103-COMPUTER ORGANIZATION

(MCA-S.V.UNIVERSITY,TIRUPATI)

1-SEMESTER

STUDY MATERIAL

PREPARED BY: MISS.C.YAMINI M.C.A,
Department of Computer Science,
KMM INSTITUTE OF POST GRADUATION STUDIES.

KMM INSTITUTE OF POST GRADUATION STUDIES

Ramireddipalle, TIRUPATI-517102

Unit-1

1.Computer Organization:

The basic functional units of computer are made of electronics circuit and it works with electrical signal. We provide input to the computer in form of electrical signal and get the output in form of electrical signal.

There are two basic types of electrical signals, namely, **analog** and **digital**. The analog signals are continuous in nature and digital signals are discrete in nature.

The electronic device that works with continuous signals is known as **analog device** and the electronic device that works with discrete signals is known as **digital device**. In present days most of the computers are digital in nature and we will deal with Digital Computer in this course.

Computer is a digital device, which works on two levels of signal. We say these two levels of signal as **High** and **Low**. The High-level signal basically corresponds to some high-level signal (say 5 Volt or 12 Volt) and Low-level signal basically corresponds to Low-level signal (say 0 Volt). This is one convention, which is known as positive logic. There are others convention also like negative logic.

Since Computer is a digital electronic device, we have to deal with two kinds of electrical signals. But while designing a new computer system or understanding the working principle of computer, it is always difficult to write or work with 0V or 5V.

To make it convenient for understanding, we use some logical value, say,
LOW (L) - will represent 0V and
HIGH (H) - will represent 5V

Computer is used to solve mainly numerical problems. Again it is not convenient to work with symbolic representation. For that purpose we move to numeric representation. In this convention, we use 0 to represent **LOW** and 1 to represent **HIGH**.

0 means LOW
1 means HIGH

To know about the working principle of computer, we use two numeric symbols only namely 0 and 1. All the functionalities of computer can be captured with 0 and 1 and its theoretical background corresponds to two valued boolean algebra.

With the symbol 0 and 1, we have a mathematical system, which is knows as **binary number system**. Basically binary number system is used to represent the information and manipulation of information in computer. This information is basically strings of 0s and 1s.

The smallest unit of information that is represented in computer is known as Bit (Binary Digit), which is either 0 or 1. Four bits together is known as **Nibble**, and Eight bits together is known as **Byte**.

Computer Organization and Architecture

Computer technology has made incredible improvement in the past half century. In the early part of computer evolution, there were no stored-program computer, the computational power was less and on the top of it the size of the computer was a very huge one.

Today, a personal computer has more computational power, more main memory,more disk storage, smaller in size and it is available in effordable cost.

This rapid rate of improvement has come both from advances in the technology used to build computers and from innovation in computer design. In this course we will mainly deal with the innovation in computer design.

The task that the computer designer handles is a complex one: Determine what attributes are important for a new machine, then design a machine to maximize performance while staying within cost constraints.

This task has many aspects, including instruction set design, functional organization, logic design, and imple mentation.

While looking for the task for computer design, both the terms computer organization and computer architecture come into picture.

It is difficult to give precise definition for the terms Computer Organization and Computer Architecture. But while describing computer system, we come across these terms, and in literature, computer scientists try to make a distinction between these two terms.

Computer architecture refers to those parameters of a computer system that are visible to a programmer or those parameters that have a direct impact on the logical execution of a program. Examples of architectural attributes include the instruction set, the number of bits used to represent different data types, I/O mechanisms, and techniques for addressing memory.

Computer organization refers to the operational units and their interconnections that realize the architectural specifications. Examples of organizational attributes include those hardware details transparent to the programmer, such as control signals, interfaces between the computer and peripherals, and the memory technology used.

In this course we will touch upon all those factors and finally come up with the concept how these attributes contribute to build a complete computer system.

2.Basic Computer Model and different units of Computer

The model of a computer can be described by four basic units in high level abstraction which is shown in figure 1.1. These basic units are:

- Central Processor Unit
  Input Unit
  Output Unit
  Memory Unit

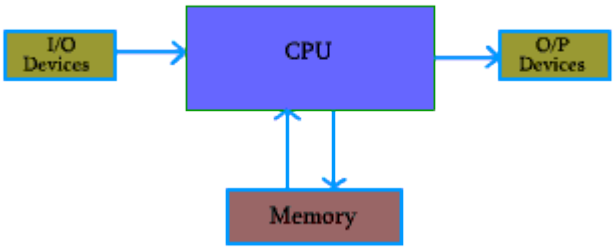


Figure 1.1: Basic Unit of a Computer

Basic Computer Model and different units of Computer

A. Central Processor Unit (CPU) :

Central processor unit consists of two basic blocks :

- The program control unit has a set of registers and control circuit to generate control signals.
- The execution unit or data processing unit contains a set of registers for storing data and an Arithmatic and Logic Unit (ALU) for execution of arithmatic and logical operations.

In addition, CPU may have some additional registers for temporary storage of data.

B. Input Unit :

With the help of input unit data from outside can be supplied to the computer. Program or data is read into main storage from input device or secondary storage under the control of CPU input instruction.

Example of input devices: Keyboard, Mouse, Hard disk, Floppy disk, CD-ROM drive etc.

C. Output Unit :

With the help of output unit computer results can be provided to the user or it can be stored in stograge device permanently for future use. Output data from main storage go to output device under the control of CPU output instructions.

Example of output devices: Printer, Monitor, Plotter, Hard Disk, Floppy Disk etc.

D. Memory Unit :

Memory unit is used to store the data and program. CPU can work with the information stored in memory unit. This memory unit is termed as primary memory or main memory module. These are basically semi conductor memories.

There ate two types of semiconductor memories -

- **Volatile Memory** **:** RAM (Random Access Memory).

- **Non-Volatile Memory :** ROM (Read only Memory), PROM (Programmable ROM) EPROM (Erasable PROM), EEPROM (Electrically Erasable PROM).

- **Secondary Memory :**
- There is another kind of storage device, apart from primary or main memory, which is known as secondary memory. **Secondary memories are non volatile memory** and it is used for permanent storage of data and program.
- Example of secondary memories:

Hard Disk, Floppy Disk, Magenetic Tape	------	These are magnetic devices,
CD-ROM	------	is optical device
Thumb drive (or pen drive)	------	is semiconductor memory.

Main Memory Organization

Main memory unit is the storage unit, There are several location for storing information in the main memory module.

The capacity of a memory module is specified by the number of memory location and the information stored in each location.

A memory module of capacity 16 X 4 indicates that, there are 16 location in the memory module and in each location, we can store 4 bit of information.

We have to know how to indicate or point to a specific memory location. This is done by address of the memory location.

We need two operation to work with memory.

READ Operation: This operation is to retrive the data from memory and bring it to CPU register

WRITE Operation: This operation is to store the data to a memory location from CPU register

We need some mechanism to distinguish these two operations READ and WRITE.

Main Memory Organization

Main memory unit is the storage unit, There are several location for storing information in the main memory module.

The capacity of a memory module is specified by the number of memory location and the information stored in each location.

A memory module of capacity 16 X 4 indicates that, there are 16 location in the memory module and in each location, we can store 4 bit of information.

We have to know how to indicate or point to a specific memory location. This is done by address of the memory location.

We need two operation to work with memory.

READ Operation: This operation is to retrive the data from memory and bring it to CPU register

WRITE Operation: This operation is to store the data to a memory location from CPU register

We need some mechanism to distinguish these two operations READ and WRITE.

3.Binary Number System

We have already mentioned that computer can handle with two type of signals, therefore, to represent any information in computer, we have to take help of these two signals.

These two signals corresponds to two levels of electrical signals, and symbolically we represent them as 0 and 1.

In our day to day activities for arithmatic, we use the *Decimal Number System*. The decimal number system is said to be of base, or radix 10, because it uses ten digits and the coefficients are multiplied by power of 10.

A decimal number such as 5273 represents a quantity equal to 5 thousands plus 2 hundres, plus 7 tens, plus 3 units. The thousands, hundreds, etc. are powers of 10 implied by the position of the coefficients. To be more precise, 5273 should be written as:

$$5\times10^3 + 2\times10^2 + 7\times10^1 + 3\times10^0$$

However, the convention is to write only the coefficient and from their position deduce the necessary power of 10.

In decimal number system, we need 10 different symbols. But in computer we have provision to represent only two symbols. So directly we can not use decimal number system in computer arithmatic.

For computer arithmatic we use **binary number system**. The binary number system uses two symbols to represent the number and these two symbols are 0 and 1.

The binary number system is said to be of base 2 or radix 2, because it uses two digits and the coefficients are multiplied by power of 2.

The binary number 110011 represents the quantity equal to:

$$1\times2^5 + 1\times2^4 + 0\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = 51$$ (in decimal)

We can use binary number system for computer arithmatic.

Representation of Unsigned Integers

Any integer can be stored in computer in binary form. As for example: The binary equivalent of integer 107 is 1101011, so 1101011 are stored to represent 107.

What is the size of Integer that can be stored in a Computer?
It depends on the word size of the Computer. If we are working with 8-bit computer, then we can use only 8 bits to represent the number.The eight bit computer means the storage organization for data is 8 bits.

In case of 8-bit numbers, the minimum number that can be stored in computer is 00000000 (0) and maximum number is 11111111 (255) (if we are working with natural numbers).

So, the domain of number is restricted by the storage capacity of the computer. Also it is related to number system; above range is for natural numbers.

In general, for *n-bit* number, the range for natural number is from $0 \; to \; 2^n - 1$

Any arithmetic operation can be performed with the help of binary number system. Consider the following two examples, where decimal and binary additions are shown side by side.

01101000	**104**
00110001	**49**
----------	------
10011001	**153**

In the above example, the result is an 8-bit number, as it can be stored in the 8-bit computer, so we get the correct results.

10000001	**129**
10101010	**178**
----------	------
100101011	**307**

In the above example, the result is a 9-bit number, but we can store only 8 bits, and the most significant bit (MSB) cannot be stored.

The result of this addition will be stored as (00101011) which is 43 and it is not the desired result. Since we cannot store the complete result of an operation, and it is known as the overflow case.

The smallest unit of information is known as **BIT** (BInary digit).

The binary number 110011 consists of 6 bits and it represents:

$$1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

For an *n-bit* number the coefficient is - a_j multiplied by 2^j where, ($0 \le j < n$)

The coefficient $a(n-1)$ is multiplied by $2^{(n-1)}$ and it is known as most significant bit (MSB).

The coefficient a_0 is multiplied by 2^0 and it is known as least significant bit (LSB).

For our convenient, while writing in paper, we may take help of other number systems like octal and hexadecimal. It will reduce the burden of writing long strings of 0s and 1s.

Octal Number : The octal number system is said to be of base, or radix 8, because it uses 8 digits and the coefficients are multiplied by power of 8.
Eight digits used in octal system are: 0, 1, 2, 3, 4, 5, 6 and 7.

Hexadecimal number : The hexadecimal number system is said to be of base, or radix 16, because it uses 16 symbols and the coefficients are multiplied by power of 16.
Sixteen digits used in hexadecimal system are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.

Consider the following addition example:

Binary	Octal	Hexadecimal	Decimal
01101000	150	68	104
00111010	072	3A	58
----------	------	------	-----
10100010	242	A2	162

Signed Integer

We know that for *n-bit* number, the range for natural number is from $0 \; to \; 2^n - 1$.

For *n-bit*, we have all together 2^n different combination, and we use these different combination to represent 2^n numbers, which ranges from $0 \; to \; 2^n - 1$.

If we want to include the negative number, naturally, the range will decrease. Half of the combinations are used for positive number and other half is used for negative number.

For *n-bit* represenatation, the range is from $(-2^{n-1} - 1) \; to \; (+2^{n-1} - 1)$.

For example, if we consider 8-bit number, then range

for natural number is from 0 to 255; but for signed integer the range is from -127 to +127.

Representation of signed integer

We know that for *n-bit* number, the range for natural number is from $0 \; to \; 2^n - 1$.

There are **three different schemes** to represent negative number:

- **Signed-Magnitude form.**
- **1's complement form.**
- **2's complement form.**

Signed magnitude form:

In signed-magnitude form, one particular bit is used to indicate the sign of the number, whether it is a positive number or a negative number. Other bits are used to represent the magnitude of the number.

For an n-bit number, one bit is used to indicate the signed information and remaining *(n-1) bits* are used to represent the magnitude. Therefore, the range is from $(-2^{n-1}-1)$ *to* $(+2^{n-1}-1)$.

Generally, Most Significant Bit (MSB) is used to indicate the sign and it is termed as signed bit. 0 in signed bit indicates positive numvber and 1 in signed bit indicates negative number.

For example, 01011001 represents + 169 and
11011001 represents - 169
What is 00000000 and 10000000 in signed magnitude form?

The concept of complement

The concept of complements is used to represent signed number.

Consider a number system of *base-r* or *radix-r*. There are two types of complements,

- The radix complement or the *r's* complement.
- The diminished radix complement or the *(r - 1)'s* complement.

Diminished Radix Complement : Given a number *N* in base *r* having *n* digits, the *(r - 1)'s* complement of *N* is defined as $(r^n - 1) - N$ For decimal numbers, $r = 10$ and $r - 1 = 9$, so the 9's complement of *N* is $(10^n - 1) - N$.

e.g., 9's complement of 5642 is 9999 - 5642 = 4357

Radix Complement : The *r's* complement of an *n-digit* number in base *r* is defined as $(r^N - N)$ for $N \neq 0$ and 0 for $N = 0$.
r's complement is obtained by adding 1 to the (r - 1)'s complement, since $(r^n - N) = [(r^n - 1) - N] + 1$

e.g., 10's complement of 5642 is 9's complement of 5642 + 1, i.e., 4357 + 1 = 4358
e.g., 2's complement of 1010 is 1's complement of 1010 + 1, i.e., 0101 + 1 = 0110.

Representation of Signed integer in 1's complement form:

Consider the eight bit number 01011100, 1's complements of this number is 10100011. If we perform the following addition:

0 1 0 1 1 1 0 0	
1 0 1 0 0 0 1 1	
----------	
1 1 1 1 1 1 1 1	

If we add 1 to the number, the result is 100000000.

Since we are considering an eight bit number, so the 9th bit (MSB) of the result can not be stored. Therefore, the final result is 00000000.

Since the addition of two number is 0, so one can be treated as the negative of the other number. So, 1's complement can be used to represent negative number.

Representation of Signed integer in 2's complement form:

Consider the eight bit number 01011100, 2's complements of this number is 10100100. If we perform the follwoing addition:

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

Since we are considering an eight bit number, so the 9th bit (MSB) of the result can not be stored. Therefore, the final result is 00000000.

Since the addition of two number is 0, so one can be treated as the negative of the other number. So, 2's complement can be used to represent negative number.

Decimal	2's Complement	1's complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	-----	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
-8	1000	------	-------

Representation of Real Number

Binary representation of 41.6875 is 101001.1011

Therefore any real number can be converted to binary number system

There are **two schemes** to represent real number :

- Fixed-point representation

- Floating-point representation

- **Fixed-point representation:**
- Binary representation of 41.6875 is 101001.1011
- To store this number, we have to store **two information**,
  -- the part before decimal point and
  -- the part after decimal point.
- This is known as fixed-point representation where the position of decimal point is fixed and number of bits before and after decimal point are also predefined.
- If we use 16 bits before decimal point and 7 bits after decimal point, in signed magnitude form, the range is
- $-2^{16} - 1$ to $+2^{16} - 1$ and the precision is $2^{(-7)}$
- One bit is required for sign information, so the total size of the number is 24 bits
- (1(sign) + 16(before decimal point) + 7(after decimal point)).

Floating-point representation:

- In this representation, numbers are represented by a mantissa comprising the significant digits and an exponent part of Radix R. The format is:

$$mantissa * R^{exponent}$$

- Numbers are often normalized, such that the decimal point is placed to the right of the **first non zero digit**.
- For example, the decimal number,

5236 is equivqlent to $.5236 * 10^4$

- To store this number in floating point representation, we store 5236 in mantissa part and 4 in exponent part.

IEEE standard floating point format:

IEEE has proposed two standard for representing floating-point number:

- Single precision
- Double precision

Single Precision:

S	E	M

S: sign bit: 0 denoted + and 1 denotes -

E: 8-bit excess -27 exponent

M: 23-bit mantissa

Double Precision:

S	E	M

S: sign bit: 0 denoted + and 1 denotes -

E: 11-bit excess -1023 exponent

M: 52-bit mantissa

Representation of Character

Since we are working with 0's and 1's only, to represent character in computer we use strings of 0's and 1's only.

To represent character we are using some coding scheme, which is nothing but a mapping function. Some of standard coding schemes are: **ASCII, EBCDIC, UNICODE**.

ASCII : American Standard Code for Information Interchange.
It uses a 7-bit code. All together we have 128 combinations of 7 bits and we can represent 128 character.
As for example 65 = 1000001 represents character 'A'.

EBCDIC : Extended Binary Coded Decimal Interchange Code.
It uses 8-bit code and we can represent 256 character.

UNICODE : It is used to capture most of the languages of the world. It uses 16-bit

Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language. The Unicode Standard has been adopted by such industry leaders as Apple, HP, IBM, JustSystem, Microsoft, Oracle, SAP, Sun, Sybase, Unisys and many others.

4.A Brief History of Computer Architecture

Computer Architecture is the field of study of selecting and interconnecting hardware components to create computers that satisfy functional performance and cost goals. It refers to those attributes of the computer system that are visible to a programmer and have a direct effect on the execution of a program.

Computer Architecture concerns Machine Organization, interfaces, application, technology, measurement & simulation that Includes:

- **Instruction set**
- **Data formats**
- **Principle of Operation** (formal description of every operation)
- **Features** (organization of programmable storage, registers used, interrupts mechanism, etc.)

In short, it is the combination of Instruction Set Architecture, Machine Organization and the related hardware.

Generation: The Brief History of Computer Architecture

First Generation (1940-1950) :: Vacuum Tube

- **ENIAC [1945]:** Designed by Mauchly & Echert, built by US army to calculate trajectories for ballistic shells during Worls War II. Around 18000 vacuum tubes and

1500 relays were used to build ENIAC, and it was programmed by manually setting switches

- **UNIVAC [1950]:** the first commercial computer
- **John Von Neumann architecture:** Goldstine and Von Neumann took the idea of ENIAC and developed concept of storing a program in the memory. Known as the Von Neumann's architecture and has been the basis for virtually every machine designed since then.

Features:

- Electron emitting devices
- Data and programs are stored in a single read-write memory
- Memory contents are addressable by location, regardless of the content itself
- Machine language/Assemble language
- Sequential execution

Second Generation (1950-1964) :: Transistors

- William Shockley, John Bardeen, and Walter Brattain invent the **transistor** that reduce size of computers and improve reliability. Vacuum tubes have been replaced by transistors.
- **First operating Systems:** handled one program at a time
- **On-off switches** controlled by electronically.
- **High level languages**
- **Floating point arithmetic**

Third Generation (1964-1974) :: Integrated Circuits (IC)

- **Microprocessor chips** combines thousands of transistors, entire circuit on one computer chip.
- **Semiconductor memory**
- **Multiple computer models** with different performance characteristics
- The **size** of computers has been reduced drastically

Fourth Generation (1974-Present) :: Very Large-Scale Integration (VLSI) / Ultra Large Scale Integration (ULSI)

- **Combines** millions of transistors
- **Single-chip processor** and the single-board computer emerged
- Creation of the **Personal Computer (PC)**
- Use of **data communications**
- Massively **parallel machine**

Evolution of Instruction Sets

Instruction Set Architecture (ISA) Abstract interface between the Hardware and lowest-level Software

- 1950: **Single Accumulator**: EDSAC
- 1953: **Accumulator plus Index Registers:** Manchester Mark I, IBM 700 series
- **Separation of programming Model from implementation**:
  - 1963: High-level language Based: B5000
  - 1964: Concept of a Family: IBM 360
- **General Purpose Register Machines**:
  - 1963-1976: **Load/Store Architecture**: CDC 6600, Cray 1
  - 1977-1980: **CISC** - Complex Instruction Sets computer: Vax, Intel 432
  - 1987: **RISC**: Reduced Instruction Set Computer: Mips, Sparc, HP-PA, IBM RS6000

Typical RISC:

- Simple, no complex addressing
- Constant length instruction, 32-bit fixed format
- Large register file
- Hard wired control unit, no need for micro programming
- Just about every opposites of CISC

Major advances in computer architecture are typically associated with landmark instruction set designs. Computer architecture's definition itself has been through bit changes. The following are the main concern for computer architecture through different times:

- 1930-1950: Computer arithmetic
  - Microprogramming
  - Pipelining
  - Cache
  - Timeshared multiprocessor
- 1960: Operating system support, especially memory management
  - Virtual memory
- 1970-1980: **Instruction Set Design**, especially for compilers; **Vector processing** and **shared memory multiprocessors**
  - RISC
- 1990s: Design of CPU, memory system, I/O system, multi-processors, networks
  - CC-UMA multiprocessor
  - CC-NUMA multiprocessor
  - Not-CC-NUMA multiprocessor
  - Message-passing multiprocessor
  - 2000s: Special purpose architecture, functionally reconfigurable,special considerations for low power/mobile processing, chip multiprocessors, memory systems
    - Massive SIMD
    - Parallel processing multiprocessor

Under a rapidly changing set of forces, computer technology keeps at dramatic change, for example:

- **Processor clock rate** at about 20% increase a year
- **Logic capacity** at about 30% increase a year
- **Memory speed** at about 10% increase a year
- **Memory capacity** at about 60% increase a year
- **Cost per bit** improves about 25% a year
- **The disk capacity** increase at 60% a year.

5.A Brief History of Computer Organization

If **computer architecture** is a view of the *whole design with the important characteristics* visible to programmer, **computer organization** is *how features are implemented with the specific building blocks* visible to designer, such as control signals, interfaces, memory technology, etc. Computer architecture and organization are closely related, though not exactly the same.

A stored program computer has the following basic units:

- **Processor** -- center for manipulation and control
- **Memory** -- storage for instructions and data for currently executing programs
- **I/O system** -- controller which communicate with "external" devices:
  secondary memory, display devices, networks
- **Data-path & control** -- collection of parallel wires, transmits data, instructions, or control signal

Computer organization defines the ways in which these components are interconnected and controlled. It is the capabilities and performance characteristics of those principal functional units. Architecture can have a number of organizational implementations, and organization differs between different versions. Such, all **Intel x86** families share the same basic architecture, and **IBM system/370** family share their basic architecture.

The history of Computer Organization

Computer architecture has progressed four generation: **vacuum tubes**, **transistors**, **integrated circuits**, and **VLSI**. Computer organization has also made its historic progression accordingly.

The advance of microprocessor (Intel)

- **1977:** 8080 - the first general purpose microprocessor, 8 bit data path, used in first personal computer
- **1978:** 8086 - with 16 bit, 1MB addressable, instruction cache, prefetch few instructions

- **1980:** 80186 - identical to 8086 widh additional reserved interrupt vectors and some very powerful buli-in I/O functions.
- **1982:** 80286 - 24 Mbyte addressable memory space, plus instructions
- **1985:** 80386 - 32 bit, new addressing modes and support for multitasking
- **1989 -- 1995:**
  - 80486 - 25, 33, MHz, 1.2 M transistors, 5 stage pipeline, sophisticated powerful cache and instruction pipelining, built in math co-processor.
  - Pentium - 60, 66 MHz, 3.1 M transistor, branch predictor, pipelined floating point, multiple instructions executed in parallel, first superscalar IA-32.
  - PentiumPro - Increased superscalar, register renaming, branch prediction, data flow analysis, and speculative execution
- **1995 -- 1997:** Pentium II - 233, 166, 300 MHz, 7.5 M transistors, first compaction of micro-architecture, MMX technology, graphics video and audio processing.
- **1999:** Pentium III - additional floating point instructions for 3D graphics
- **2000:** Pentium IV - Further floating point and multimedia enhancements

Evolution of Memory

- 1970: **RAM /DRAM**, 4.77 MHz
- 1987: **FPM** - fast page mode DRAM, 20 MHz
- 1995, **EDO** - Extended Data Output, which increases the read cycle between memory and CPU, 20 MHz
- 1997- 1998: **SDRAM** - Synchronous DRAM, which synchronizes itself with the CPU bus and runs at higher clock speeds, PC66 at 66 MHz, PC100 at 100 MHz
- 1999: **RDRAM** - Rambus DRAM, which DRAM with a very high bandwidth, 800 MHz
- 1999-2000: **SDRAM** - PC133 at 133 MHz, DDR at 266 MHz.
- 2001: **EDRAM** - Enhanced DRAM, which is dynamic or power-refreshed RAM, also know as cached DRAM.

6.Basic Operational Concepts

An Instruction consists of two parts, an Operation code and operand/s

- Let us see a typical instruction ADD LOCA, R0 This instruction is an addition operation. The following are the steps to execute the instruction
- **Step 1:** Fetch the instruction from main memory into the processor
- **Step 2:** Fetch the operand at location LOCA from main memory into the processor
- **Step 3:** Add the memory operand (i.e. fetched contents of LOCA) to the contents of register R0
- **Step 4:** Store the result (sum) in R0.

- The same instruction can be realized using two instructions as
- Load LOCA, R1
- Add R1, R0
- The steps to execute the instructions can be enumerated as below:
- **Step 1:** Fetch the instruction from main memory into the processor
- **Step 2:** Fetch the operand at location LOCA from main memory into the processor Register R1
- **Step 3:** Add the content of Register R1 and the contents of register R0
- **Step 4:** Store the result (sum) in R0.
- Figure 3 below shows how the memory and the processor are connected. As shown in the diagram, in addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes. The instruction register holds the instruction that is currently being executed. The program counter keeps track of the execution of the program. It contains the memory address of the next instruction to be fetched and executed. There are n general purpose registers R0 to Rn-1 which can be used by the programmers during writing programs.

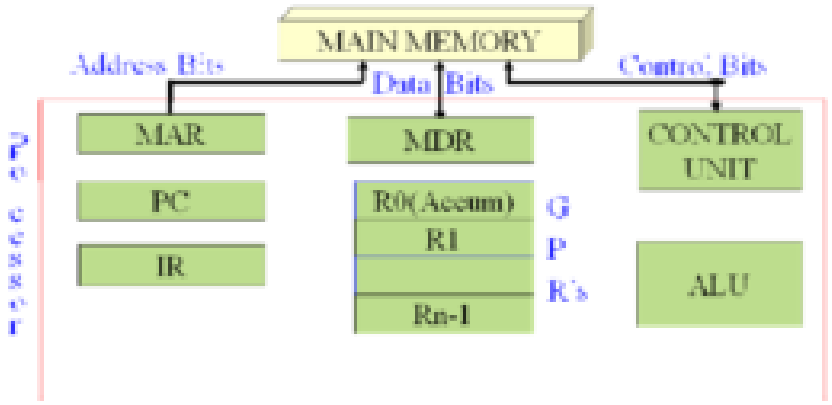


Figure 3: Connections between the processor and the memory

- The interaction between the processor and the memory and the direction of flow of information is as shown in the diagram below:

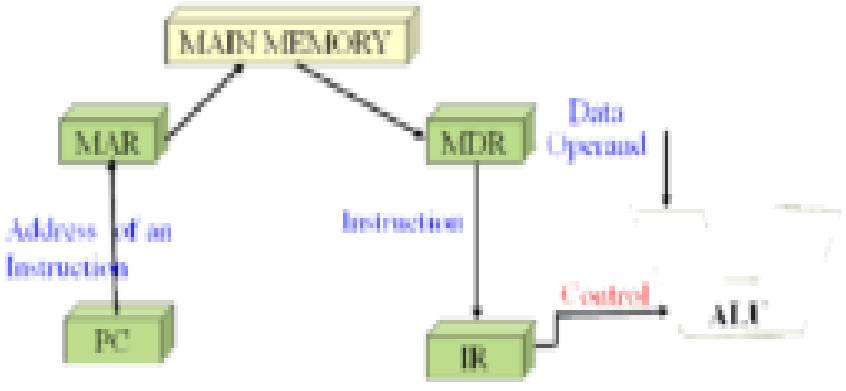
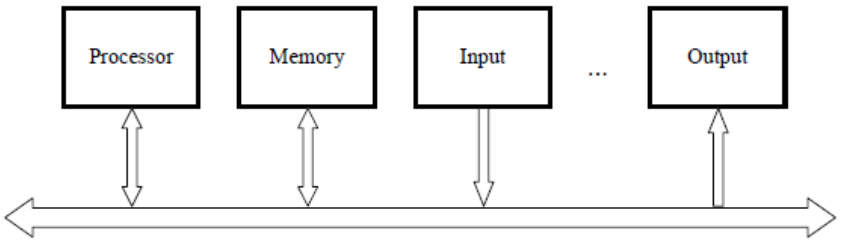


Figure 4: Interaction between the memory and the ALU

7.Bus structure

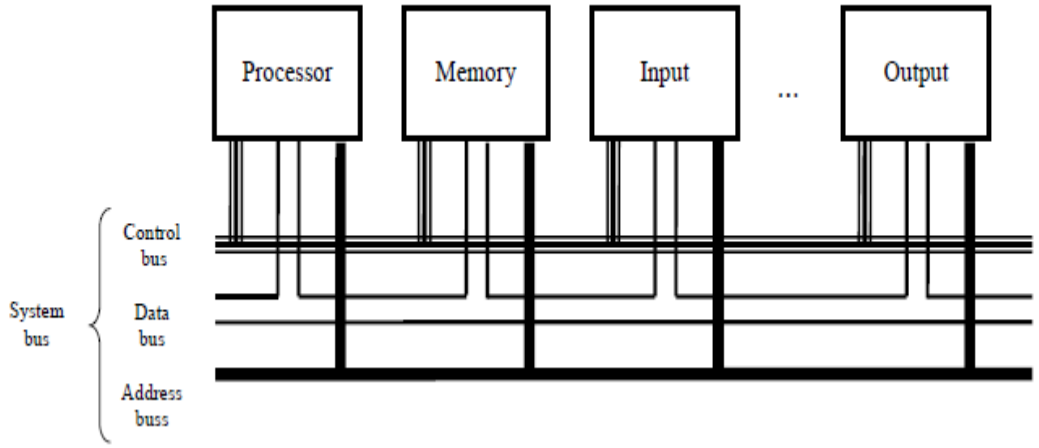
Single bus structure: In computer architecture, a bus is a subsystem that transfers data between components inside a computer, or between computers. Early computer buses were literally parallel electrical wires with multiple connections, but Modern computer buses can use both parallel and bit serial connections.



- **Figure 1.3.1 Single bus structure**

To achieve a reasonable speed of operation, a computer must be organized so that all its units can handle one full word of data at a given time. When a word of data is transferred between units, all its bits are transferred in parallel, that is, the bits are transferred simultaneously over many wires, or lines, one bit per line. A group of lines that serves as a connecting path for several devices is called a bus. In addition to the lines that carry the data, the bus must have lines for address and control purposes. The simplest way to interconnect functional units is to use a single bus, as shown in Figure 1.3.1. All units are connected to this bus. Because the bus can be used for only one transfer at a time, only two units can actively use the bus at any given time. Bus control lines are used to arbitrate multiple requests for use of the bus. The main virtue of the single-bus structure is its low cost and is flexibility for attaching peripheral" devices. Systems that contain multiple buses achieve more concurrency in operations by allowing two or more transfers to be carried out at the same time. This leads to better performance but at an increased cost.

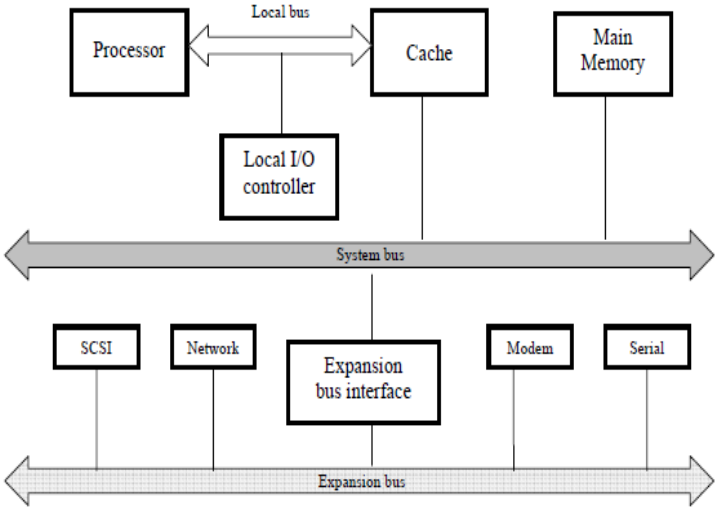
- **Parts of a System bus**: Processor, memory, Input and output devices are connected by system bus, which consists of separate busses as shown in figure 1.3.2. They are:
  (i)Address bus: Address bus is used to carry the address. It is unidirectional bus. The address is sent to from CPU to memory and I/O port and hence unidirectional. It consists of 16, 20, 24 or more parallel signal lines.
  (ii)Data bus: Data bus is used to carry or transfer data to and from memory and I/O ports. They are bidirectional. The processor can read on data lines from memory and I/O port and as well as it can write data to memory. It consists of 8, 16, 32 or more parallel signal lines.
  (iii)Control bus: Control bus is used to carry control signals in order to regulate the control activities. They are bidirectional. The CPU sends control signals on the control bus to enable the outputs of addressed memory devices or port devices. Some of the control signals are: MEMR (memory read), MEMW (memory write), IOR (I/O read), IOW (I/O write), BR (bus request), BG (bus grant), INTR (interrupt request), INTA (interrupt acknowledge), RST (reset), RDY (ready), HLD (hold), HLDA (hold acknowledge),



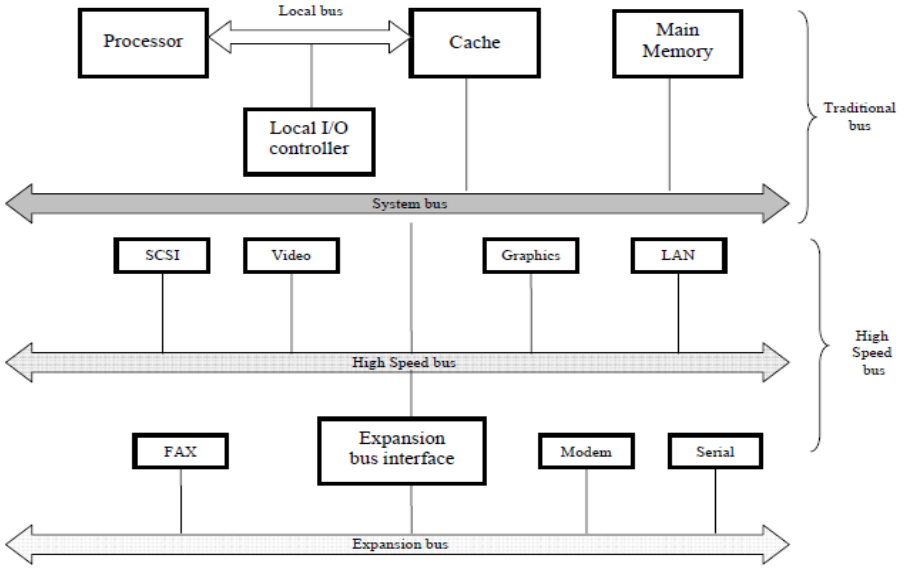
- **Figure 1.3.2 Bus interconnection scheme**

The devices connected to a bus vary widely in their speed of operation. Some electromechanical devices, such as keyboards and printers are relatively slow. Other devises like magnetic or optical disks, are considerably faster. Memory and processor units operate at electronic speeds, making them the fastest parts of a computer. Because all these devices must communicate with each other over a bus, an efficient transfer mechanism that is not constrained by the slow devices and that can be used to smooth out the differences in timing among processors, memories, and external devices is necessary.

- A common approach is to include buffer registers with the devices to hold the information during transfers. To illustrate this technique, consider the transfer of an encoded character from a processor to a character printer. The processor sends the character over the bus to the printer buffer. Since the buffer is an electronic register, this transfer requires relatively little time. Once the buffer is loaded, the printer can start printing without further intervention by the processor. The bus and the processor are no longer needed and can be released for other activity. The printer continues printing the character in its buffer and is not available for further transfers until this process is completed. Thus, buffer registers smooth out timing differences among processors, memories, and I/O devices. They prevent a high-speed processor from being locked to a slow I/O device during a sequence of data transfers. This allows the processor to switch rapidly from one device to another, interweaving its processing activity with data transfers involving several I/O devices.
- The Figure 1.3.3 shows traditional bus configurations and the Figure 1.3.4 shows high speed bus configurations. The traditional bus connection uses three buses: local bus, system bus and expanded bus. The high speed bus configuration uses high-speed bus along with the three buses used in the traditional bus connection. Here, cache controller is connected to highspeed bus. This bus supports connection to high-speed LANs, such as Fiber Distributed Data Interface (FDDI), video and graphics workstation controllers, as well as interface controllers to local peripheral including SCSI.



- **Figure 1.3.3 Traditional bus configuration**



- **Figure 1.3.4 High speed bus configuration**

8.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 response time, throughput and execution time of a computer system.

Response time is the time from 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

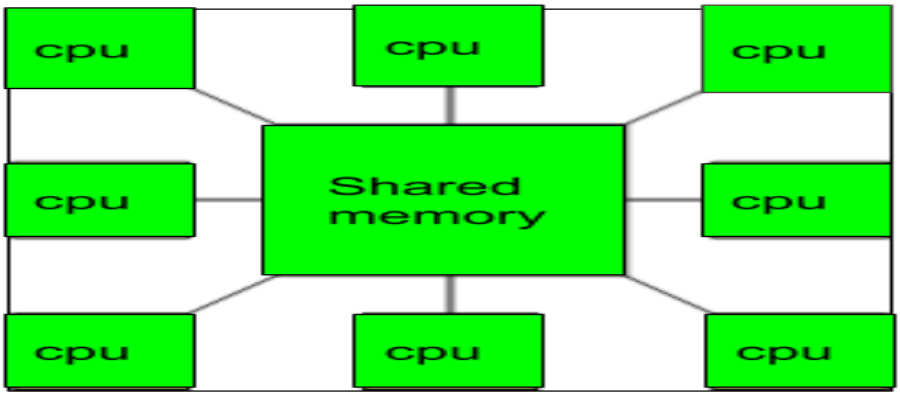
The units for CPU Execution time are:

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

9.Introduction of Multiprocessor and Multicomputer

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 shares the common memory but in a distributed memory multiprocessor, every CPU has its own private memory.



Applications of Multiprocessor –

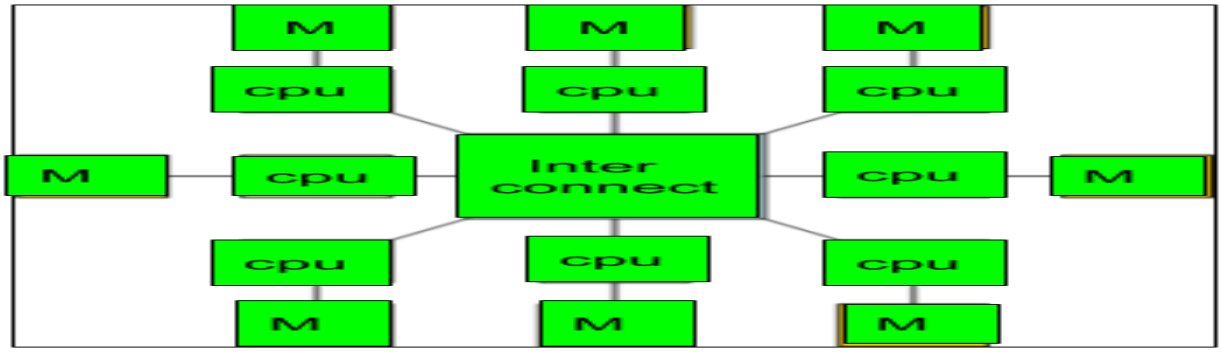
1. As a uniprocessor, such as single instruction, single data stream (SISD).
2. As a multiprocessor, such as single instruction, multiple data stream (SIMD), which is usually used for vector processing.
3. Multiple series of instructions in a single perspective, such as multiple instruction, single data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

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.

10.Logic Gates:

- The logic gates are the main structural part of a digital system.
- Logic Gates are a block of hardware that produces signals of binary 1 or 0 when input logic requirements are satisfied.
- Each gate has a distinct graphic symbol, and its operation can be described by means of algebraic expressions.
- The seven basic logic gates includes: AND, OR, XOR, NOT, NAND, NOR, and XNOR.
- The relationship between the input-output binary variables for each gate can be represented in tabular form by a truth table.
- Each gate has one or two binary input variables designated by A and B and one binary output variable designated by x.

AND GATE:

The AND gate is an electronic circuit which gives a high output only if all its inputs are high. The AND operation is represented by a dot (.) sign.

AND Gate:

Algebraic Function: x = AB

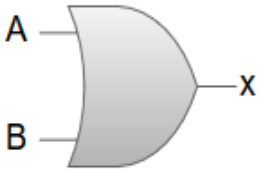
Truth Table:

A	B	x
0	0	0
0	1	0
1	0	0
1	1	1

OR GATE:

The OR gate is an electronic circuit which gives a high output if one or more of its inputs are high. The operation performed by an OR gate is represented by a plus (+) sign.

OR Gate:



Algebraic Function: x = A + B

Truth Table:

A	B	x
0	0	0
0	1	1
1	0	1
1	1	1

NOT GATE:

The NOT gate is an electronic circuit which produces an inverted version of the input at its output. It is also known as an **Inverter**.

NAND GATE:

The NOT-AND (NAND) gate which is equal to an AND gate followed by a NOT gate. The NAND gate gives a high output if any of the inputs are low. The NAND gate is represented by a AND gate with a small circle on the output. The small circle represents inversion.

NAND Gate:

Algebraic Function: x = (AB)'

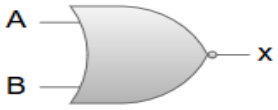
Truth Table:

A	B	x
0	0	1
0	1	1
1	0	1
1	1	0

NOR GATE:

The NOT-OR (NOR) gate which is equal to an OR gate followed by a NOT gate. The NOR gate gives a low output if any of the inputs are high. The NOR gate is represented by an OR gate with a small circle on the output. The small circle represents inversion.

NOR Gate:



Algebraic Function: x = (A+B)'

Truth Table:

A	B	x
0	0	1
0	1	0
1	0	0
1	1	0

Exclusive-OR/ XOR GATE:

The 'Exclusive-OR' gate is a circuit which will give a high output if one of its inputs is high but not both of them. The XOR operation is represented by an encircled plus sign.

XOR Gate:

Algebraic Function: $x = A \oplus B$

or

$x = A'B + AB'$

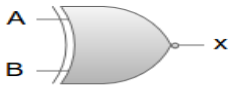
Truth Table:

A	B	x
0	0	0
0	1	1
1	0	1
1	1	0

EXCLUSIVE-NOR/Equivalence GATE:

The 'Exclusive-NOR' gate is a circuit that does the inverse operation to the XOR gate. It will give a low output if one of its inputs is high but not both of them. The small circle represents inversion.

Exclusive-NOR Gate:



Algebraic Function: $x = (A \oplus B)'$

or

$x = A'B' + AB$

Truth Table:

A	B	x
0	0	1
0	1	0
1	0	0
1	1	1

Boolean algebra

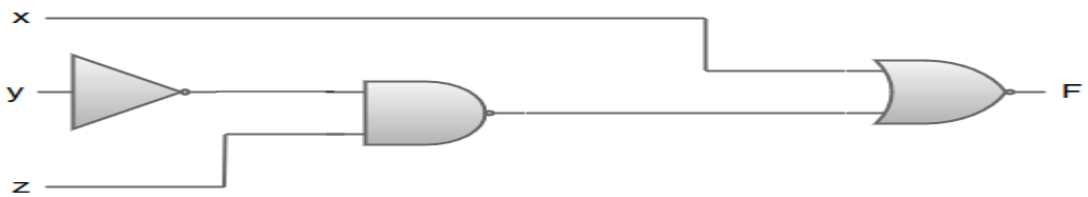
Boolean algebra can be considered as an algebra that deals with binary variables and logic operations. Boolean algebraic variables are designated by letters such as A, B, x, and y. The basic operations performed are AND, OR, and complement.

The Boolean algebraic functions are mostly expressed with binary variables, logic operation symbols, parentheses, and equal sign. For a given value of variables, the Boolean function can be either 1 or 0. For instance, consider the Boolean function:

F = x + y'z

The logic diagram for the Boolean function F = x + y'z can be represented as:

F = x + y'z



- The Boolean function F = x + y'z is transformed from an algebraic expression into a logic diagram composed of AND, OR, and inverter gates.
- Inverter at input 'y' generates its complement y'.
- There is an AND gate for the term y'z, and an OR gate is used to combine the two terms (x and y'z).
- The variables of the function are taken to be the inputs of the circuit, and the variable symbol of the function is taken as the output of the circuit.
- The truth table for the Boolean function F = x + y'z can be represented as:

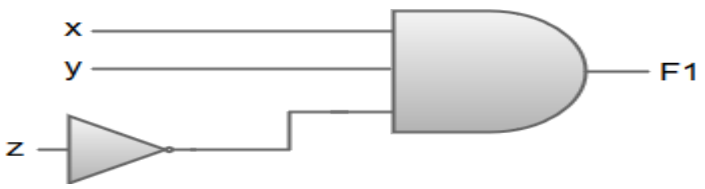
F = x + y'z

x	y	z	F
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	1

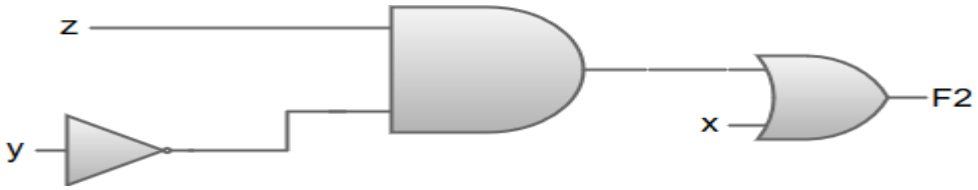
11.Examples of Boolean algebra simplifications using logic gates

In this section, we will look at some of the examples of Boolean algebra simplification using Logic gates.

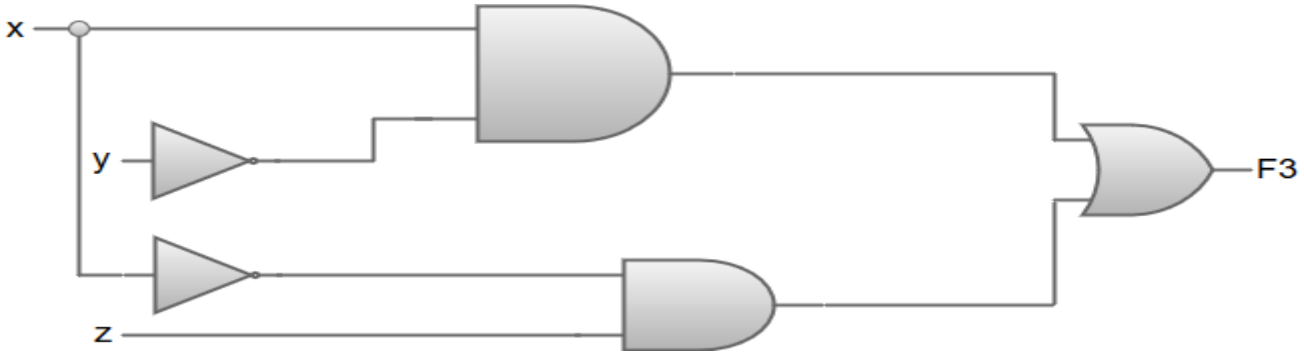
1. F1 = xyz'



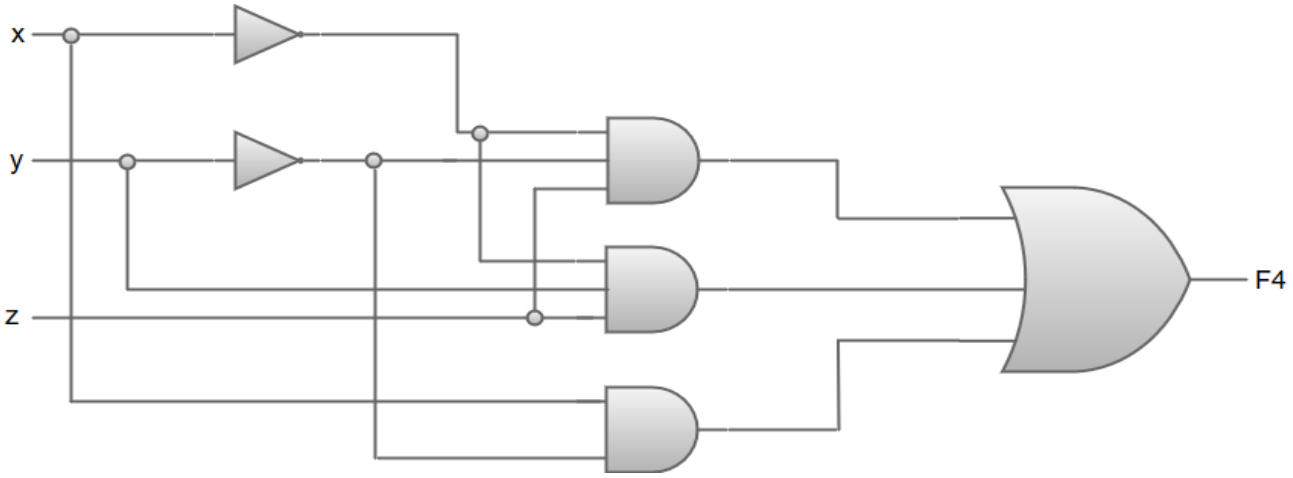
2. F2 = x + y'z



3. F3 = xy' + x'z



4. F4 = x'y'z + x'yz + xy'



Truth tables for F1= xyz', F2= x+y'z, F3= xy'+x'z and F4= x'y'z+x'yz+xy'

x	y	z	F1	F2	F3	F4
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1

1	0	1	0	1	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	0

12.Laws of Boolean algebra

The basic Laws of Boolean Algebra can be stated as follows:

- Commutative Law states that the interchanging of the order of operands in a Boolean equation does not change its result. For example:
    1. OR operator → A + B = B + A
    2. AND operator → A * B = B * A
- Associative Law of multiplication states that the AND operation are done on two or more than two variables. For example:
  A * (B * C) = (A * B) * C
- Distributive Law states that the multiplication of two variables and adding the result with a variable will result in the same value as multiplication of addition of the variable with individual variables. For example:
  A + BC = (A + B) (A + C).
- Annulment law:
  A.0 = 0
  A + 1 = 1
- Identity law:
  A.1 = A
  A + 0 = A
- Idempotent law:
  A + A = A
  A.A = A
- Complement law:
  A + A' = 1
  A.A'= 0
- Double negation law:
  ((A)')' = A

- Absorption law:

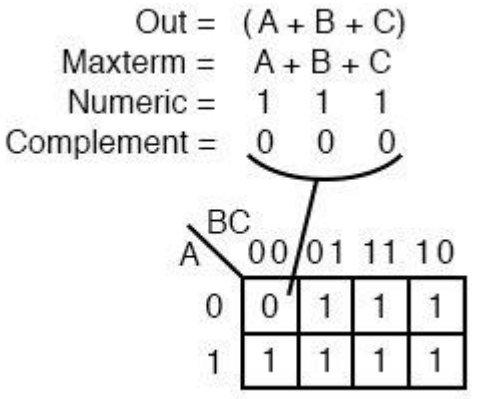
  A.(A+B) = A

  A + AB = A

De Morgan's Law is also known as De Morgan's theorem, works depending on the concept of Duality. Duality states that interchanging the operators and variables in a function, such as replacing 0 with 1 and 1 with 0, AND operator with OR operator and OR operator with AND operator.

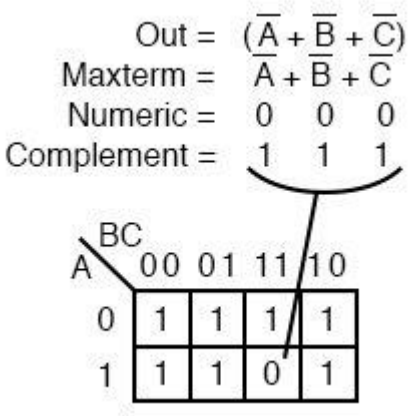
13.Minterm

A *minterm* is a Boolean expression resulting in **1** for the output of a single cell, and **0**s for all other cells in a Karnaugh map, or truth table. If a minterm has a single **1** and the remaining cells as **0**s, it would appear to cover a minimum area of **1**s



Maxterm

A *maxterm* is a Boolean expression resulting in a **0** for the output of a single cell expression, and **1**s for all other cells in the Karnaugh map, or truth table. The illustration above left shows the maxterm **(A+B+C)**, a single sum term, as a single **0** in a map that is otherwise **1**s.



14.Introduction of K-Map (Karnaugh Map): In many digital circuits and practical problems we need to find expression with minimum variables. We can minimize Boolean expressions of 3, 4 variables very easily using K-map without using any Boolean algebra theorems. K-map can take two forms Sum of Product (SOP) and Product of Sum (POS) according to the need of problem. K-map is table like representation but it gives more information than TRUTH TABLE. We fill grid of K-map with 0's and 1's then solve it by making groups.

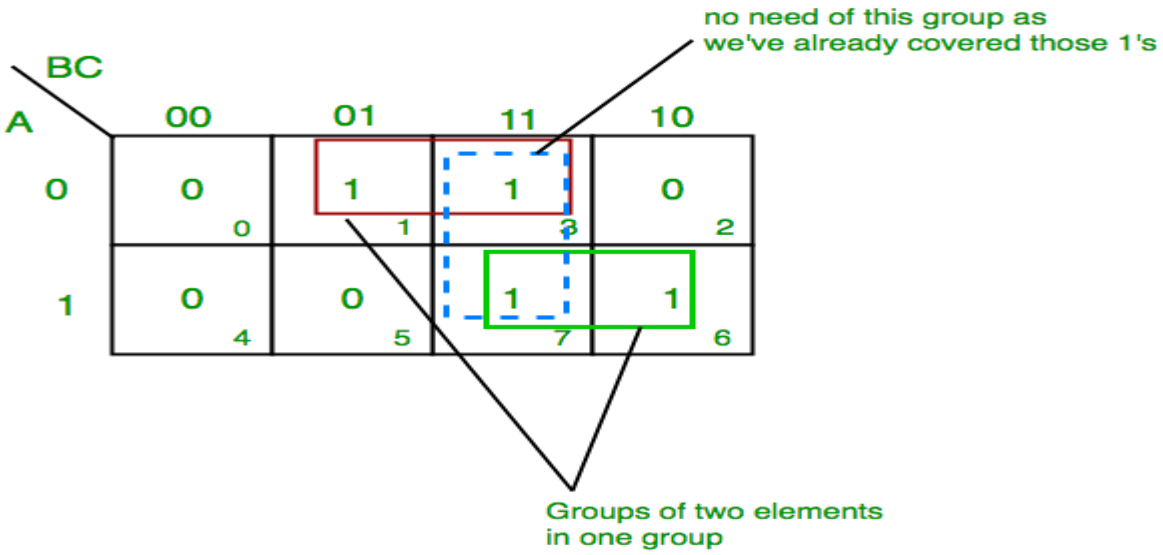
Steps to solve expression using K-map-

1. Select K-map according to the number of variables.
2. Identify minterms or maxterms as given in problem.
3. For SOP put 1's in blocks of K-map respective to the minterms (0's elsewhere).
4. For POS put 0's in blocks of K-map respective to the maxterms(1's elsewhere).
5. Make rectangular groups containing total terms in power of two like 2,4,8 ..(except 1) and try to cover as many elements as you can in one group.
6. From the groups made in step 5 find the product terms and sum them up for SOP form.

SOP FORM

1. **K-map of 3 variables-**

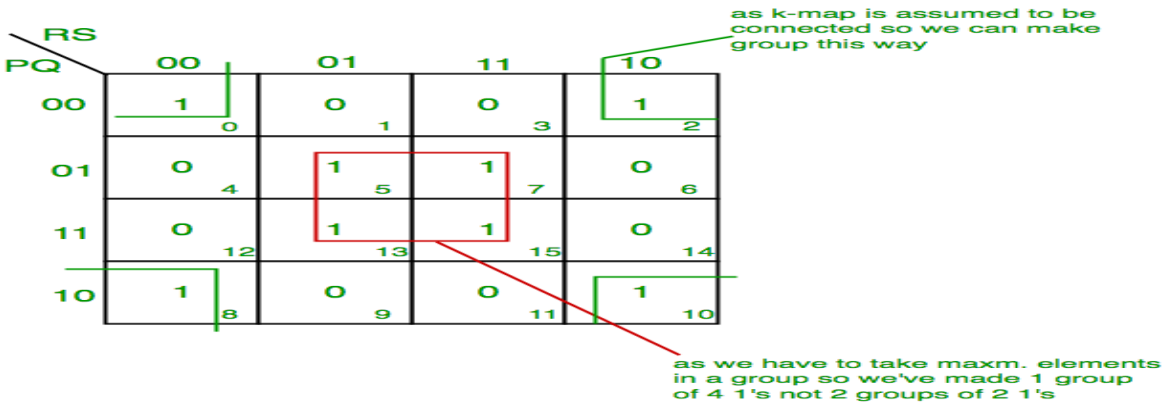
Z= ∑A,B,C(1,3,6,7)



(A'C+AB)

K-map for 4 variables

F(P,Q,R,S)=∑(0,2,5,7,8,10,13,15)

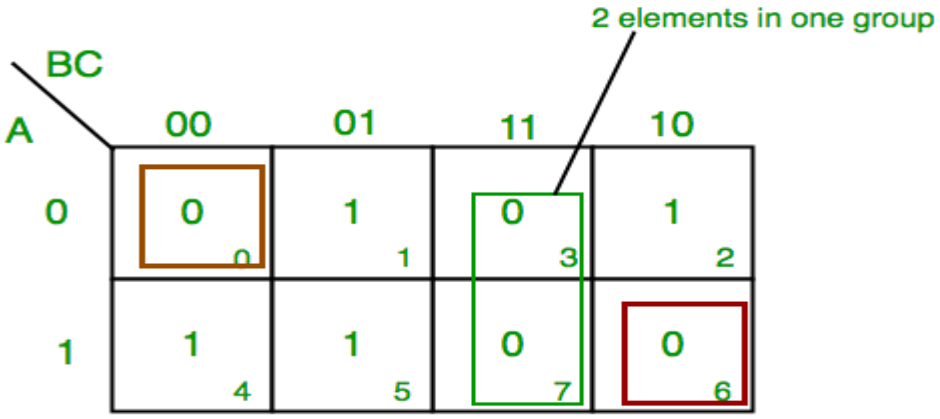


Final expression (QS+Q'S')

POS FORM

1. **K-map of 3 variables-**

F(A,B,C)=π(0,3,6,7)



(A' + B' + C) (B' + C') (A + B + C)

15.Don't Care (X) Conditions in K-Maps

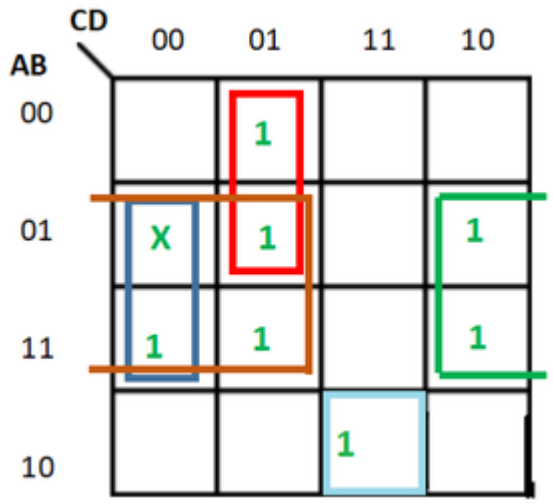
One of the very significant and useful concept in simplifying the output expression using K-Map is the concept of "Don't Cares". The "Don't Care" conditions allow us to replace the empty cell of a K-Map to form a grouping of the variables which is larger than that of forming groups without don't cares. While forming groups of cells, we can consider a "Don't Care" cell as 1 or 0 or we can also ignore that cell. Therefore, "Don't Care" condition can help us to form a larger group of cells. A Don't Care cell can be represented by a cross(X) in K-Maps representing a invalid combination. For example, in Excess-3 code system, the states 0000, 0001, 0010, 1101, 1110 and 1111 are invalid or unspecified. These states are called don't cares.

A standard SOP function having don't cares can be converted into a POS expression by keeping don't cares as they are, and writing the missing minterms of the SOP form as the maxterm of POS form. Similarly, a POS function having don't cares can be converted to SOP form keeping the don't cares as they are and writing the missing maxterms of the POS expression as the minterms of SOP expression.

Example-1:
Minimise the following function in SOP minimal form using K-Maps:

f = m(1, 5, 6, 11, 12, 13, 14) + d(4)

Explanation:
The SOP K-map for the given expression is:



Therefore, SOP minimal is,

f = BC' + BCD' + A'C'D + AB'CD

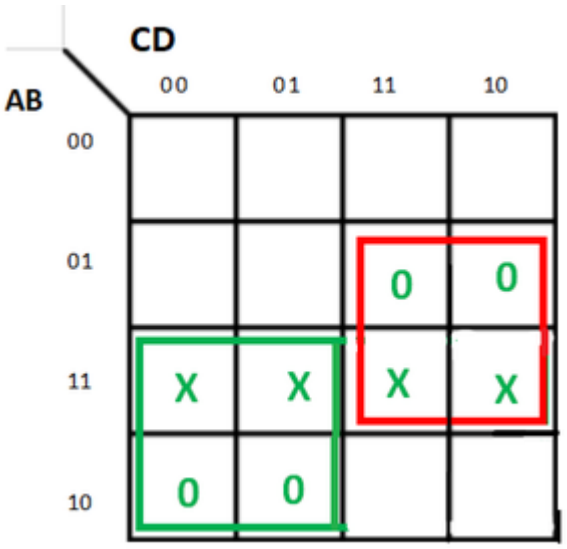
Example-2:
Minimise the following function in POS minimal form using K-Maps:

F(A, B, C, D) = m(0, 1, 2, 3, 4, 5) + d(10, 11, 12, 13, 14, 15)

Explanation:
Writing the given expression in POS form:

F(A, B, C, D) = M(6, 7, 8, 9) + d(12, 13, 14, 15)

The POS K-map for the given expression is:



Therefore, POS minimal is,

F = (A'+ C)(B' + C')

Significance of "Don't Care" Conditions:
Don't Care conditions has the following significance in designing of the digital circuits:

1. **Simplification of the output:**
   These conditions denotes inputs that are invalid for a given digital circuit. Thus, they can used to further simplify the boolean output expression of a digital circuit.

2. **Reduction in number of gates required:**
   Simplification of the expression reduces the number of gates to be used for implementing the given expression. Therefore, don't cares make the digital circuit design more economical.

3. **Reduced Power Consumption:**
   While grouping the terms long with don't cares reduces switching of the states. This decreases the memory space that is required to represent a given digital circuit which in turn results in less power consumption.

4. **Represent Invalid States in Code Converters:**
   These are used in code converters. For example- In design of 4-bit BCD-to-XS-3 code converter, the input combinations 1010, 1011, 1100, 1101, 1110, and 1111 are don't cares.

5. **Prevention of Hazards in Digital Circuits:**
   Don't cares also prevents hazards in digital systems.

16.Flip Flops

Do you know!! computers and calculators use Flip-flop for their memory. A combination of number of flip flops will produce some amount of memory.

Flip flop is formed using logic gates, which are in turn made of transistors. Flip flop are basic building blocks in the memory of electronic devices. Each flip flop can store one bit of data.

These are also called as sequential logic circuits. Also know these before learning about fliplfops.

- Sequential Logic circuits
- Latches

Flip – flops have two stable states and hence they are bistable multivibrators. The two stable states are High (logic 1) and Low (logic 0).

The term flip – flop is used as they can switch between the states under the influence of a control signal (clock or enable) i.e. they can 'flip' to one state and 'flop' back to other state.

- Flip – flops are a binary storage device because they can store binary data (0 or 1).
- Flip – flops are edge sensitive or edge triggered devices i.e. they are sensitive to the transition rather than the duration or width of the clock signal.
- They are also known as signal change sensitive devices which mean that the change in the level of clock signal will bring change in output of the flip flop.
- A Flip – flop works depending on clock pulses.
- Flip flops are also used to control the digital circuit's functionality. They can change the operation of a digital circuit depending on the state.

Some of the most common flip – flops are SR Flip – flop (Set – Reset), D Flip – flop (Data or Delay), JK Flip – flop and T Flip – flop.

Latches vs Flip-Flops

Latches and flip – flops are both 1 – bit binary data storage devices. The main difference between a latch and a flip – flop is the triggering mechanism. Latches are transparent when enabled ,whereas flip – flops are dependent on the transition of the clock signal i.e. either positive edge or negative edge.

The modern usage of the term flip – flop is reserved to clocked devices and term latch is to describe much simpler devices. Some of the other differences between latches and flip – flops are listed in below table.

LATCH	FLIP – FLOP
Latches do not require clock signal.	Flip – flops have clock signals
A latch is an asynchronous device.	A flip – flop is a synchronous device.
Latches are transparent devices i.e. when they are enabled, the output changes immediately if the input changes.	A transition from low to high or high to low of the clock signal will cause the flip – flop to either change its output or retain it depending on the input signal.
A latch is a Level Sensitive device (Level Triggering is involved).	A flip – flop is an edge sensitive device (Edge Triggering is involved).
Latches are simpler to design as there is no clock signal (no careful routing of clock signal is required).	When compare to latches, flip – flops are more complex to design as they have clock signal and it has to be carefully routed. This is because all the flip – flops in a design should have a clock signal and the delay in the clock reaching each flip – flop must be minimum or negligible.
The operation of a latch is faster as they do not have to wait for any clock signal.	Flip - flops are comparatively slower than latches due to clock signal.
The power requirement of a latch is less.	Power requirement of a flip – flop is more.
A latch works based on the enable signal.	A flip – flop works based on the clock signal.

Types of flip flops

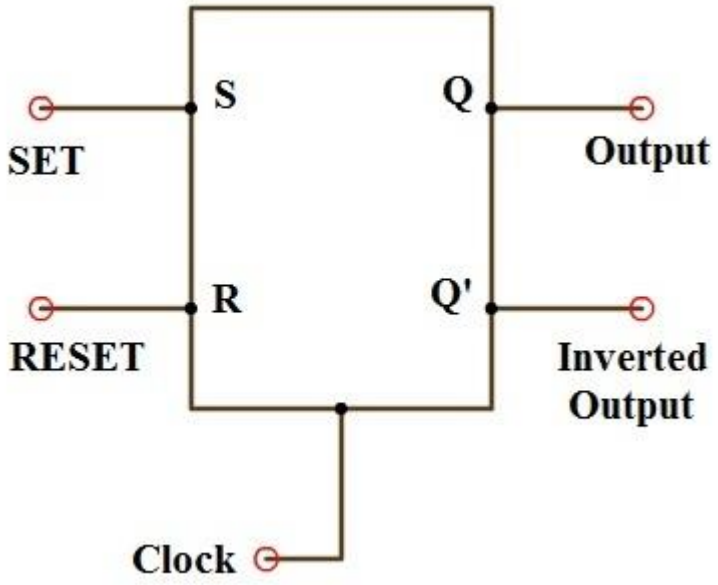
Based on their operations, flip flops are basically 4 types. They are

1. R-S flip flop
2. D flip flop
3. J-K flip flop
4. T flip flop

S-R Flip Flop

The S-R flip-flop is basic flip-flop among all the flip-flops. All the other flip flops are developed after SR-flip-flop.

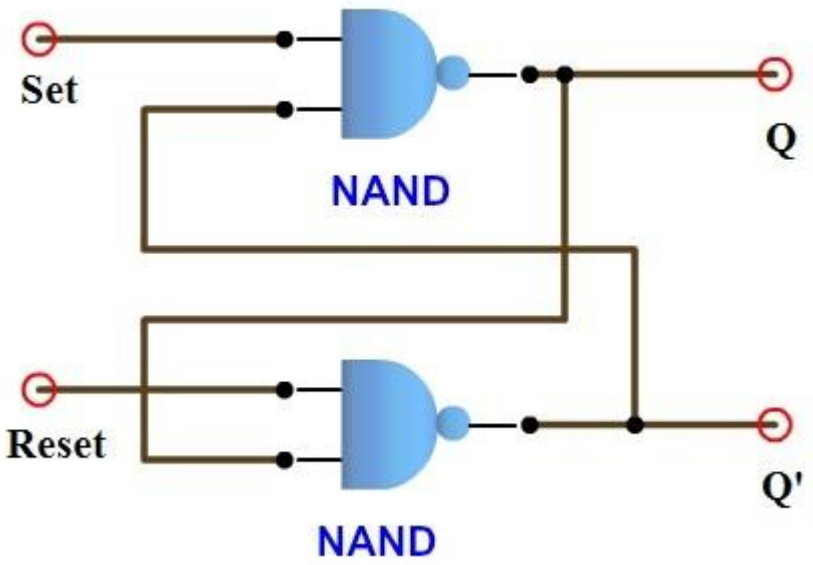
SR flip flop is represented as shown below.



S-R stands for SET and RESET. This can also be called RS flip-flop. Difference is RS is inverted SR flip-flop.

Any flip flop can be build using logic gates. NAND and NOR gates were used as they are universal gates.

Here is the SR flip-flop using NAND gates.



Truth Table of SR Flip Flop

Sno	S	R	Q	Q'	State
1	1	0	1	0	Q is set to 1
2	1	1	1	0	No change
3	0	1	0	1	Q' is set to 1
4	1	1	0	1	No change
5	0	0	1	1	Invalid

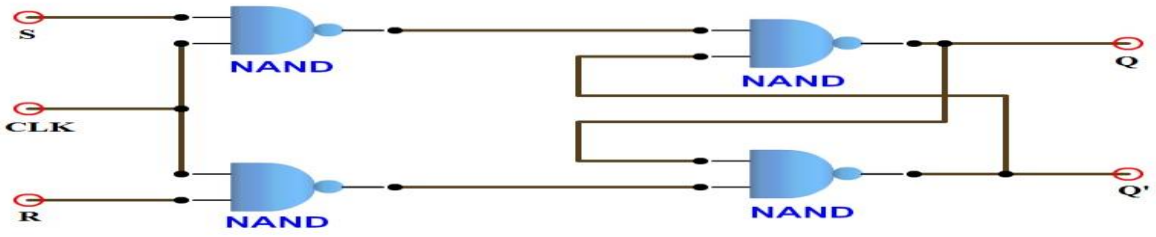
Working

From the above truth table it is clear that SR flip flop will be set or reset for four conditions.

1. For last condition it will be in invalid state.
2. SR Flip-flop will be set when S=1 and R=0, if S=1 and R=1 then previous state is remembered by the flip flop.
3. Flip-flop will be reset when S=0 and R=1, if S=1 and R=1, then it will remember the previous state.
4. But when both the inputs are zeros, SR Flip flop will be in an uncertain state where both Q and Q' will be same. This is not same allowed..

This is indeterminate state is avoided by adding gates extra gates to the existing flip flop. This is called clocked or gated SR Flip flop. This produces the output only for the High clock pulse.

The circuit of a clocked SR flip – flop using NAND gates is shown below.

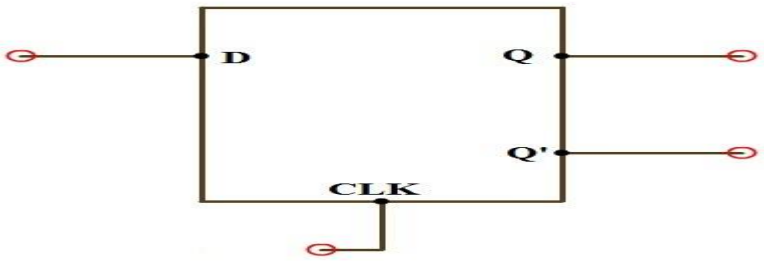


Know in detail about SR Flip FlopD flip flop

D flip flop

In the SR flip flop an uncertain state occurred. This can be avoided by using D flip flop. Here D stands for "Data".
It is constructed from SR flip flop. The two inputs (S &R) of the clocked SR flip flop are connected to an inverter.

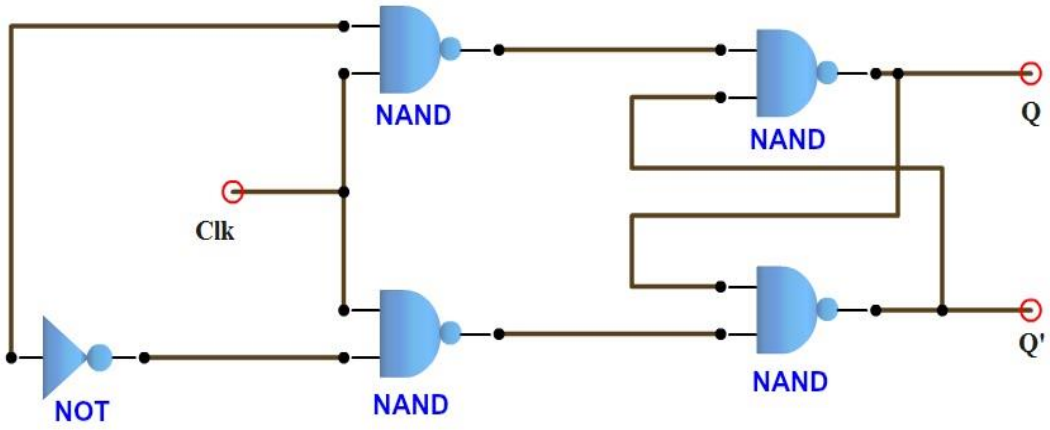
It is one of the most widely used flip – flops. It has a clock signal (Clk) as one input and Data (D) as other. There are two outputs and these outputs are complement to each other. The symbol of D flip – flop is shown below.



Truth table

Clk	D	Q	Q'	State
0	0	Q	Q'	No change in state
1	0	0	1	Resets Q to 0
1	1	1	1	Sets Q to 1

D flip – flop using NAND gates is shown below.



Working

- D flip flop will work depending on the clock signal.
- When the clock is low there will be no change in the output of the flip flop i.e. it remembers the previous state.
- When the clock signal is high and if it receives any data on its data pin, it Changes the state of output.
- When data is high Q reset to 0,while Q is set to 0 if data is low.

A master slave D flip flop can be constructed using D-flip flop.

Know in detail about D-flip flop.

J-K Flip Flop

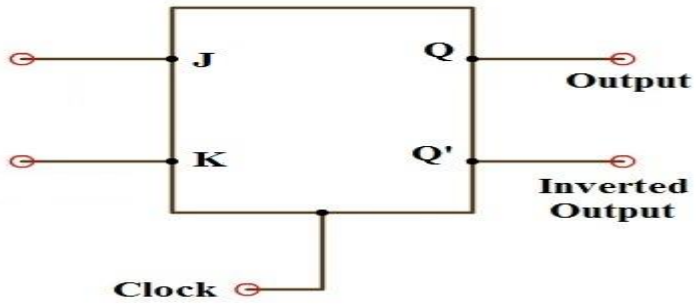
JK flip – flop is named after Jack Kilby, an electrical engineer who invented IC.

A JK flip – flop is a modification of SR flip – flop. In this the J input is similar to the set input of SR flip – flop and the K input is similar to the reset input of SR flip – flop. The condition J = K = 1 which is not allowed in SR flip – flop (S = R = 1) is interpreted as a toggle command.

The JK flip flop has

- Two data inputs J and K.
- One clock signal input (CLK).
- Two outputs Q and Q'.

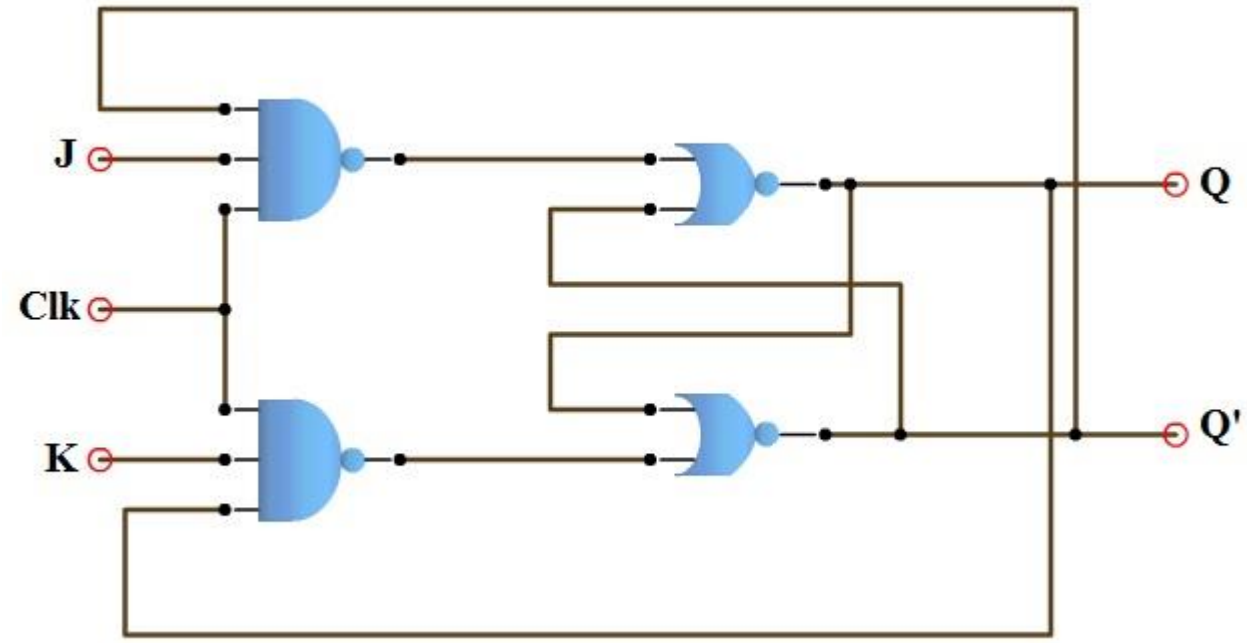
The symbol of a JK flip – flop is shown below.



Truth Table

Clk	J	K	Q	Q'	State
1	0	0	Q	Q'	No change in state
1	0	1	0	1	Resets Q to 0
1	1	0	1	0	Sets Q to 1
1	1	1	-	-	Toggles

The circuit of a JK flip – flop using gates is shown below. It is similar to a modified NAND SR flip – flop.



Working

- When J is low and K is low, then Q returns its previous state value i.e. it holds the current state.
- When J is low and K is high, then flip – flop will be in reset state i.e. Q = 0, Q' =1.
- When J is high and K is low then flip – flop will be in set state i.e. Q = 1, Q' =0.
- When J is high and K is high then flip – flop will be in Toggle state or flip state. This means that the output will complement to the previous state value.

To Know in detail about JK Flip Flop

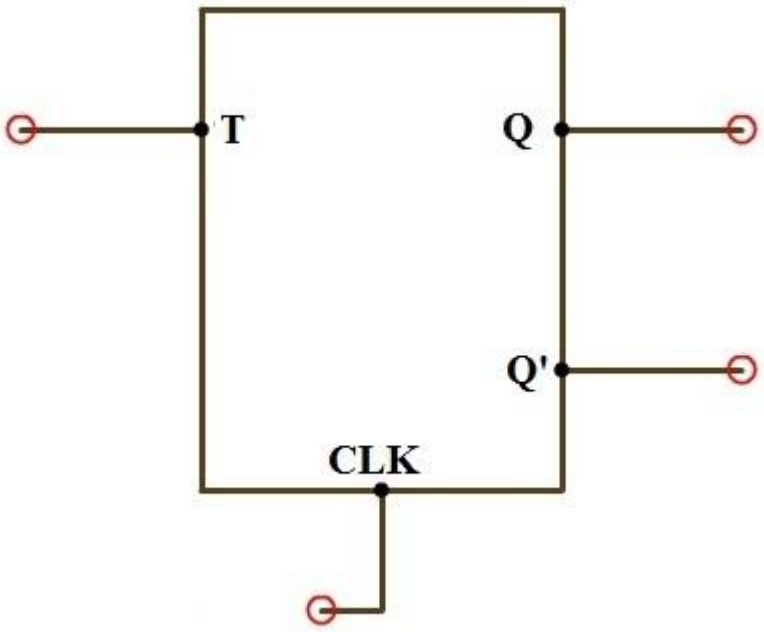
T Flip Flop

T flip flop is also known as "Toggle Flip – flop". Toggle is to change the output to complement of the previous state in the presence of clock input signal.

The T flip flop has

- T input.
- One clock signal input (CLK).
- Two outputs Q and Q'.

The symbol of a T flip – flop is shown below.



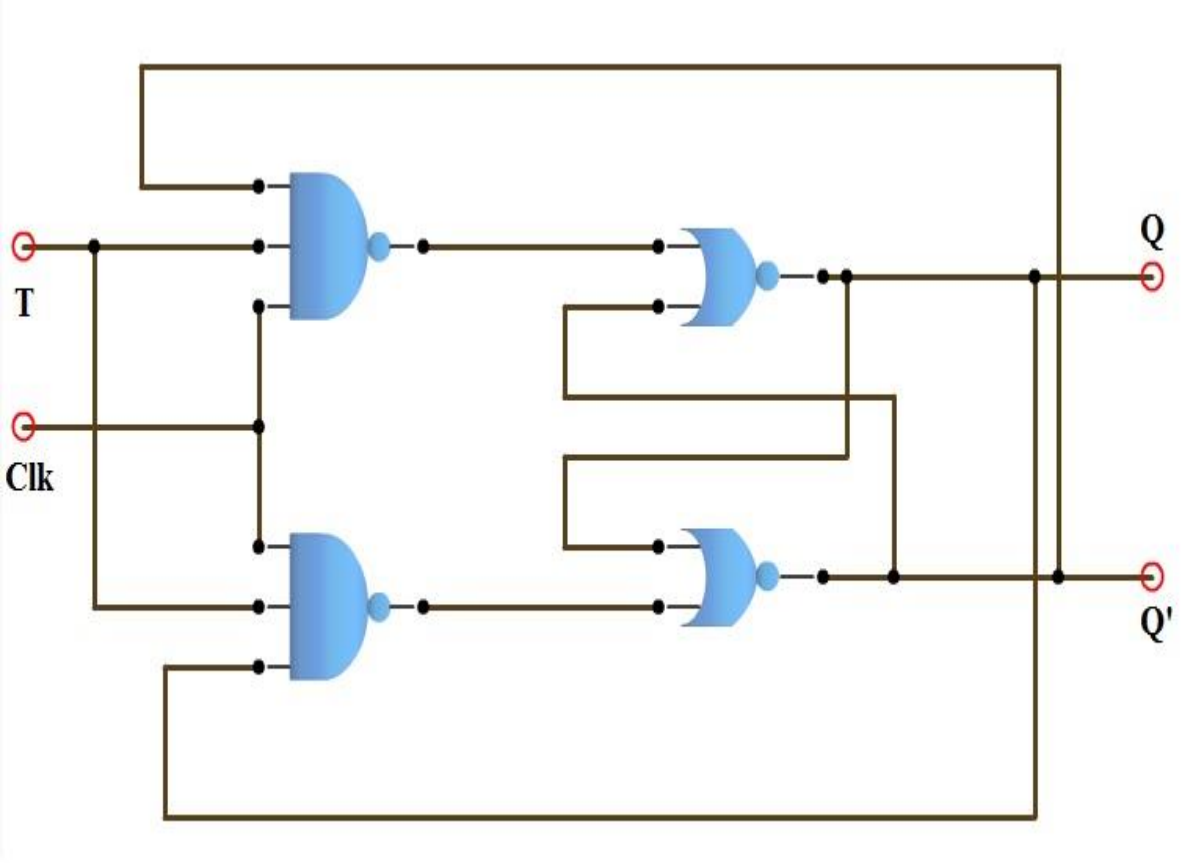
We can construct a T flip – flop by using any other flip – flops.

- SR flip – flop: By connecting the feedback of outputs of SR flip – flop to the inputs (S & R).
- D flip – flop: Connecting the Q' to its Data input of D flip – flop as feedback path.
- J K flip – flop: By combing the J & K inputs of JK flip – flop, to make as single input, we can design the T flip – flop.

Truth Table

T	Q	Q'
0	0	0
1	0	1
0	1	0
1	1	0

The circuit of a T flip – flop made from NAND JK flip – flop is shown below.



Working

The operation of the T flip flop is explained below.

When the T input is low, then the next sate of the T flip – flop is same as the present state i.e. it holds the current state.

- T = 0 and present state = 0 then the next state = 0.
- T = 0 and present state = 1 then the next state = 1.

When the T input is high, then the next sate of the T flip – flop is toggled i.e. it is same as the complement of present state on clock transition.

- T = 1 and present state = 0 then the next state = 1.
- T = 1 and present state = 1 then the next state = 0.

Know in detail about T-Flip flop

WHERE WE USE FLIP FLOPS??

Flip flops are widely used in

- Registers: As the flip flops have two stable states, we use them in memory elements like registers, for data storage. Generally we use registers in electronic devices like computers.

- Counters: The groups of interconnected flip flops are uses as counters, to count the increment or decrement of an event occurrence.
- Frequency division: Flip flops are used as frequency division circuits, which divide the input frequency to exactly to its half. Frequency division circuits are used to regularize the frequency of electronic circuits.
- Data transfer: We use shift registers (A special-type of registers) to transfer the data from one flip flop to another, which are connected in a specific order.

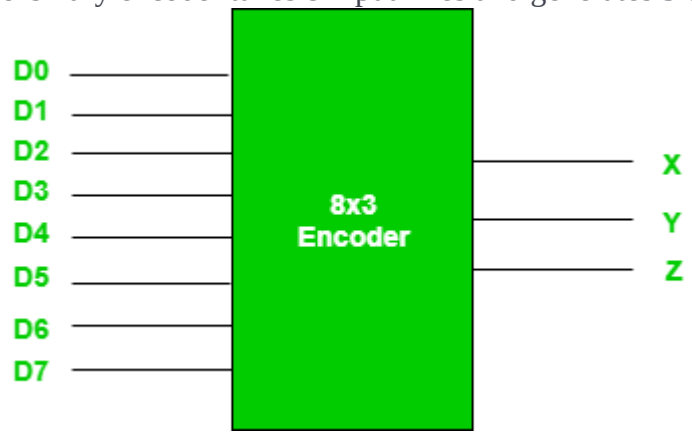
17.Encoders and Decoders in Digital Logic

Binary code of N digits can be used to store 2^N distinct elements of coded information. This is what encoders and decoders are used for. **Encoders** convert 2^N lines of input into a code of N bits and **Decoders** decode the N bits into 2^N lines.

1. Encoders –

An encoder is a combinational circuit that converts binary information in the form of a 2^N input lines into N output lines, which represent N bit code for the input. For simple encoders, it is assumed that only one input line is active at a time.

As an example, let's consider **Octal to Binary** encoder. As shown in the following figure, an octal-to-binary encoder takes 8 input lines and generates 3 output lines.



Truth Table –

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

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

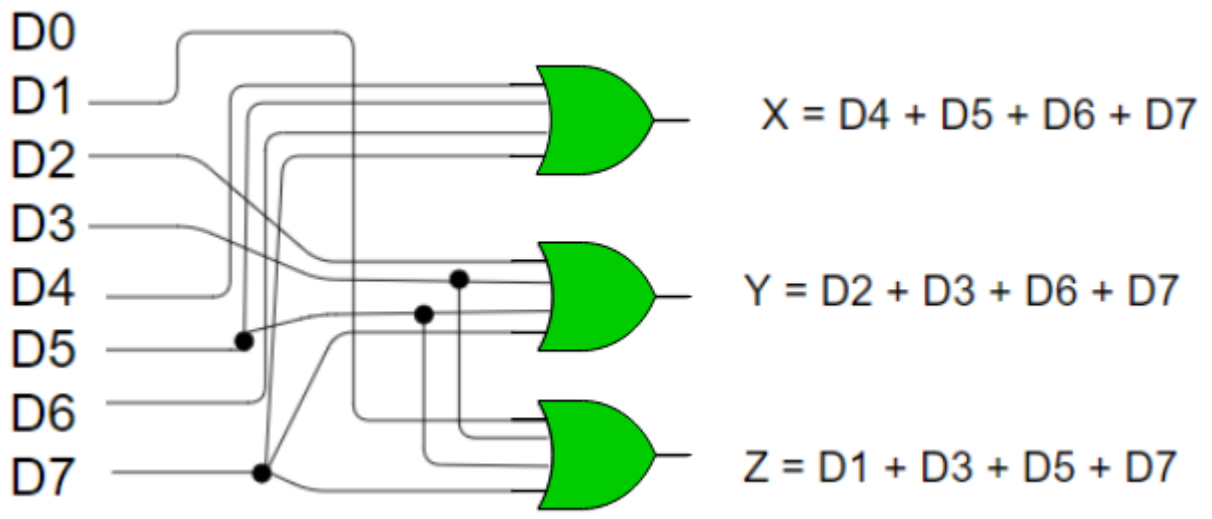
As seen from the truth table, the output is 000 when D0 is active; 001 when D1 is active; 010 when D2 is active and so on.

Implementation –
From the truth table, the output line Z is active when the input octal digit is 1, 3, 5 or 7. Similarly, Y is 1 when input octal digit is 2, 3, 6 or 7 and X is 1 for input octal digits 4, 5, 6 or 7. Hence, the Boolean functions would be:
X = D4 + D5 + D6 + D7

Y = D2 +D3 + D6 + D7

Z = D1 + D3 + D5 + D7

Hence, the encoder can be realised with OR gates as follows:



One limitation of this encoder is that only one input can be active at any given time. If m

One limitation of this encoder is that only one input can be active at any given time. If more than one inputs are active, then the output is undefined. For example, if D6 and D3 are both active, then, our output would be 111 which is the output for D7. To overcome this, we use Priority Encoders.

Another ambiguity arises when all inputs are 0. In this case, encoder outputs 000 which actually is the output for D0 active. In order to avoid this, an extra bit can be added to the output, called the valid bit which is 0 when all inputs are 0 and 1 otherwise.

Priority Encoder –
A priority encoder is an encoder circuit in which inputs are given priorities. When more than one inputs are active at the same time, the input with higher priority takes precedence and the output corresponding to that is generated.
Let us consider the 4 to 2 priority encoder as an example.
From the truth table, we see that when all inputs are 0, our V bit or the valid bit is zero and outputs are not used. The x's in the table show the don't care condition, i.e, it may either be 0 or 1. Here, D3 has highest priority, therefore, whatever be the other inputs, when D3 is high, output has to be 11. And D0 has the lowest priority, therefore the output would be 00 only when D0 is high and the other input lines are low. Similarly, D2 has higher priority over D1 and D0 but lower than D3 therefore the output would be 010 only when D2 is high and D3 are low (D0 & D1 are don't care).

Truth Table –

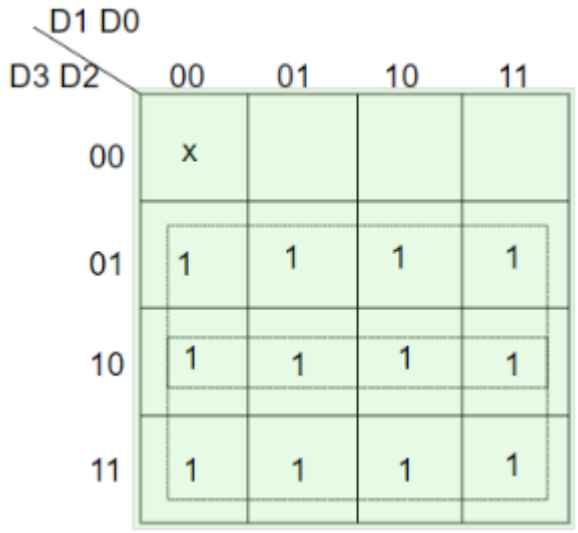
D3	D2	D1	D0	X	Y	V
0	0	0	0	x	x	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Implementation –

It can clearly be seen that the condition for valid bit to be 1 is that at least any one of the inputs should be high. Hence,

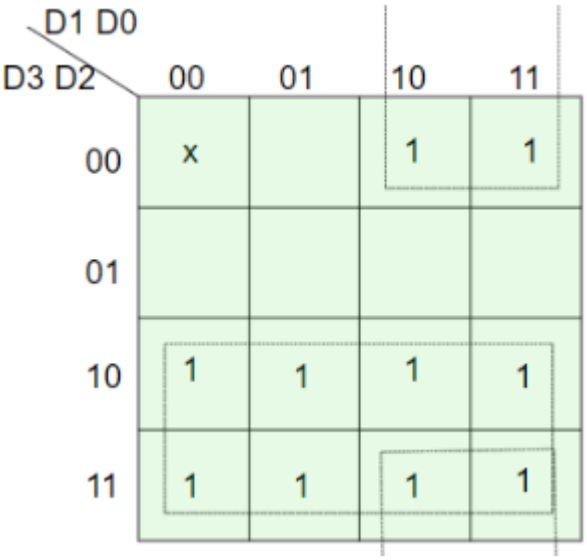
V = D0 + D1 + D2 + D3

For X:



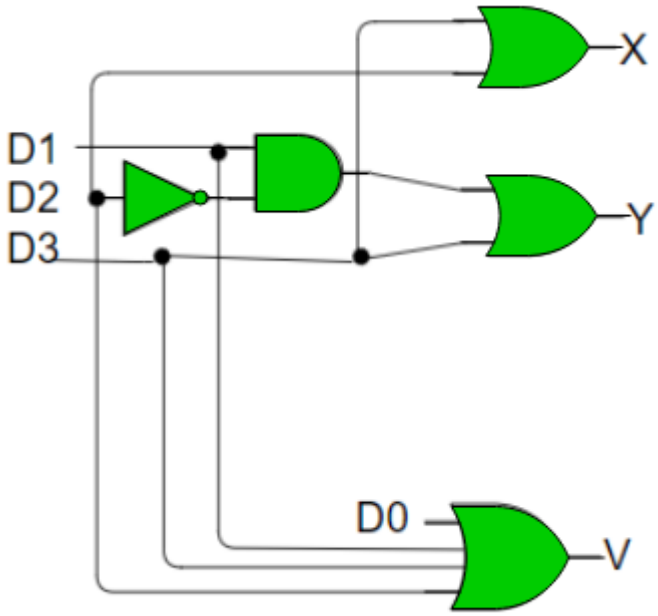
=> X = D2 + D3

For Y:



=> Y = D1 D2’ + D3

Hence, the priority 4-to-2 encoder can be realized as follows:



2. 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

X	Y	Z	D0	D1	D2	D3	D4	D5	D6	D7
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

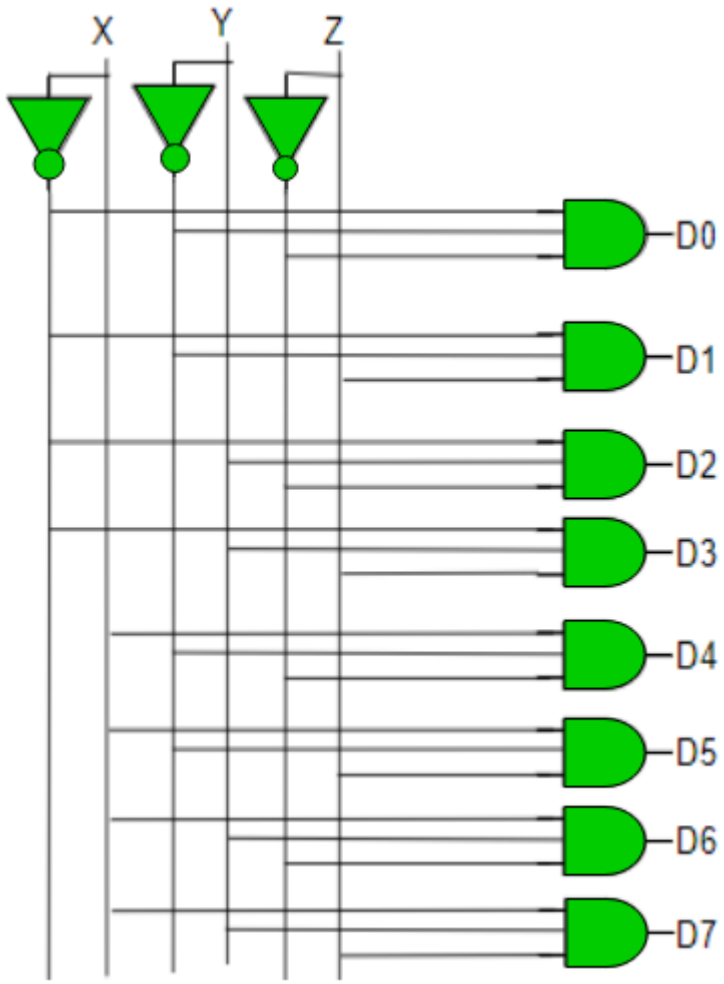
D4 = X Y’ Z’

D5 = X Y’ Z

D6 = X Y Z’

D7 = X Y Z

Hence,



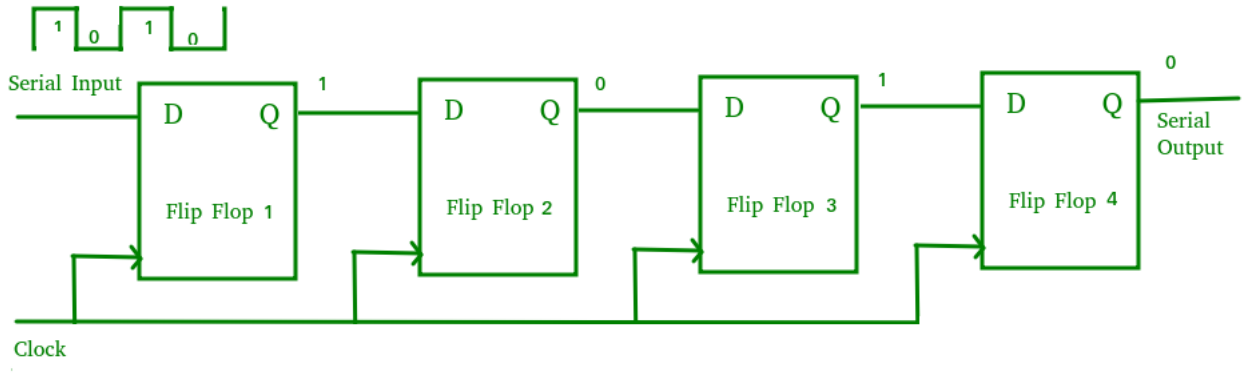
18.Shift Registers in Digital Logic Flip flops can be used to store a single bit of binary data (1or 0). However, in order to store multiple bits of data, we need multiple flip flops. N flip flops are to be connected in an order to store n bits of data. A **Register** is a device which is used to store such information. It is a group of flip flops connected in series used to store multiple bits of data.
The information stored within these registers can be transferred with the help of **shift registers**. Shift Register is a group of flip flops used to store multiple bits of data. The bits stored in such registers can be made to move within the registers and in/out of the registers by applying clock pulses. An n-bit shift register can be formed by connecting n flip-flops where each flip flop stores a single bit of data.
The registers which will shift the bits to left are called “Shift left registers”.
The registers which will shift the bits to right are called “Shift right registers”.
Shift registers are basically of 4 types. These are:

1. Serial In Serial Out shift register
2. Serial In parallel Out shift register
3. Parallel In Serial Out shift register
4. Parallel In parallel Out shift register

Serial-In Serial-Out Shift Register (SISO) –

The shift register, which allows serial input (one bit after the other through a single data line) and produces a serial output is known as Serial-In Serial-Out shift register. Since there is only one output, the data leaves the shift register one bit at a time in a serial pattern, thus the name Serial-In Serial-Out Shift Register.

The logic circuit given below shows a serial-in serial-out shift register. The circuit consists of four D flip-flops which are connected in a serial manner. All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.

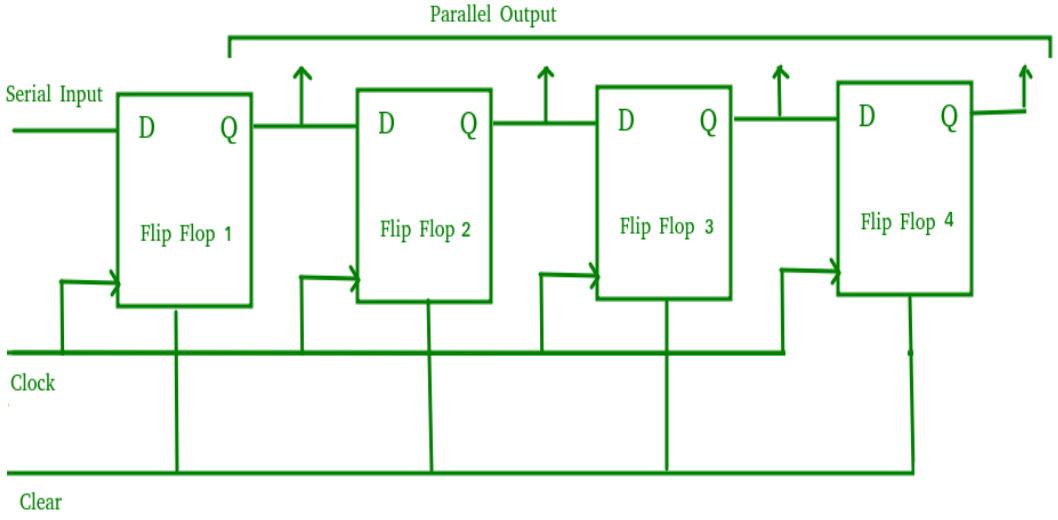


The above circuit is an example of shift right register, taking the serial data input from the left side of the flip flop. The main use of a SISO is to act as a delay element.

Serial-In Parallel-Out shift Register (SIPO) –

The shift register, which allows serial input (one bit after the other through a single data line) and produces a parallel output is known as Serial-In Parallel-Out shift register.

The logic circuit given below shows a serial-in-parallel-out shift register. The circuit consists of four D flip-flops which are connected. The clear (CLR) signal is connected in addition to the clock signal to all the 4 flip flops in order to RESET them. The output of the first flip flop is connected to the input of the next flip flop and so on. All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.

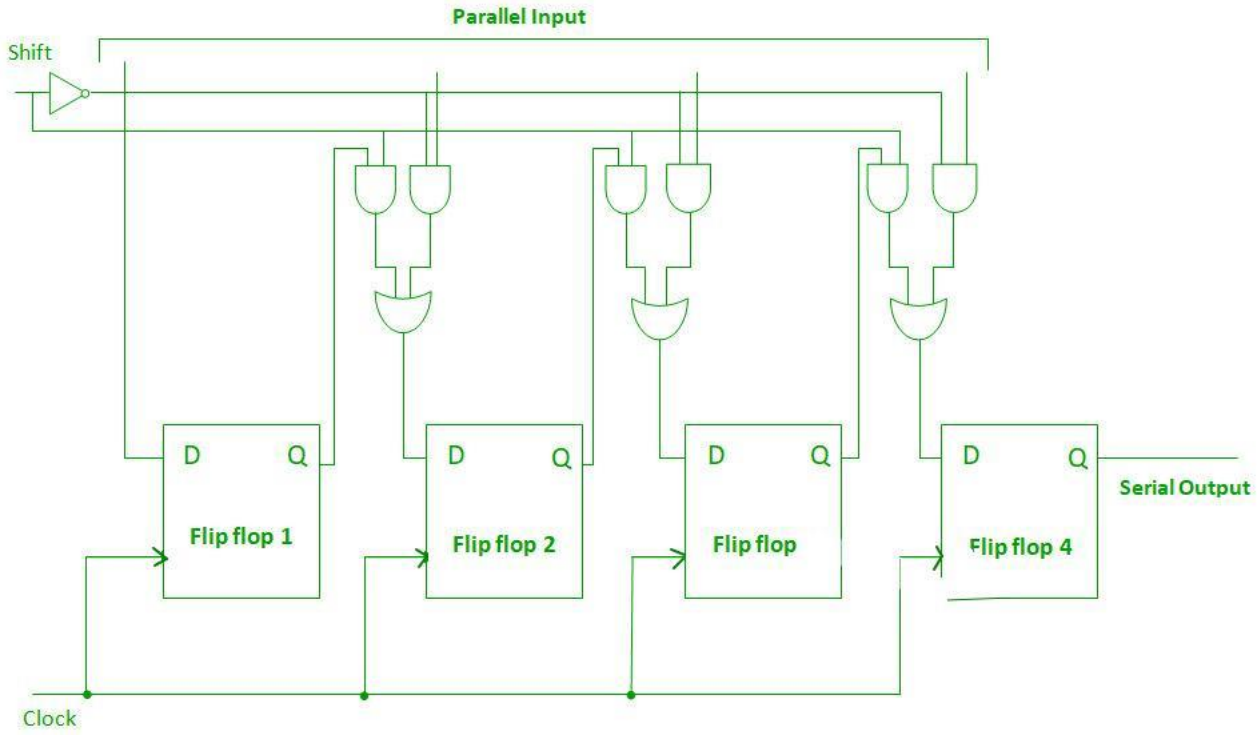


The above circuit is an example of shift right register, taking the serial data input from the left side of the flip flop and producing a parallel output. They are used in communication lines where demultiplexing of a data line into several parallel lines is required because the main use of the SIPO register is to convert serial data into parallel data.

Parallel-In Serial-Out Shift Register (PISO) –

The shift register, which allows parallel input (data is given separately to each flip flop and in a simultaneous manner) and produces a serial output is known as Parallel-In Serial-Out shift register.

The logic circuit given below shows a parallel-in-serial-out shift register. The circuit consists of four D flip-flops which are connected. The clock input is directly connected to all the flip flops but the input data is connected individually to each flip flop through a multiplexer at the input of every flip flop. The output of the previous flip flop and parallel data input are connected to the input of the MUX and the output of MUX is connected to the next flip flop. All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop.

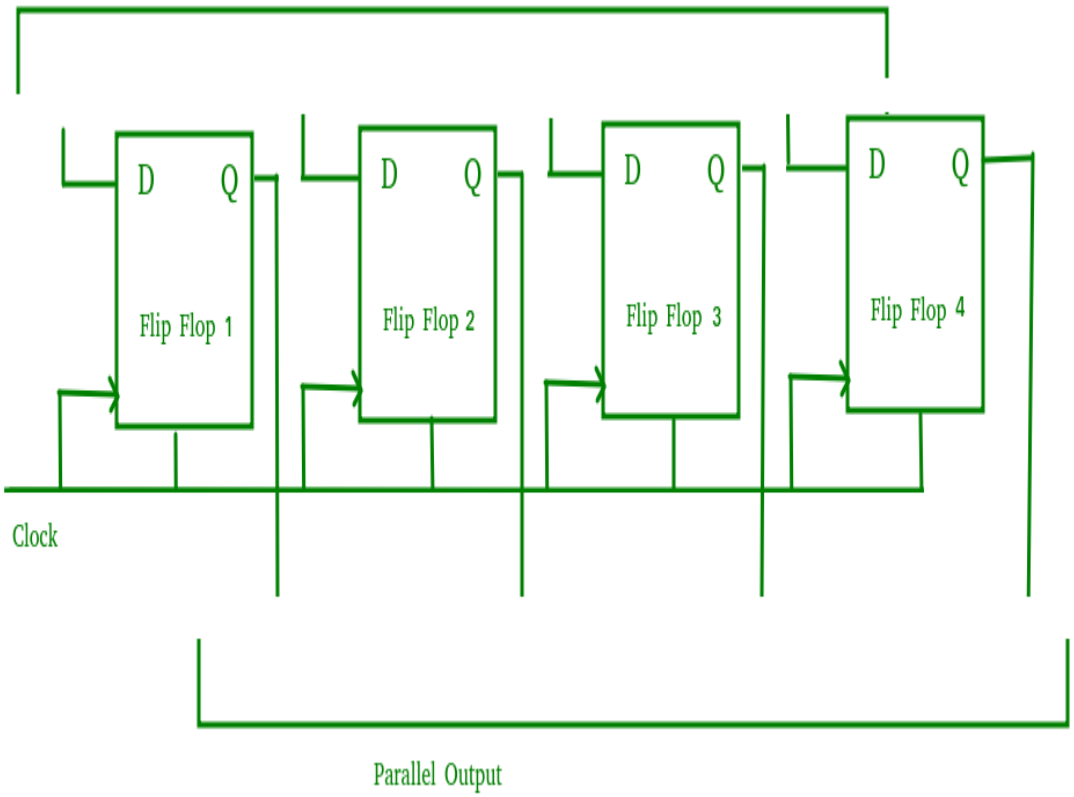


A Parallel in Serial out (PISO) shift register us used to convert parallel data to serial data.

Parallel-In Parallel-Out Shift Register (PIPO) –

The shift register, which allows parallel input (data is given separately to each flip flop and in a simultaneous manner) and also produces a parallel output is known as Parallel-In parallel-Out shift register.

The logic circuit given below shows a parallel-in-parallel-out shift register. The circuit consists of four D flip-flops which are connected. The clear (CLR) signal and clock signals are connected to all the 4 flip flops. In this type of register, there are no interconnections between the individual flip-flops since no serial shifting of the data is required. Data is given as input separately for each flip flop and in the same way, output also collected individually from each flip flop.



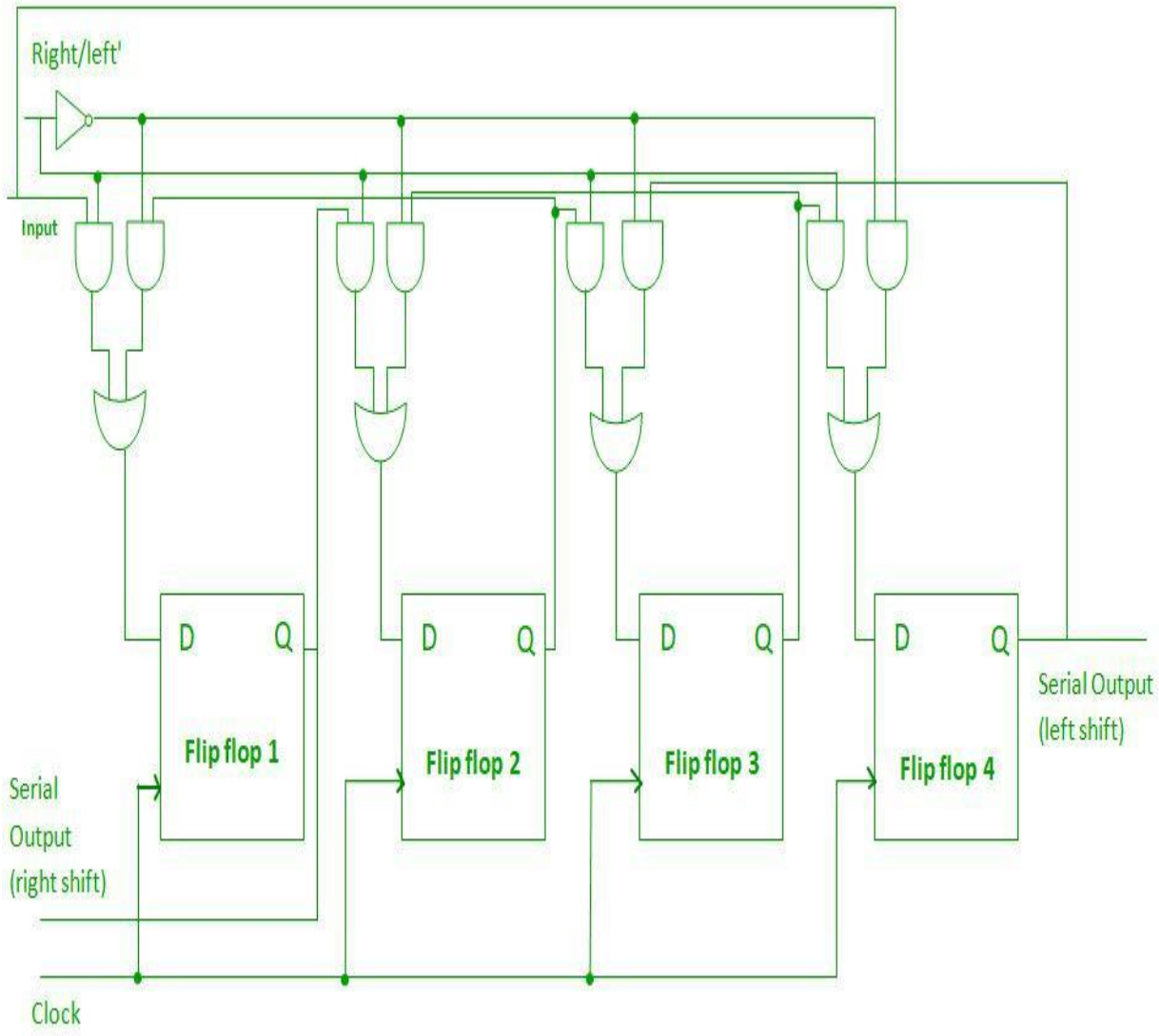
A Parallel in Parallel out (PIPO) shift register is used as a temporary storage device and like SISO Shift register it acts as a delay element.

Bidirectional Shift Register –

If we shift a binary number to the left by one position, it is equivalent to multiplying the number by 2 and if we shift a binary number to the right by one position, it is equivalent to dividing the number by 2.To perform these operations we need a register which can shift the data in either direction.

Bidirectional shift registers are the registers which are capable of shifting the data either right or left depending on the mode selected. If the mode selected is 1(high), the data will be shifted towards the right direction and if the mode selected is 0(low), the data will be shifted towards the left direction.

The logic circuit given below shows a Bidirectional shift register. The circuit consists of four D flip-flops which are connected. The input data is connected at two ends of the circuit and depending on the mode selected only one and gate is in the active state.



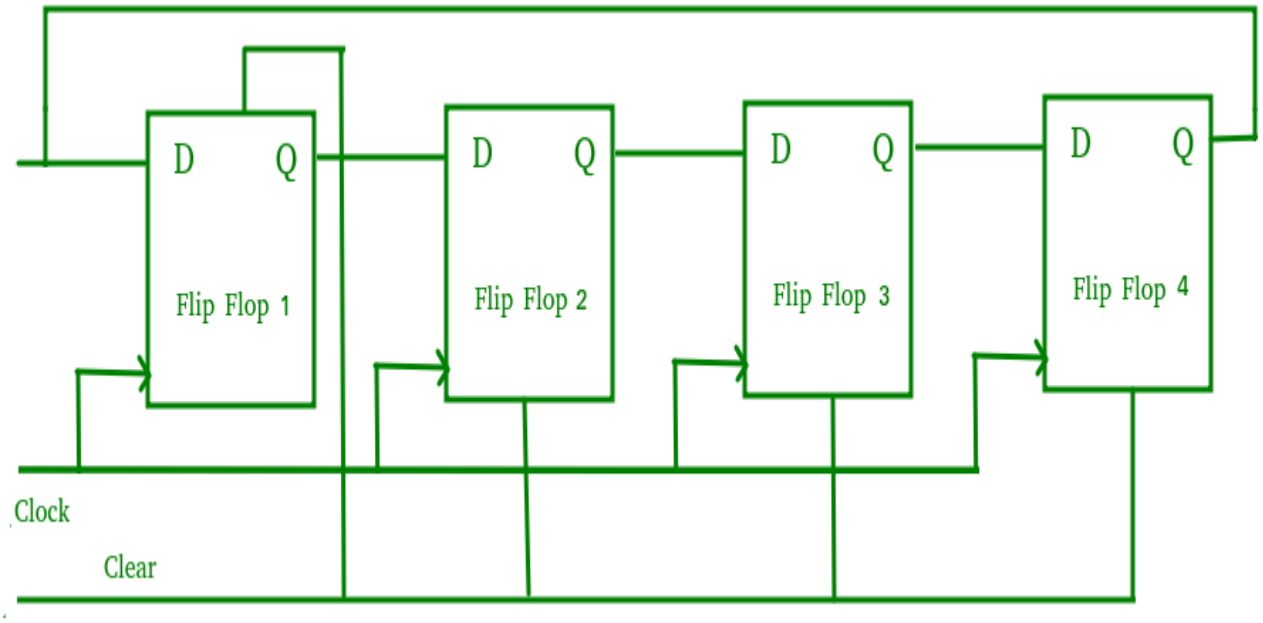
Shift Register Counter –

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

1. **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, thus the name ring counter. The data pattern within the shift register will circulate as long as clock pulses are applied.

   The logic circuit given below shows a Ring Counter. The circuit consists of four D flip-flops which are connected. Since the circuit consists of four flip flops the data pattern will repeat after every four clock pulses as shown in the truth table below:

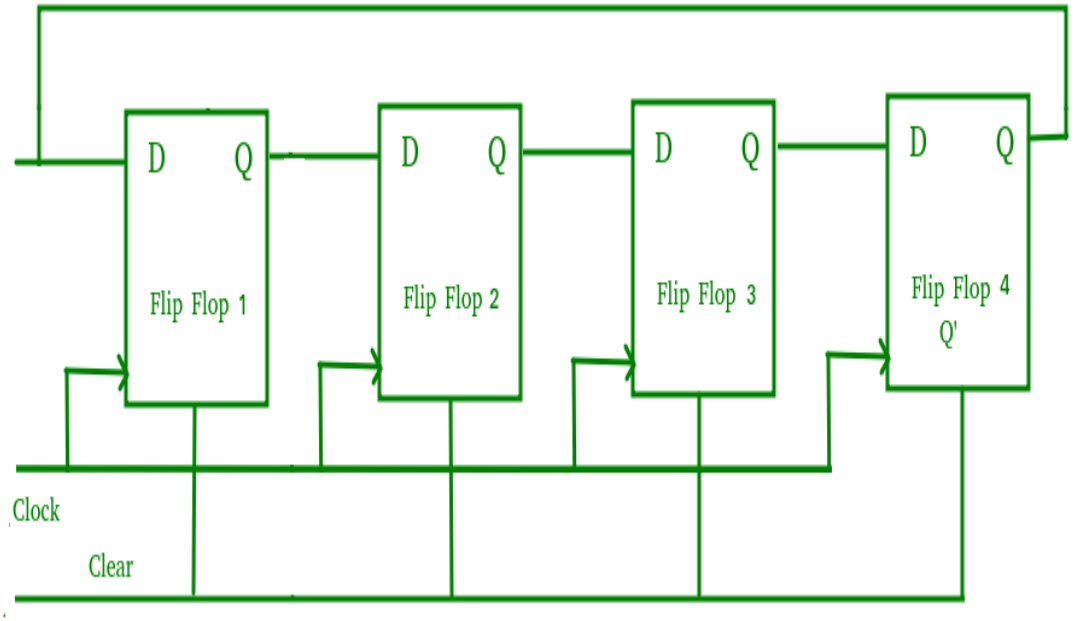
Clock Pulse	Q1	Q2	Q3	Q4
0	1	0	0	1
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1



A Ring counter is generally used because it is self-decoding. No extra decoding circuit is needed to determine what state the counter is in.

2. **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.
The logic circuit given below shows a Johnson Counter. The circuit consists of four D flip-flops which are connected. An n-stage Johnson counter yields a count sequence of 2n different states, thus also known as a mod-2n counter. Since the circuit consists of four flip flops the data pattern will repeat every eight clock pulses as shown in the truth table below:

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



The main advantage of Johnson counter is that it only needs n number of flip-flops compared to the ring counter to circulate a given data to generate a sequence of 2n states.

Applications of shift Registers –

- The shift registers are used for temporary data storage.
- The shift registers are also used for data transfer and data manipulation.

- The serial-in serial-out and parallel-in parallel-out shift registers are used to produce time delay to digital circuits.
- The serial-in parallel-out shift register is used to convert serial data into parallel data thus they are used in communication lines where demultiplexing of a data line into several parallel line is required.
- A Parallel in Serial out shift register us used to convert parallel data to serial data.

19.Counters in Digital Logic

According to Wikipedia, in digital logic and computing, a **Counter** is a device which stores (and sometimes displays) the number of times a particular event or process has occurred, often in relationship to a clock signal. Counters are used in digital electronics for counting purpose, they can count specific event happening in the circuit. For example, in UP counter a counter increases count for every rising edge of clock. Not only counting, a counter can follow the certain sequence based on our design like any random sequence 0,1,3,2… .They can also be designed with the help of flip flops.

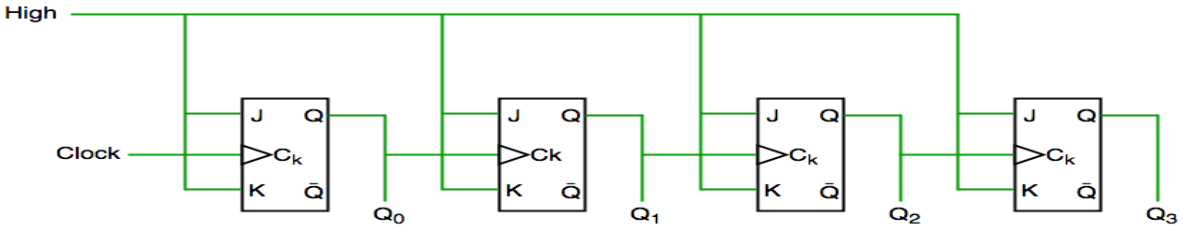
Counter Classification

Counters are broadly divided into two categories

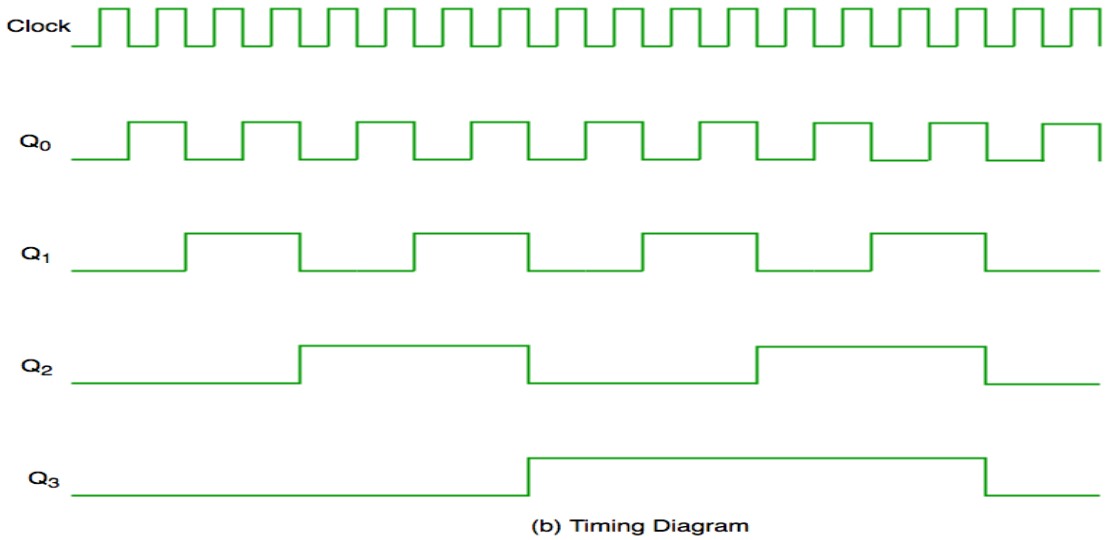
1. Asynchronous counter
2. Synchronous counter

1. **Asynchronous Counter**

In asynchronous counter we don't use universal clock, only first flip flop is driven by main clock and the clock input of rest of the following flip flop is driven by output of previous flip flops. We can understand it by following diagram-



(a) Asynchronous counter



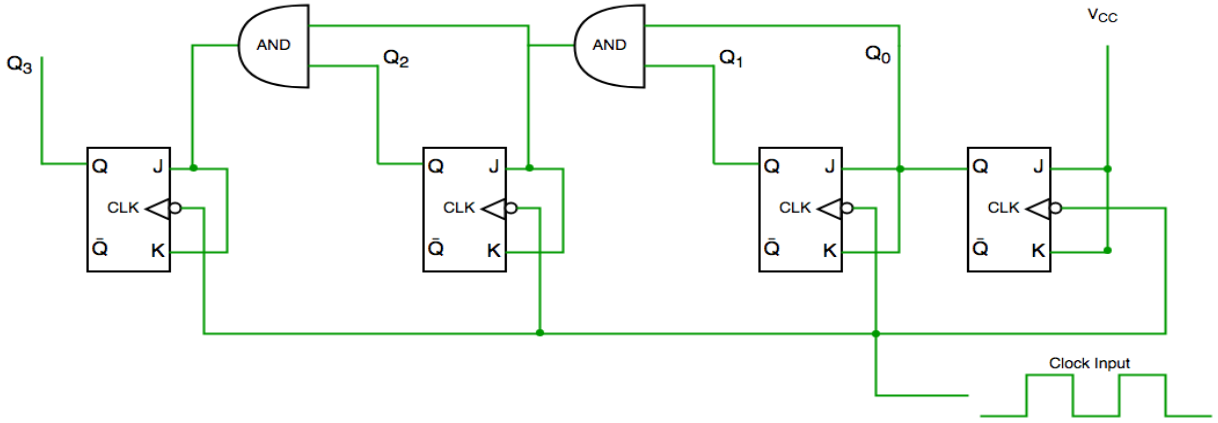
(b) Timing Diagram

It is evident from timing diagram that Q0 is changing as soon as the rising edge of clock pulse is encountered, Q1 is changing when rising edge of Q0 is encountered(because Q0 is like clock pulse

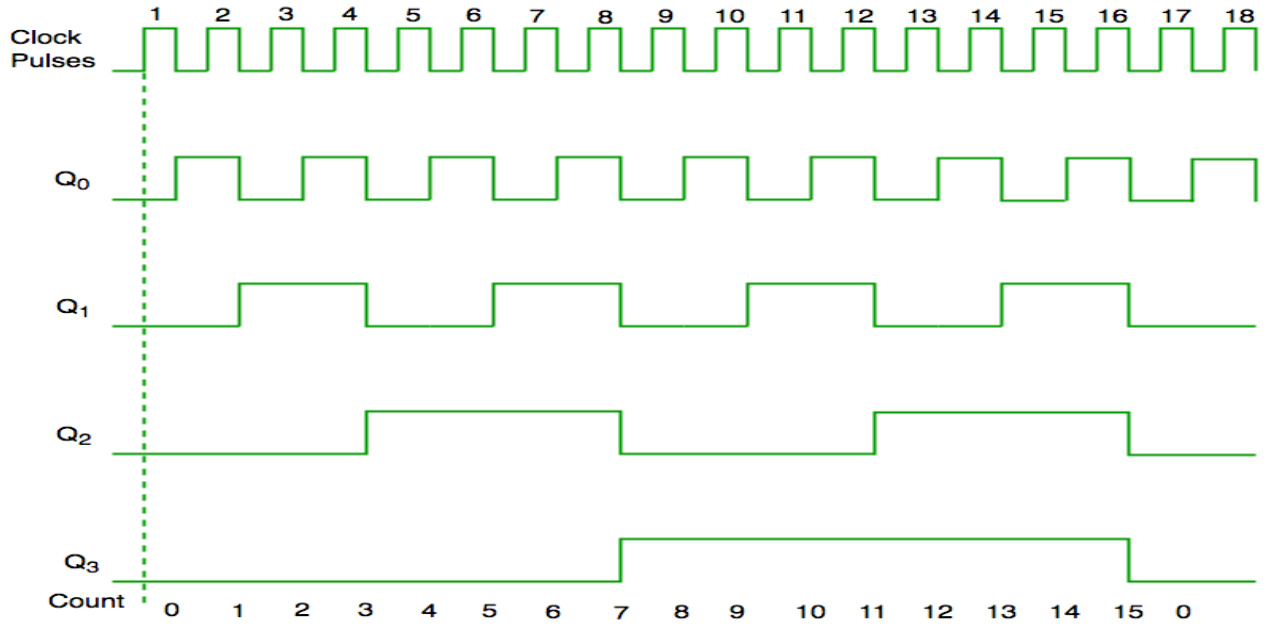
for second flip flop) and so on. In this way ripples are generated through Q0,Q1,Q2,Q3 hence it is also called **RIPPLE counter.**

2. **Synchronous Counter**

Unlike the asynchronous counter, synchronous counter has one global clock which drives each flip flop so output changes in parallel. The one advantage of synchronous counter over asynchronous counter is, it can operate on higher frequency than asynchronous counter as it does not have cumulative delay because of same clock is given to each flip flop.



Synchronous counter circuit



Timing diagram synchronous counter

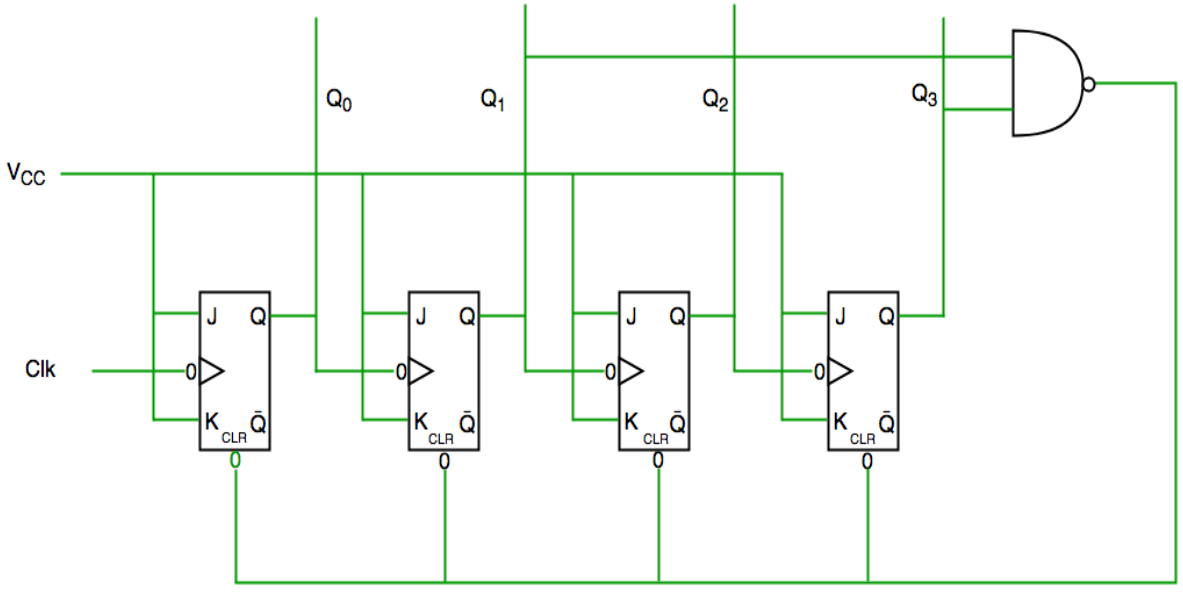
From circuit diagram we see that Q0 bit gives response to each falling edge of clock while Q1 is dependent on Q0, Q2 is dependent on Q1 and Q0 , Q3 is dependent on Q2,Q1 and Q0.

Decade Counter

A decade counter counts ten different states and then reset to its initial states. A simple decade counter will count from 0 to 9 but we can also make the decade counters which can go through any ten states between 0 to 15(for 4 bit counter).

Clock pulse	Q3	Q2	Q1	Q0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	0	0	0	0

Truth table for simple decade counter



Decade counter circuit diagram

We see from circuit diagram that we have used nand gate for Q3 and Q1 and feeding this to clear input line because binary representation of 10 is—

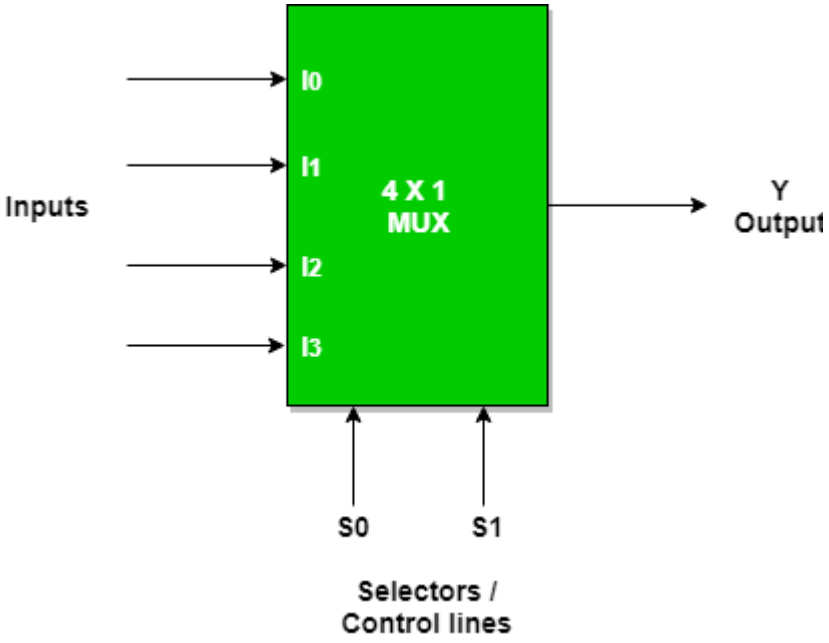
1010

And we see Q3 and Q1 are 1 here, if we give NAND of these two bits to clear input then counter will be clear at 10 and again start from beginning.

Important point: Number of flip flops used in counter are always greater than equal to (**$\log_2 n$**) where n=number of states in counter.

20.Multiplexers in Digital Logic

It is a combinational circuit which have many data inputs and single output depending on control or select inputs. For N input lines, log n (base2) selection lines, or we can say that for 2^n input lines, n selection lines are required. Multiplexers are also known as **"Data n selector, parallel to serial convertor, many to one circuit, universal logic circuit"**. Multiplexers are mainly used to increase amount of the data that can be sent over the network within certain amount of time and bandwidth.



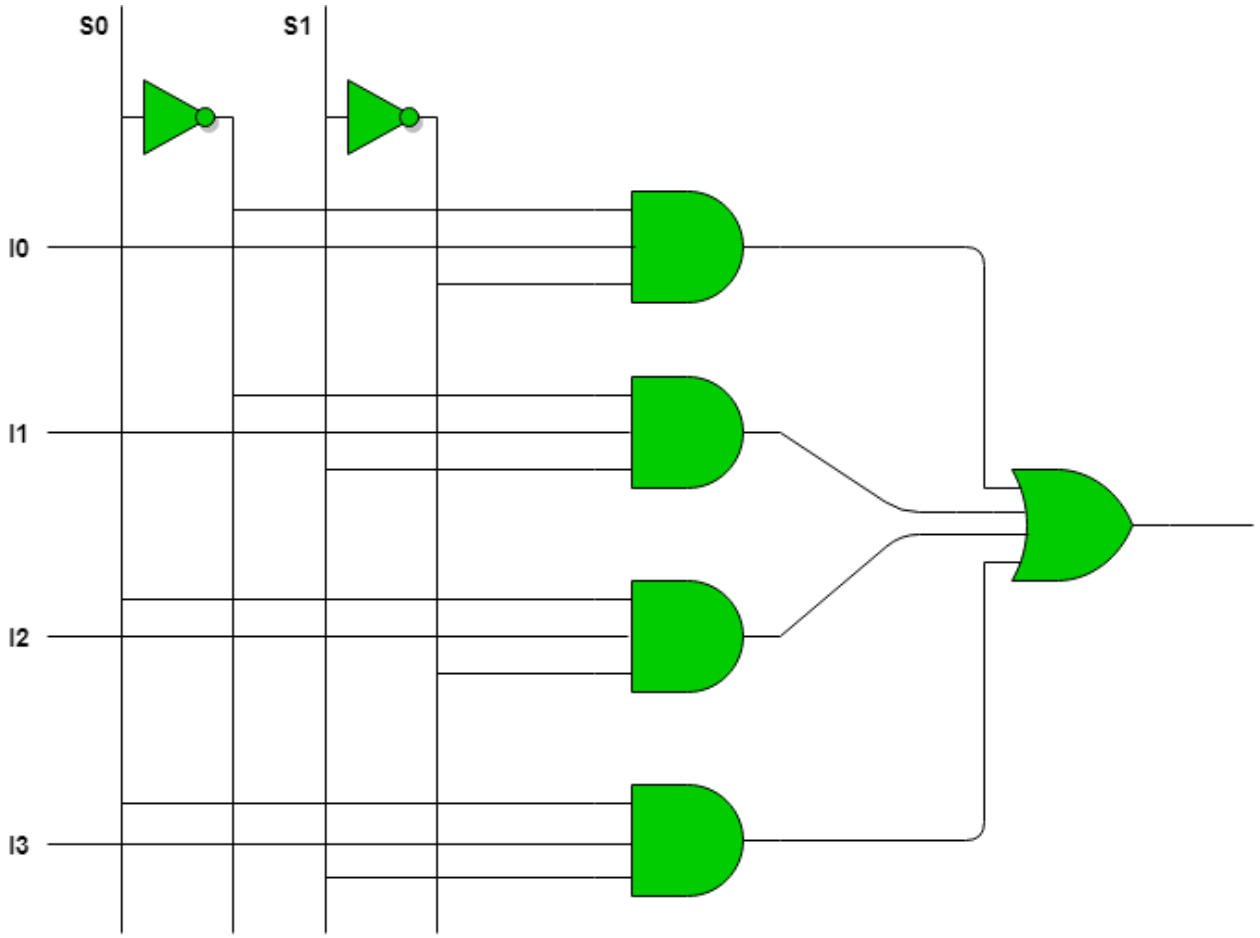
Now the implementation of 4:1 Multiplexer using truth table and gates.

Truth Table

S0	S1	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

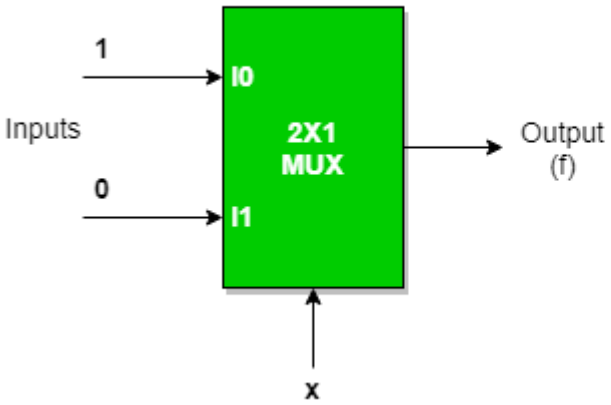
So, final equation,

Y = S0'.S1'.I0 + S0'.S1.I1 + S0.S1'.I2 + S0.S1.I3



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

a) Implementation of NOT gate using 2 : 1 Mux
NOT Gate :

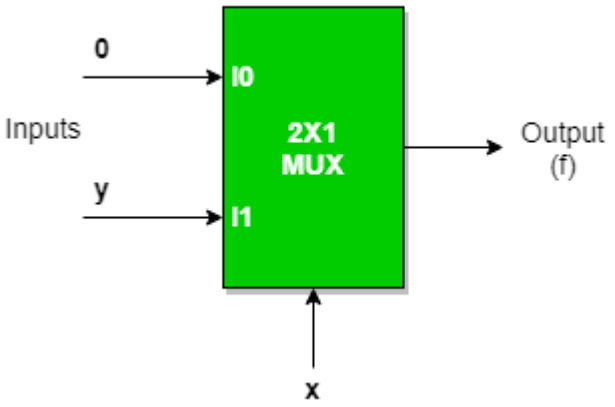


Truth Table

x	f
0	1
1	0

We can analyze it
Y = x'.1 + x.0 = x'
It is NOT Gate using 2:1 MUX.
The implementation of NOT gate is done using "n" selection lines. It cannot be implemented using "n-1" selection lines. Only NOT gate cannot be implemented using "n-1" selection lines.

b) Implementation of AND gate using 2 : 1 Mux
AND GATE



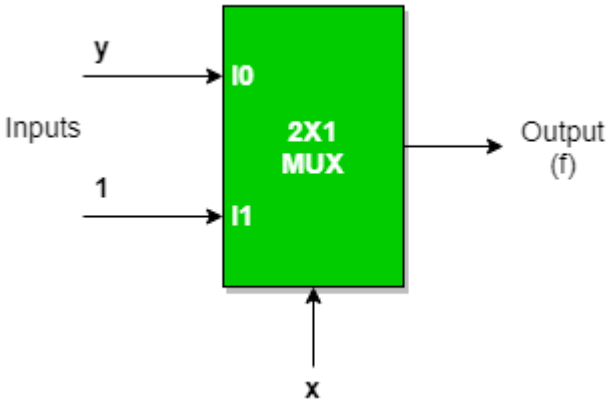
Truth Table

x	y	f	f→y
0	0	0	f = 0
0	1	0	
1	0	0	f = y
1	1	1	

This implementation is done using "n-1" selection lines.

c) Implementation of OR gate using 2 : 1 Mux using "n-1" selection lines.

OR GATE

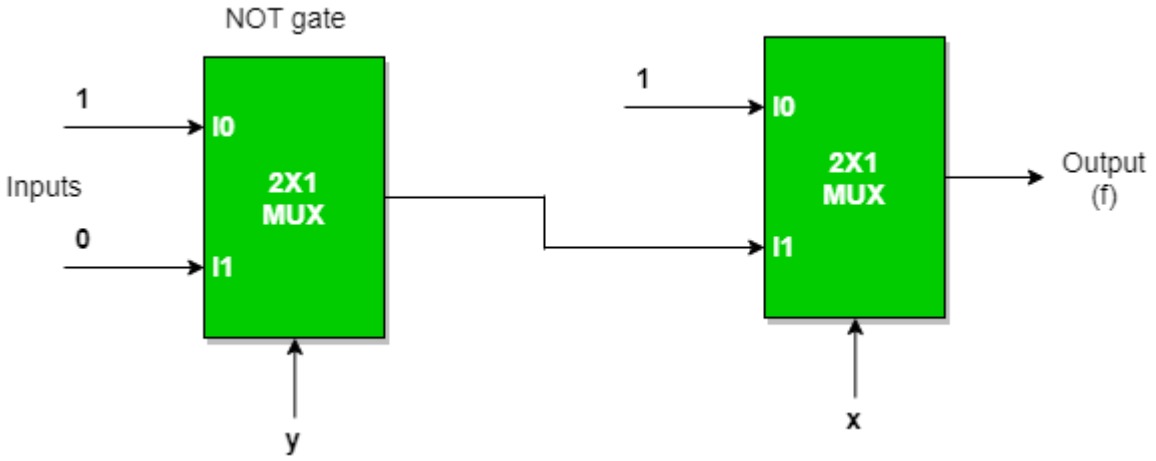


Truth Table

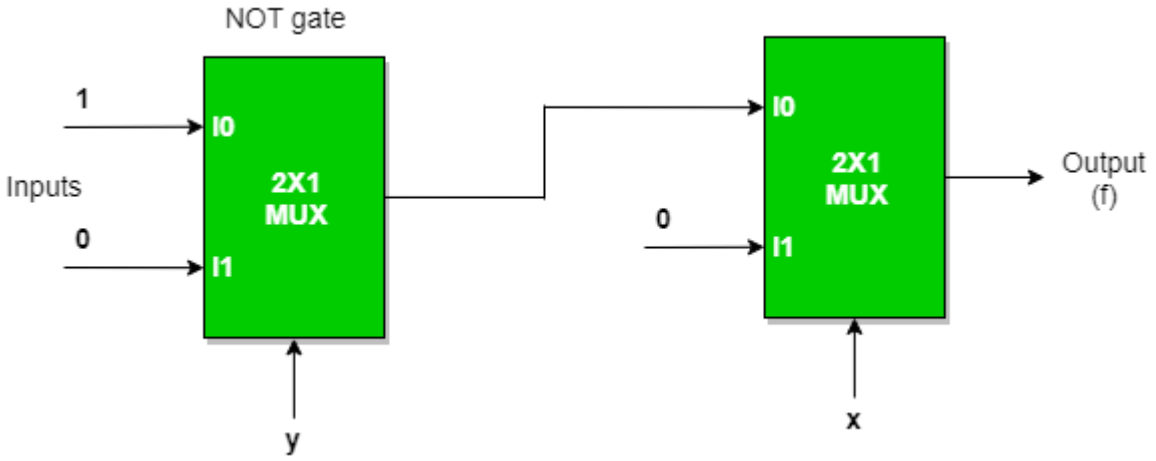
x	y	f	f→y
0	0	0	f = y
0	1	1	
1	0	1	f = 1
1	1	1	

Implementation of NAND, NOR, XOR and XNOR gates requires two 2:1 Mux. First multiplexer will act as NOT gate which will provide complemented input to the second multiplexer.

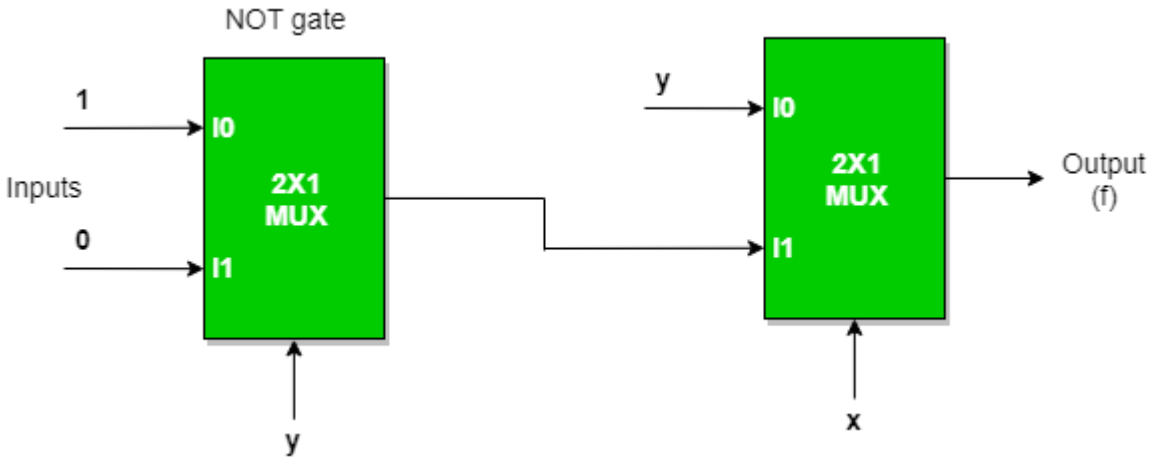
d) Implementation of NAND gate using 2 : 1 Mux
NAND GATE



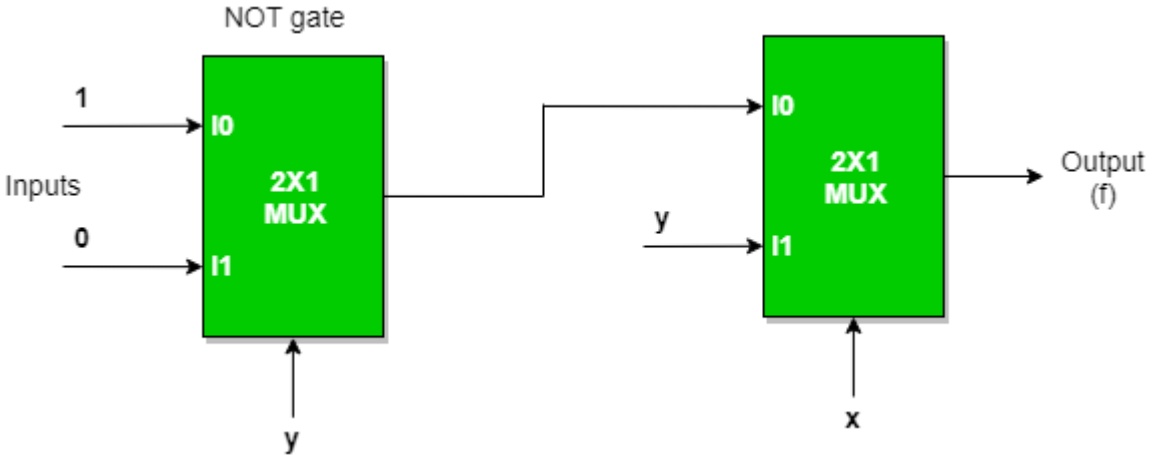
e) Implementation of NOR gate using 2 : 1 Mux
NOR GATE



f) Implementation of EX-OR gate using 2 : 1 Mux
EX-OR GATE

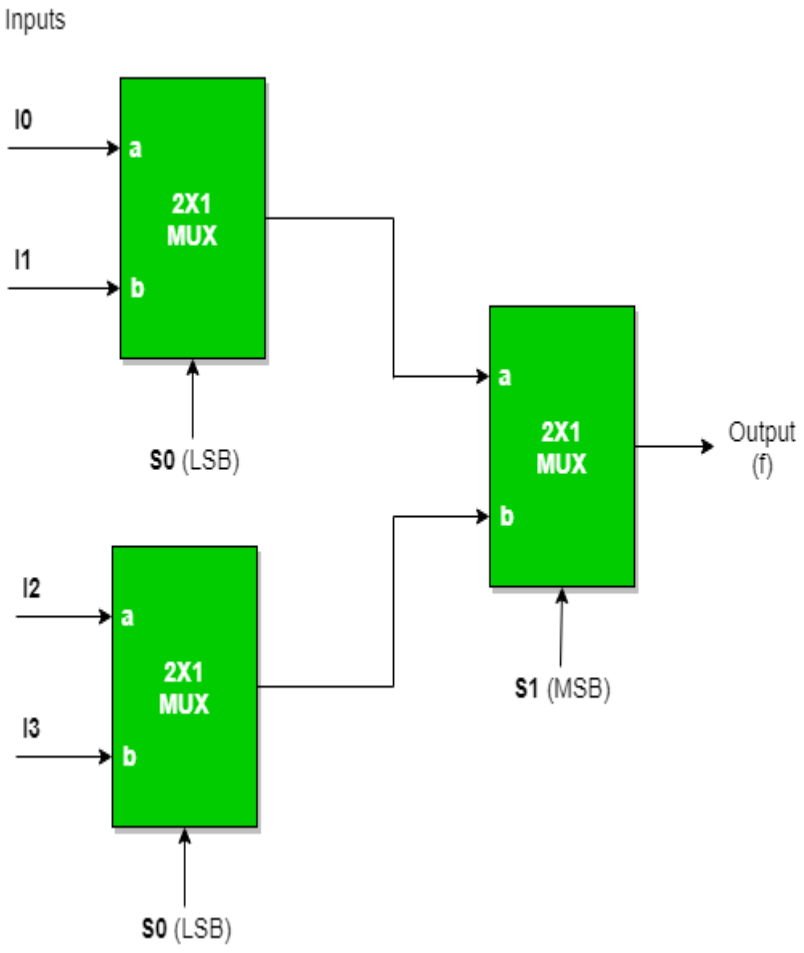


g) Implementation of EX-NOR gate using 2 : 1 Mux
EX-NOR GATE



Implementation of Higher order MUX using lower order MUX

a) 4 : 1 MUX using 2 : 1 MUX
Three(3) 2 : 1 MUX are required to implement 4 : 1 MUX.



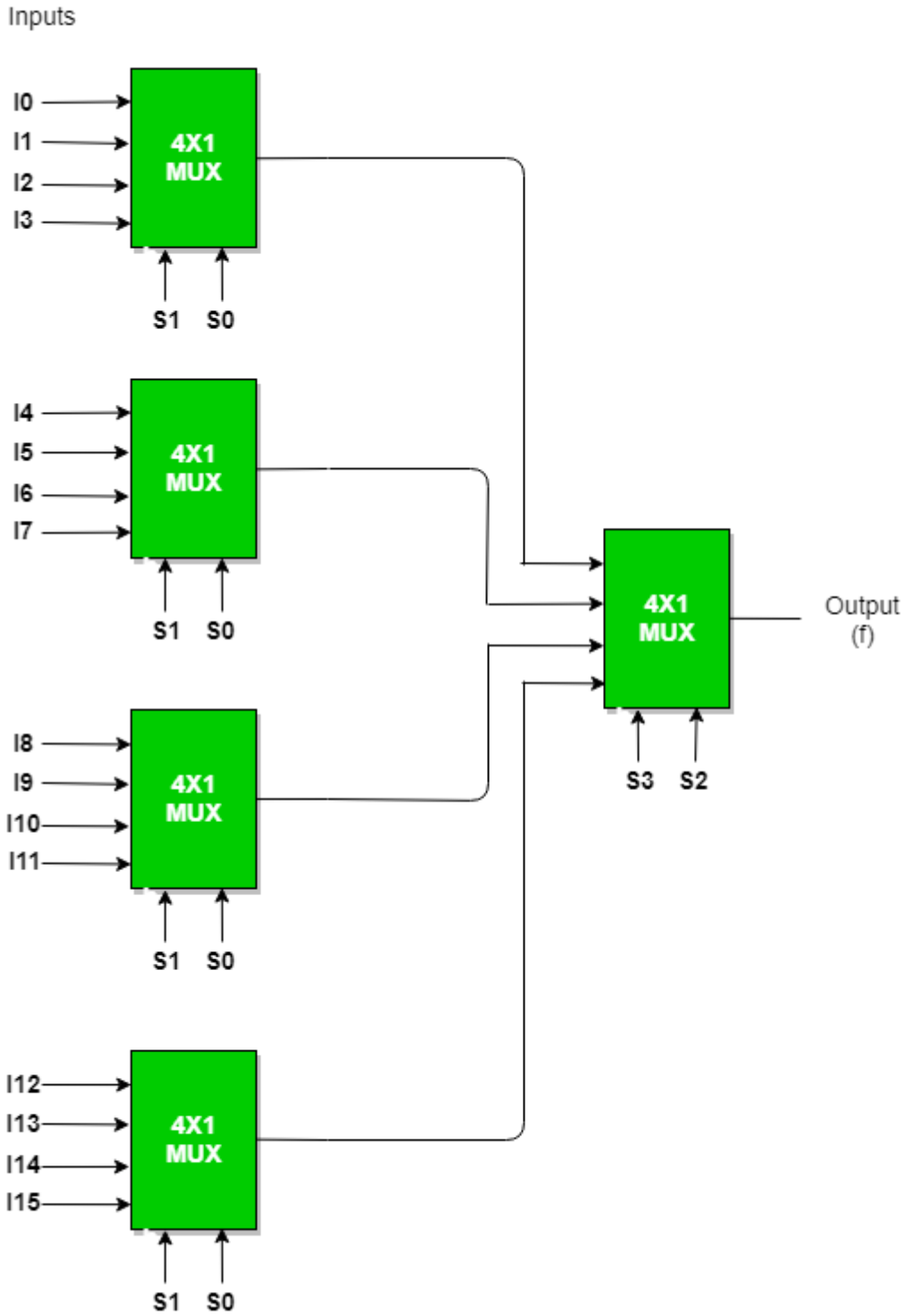
Truth Table

S0	S1	f
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Similarly,

While 8 : 1 MUX require **seven(7)** 2 : 1 MUX, 16 : 1 MUX require **fifteen(15)** 2 :1 MUX, 64 : 1 MUX requires **sixty three(63)** 2 : 1 MUX.
Hence, we can draw a conclusion,
2^n : 1 MUX requires **(2^n – 1)** 2 : 1 MUX.

b) 16 : 1 MUX using 4 : 1 MUX



In general, to implement B : 1 MUX using A : 1 MUX , one formula is used to implement the same.
B / A = K1,

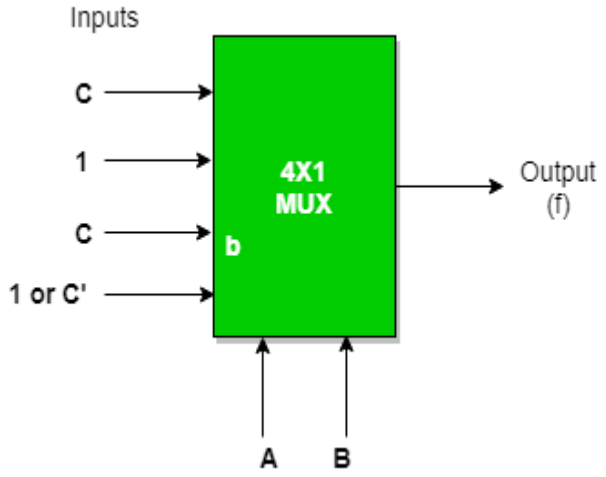
K1/ A = K2,
K2/ A = K3

……

K_{N-1} / A = K_N = 1 (till we obtain 1 count of MUX).
And then add all the numbers of MUXes = K1 + K2 + K3 + …. + K_N.
For example : To implement 64 : 1 MUX using 4 : 1 MUX
Using the above formula, we can obtain the same.
64 / 4 = 16
16 / 4 = 4
4 / 4 = 1 (till we obtain 1 count of MUX)
Hence, total number of 4 : 1 MUX are required to implement 64 : 1 MUX = 16 + 4 + 1 = 21.
An example to implement a boolean function if minimal and don't care terms are given using MUX.
f (A, B, C) = Σ (1, 2, 3, 5, 6) with don't care (7) using 4 : 1 MUX using as
a) AB as select : Expanding the minterms to its boolean form and will see its 0 or 1 value in Cth place so that they can be placed in that manner.

A	B	C	f
0	0	0	C′
0	0	1	C
0	1	0	C′
0	1	1	C
1	0	0	C′
1	0	1	C
1	1	0	C′
1	1	1	C

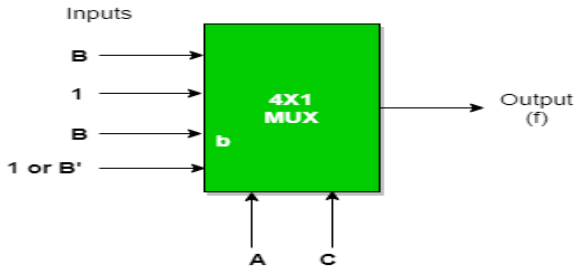
	I0	I1	I2	I3
C′	0	2	4	6
C	1	3	5	7 is don't care (can consider or not)
	C	1	C	1 (in case 7 is considered or C′)



b) AC as select : Expanding the minterms to its boolean form and will see its 0 or 1 value in Bth place so that they can be place in that manner.

A	B	C	f
0	0	0	B′
0	0	1	B′
0	1	0	B
0	1	1	B
1	0	0	B′
1	0	1	B′
1	1	0	B
1	1	1	B

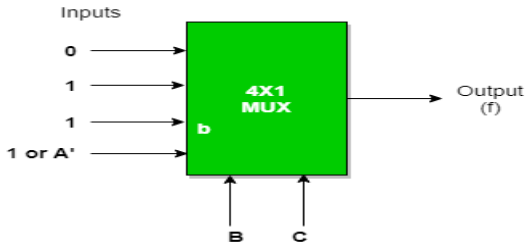
	I0	I1	I2	I3
B′	0	1	4	5
B	2	3	6	7 is don't care (can consider or not)
	B	1	B	1 (in case 7 is considered) or B′



c) BC as select : Expanding the minterms to its boolean form and will see its 0 or 1 value in A^{th} place so that they can be place in that manner.

A	B	C	f
0	0	0	A′
0	0	1	A′
0	1	0	A′
0	1	1	A′
1	0	0	A
1	0	1	A
1	1	0	A
1	1	1	A

	I0	I1	I2	I3
A′	0	1	2	3
A	4	5	6	7 is don't care (can considered or not)
	0	1	1	1 (in case 7 is considered) or A′



21.Programmable Logic Devices

A logic device is an electronic component which performs a definite function which is decided at the time of manufacture and will never change. For example, a not gate always inverts the logic level of the input signal and does/can-do-nothing else. On the other hand, **Programmable Logic Devices** (PLDs) are the components which do not have a specific function associated with them. These can be configured to perform a certain function by the user, on a need basis and can further be changed to perform some other function at the later point of time, i.e. these are re-configurable. However, the amount of flexibility offered depends on their type.

Types of Programmable Logic Devices

Programmable Logic Array (PLA)

This device comprises of programmable AND gate and OR gate arrays which are to be configured by the user to obtain the output.

Programmable Array Logic (PAL)

PALs use an OR gate array with fixed logic while an AND gate array which can be programmed as per the requirement of the user. As a result, these devices express the output as a combination of inputs in sum-of-products form.

Generic Logic Array (GLA)

These devices had their properties similar to those of PALs in addition to which they were electrically erasable and re-programmable. This important feature proved to be meritorious as it considerably eased the prototype design which in turn reduced the time to market.

Complex Programmable Logic Device (CPLD)

CPLDs are denser than PALs and comprise of a large number of programmable logical elements. The interconnection between these macro cells is to be established by the user through the interconnecting network. Here sum-of-product establishing logical elements are combined together to form structures in order to reduce the number of input-output (IO) pins. This facilitates the implementation of more complex logic design with slightly worse propagation time when compared to that of PALs. These offer predictable timing characteristics making them most suitable for critical control applications with high performance. CPLDs are preferred to implement combinational logic based designs.

22.Introduction of Sequential Circuits

A **Sequential circuit** combinational logic circuit that consists of inputs variable (X), logic gates (Computational circuit), and output variable (Z).

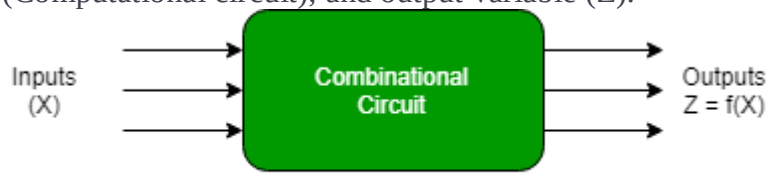


Figure: Combinational Circuits

Combinational circuit produces an output based on input variable only, but **Sequential circuit** produces an output based on **current input and previous input variables**. That means sequential circuits include memory elements which 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.

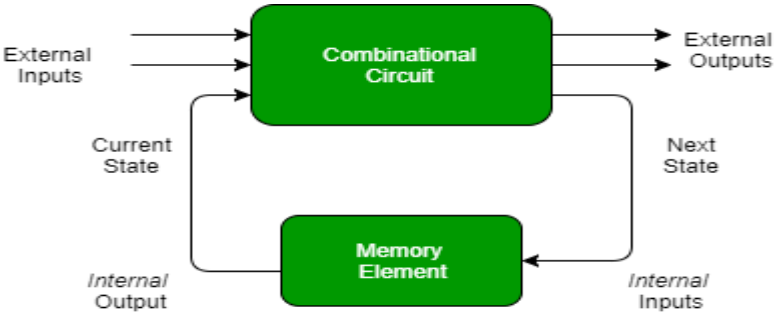


Figure: Sequential Circuit

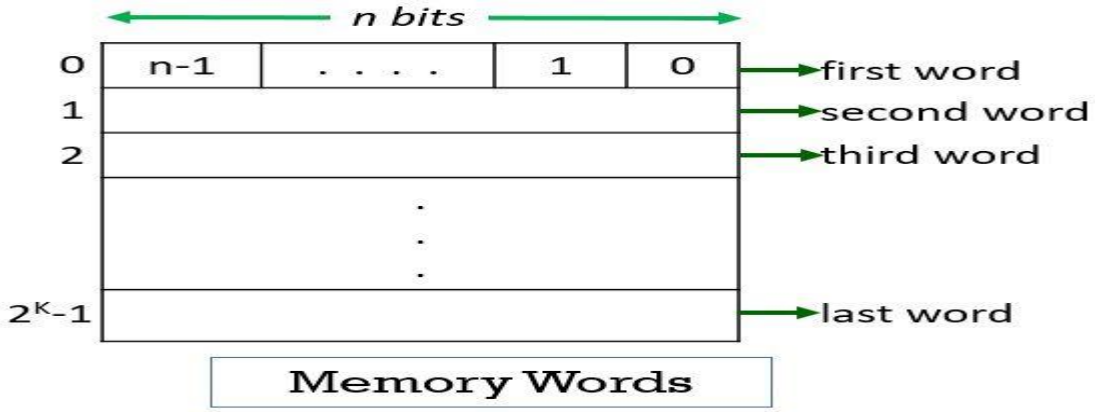
Unit-2

1.Memory locations and addresses: Memory locations and addresses determine how the computer's memory is organized so that the user can efficiently store or retrieve information from the computer. The computer's memory is made of a silicon chip which has millions of storage cell, where each storage cell is capable to store a *bit* of information which value is either 0 or 1.

But the fact is, computer memory holds instructions and data. And a single bit is very small to hold this information so bits are rarely used individually. As a solution to this, the bits are grouped in fixed sizes of n bits.

The memory of the computer is organized in such a way that the group of these n bits can be stored and retrieved easily by the computer in a single operation.

The group of n bit is termed as ***word*** where n is termed as the *word length*. The word length of the computer has evolved from 8, 16, 24, 32 to 64 bits. General-purpose computers nowadays have 32 to 64 bits. The group of 8 bit is called a *byte*.



Now, whenever you want to store any instruction or data may it be of a byte or a word you have to access a memory location. To access the memory location either you must know the memory location by its unique name or it is required to provide a unique address to each memory location.

The memory locations are addressed from 0 to 2^K-1 i.e. a memory has 2^K addressable locations. And thus the address space of the computer has 2^K addresses. Let us try some suitable values for K.

2^{10} = 1024 = 1K (Kilobyte)
2^{20} = 1,048,576 = 1M (Megabyte)
2^{30} = 1073741824 = 1G (Gigabyte)
2^{40} = 1.0995116e+12 = 1T (Terabyte)

Byte Addressability

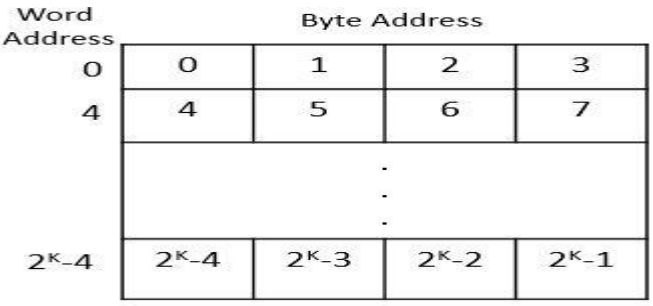
Till now we have gone through three information storing quantities bit, byte and word. We have seen above that 8 bits together form a byte and this is the fix for every memory. But the word length varies from memory to memory and it ranges from 16 to 64 bit.

Well, it is impossible to allot a unique address to each bit in memory. As a solution, most modern computers assign successive addresses to successive byte locations in memory. This assignment of addresses to individual byte locations is termed byte addressability and memory is referred to as byte-addressable memory.

If we assign an address to individual byte locations in the memory like 0, 1, 2, 3…. .Now if the word length of the machine is 16 bit then the successive words are located at addresses 0, 2, 4, 6… where each word would have 2 bytes of information. Similarly, if we have a machine with a word length of 32 bit then the successive words are located at the addresses 0, 4, 8, 12… where each word would have 4 bytes of information and it could store or retrieve 4 bytes of instruction or data in a single and basic operation.

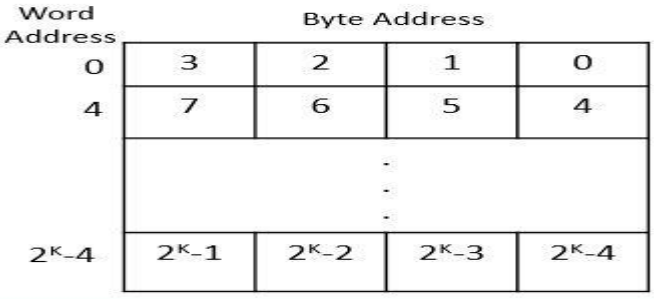
Big-Endian and Little-Endian Assignments in Byte Addresses

The big-endian and little-endian are two methods of assigning byte addresses across the words in the memory. In the **big-endian assignmen**t, the lower byte addresses are used for the leftmost bytes of the word. Observe the word 0 in the image below, the leftmost bytes of the word have lower byte addresses.



Big-Endian Assignment

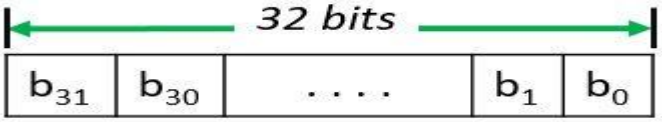
In the **little-endian assignment,** the lower byte addresses are used for the rightmost bytes of the word. Observe the word 0 in the image below the rightmost bytes of word 0 has lower byte addresses.



Little-Endian Assignment

The leftmost bytes of the word are termed as *most significant bytes* and the rightmost bytes of the words are termed as least significant bytes.

Thus the big-endian and little-endian specify the ordering of bytes inside a word. Similarly, the bits must be labelled inside the byte or a word and the most common way of labelling bits in a byte or word is as shown in the figure below i.e. labelling the bits as b_7, b_6,……,b_1, b_0 from left to write as we do in little-endian assignment.



Labelling bits within a Byte

Word Alignment

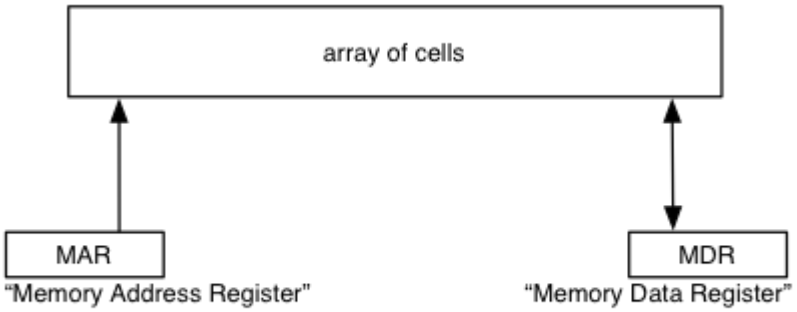
In a machine with word length 32-bit, the word boundaries occur at the bytes addresses 0, 4, 8… It is said that the word has *aligned addresses* if they begin with the byte address that is multiple of the number of bytes present in that word. For example, the word address 4 has four bytes in it with byte address 4, 5 and 6. The word address 4 starts with the byte address 4 which is multiple of the number of bytes in word 4.

In case if the word address begins with the arbitrary byte address the word is said to have *unaligned addresses*. But conventionally the words have aligned addresses as this lets the access of memory operand more efficiently.

Memory Operations

So, this is all about the memory locations and how they are addressed to store and retrieve the instructions or data more efficiently. With memory addresses, it becomes easy to identify a specific memory location.

- There are two key operations on memory:
    1. **fetch(address)** returns value without changing the value stored at that address.
    2. **store(address, value)** writes new value into the cell at the given address.
- This type of memory is *random-access*, meaning that CPU can access any value of the array at any time (vs. sequential access, like on a tape).
- Such memories are called **RAM** (random-access memory.)
- Some memory is non-volatile, or read-only (**ROM** or read-only memory.)



fetch (addr):
1. Put *addr* into MAR
2. Tell memory unit to "load"
3. Memory copies data into MDR

store (addr, new-value):
1. Put *addr* into MAR
2. Put *new-value* into MDR
3. Tell memory unit to "store"
4. Memory stores data from MDR into memory cell.

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

• Two memory operations are:

1) Load (Read/Fetch) &

2) Store (Write).

• The Load operation transfers a copy of the contents of a specific memory-location to the processor. The memory contents remain unchanged.

• **Steps for Load operation:**

1) Processor sends the address of the desired location to the memory.

2) Processor issues „read" signal to memory to fetch the data.

3) Memory reads the data stored at that address.

4) Memory sends the read data to the processor

• The Store operation transfers the information from the register to the specified memory-location. This will destroy the original contents of that memory-location.

• **Steps for Store operation are:**

1) Processor sends the address of the memory-location where it wants to store data.

2) Processor issues „write" signal to memory to store the data.

3) Content of register(MDR) is written into the specified memory-location.

INSTRUCTIONS & INSTRUCTION SEQUENCING

The tasks carried out by a computer program consist of a sequence of small steps, suchas adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen.

• A computer must have instructions capable of performing 4 types of operations:

1) Data transfers between the memory and the registers (MOV, PUSH, POP, XCHG).

2) Arithmetic and logic operations on data (ADD, SUB, MUL, DIV, AND, OR, NOT).

3) Program sequencing and control(CALL.RET, LOOP, INT).

4) I/0 transfers (IN, OUT).

REGISTER TRANSFER NOTATION (RTN)

Here we describe the transfer of information from one location in a computer to another.Possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify such locationssymbolically with convenient names.

• The possible locations in which transfer of information occurs are:

1) Memory-location

2) Processor register &

3) Registers in I/O device.

INSTRUCTION EXECUTION & STRAIGHT LINE SEQUENCING

• The program is executed as follows:

1) Initially, the address of the first instruction is loaded into PC (Figure 2.8).

2) Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses.

This is called Straight-Line sequencing.

3) During the execution of each instruction, PC is incremented by 4 to point to next instruction.

• There are 2 phases for Instruction Execution:

1) Fetch Phase: The instruction is fetched from the memory-location and placed in the IR.

2) Execute Phase: The contents of IR is examined to determine which operation is to beperformed. The specified-operation is then performed by the processor.

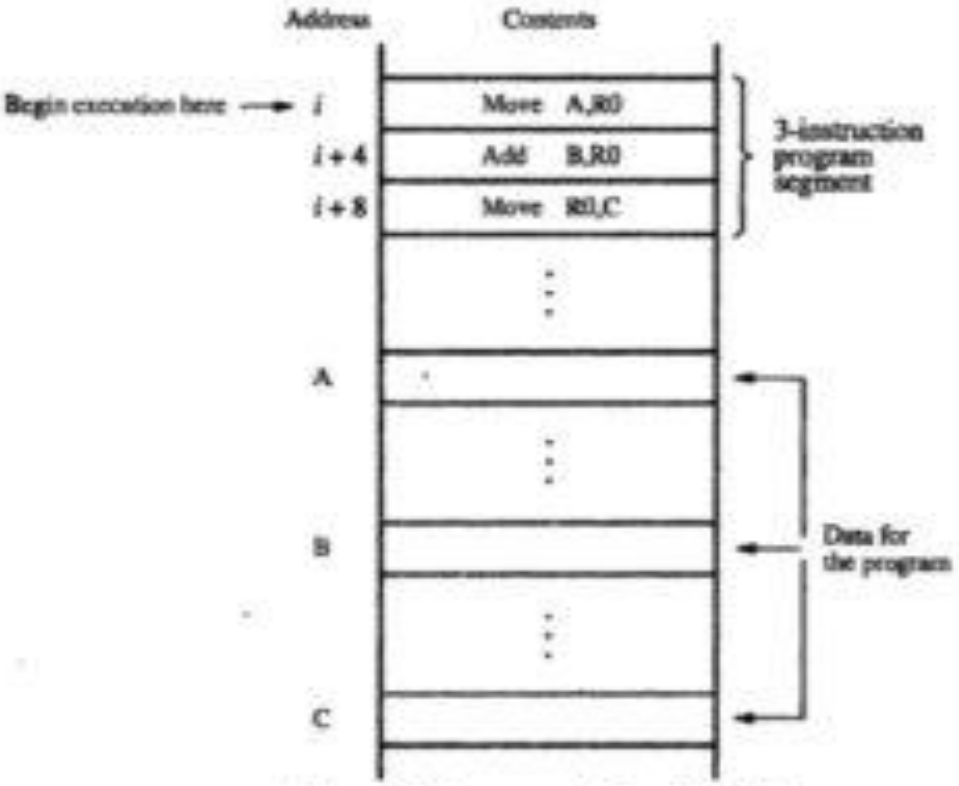


Figure 2.8 A program for C ← [A] + [B].

Program Explanation

• Consider the program for adding a list of n numbers (Figure 2.9).

• The Address of the memory-locations containing the n numbers are symbolically given as NUM1, NUM2…..NUMn.

• Separate Add instruction is used to add each number to the contents of register R0.

• After all the numbers have been added, the result is placed in memory-location SUM.

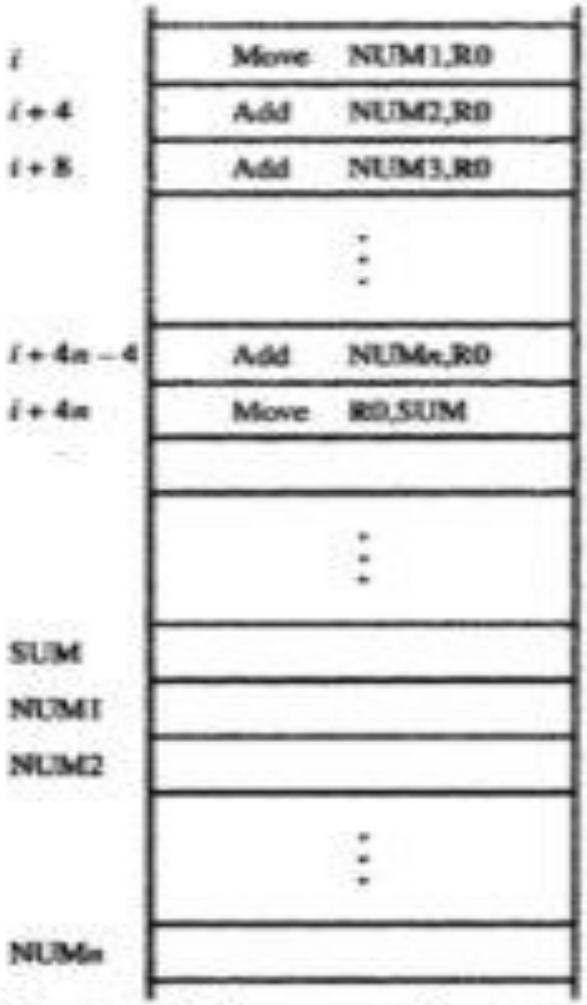


Figure 2.9 A straight-line program for adding *n* numbers.

Addressing Modes:

Addressing Modes– The term addressing modes refers to the way in which the operand of an instruction is specified. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually executed.

Addressing modes for 8086 instructions are divided into two categories:

1) Addressing modes for data

2) Addressing modes for branch

The 8086 memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, records, pointers, and other complex data types. The key to good assembly language programming is the proper use of memory addressing modes.

The memory address of an operand consists of two components:

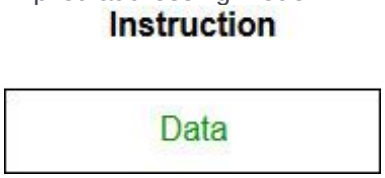
IMPORTANT TERMS

- **Starting address** of memory segment.
- **Effective address or Offset**: An offset is determined by adding any combination of three address elements: **displacement, base and index.**
    - **Displacement:** It is an 8 bit or 16 bit immediate value given in the instruction.
    - **Base**: Contents of base register, BX or BP.
    - **Index**: Content of index register SI or DI.

According to different ways of specifying an operand by 8086 microprocessor, different addressing modes are used by 8086.

Addressing modes used by 8086 microprocessor are discussed below:

- **Implied mode:**: In implied addressing the operand is specified in the instruction itself. In this mode the data is 8 bits or 16 bits long and data is the part of instruction.Zero address instruction are designed with implied addressing mode.

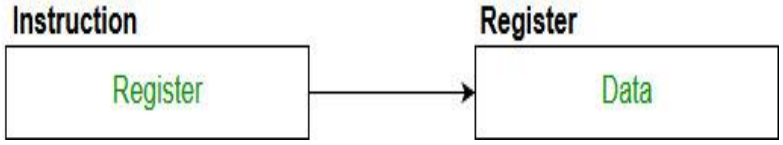


Example: CLC (used to reset Carry flag to 0)

- **Immediate addressing mode (symbol #):**In this mode data is present in address field of instruction .Designed like one address instruction format.
  Note: Limitation in the immediate mode is that the range of constants are restricted by size of address field.

Example: MOV AL, 35H (move the data 35H into AL register)

- **Register mode:** In register addressing the operand is placed in one of 8 bit or 16 bit general purpose registers. The data is in the register that is specified by the instruction.
  Here one register reference is required to access the data.



- **Register Indirect mode**: In this addressing the operand's offset is placed in any one of the registers BX,BP,SI,DI as specified in the instruction. The effective address of the data is in the base register or an index register that is specified by the instruction.
  Here two register reference is required to access the data.

The 8086 CPUs let you access memory indirectly through a register using the register indirect addressing modes.

- MOV AX, [BX](move the contents of memory location s

  addressed by the register BX to the register AX)

- **Auto Indexed (increment mode)**: Effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next consecutive memory location.**(R1)+**.
  Here one register reference,one memory reference and one ALU operation is required to access the data.
  Example:
- Add R1, (R2)+ // OR
- R1 = R1 +M[R2]

  R2 = R2 + d

  Useful for stepping through arrays in a loop. R2 – start of array d – size of an element
- **Auto indexed (decrement mode)**: Effective address of the operand is the contents of a register specified in the instruction. Before accessing the operand, the contents of this register are automatically decremented to point to the previous consecutive memory location. **–(R1)**
  Here one register reference,one memory reference and one ALU operation is required to access the data.
  Example:
  Add R1,-(R2) //OR

  R2 = R2-*d*

R1 = R1 + M[R2]
Auto decrement mode is same as auto increment mode. Both can also be used to implement a stack as push and pop . Auto increment and Auto decrement modes are useful for implementing "Last-In-First-Out" data structures.

- **Direct addressing/ Absolute addressing Mode (symbol [ ]):** The operand's offset is given in the instruction as an 8 bit or 16 bit displacement element. In this addressing mode the 16 bit effective address of the data is the part of the instruction.
Here only one memory reference operation is required to access the data.

Example:ADD AL,[0301] //add the contents of offset address 0301 to AL

- **Indirect addressing Mode (symbol @ or ())**:In this mode address field of instruction contains the address of effective address.Here two references are required.
1st reference to get effective address.
2nd reference to access the data.
Based on the availability of Effective address, Indirect mode is of two kind:

1. Register Indirect:In this mode effective address is in the register, and corresponding register name will be maintained in the address field of an instruction.
Here one register reference,one memory reference is required to access the data.
2. Memory Indirect:In this mode effective address is in the memory, and corresponding memory address will be maintained in the address field of an instruction.
Here two memory reference is required to access the data.

- **Indexed addressing mode**: The operand's offset is the sum of the content of an index register SI or DI and an 8 bit or 16 bit displacement.
Example:MOV AX, [SI +05]

- **Based Indexed Addressing:** The operand's offset is sum of the content of a base register BX or BP and an index register SI or DI.
Example: ADD AX, [BX+SI]

Based on Transfer of control, addressing modes are:

- **PC relative addressing mode:** PC relative addressing mode is used to implement intra segment transfer of control, In this mode effective address is obtained by adding displacement to PC.
- EA= PC + Address field value

  PC= PC + Relative value.

- **Base register addressing mode:**Base register addressing mode is used to implement inter segment transfer of control.In this mode effective address is obtained by adding base register value to address field value.
- EA= Base register + Address field value.

  PC= Base register + Relative value.

Unit-3

1.Accessing I/O Devices.: In computing, input/output, or I/O, refers to the communication between an information processing system (computer), and the outside world. Inputs are the signals or data received by the system, and outputs are the signals or data sent from it. I/O devices are used by a person (or other system) to communicate with a computer.
Some of the input devices are keyboard, mouse, track ball, joy stick, touch screen, digital camera, webcam, image scanner, fingerprint scanner, barcode reader, microphone and so on. Some of the output devices are speakers, headphones, monitors and printers. Devices for communication between computers, such as modems and network cards, typically serve for both input and output. I/O devices

can be connected to a computer through a single bus which enables the exchange of information. The bus consists of three sets of lines used to carry address, data, and control signals. Each I/O device is assigned a unique set of addresses. When the processor places a particular address on the address lines, the device that recognizes this address responds to the commands issued on the control lines. The processor requests either a read or a write operation, and the requested data are transferred over the data lines. Figure 5.1 shows the simple arrangement of I/O devices to processor and memory with single bus.

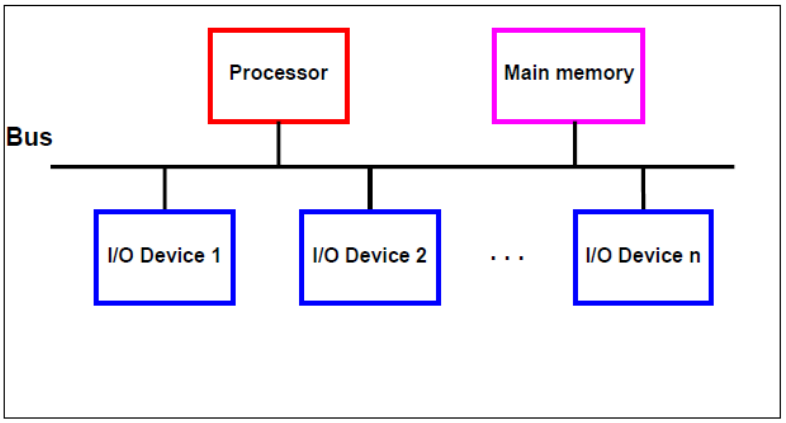


Figure 5.1 A Single bus structure

Memory-mapped I/O: The arrangement of I/O devices and the memory share the same address space is called memory-mapped I/O. With memory-mapped I/O, any machine instruction that can access memory can be used to transfer data to or from an I/O device. For example, if DATAIN is the address of the input buffer associated with the keyboard, the instruction

Move DATAIN,R0

reads the data from DATAIN and stores them into processor register RO. Similarly, the instruction

Move R0,DATAOUT

sends the contents of register R0 to location DATAOUT, which may be the output data buffer of a display unit or a printer. Most computer systems use memory-mapped I/O. Some processors have special In and Out instructions to perform I/O transfers.

Figure 5.2 illustrates the hardware required to connect an I/O device to the bus. The address decoder enables the device to recognize its address when this address appears on the address lines. The data register holds the data being transferred to or from the processor. The status register contains information relevant to the operation of the I/O device. Both the data and status registers are connected to the data bus and assigned unique addresses. The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's interface circuit.

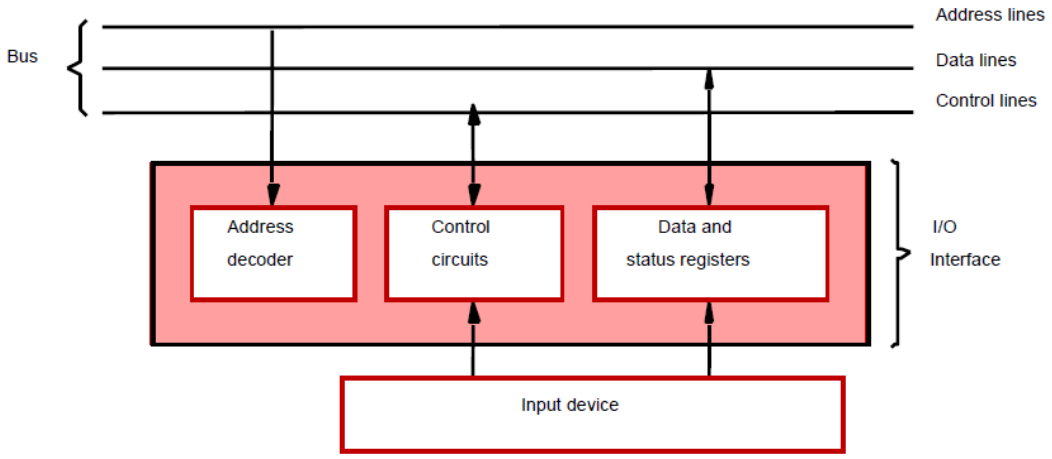


Figure 5.2 I/O interface for an input device

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

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

Program-controlled I/O: Consider a simple example of I/O operations involving a keyboard and a display device in a computer system. The four registers shown in Figure 5.3 are used in the data transfer operations. Register STATUS contains two control flags, SIN and SOUT, which provide status information for the keyboard and the display unit, respectively. The two flags KIRQ and DIRQ in this register are used in conjunction with interrupts. They, and the KEN and DEN bits in register CONTROL, Data from the keyboard are made available in the DATAIN register, and data sent to the display are stored in the DATAOUT register.

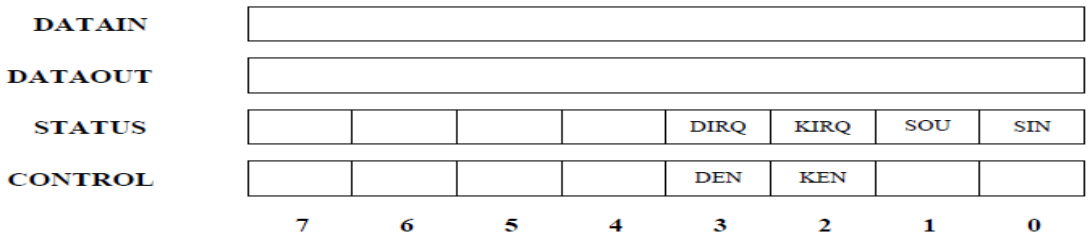


Figure 5.3 Registers in keyboard and display interfaces.

The program in Figure 5.4 reads a line of characters from the keyboard and stores it in a memory buffer starting at location LINE. Then, it calls a subroutine PROCESS to process the input line. As each character is read, it is echoed back to the display. Register R0 is used as a pointer to the memory buffer area. The contents of R0 are updated using the Autoincrement addressing mode so that successive characters are stored in successive memory locations. Each character is checked to see if it

is the Carriage Return (CR) character, which has the ASCII code 0D (hex). If it is, a Line Feed character (ASCII code 0A) is sent to move the cursor one line down on the display and subroutine PROCESS is called. Otherwise, the program loops back to wait for another character from the keyboard.

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

```
        Move        #LINE,R0          Initialize memory pointer.
WAITK   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 input line.
```

Figure 5.4 A program that reads one line from the keyboard, stores it in memory buffer, and echoes it back to the display

2.Interrupts: Interrupt is a hardware signal to the processor from I/O devices through one of the control line called interrupt-request line. The routine executed in response to an interrupt request is called the interrupt-service routine, Interrupts bear considerable resemblance to subroutine calls. Assume that an interrupt request arrives during execution of instruction i in Figure 4.5. The processor first completes execution of instruction i. Then, it loads the program counter with the address of the first instruction of the interrupt-service routine. For the time being, let us assume that this address is hardwired in the processor. After execution of the interrupt-service routine, the processor has to come back to instruction i 1. Therefore, when an interrupt occurs, the current contents of the PC, which point to instruction i 1, must be put in temporary storage in a known location. A Return from-interrupt instruction at the end of the interrupt-service routine reloads the PC from that temporary storage location, causing execution to resume at instruction i 1. In many processors, the return address is saved on the processor stack. Alternatively, it may be saved in a special location, such as a register provided for this purpose.

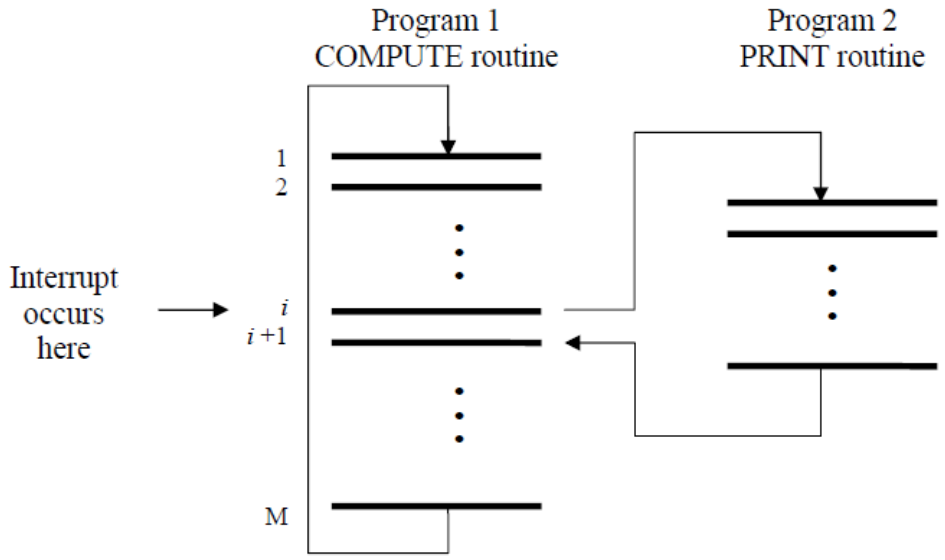


Figure 5.5 Transfer of control through the use of interrupts

3.DIRECT MEMORY ACCESS (DMA): The main idea of direct memory access (DMA) is to enable peripheral devices to cut out the "middle man" role of the CPU in data transfer. It allows peripheral devices to transfer data directly from and to memory without the intervention of the CPU. Having peripheral devices access memory directly would allow the CPU to do other work, which would lead to improved performance, especially in the cases of large transfers. The DMA controller is a piece of hardware that controls one or more peripheral devices. It allows devices to transfer data to or from the system's memory without the help of the processor. In a typical DMA transfer, some event notifies the DMA controller that data needs to be transferred to or from memory. Both the DMA and CPU use memory bus and only one or the other can use the memory at the same time. The DMA controller then sends a request to the CPU asking its permission to use the bus. The CPU returns an acknowledgment to the DMA controller granting it bus access. The DMA can now take control of the bus to independently conduct memory transfer. When the transfer is complete the DMA relinquishes its control of the bus to the CPU. Processors that support DMA provide one or more input signals that the bus requester can assert to gain control of the bus and one or more output signals that the CPU asserts to indicate it has relinquished the bus. Figure 8.10 shows how the DMA controller shares the CPU's memory bus.

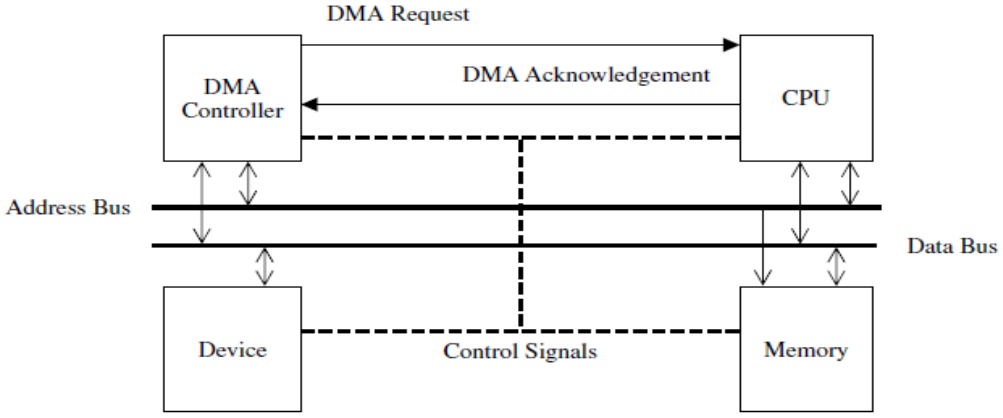


Figure 8.10 DMA controller shares the CPU's memory bus

Direct memory access controllers require initialization by the CPU. Typical setup parameters include the address of the source area, the address of the destination area, the length of the block, and whether the DMA controller should generate a processor interrupt once the block transfer is complete. A DMA

controller has an address register, a word count register, and a control register. The address register contains an address that specifies the memory location of the data to be transferred. It is typically possible to have the DMA controller automatically increment the address register after each word transfer, so that the next transfer will be from the next memory location. The word count register holds the number of words to be transferred. The word count is decremented by one after each word transfer. The control register specifies the transfer mode.

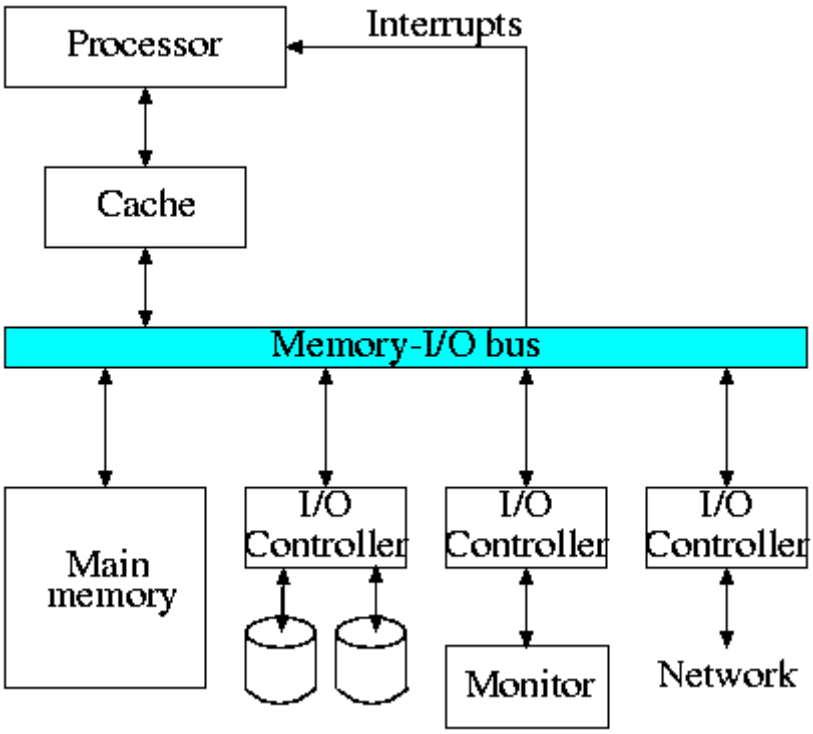
Direct memory access data transfer can be performed in burst mode or singlecycle mode. In burst mode, the DMA controller keeps control of the bus until all the data has been transferred to (from) memory from (to) the peripheral device. This mode of transfer is needed for fast devices where data transfer cannot be stopped until the entire transfer is done. In single-cycle mode (cycle stealing), the DMA controller relinquishes the bus after each transfer of one data word. This minimizes the amount of time that the DMA controller keeps the CPU from controlling the bus, but it requires that the bus request/acknowledge sequence be performed for every single transfer. This overhead can result in a degradation of the performance. The single-cycle mode is preferred if the system cannot tolerate more than a few cycles of added interrupt latency or if the peripheral devices can buffer very large amounts of data, causing the DMA controller to tie up the bus for an excessive amount of time. **The following steps summarize the DMA operations:**

1. DMA controller initiates data transfer.
2. Data is moved (increasing the address in memory, and reducing the count of words to be moved).
3. When word count reaches zero, the DMA informs the CPU of the termination by means of an interrupt.
4. The CPU regains access to the memory bus.

A DMA controller may have multiple channels. Each channel has associated with it an address register and a count register. To initiate a data transfer the device driver sets up the DMA channel's address and count registers together with the direction of the data transfer, read or write. While the transfer is taking place, the CPU is free to do other things. When the transfer is complete, the CPU is interrupted. Direct memory access channels cannot be shared between device drivers. A device driver must be able to determine which DMA channel to use. Some devices have a fixed DMA channel, while others are more flexible, where the device driver can simply pick a free DMA channel to use.

Linux tracks the usage of the DMA channels using a vector of dma_chan data structures (one per DMA channel). The dma_chan data structure contains just two fields, a pointer to a string describing the owner of the DMA channel and a flag indicating if the DMA channel is allocated or not.

or must inform the device that its request h been recognized so that it may remove its interrupt-request signal. This may be accomplished by means of a special control signal on the bus called an interrupt-acknowledge signal. The execution of an instruction in the interrupt - service routine that accesses a status or data register in the device interface implicitly informs the device that its interrupt request has been recognized.



4.BUSES: A bus in computer terminology represents a physical connection used to carry a signal from one point to another. The signal carried by a bus may represent address, data, control signal, or power. Typically, a bus consists of a number of connections running together. Each connection is called a bus line. A bus line is normally identified by a number. Related groups of bus lines are usually identified by a name. For example, the group of bus lines 1 to 16 in a given computer system may be used to carry the address of memory locations, and therefore are identified as address lines.

Synchronous Buses: In synchronous buses, the steps of data transfer take place at fixed clock cycles. Everything is synchronized to bus clock and clock signals are made available to both master and slave. The bus clock is a square wave signal. A cycle starts at one rising edge of the clock and ends at the next rising edge, which is the beginning of the next cycle. A transfer may take multiple bus cycles depending on the speed parameters of the bus and the two ends of the transfer.One scenario would be that on the first clock cycle, the master puts an address on the address bus, puts data on the data bus, and asserts the appropriate control lines. Slave recognizes its address on the address bus on the first cycle and reads the new value from the bus in the second cycle. Synchronous buses are simple and easily implemented. However, when connecting devices with varying speeds to a synchronous bus, the slowest device will determine the speed of the bus. Also, the synchronous bus length could be limited to avoid clock-skewing problems.

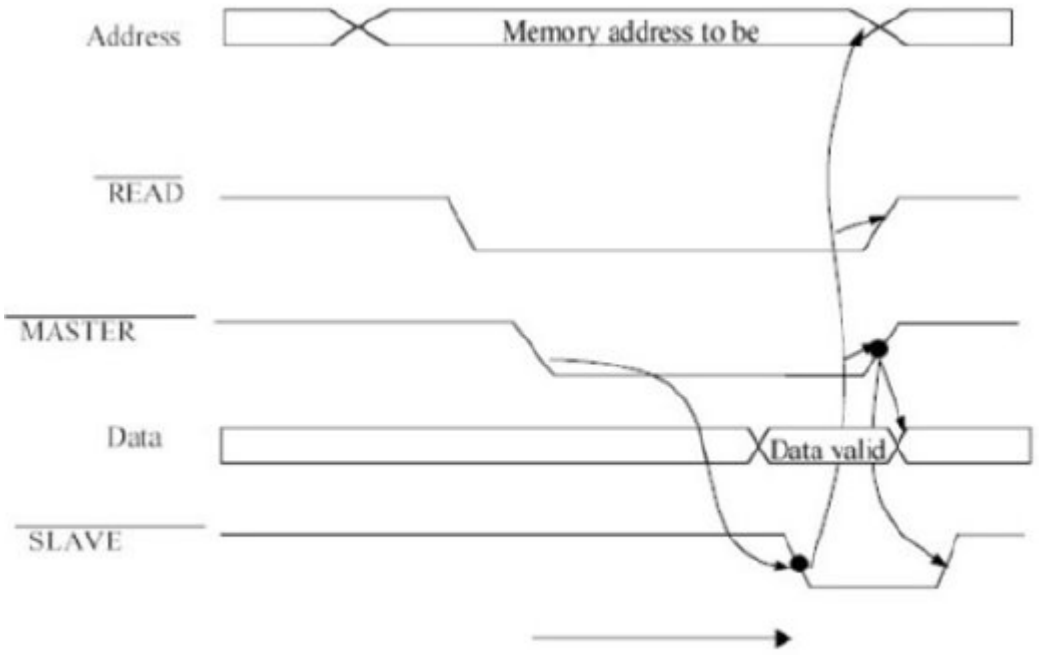


Figure 6. Read Operations on an Asynchronous Bus

A memory read transaction on the synchronous bus typically proceeds as illustrated in Fig. 5. During the first clock cycle the CPU places the address of the location it wants to read, on the address lines of the bus. Later during the same clock cycle, once the address lines have stabilized, the READ request is asserted by the CPU. Many times, some of these control signals are active low and asserting the signal means that they are pulled low. A few clock cycles are needed for the memory to perform accessing of the requested location. In a simple non-pipelined bus, these appear as wait states and the data is placed on the bus by the memory after the tow or three wait cycles. The CPU then releases the bus by deasserting the READ control signal. The write transaction is similar except that the processor is the data source and the WRITE signal is the one that is asserted. Different bus architectures synchronize bus operations with respect to the rising edge or falling edge or level of the clock signal.

Asynchronous Buses: There are no fixed clock cycles in asynchronous buses. Handshaking is used instead. Figure 8.11 shows the handshaking protocol. The master asserts the data-ready line

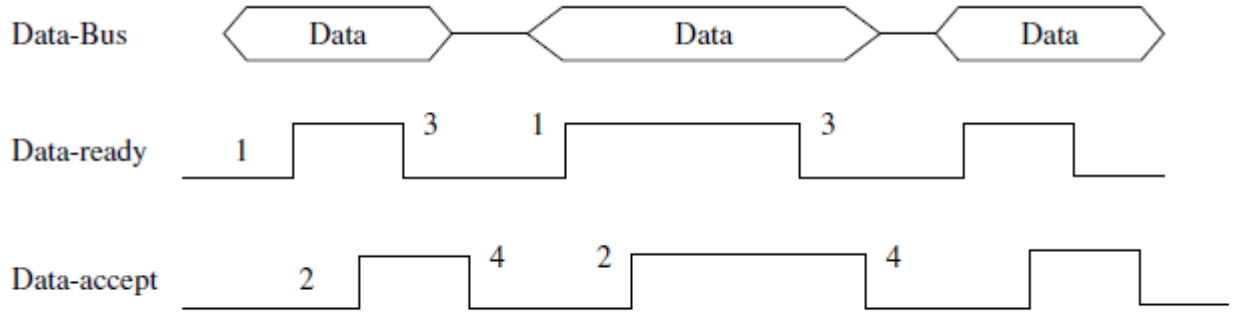


Figure 8.11 Asynchronous bus timing using handshaking protocol

(point 1 in the figure) until it sees a data-accept signal. When the slave sees a dataready signal, it will assert the data-accept line (point 2 in the figure). The rising of the data-accept line will trigger the falling of the data-ready line and the removal of data from the bus. The falling of the data-ready line (point 3 in the figure) will trigger the falling of the data-accept line (point 4 in the figure). This

handshaking, which is called fully interlocked, is repeated until the data is completely transferred. Asynchronous bus is appropriate for different speed devices.

An asynchronous bus has no system clock. Handshaking is done to properly conduct the transmission of data between the sender and the receiver. The process is illustrated in Fig. 6. For example, in an asynchronous read operation, the bus master puts the address and control signals on the bus and then asserts a synchronization signal. The synchronization signal from the master prompts the slave to get synchronized and once it has accessed the data, it asserts its own synchronization signal. The slave's synchronization signal indicates to the processor that there is valid data on the bus, and it reads the data. The master then deasserts its synchronization signal, which indicates to the slave that the master has read the data. The slave then deasserts its synchronization signal. This method of synchronization is referred to as a full handshake. Note that there is no clock and that starting and ending of the data transfer are indicated by special synchronization signals. An asynchronous communication protocol can be considered as a pair of Finite State machines (FSMs) that operate in such a way that one FSM does not proceed until the other FSM has reached a certain state.

Synchronous buses are typically faster than asynchronous buses because there is no overhead to establish a time reference for each transaction. Another reason that helps the synchronous bus to operate fast is that the bus protocol is predetermined and very little logic is involved in implementing the Finite State machine. However, synchronous buses are affected by clock skew and they cannot be very long. But asynchronous buses work well even when they are long because clock skew problems do not affect them. Thus asynchronous buses can handle longer physical distances and higher number of devices. Processor-memory buses are typically synchronous because the devices connected to the bus are fast, are small in number and are located in close proximity. I/O buses are typically asynchronous because many peripherals need only slow data rates and are physically situated far away.

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

HANDLING MULTIPLE DEVICES: Let us now consider the situation where a number of devices capable of initiating interrupts are connected to the processor. Because these devices are operationally independent, there is no definite order in which they will generate interrupts. For example, device X may request an interrupt while an interrupt caused by device Y is being serviced, or several devices may request interrupts at exactly the same time. This gives rise to a number of questions:
1. How can the processor recognize the device requesting an interrupt?
2. Given that different devices are likely to require different interrupt -service routines, how can the processor obtain the starting address of the appropriate routine in each case?
3. Should a device be allowed to interrupt the processor while another interrupt is being serviced?
4. How should two or more simultaneous interrupt requests be handled?
If two devices have activated the line at the same time, it must be possible to break the tie and elect one of the two requests for service. When the interrupt-service routine for the selected device has been completed, the second request can be serviced.

5.Interface circuits: An Input/output (I/O) interface consists of the circuitry required to connect an I/O device to a computer bus. On one side of the interface we have the bus signals for address, data, and control. On the other side we have a data path with its associated controls to transfer data between the interface and the I/O device. This side is called a port, and it can be classified as either a parallel or a serial port. A parallel port transfers data in the form of a number of bits, typically 8 or 16, simultaneously to or from the device. A serial port transmits and receives data one bit at a time. Communication with the bus is the same for both formats; the conversion from the parallel to the serial format, and vice versa, takes place inside the interface circuit. I/O interface does the following:
1. Provides a storage buffer for at least one word of data (or one byte, in the case of byte-oriented devices)
2. Contains status flags that can be accessed by the processor to determine whether the buffer is full (for input) or empty (for output)
3. Contains address-decoding circuitry to determine when it is being addressed by the processor
4. Generates the appropriate timing signals required by the bus control scheme
5. Performs any format conversion that may be necessary to transfer data between the bus and the I/O device, such as parallel-serial conversion in the case of a serial port

Parallel port: Figure 5.20 shows the hardware components needed for connecting a keyboard to a processor. A typical keyboard consists of mechanical switches that are normally open. When a key is pressed, its switch closes and establishes a path for an electrical signal. This signal is detected by an encoder circuit that generates the ASCII code for the corresponding character. A difficulty with such push-button switches is that the contacts bounce when a key is pressed. Although bouncing may last only one or two milliseconds, this is long enough for the computer to observe a single pressing of a key as several distinct electrical events; this single pressing could be erroneously interpreted as the key being pressed and released rapidly several times. The effect of bouncing must be eliminated. We can do this in two ways: A simple de-bouncing circuit can be included, or a software approach can be used. When debouncing is implemented in software, the I/O routine that reads a character from the keyboard waits long enough to ensure that bouncing has subsided. Figure 5.20 illustrates the hardware approach; debouncing circuits are included as a part of the encoder block.

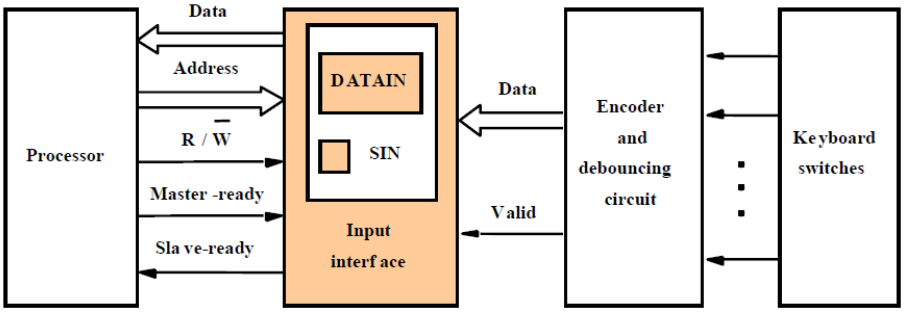


Figure 5.20 Keyboard to processor connection.

The output of the encoder consists of the bits that represent the encodedcharacter and one control signal called Valid, which indicates that a key is being pressed. This information is sent to the interface circuit, which contains a data register, DATAIN, and a status flag, SIN. When a key is pressed, the valid signal changes from 0 to 1,, causing the ASCII code to be loaded into DATAIN and SIN to be set to 1. The status flag SIN is cleared to 0 when the processor reads the contents of the DATAIN register. The interface circuit is connected to an asynchronous bus on which transfers are controlled using the handshake signals Master-ready and Slave-ready. The third control line, R/W distinguishes read and write transfers.

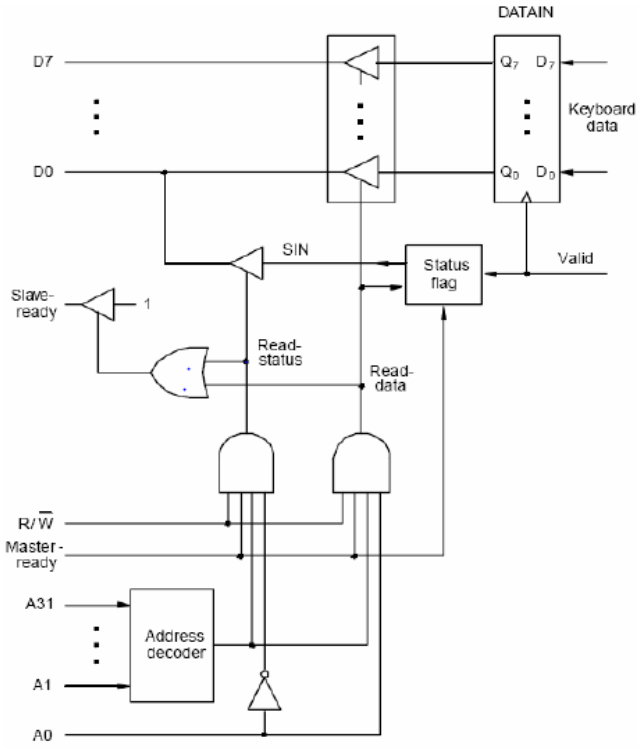


Figure 5.21 Input interface circuit.

Figure 5.21 shows a suitable circuit for an input interface. The output lines of the DATAIN register are connected to the data lines of the bus by means of three-state drivers, which are turned on when the processor issues a read instruction with the ad- dress that selects this register. The SIN signal is generated by a status flag circuit. This signal is also sent to the bus through a three-state driver. It is connected to bit DO, which means it will appear as bit 0 of the status register. Other bits of this register do not contain valid information. An address decoder is used to select the input interface when the high-order 31 bits of an address correspond to any of the addresses assigned to this interface. Address bit AO determines whether the status or the data registers is to be read when the Master-ready signal is active. The control handshake is accomplished by activating the Slaveready signal when either Read-status or Read-data is equal to 1.

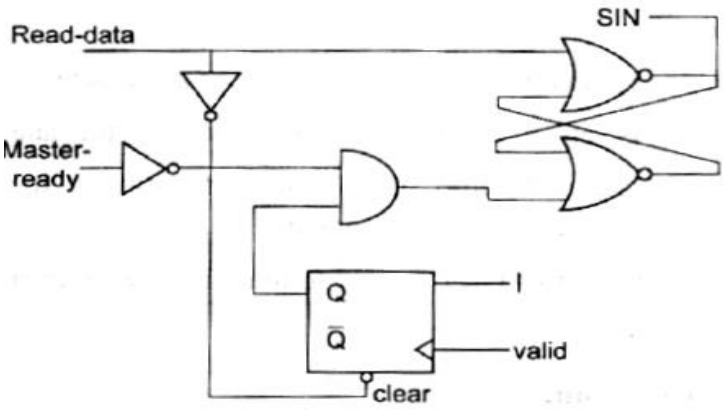


Figure 5.22 Circuit for the status Rag block in Figure 5.21.

A possible implementation of the status flag circuit is shown in Figure 5.21. An edgetriggered D flip-flop is set to 1 by a rising edge on the Valid signal line. this event changes the state of the NOR latch such that SIN is set to 1. The state of this latch must not change while SIN is being read by the processor. Hence, the circuit ensures that SIN can be set only while Masterready is equal to

0. Both the flip- flop and the latch are reset to 0 when Read-data is set to 1 to read the DATAIN register.

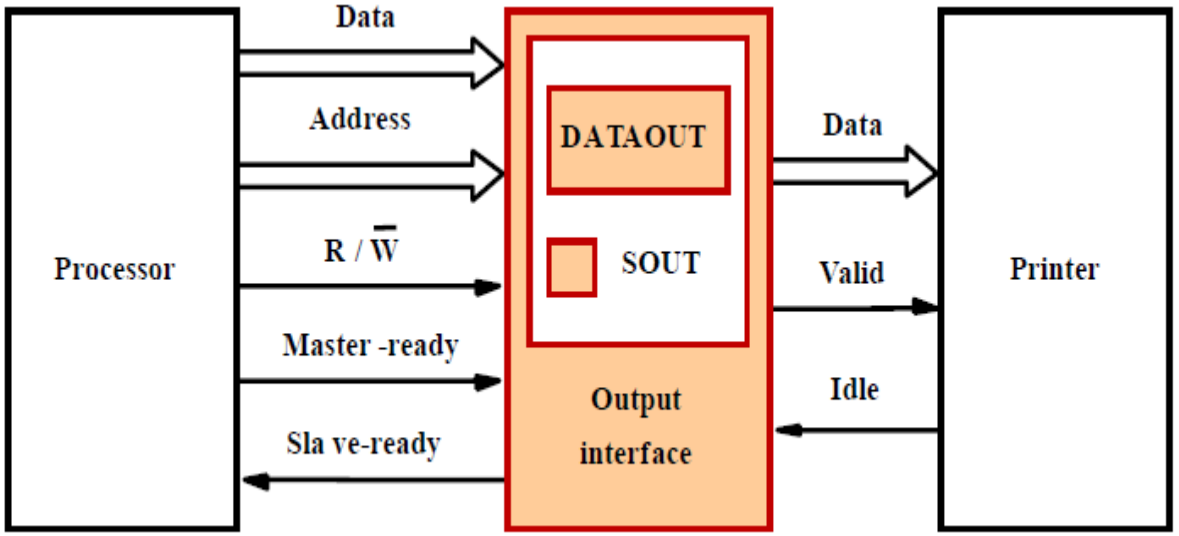


Figure 5.23 Printer to processor connection.

Let us now consider an output interface that can be used to connect an output device, such as a printer, to a processor, as shown in Figure 5.23. The printer operates under control of the handshake signals Valid and Idle in a manner similar to the handshake used on the bus with the Master-ready and Slave-ready signals. When it is ready to accept a character, the printer asserts its Idle signal. The interface circuit can then place a new character on the data lines and activate the Valid signal. In response, the printer starts printing the new character and negates the Idle signal, which in turn causes the interface to deactivate the Valid signal.

Serial port

A serial port is used to connect the processor to I/O devices that require transmission of data one bit at a time. The key feature of an interface circuit for a serial port is that it is capable of communicating in a bit-serial fashion on the device side and in a bit-parallel fashion on the bus side. The transformation between the parallel and serial formats is achieved with shift registers that have parallel access capability. A block diagram of a typical serial interface is shown in Figure 5.27. It includes the familiar DATAIN and DATAOUT registers. The input shift register accepts bit-serial input from the I/O device. When all 8 bits of data have been received, the contents of this shift register are loaded in parallel into the DATAIN register. Similarly, output data in the DATAOUT register are loaded into the output shift register, from which e bits are shifted out and sent to the I/O device.

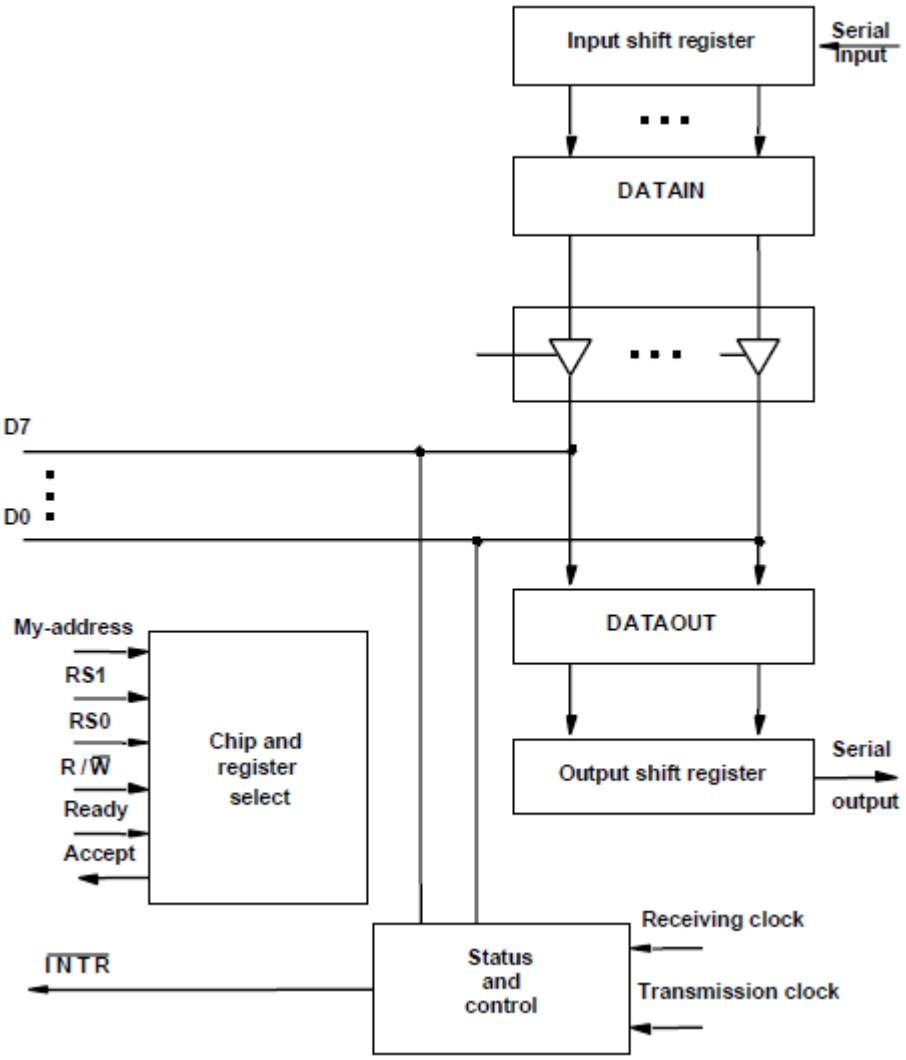


Figure 5.27 A serial interface.

The part of the interface that deals with the bus is the same as in the parallel interface described earlier. The status flags SIN and SOUT serve similar functions. The SIN flag is set to 1

- when new data are loaded in DATAIN; it is cleared to 0 when the processor reads the contents of DATAIN. As soon as the data are transferred from the input shift register into the DATAIN register, the shift register can start accepting the next 8-bit character from the I/O device. The SOUT flag indicates whether the output buffer is available. It is cleared to 0 when the processor writes new data into the DATAOUT register and set to 1 when data are transferred from DATAOUT into the output shift register.
- The double buffering used in the input and output paths are important. A simpler interface could be implemented by turning DATAIN and DATA OUT into shift registers and eliminating the shift registers in Figure 5.27. However, this would impose awkward restrictions on the operation of the I/O device; after receiving one character from the serial line, the device cannot start receiving the next character until the processor reads the contents of DATAIN. Thus, a pause would be needed between two characters to allow the processor to read the input data. With the double buffer, the transfer of the second character can begin as soon as the first character is loaded from the shift register into the DATAIN register. Thus, provided the processor reads the contents of DATAIN before the serial transfer of the second character is completed, the interface can receive a continuous stream of serial data. An analogous situation occurs in the output path of the interface.

Because it requires fewer wires, serial transmission is convenient for connecting devices that are physically far away from the computer. The speed of transmission, often given as a bit rate, depends on the nature of the devices connected. To accommodate a range of devices, a serial interface must be able to use a range of clock speeds. The circuit in Figure 5.27 allows separate clock signals to be used for input and output operations for increased flexibility. Because serial interfaces play a vital role in connecting I/O devices, several widely used standards have been developed. A standard circuit that includes the features of our example in Figure 5.27 is known as a Universal Asynchronous Receiver Transmitter (UART). It is intended for use with low-speed serial devices. Data transmission is performed using the asynchronous start-stop format. To facilitate connection to communication links, a popular standard known as RS-232-C was developed.

6.SCSI (Small Computer System Interface): Data transfers on the SCSI bus are always controlled by the target controller. To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it. Then the controller starts a data transfer operation to receive a command from the initiator. Assume that the processor wishes to read a block of data from a disk drive and that these data are stored in two disk sectors that are not contiguous. The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

1. The SCSI controller, acting as an initiator, contends for control of the bus.
2. When the 'initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.
3. The target starts an output operation (from initiator to target); in response to this, the initiator sends a command specifying the required read operation.
4. The target, realizing that it first needs to perform a disk seek operation, sends a message to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
5. The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read operation. Then, it reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data to the initiator, the target requests control of the bus. After it wins arbitration, it res elects the initiator controller, thus restoring the suspended connection.
6. The target transfers the contents of the data buffer to the initiator and then suspends the connection again. Data are transferred either 8 or 16 bits in parallel, depending on the width of the bus.
7. The target controller sends a command to the disk drive to perform another seek operation. Then, it transfers the contents of the second disk sector to the initiator, as before. At the end of this transfer, the logical connection between the two controllers is terminated.
8. As the initiator controller receives the data, it stores them into the main memory using the DMA approach.
9. The SCSI controller sends 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. The SCSI bus standard defines a wide range of control messages that can be ex- changed between the controllers to handle different types of I/O devices. Messages are also defined to deal with various error or failure conditions that might arise during device operation or data transfer.

7.USB (Universal Serial Bus).: Universal Serial Bus (USB) is an industry standard developed through a collaborative effort of several computer and communications companies, including Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, Nortel Networks, and Philips. USB is a simple and low cost

mechanism to connect the devices such as keyboards, mouse, cameras, speakers, printer and display devices to the computer.

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

- 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: Only a few ports are provided in a typical computer. 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. An objective of the USB is to make it possible to add many devices to a computer system at any time, without opening the computer box.

Device Characteristics: The different kinds of devices may be connected to a computer cover a wide range of functionality. The speed, volume, and timing constraints associated with data transfers to and from such devices vary significantly. In the case of a keyboard, one byte of data is generated every time a key is pressed, which may happen at any time. These data should be transferred to the computer promptly. Since the event of pressing a key is not synchronized to any other event in -8 computer system, the data generated by the keyboard are called asynchronous. Furthermore, the rate at which the data are generated is quite low. It is limited by the speed of the human operator to about 100 bytes per second, which is less than 1000 bits per second.

Let us consider a different source of data. Many computers have a microphone either externally attached or built in. The sound picked up by the microphone produces an analog electrical signal, which must be converted into a digital form before it can be handled by the computer. This is accomplished by sampling the analog signal periodically. For each sample, an analog-to-digital (A/D) converter generates an n-bit number representing the magnitude of the sample. The number of bits, n, is selected based on, the desired precision with which to represent each sample. Later, when these data are sent to a speaker, a digital-to-analog (D/A) converter is used to restore the original analog signal from the digital format. The sampling process yields a continuous stream of digitized samples that arrive at regular intervals, synchronized with the sampling clock. Such a data stream is called isochronous, meaning that successive events are separated by equal periods of time.

Plug-and-play: The plug-and-play feature means that a new device, such as an additional speaker, can be connected at any time while the system is operating. The system should detect the existence of this new device automatically, identify the appropriate device-driver soft- ware and any other facilities needed to service that device, and establish the appropriate addresses and logical connections to enable them to communicate. The plug-and-play requirement has many implications at all levels in the system, from the hardware to the operating system and the applications software. One of the primary objectives of the design of the USB has been to provide a plug-and-play capability.

8.USB Architecture: A serial transmission format has been chosen for the USB because a serial bus satisfies the low-cost and flexibility requirements. Clock and data information are encoded together and transmitted as a single signal. Hence, there are no limitations on clock frequency or distance arising from data skew. Therefore, it is possible to provide a high data transfer bandwidth by using a high clock frequency. As pointed out earlier, the USB offers three bit rates, ranging from 1.5 to 480 megabits/s, to suit the needs of different I/O devices.
To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure shown in Figure 5.33. Each node of the tree has a device called a hub, which acts as an intermediate control point between the host and the I/O devices. At the root of the tree, a root hub connects the entire tree to the host computer. The leaves of the tree are the I/O devices being served (for example, keyboard, speaker, or digital TV), which are called functions in USB terminology.

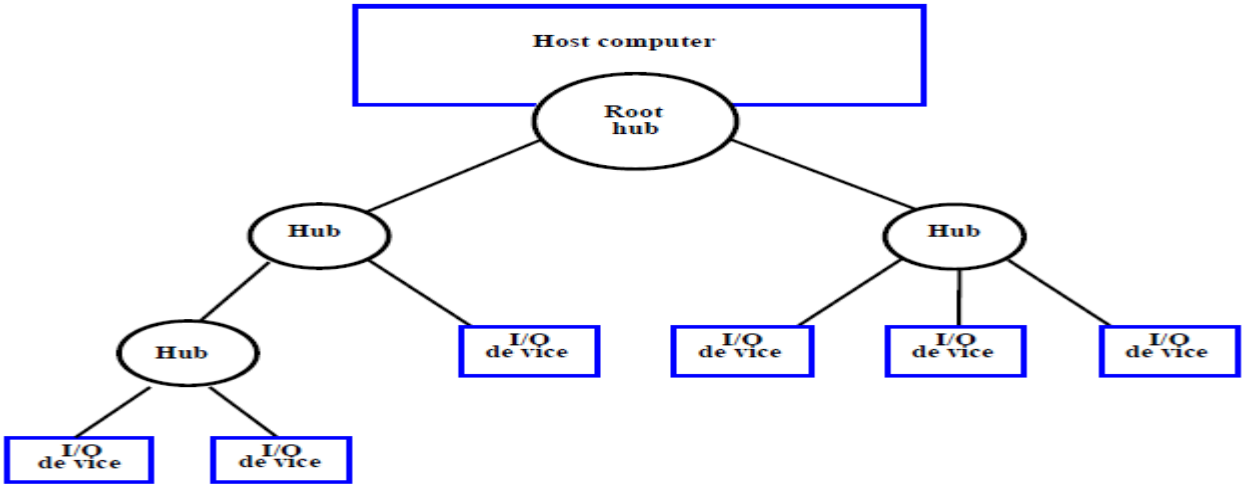


Figure 5.33 Universal Serial Bus tree structure.
The tree structure enables many devices to be connected while using only simple point -topoint serial links. Each hub has a number of ports where devices may be connected, including other hubs. In normal operation, a hub copies a message that it receives from its upstream connection to all its downstream ports. As a result, a message sent by the host computer is broadcast to ill VO devices, but only the addressed device will respond to that message. A message from an I/O device is sent only upstream towards the root of the tree and is not seen by other devices. Hence, the USB enables the host to communicate with the I/O devices, but it does not enable these devices to communicate with each other.

The USB operates strictly on the basis of polling. A device may send a message only in response to a poll message from the host. Hence, upstream messages do not encounter conflicts or interfere with each other, as no two devices can send messages at the same time. This restriction allows hubs to be simple, Low-cost devices.

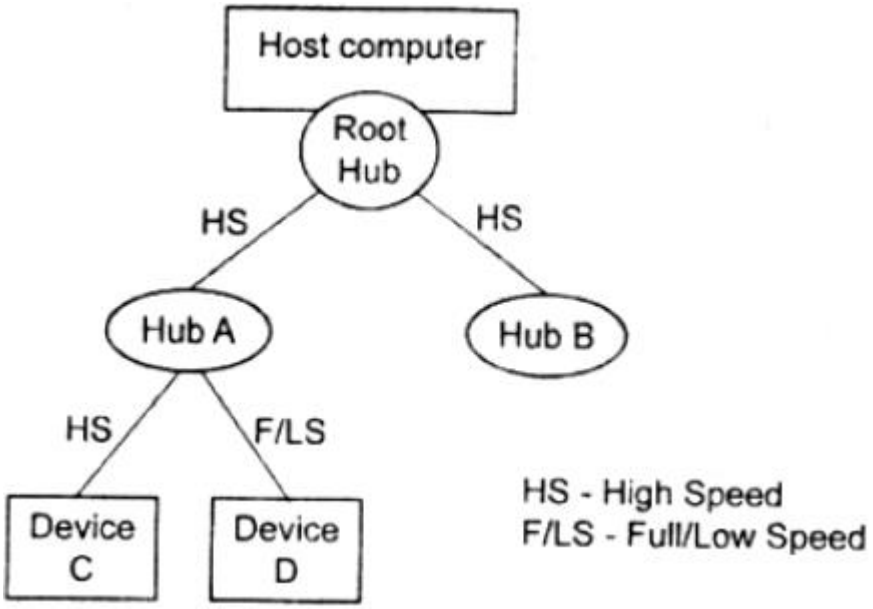


Figure 5.34 USB Split bus operations.

The mode of operation described above is observed for all devices operating at either low speed or full speed. However, one exception has been necessitated by the introduction of highpeed operation in USB version 2.0. Consider the situation in Figure 5.34. Hub A is connected to the root hub by a high-speed link. This hub serves one high-speed device, C, and one low-speed device, D. Normally, a message to device D would be sent at low speed from the root hub. At 1.5 megabits/s, even a short message takes several tens of microseconds. For the duration of this message, no other data transfers can take place, thus reducing the effectiveness of the high-speed links and introducing unacceptable delays for high-speed devices. To mitigate this problem, the USB protocol requires that a message transmitted on a high-speed link is always transmitted at high speed, even when the ultimate receiver is a low-speed device. Hence, a message intended for device D is sent at high speed from the root hub to hub A, then forwarded at low speed to device D. The latter transfer will take a long time, during which high-speed traffic to other nodes is allowed to continue. For example, the root hub may exchange several messages with device C while the low-speed message is being sent from hub A to device D. During this period, the bus is said to be split between high-speed and low-speed traffic. The message to device D is preceded and followed by special commands to hub A to start and end the split-traffic mode of operation, respectively.
The USB standard specifies the hardware details of USB interconnections as well as the organization and requirements of the host software. The purpose of the USB software is to provide bidirectional communication links between application software and I/O devices. These links are called pipes. Any data entering at one end of a pipe is delivered at the other end. Issues such as addressing, timing, or error detection and recovery are handled by the USB protocols. The software that transfers data to or from a given IJO device is called the device driver for that device. The device drivers depend on the characteristics of the devices they support. Hence, a more precise description of the USB pipe is that it connects an VO device to its device driver. It is established when a device is connected and assigned a unique address by the USB software. Once established, data may flow through the pipe at any time.

Addressing: I/O devices are normally identified by assigning them a unique memory address. In fact, a device usually has several addressable locations to enable the software to send and receive control

and status information and to transfer data. When a USB is connected to a host computer, its root hub is attached to the processor bus, where it appears as a single device. The host software communicates with individual devices attached to the USB by sending packets of information, which the root hub forwards to the appropriate device in the USB tree.
Each device on the USB, whether it is a hub or an IJO device, is assigned a 7 -bit address. This address is local to the USB tree and is not related in any way to the addresses used on the processor bus. A hub may have any number of devices or other hubs connected to it, and addresses are assigned arbitrarily. When a device is first connected to a hub, or when it is powered on, it has the address 0. The hardware of the hub to which this device is connected is capable of detecting that the device has been connected, and it records this fact as part of its own status information. Periodically, the host polls each hub to collect status information and learn about new devices that may have been added or disconnected.

9.USB Protocol: All information transferred over the USB is organized in packets, where a packet consists of one or more bytes of information. There are many types of packets that perform a variety of control functions. The information transferred on the USB can be divided into two broad categories: control and data. Control packets perform such tasks as addressing a device to initiate data transfer, acknowledging that data have been received correctly, or indicating an error. Data packets carry information that is delivered to a device. For example, put and output data are transferred inside data packets.

A packet consists of one or more fields containing different kinds of information. The first field of any packet is called the packet identifier, PID, which identifies the type of that packet. There are four bits of information in this field, but they are transmitted twice. The first time they are sent with their true values, and the second time with each bit complemented, as shown in Figure 5.35(a). This enables the receiving device to verify that the PID byte has been received correctly.

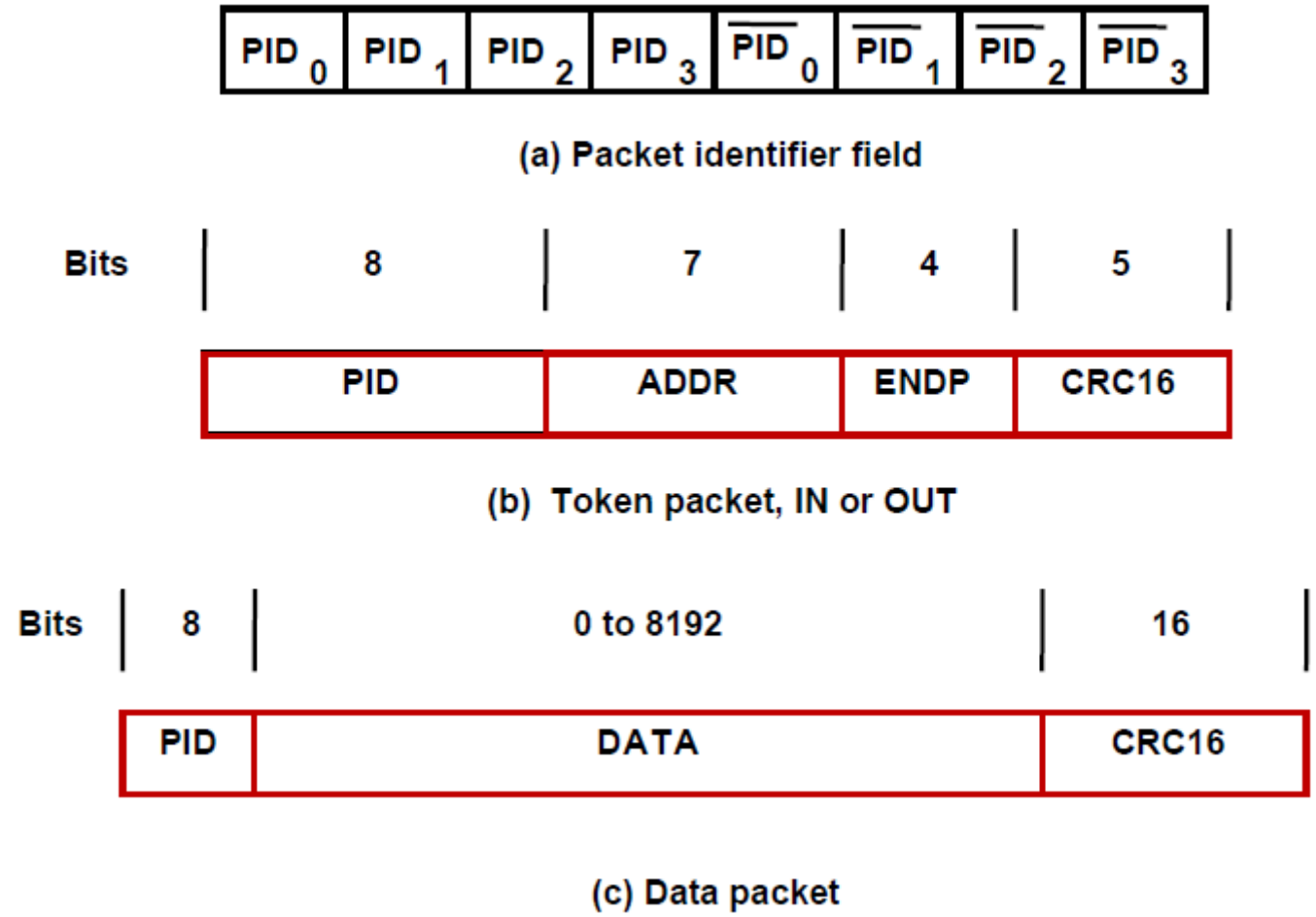


Figure 5.35. USB packet format.

The four PID bits identify one of 16 different packet types. Some control packets, such as ACK (Acknowledge), consist only of the PID byte. Control packets used for controlling data transfer operations are called token packets. They have the format shown in Figure 5.35(b). A token packet starts with the PID field, using one of two PID values to distinguish between an IN packet and an OUT packet, which control input and output transfers, respectively. The PID field is followed by the 7 -bit address of a device and the 4-bit endpoint number within that device. The packet ends with 5 bits for error checking, using a method called cyclic redundancy check (CRC). The CRC bits are computed based on the contents of the address and endpoint fields. By performing an inverse computation, the receiving device can determine whether the packet has been received correctly. Data packets, which carry input and output data, have the format shown in Figure 4.45c. The packet identifier field is followed by up to 8192 bits of data, then 16 error-checking bits. Three different PID patterns are used to identify data packets, so that data packets may be numbered 0, 1, or 2. Note that data packets do not carry a device address or an endpoint number. This information is included in the IN or OUT token packet that initiates the transfer. Consider an output device connected to a USB hub, which in turn is connected to a host computer. An example of an output operation is shown in Figure 5.36. The host computer sends a token packet of type OUT to the hub, followed by a data packet containing the output data. The PID field of the data packet identifies it as data packet number o. The hub verifies that the transmission has been error free by checking the error control bits, and then sends an acknowledgment packet (ACK) back to the host. The hub forwards the token and data packets downstream. All I/O devices receive this sequence of packets, but only the device that recognizes its address in the token packet accepts the data in the packet that follows. After verifying that transmission has been error free, it sends an ACK packet to the hub.

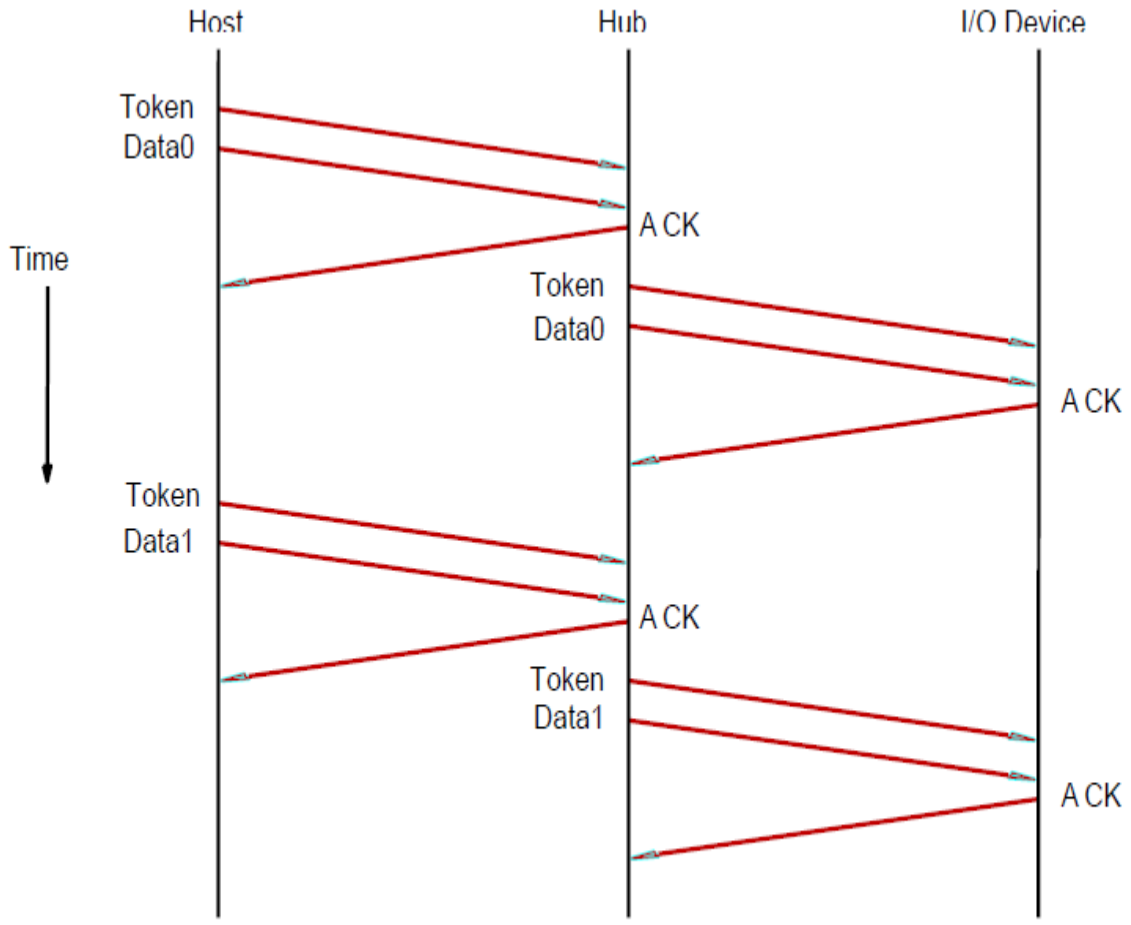


Figure 5.36 An output transfer

Successive data packets on a full-speed or low-speed pipe carry the numbers 0 and 1, alternately. This simplifies recovery from transmission errors. If a token, data, or acknowledgment packet is lost as a result of a transmission error, the sender resends the entire sequence. By checking the data packet number in the PID field, the receiver can detect and discard duplicate packets. High-speed data packets are sequentially numbered 0, 1, 2, 0, and so on. Input operations follow a similar procedure. The host sends a token packet of type IN containing the device address. In effect, this packet is a poll asking the device to send any input data it may have. The device responds by sending a data packet followed by an ACK. If it has no data ready, it responds by sending a negative acknowledgment (NAK) instead.

Electrical characteristics: The cables used for USB connections consist of four wires. Two are used to carry power, 5 V and Ground. Thus, a hub or an I/O device may be powered directly from the bus, or it may have its own external power connection. The other two wires are used to carry data. Different signaling schemes are used for different speeds of transmission. At low speed, 1s and 0s are transmitted by sending a high voltage state (5 V) on one or the other of the two signal wires. For high-speed links, differential transmission is used.

10.I/O device

Input / Output and Storage Devices		
Input	**Output**	**Storage**
Keyboard	Monitor	Floppy Disk
Mouse	Printers (all types)	Diskette
Scanner	Audio Card	Hard Disk
Touchpad	Plotters	Disk Cartridge
Pointing Sticks	LCD Projection Panels	CD-ROM
Joysticks	Computer Output Microfilm (COM)	Optical Disk
Trackballs	Facsimile (FAX)	Magnetic Tape
Touch Screen	Speaker(s)	Cartridge Tape
Light Pen	.	Reel Tape
Digitizer	.	PC Card
Graphics Tablet	.	*RAID
Pen Input	.	*Memory Button
Microphone	.	*Smart Card
Electronic Whiteboard	.	*Optical Memory Card

Keyboard: The computer keyboard is used to enter text information into the computer, as when you type the contents of a report. The keyboard can also be used to type commands directing the computer to perform certain actions. In addition to the keys of the main keyboard (used for typing text), keyboards usually also have a numeric keypad (for entering numerical data efficiently), a bank of editing keys (used in text editing operations), and a row of function keys along the top (to easily invoke certain program functions). Laptop computers, which don't have room for large keyboards, often include a "fn" key so that other keys can perform double duty (such as having a numeric keypad function embedded within the main keyboard keys).

Mouse: The mouse pointing device sits on your work surface and is moved with your hand. In older mice, a ball in the bottom of the mouse rolls on the surface as you move the mouse, and internal rollers sense the ball movement and transmit the information to the computer via the cord of the mouse. The newer optical mouse does not use a rolling ball, but instead uses a light and a small optical sensor to detect the motion of the mouse by tracking a tiny image of the desk surface. Optical mice avoid the problem of a dirty mouse ball, which causes regular mice to roll unsmoothly if the mouse ball and internal rollers are not cleaned frequently.

Scanners: A scanner is a device that images a printed page or graphic by digitizing it, producing an image made of tiny pixels of different brightness and color values which are represented numerically and sent to the computer. Scanners scan graphics, but they can also scan pages of text which are then run through OCR (Optical Character Recognition) software that identifies the individual letter shapes and creates a text file of the page's contents.

Monitors: CRT (Cathode Ray Tube), LCD (Liquid Crystal Display)
Printers: Laser Printer, Inkjet Printer, Dot matrix printer

11.Processors

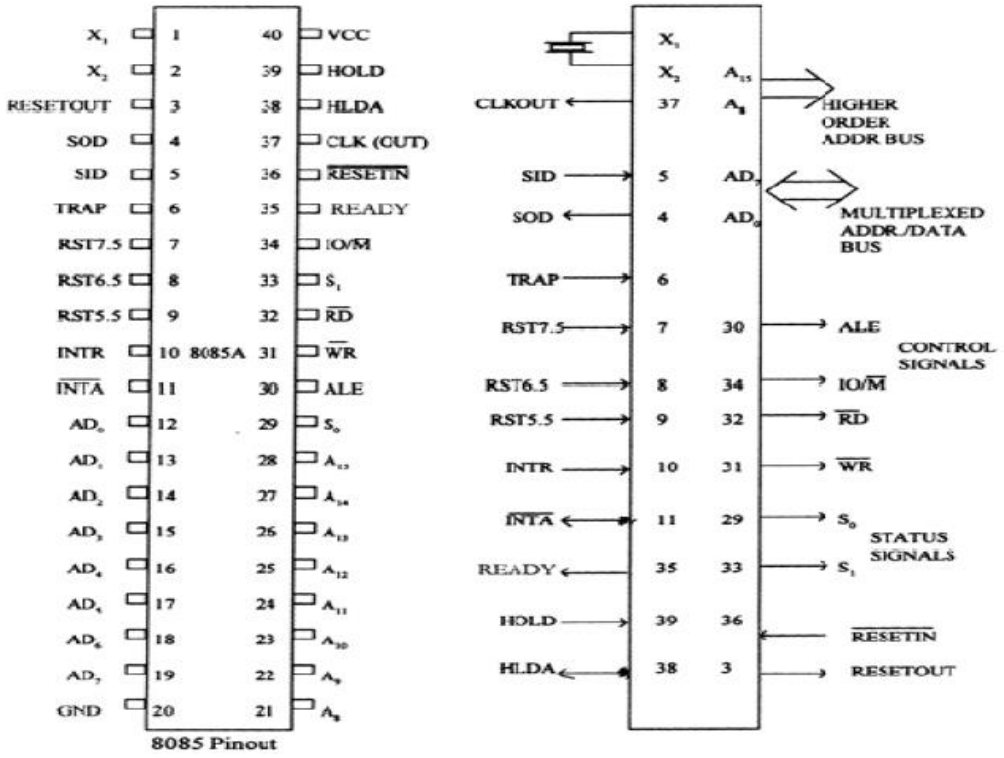


Figure 5.37 8085 processor block diagram

General-purpose registers and program counter

User	FIQ	IRQ	Supervisor	Abort	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_irq	R13-svc	R13_abt	R13_und
R14	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und
R15	R15	R15	R15	R15	R15

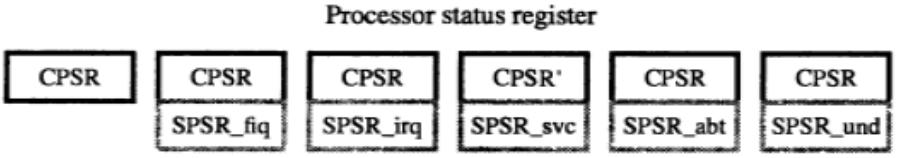


Figure 5.38 Registers of Arm Processor

Unit-4

1.Basic concepts of memory system: Computer should have a large memory to facilitate execution of programs that are large and deal with huge amounts of data. The memory should be fast, large, and inexpensive. Unfortunately, it is impossible to meet all three of these requirements simultaneously. Increased speed and size are achieved at increased cost. To solve this problem, much work has gone into developing clever structures that improve the apparent speed and size of the memory, yet keep the cost reasonable.

The maximum size of the memory that can be used in any computer is determined by the addressing scheme. For example, a 16-bit computer that generates 16-bit addresses is capable of addressing up to 216 = 64K (65536) memory locations. Similarly, machines whose instructions generate 32-bit addresses can utilize a memory that contains up to 232 = 4G (giga) memory locations, whereas machines with 4O-bit addresses can access up to 240 = 1 T (tera) locations. The number of locations represents the size of the address space of the computer.

From the system standpoint, we can view the memory unit as a black box. 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 con1rollines Read / Write (R / W) and Memory Function Completed (MFC) for coordinating data transfers. Other control lines may be added to indicate the number of bytes to be transferred. The connection between the processor and the memory is shown schematically in Figure 4.1.

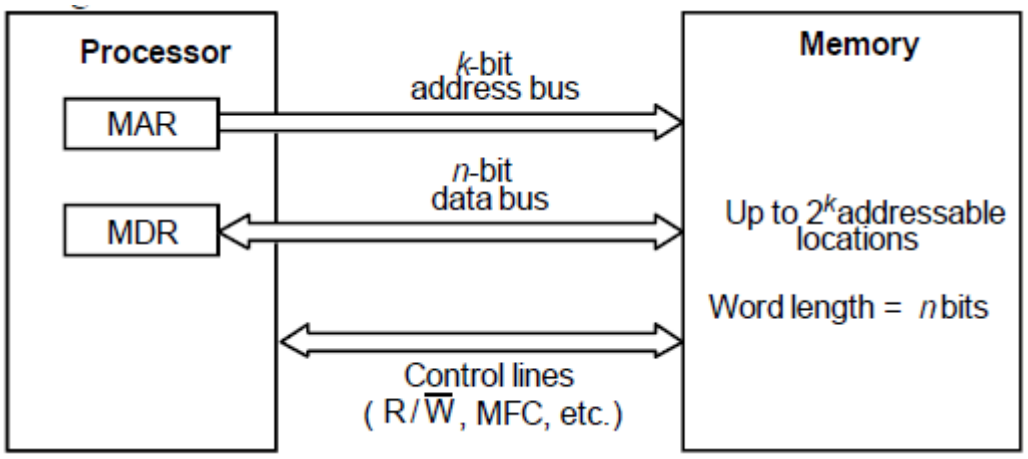


Figure 4.1 Connection of the memory to the processor

- The processor reads data from the memory by loading the address of the required memory location into the MAR register and setting the R / W line to 1. The memory responds by placing the data from the addressed location onto the data lines, and confirms this action by asserting the MFC signal. Upon receipt of the MFC signal, the processor loads the data on the data lines into the MDR register. The processor writes data into a memory location by loading the address of this location into MAR and loading the data into MDR. It indicates that a write operation is involved by setting the R/ W line to 0. If read or write operations involve consecutive address locations in the main memory, then a "block transfer" operation can be performed in which the only address sent to the memory is the one that identifies the first location.
- The time between the Read and the MFC signals is referred to as the memory access time. The memory cycle time is the minimum time delay required between the initiations of two successive memory operations, If any location can be accessed for a Read or Write operation some fixed amount of time that is independent of the location's address in a memory unit is called random-access memory (RAM). One way to reduce the memory access time is to use a cache memory. This is a small, fast memory that is inserted between the larger, slower main memory and the processor. It holds the currently active segments of a program and their data.
- Virtual memory is used to increase the apparent size of the physical memory. Data are addressed in a virtual address space that can be as large as the addressing capability of the processor. But at any given time, only the active portion of this space is mapped onto locations in the physical memory. The remaining virtual addresses are mapped onto the bulk: storage devices used, which are usually magnetic disks. 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.

2. Semiconductor RAM Memories: Semiconductor memories are available in a wide range of speeds. Their cycle times range from l00ns to less than 10 ns.

INTERNAL ORGANIZATION OF MEMORY CHIPS: Memory cells are usually organized in the form of an array, in which each cell is capable of storing one bit of information. A possible organization is illustrated in Figure 4.2.

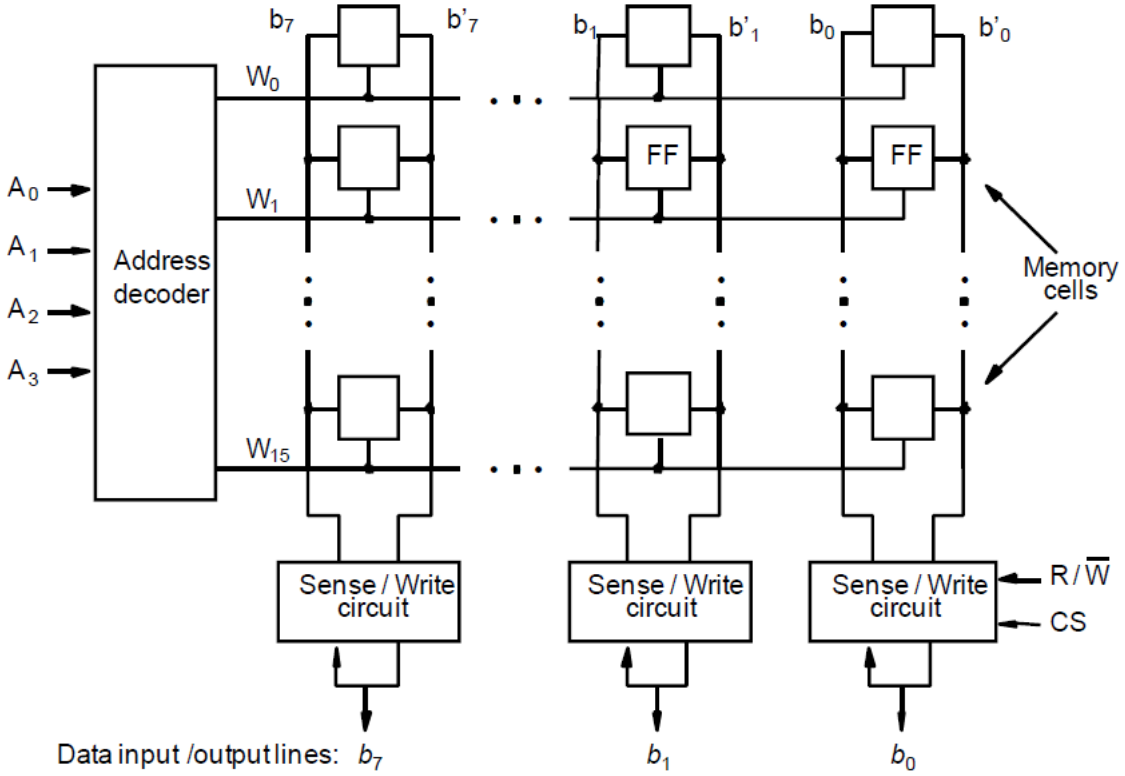
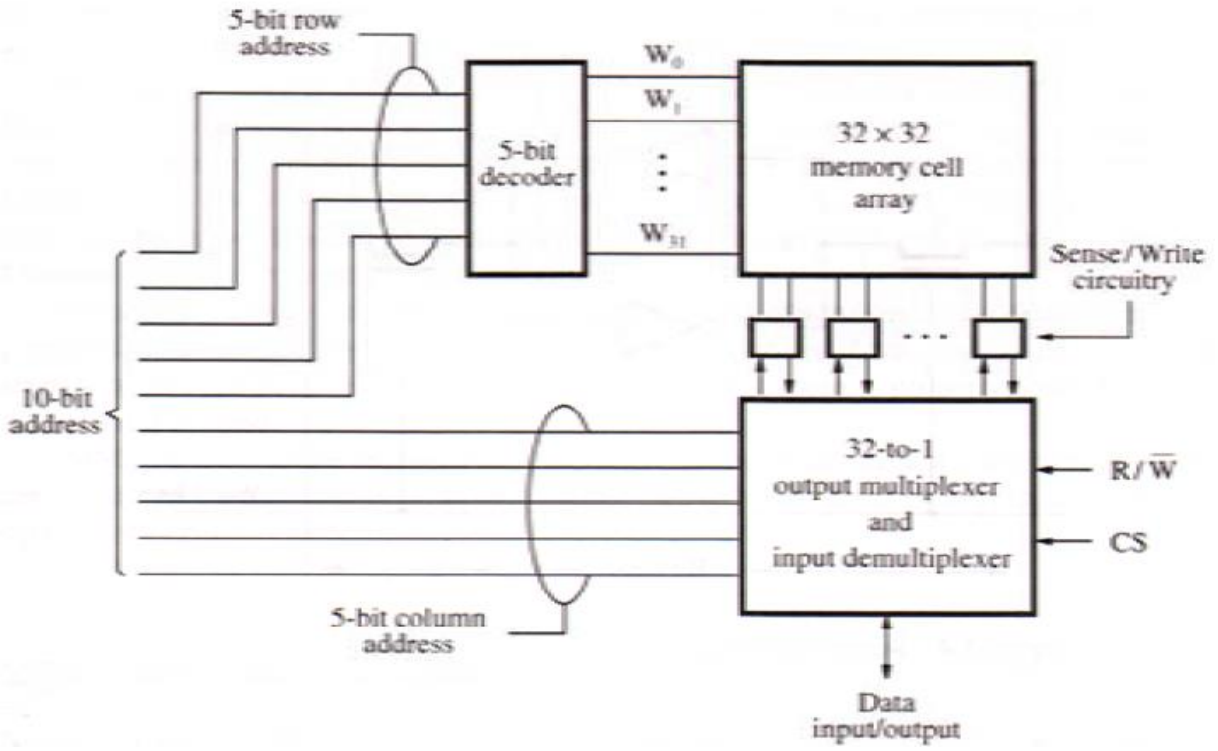


Figure 4.2 Organization of bit cell in a memory chip

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

Figure 5.2 is an example of a very small memory chip consisting of 16 words of 8 bits each. This is referred to as a 16 x 8 organization. The data input and the data output of each Sense/Write circuit are connected to a single bidirectional data line that, can be connected to the data bus of a computer. Two control lines, R/W and CS, are provided in addition to address and data lines. The R/W (Read/Write) input specifies the required operation, and the CS (Chip Select) input selects a given chip in a multichip memory system.

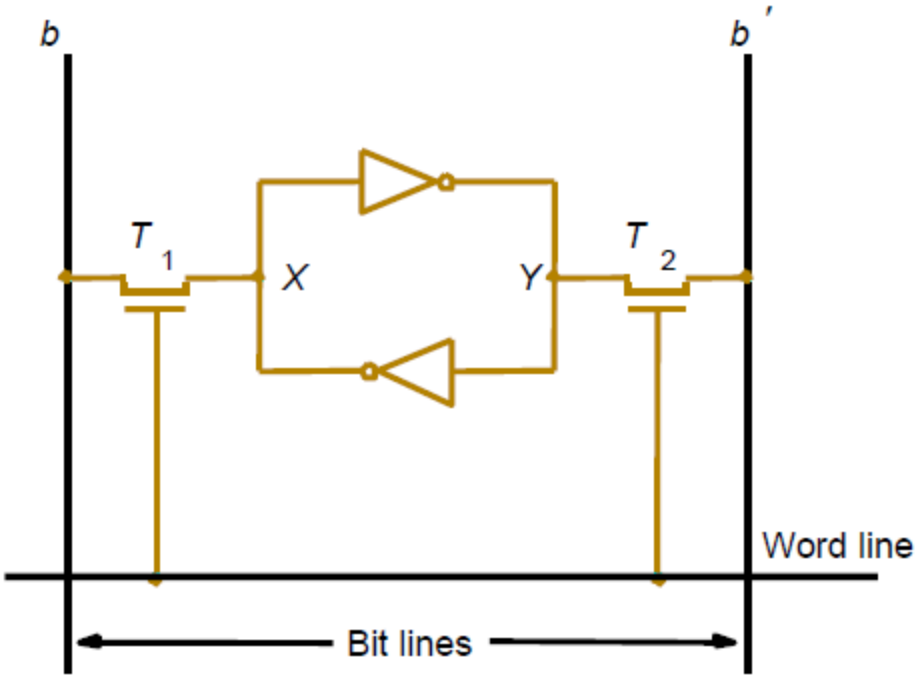


- The memory circuit in Figure 4.2 stores 128 bits and requires 14 external connections for address, data, and control lines. Of course, it also needs two lines for power supply and ground connections. Consider now a slightly larger memory circuit, one that has l K (1024) memory cells.
- This circuit can be organized as a 128 x 8 memory, requiring a total of 19 external connections. Alternatively, the same number of cells can be organized into a l K x 1 format. In this case, a 100bit address is needed, but there is only one data line, resulting in 15 external connections. Figure 4.3 shows such an organization.
- The required 100bit address is divided into two groups of 5 bits each to form the row and column addresses for the cell array. A row address selects a row of 32 cells, all of which are accessed in parallel. However, according to the column address, only one of these cells is connected to the external data line by the output multiplexer and input demultiplexer. For an example, a 4M-bit chip may have a 512K x 8 organization, in which case 19 address and 8 data input/output pins are needed.

3.STATIC MEMORIES: Static memories are the Memories that consist of circuits capable of retaining their state as long as power is applied are known as static memories. Figure 4.4 illustrates how a static RAM (SRAM) cell may be implemented. Two inverters are cross-connected to form a latch. The latch is connected to two bit lines by transistors Tl and T2. These transistors act as switches that can be opened or closed under control of the word line. When the word line is at ground level, the transistors are turned off and the latch retains its state. For example, let us assume that the cell is in state 1 if the logic value at point X is 1 and at point Y is 0. This state is maintained as long as the signal on the word line is at ground level.

Read operation: In order to read the state of the SRAM cell, the word line is activated to close switches Tl and T2. If the cell is in state 1, the signal on bit line b is high and the signal on bit line b' is low. The opposite is true if the cell is in state 0. Thus, b and b' are complements of each other. Sense/Write circuits at the end of the bit lines monitor the state of b and b' and set the output accordingly.

Write operation: The state of the cell is set by placing the appropriate value on bit line b and its complement on b', and then activating the word line. This forces the cell into the corresponding state. The required signals on the bit lines are generated by the Sense/Write circuit. if Tl and T2 are turned on (closed), it lines b and b ' will have high and low signals, respectively.



<--Figure 4.4 A static RAM cell

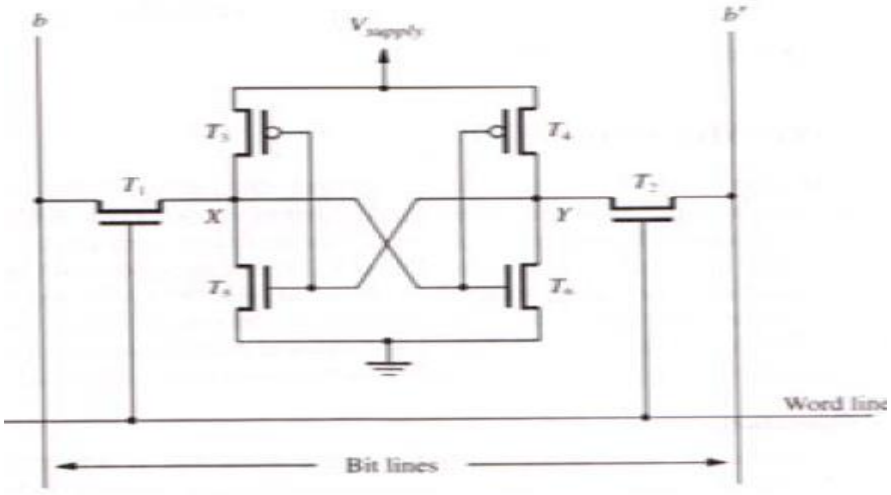


Figure 4.5 An example of a CMOS memory cell-->

The power supply voltage, Vsupply, is 5 V in older CMOS SRAMs or 3.3 V in new lowvoltage versions. Note that "continuous power is needed for the cell to retain its state. If power is interrupted, the cell's contents will be lost. When power is restored, the latch will settle into a stable state, but it will not necessarily be the same state the cell was in before the interruption. Hence, SRAMs are said to be volatile memories because their contents are lost when power is interrupted.

A major advantage of CMOS SRAMs is their very low power consumption because current flows in the cell only when the cell is being accessed. Otherwise, Tl, T2 and one transistor in each inverter are

turned off, ensuring that there is no active path between Vsupply and ground. Static RAMs can be accessed very quickly. Access times of just a few nanoseconds are found in commercially available chips. SRAMs are used in applications where speed is of critical concern.

4.Asynchronous DRAMs: Static RAMs are fast, but they come at a high cost because their cells require several transistors. Less expensive RAMs can be implemented if simpler cells are used. However, such cells do not retain their state indefinitely; hence, they are called dynamic RAMs (DRAMs). Information is stored in a dynamic memory cell in the form of a charge on a capacitor, and this charge can be maintained for only tens of milliseconds. Since the cell is required to store information for a much longer time, its contents must be periodically refreshed by restoring the capacitor charge to its full value. An example of a dynamic memory cell that consists of a capacitor, C, and a transistor, T, is shown in Figure 4.6. In order to store information in this cell, transistor T is turned on and an appropriate voltage is applied to the bit line. This causes a known amount of charge to be stored in the capacitor.

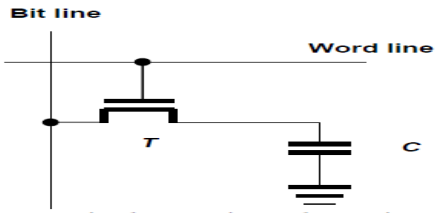


Figure 4.6 A single transistor dynamic memory cell

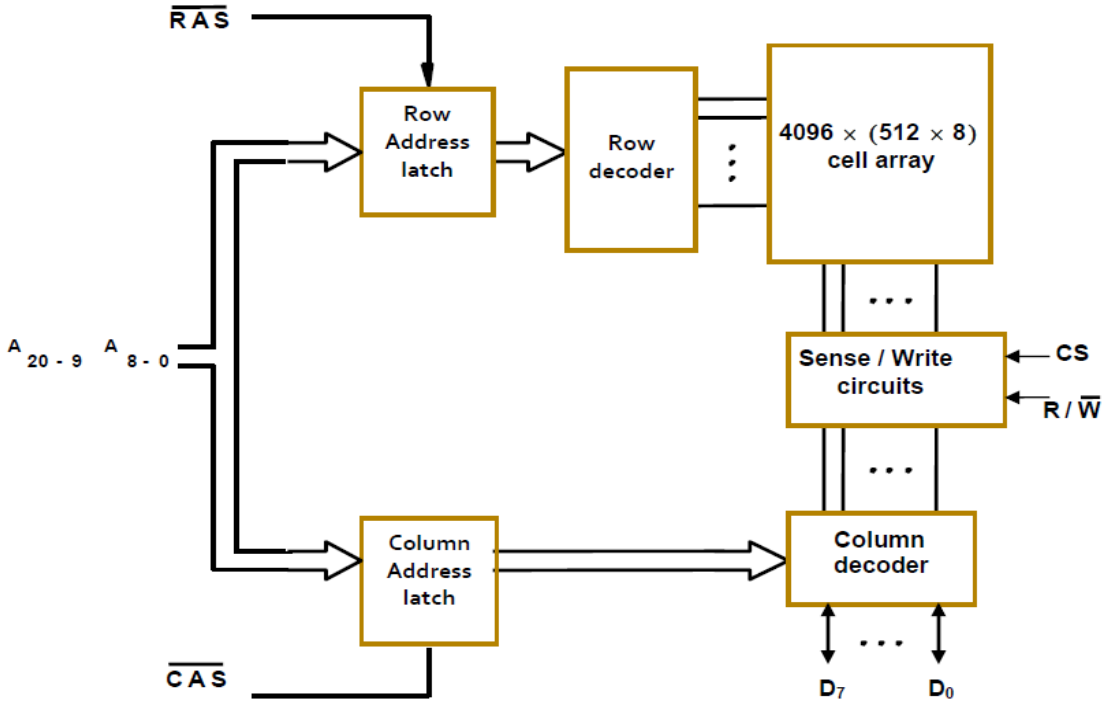


Figure 4.7 Internal Organization of a 2M x 8 dynamic memory chip

A 16-megabit DRAM chip, configured as 2M x 8, is shown in Figure 4.7. The cells are organized in the form of a 4 K x 4 K array. The 4096 cells in each row are divided into 512 groups of 8, so that a row can store 512 bytes of data. Therefore, 12 address bits are needed to select a row. Another 9 bits are needed to specify a group of 8 bits in the selected row. Thus, a 21-bit address is needed to access a byte in this memory. The high-order 12 bits and the low-order 9 bits of the address constitute the row and column addresses of a byte, respectively. To reduce the number of pins needed for external connections, the row and column addresses are multiplexed on 12 pins. During a Read or a Write

operation, the row address is applied first. It is loaded into the row address latch in response to a signal pulse on the Row Address Strobe (RAS) input of the chip.

Then a Read operation is initiated, in which all cells on the selected row are read and refreshed. Shortly after the row address is loaded, the column address is applied to the address pins and loaded into the column address latch under control of the Column Address Strobe (CAS) signal. The information in this latch is decoded and the appropriate group of 8 Sense/Write circuits are selected. If the R/W control signal indicates a Read operation, the output values of the selected circuits are transferred to the data lines, D7-0. For a Write operation, the information on the D7-0 lines is transferred to the selected circuits. This information is then used to overwrite the contents of the selected cells in the corresponding 8 columns. We should note that in commercial DRAM chips, the RAS and CAS control signals are active low so that they cause the latching of addresses when they change from high to low. To indicate this fact, these signals are shown on diagrams as RAS and CAS .

Applying a row address causes all cells on the corresponding row to be read and refreshed during both Read and Write operations. To ensure that the contents of a DRAM are maintained, each row of cells must be accessed periodically. A refresh circuit usually performs this function automatically. Many dynamic memory chips incorporate a refresh facility within the chips themselves. In this case, the dynamic nature of these memory chips is almost invisible to the user. In the DRAM described in this section, the timing of the memory device is controlled asynchronously. A specialized memory controller circuit provides the necessary control signals, RAS and CAS, that govern the timing. The processor must take into account the delay in the response of the memory. Such memories are referred to as asynchronous DRAMs.

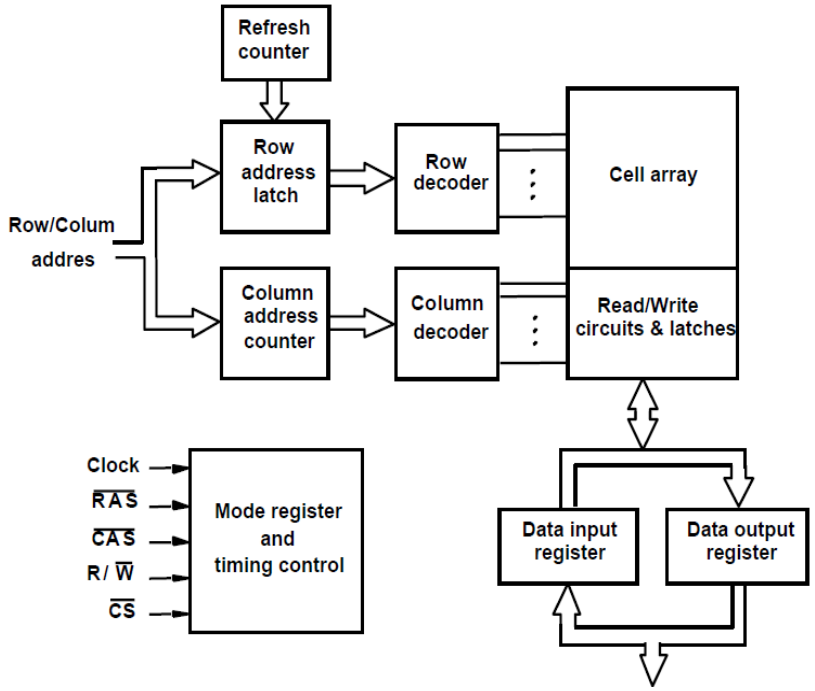
Because of their high density and low cost, DRAMs are widely used in the memory units of computers. Available chips range in size from 1M to 256M bits, and even larger chips are being developed. To reduce the number of memory chips needed in a given computer, a DRAM chip is organized to read or write a number of bits in parallel, as indicated in Figure 4.7. To provide flexibility in designing memory systems, these chips are manufactured in different organizations. For example, a 64-Mbit chip may be organized as 16M x 4, 8M x 8, or 4M x 16.

5.Synchronous DRAMs: DRAMs whose operation is directly synchronized with a clock signal are known as synchronous DRAMs (SDRAMs). Figure 4.8 indicates the structure of an SDRAM. The cell array is the same as in asynchronous DRAMs. The address and data connections are buffered by means of registers. We should particularly note that the output of each sense amplifier is connected to a latch. A Read operation causes the contents of all cells in the selected row to be loaded into these latches. But, if an access is made for refreshing purposes only, it will not change the contents of these latches; it will merely refresh the contents of the cells. Data held in the hitches that correspond to the selected column(s) are transferred into the data output register, thus becoming available on the data output pins. SDRAMs have several different modes of operation, which can be selected by writing control information into a mode register. In SDRAMs, it is not necessary to provide externally generated pulses on the CAS line to select successive columns.

Figure 4.9 shows a timing diagram for a typical burst read of length 4. First, the row address is latched under control of the RAS signal. The memory typically takes 2 or 3 clock cycles (we use 2 in the figure) to activate the selected row. Then, the column address is latched under control of the CAS signal. After a delay of one clock cycle, the first set of data bits is placed on the data lines. The SDRAM automatically increments the column address to access the next three sets of bits in the selected row, which are placed on the data lines in the next 3 clock cycles.

SDRAMs have built-in refresh circuitry. A part of this circuitry is a refresh counter, which provides the addresses of the rows that are selected for refreshing. In a typical SDRAM, each row must be

refreshed at least every 64 ms. Commercial SDRAMs can be used with clock speeds above 100 MHz. These chips are designed to meet the requirements of commercially available processors that are used in large volume. For example, Intel has defined PCl00 and PC133 bus specifications in which the system bus (to which the main memory is connected) is controlled by a 100 or 133 MHz clock, respectively. Therefore, major manufacturers of memory chips produce 100 and 133 MHz SDRAM chips. Transfers between the memory and the processor involve single words of data or small blocks of words. Large blocks, constituting a page of data, are transferred between the memory and the disks. The speed and efficiency of these transfers have a large impact on the performance of a computer system.



<------Figure 4.8 Synchronous DRAM

M

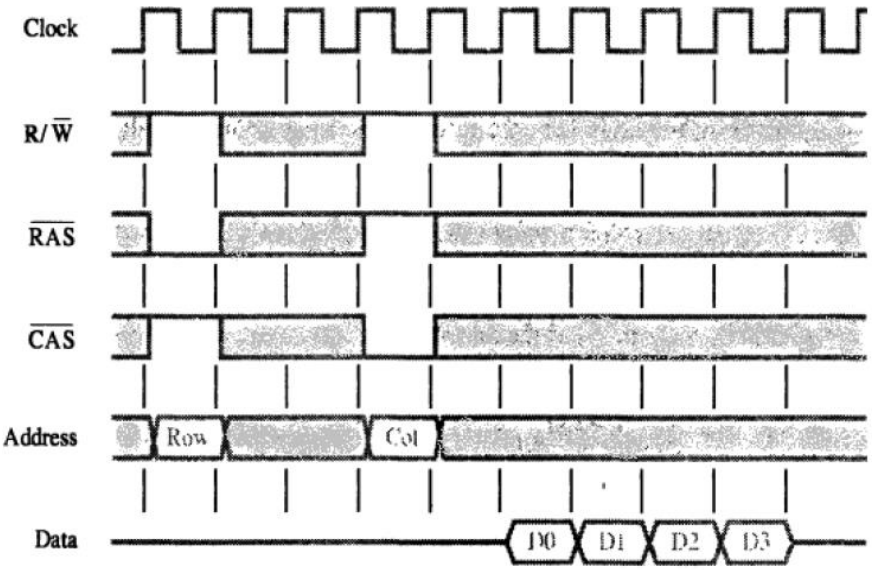


Figure 4.9 Burst read of length 4 in SDRAM------->

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

When transferring blocks of data, it is of interest to know how much time is needed to transfer an entire block. Since blocks can be variable in size, it is useful to define a performance measure in terms of the number of bits or bytes that can be transferred in one second. This measure is often referred to as the memory bandwidth. The bandwidth of a memory unit (consisting of one or more memory chips) depends on the speed of access to the stored data and on the number of bits that can be accessed in parallel.

6.STRUCTURE OF LARGER MEMORIES: Many memory chips are connected to form a much larger memory.

Static Memory System: Consider a memory consisting of 2M (2,097,152) words of 32 bits each. Figure 4.10 shows how we can implement this memory using 512K x 8 static memory chips. Each column in the figure consists of four chips, which implement one byte position. Four of these sets provide the required 2M x 32 memory. Each chip has a control input called Chip Select. When this input is set to 1, it enables the chip to accept data from or to place data on its data lines. The data output for each chip is of the three-state type. Only the selected chip places data on the data output line, while all other outputs are in the high-impedance state. Twenty one address bits are needed to select a 32-bit word in this memory. The high-order 2 bits of the address are decoded to determine which of the four Chip Select control signals should be activated and the remaining 19 address bits are used to access specific byte locations inside each chip of the selected row. The R/W inputs of all chips are tied together to provide a common Rea d/Write control (not shown in the figure).

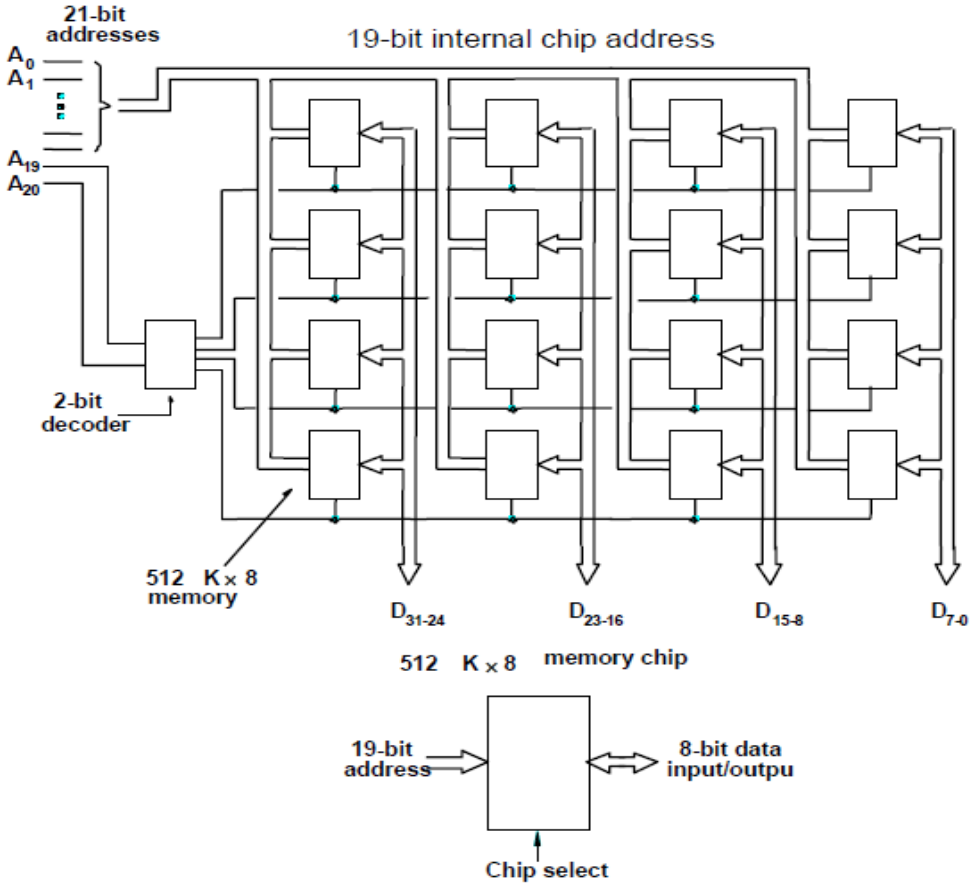


Figure 4.10 Organization of a 2M x 32 memory module using 512K x 8 static memory chips.

Dynamic Memory System: The organization of large dynamic memory systems is essentially the same as the memory shown in Figure 4.10. However, physical implementation is often done more conveniently in the form of memory modules. Modem computers use very large memories; even a small personal computer is likely to have at least 32M bytes of memory. Typical workstations have at least 128M bytes of memory. A large memory leads to better performance because more of the programs and data used in processing can be held in the memory, thus reducing the frequency of accessing the information in secondary storage. However, if a large memory is built by placing DRAM chips directly on the main system printed-circuit board that contains the processor, often referred to as a motherboard, it will occupy an unacceptably large amount of space on the board. Also, it is awkward to provide for future expansion of the memory, because space must be allocated and wiring provided for the maximum expected size. These packaging considerations have led to the development of larger memory units known as SIMMs (Single In-line Memory Modules) and DIMMs (Dual In-line Memory Modules). Such a module is an assembly of several Memory chips on a separate small board that plugs vertically into a single socket on the motherboard. SIMMs and DIMMs of different sizes are designed to use the same size socket.

7.MEMORY SYSTEM CONSIDERATION: There are several factors that determine the choice of a RAM chip for a given application. These factors are the cost, speed, power dissipation, and size of the chip. Static RAMs are generally used only when very fast operation is the primary requirement. Their cost and size are adversely affected by the complexity of the circuit that realizes the basic cell. They are used mostly in cache memories. Dynamic RAMs are the predominant choice for implementing computer main memories. The high densities achievable in these chips make large memories economically feasible.

Memory' Controller: To reduce the number of pins, the dynamic memory chips use multiplexed address inputs. The address is divided into two parts. The high-order address bits, which select a row in the cell array, are provided first and latched into the memory chip under control of the RAS signal. Then, the low-order address bits, which select a column, are provided on the same address pins and latched using the CAS signal.

A typical processor issues all bits of an address at the same time. The required multiplexing of address bits is usually performed by a memory controller circuit, which is interposed between the processor and the dynamic memory as shown in Figure 4.11. The controller accepts a complete address and the R/W signal from the processor, under control of a Request signal which indicates that a memory access operation is needed. The controller then forwards the row and column portions of the address to the memory and generates the RAS and CAS signals. Thus, the controller provides the RAS-CAS timing, in addition to its address multiplexing function. It also sends the R/W and CS signals to the memory. The CS signal is usually active low, hence it is shown as CS in Figure 4.11. Data lines are connected directly between the processor and the memory. Note that the clock signal is needed in SDRAM chips.

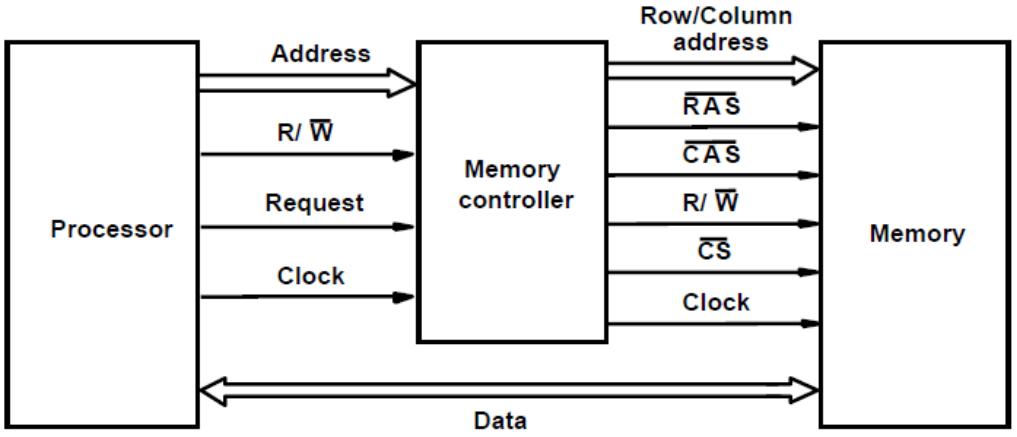


Figure 4.11 Use of memory controller

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

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

8.Rambus Memory: A very wide bus is expensive and requires a lot of space on a motherboard. An alternative approach is to implement a narrow bus that is much faster. This approach was used by Rambus Inc. to develop a proprietary design known as Rambus. The key feature of Rambus technology is a fast signaling method used to transfer information between chips. Instead of using signals that have voltage levels of either 0 or Vsupply to represent the logic values, the signals consist of much smaller voltage swings around a reference voltage, Vref. The reference voltage is about 2 V, and the two logic values are represented by 0.3 V swings above and below Vref. This type of signaling is generally known as differential signaling. Small voltage swings make it possible to have short transition times, which allows for a high speed of transmission.

Rambus requires specially designed memory chips. These chips use cell arrays based on the standard DRAM technology. Multiple banks of cell arrays are used to access more than one word at a time.

Circuitry needed to interface to the Rambus channel is included on the chip. Such chips are known as Rambus DRAMs (RDRAMs).

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

9.Read Only Memories (ROM): RAM memories are volatile memories, which mean that they lose the stored information if power is turned off. There are many applications that need memory devices which retain the stored retained even if power is turned off. A practical solution is to provide a small amount of nonvolatile memory that holds the instructions whose execution results in loading the boot program from the disk. The contents of such memory can be read as if they were SRAM or DRAM memories. But, a special writing process is needed to place the information into this memory. Since its normal operation involves only reading of stored data, a memory of this type is called read-only memory (ROM).

ROM: Figure 4.12 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.

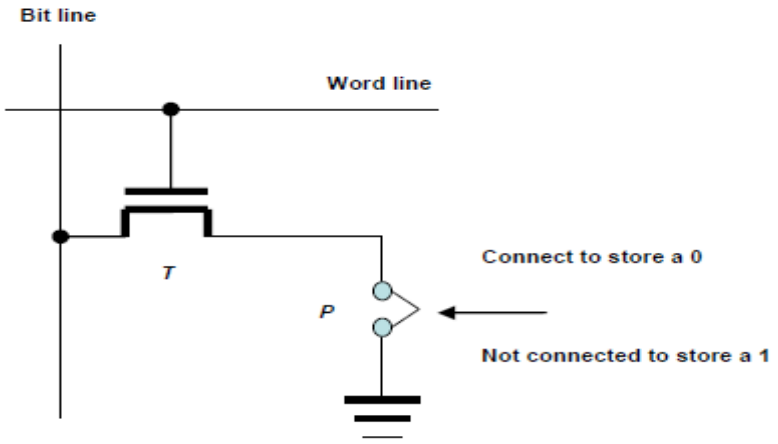


Figure 4.12 A ROM cell

PROM: Some ROM designs allow the data to be loaded by the user, thus providing a programmable ROM (PROM). Programmability is achieved by inserting a fuse at point P in Figure 4.12. Before it is programmed, the memory contains all 0s. The user can insert 1s at the required locations by burning out the fuses at these locations using high-current pulses. Of course, this process is irreversible. PROMs provide flexibility and convenience not available with ROMs. The latter are economically attractive for storing fixed programs and data when high volumes of ROMs are produced. However, the cost of preparing the masks needed for storing a particular information pattern in ROMs makes them very expensive when only a small number are required. In this case, PROMs provide a faster and considerably less expensive approach because they can be programmed directly by the user.

EPROM: Another type of ROM chip allows the stored data to be erased and new data to be loaded. Such an erasable, reprogrammable ROM is usually called an EPROM. It provides considerable

flexibility during the development phase of digital systems. Since EPROMs are capable of retaining stored information for a long time, they can be used in place of ROMs while software is being developed. In this way, memory changes and updates can be easily made. An EPROM cell has a structure similar to the ROM cell in Figure 4.12. In an EPROM cell, however, 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. This transistor can be programmed to behave as a permanently open switch, by injecting charge into it that becomes trapped inside. Thus, an EPROM cell can be used to construct a memory in the same way as the previously discussed ROM cell.

10.FLASH MEMORIES: An approach similar to EEPROM technology has more recently given rise to flash memory devices. A flash cell is based on a single transistor controlled by trapped charge, just like an EEPROM cell. While similar in some respects, there are also substantial differences between flash and EEPROM devices. In EEPROM it is possible to read and write the contents of a single cell. In a flash device it is possible to read the contents of a single cell, but it is only possible to write an entire block of cells. Prior to writing, the previous contents of the block are erased. Flash devices have greater density, which leads to higher capacity and a lower cost per bit. They require a single power supply voltage, and consume less power in their operation. The low power consumption of flash memory makes it attractive for use in portable equipment that is battery driven. 1Ypical applications include hand-held computers, cell phones, digital cameras, and MP3 music players.

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 such modules: flash cards and flash drives.

Flash Cards

One way of constructing a larger module is to mount flash chips on a small card. Such flash cards have a standard interface that makes them usable in a variety of products. A card is simply plugged into a conveniently accessible slot. Flash cards come in a variety of memory sizes. Typical sizes are 8, 32, 64 MB and so on.

Flash drive

Larger flash memory modules have been developed to replace, hard disk drives. These flash drives are designed to fully emulate the bard disks, to the point that they can be fitted into standard disk drive bays. The fact that flash drives are solid state electronic devices that have no movable parts provides some important advantages. They have shorter seek and access times, which results in faster response. They have lower power consumption, which makes them attractive for battery driven applications, and they are also insensitive to vibration. Another type of ROM chip allows the stored data to be erased and new data to be loaded. The disadvantages of flash drives are less storage capacity, higher cost per bit and it will become weak after it has been written several times.

11.Speed, Size and Cost of Memories: An ideal memory would be fast, large, and inexpensive. a very fast memory can be implemented if SRAM chips are used. But these chips are expensive because their basic cells have six transistors, which preclude packing a very large number of cells onto a single chip. Thus, for cost reasons, it is impractical to build a large memory using SRAM chips. The alternative is to use Dynamic RAM chips, which have much simpler basic cells and thus are much less expensive. But such memories are significantly slower.

Although dynamic memory units in the range of hundreds of megabytes can be implemented at a reasonable cost, the affordable size is still small compared to the demands of large programs with voluminous data. A solution is provided by using secondary storage, mainly magnetic disks, to implement large memory spaces. Very large disks are available at a reasonable price, and they are used

extensively in computer systems. However, they are much slower than the semiconductor memory units. So A huge amount of cost-effective storage can be provided by magnetic disks. A large, yet affordable, main memory can be built with dynamic RAM technology. This leaves SRAMs to be used in smaller units where speed is of the essence, such as in cache memories.

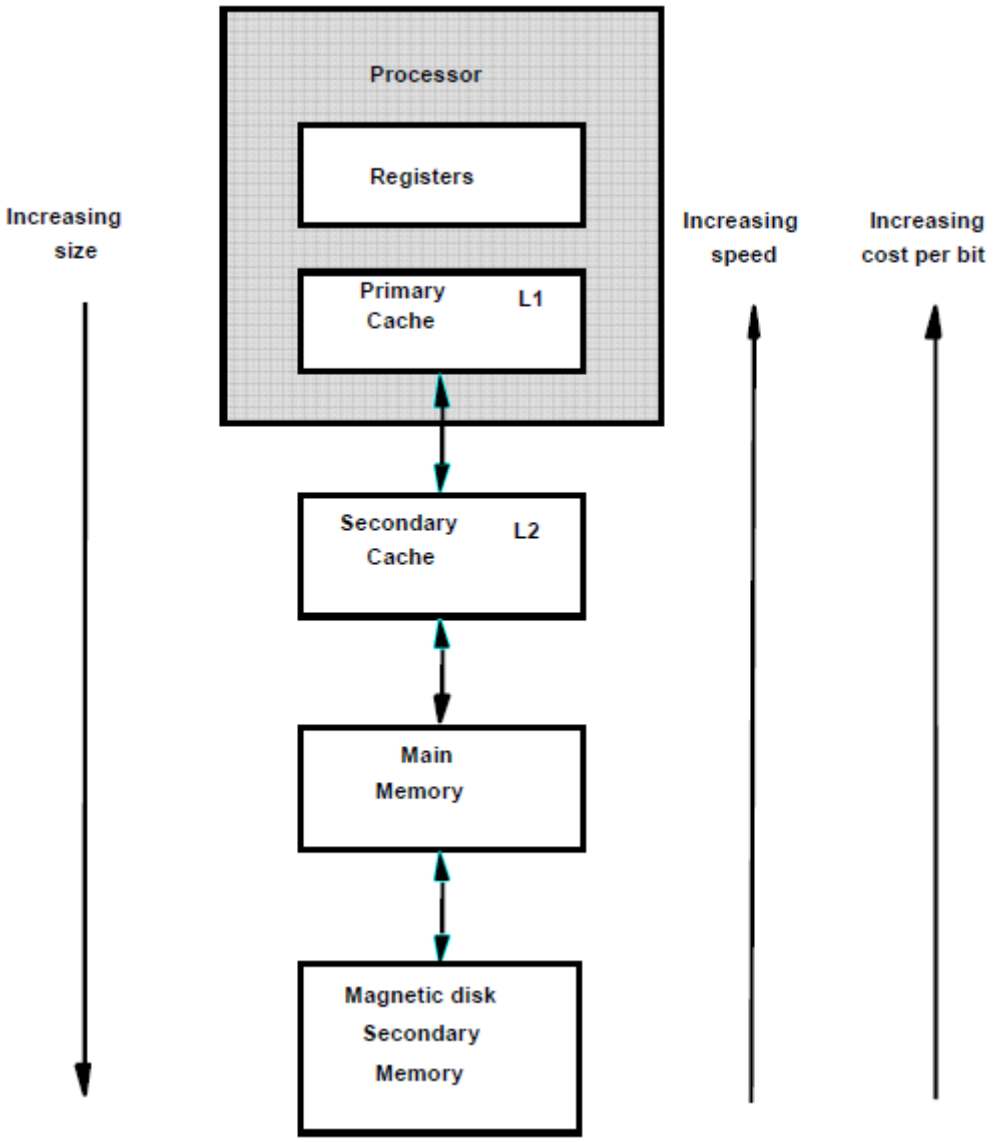


Figure 4.13 Memory hierarchy

All of these different types of memory units are employed effectively in a computer. The entire computer memory can be viewed as the hierarchy depicted in Figure 4.13. The fastest access is to data held in processor registers. Therefore, if we consider the registers to be part of the memory hierarchy, then the processor registers are at the top in terms of the speed of access. Of course, the registers provide only a minuscule portion of the required memory.

At the next level of the hierarchy is a relatively small amount of memory that can be implemented directly on the processor chip. This memory, called a processor cache, holds copies of instructions and data stored in a much larger memory that is provided externally. There are often two levels of caches. A primary cache is always located on the processor chip. This cache is small because it competes for space on the processor chip, which must implement many other functions.

12.MAPPING FUNCTIONS: The process of keeping information of moved data blocks from main memory to cache memory is known as mapping. For example: Consider a cache consisting of 128

blocks of 16 words each, for a total of 2048 (2K) words, and assumes that the main memory is addressable by a 16-bit address. The main memory has 64K words, which we will view as 4K (4096) blocks of 16 words each. These mappings can be done in three ways. They are Direct mapping, Associative mapping and Set associative mapping.

Direct mapping: The simplest way to determine cache locations in which to store memory blocks is the direct-mapping technique. In this technique, block j of the main memory maps onto block j modulo 128 of the cache, as depicted in Figure 4.15. Thus, whenever one of the main memory blocks 0, 128, 256,… is loaded in the cache, it is stored in cache block 0. Blocks 1, 129, 257,… are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full. For example, instructions of a program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed, both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block. In this case, the replacement algorithm is trivial.

Placement of a block in the cache is determined from the memory address. The memory address can be divided into three fields, as shown in Figure 4.15. The low-order 4 bits select one of 16 words in a block. When a new block enters the cache, the 7 -bit cache block field determines the cache position in which this block must be stored. The high-order 5 bits of the memory address of the block are stored in 5 tag bits associated with its location in the cache. They identify which of the 32 blocks that are mapped into this cache position are currently resident in the cache. As execution proceeds, the 7 -bit cache block field of each address generated by the processor points to a particular block location in the cache. The high-order 5 bits of the address are compared with the tag bits associated with that cache location. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache. The direct-mapping technique is easy to implement, but it is not very flexible.

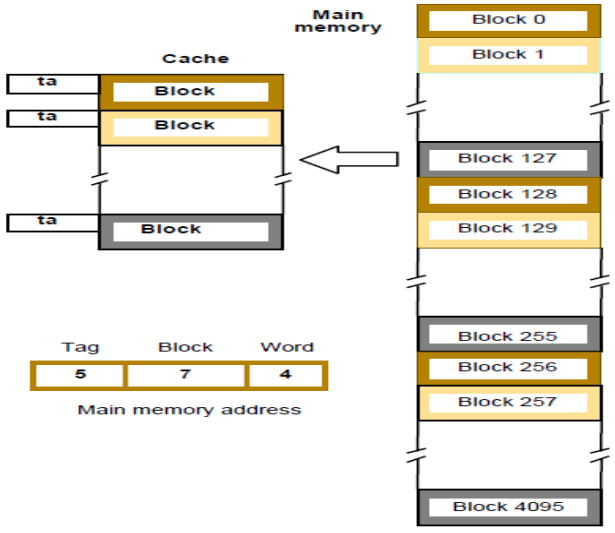


Figure 4.15 Direct-mapped cache.

Associative mapping: Figure 4.16 shows a much more flexible mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the associative-mapping technique. It gives complete freedom in choosing the cache location in which to place the memory block. Thus, the space in the cache can be used more

efficiently. A new block that has to be brought into the cache has to replace (eject) an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced. Many replacement algorithms are possible; the cost of an associative cache is higher than the cost of a direct-mapped cache because of the need to search all 128 tag patterns to determine whether a given block is in the cache. A search of this kind is called an associative search. For performance reasons, the tags must be searched in parallel.

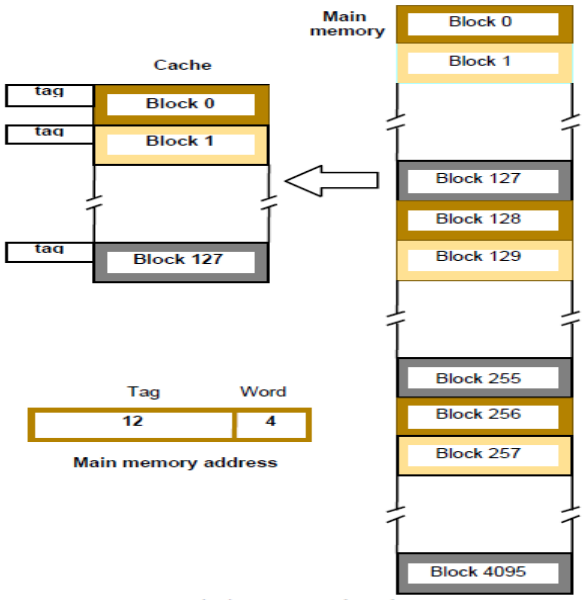


Figure 4.16 Associative-mapped cache.

Set-Associative mapping

A combination of the direct- and associative mapping techniques can be used. Blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement. At the same time, the hardware cost is reduced by decreasing the size of the associative search. An example of this set-associative-mapping technique is shown in Figure 4.17 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, …, 4032 map into cache set 0, and they can occupy either of the two block positions within this set. Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

The number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer. For the main memory and cache sizes in Figure 4.17, four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping method. A cache that has k blocks per set is referred to as a k-way set-associative cache.

12.improving cache performance: Two key factors in the commercial success of a computer are performance and cost; the best possible performance at the lowest cost is the objective. The challenge in considering design alternatives is to improve the performance without increasing the cost. A common measure of success is the price/performance ratio. Performance depends on how fast machine instructions can be brought into the processor for execution and how fast they can be executed. The memory hierarchy is used for the best price/performance ratio. The main purpose of this hierarchy is to create a memory I that the processor sees as having a short access time and a large capacity. Each

level of the hierarchy plays an important role. The speed and efficiency of data transfer between various levels of the hierarchy are also of great significance. It is beneficial if transfers to and from the faster units can done at a rate equal to that of the faster unit. This is not possible if both the slow and the fast units are accessed in the same manner, but it can be achieved when parallelism is used in the organization of the slower unit. An effective way to introduce parallelism is to use an interleaved organization.

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

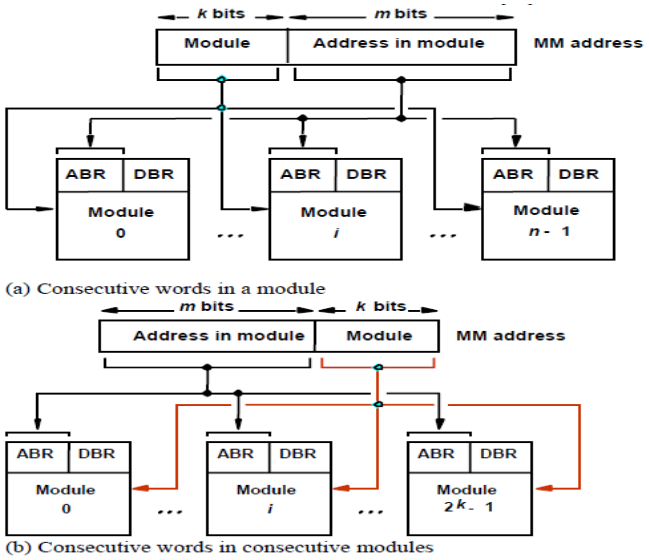


Figure 4.18 Addressing multiple-module memory systems.

How individual addresses are distributed over the modules is critical in determining the average number of modules that can be kept busy as computations proceed. Two methods of address layout are indicated in Figure 4.18. In the first case, the memory address generated by the processor is decoded as shown in Figure 4.18a. 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, as happens when a block of data is transferred to a cache, only one module is involved. At the same time, however, devices with direct memory access (DMA) ability may be accessing information in other memory modules.

The second and more effective way to address the modules is shown in Figure 4.18b. It 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 that module. In this way, consecutive addresses are located in successive modules. Thus, any component of the system that generates requests for access to consecutive memory locations can keep several modules busy at anyone time. This results in both faster accesses to a block of data and higher average utilization of the memory system as a whole. To implement the interleaved structure, there must be 2k modules; otherwise, there will be gaps of nonexistent locations in the memory address space.

13.Virtual memory: Te 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. Of course, 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. The operating system moves programs and data automatically between the main memory and secondary storage. This process is known as swapping. Thus, the application programmer does not need to be aware of limitations imposed by the available main memory.

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

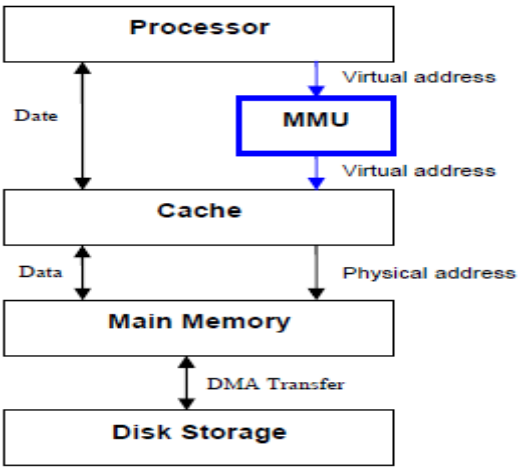


Figure 4.19 Virtual memory organization.
Figure 4.19 shows a typical organization that implements virtual memory. A special hardware unit, called the Memory Management Unit (MMU), translates virtual addresses into physical addresses. When the desired data (or instructions) are in the main memory, these data are fetched as described in our presentation of the ache mechanism. If the data are not in the main memory, the MMU causes the operating system to bring the data into the memory from the disk. The DMA scheme is used to perform the data Transfer between the disk and the main memory.

13.ADDRESS TRANSLATION: The process of translating a virtual address into physical address is known as address translation. It can be done with the help of MMU. A simple method for translating virtual addresses into physical addresses is to assume that all programs and data are composed of fixed-length units called pages, each of which consists of a block of words that occupy contiguous locations in the main memory. Pages commonly range from 2K to 16K bytes in length. They constitute the basic unit of information that is moved between the main memory and the disk whenever the translation mechanism determines that a move is required. Pages should not be too small, because the access time of a magnetic disk is much longer (several milliseconds) than the access time of the main memory. The reason for this is that it takes a considerable amount of time to locate the data on the disk, but once located, the data can be transferred at a rate of several megabytes per second. On the other hand, if pages are too large it is possible that a substantial portion of a page may not be used, yet this unnecessary data will occupy valuable space in the main memory.

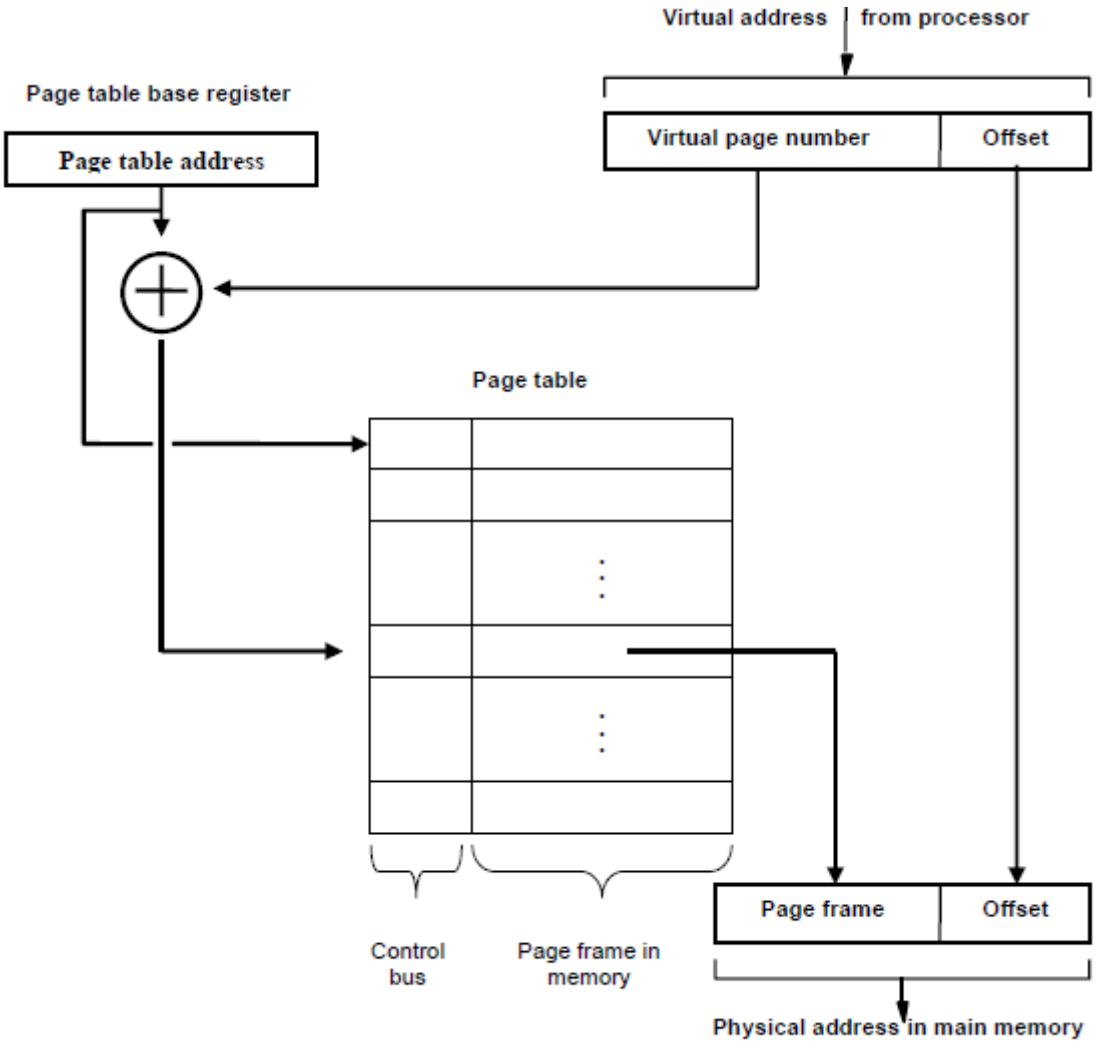


Figure 4.20 Virtual memory address translation.

The cache bridges the speed gap between the processor and the main memory and is implemented in hardware. The virtual-memory mechanism bridges the size and speed gaps between the main memory and secondary storage and is usually implemented in part by software techniques. Conceptually, cache techniques and virtual- memory techniques are very similar. They differ mainly in the details of their implementation.

A virtual-memory address translation method based on the concept of fixed-length pages is shown schematically in Figure 4.20. Each virtual address generated by the processor, whether it is for an instruction fetch or an operand fetch/store operation, is interpreted as a virtual page number (high-order bits) followed by an offset (low-order bits) that specifies the location of a particular byte (or word) within a page. Information about the main memory location of each page is kept in a page table. This information includes the main memory address where the page is stored and the current status of the page. An area in the main memory that can hold one page is called a page frame. The starting address of the page table is kept in a page table base register. By adding the virtual page number to the contents of this register, the address of the corresponding entry in the page table is obtained. The contents of this location give the starting address of the page if that page currently resides in the main memory.

14. Memory management requirements: Virtual memory fills the gap between the physical memory and secondary memory (disc), when one large program is being executed and it does not fit into the

available physical memory, parts of it (pages) are moved from the disk into the main memory when they are to be executed. Some software routines are needed to manage this movement of program segments. Management routines are part of the operating system of the computer. It is convenient to assemble the operating system routines into a virtual address space, called the system space, that is separate from the virtual space in which user application programs reside. The latter space is called the user space.

In fact, there may be a number of user spaces, one for each user. This is arranged by providing a separate page table for each user program. The MMU uses a page table base register to determine the address of the table to be used in the translation process. Hence, by changing the contents of this register, the operating system can switch from one space to another. The physical main memory is thus shared by the active pages of the system space and several user spaces. However, only the pages that belong to one of these spaces are accessible at any given time.

In any computer system in which independent user programs coexist in the main memory, the notion of protection must be addressed. No program should be allowed to destroy either the data or instructions of other programs in the memory. Such protection can be provided in several ways. Let us first consider the most basic form of protection. Recall that in the simplest case, the processor has two states, the supervisor state and the user state. As the names suggest, the processor is usually placed in the supervisor state when operating system routines are being executed and in the user state to execute user programs.

15 Secondary memory: Most computer systems are economically realized in the form of magnetic disks, optical disks, and magnetic tapes, which are usually referred to as secondary storage devices. Secondary storage memories are Non volatile, Low cost per bit and mass storage of information.

MAGNETIC HARD DISKS: The storage medium in a magnetic-disk system consists of one or more disks mounted on a common spindle. A thin magnetic film is deposited on each disk, usually on both sides. The disks are placed in a rotary drive so that the magnetized surfaces move in close proximity to read/write heads, as shown in Figure 4.26a. The disks rotate at a uniform speed. Each head consists of a magnetic yoke and a magnetizing coil, as indicated in Figure 4.26b.

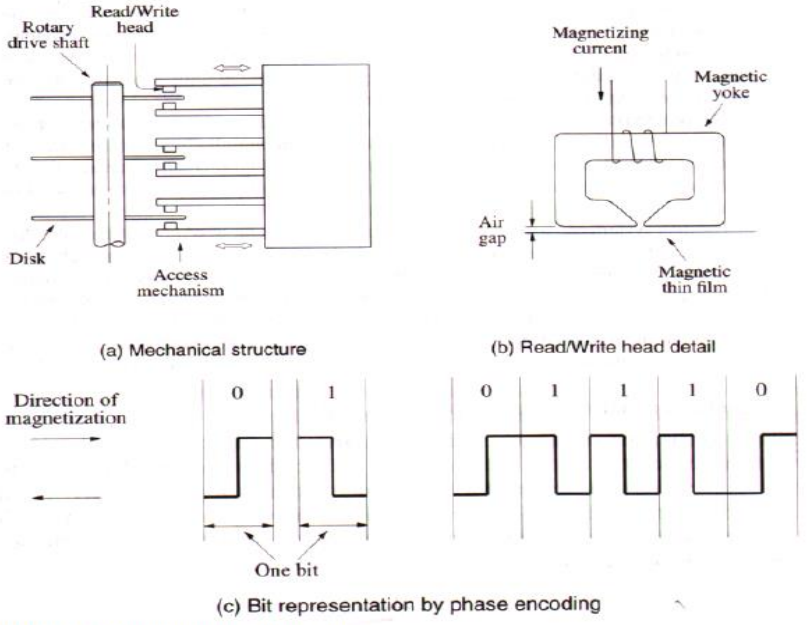


Figure 4.26 Magnetic disk principles.

Digital information can be stored on the magnetic film by applying current pulses of suitable polarity to the magnetizing coil. This causes the magnetization of the film in the area immediately underneath the head to switch to a direction parallel to the applied field. The same head can be used for reading the stored information. In this case, changes in the magnetic field in the vicinity of the head caused by the movement of the film relative to the yoke induce a voltage in the coil, which now serves as a sense coil. The polarity of this voltage is monitored by the control circuitry to determine the state of magnetization of the film. Only changes in the magnetic field under the head can be sensed during the Read operation. Therefore, if the binary states 0 and 1 are represented by two opposite states of magnetization, a voltage is induced in the head only at 0-to-l and at I-to-O transitions in the bit stream. A long string of 0s or 1 s causes an induced voltage only at the beginning and end of the string. To determine the number of consecutive 0s or 1s stored, a clock must provide information for synchronization. In some early designs, a clock was stored on a separate track, where a change in magnetization is forced for each bit period. Using the clock signal as a reference, the data stored on other tracks can be read correctly.

The modem approach is to combine the clocking information with the data. Several different techniques have been developed for such encoding. One simple scheme, depicted in Figure 4.26c, is known as phase encoding or Manchester encoding. In this scheme, changes in magnetization occur for each data bit, as shown in the figure. Note that a change in magnetization is guaranteed at the midpoint of each bit period, thus providing the clocking information. The drawback of Manchester encoding is its poor bit-storage density. The space required to represent each bit must be large enough to accommodate two changes in magnetization. We use the Manchester encoding example to illustrate how a self-clocking scheme may be implemented, because it is easy to understand. Other, more compact codes have been developed. They are much more efficient and provide better storage density. They also require more complex control circuitry.

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

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

The main feature of floppy disks is their low cost and shipping convenience. However, they have much smaller storage capacities, longer access times, and higher failure rates than hard disks. Current standard floppy disks are 3.25 inches in diameter and store 1.44 or 2 Mbytes of data. Larger super-floppy disks are also available. One type of such disks, known as the zip disk, can store more than 100 Mbytes. In recent years, the attraction of floppy-disk technology has been diminished by the emergence of rewritable compact disks.

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

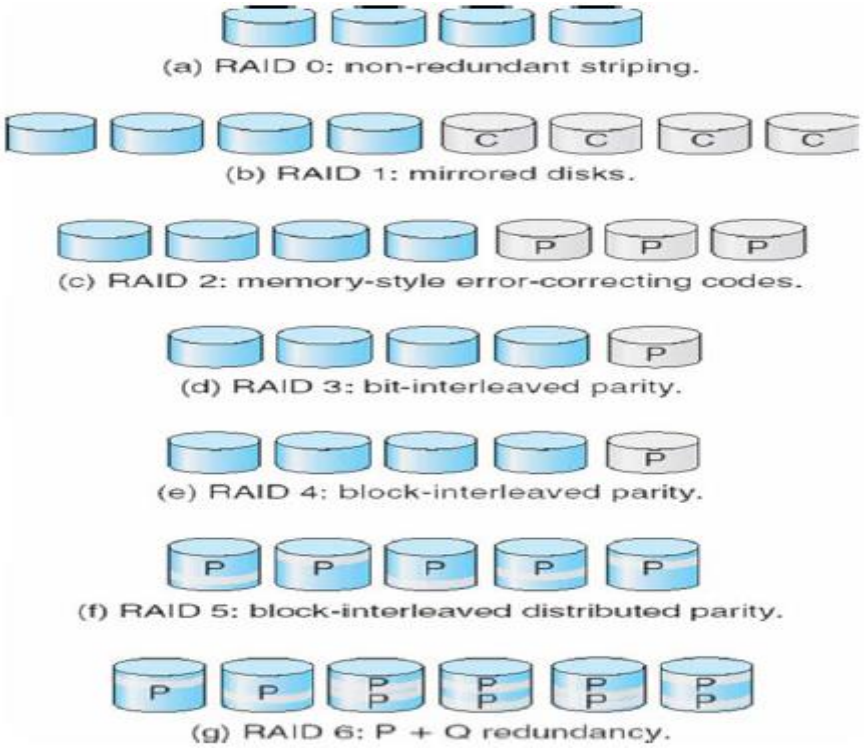


Figure 4.29 RAID Levels

17. OPTICAL DISKS: Large storage devices can also be implemented using optical means. The familiar compact disk (CD), used in audio systems, was the first practical application of this technology. Soon after, the optical technology was adapted to the computer environment to provide high-capacity readonly storage referred to as CD-ROM. The first generation of CDs was developed in the mid-1980s by the Sony and Philips companies, which also published a complete specification for these devices.

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

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

A cross-section of a small portion of a CD is shown in Figure 4.30a. The bottom layer is polycarbonate plastic, which functions as a clear glass base. The surface of this plastic is programmed to store data by indenting it with pits. The unintended parts are called lands. A thin layer of reflecting aluminum material is placed on top of a programmed disk. The aluminum is then covered by a protective acrylic. Finally, the topmost layer is deposited and stamped with a label. The total thickness of the disk is 1.2mm. Almost all of it is contributed by the polycarbonate plastic. The other layers are very thin. The laser source and the photodetector are positioned below the polycarbonate plastic. The emitted beam

travels through this plastic, reflects off the aluminum layer, and travels back toward the photo detector. Note that from the laser side, the pits actually appear as bumps with respect to the lands.

18. OTHER STORAGE DEVICES

USB flash drive: A USB flash drive is a data storage device that consists of flash memory with an integrated Universal Serial Bus (USB) interface. USB flash drives are typically removable and rewritable, and physically much smaller than a floppy disk. Most weigh less than 30 grams. As of January 2012 drives of 1 terabytes (TB) are available and storage capacities as large as 2 terabytes are planned, with steady improvements in size and price per capacity expected. Some allow up to 100,000 write/erase cycles (depending on the exact type of memory chip used) and 10 years shelf storage time.

Zip drive: The Zip drive is a medium-capacity removable disk storage system that was introduced by Iomega in late 1994. Originally, Zip disks launched with capacities of 100 MB, but later versions increased this to first 250 MB and then 750 MB.

Memory card: A memory card or flash card is an electronic flash memory data storage device used for storing digital information. They are commonly used in many electronic devices, including digital cameras, mobile phones, laptop computers, MP3 players, and video game consoles. They are small, re-recordable, and able to retain data without power.

20. AUXILIARY STORAGE DEVICES: Auxiliary storage also known as auxiliary memory or secondary storage, is the memory that supplements the main storage. This is a long-term, non-volatile memory. The term non-volatile means that stores and retains the programs and data even after the computer is switched off. Unlike RAM which loses the contents when the computer is turned off, and ROM, to which it is not possible to add anything new, auxiliary storage devices allow the computer to record information semi- permanently, so it can be read later by the same computer or by another computer. Auxiliary storage devices are also useful in transferring data or programs from one computer to another.

They also function as back-up devices which allow to back-up the valuable information. So even if by some accident the computer crashes and the stored data is unrecoverable, we can restore it from the back-ups. The most common types of auxiliary storage devices are magnetic tapes, magnetic disks, floppy disks, hard disks, etc.There are two types of auxiliary storage devices.

T**his classification is based on the type of data access:**

1. sequential
2. random.

Based on the type of access, they are called sequential-access media or random-media. In the case of sequential-access media, the data stored in the media can only be read in sequence and to get to a particular point on the media, we have to go through all the preceding points. Magnetic tapes are examples of sequential-access media. In contrast, disks are random-access also called direct-access media because a disk drive can access any point at random without passing through intervening points. Other examples of direct access media are floppy diskettes, optical disks, zip disks, etc.

MAGNETIC TAPE: Magnetic tape is a magnetically coated strip of plastic on which data can be encoded. Tapes for computers are similar to the tapes used to store music. Some computers, in fact, enable us to use normal cassette tapes. Storing data on tapes is considerably cheaper than storing data on disks. Tapes also have large storage capacities, ranging from a few hundred kilobytes to several gigabytes. Accessing data on tapes, however, is much slower than accessing data on disks. Because

tapes are so slow, they are generally used only for long-term storage and backup. Data to be used regularly is almost always kept on a disk. Tapes are also used for transporting large amounts of data. Tapes come in a variety of sizes and formats as given in Table 4.2.1. Tapes are sometimes called streamers or streaming tapes.

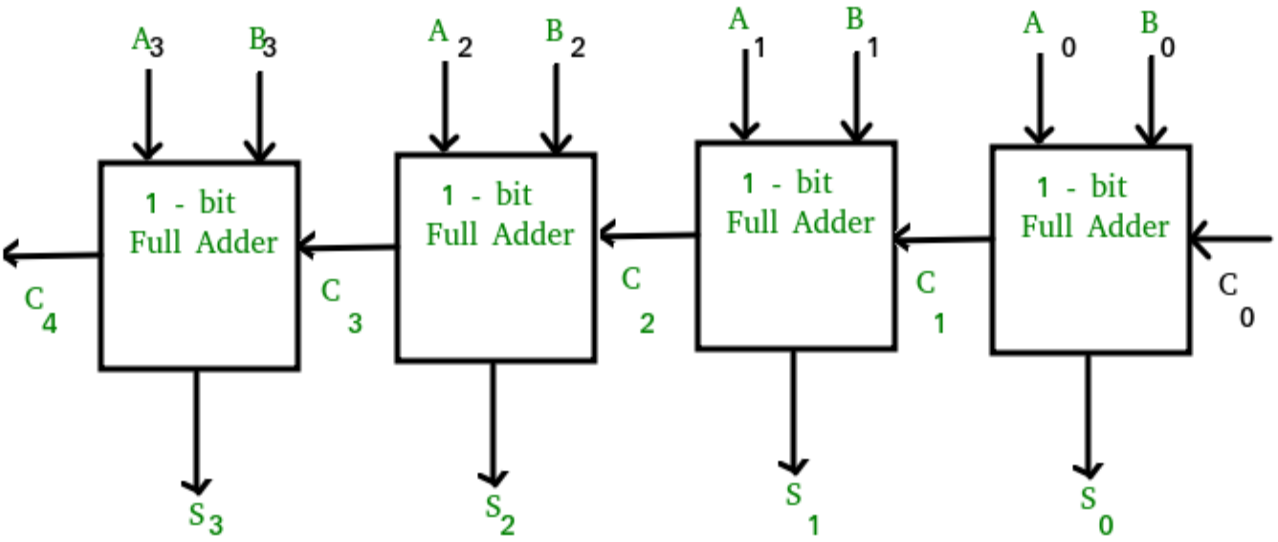
Type	Capacity	Description
Half Inch	60MB - 400MB	Half-inch tapes come both as 9 track reels and as cartridges. These tapes are relatively cheap, but require expensive tape drives
Quarter Inch	40MB - 5GB	**Quarter Inch Cartridges** (QIC tapes) are relatively inexpensive and support fast data transfer rates, QIC mini cartridges are even less expensive, but their data capacities are smaller and their transfer rates are lower.
8-mm Helical scan	1 GB - 5 GB	8 min helical-scan cartridges use the same technology as VCR tapes and have the great capacity. But they require expensive tape drives and have relatively slow data transfer rates.
4-mm DAT	2GB - 24GB	**DAT (Digital Audio Tape)** cartridges have the greatest capacity but they require expensive tape drives and have relatively slow data transfer Rates.

Helical-scan Cartridge: It's a type of magnetic tape that uses the same technology as VCR tapes. The term helical scan usually refers to 8-mm tapes, although 4-mm tapes (called DAT tapes) use the same technology. The 8-mm helical-scan tapes have data capacities from 2.5GB to 5 GB.

Fig 4.2.1: DAT Cartridge

DAT Cartridge: This is a type of magnetic tape that uses an ingenious scheme called helical scan to record data, as shown in Fig. 4.2.1. A DAT cartridge is slightly larger than a credit card and contains a magnetic tape that can hold from 2 to 24 gigabytes of data. It can support data transfer rates of about 2 MBPS (Million Bytes Per Second). Like other types of tapes, DATs are sequential-access media. The most common format for DAT cartridges is DDS (Digital Data Storage) which is the industry standard for digital audio tape (DAT) formats. The latest format, DDS-3, specifies tapes that can hold 24 GB (the equivalent of over 40 CD ROMs) and support data transfer rates of 2 MBPS.

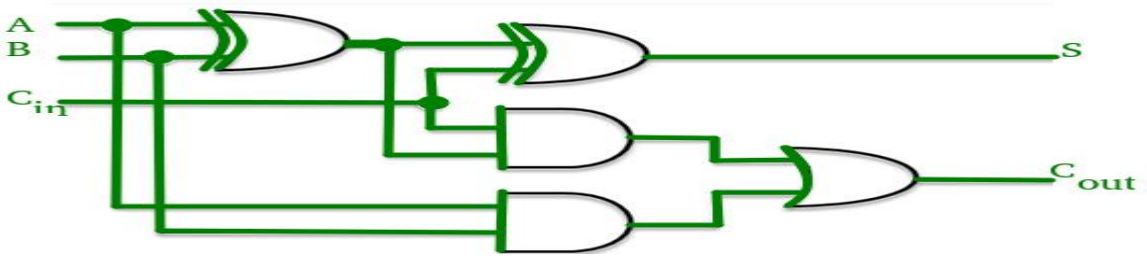
Carry Look-Ahead Adder :
In ripple carry adders, for each adder block, the two bits that are to be added are available instantly. However, each adder block waits for the carry to arrive from its previous block. So, it is not possible to generate the sum and carry of any block until the input carry is known. The block waits for the block to produce its carry. So there will be a considerable time delay which is carry propagation delay.



Consider the above 4-bit ripple carry adder. The sum is produced by the corresponding full adder as soon as the input signals are applied to it. But the carry input is not available on its final steady state value until carry is available at its steady state value. Similarly depends on and on . Therefore, though the carry must propagate to all the stages in order that output and carry settle their final steady-state value.

The propagation time is equal to the propagation delay of each adder block, multiplied by the number of adder blocks in the circuit. For example, if each full adder stage has a propagation delay of 20 nanoseconds, then will reach its final correct value after 60 (20 × 3) nanoseconds. The situation gets worse, if we extend the number of stages for adding more number of bits.

Carry Look-ahead Adder :
A carry look-ahead adder reduces the propagation delay by introducing more complex hardware. In this design, the ripple carry design is suitably transformed such that the carry logic over fixed groups of bits of the adder is reduced to two-level logic. Let us discuss the design in detail.



A	B	C	C +1	Condition
0	0	0	0	
0	0	1	0	No Carry
0	1	0	0	Generate
0	1	1	1	
1	0	0	0	No Carry
1	0	1	1	Propogate
1	1	0	1	Carry
1	1	1	1	Generate

Consider the full adder circuit shown above with corresponding truth table. We define two variables as **'carry generate'** and **'carry propagate'** then,

Multiplication Algorithm in Signed Magnitude Representation: Multiplication of two fixed point binary number in *signed magnitude representation* is done with process of *successive shift* and *add operation*.

```
  10111 (Multiplicand)
x 10011 (Multiplier)
  10111
  10111
 00000
 00000
10111
011011010   (Product)
```

In the multiplication process we are considering successive bits of the multiplier, least significant bit first.

If the multiplier bit is 1, the multiplicand is copied down else 0's are copied down.

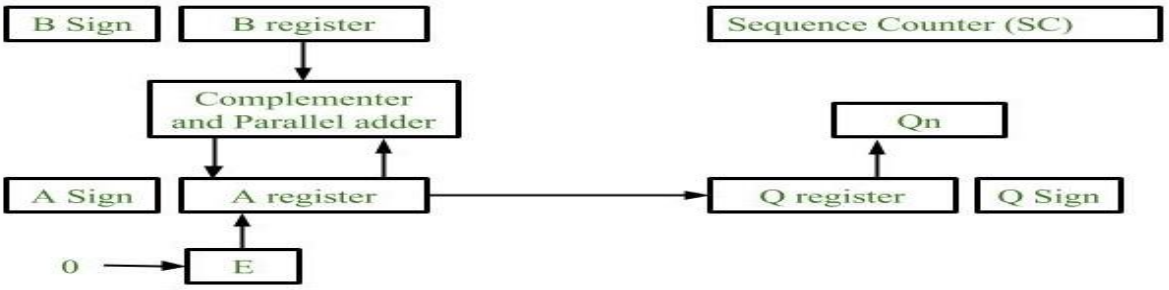
The numbers copied down in successive lines are shifted one position to the left from the previous number.

Finally numbers are added and their sum form the product.

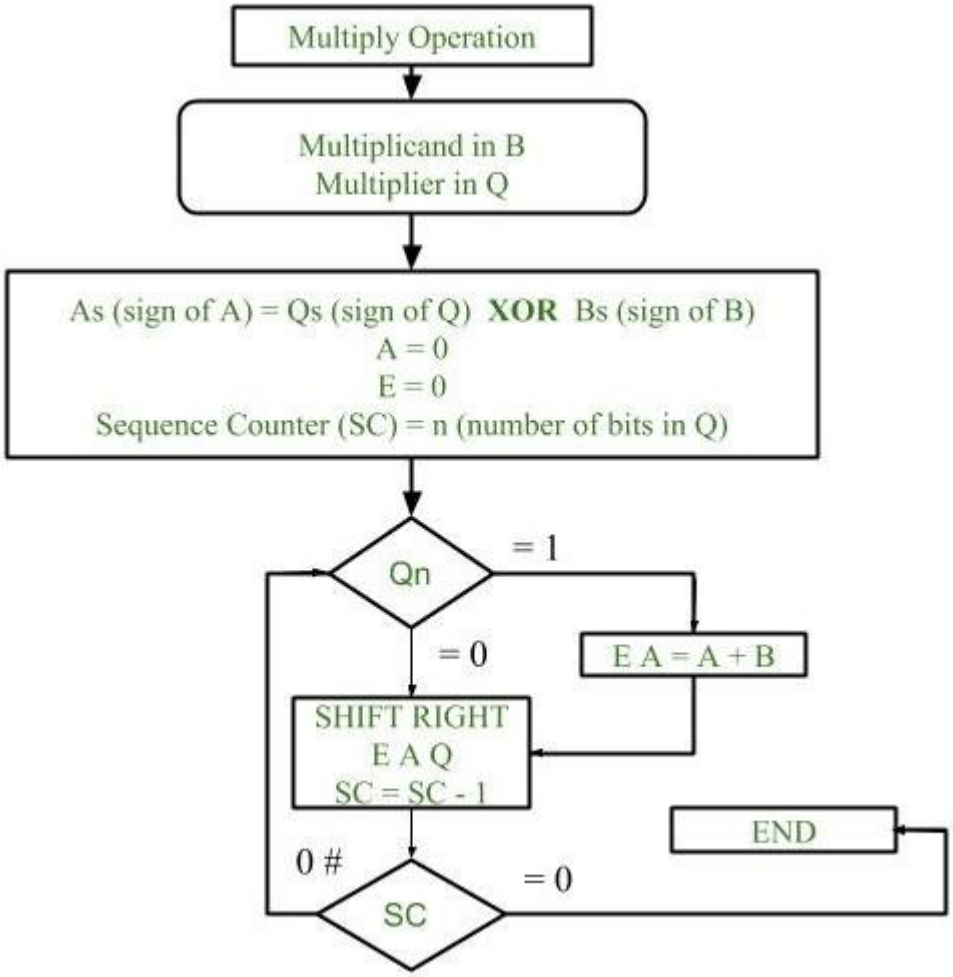
The sign of the product is determined from the sign of the multiplicand and multiplier. If they are alike, sign of the product is positive else negative.

Hardware Implementation :

Following components are required for the *Hardware Implementation* of multiplication algorithm :



Flowchart of Multiplication:



1. Initially multiplicand is stored in B register and multiplier is stored in Q register.
2. Sign of registers B (Bs) and Q (Qs) are compared using **XOR** functionality (i.e., if both the signs are alike, output of XOR operation is 0 unless 1) and output stored in As (sign of A register). **Note:** Initially 0 is assigned to register A and E flip flop. Sequence counter is initialized with value n, n is the number of bits in the Multiplier.
3. Now least significant bit of multiplier is checked. If it is 1 add the content of register A with Multiplicand (register B) and result is assigned in A register with carry bit in flip flop E. Content of E A Q is shifted to right by one position, i.e., content of E is shifted to most significant bit (MSB) of A and least significant bit of A is shifted to most significant bit of Q.

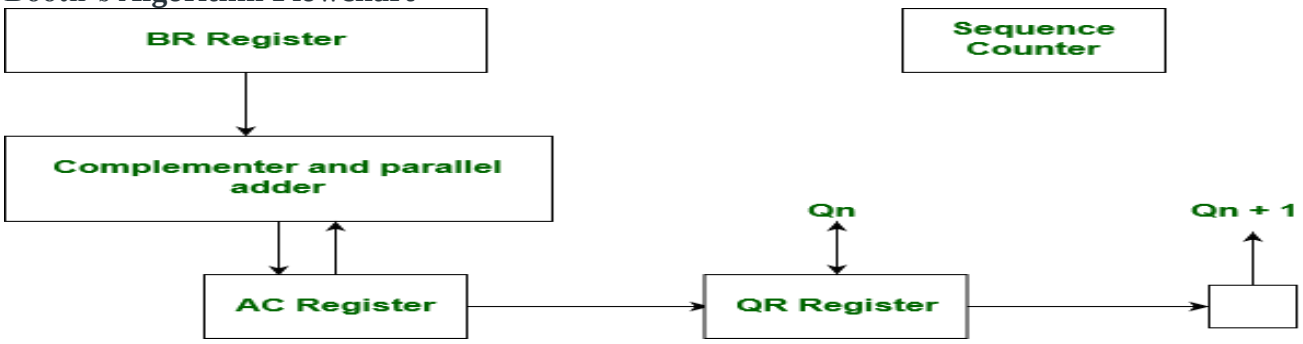
4. If Qn = 0, only shift right operation on content of E A Q is performed in a similar fashion.
5. Content of Sequence counter is decremented by 1.
6. Check the content of Sequence counter (SC), if it is 0, end the process and the final product is present in register A and Q, else repeat the process.

Booth's Algorithm: Booth algorithm gives a procedure for **multiplying binary integers** in signed 2's complement representation **in efficient way**, i.e., less number of additions/subtractions required. It operates on the fact that strings of 0's in the multiplier require no addition but just shifting and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as 2^(k+1) to 2^m. As in all multiplication schemes, booth algorithm requires examination **of the multiplier bits** and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to following rules:

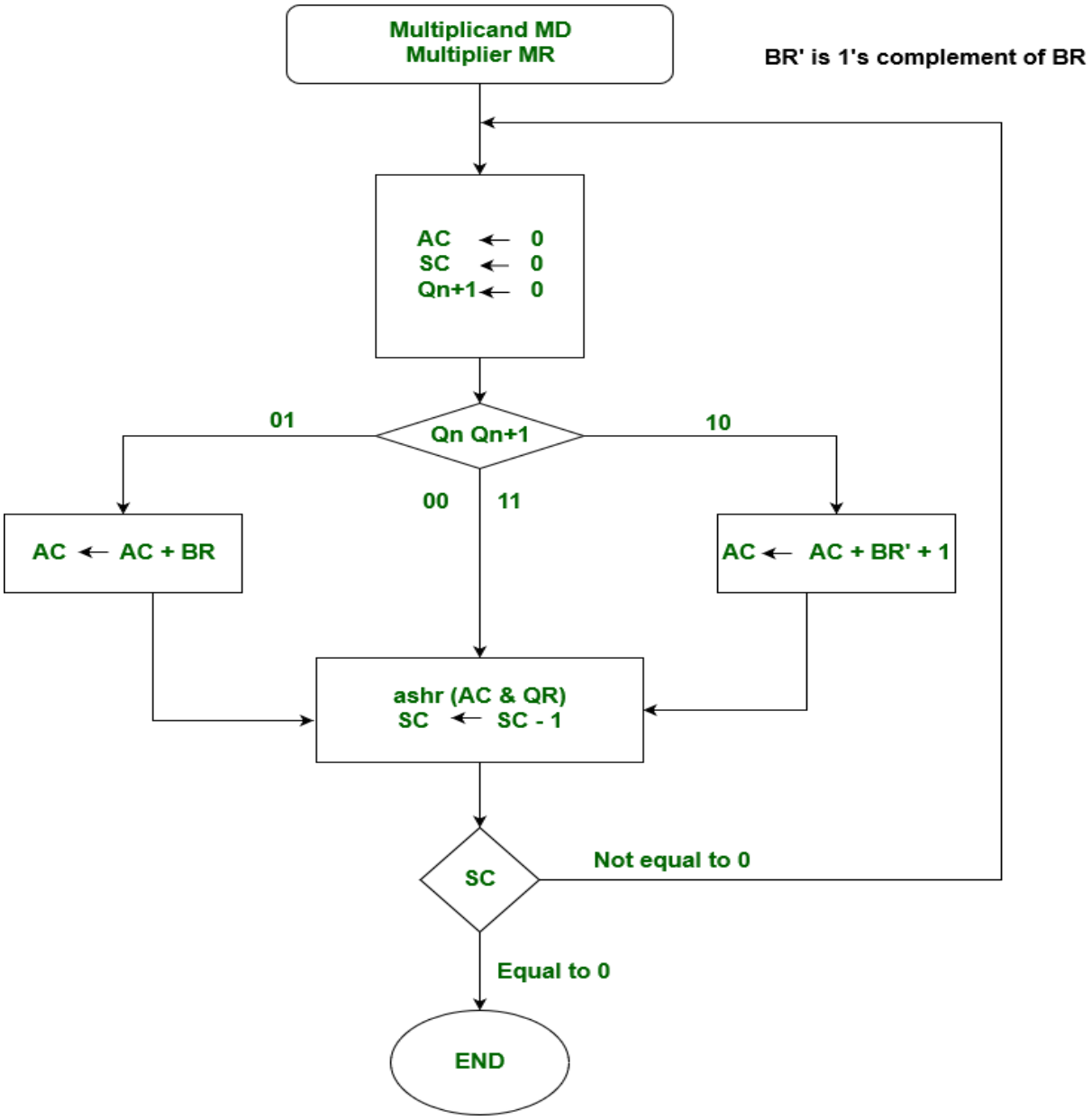
1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous '1') in a string of 0's in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

Hardware Implementation of Booths Algorithm – The hardware implementation of the booth algorithm requires the register configuration shown in the figure below.

Booth's Algorithm Flowchart –



We name the register as A, B and Q, AC, BR and QR respectively. Qn designates the least significant bit of multiplier in the register QR. An extra flip-flop Qn+1is appended to QR to facilitate a double inspection of the multiplier.The flowchart for the booth algorithm is shown below.

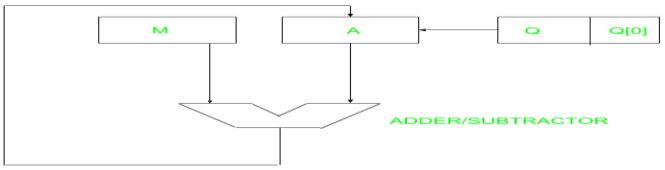


AC and the appended bit Qn+1 are initially cleared to 0 and the sequence SC is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Qn and Qn+1are inspected. If the two bits are equal to 10, it means that the first 1 in a string has been encountered. This requires subtraction of the multiplicand from the partial product in AC. If the 2 bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC.

When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. As a consequence, the 2 numbers that are added always have a opposite signs, a condition that excludes an overflow. The next step is to shift right the partial product and the multiplier (including Qn+1). This is an arithmetic shift right (ashr) operation which AC and QR ti the right and leaves the sign bit in AC unchanged. The sequence counter is decremented and the computational loop is repeated n times.

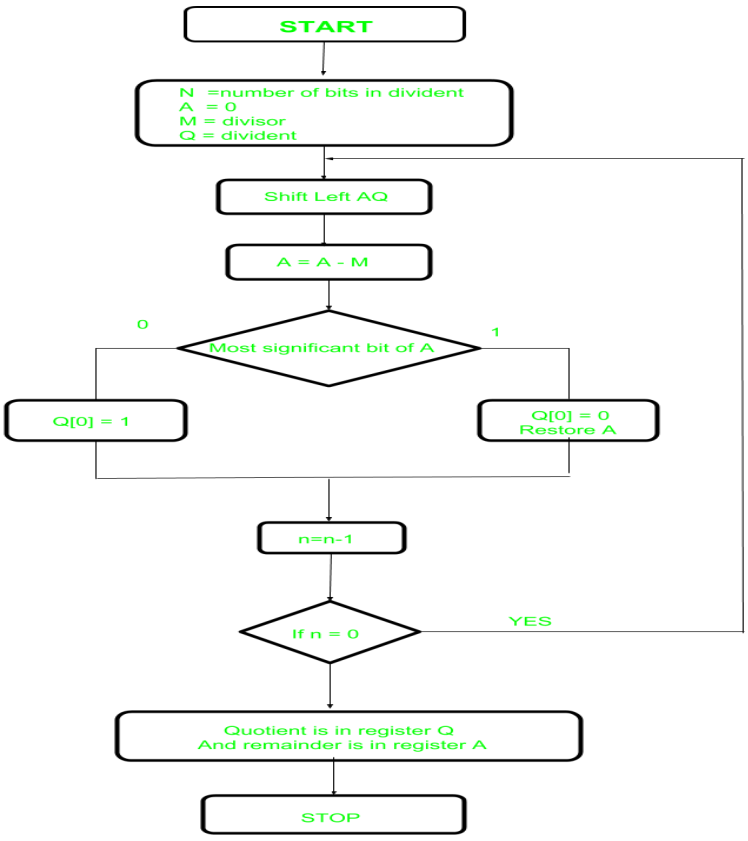
Restoring Division Algorithm For Unsigned Integer: A division algorithm provides a quotient and a remainder when we divide two number. They are generally of two type **slow algorithm and fast**

algorithm. Slow division algorithm are restoring, non-restoring, non-performing restoring, SRT algorithm and under fast comes Newton–Raphson and Goldschmidt.
In this article, will be performing restoring algorithm for unsigned integer. Restoring term is due to fact that value of register A is restored after each iteration.



Here, register Q contain quotient and register A contain remainder. Here, n-bit dividend is loaded in Q and divisor is loaded in M. Value of Register is initially kept 0 and this is the register whose value is restored during iteration due to which it is named Restoring.

Here, register Q contain quotient and register A contain remainder. Here, n-bit dividend is loaded in Q and divisor is loaded in M. Value of Register is initially kept 0 and this is the register whose value is restored during iteration due to which it is named Restoring.



Let’s pick the step involved:

- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Then the content of register A and Q is shifted left as if they are a single unit
- **Step-3:** Then content of register M is subtracted from A and result is stored in A
- **Step-4:** Then the most significant bit of the A is checked if it is 0 the least significant bit of Q is set to 1 otherwise if it is 1 the least significant bit of Q is set to 0 and value of register A is restored i.e the value of A before the subtraction with M

- **Step-5:** The value of counter n is decremented
- **Step-6:** If the value of n becomes zero we get of the loop otherwise we repeat from step 2
- **Step-7:** Finally, the register Q contain the quotient and A contain remainder

Examples:

Perform Division Restoring Algorithm

Dividend = 11

Divisor = 3

n	M	A	Q	Operation
4	00011	00000	1011	initialize
	00011	00001	011_	shift left AQ
	00011	11110	011_	A=A-M
	00011	00001	0110	Q[0]=0 And restore A
3	00011	00010	110_	shift left AQ
	00011	11111	110_	A=A-M
	00011	00010	1100	Q[0]=0
2	00011	00101	100_	shift left AQ
	00011	00010	100_	A=A-M
	00011	00010	1001	Q[0]=1
1	00011	00101	001_	shift left AQ
	00011	00010	001_	A=A-M
	00011	00010	0011	Q[0]=1

Floating Point Arithmetic Unit: When you have to represent very small or very large numbers, a fixed point representation will not do. The accuracy will be lost. Therefore, you will have to look at floating-point representations, where the binary point is assumed to be floating. When you consider a decimal number 12.34 * 107, this can also be treated as 0.1234 * 109, where 0.1234 is the fixed-point mantissa. The other part represents the exponent value, and indicates that the actual position of the binary point is 9 positions to the right (left) of the indicated binary point in the fraction. Since the binary point can be moved to any position and the exponent value adjusted appropriately, it is called a

floating-point representation. By convention, you generally go in for a normalized representation, wherein the floating-point is placed to the right of the first nonzero (significant) digit. The base need not be specified explicitly and the sign, the significant digits and the signed exponent constitute the representation.

The IEEE (Institute of Electrical and Electronics Engineers) has produced a standard for floating point arithmetic. This standard specifies how single precision (32 bit) and double precision (64 bit) floating point numbers are to be represented, as well as how arithmetic should be carried out on them. The IEEE single precision floating point standard representation requires a 32 bit word, which may be represented as numbered from 0 to 31, left to right. The first bit is the sign bit, S, the next eight bits are the exponent bits, 'E', and the final 23 bits are the fraction 'F'. Instead of the signed exponent E, the value stored is an unsigned integer E' = E + 127, called the excess-127 format. Therefore, E' is in the range 0 £ E' £ 255.

UNIT 5

BASIC PROCESSINGUNIT

SOME FUNDAMENTAL CONCEPTS

- To execute an instruction, processor has to perform following 3 steps:
  1) Fetch contents of memory-location pointed to by PC. Content of this location is an instruction to be executed. The instructions are loaded into IR, Symbolically, this operation can be written as
     IR□ [[PC]]
  2) Increment PC by 4
     PC□ [PC] +4
  3) Carry out the actions specified by instruction (in the IR).
- The first 2 steps are referred to as fetch phase;
  Step 3 is referred to as execution phase

SINGLE BUS ORGANIZATION

- MDR has 2 inputs and 2 outputs. Data may be loaded
  → into MDR either from memory-bus (external) or
  → from processor-bus (internal).
- MAR"s input is connected to internal-bus, and MAR"s output is connected to external-bus.
- Instruction-decoder & control-unit is responsible for
  → issuing the signals that control the operation of all the units inside the processor (and for interacting with memory bus).
  → implementing the actions specified by the instruction (loaded in the IR)
- Registers R0 through R(n-1) are provided for general purpose use by programmer.
- Three registers Y, Z & TEMP are used by processor for temporary storage during execution of some instructions. These are transparent to the programmer i.e. programmer need not be concerned with them because they are never referenced explicitly by any instruction.
- MUX(Multiplexer) selects either
  → output of Y or
  → constant-value 4(is used to increment PC content).This is provided as input A of ALU.
- B input of ALU is obtained directly from processor-bus.
- As instruction execution progresses, data are transferred from one register to another, often passing through ALU to perform arithmetic or logic operation.
- An instruction can be executed by performing one or more of the following operations:

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

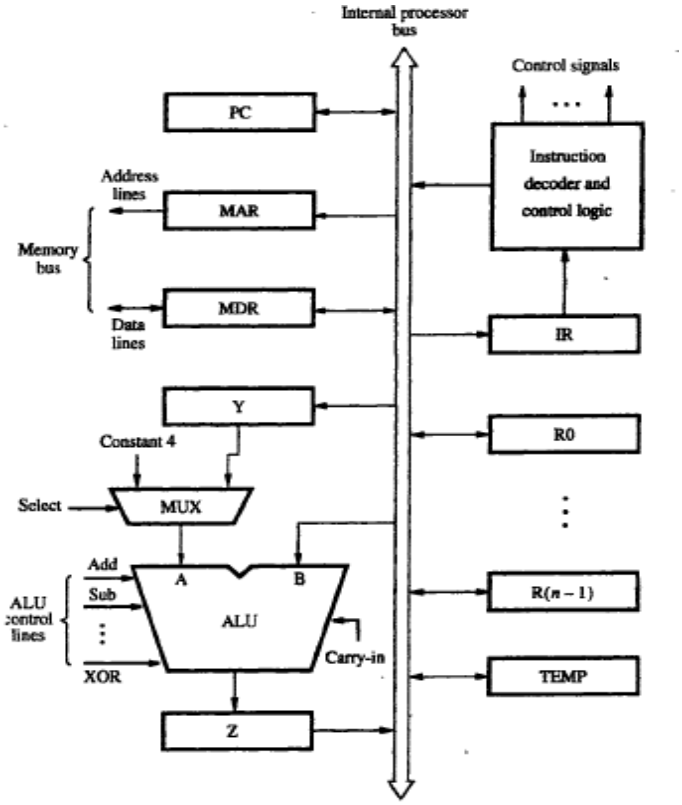


Figure 7.1 Single-bus organization of the datapath inside a processor.

REGISTER TRANSFERS

- Instruction execution involves a sequence of steps in which data are transferred from one register to another.
- Input & output of register Ri is connected to bus via switches controlled by 2 control-signals: Ri_{in} & Ri_{out}. These are called *gating signals.*
- When Ri_{in}=1, data on bus is loaded into Ri.
  Similarly, when Ri_{out}=1, content of Ri is placed on bus.
- When Ri_{out}=0, bus can be used for transferring data from other registers.
- All operations and data transfers within the processor take place within time-periods defined by the processor- clock.
- When edge-triggered flip-flops are not used, 2 or more clock-signals may be needed to guarantee proper transfer of data. This is known as *multiphase clocking*.

Input & Output Gating for one Register Bit

- A 2-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop.
- When Ri_{in}=1, mux selects data on bus. This data will be loaded into flip-flop at rising-edge of clock. When Ri_{in}=0, mux feeds back the value currently stored in flip-flop.
- Q output of flip-flop is connected to bus via a tri-state gate.
  When Ri_{out}=0, gate's output is in the high-impedance state. (This corresponds to the open- circuit state of a switch).
  When Ri_{out}=1, the gate drives the bus to 0 or 1, depending on the value of Q.

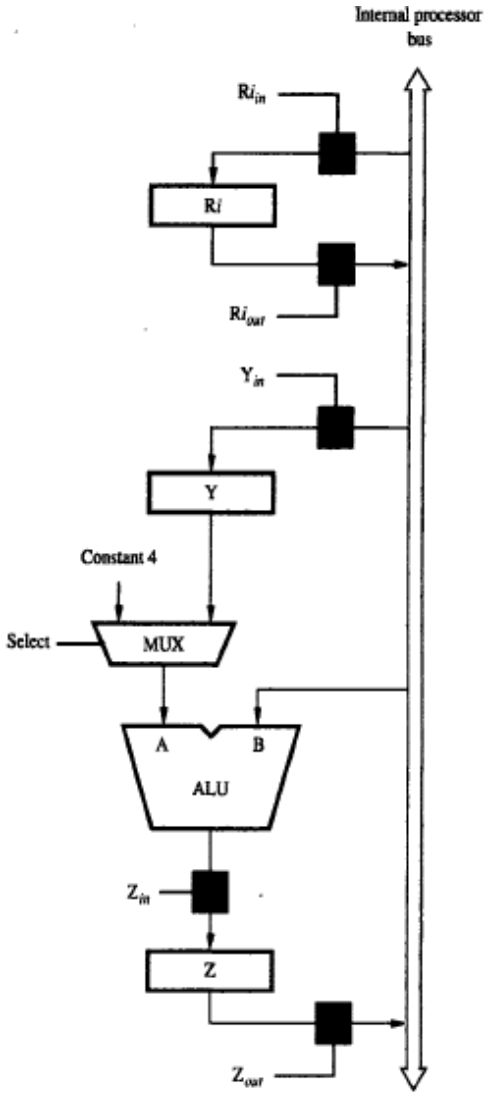


Figure 7.2 Input and output gating for the registers in Figure 7.1.

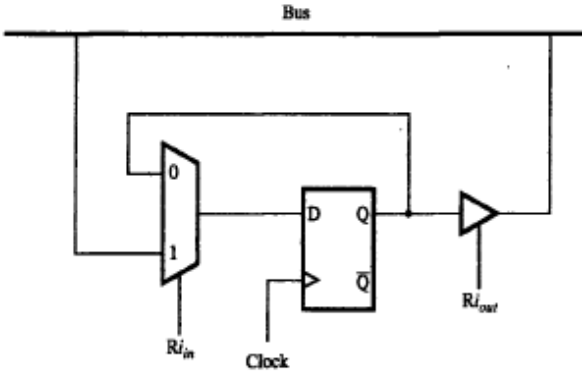


Figure 7.3 Input and output gating for one register bit.

PERFORMING AN ARITHMETIC OR LOGIC OPERATION

- The ALU performs arithmetic operations on the 2 operands applied to its A and B inputs.
- One of the operands is output of MUX & the other operand is obtained directly from bus.
- The result (produced by the ALU) is stored temporarily in register Z.
- The sequence of operations for [R3]□[R1]+[R2] is as follows
  1) R1out, Yin //transfer the contents of R1 to Y register
  2) R2out, SelectY, Add, Zin //R2 contents are transferred directly to B input of ALU. // The numbers of added. Sum stored in register Z
  3) Zout, R3in //sum is transferred to register R3
- The signals are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive.

Write the complete control sequence for the instruction : Move (R_S),R_d

- This instruction copies the contents of memory-location pointed to by R_S into R_d. This is a memory read operation. This requires the following actions

→ fetch the instruction
→ fetch the operand (i.e. the contents of the memory-location pointed by Rs).
→ transfer the data to Rd.

- The control-sequence is written as follows
  1) PCout, MARin, Read, Select4, Add, Zin
  2) Zout, PCin, Yin, WMFC
  3) MDRout, IRin
  4) Rs, MARin, Read
  5) MDRinE, WMFC
  6) MDRout, Rd, End

FETCHING A WORD FROM MEMORY

- To fetch instruction/data from memory, processor transfers required address to MAR (whose output is connected to address-lines of memory-bus).
  At the same time, processor issues Read signal on control-lines of memory-bus.
- When requested-data are received from memory, they are stored in MDR. From MDR, they are transferred to other registers
- MFC (Memory Function Completed): Addressed-device sets MFC to 1 to indicate that the contents of the specified location
  → have been read &
  → are available on data-lines of memory-bus
- Consider the instruction Move (R1),R2. The sequence of steps is:
  1) R1out, MARin, Read ;desired address is loaded into MAR & Read command is issued
  2) MDRinE, WMFC ;load MDR from memory bus & Wait for MFC response from memory
  3) MDRout, R2in ;load R2 from MDR

where WMFC=control signal that causes

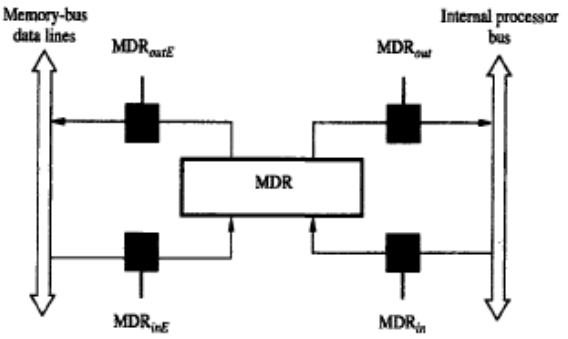


Figure 7.4 Connection and control signals for register MDR.

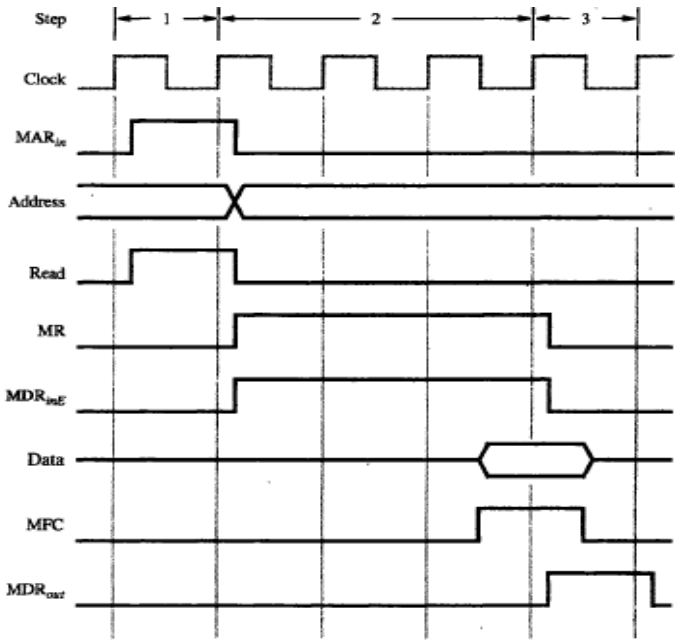


Figure 7.5 Timing of a memory Read operation.

processor's control circuitry to wait for arrival
of MFC signal

Storing a Word in Memory

- Consider the instruction *Move R2,(R1)*. This requires the following sequence:
  1) $R1_{out}$, MAR_{in} ;desired address is loaded into MAR
  2) $R2_{out}$, MDR_{in}, Write ;data to be written are loaded into MDR & Write command is issued
  3) MDR_{outE}, WMFC ;load data into memory location pointed by R1 from MDR

EXECUTION OF A COMPLETE INSTRUCTION

- Consider the instruction *Add (R3),R1* which adds the contents of a memory-location pointed by R3 to register R1. Executing this instruction requires the following actions:
  1) Fetch the instruction.
  2) Fetch the first operand.
  3) Perform the addition.
  4) Load the result into R1.
- Control sequence for execution of this instruction is as follows
  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
- Instruction execution proceeds as follows:

  Step1--> The instruction-fetch operation is initiated by loading contents of PC into MAR & sending a Read request to memory. The Select signal is set to Select4, which causes the Mux to select constant 4. This value is added to operand at input B (PC's content), and the result is stored in Z

  Step2--> Updated value in Z is moved to PC.

  Step3--> Fetched instruction is moved into MDR and then to IR.

  Step4--> Contents of R3 are loaded into MAR & a memory read signal is issued. Step5--> Contents of R1 are transferred to Y to prepare for addition.

  Step6--> When Read operation is completed, memory-operand is available in MDR, and the addition is performed.

  Step7--> Sum is stored in Z, then transferred to R1.The End signal causes a new instruction fetch cycle to begin by returning to step1.

BRANCHING INSTRUCTIONS

- Control sequence for an unconditional branch instruction is as follows:
  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
- The processing starts, as usual, the fetch phase ends in step3.
- In step 4, the offset-value is extracted from IR by instruction-decoding circuit.

- Since the updated value of PC is already available in register Y, the offset X is gated onto the bus, and an addition operation is performed.
- In step 5, the result, which is the branch-address, is loaded into the PC.
- 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 1000 and branch target-address is 1200, then the value of X must be 196, since the PC will be containing the address 1004 after fetching the instruction at location 1000).
- In case of conditional branch, we need to check the status of the condition-codes before loading a new value into the PC.

e.g.: Offset-field-of-IR$_{out}$, Add, Z$_{in}$, If N=0 then End
If N=0, processor returns to step 1 immediately after step 4. If N=1, step 5 is performed to load a new value into PC.

MULTIPLE BUS ORGANIZATION

- All general-purpose registers are combined into a single block called the *register file*.
- Register-file has 3 ports. There are 2 outputs allowing the contents of 2 different registers to be simultaneously placed on the buses A and B.
- Register-file has 3 ports.
  1) Two output-ports allow the contents of 2 different registers to be simultaneously placed on buses A & B.
  2) Third input-port allows data on bus C to be loaded into a third register during the same clock-cycle.
- Buses A and B are used to transfer source-operands to A & B inputs of ALU.
- Result is transferred to destination over bus C.
- Incrementer-unit is used to increment PC by 4.
- Control sequence for the instruction *Add R4,R5,R6* is as follows
  1) PC$_{out}$, R=B, MAR$_{in}$, Read, IncPC
  2) WMFC
  3) MDR$_{out}$, R=B, IR$_{in}$
  4) R4$_{out}$A, R5$_{out}$B, SelectA, Add, R6$_{in}$, End
- Instruction execution proceeds as follows:

Step 1--> Contents of PC are passed through ALU using R=B control-signal and loaded into MAR to start a memory Read operation. At the same time, PC is incremented by 4.
Step2--> Processor waits for MFC signal from memory.
Step3--> Processor loads requested-data into MDR, and then transfers them to IR. Step4--> The instruction is decoded and add operation take place in a single step.

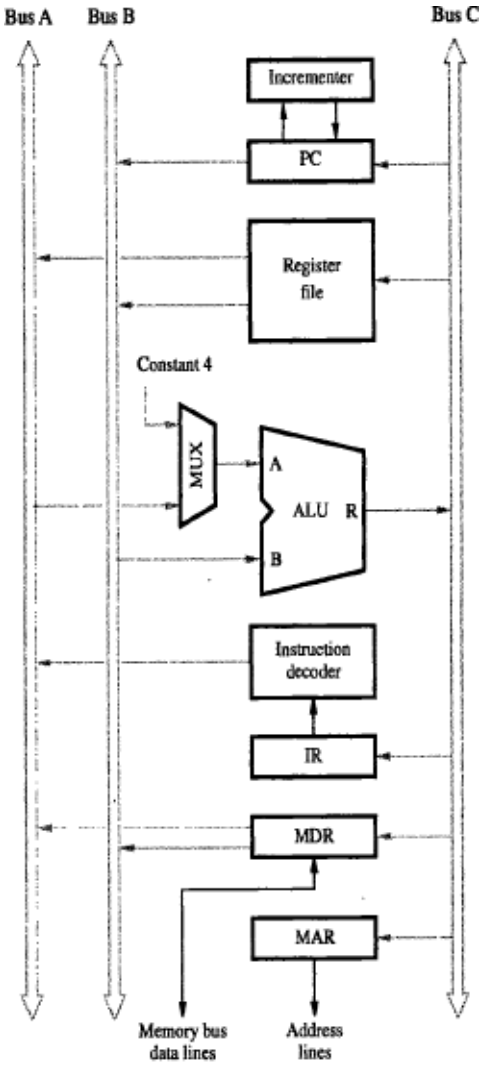


Figure 7.8 Three-bus organization of the datapath.

Note:

To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence. There are two approaches for this purpose:
1) Hardwired control and 2) Microprogrammed control.

HARDWIRED CONTROL

- Decoder/encoder block is a combinational-circuit that generates required control-outputs depending on state of all its inputs.
- Step-decoder provides a separate signal line for each step in the control sequence. Similarly, output of instruction-decoder consists of a separate line for each machine instruction.
- For any instruction loaded in IR, one of the output-lines INS_1 through INS_m is set to 1, and all other lines are set to 0.
- The input signals to encoder-block are combined to generate the individual control-signals Y_{in}, PC_{out}, Add, End and so on.
- For example, $Z_{in}=T1+T6.ADD+T4.BR$;This signal is asserted during time-slot T_1 for all instructions,

during T6 for an Add instruction
during T4 for unconditional branch instruction

- When RUN=1, counter is incremented by 1 at the end of every clock cycle. When RUN=0, counter stops counting.
- Sequence of operations carried out by this machine is determined by wiring of logic elements, hence the name "hardwired".
- Advantage: Can operate at high speed. Disadvantage: Limited flexibility.

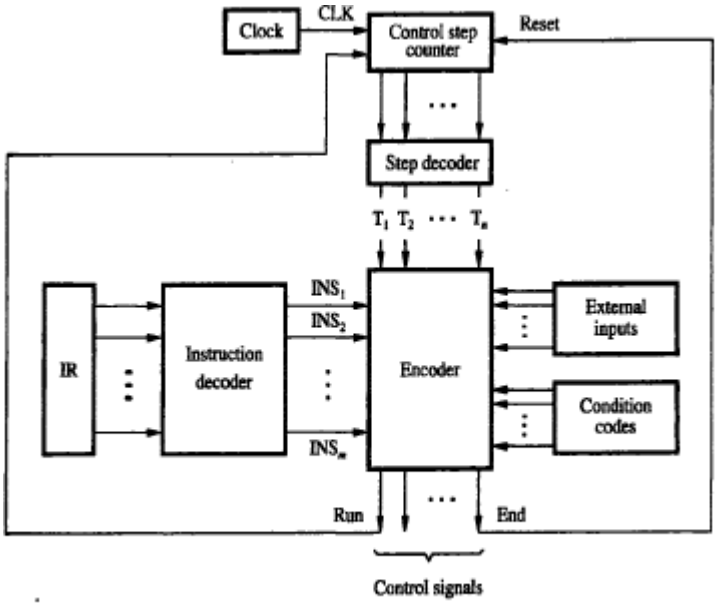


Figure 7.11 Separation of the decoding and encoding functions.

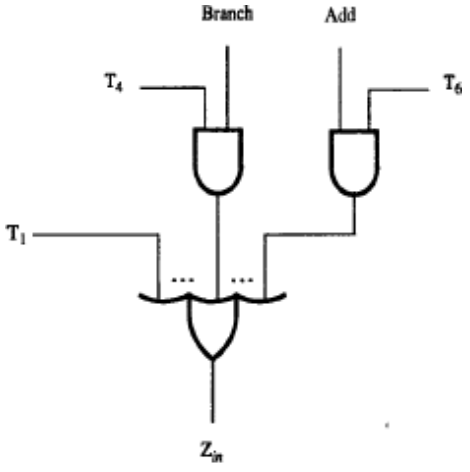


Figure 7.12 Generation of the Z_{in} control signal for the processor in Figure 7.1.

COMPLETE PROCESSOR

- This has separate processing-units to deal with integer data and floating-point data.
- A data-cache is inserted between these processing-units & main-memory.
- Instruction-unit fetches instructions
  → from an instruction-cache or
  → from main-memory when desired instructions are not already in cache
- Processor is connected to system-bus &
  hence to the rest of the computer by means of a bus interface
- Using separate caches for instructions & data is common practice in many processors today.
- A processor may include several units of each type to increase the potential for concurrent operations.

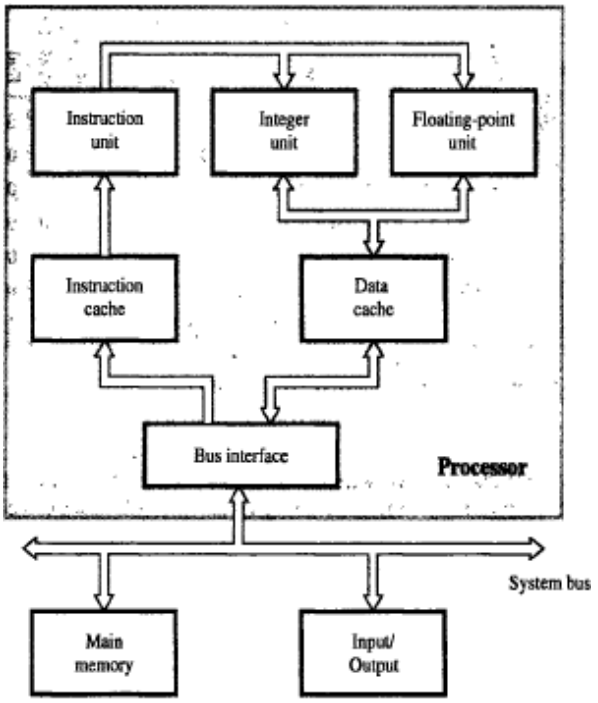


Figure 7.14 Block diagram of a complete processor.

MICROPROGRAMMED CONTROL

- Control-signals are generated by a program similar to machine language programs.
- *Control word(CW)* is a word whose individual bits represent various control-signals(like Add, End, Z_{in}). {Each of the control-steps in control sequence of an instruction defines a unique combination of 1s & 0s in the CW}.
- Individual control-words in microroutine are referred to as *microinstructions.*
- A sequence of CWs corresponding to control-sequence of a machine instruction constitutes the *microroutine*.
- The microroutines for all instructions in the instruction-set of a computer are stored in a special memory called the *control store(CS).*

- Control-unit generates control-signals for any instruction by sequentially reading CWs of corresponding microroutine from CS.
- *Microprogram counter(µPC)* is used to read CWs sequentially from CS.
- Every time a new instruction is loaded into IR, output of "starting address generator" is loaded into µPC.
- Then, µPC is automatically incremented by clock,
  causing successive microinstructions to be read from CS.
  Hence, control-signals are delivered to various parts of processor in correct sequence.

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

Figure 7.15 An example of microinstructions for Figure 7.6.

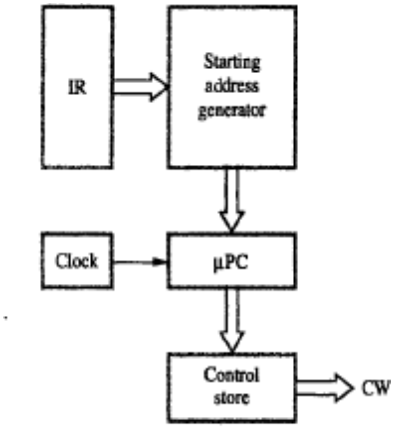


Figure 7.16 Basic organization of a microprogrammed control unit.

ORGANIZATION OF MICROPROGRAMMED CONTROL UNIT (TO SUPPORT CONDITIONAL BRANCHING)

• In case of conditional branching, microinstructions specify which of the external inputs, condition-codes should be checked as a condition for branching to take place.
• The *starting and branch address generator block* loads a new address into µPC when a microinstruction instructs it to do so.
• To allow implementation of a conditional branch, inputs to this block consist of
→ external inputs and condition-codes
→ contents of IR
• µPC is incremented every time a new microinstruction is fetched from microprogram memory except in following situations

i) When a new instruction is loaded into IR, µPC is loaded with starting-address of microroutine for that instruction.

ii) When a Branch microinstruction is encountered and branch condition is satisfied, µPC is loaded with branch-address.

iii) When an End microinstruction is encountered, µPC is loaded with address of first CW in microroutine for instruction fetch cycle.

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

Figure 7.17 Microroutine for the instruction Branch < 0.

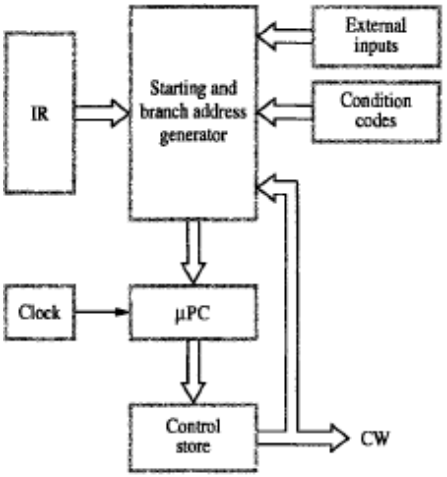


Figure 7.18 Organization of the control unit to allow conditional branching in the microprogram.

MICROINSTRUCTIONS

- Drawbacks of microprogrammed control:
    1) Assigning individual bits to each control-signal results in long microinstructions because the number of required signals is usually large.
    2) Available bit-space is poorly used because only a few bits are set to 1 in any given microinstruction.
- Solution: Signals can be grouped because
    1) Most signals are not needed simultaneously.
    2) Many signals are mutually exclusive.
- Grouping control-signals into fields requires a little more hardware because decoding-circuits must be used to decode bit patterns of each field into individual control signals.
- Advantage: This method results in a smaller control-store (only 20 bits are needed to store the patterns for the 42 signals).

Vertical organization	Horizontal organization
Highly encoded schemes that use compact codes to specify only a small number of control functions in each microinstruction are referred to as a vertical organization	The minimally encoded scheme in which many resources can be controlled with a single microinstuction is called a horizontal organization
This approach results in considerably slower operating speeds because more micrinstructions are needed to perform the desired control functions	This approach is useful when a higher operating speed is desired and when the machine structure allows parallel use of resources

Microinstruction

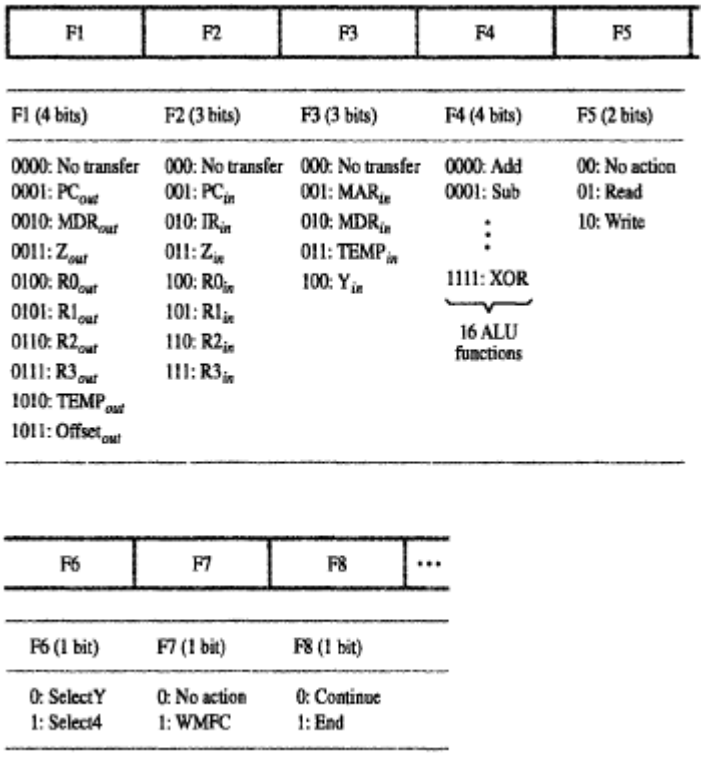


Figure 7.19 An example of a partial format for field-encoded microinstructions.

- **MICROPROGRAM SEQUENCING** Two major disadvantage of microprogrammed control is:
    1) Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control-store.
    2) Execution time is longer because it takes more time to carry out the required branches.
- Consider the instruction *Add src,Rdst* ;which adds the source-operand to the contents of Rdst and places the sum in Rdst.
- Let source-operand can be specified in following addressing modes: register, autoincrement, autodecrement and indexed as well as the indirect forms of these 4 modes.
- Each box in the chart corresponds to a microinstruction that controls the transfers and operations indicated within the box.
- The microinstruction is located at the address indicated by the octal number (001,002).

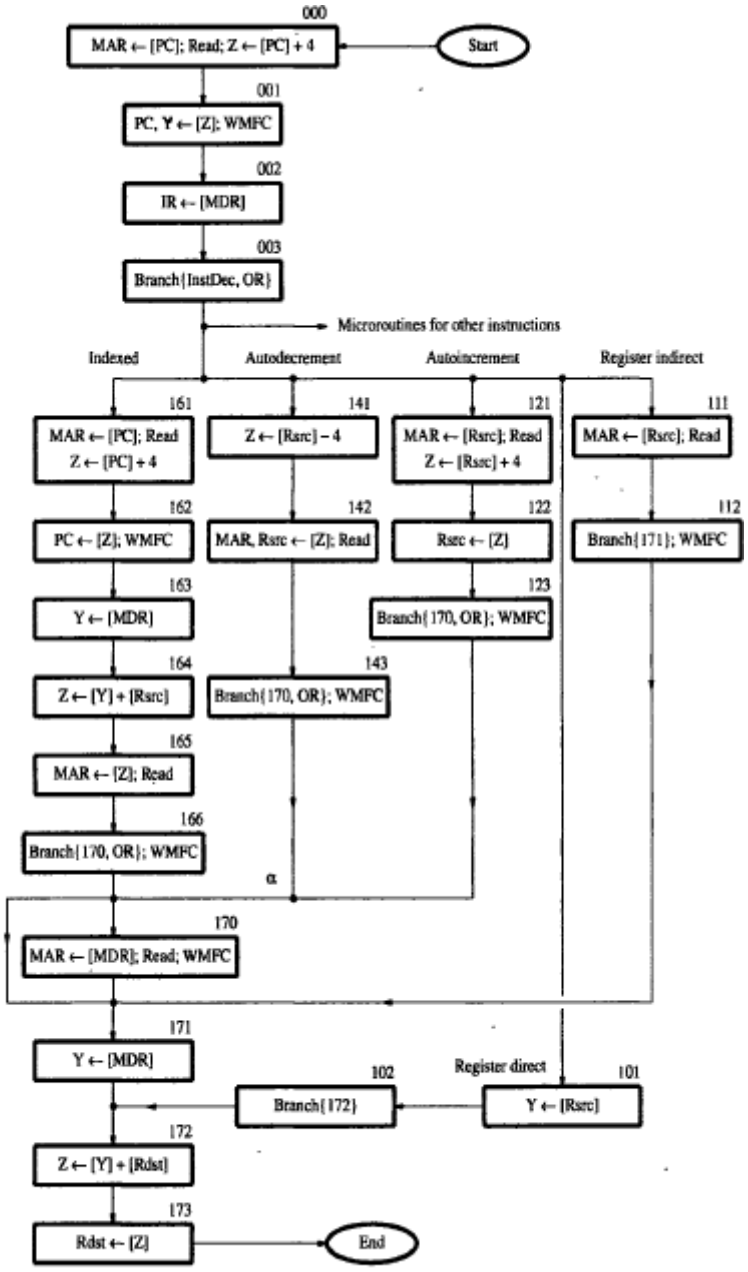


Figure 7.20 Flowchart of a microprogram for the Add src,Rdst instruction.

BRANCH ADDRESS MODIFICATION USING BIT-ORING

- Consider the point labeled in the figure. At this point, it is necessary to choose between direct and indirect addressing modes.
- If indirect-mode is specified in the instruction, then the microinstruction in location 170 is performed to fetch the operand from the memory.

If direct-mode is specified, this fetch must be bypassed by branching immediately to location 171.

- The most efficient way to bypass microinstruction 170 is to have the preceding branch microinstructions specify the address 170 and then use an OR gate to change the LSB of this address

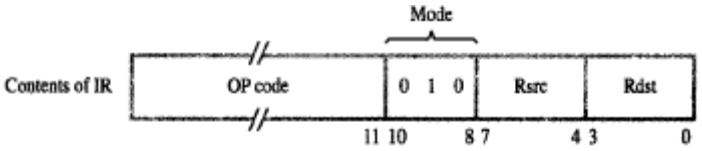
to 1 if the direct addressing mode is involved. This is known as the *bit-ORing*technique.

WIDE BRANCH ADDRESSING

- The instruction-decoder(InstDec) generates the starting-address of the microroutine that implements the instruction that has just been loaded into the IR.
- Here, register IR contains the Add instruction, for which the instruction decoder generates the microinstruction address 101. (However, this address cannot be loaded as is into the μPC).
- The source-operand can be specified in any of several addressing-modes. The bit-ORing technique can be used to modify the starting-address generated by the instruction-decoder to reach the appropriate path.

Use of WMFC

- WMFC signal is issued at location 112 which causes a branch to the microinstruction in location 171.
- WMFC signal means that the microinstruction may take several clock cycles to complete. If the branch is allowed to happen in the first clock cycle, the microinstruction at location 171 would be fetched and executed prematurely. To avoid this problem, WMFC signal must inhibit any change in the contents of the μPC during the waiting-period. **Detailed Examination**
- Consider *Add (Rsrc)+,Rdst;* which adds Rsrc content to Rdst content, then stores the sum in Rdst and finally increments Rsrc by 4 (i.e. auto-increment mode).
- In bit 10 and 9, bit-patterns 11, 10, 01 and 00 denote indexed, auto-decrement, auto-increment and register modes respectively. For each of these modes, bit 8 is used to specify the indirect version.
- The processor has 16 registers that can be used for addressing purposes; each specified using a 4-bit-code.
- There are 2 stages of decoding:
    1) The microinstruction field must be decoded to determine that an Rsrc or Rdst register is involved.



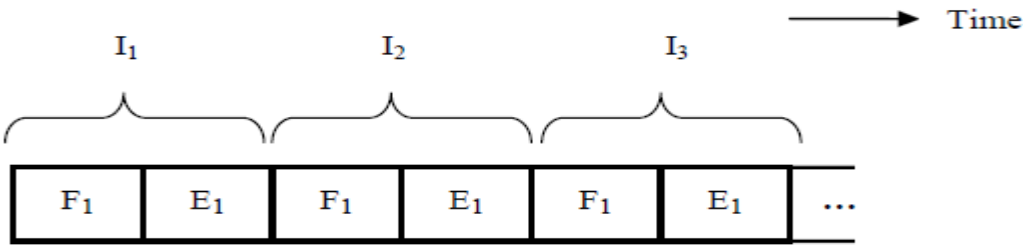
Address (octal)	Microinstruction
000	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
001	Z_{out}, PC_{in}, Y_{in}, WMFC
002	MDR_{out}, IR_{in}
003	μBranch {μPC ← 101 (from Instruction decoder); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$}
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, Z_{in}
122	Z_{out}, $Rsrc_{in}$
123	μBranch {μPC ← 170; $\mu PC_0 \leftarrow [\overline{IR_8}]$}, WMFC
170	MDR_{out}, MAR_{in}, Read, WMFC
171	MDR_{out}, Y_{in}
172	$Rdst_{out}$, SelectY, Add, Z_{in}
173	Z_{out}, $Rdst_{in}$, End

Figure 7.21 Microinstruction for Add (Rsrc)+,Rdst.

The de

Basic concepts of pipeline: In computer architecture Pipelining means executing machine instructions concurrently. The pipelining is used in modern computers to achieve high performance. 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 anyone operation is not changed.

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. The processor executes a program by fetching and executing instructions, one after the other. Let Fi and Ei refer to the fetch and execute steps for instruction Ii. Executions of a program consists of a sequence of fetch and execute steps, as shown in Figure 3.l.



Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure 3.2. 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. The data can be operated by the instructions are inside the block labeled "Execution unit".

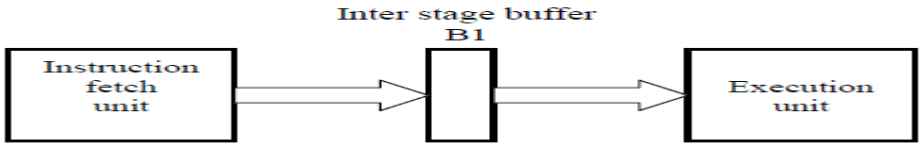


Figure 3.2 Hardware organization of pipelining.

The computer is controlled by a clock whose period is such that the fetch and execute steps of any instruction can each be completed in one clock cycle. Operation of the computer proceeds as in Figure 3.3. In the first clock cycle, the fetch unit fetches an instruction I1 (step F1) and stores it in buffer Bl at the end of the clock cycle. In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I2 (step F2). Meanwhile, the execution unit performs the operation specified by instruction I1, which is available to it in buffer Bl (step E1). By the end of the second 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.

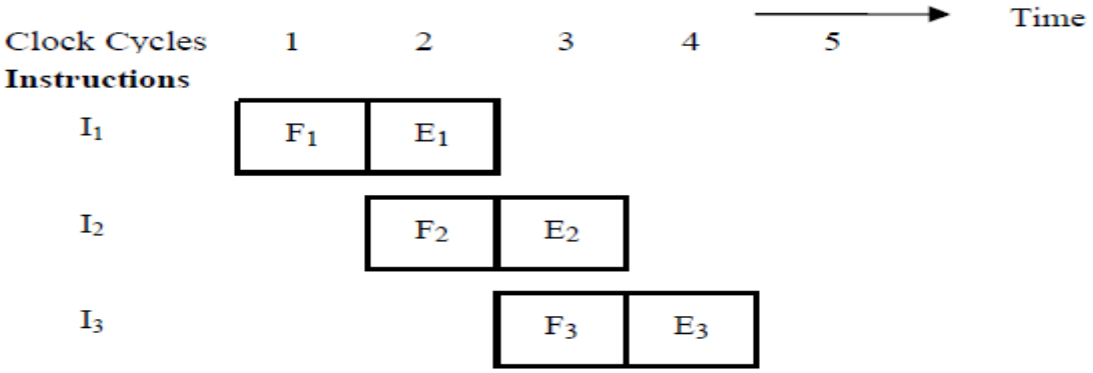


Figure 3.3 Pipelined executions of instructions (Instructions Pipelining).

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 8.4. Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Figure 8.5. These units must be capable of performing their tasks simultaneously and without interfering with one another. Information is passed from one unit to the next through a storage buffer.

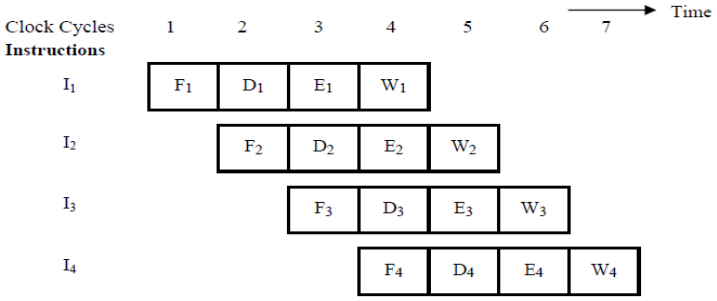


Figure 3.4 Instruction execution divided into four steps.

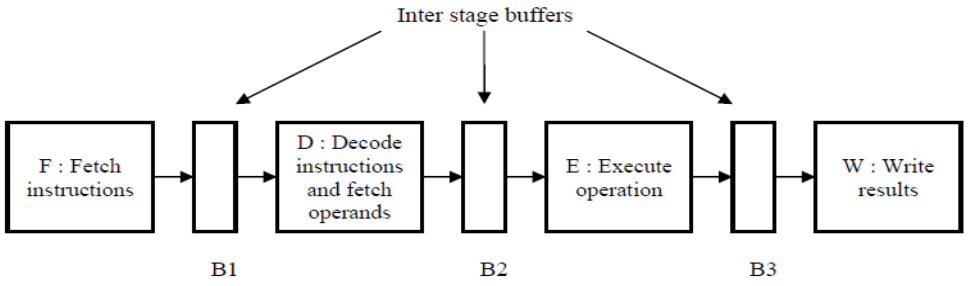


Figure 3.5 Hardware organization of a 4-stage pipeline.

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.
- Buffer B3 holds the results produced by the execution unit and the destination information for instruction 11.

PIPELINE PERFORMANCE: The pipelined processor in Figure 3.4 completes the processing of one instruction in each clock cycle, which means that the rate of instruction processing is four times that of sequential operation. The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages. However, this increase would be achieved only if pipelined operation as depicted in Figure 3.4 could be sustained without interruption throughout program execution. Unfortunately, this is not the case.

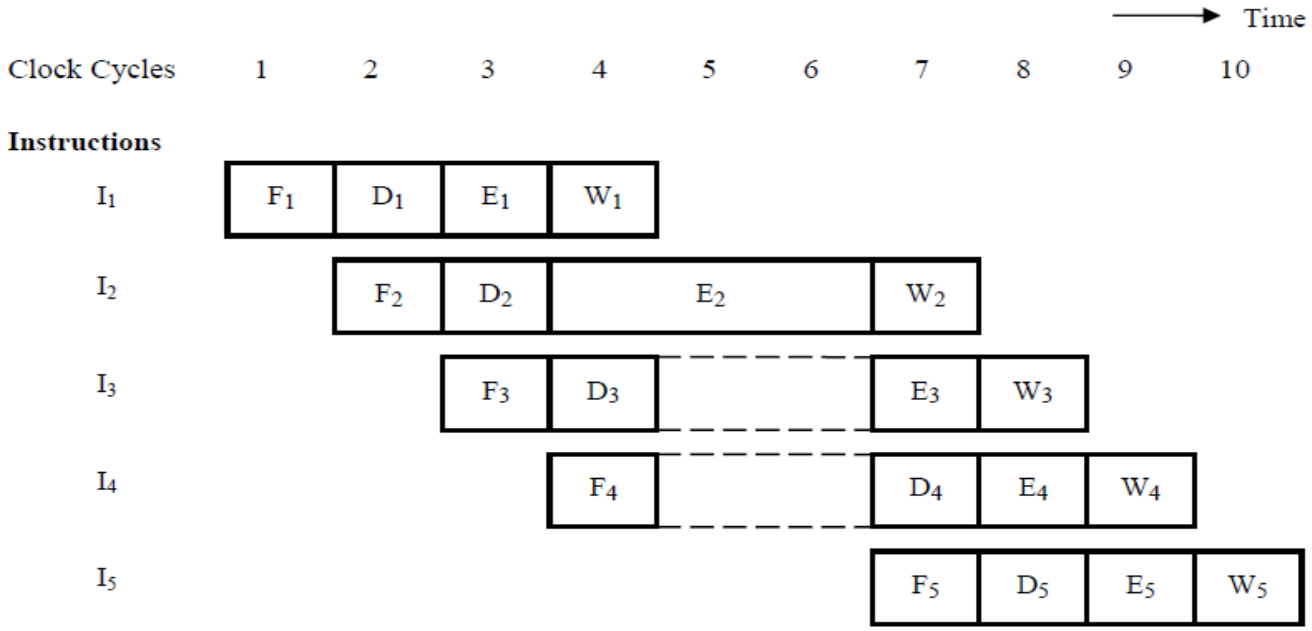


Figure 3.6 Effect of an execution operation taking more than one clock cycle.
Figure 3.6 shows an example in which the operation specified in instruction I2 requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the Write stage must be told to do nothing, because it has no data to work with. Meanwhile, the information in buffer B2 must remain intact until the Execute stage has completed its operation. This means that stage 2 and, in turn, stage1 are blocked from accepting new instructions because the information in B1 cannot be overwritten. Thus, steps D4 and F5 must be postponed as shown in figure 3.6.

Pipelined operation in Figure 3.6 is said to have been stalled for two clock cycles. Normal pipelined operation resumes in cycle 7. Any condition that causes the pipeline to stall is called a hazard.

I**. DATA HAZARDS**: A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result some operation has to be delayed, and the pipeline stalls.

II. CONTROL/INSTRUCTION HAZARDS: The pipeline may also be stalled because of a delay in the availability of an instruction. For example, this may be a result of a miss in the cache, requiring the

instruction to be fetched from the main memory. Such hazards are often called control hazards or instruction hazards.

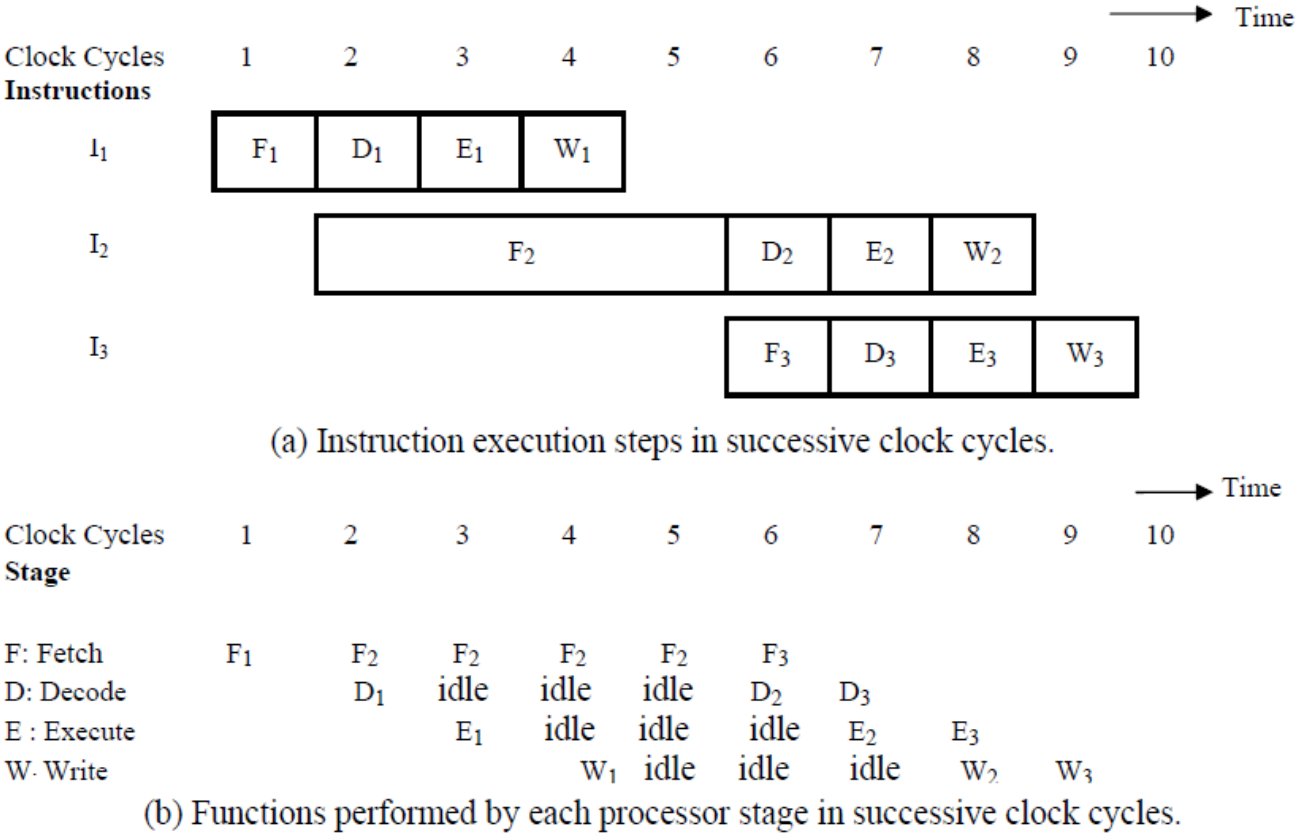


Figure 3.7 Pipeline stall caused by a cache miss in F2.

The effect of a cache miss on pipelined operation is illustrated in Figure 3.7. Instruction I1 is fetched from the cache in cycle 1, and its execution proceeds normally. However, the fetch operation for instruction I2, which is started in cycle 2, results in a cache miss. The instruction fetch unit must now suspend any further fetch requests and wait for I2 to arrive. We assume that instruction I2 is received and loaded into buffer Bl at the end of cycle 5. The pipeline resumes its normal operation at that point.

Data hazards: A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result some operation has to be delayed, and the pipeline stalls. A data hazard is a situation in which the pipeline is stalled because the data to be operated on are delayed for some reason. Consider a program that contains two instructions, I1 followed by I2. When this program is executed in a pipeline, the execution of 12 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 ← 3 A
B ← 4 * A

When these operations are performed in the order given, the result is B == 32. But if they are performed concurrently, the value of A used in computing B would be the original value, 5, leading to an incorrect result. If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because the data used in the second instruction depend on the result of the first instruction. On the other hand, the two operations

A ← 5 x C

B ← 20 C

can be performed concurrently, because these operations are independent.

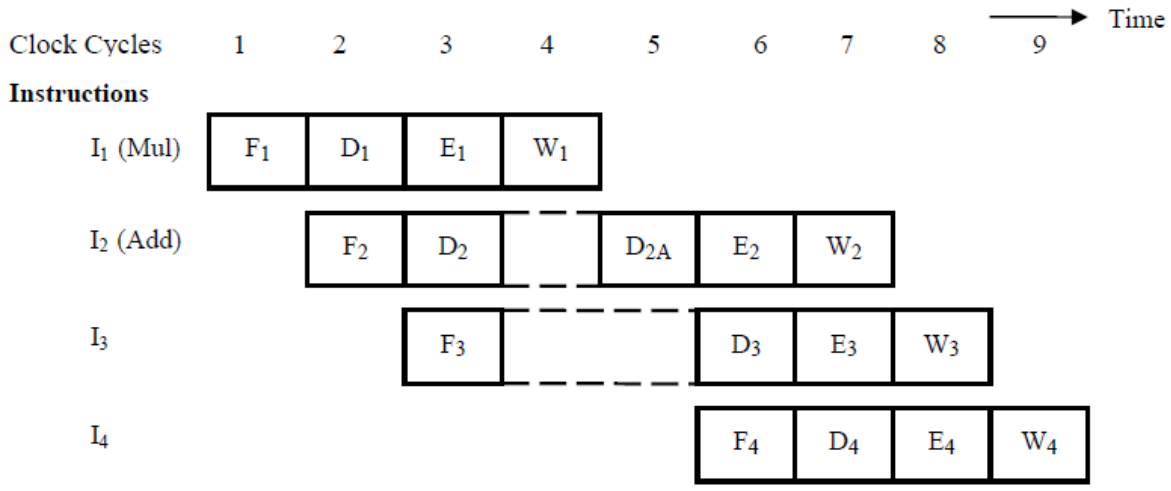
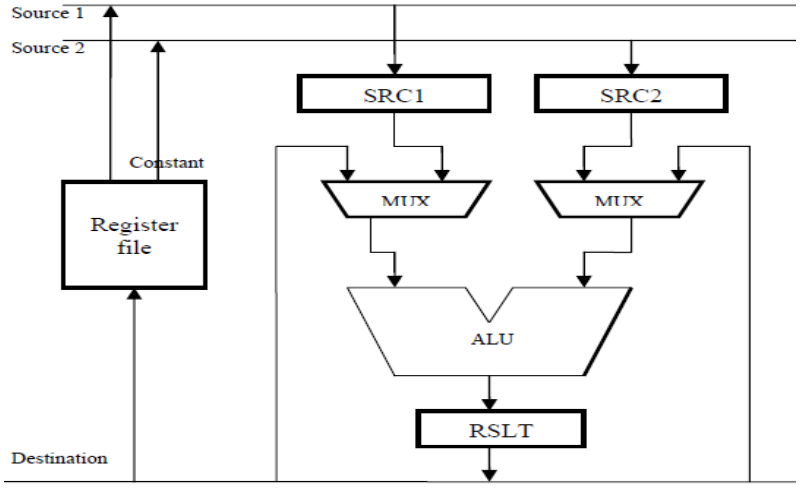
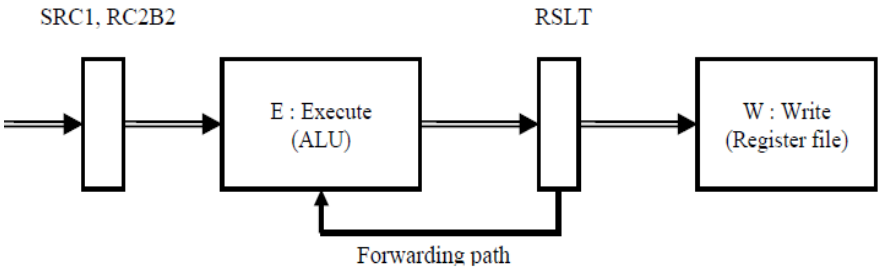


Figure 3.9 Pipeline stalled by data dependency between D2 and W1.
This example illustrates a basic constraint that must be enforced to guarantee correct results. When two operations depend on each other, they must be performed sequentially in the correct order. This rather obvious condition has far-reaching consequences. Understanding its implications is the key to understanding the variety of design alternatives and trade-off's encountered in pipelined computers. For example, the two instructions Mul R2,R3,R4 and Add RS,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 3.9. 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 D2A in the figure. Instruction h 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 3.9, 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 El. 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.



(a) Forwarding data path



(b) Posision of the source and result registers in the processor pipeline.

Figure 3.10 Operand forwarding in a pipelined processor. Figure 3.10a shows a part of the processor datapath involving the ALU and theregister file. This arrangement is similar to the three-bus structur, except that registers SRCl, SRC2, and RSLT have been added. These registers constitute interstage buffers needed for pipelined operation, as illustrated in Figure 3.10b. With reference to Figure 3.10b, 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 SRCI or SRC2 register. When the instructions in Figure 3.9 are executed in the datapath of Figure 3.10, 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 SRCI 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: 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. 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. 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 EFFECT

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 autoincrement or autodecrement 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 autoincrement or autodecrement 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 autoincrement and autodecrement 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 Rl 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 Rl,R3

AddWithCarry 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 ← [R2] [R4] 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 pipe lined 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.

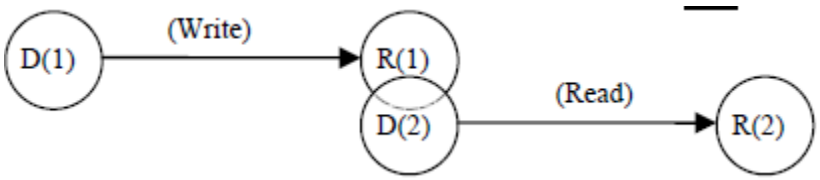
CLASSIFICATION OF DATA DEPENDENT HAZARDS

The Data dependent hazards can be classified into three types according to various data update patterns, Consider two instructions I1 and I2, with I1 occurring before I2 in program order.

I. Read After Write (RAW) (flow dependence hazard) ($R(1) \cap D(2) \neq \varphi$)

(I2 tries to read a source before I1 writes to it) A read after write (RAW) data hazard refers to a situation where an instruction refers to a result that has not yet been calculated or retrieved. This can occur because even though an instruction is executed after a previous instruction, the previous instruction has not been completely processed through the pipeline. ExampleFor example:

I1. R2 <- R1 R3
I2. R4 <- R2 R3

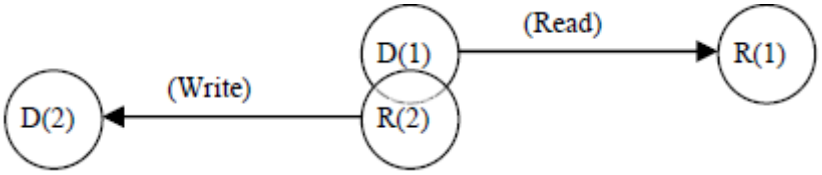


D(1): Domain of instruction 1
R(1): Range of instruction 1
D(2): Domain of instruction 2
R(2): Range of instruction 2

The first instruction is calculating a value to be saved in register R2, and the second is going to use this value to compute a result for register R4. However, in a pipeline, when we fetch the operands for the 2nd operation, the results from the first will not yet have been saved, and hence we have a data dependency. There is a data dependency with instruction I2, as it is dependent on the completion of instruction I1.

II. Write After Read (WAR) (Anti dependence hazard) ($D(1) \cap R(2) \neq \varphi$)

(I2 tries to write a destination before it is read by I1) A write after read (WAR) data hazard represents a problem with concurrent execution. ExampleFor example: I1. R4 <- R1 R3
I2. R3 <- R1 R2

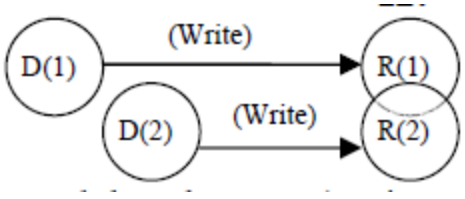


If we are in a situation that there is a chance that I2 may be completed before I1 (i.e. with concurrent execution) we must ensure that we do not store the result of register R3 before I1 has had a chance to fetch the operands.

III. Write After Write (WAW) (Output dependence hazard) ($R(1) \cap R(2) \neq \varphi$)

(I2 tries to write an operand before it is written by I1) A write after write (WAW) data hazard may occur in a concurrent execution environment. ExampleFor example:

I1. R2 <- R4 R7
I2. R2 <- R1 R2



There must delay the WB (Write Back) of I2 until the execution of I1.

Possible hazards for various instruction types

	First Instruction			
Second Instruction	Arithmetic & load type	Store type	Branch type	Conditional Branch type
Arithmetic & load type	RAW, WAR, WAW	RAW, WAR	WAR	WAR
Store type	RAW, WAR	WAW		
Branch type	RAW		WAW	
Conditional Branch type	RAW		WAW	

Instruction hazards: Pipeline execution of instructions will reduce the time and improves the performance. Whenever this stream is interrupted, the pipeline stalls, as figure 3.7 illustrates for the case of a cache miss. A branch instruction may also cause the pipeline to stall. The effect of branch instructions and the techniques that can be used for mitigating their impact are discussed with unconditional branches and conditional branches.

UNCONDITIONAL BRANCHES: A sequence of instructions being executed in a two-stage pipeline is shown in Figure 3.11. 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 13 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.

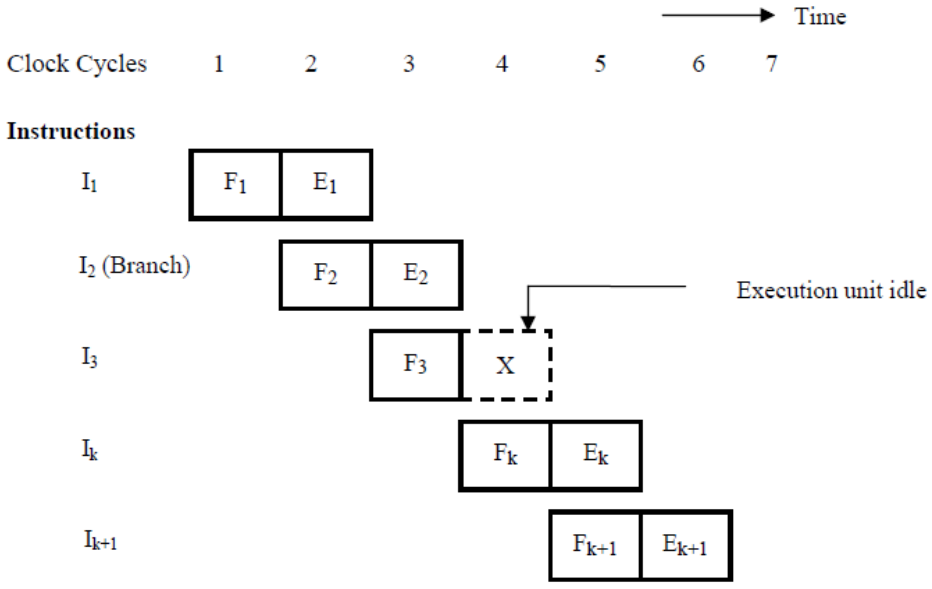


Figure 3.11 An idle cycle caused by a branch instruction.

The time lost as a result of a branch instruction is often referred to as the branch penalty (Time loss). In Figure 3.11, the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher. For examp1e, Figure 3.10a shows the effect of a branch instruction on a four stage pipeline. The

branch address is computed in step E2. Instructions I3 and I4 must be discarded, and the target instruction, Ik, 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 3.10b. In this case, the branch penalty is only one clock cycle.

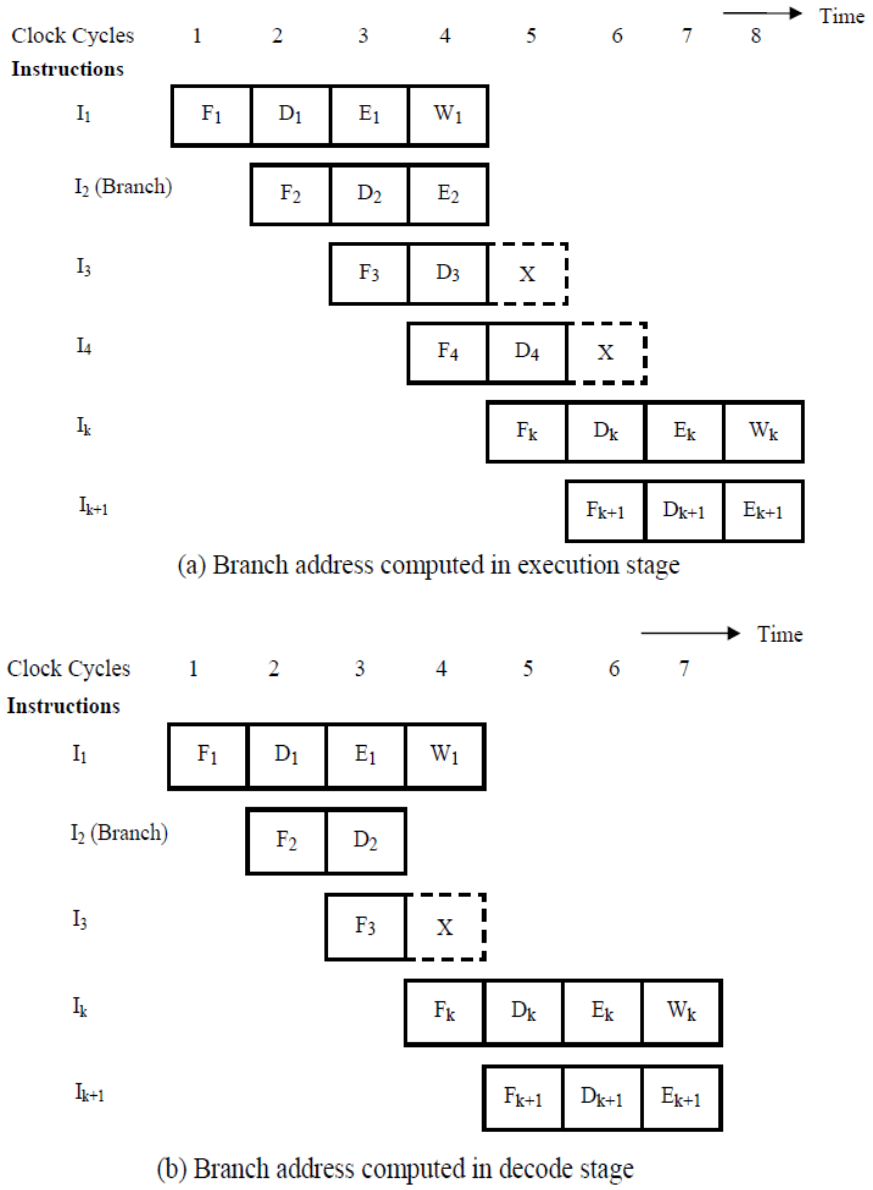


Figure 3.10 Branch timing.

Either a cache miss or a branch instruction stalls the pipeline for one or more clock cycles. To reduce the effect of these interruptions, many processors employ sophisticated fetch units that can fetch instructions before they are needed and put them in a queue. Typically, the instruction queue can store several instructions. A separate unit, which we call the dispatch unit, takes instructions from the front of the queue and sends them to the execution unit. This leads to the organization shown in Figure 3.11. 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. 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. The fetch unit continues to fetch instructions and add them to the queue.

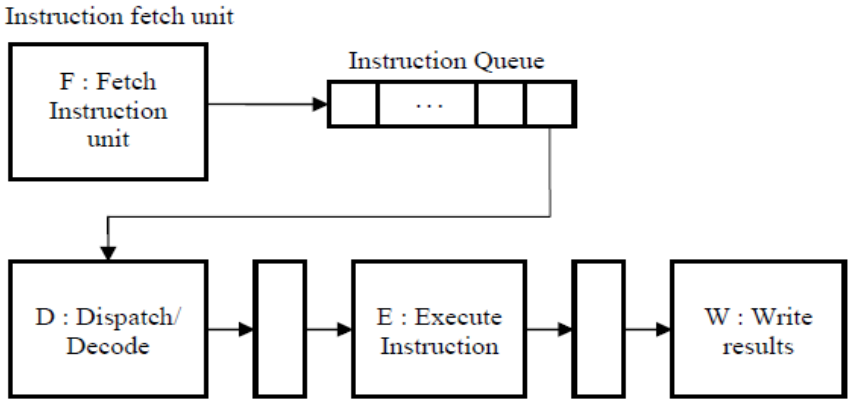
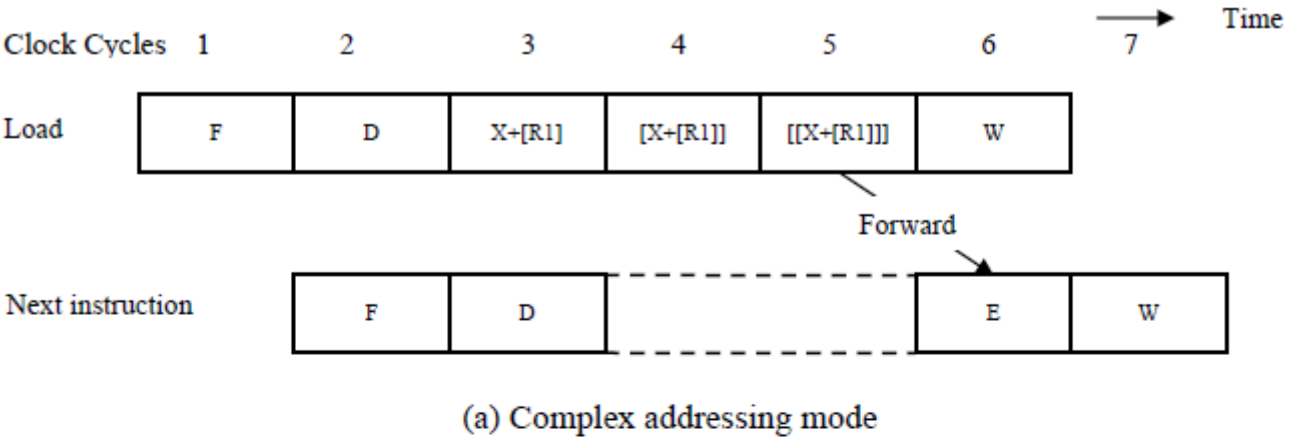


Figure 3.11 Use of instruction queue in hardware organization.
To be effective, the fetch unit must have sufficient decoding and processing capability to recognize and execute branch instructions. It attempts to keep the instruction queue filled at all times to reduce the impact of occasional delays when fetching instructions. 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. The fetch unit continues to fetch instructions and add them to the queue.

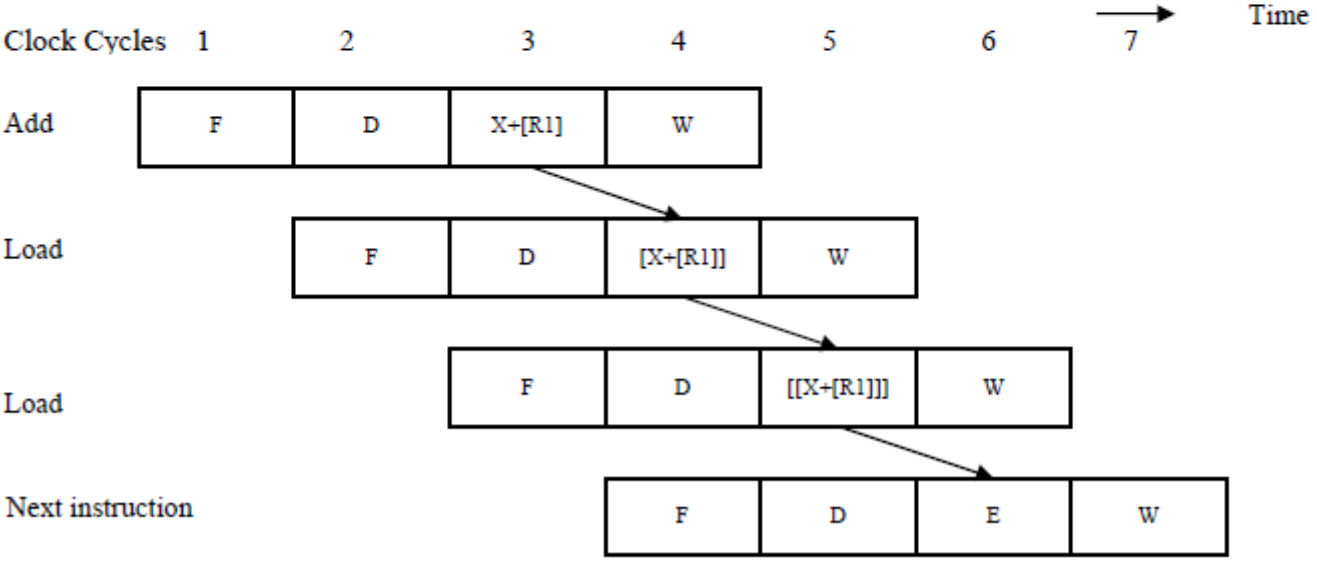
Influence on Instruction Sets: Some instructions are much better suited to pipelined execution than other instructions. For example, instruction side effects can lead to undesirable data dependencies. The machine instructions are influenced by addressing modes and condition code flags.

I. 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.

Two important considerations in this regard are the side effects of addressing 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.



(a) Complex addressing mode



(a) Simple addressing mode

Figure 3.17 Equivalent operations using complex and simple addressing mode.

Assume a simple model for accessing operands in the memory. The load instruction Load X(RI),R2 takes five cycles to complete execution, However, the instruction Load (Rl),R2 can be organized to fit a four-stage pipeline because no address computation is required. Access to memory can take place in stage E. A more complex addressing mode may require several accesses to the memory to reach the named operand.

For example, the instruction Load (X(Rl)),R2

It may be executed as shown in Figure 3.17a, assuming that the index offset, X, is given in the instruction, word. Mter computing the address in cycle 3, the processor needs to access memory twice - first to read location X [Rl] in clock cycle 4 and then to read location [X [Rl]] 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 in figure 3.17. To implement the same Load operation using only simple addressing modes requires several instructions. For example, on a computer that allows three operand addresses like

Add #X,Rl,R2
Load (R2),R2
Load (R2),R2

The Add instruction performs the operation R2 ← X [RI], 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 3.17b.
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 modem processors are designed to take maximum advantage of pipelined hardware. The addressing modes used in modem 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 architecture, 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.

II. Condition modes

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

Add Rl,R2
Compare R3,R4
Branch=0 …

Assume that the execution of the Compare and Branch = 0 instructions proceeds as in Figure 3.14. The branch decision takes place in step E3 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,**

Compare R3,R4
Add Rl,R2
Branch=0 …

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.

Datapath and control considerations: Consider the three-bus structure suitable for pipelined execution with a slight modification to support a 4-stage pipeline as shown in figure 3.18.

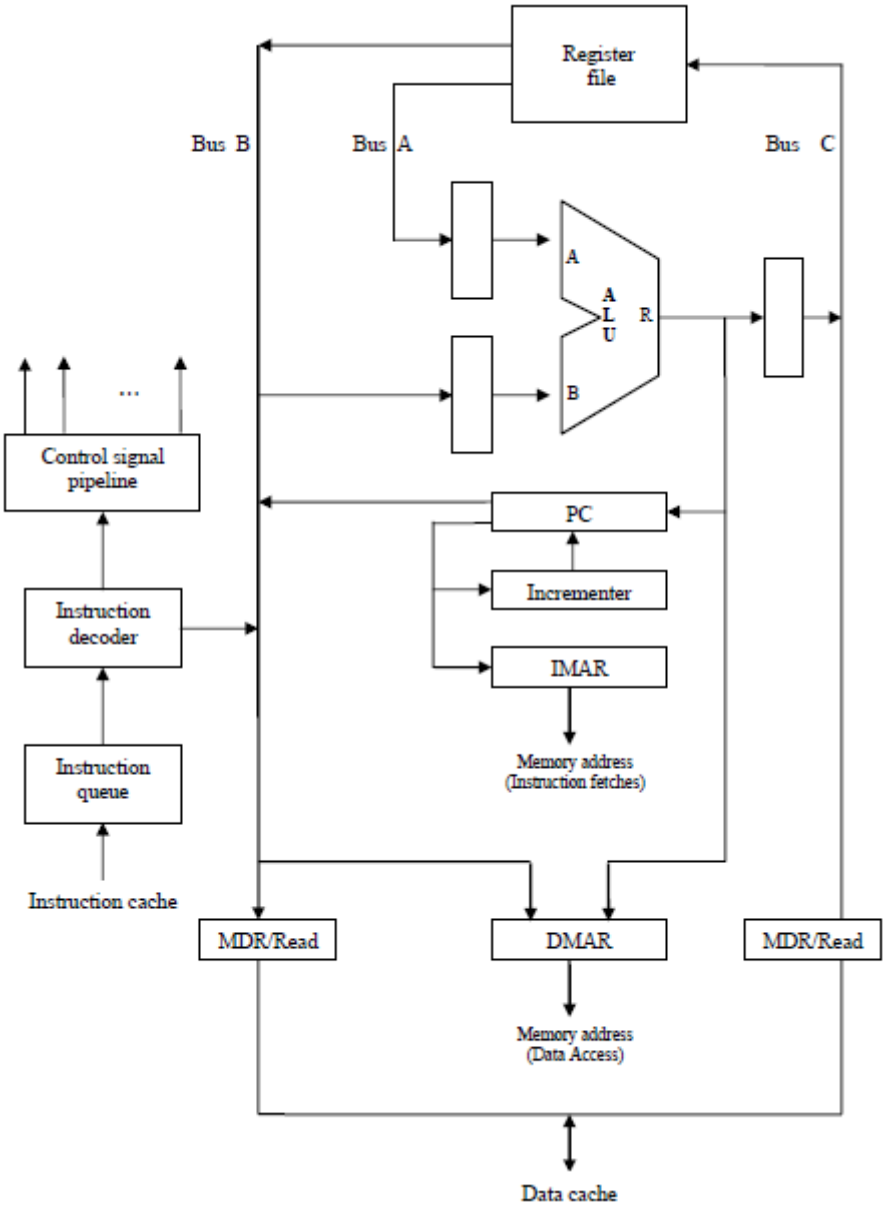


Figure 3.18 Datapath modified for pipelined execution with interstage buffers at the input and output of the ALU.

Several important changes are

1. There are separate instruction and data caches that use separate address and data connections to the processor. This requires two versions of the MAR register, IMAR for accessing tile instruction cache and DMAR for accessing the data cache.
2. The PC is connected directly to the IMAR, so that the contents of the PC can be transferred to IMAR at the same time that an independent ALU operation is taking place.
3. The data address in DMAR can be obtained directly from the register file or from the ALU to support the register indirect and indexed addressing modes.
4. Separate MDR registers are provided for read and write operations. Data can be transferred directly between these registers and the register file during load and store operations without the need to pass through the ALU.
5. Buffer registers have been introduced at the inputs and output of the ALU. These are registers SRCl, SRC2, and RSLT. Forwarding connections may be added if desired.
6. The instruction register has been replaced with an instruction queue, which is loaded from the instruction cache.
7. The output of the instruction decoder is connected to the control signal pipeline. This pipeline holds the control signals in buffers B2 and B3 in Figure 3.3.

The following operations can be performed independently in the processor of Figure 3.18:

- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two registers from the register file
- Writing into one register in the register file
- Performing an ALU operation

The processor execution time T, of a program that has a dynamic instruction count N is given by

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

Where S is the average number of clock cycles it takes to fetch and execute one instruction and R is the clock rate. This simple model assumes that instructions are executed one after the other, with no overlap. A useful performance indicator is the instruction throughput, which is the number of instructions executed per second. For sequential execution, the throughput, Ps is given by

$$P_s = R/S$$

In general, an n-stage pipeline has the potential to increase throughput n times. Thus, it would appear that the higher the value of n, the larger the performances gain. Any time a pipeline is stalled, the instruction throughput is reduced. Hence, the performance of a pipeline is highly influenced by factors such as branch and cache miss penalties.

I. SUPERSCALAR OPERATION: Pipelining makes it possible to execute instructions concurrently. Several instructions are present in the pipeline at the same time, but they are in different stages of their execution. While one instruction is performing an ALU operation, another instruction is being decoded and yet another is being fetched from the memory. Instructions enter the pipeline in strict program order. In the absence of hazards, one instruction enters the pipeline and one instruction completes

execution in each clock cycle. This means that the maximum throughput of a pipelined processor is one instruction per clock cycle.

A more aggressive approach is to equip the processor with multiple processing units to handle several instructions in parallel in each processing stage. With this arrangement, several instructions start execution in the same clock cycle, and the processor is said to use multiple-issue. Such processors are capable of achieving an instruction execution throughput of more than one instruction per cycle. They are known as superscalar processors. Many modem high-performance processors use this approach.

The instruction queue filled is introduced, so that a processor should be able to fetch more than one instruction at a time from the cache. For superscalar operation, this arrangement is essential. Multiple-issue operation requires a wider path to the cache and multiple execution units. Separate execution units are provided for integer and floating-point instructions.

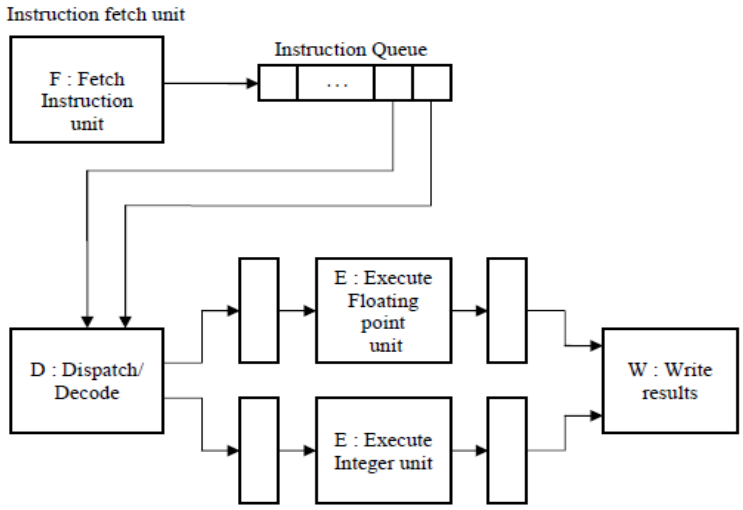


Figure 3.19 A Processor with two execution units

Figure 8.19 shows an example of a processor with two execution units, one for integer and one for floating-point operations. The Instruction fetch unit is capable of reading two instructions at a time and storing them in the instruction queue. In each clock cycle, the Dispatch unit retrieves and decodes up to two instructions from the front of the queue. If there is one integer, one floatingpoint instruction, and no hazards, both instructions are dispatched in the same clock cycle.

In a superscalar processor, the detrimental effect on performance of various hazards becomes even more pronounced. The compiler can avoid many hazards through judicious selection and ordering of instructions. For example, for the processor in Figure 3.19, the compiler should strive to interleave floating-point and integer instructions. This would enable the dispatch unit to keep both the integer and floating-point units busy most of the time. In general, high performance is achieved if the compiler is able to arrange program instructions to take maximum advantage of the available hardware units.

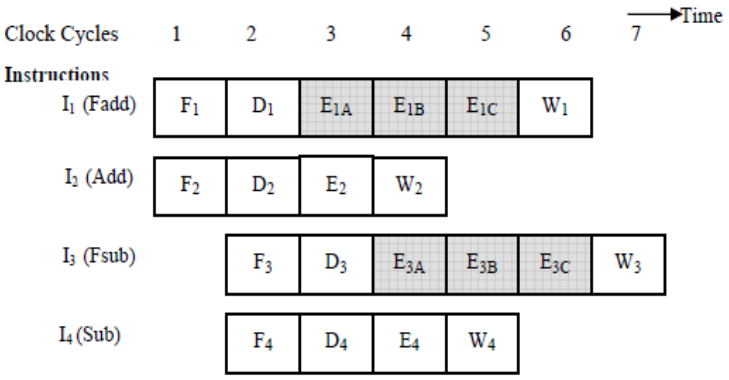
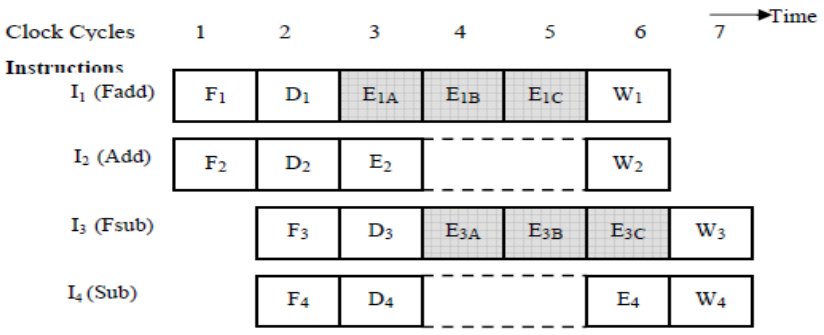


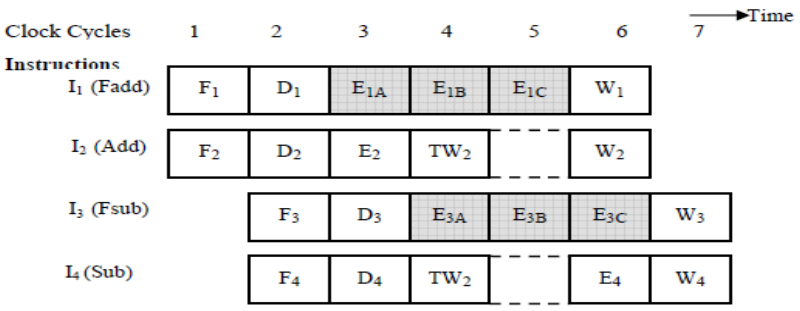
Figure 3.20 An Example of instruction flow in the Processor with assuming no hazards

Pipeline timing is shown in Figure 8.20. The blue shading indicates operations in the floating-point unit. The floating-point unit takes three clock cycles to complete the floating-point operation specified in I1. The integer unit completes execution of I2 in one clock cycle. We have also assumed that the floating-point unit is organized internally as a three-stage pipeline. Thus, it can still accept a new instruction in each clock cycle. Hence, instructions I3 and I4 enter the dispatch unit in cycle 3, and both are dispatched in cycle 4. The integer unit can receive a new instruction because instruction I2 has proceeded to the Write stage. Instruction I1 is still in the execution phase, but it has moved to the second stage of the internal pipeline in the floating-point unit. Therefore, instruction I3 can enter the first stage. Assuming that no hazards are encountered, the instructions complete execution as shown.

OUT-OF-ORDER EXECUTION: In Figure 8.20, instructions are dispatched in the same order as they appear in the program. However, their execution is completed out of order. Suppose one issue arise from dependencies among instructions. For example, if instruction I2 depends on the result of I1, the execution of I2 will be delayed. As long as such dependencies are handled correctly, there is no reason to delay the execution of an instruction. However, a new complication arises when we consider the possibility of an instruction causing an exception. Exceptions may be caused by a bus error during an operand fetch or by an illegal operation, such as an attempt to divide by zero. The results of I2 are written back into the register file in cycle 4. If instruction I1 causes an exception, program execution is in an inconsistent state. The program counter points to the instruction in which the exception occurred. However, one or more of the succeeding instructions have been executed to completion. If such a situation is permitted, the processor is said to have imprecise exceptions.



(a) Delayed write



(b) Using temporary registers

To guarantee a consistent state when exceptions occur, the results of the execution of instructions must be written into the destination locations strictly in program order. This means we must delay step W2 in Figure 8.20 until cycle 6. In turn, the integer execution unit must retain the result of instruction I2, and hence it cannot accept instruction I4 until cycle 6, as shown in Figure 8.2Ia. If an exception occurs during an instruction, all subsequent instructions that may have been partially executed are discarded. This is called a precise exception.
It is easier to provide precise exceptions in the case of external interrupts. When an external interrupt is received, the Dispatch unit stops reading new instructions from the instruction queue and the instructions remaining in the queue are discarded. All instructions whose execution is pending continue to completion. At this point, the processor and all its registers are in a consistent state, and interrupt processing can begin.

UltraSPARC Architecture belongs to the SPARC (Scalable Processor Architecture) family of processors. This architecture is suitable for wide range of microcomputers and supercomputers. UltraSPARC is example of RISC (Reduced Instruction Set Computer).

UltraSPARC architecture:

1. **Memory:**
   Memory consists of 8 bit-bytes. Two consecutive bytes form a halfword, four bytes form a word, eight bytes form a doubleword. UltraSPARC programs operates on **Virtual Address Space** (2^{64} bytes). Virtual Address Space is divided into pages and these pages are stored in the physical memory or on disk.

2. **Registers:**
   UltraSPARC architecture include a large file of registers that have more than 100 general purpose registers. Any procedure can access only 32 registers only. The SPARC hardware uses window into registers file to manage all the operations of different procedures.
   Beside these register files, UltraSPARC also uses Program Counter, code register, and other control registers.

1. **Data Formats:**
   - Integers are stored as 8-, 16-, 32-, or 64-bit Binary numbers.
   - Characters are represented using 8-bit ASCII codes.
   - Floating points are represented using three different formats namely single-percision format, double-percision format, quad-percision format.

2. **Instruction Formats:**
   SPARC architecture use three basic instruction formats. All the instructions are of 32-bit long and first two bits are used to identify which format is being used.
   Format 1- Used for Call instruction.

   Format 2- Used for branch instructions.

   Format 3- Used by all the remaining instructions like register load and store.

1. Where,
2. n=Indirect mode,
3. i=Immediate addressing,
4. x=Index addressing,
5. b=Base addressing,
6. p= Program counter,

   e=Exponential addressing

7. **Addressing Modes:**
   Operands in memory are addressed using one of the following three modes:
8. **Mode** **Target address(TA) calculation**
9. PC-relative TA=(PC) + displacement
10.
11. Register indirect TA=(register) + displacement
12. with displacement
13.
14. Register indirect TA=(register-1) + (register-2)
15. indexed

    PC-relative is used only for branch instructions.

16. **Instruction Set:**
    This architecture have less number of instructions as compared to CISC machines. The only instructins that access memory are load and stores. All other instructions operates on register only. Instruction execution on a SPARC system is pipelined which means while one instruction is executed next one is being fetched from memory and decoded.

17. **Input and Output:**
    Communication between I/O devices and SPARC operation are accomplished through memory. Input and Output can be performed with the regular instruction set of the computer, and no special I/O instructions are needed.