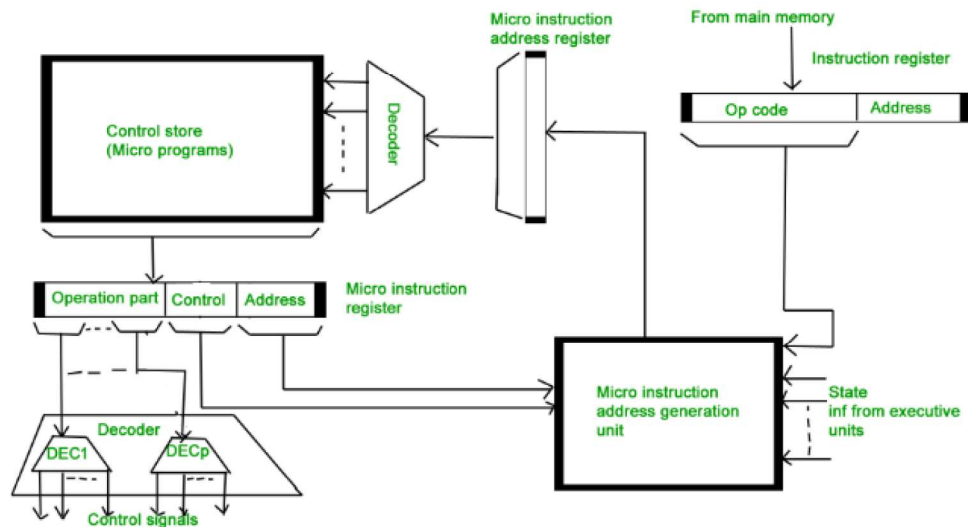
The control hardware can be viewed as a state machine that changes from one state to another in every clock cycle, depending on the contents of the instruction register, the condition codes, and the external inputs. The outputs of the state machine are the control signals. The sequence of the operation carried out by this machine is determined by the wiring of the logic elements and hence is named “hardwired”.

- Fixed logic circuits that correspond directly to the Boolean expressions are used to generate the control signals.
- Hardwired control is faster than microprogrammed control.
- A controller that uses this approach can operate at high speed.
- RISC architecture is based on the hardwired control unit



Micro-programmed Control Unit

- The control signals associated with operations are stored in special memory units inaccessible by the programmer as Control Words.
- Control signals are generated by a program that is similar to machine language programs.
- The micro-programmed control unit is slower in speed because of the time it takes to fetch microinstructions from the control memory.



Some Important Terms

1. **Control Word:** A control word is a word whose individual bits represent various control signals.
2. **Micro-routine:** A sequence of control words corresponding to the control sequence of a machine instruction constitutes the micro-routine for that instruction.
3. **Micro-instruction:** Individual control words in this micro-routine are referred to as microinstructions.
4. **Micro-program:** A sequence of micro-instructions is called a micro-program, which is stored in a ROM or RAM called a Control Memory (CM).
5. **Control Store:** the micro-routines for all instructions in the instruction set of a computer are stored in a special memory called the Control Store.

Types of Micro-programmed Control Unit

Based on the type of Control Word stored in the Control Memory (CM), it is classified into two types :

1. Horizontal Micro-programmed Control Unit :

The control signals are represented in the decoded binary format that is 1 bit/CS. Example: If 53 Control signals are present in the processor then 53 bits are required. More than 1 control signal can be enabled at a time.

- It supports longer control words.

- It is used in parallel processing applications.
- It allows a higher degree of parallelism. If degree is n , n CS is enabled at a time.
- It requires no additional hardware(decoders). It means it is faster than Vertical Microprogrammed.
- It is more flexible than vertical microprogrammed

2. Vertical Micro-programmed Control Unit :

The control signals are represented in the encoded binary format. For N control signals- $\log_2(N)$ bits are required.

- It supports shorter control words.
- It supports easy implementation of new control signals therefore it is more flexible.
- It allows a low degree of parallelism i.e., the degree of parallelism is either 0 or 1.
- Requires additional hardware (decoders) to generate control signals, it implies it is slower than horizontal microprogrammed.
- It is less flexible than horizontal but more flexible than that of a hardwired control unit.

Pipelining :

Concepts:

The speed of execution of programs is influenced by many factors. One way to improve performance is to use faster circuit technology to build the processor and the main memory. Another possibility is to arrange the hardware so that more than one operation can be performed at the same time. In this way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

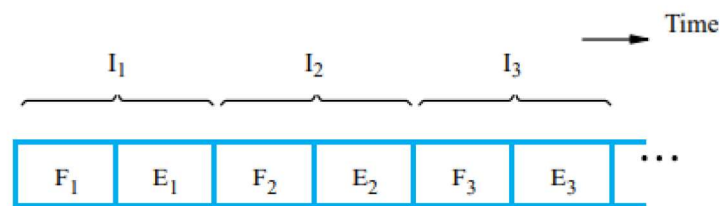
We have encountered concurrent activities several times before. Chapter 1 introduced the concept of multiprogramming and explained how it is possible for I/O transfers and computational activities to proceed simultaneously. DMA devices make this possible because they can perform I/O transfers independently once these transfers are initiated by the processor.

Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly-line operation. Readers are undoubtedly familiar with the assembly line used in car manufacturing. The first station in an assembly line may prepare the chassis of a car, the next station adds the body, the next one installs the engine, and so on. While one group of workers

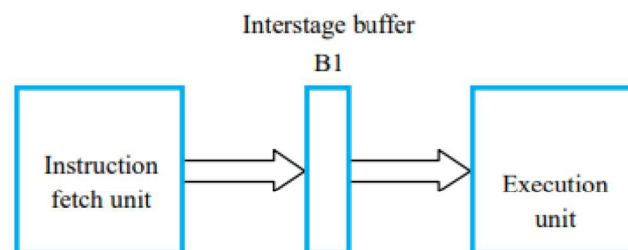
is installing the engine on one car, another group is fitting a car body on the chassis of another car, and yet another group is preparing a new chassis for a third car. It may take days to complete work on a given car, but it is possible to have a new car rolling off the end of the assembly line every few minutes.

Consider how the idea of pipelining can be used in a computer. The processor executes a program by fetching and executing instructions, one after the other. Let F_i and E_i refer to the fetch and execute steps for instruction I_i . Execution of a program consists of a sequence of fetch and execute steps, as shown in Figure 8.1a.

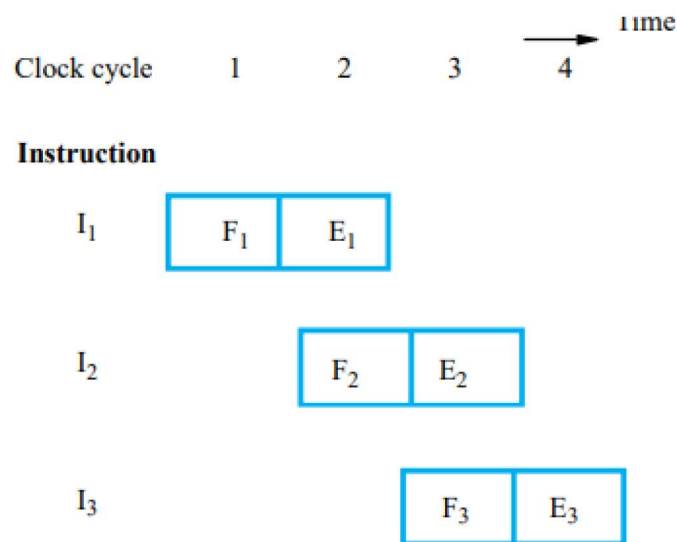
Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure 5.1b. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction. For the purposes of this discussion, we assume that both the source and the destination of the data operated on by the instructions are inside the block labeled “Execution unit”



(a) Sequential execution



(b) Hardware organization



(c) Pipelined execution

By the end of thesecond clock cycle, the execution of instruction I1 is completed and instruction I2 is available. Instruction I2 is stored in B1, replacing I1 , which is no longer needed. Step E2 is performed by the execution unit during the third clock cycle, while instruction I3 is being fetched by the fetch unit. In this manner, both the fetch and execute units are kept busy all the time. If the pattern in Figure 8.1c can be sustained for a long time, the completion rate of instruction execution will be twice that achievable by the sequential operation depicted in Figure 5.1a.

In summary, the fetch and execute units in Figure 5.1b constitute a two-stage pipeline in which each stage performs one step in processing an instruction. An inter- stage storage buffer, B1, is needed to hold the information being passed from one stage to the next. New information is loaded into this buffer at the end of each clock cycle. The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:

F Fetch: read the instruction from the memory.

D Decode: decode the instruction and fetch the source operand(s).

E Execute: perform the operation specified by the instruction.

W Write: store the result in the destination location.

The sequence of events for this case is shown in Figure 8.2a. Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Figure 8.2b. These units must be capable of performing their tasks simultane- ously and without interfering with one another. Information is passed from one unit to the next through a storage buffer. As an instruction

progresses through the pipeline, all the information needed by the stages downstream must be passed along. For example, during clock cycle 4, the information in the buffers is as follows:

- Buffer B1 holds instruction I3 , which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.
- Buffer B2 holds both the source operands for instruction I2 and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction I2 (step W2). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.
- Buffer B3 holds the results produced by the execution unit and the destination information for instruction I1 .

Data Hazards:

A data hazard is a situation in which the pipeline is stalled because the data to be operated on are delayed for some reason, as illustrated in Figure 8.3. We will now examine the issue of availability of data in some detail.

Consider a program that contains two instructions, I1 followed by I2 . When this program is executed in a pipeline, the execution of I2 can begin before the execution of I1 is completed. This means that the results generated by I1 may not be available for use by I2 . We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially. The potential for obtaining incorrect results when operations are performed concurrently can be demonstrated by a simple example. Assume that $A = 5$, and consider the following two operations:

$$A \leftarrow 3 + A \quad B \leftarrow 4 \times A$$

When these operations are performed in the order given, the result is $B = 32$. But if they are performed concurrently, the value of A used in computing B would be the original value, 5, leading to an incorrect result. If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because the data used in the second instruction depend on the result of the first instruction. On the other hand, the two operations

$$A \leftarrow 5 \times C$$

$$B \leftarrow 20 + C$$

can be performed concurrently, because these operations are independent.

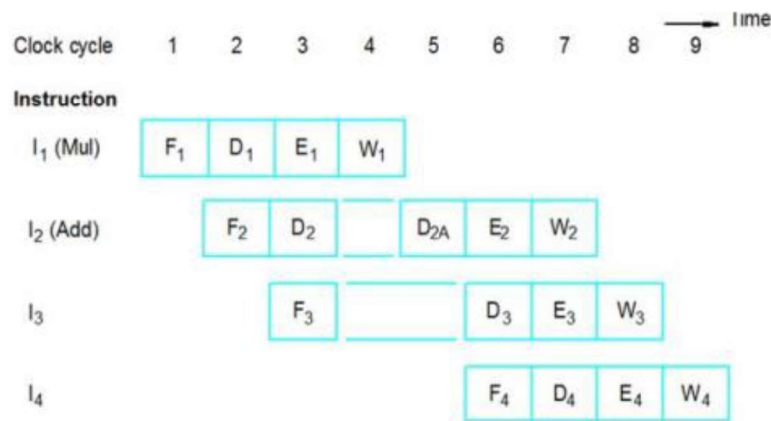


Figure 5.6 Pipeline stalled by data dependency between D₂ and W₁ .

This example illustrates a basic constraint that must be enforced to guarantee correct results. When two operations depend on each other, they must be performed sequentially in the correct order. This rather obvious condition has far-reaching consequences. Understanding its implications is the key to understanding the variety of design alternatives and trade-offs encountered in pipelined computers.

Consider the pipeline in Figure 5.2. The data dependency just described arises when the destination of one instruction is used as a source in the next instruction. For example, the two instructions

MulR2,R3,R4

AddR5,R4,R6

give rise to a data dependency. The result of the multiply instruction is placed into register R4, which in turn is one of the two source operands of the Add instruction. Assuming that the multiply operation takes one clock cycle to complete, execution would proceed as shown in Figure 5.6.

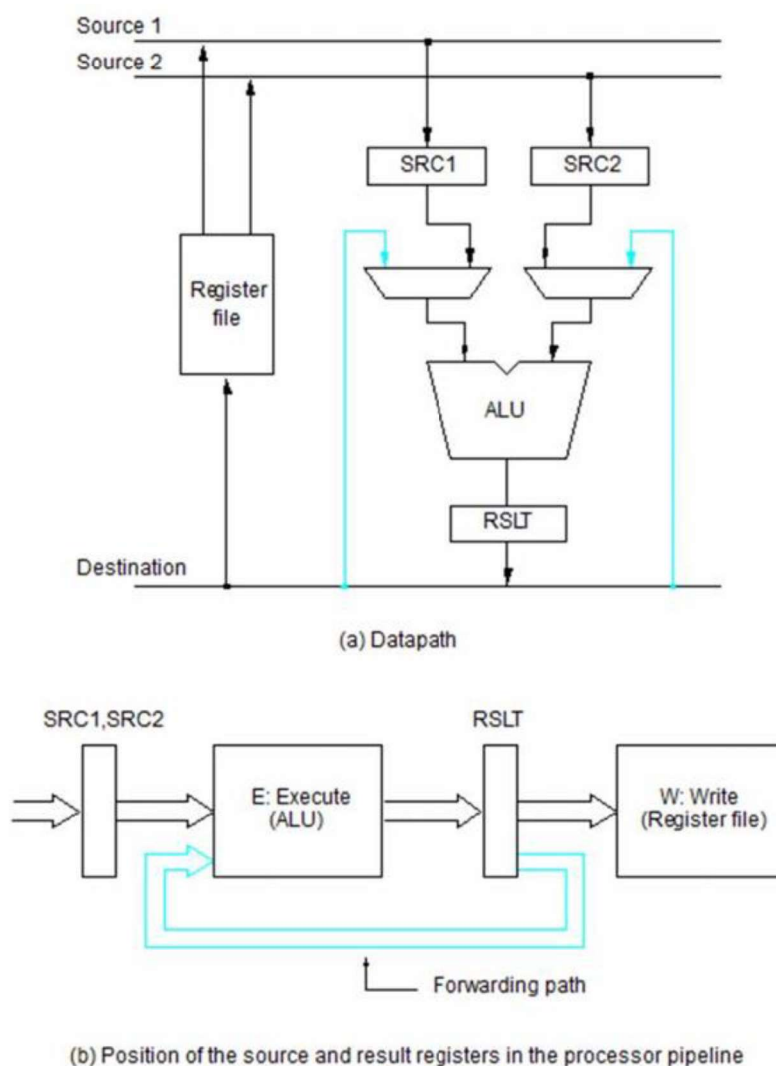
As the Decode unit decodes the Add instruction in cycle 3, it realizes that R4 is used as a source operand. Hence, the D step of that instruction cannot be completed until the W step of the multiply instruction has been completed. Completion of step D₂ must be delayed to clock cycle 5, and is shown as step D_{2A} in the figure. Instruction I₃ is fetched in cycle 3, but its decoding must be delayed because step D₃ cannot precede D₂. Hence, pipelined execution is stalled for two cycles.

Operand Forwarding :

The data hazard just described arises because one instruction, instruction I₂ in Figure 5.6, is waiting for data to be written in the register file. However, these data are available at the output of the ALU once the Execute stage completes step E₁. Hence, the delay can be reduced, or possibly eliminated, if we arrange for the result of instruction I₁ to be forwarded directly for use in step E₂.

Figure 5.7a shows a part of the processor datapath involving the ALU and the register file. This arrangement is similar to the three-bus structure except that registers SRC1, SRC2, and RSLT have been added. These registers constitute the interstage buffers needed for pipelined operation, as illustrated in Figure 8.7b. With reference to Figure 8.2b, registers SRC1 and SRC2 are part of buffer B2 and RSLT is part of B3. The data forwarding mechanism is provided by the blue connection lines. The two multiplexers connected at the inputs to the ALU allow the data on the destination bus to be selected instead of the contents of either the SRC1 or SRC2 register.

When the instructions in Figure 5.6 are executed in the datapath of Figure 5.7, the operations performed in each clock cycle are as follows. After decoding instruction I2 and detecting the data dependency, a decision is made to use data forwarding. The operand not involved in the dependency, register R2, is read and loaded in register SRC1 in clock cycle 3. In the next clock cycle, the product produced by instruction I1 is available in register RSLT, and because of the forwarding connection, it can be used in step E2. Hence, execution of I2 proceeds without interruption.



HANDLING DATA HAZARDS IN SOFTWARE

In Figure 5.6, we assumed the data dependency is discovered by the hardware while the instruction is being decoded. The control hardware delays reading register R4 until cycle 5, thus introducing a 2-cycle stall unless operand forwarding is used. An alternative approach is to leave the task of detecting data dependencies and dealing with them to the software. In this case, the compiler can introduce the two cycle delay needed between instructions I1 and I2 by inserting NOP (No-operation) instructions, as follows:

I1: Mul R2,R3,R4 NOP NOP

I2: Add R5,R4,R6

If the responsibility for detecting such dependencies is left entirely to the software, the compiler must insert the NOP instructions to obtain a correct result. This possibility illustrates the close link between the compiler and the hardware. A particular feature can be either implemented in hardware or left to the compiler. Leaving tasks such as inserting NOP instructions to the compiler leads to simpler hardware. Being aware of the need for a delay, the compiler can attempt to reorder instructions to perform useful tasks in the NOP slots, and thus achieve better performance. On the other hand, the insertion of NOP instructions leads to larger code size. Also, it is often the case that a given processor architecture has several hardware implementations, offering different features. NOP instructions inserted to satisfy the requirements of one implementation may not be needed and, hence, would lead to reduced performance on a different implementation.

Side Effects:

The data dependencies encountered in the preceding examples are explicit and easily detected because the register involved is named as the destination in instruction I1 and as a source in I2. Sometimes an instruction changes the contents of a register other than the one named as the destination. An instruction that uses an auto increment or auto decrement addressing mode is an example. In addition to storing new data in its destination location, the instruction changes the contents of a source register used to access one of its operands. All the precautions needed to handle data dependencies involving the destination location must also be applied to the registers affected by an auto increment or auto decrement operation. When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a side effect. For example, stack instructions, such as push and pop, produce similar side effects because they implicitly use the auto increment and auto decrement addressing modes.

Another possible side effect involves the condition code flags, which are used by instructions such as conditional branches and add-with-carry. Suppose that registers R1 and R2 hold a double-precision integer number that we wish to add to another double-precision number in registers R3 and R4. This may be accomplished as follows:

Add R1,R3

Add With Carry R2,R4

An implicit dependency exists between these two instructions through the carry flag. This flag is set by the first instruction and used in the second instruction, which performs the operation

$R4 \leftarrow [R2] + [R4] + \text{carry}$

Instructions that have side effects give rise to multiple data dependencies, which lead to a substantial increase in the complexity of the hardware or software needed to resolve them. For this reason, instructions designed for execution on pipelined hardware should have few side effects. Ideally, only the contents of the destination location, either a register or a memory location, should be affected by any given instruction. Side effects, such as setting the condition code flags or updating the contents of an address pointer, should be kept to a minimum. However, that the auto increment and auto decrement addressing modes are potentially useful. Condition code flags are also needed for recording such information as the generation of a carry or the occurrence of overflow in an arithmetic operation. In Section 8.4 we show how such functions can be provided by other means that are consistent with a pipelined organization and with the requirements of optimizing compilers.

Instruction Hazards:

The purpose of the instruction fetch unit is to supply the execution units with a steady stream of instructions. Whenever this stream is interrupted, the pipeline stalls, as Figure 5.4 illustrates for the case of a cache miss. A branch instruction may also cause the pipeline to stall. We will now examine the effect of branch instructions and the techniques that can be used for mitigating their impact. We start with unconditional branches

Unconditional Branches:

Figure 5.8 shows a sequence of instructions being executed in a two-stage pipeline. Instructions I1 to I3 are stored at successive memory addresses, and I2 is a branch instruction. Let the branch target be instruction Ik. In clock cycle 3, the fetch operation for instruction I3 is in progress at the same time that the branch instruction is being decoded and the target address computed. In clock cycle 4, the processor must discard I3, which has been incorrectly fetched, and fetch instruction Ik. In the

meantime, the hardware unit responsible for the Execute (E) step must be told to do nothing during that clock period. Thus, the pipeline is stalled for one clock cycle.

The time lost as a result of a branch instruction is often referred to as the branch penalty. In Figure 5.8, the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher. For example, Figure 5.9a shows the effect of a branch instruction on a four-stage pipeline. We have assumed that the branch address is computed in step E2. Instructions I3 and I4 must be discarded, and the target instruction, I_k, is fetched in clock cycle 5. Thus, the branch penalty is two clock cycles.

Reducing the branch penalty requires the branch address to be computed earlier in the pipeline. Typically, the instruction fetch unit has dedicated hardware to identify a branch instruction and compute the branch target address as quickly as possible after an instruction is fetched. With this additional hardware, both of these tasks can be performed in step D2, leading to the sequence of events shown in Figure 5.9b. In this case, the branch penalty is only one clock cycle.

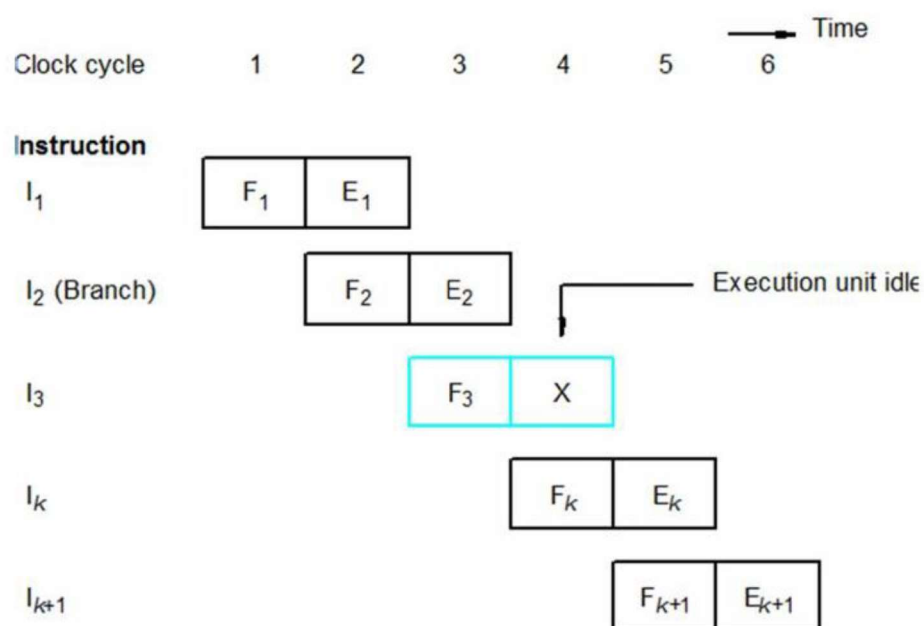
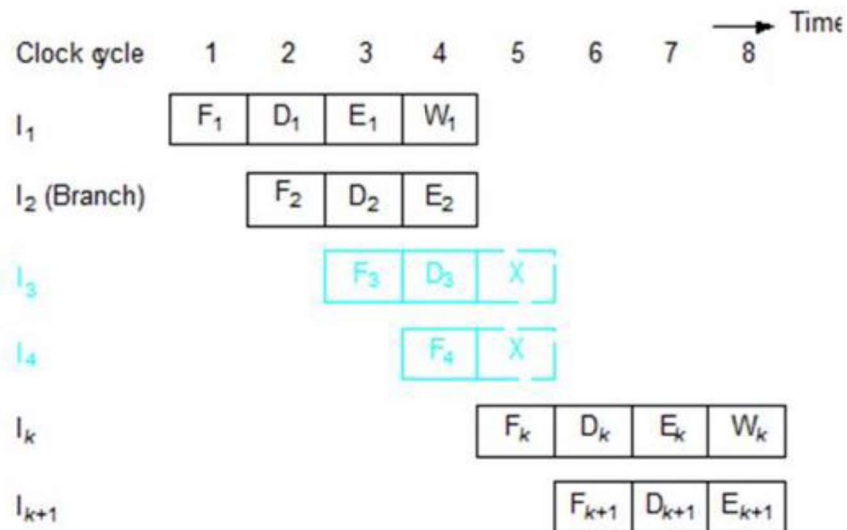
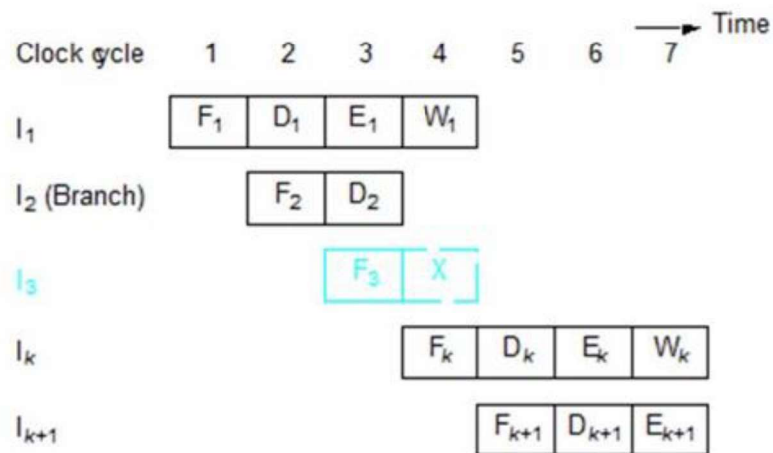


Figure 5.8 An idle cycle caused by a branch instruction.



(a) Branch address computed in Execute stage



(b) Branch address computed in Decode stage

Fig:5.9 Branch timing

Instruction Queue and Prefetching

Either a cache miss or a branch instruction stalls the pipeline for one or more clock cycles.

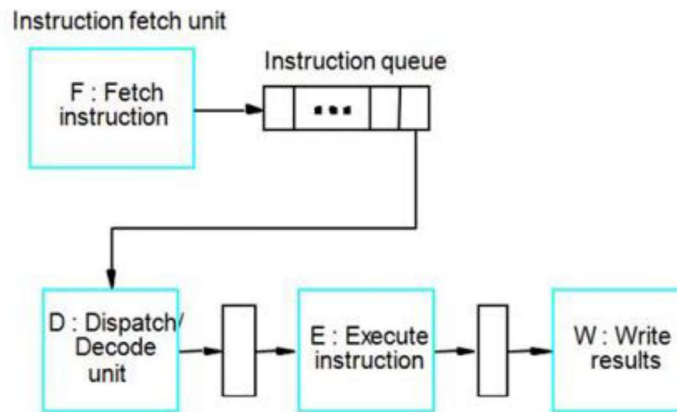


Fig:5.10 Use of an instruction queue in the hardware organization of Figure 5.2b.

To reduce the effect of these interruptions, many processors employ sophisticated fetch units that can fetch instructions before they are needed and put them in a queue.

Typically, the instruction queue can store several instructions. A separate unit, which we call the dispatch unit, takes instructions from the front of the queue and sends them to the execution unit. This leads to the organization shown in Figure 5.10. The dispatch unit also performs the decoding function.

To be effective, the fetch unit must have sufficient decoding and processing capability to recognize and execute branch instructions. It attempts to keep the instruction queue filled at all times to reduce the impact of occasional delays when fetching instructions. When the pipeline stalls because of a data hazard, for example, the dispatch unit is not able to issue instructions from the instruction queue. However, the fetch unit continues to fetch instructions and add them to the queue. Conversely, if there is a delay in fetching instructions because of a branch or a cache miss, the dispatch unit continues to issue instructions from the instruction queue.

We have assumed that initially the queue contains one instruction. Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one. Hence, the queue length remains the same for the first four clock cycles. (There is both an F and a D step in each of these cycles.) Suppose that instruction I1 introduces a 2-cycle stall. Since space is available in the queue, the fetch unit continues to fetch instructions and the queue length rises to 3 in clock cycle 6.

Instruction I5 is a branch instruction. Its target instruction, I_k, is fetched in cycle 7, and instruction I6 is discarded. The branch instruction would normally cause a stall in cycle 7 as a result of discarding instruction I6. Instead, instruction I4 is dispatched from the queue to the decoding stage. After discarding I6, the queue length drops to 1 in cycle 8. The queue length will be at this value until another stall is encountered.

Conditional Branches And Branch Prediction:

A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction. The decision to branch cannot be made until the execution of that instruction has been completed.

Branch instructions occur frequently. In fact, they represent about 20 percent of the dynamic instruction count of most programs. (The dynamic count is the number of instruction executions, taking into account the fact that some program instructions are executed many times because of loops.) Because of the branch penalty, this large percentage would reduce the gain in performance expected from pipelining. Fortunately, branch instructions can be handled in several ways to reduce their negative impact on the rate of execution of instructions.

Delayed Branch:

In Figure 5.8, the processor fetches instruction I3 before it determines whether the current instruction, I2, is a branch instruction. When execution of I2 is completed and a branch is to be made, the processor must discard I3 and fetch the instruction at the branch target. The location following a branch instruction is called a branch delay slot. There may be more than one branch delay slot, depending on the time it takes to execute a branch instruction. For example, there are two branch delay slots in Figure 5.9a and one delay slot in Figure 5.9b. The instructions in the delay slots are always fetched and at least partially executed before the branch decision is made and the branch target address is computed.

A technique called delayed branching can minimize the penalty incurred as a result of conditional branch instructions. The idea is simple. The instructions in the delay slots are always fetched. Therefore, we would like to arrange for them to be fully executed whether or not the branch is taken. The objective is to be able to place useful instructions in these slots. If no useful instructions can be placed in the delay slots, these slots must be filled with NOP instructions. This situation is exactly the same as in the case of data dependency.

Consider the instruction sequence given in Figure 5.12a. Register R2 is used as a counter to determine the number of times the contents of register R1 are shifted left. For a processor with one delay slot, the instructions can be reordered as shown in Figure 5.12b. The shift instruction is fetched while the branch instruction is being executed. After evaluating the branch condition, the processor fetches the instruction at LOOP or at NEXT, depending on whether the branch condition is true or false, respectively.

In either case, it completes execution of the shift instruction. The sequence of events during the last two passes in the loop is illustrated in Figure 5.13. Pipelined operation is not interrupted at any time, and there are no idle cycles. Logically, the program is executed as if the branch instruction were placed after the shift instruction. That is, branching takes place one instruction later than where the branch instruction appears in the instruction sequence in the memory, hence the name “delayed branch.”

LOOP	Shift_left	R1
	Decrement	R2
	Branch=0	LOOP
NEXT	Add	R1,R3

(a) Original program loop

LOOP	Decrement	R2
	Branch=0	LOOP
	Shift_left	R1
NEXT	Add	R1,R3

Fig:5.12 Reordering of instructions for a delayed

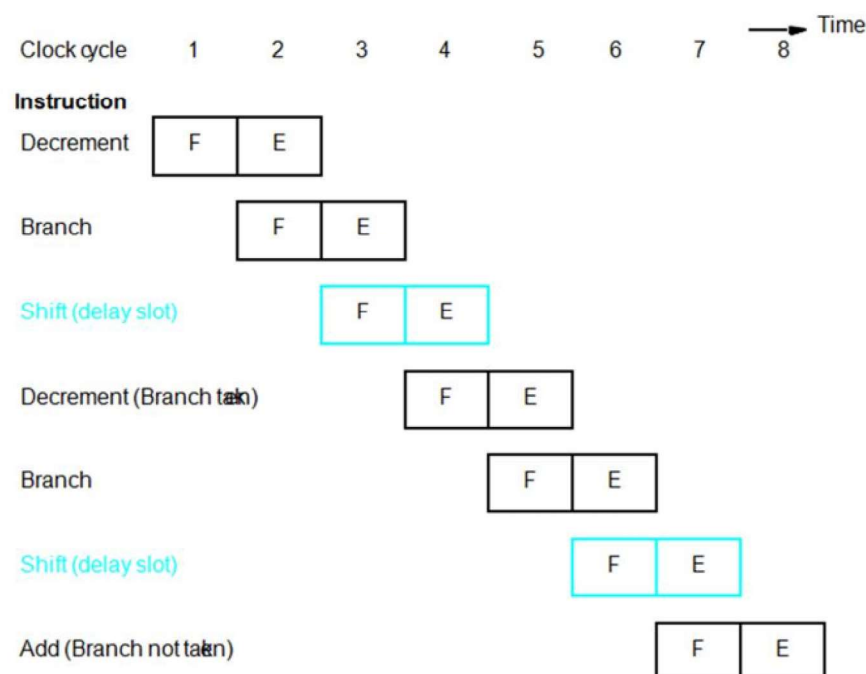


Fig:5.13 Execution timing showing the delay slot being filled during the last two passes through the loop in Figure 5.12

The effectiveness of the delayed branch approach depends on how often it is possible to reorder instructions as in Figure 5.12. Experimental data collected from many programs indicate that sophisticated compilation techniques can use one branch delay slot in as many as 85 percent of the cases. For a processor with two branch delay slots, the compiler attempts to find two instructions preceding the branch instruction that it can move into the delay slots without introducing a logical error. The chances of finding two such instructions are considerably less than the chances of finding one. Thus, if increasing the number of pipeline stages involves an increase in the number of branch delay slots, the potential gain in performance may not be fully realized.

Branch Prediction:

Another technique for reducing the branch penalty associated with conditional branches is to attempt to predict whether or not a particular branch will be taken. The simplest form of branch prediction is to assume that the branch will not take place and to continue to fetch instructions in sequential address order. Until the branch condition is evaluated, instruction execution along the predicted path must be done on a speculative basis. Speculative execution means that instructions are executed before the processor is certain that they are in the correct execution sequence. Hence, care must be taken that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed. If the branch decision indicates otherwise, the instructions and all their associated data in the execution units must be purged, and the correct instructions fetched and executed.

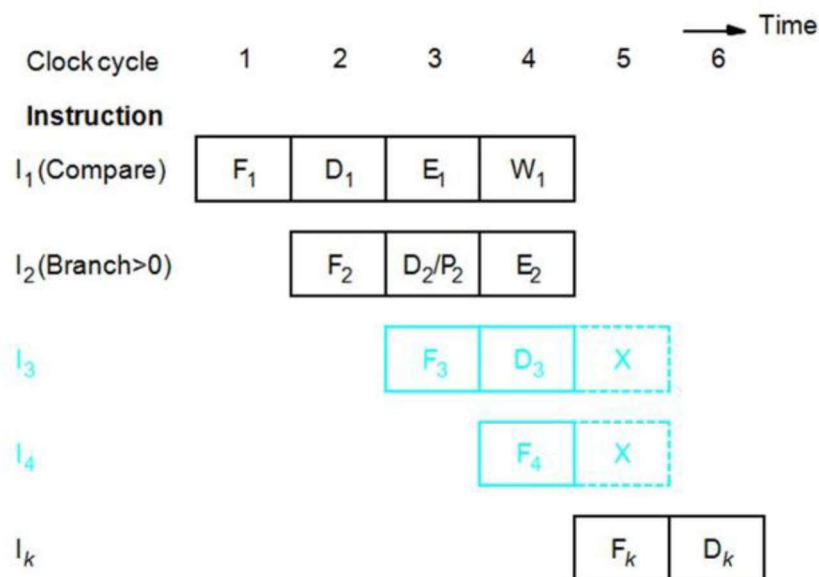


Figure 5.14. Timing when a branch decision has been incorrectly predicted as not taken.

An incorrectly predicted branch is illustrated in Figure 5.14 for a four-stage pipeline.

The figure shows a Compare instruction followed by a Branch > 0 instruction. Branch prediction takes place in cycle 3, while instruction I_3 is being fetched. The fetch unit predicts that the branch will not

be taken, and it continues to fetch instruction I4 as I3 enters the Decode stage. The results of the compare operation are available at the end of cycle 3. Assuming that they are forwarded immediately to the instruction fetch unit, the branch condition is evaluated in cycle 4. At this point, the instruction fetch unit realizes that the prediction was incorrect, and the two instructions in the execution pipe are purged. A new instruction, Ik, is fetched from the branch target address in clock cycle 5.

If branch outcomes were random, then half the branches would be taken. Then the simple approach of assuming that branches will not be taken would save the time lost to conditional branches 50 percent of the time. However, better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken, depending on the expected program behaviour. For example, a branch instruction at the end of a loop causes a branch to the start of the loop for every pass through the loop except the last one. Hence, it is advantageous to assume that this branch will be taken and to have the instruction fetch unit start to fetch instructions at the branch target address. On the other hand, for a branch instruction at the beginning of a program loop, it is advantageous to assume that the branch will not be taken.

A decision on which way to predict the result of the branch may be made in hardware by observing whether the target address of the branch is lower than or higher than the address of the branch instruction. A more flexible approach is to have the compiler decide whether a given branch instruction should be predicted taken or not taken. The branch instructions of some processors, such as SPARC, include a branch prediction bit, which is set to 0 or 1 by the compiler to indicate the desired behaviour. The instruction fetch unit checks this bit to predict whether the branch will be taken or not taken.

With either of these schemes, the branch prediction decision is always the same every time a given instruction is executed. Any approach that has this characteristic is called static branch prediction. Another approach in which the prediction decision may change depending on execution history is called dynamic branch prediction.

Influences On Instruction Sets:

We have seen that some instructions are much better suited to pipelined execution than others. For example, instruction side effects can lead to undesirable data dependencies. In this section, we examine the relationship between pipelined execution and machine instruction features. We discuss two key aspects of machine instructions — addressing modes and condition code flags.

Addressing Modes :

Addressing modes should provide the means for accessing a variety of data structures simply and efficiently. Useful addressing modes include index, indirect, autoincrement, and autodecrement. Many processors provide various combinations of these modes to increase the flexibility of their instruction sets. Complex addressing modes, such as those involving double indexing, are often encountered.

In choosing the addressing modes to be implemented in a pipelined processor, we must consider the effect of each addressing mode on instruction flow in the pipeline. Two important considerations in this regard are the side effects of modes such as autoincrement and autodecrement and the extent to which complex addressing modes cause the pipeline to stall. Another important factor is whether a given mode is likely to be used by compilers.

To compare various approaches, we assume a simple model for accessing operands in the memory. The load instruction $\text{Load } X(R1), R2$ takes five cycles to complete execution, as indicated in Figure 5.5. However, the instruction

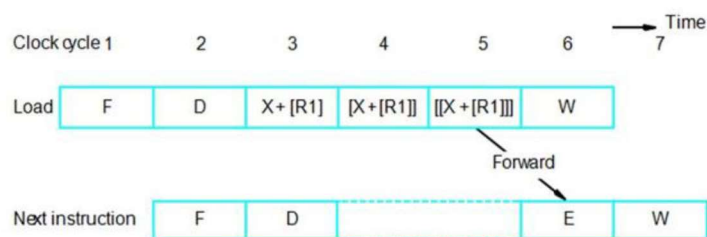
$\text{Load } (R1), R2$

can be organized to fit a four-stage pipeline because no address computation is required. Access to memory can take place in stage E. A more complex addressing mode may require several accesses to the memory to reach the named operand. For example, The instruction

$\text{Load } (X(R1)), R2$

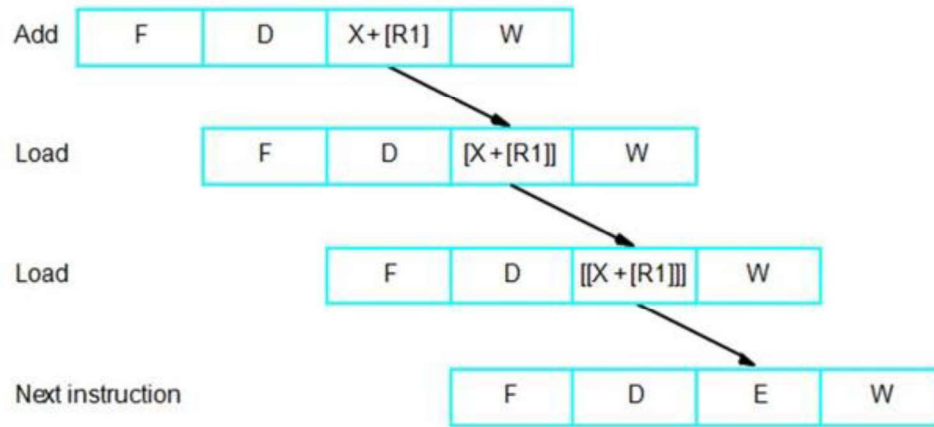
may be executed as shown in Figure 5.16a, assuming that the index offset, X , is given in the instruction word. After computing the address in cycle 3, the processor needs to access memory twice — first to read location $X+[R1]$ in clock cycle 4 and then to read location $[X+[R1]]$ in cycle 5. If $R2$ is a source operand in the next instruction, that instruction would be stalled for three cycles, which can be reduced to two cycles with operand forwarding, as shown.

$\text{Load } (X(R1)), R2$



(a) Complex addressing mode

Add #X, R1, R2
 Load (R2), R2
 Load (R2), R2



(b) Simple addressing mode

Figure 5.16 Equivalent operations using complex and simple addressing modes. To implement the same Load operation using only simple addressing modes requires several instructions. For example, on a computer that allows three operand addresses, we can use

Add #X,R1,R2

Load (R2),R2

Load (R2),R2

The Add instruction performs the operation $R2 \leftarrow X + [R1]$. The two Load instructions fetch the address and then the operand from the memory. This sequence of instructions takes exactly the same number of clock cycles as the original, single Load instruction, as shown in Figure 5.16b.

This example indicates that, in a pipelined processor, complex addressing modes that involve several accesses to the memory do not necessarily lead to faster execution. The main advantage of such modes is that they reduce the number of instructions needed to perform a given task and thereby reduce the program space needed in the main memory. Their main disadvantage is that their long execution times cause the pipeline to stall, thus reducing its effectiveness. They require more complex hardware to decode and execute them. Also, they are not convenient for compilers to work with.

The instruction sets of modern processors are designed to take maximum advantage of pipelined hardware. Because complex addressing modes are not suitable for pipelined execution, they should be avoided. The addressing modes used in modern processors often have the following features:

Access to an operand does not require more than one access to the memory.

- Only load and store instructions access memory operands.
- The addressing modes used do not have side effects

Three basic addressing modes that have these features are register, register indirect, and index. The first two require no address computation. In the index mode, the address can be computed in one cycle, whether the index value is given in the instruction or in a register. Memory is accessed in the following cycle. None of these modes has any side effects, with one possible exception. Some architectures, such as ARM, allow the address computed in the index mode to be written back into the index register. This is a side effect that would not be allowed under the guidelines above. Note also that relative addressing can be used; this is a special case of indexed addressing in which the program counter is used as the index register.

Condition Codes:

In many processors, the condition code flags are stored in the processor status register. They are either set or cleared by many instructions, so that they can be tested by subsequent conditional branch instructions to change the flow of program execution. An optimizing compiler for a pipelined processor attempts to reorder instructions to avoid stalling the pipeline when branches or data dependencies between successive instructions occur. In doing so, the compiler must ensure that reordering does not cause a change in the outcome of a computation. The dependency introduced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.

Consider the sequence of instructions in Figure 5.17a, and assume that the execution of the Compare and Branch=0 instructions proceeds as in Figure 8.14. The branch decision takes place in step E2 rather than D2 because it must await the result of the Compare instruction. The execution time of the Branch instruction can be reduced by interchanging the Add and Compare instructions, as shown in Figure 5.17b.

Add	R1,R2
Compare	R3,R4
Branch=0	...

(a) A program fragment

Compare	R3,R4
Add	R1,R2
Branch=0	...

(b) Instructions reordered

Figure 5.17 Instruction reordering.

This will delay the branch instruction by one cycle relative to the Compare instruction. As a result, at the time the Branch instruction is being decoded the result of the Compare instruction will be available and a correct branch decision will be made. There would be no need for branch prediction. However, interchanging the Add and Compare instructions can be done only if the Add instruction does not affect the condition codes.

These observations lead to two important conclusions about the way condition codes should be handled. First, to provide flexibility in reordering instructions, the condition-code flags should be affected by as few instructions as possible. Second, the compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not. An instruction set designed with pipelining in mind usually provides the desired flexibility. Figure 8.17b shows the instructions reordered assuming that the condition code flags are affected only when this is explicitly stated as part of the instruction OP code. The SPARC and ARM architectures provide this flexibility.

Data path and control constructions:

In a pipelined processor, the **datapath** is the set of hardware components (registers, ALU, memories) that perform calculations, and it is divided into stages like instruction fetch, decode, execute, memory access, and write-back. The **control unit** directs the datapath by generating control signals at each stage to manage data flow and operations, preventing interference between instructions moving through the pipeline. Registers are inserted between stages to hold intermediate data, ensuring that each instruction can progress independently.

Step 1 : label the control lines.

Control lines can be divided into five groups based on the five stages.

1. IF : Nothing to control.

control signals for reading Instruction Memory
and writing PC are always asserted (1).

2. ID: Nothing to control.

control signal of reading Register File is always asserted (1).

3. EX:

RegDst: select the destination field

ALUOp : 00 lw, sw

01 beq

10 R-format

11 immediate instructions

ALUSrc: select data for the second input to ALU

4. MEM:

Branch : asserted for branch instruction only, such as beq

MemRead: asserted for lw instruction only

MemWrite: asserted for sw only

5. WB:

RegWrite: asserted for lw and R-format

MemtoReg: select value to write back to Register File

Step 2 : setup the control values. Since pipelining the datapath does not change the meaning of control signals, the same control values in single-cycle MIPS processor can be used. In pipelined datapath, nine control signals can set to those values for each instruction.

Step 3 : Where to put the control unit?

Since control lines start with the EX stage, the control signals are needed to create during ID.

Fig.6.29 shows nine control signals are used in the last three stages as the instruction moves down the pipeline.

Pipeline registers have to be extended to include the control lines. Four control lines are used in the EX stage. The remaining five control lines pass on to the EX/MEM pipeline register. Three are used during the MEM stage, and the last two are passed to MEM/WB pipeline register for use in the WB stage. Fig. 6.30 shows the datapath with control.

Section 2 Example of Five Instructions

```
lw    $t0, 20($t1)
sub    $t1, $t2, $t3
and    $t2, $t4, $t5
or     $t3, $t6, $t7
add    $t4, $t8, $t9
```

This example had been used as one test example in the exercise design of pipelined datapath and control.

Figs. 6.31, 6.32, 6.33, 6.34, 6.35 are multiple clock pipeline diagrams that show how five instructions go through the pipelined datapath. It takes 9 clock to execute 5 instructions. By examining these five figures that label the pipeline execution including control, you could understand more about pipelined datapath and knows how to debug your design in case there are mistakes in your design.

Section 3 Data Hazards

Hazards: There are situations in pipelining when the next instruction cannot execute in the following clock cycle.

Data Hazards: an instruction depends on the results of a previous instruction still in the pipeline.

Example 1 :

```
add    $s0, $t0, $t1
sub     $t2, $s0, $t3
```


In the example, the second instruction is dependent on the result in \$s0 of the first instruction:

if \$s0 = -5 before add

\$s0 = 8 after add

then the value 8 should be used in the second instruction sub.

	Time								
Execution order	Time	CC1	CC2	CC3	CC4	CC5	CC6		
add \$s0, \$t0, \$t1		IF	---	ID---	EX---	MEM--	WB		
value in \$s0		-5	-5	-5	-5	-5/8	8		
sub \$t2, \$s0, \$t3				IF	---	ID---	EX---	MEM---	WB

For sub instruction, the value in \$s0 has to be read in its ID stage (CC3).

However, the value in \$s0 in CC3 is still -5 and not the correct value

8. We can only have the correct value in \$s0 at the end of clock 5 (CC5).

Another definition of Data Hazards:

dependencies that "go backward in time" are data hazards. For example, in the Example 1, data dependency (data in \$s0) goes backward from CC5 to CC3.