(R20CSE2102) COMPUTER ORGANIZATION & ARCHITECTURE

UNIT - I Digital Computers: Introduction, Block diagram of Digital Computer, Definition of Computer Organization, Computer Design and Computer Architecture. **Register Transfer Language and Micro operations:** Register Transfer language, Register Transfer, Bus and memory transfers, Arithmetic Micro operations, logic micro operations, shift micro operations, Arithmetic logic shift unit. Basic Computer Organization and Design: Instruction codes, Computer Registers Computer instructions, Timing and Control, Instruction cycle, Memory Reference Instructions, Input – Output and Interrupt.

Digital Computers

A Digital computer can be considered as a digital system that performs various computational tasks.

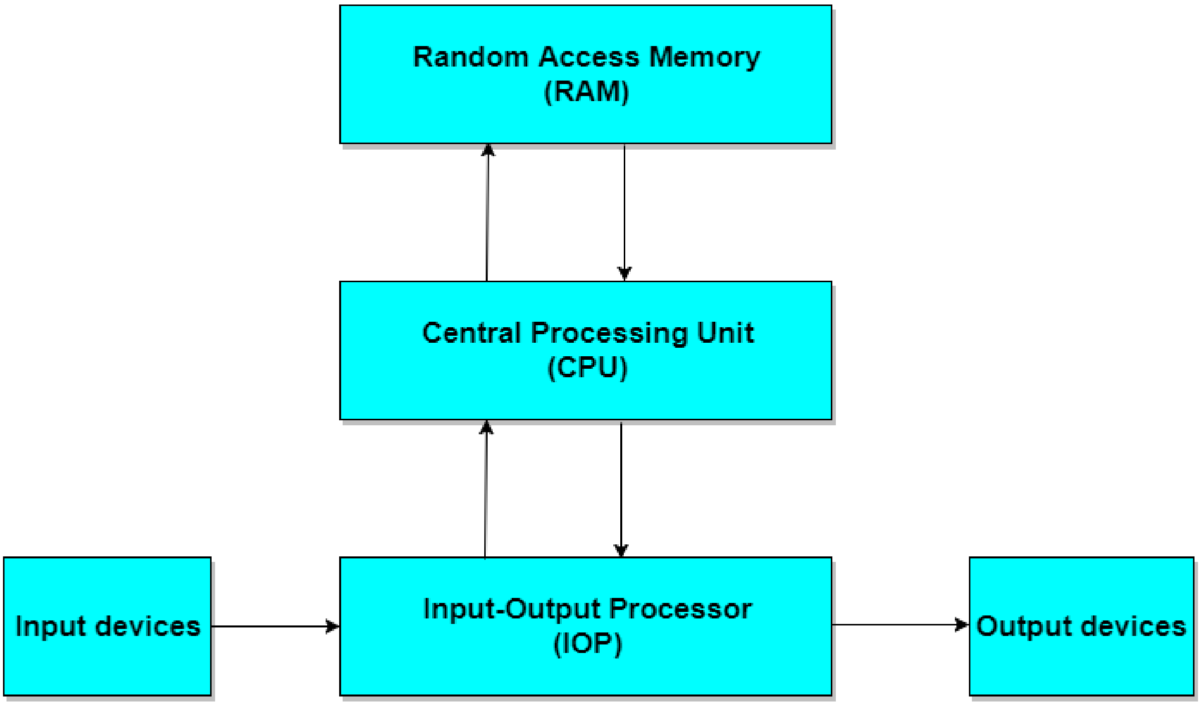
The first electronic digital computer was developed in the late 1940s and was used primarily for numerical computations.

By convention, the digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit.

A computer system is subdivided into two functional entities: Hardware and Software.

The hardware consists of all the electronic components and electromechanical devices that comprise the physical entity of the device.

The software of the computer consists of the instructions and data that the computer manipulates to perform various data-processing tasks.


Random Access Memory (RAM)
Central Processing Unit (CPU)
Input devices
Input-Output Processor (IOP)
Output devices

- The Central Processing Unit (CPU) contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and a control circuit for fetching and executing instructions.
- The memory unit of a digital computer contains storage for instructions and data.

- The Random Access Memory (RAM) for real-time processing of the data.
- The Input-Output devices for generating inputs from the user and displaying the final results to the user.
- The Input-Output devices connected to the computer include the keyboard, mouse, terminals, magnetic disk drives, and other communication devices.

What are Digital Computers?

The **digital computer** is a digital system that performs various computational tasks. The word **digital** implies that the information in the computer is represented by variables that take a limited number of discrete values. These values are processed internally by components that can maintain a limited number of discrete states.

The decimal digits 0, 1, 2, ..., 9, for example, provide 10 discrete values. The first electronic digital computer, developed in the late 1940s, was used primarily for numerical computations and the discrete elements were the digits. From this application the term **digital** computer emerged.

In practice, digital computers function more reliably if only two states are used. Because of the physical restriction of components, and because human logic tends to be binary (i.e. true or false, yes or no statements), digital components that are constrained to take discrete values are further constrained to take only two values and are said to be **binary**.

Digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a **bit**. Information is represented in digital computers in groups of bits. By using various coding techniques, groups of bits can be made to represent not only binary numbers but also other discrete symbols, such as decimal digits or letters of the alphabet.

Digital Computers: Computer Organization

Computer Organization is concerned with the way the hardware components operate and the way they are connected together to form the computer system.

The various components are assumed to be in place and the task is to investigate the organizational structure to verify that the computer parts operate as intended.

Digital Computers: Computer Design

Computer Design is concerned with the hardware design of the computer. Once the computer specifications are formulated, it is the task of the designer to develop hardware for the system.

Computer design is concerned with the determination of what hardware should be used and how the parts should be connected. This aspect of computer hardware is sometimes referred to as **computer implementation**.

Digital Computers: Computer Architecture

Computer Architecture is concerned with the structure and behaviour of the computer as seen by the user.

It includes the information, formats, the instruction set, and techniques for addressing memory. The architectural design of a computer system is concerned with the specifications of the various functional modules, such as processors and memories, and structuring them together into a computer system.

Two basic types of computer architecture are:

1. **von Neumann architecture**
2. **Harvard architecture**

---

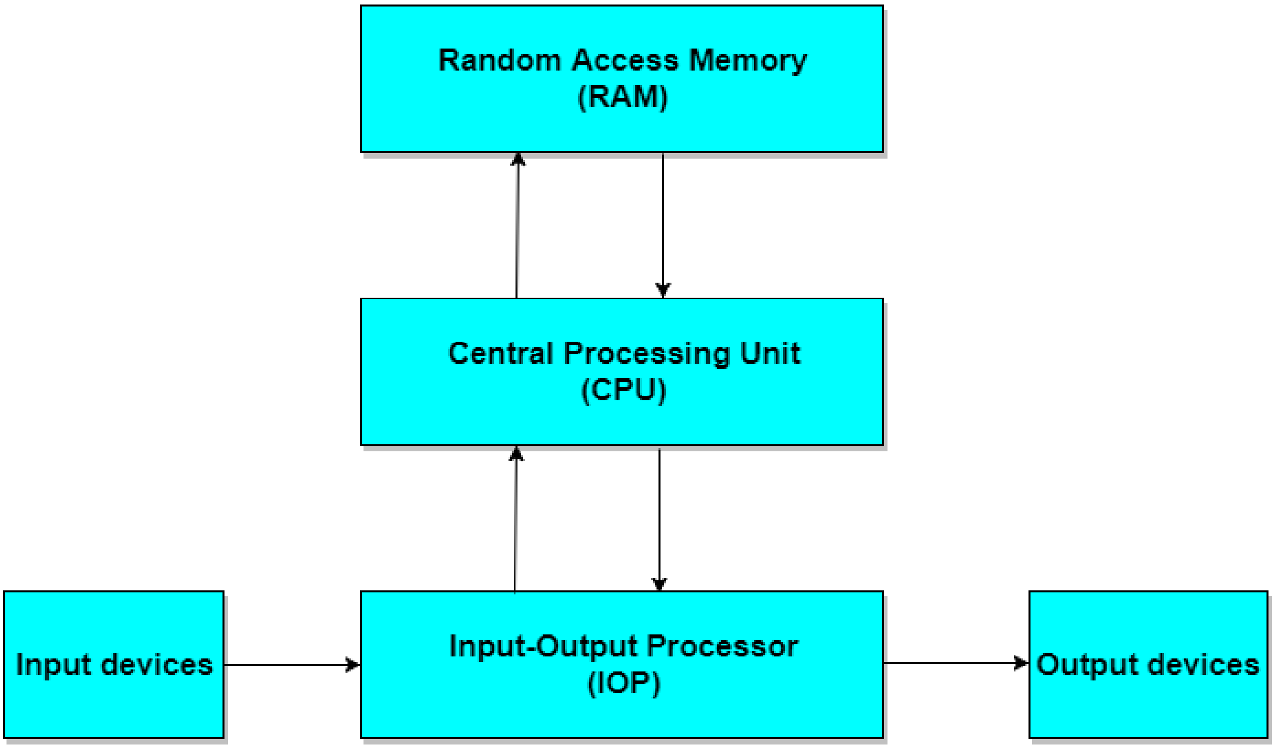
1. von Neumann architecture

The **von Neumann architecture** describes a general framework, or structure, that a computer's hardware, programming, and data should follow. Although other structures for computing have been devised and implemented, the vast majority of computers in use today operate according to the von Neumann architecture.

von Neumann envisioned the structure of a computer system as being composed of the following components:

1. **ALU:** The **Arithmetic-Logic unit** that performs the computer's computational and logical functions.
2. **RAM:** Memory; more specifically, the computer's main, or fast, memory, also known as **Random Access Memory(RAM)**.
3. **Control Unit**: This is a component that directs other components of the computer to perform certain actions, such as directing the fetching of data or instructions from memory to be processed by the ALU; and
4. **Man-machine interfaces;** i.e. input and output devices, such as keyboard for input and display monitor for output.

Block diagram of a Digital Computer



An example of computer architecture base on the von Neumann architecture is the desktop **personal computer**.

---

2. Harvard architecture

The **Harvard architecture** uses physically separate **storage** and **signal pathways** for their instructions and data. The term originated from the **Harvard Mark I** and the data in relay latches (23- digits wide).

In a computer with Harvard architecture, the CPU can read both an instruction and data from memory at the same time, leading to double the memory bandwidth.

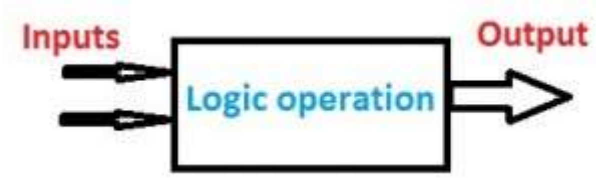
Microcontroller (single-chip microcomputer)-based computer systems and **DSP**(Digital Signal Processor)-based computer systems are examples of Harvard architecture.

BASIC LOGIC GATES WITH TRUTH TABLES

Nowadays, computers have become an integral part of life as they perform many tasks and operations in quite a short span of time. One of the most important functions of the CPU in a computer is to perform logical operations by utilizing hardware like Integrated Circuits software technologies & electronic circuits,. But, how this hardware and software perform such operations is a mysterious puzzle. In order to have a better understanding of such a complex issue, we must have to acquaint ourselves with the term Boolean Logic, developed by George Boole. For a simple operation, computers utilize binary digits rather than digital digits. All the operations are carried out by the Basic Logic gates. This article discusses an overview of what are **basic logic gates** in digital electronics and their working.

What are Basic Logic Gates?

A logic gate is a basic building block of a digital circuit that has two inputs and one output. The relationship between the i/p and the o/p is based on a certain logic. These gates are implemented using electronic switches like transistors, diodes. But, in practice, basic logic gates are built using (complementary metal-oxide-semiconductor) CMOS technology, Field-effect transistor FETS, and MOSFET(Metal Oxide Semiconductor FET)s. Logic gates are used in microprocessors, microcontrollers, embedded system applications, and in electronic and electrical project circuits. The basic logic gates are categorized into seven: AND, OR, XOR, NAND, NOR, XNOR, and NOT. These logic gates with their logic gate symbols and truth tables are explained below.



Basic Logic Gates Operation

What are the 7 Basic Logic Gates?

The basic logic gates are classified into seven types: AND gate, OR gate, XOR gate, NAND gate, NOR gate, XNOR gate, and NOT gate. The truth table is used to show the logic gate function. All the logic gates have two inputs except the NOT gate, which has only one input.

When drawing a truth table, the binary values 0 and 1 are used. Every possible combination depends on the number of inputs. If you don’t know about the logic gates and their truth tables and need guidance on them, please go through the following infographic that gives an overview of logic gates with their symbols and truth tables.

Why we use Basic Logic Gates?

The basic logic gates are used to perform fundamental logical functions. These are the basic building blocks in the digital ICs (integrated circuits). Most of the logic gates use two binary inputs and generates a single output like 1 or 0. In some electronic circuits, few logic gates are used whereas in some other circuits, microprocessors include millions of logic gates.

The implementation of Logic gates can be done through diodes, transistors, relays, molecules, and optics otherwise different mechanical elements. Because of this reason, basic logic gates are used like electronic circuits.

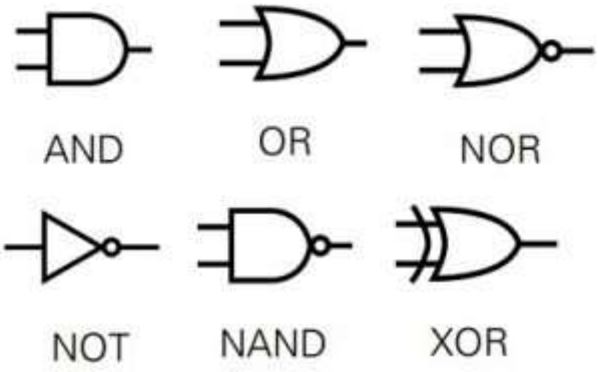
Binary & Decimal

Before talking about the truth tables of logic gates, it is essential to know the background of binary & decimal numbers. We all know the decimal numbers which we utilize in everyday calculations like 0 to 9. This kind of number system includes the base-10. In the same way, binary numbers like 0 and 1 can be utilized to signify decimal numbers wherever the base of the binary numbers is 2.

The significance of using binary numbers here is to signify the switching position otherwise voltage position of a digital component. Here 1 represents the High signal or high voltage whereas “0” specifies low voltage or low signal. Therefore, Boolean algebra was started. After that, each logic gate is discussed separately this contains the logic of the gate, truth table, and its typical symbol.

Types of Logic Gates:

The different types of logic gates and symbols with truth tables are discussed below.

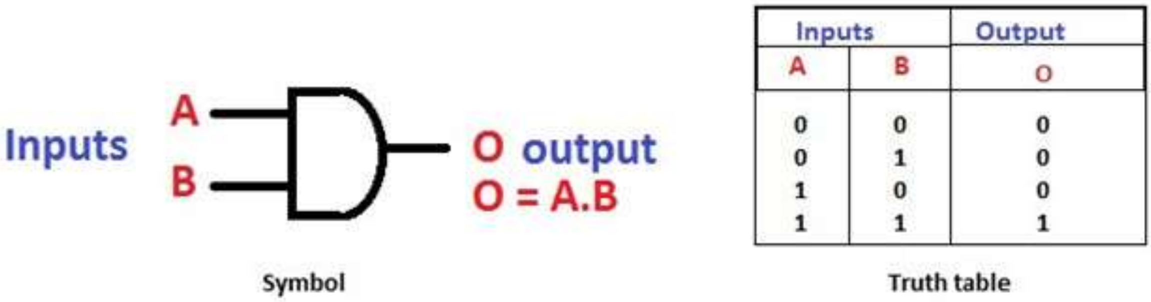


Basic Logic Gates

AND Gate

The AND gate is a digital logic gate with 'n' i/ps one o/p, which performs logical conjunction based on the combinations of its inputs. The output of this gate is true only when all the inputs are true. When one or more inputs of the AND gate's i/ps are false, then only the output of the AND gate is false. The symbol and truth table of an AND gate with two inputs is shown below.

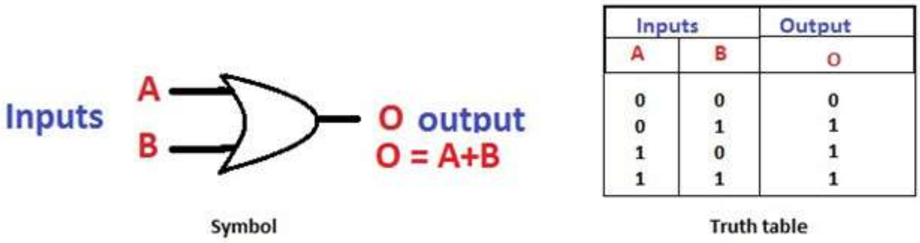
AND Gate and its Truth Table



Inputs		Output
A	B	O
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate

The OR gate is a digital logic gate with 'n' i/ps and one o/p, that performs logical conjunction based on the combinations of its inputs. The output of the OR gate is true only when one or more inputs are true. If all the i/ps of the gate are false, then only the output of the OR gate is false. The symbol and truth table of an OR gate with two inputs is shown below.

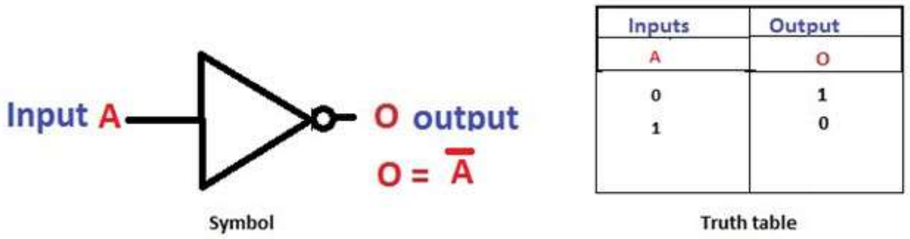


Inputs		Output
A	B	O
0	0	0
0	1	1
1	0	1
1	1	1

OR Gate and its Truth Table

NOT Gate

The NOT gate is a digital logic gate with one input and one output that operates an inverter operation of the input. The output of the NOT gate is the reverse of the input. When the input of the NOT gate is true then the output will be false and vice versa. The symbol and truth table of a NOT gate with one input is shown below. By using this gate, we can implement NOR and NAND gates

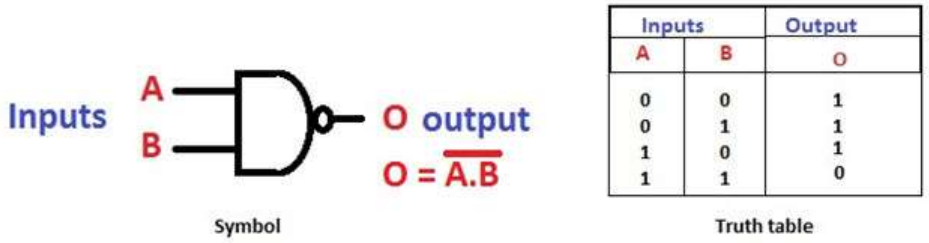


Inputs	Output
A	O
0	1
1	0

NOT Gate and Its Truth Table

NAND Gate

The NAND gate is a digital logic gate with ‘n’ i/ps and one o/p, that performs the operation of the AND gate followed by the operation of the NOT gate.NAND gate is designed by combining the AND and NOT gates. If the input of the NAND gate high, then the output of the gate will be low.The symbol and truth table of the NAND gate with two inputs is shown below.

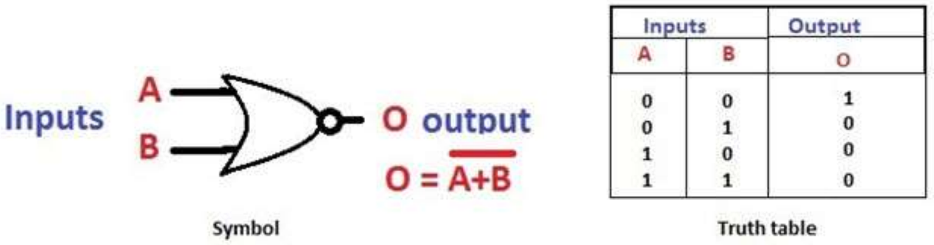


Inputs		Output
A	B	O
0	0	1
0	1	1
1	0	1
1	1	0

NAND Gate and its Truth Table

NOR Gate

The NOR gate is a digital logic gate with n inputs and one output, that performs the operation of the OR gate followed by the NOT gate. NOR gate is designed by combining the OR and NOT gate. When any one of the i/ps of the NOR gate is true, then the output of the NOR gate will be false. The symbol and truth table of the NOR gate with the truth table is shown below.

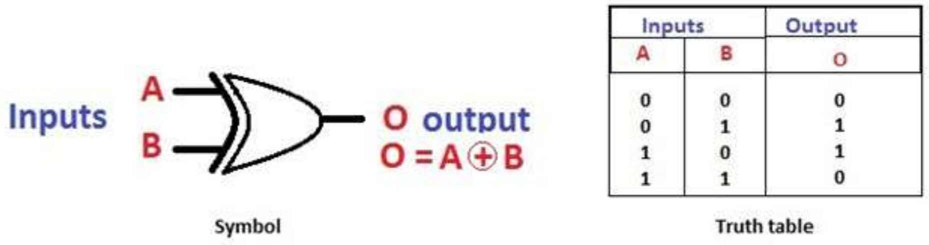


Inputs		Output
A	B	O
0	0	1
0	1	0
1	0	0
1	1	0

NOR Gate and Its Truth Table

Exclusive-OR Gate

The Exclusive-OR gate is a digital logic gate with two inputs and one output. The short form of this gate is Ex-OR. It performs based on the operation of the OR gate. . If any one of the inputs of this gate is high, then the output of the EX-OR gate will be high. The symbol and truth table of the EX-OR are shown below.



Symbol

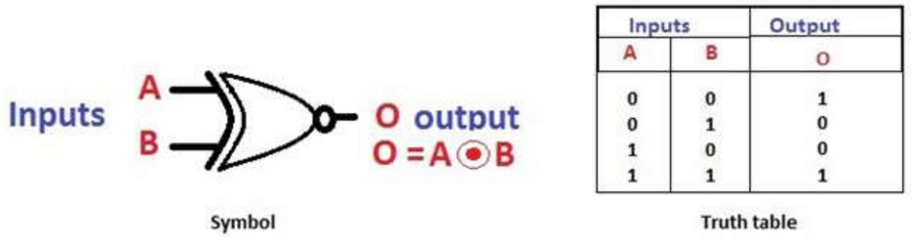
Inputs		Output
A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

Truth table

EX-OR gate and Its Truth Table

Exclusive-NOR Gate

The Exclusive-NOR gate is a digital logic gate with two inputs and one output. The short form of this gate is Ex-NOR. It performs based on the operation of the NOR gate. When both the inputs of this gate are high, then the output of the EX-NOR gate will be high. But, if any one of the inputs is high (but not both), then the output will be low. The symbol and truth table of the EX-NOR are shown below.



Symbol

Inputs		Output
A	B	O
0	0	1
0	1	0
1	0	0
1	1	1

Truth table

EX-NOR Gate and Its Truth Table

What is the Easiest Way to Learn Logic Gates?

The easiest way to learn the function of basic logic gates is explained below.

- ✓ For AND Gate – If both the inputs are high then the output is also high
- ✓ For OR Gate – If a minimum of one input is high then the output is High
- ✓ For XOR Gate – If the minimum one input is high then only the output is high
- ✓ NAND Gate – If the minimum one input is low then the output is high
- ✓ NOR Gate – If both the inputs are low then the output is high.

Register Transfer Language

Computers are the electronic devices which have several sets of digital hardware which are inter connected to exchange data. Digital hardware comprises of VLSI Chips which are used for both data processing as well as data storage facilities. Each of the Digital hardware is defined as the digital module. Digital modules are considered as the devices which store the data using the registers available in them and process the data basing on micro operational codes. These Micro operational codes will be used to coordinate the data transfer as well as for data processing. Register transfer can be defined as the process of moving the data between the registers which are controlled by the micro operations.

Therefore in order to define the micro operations we normally use the descriptive language. We explain each micro operation in detail. This type of writing is better for explanatory purpose, but to write a program this kind of writing is not sufficient.

Therefore like in other computer programming languages, we use the symbols for denoting an operation. Like to load some data into the registers we use “LOAD 5 → R0” This will be a self explanatory mnemonics. This type of program writing is called as Register Transfer Language.

As the register transfer language deals with the internals of the hardware we should be first knowing what is the internal hardware organization.

The internal hardware organization of a digital computer is best defined by specifying:

- The set of registers it contains and their function
- The sequence of micro-operations performed on the binary information stored in the registers.
- The control that initiates the sequence of micro operations.

The contents of Register transfer language:

Register Transfer language syntax will contain the micro operations along with the source and target registers on which the data modification will be performed with a set of control signals.

For Example: P: LOAD 5 → R0. The command given here is used to load a data 5 to the register R0.

LOAD → Micro Operation
5 → Data (Source Data)
R0 → General Purpose Register (Target Register)
P → Control Signal initiated to run the micro operation.
“ → ” → Data transfer from source to destination.

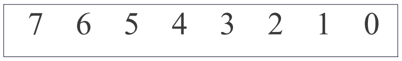
Register Transfer

Before we see how the data is transferred from one register to another register, we will see the construction of a register and its various diagrams. Registers are the fast data storage devices which are constructed using VLSI Chips with the technology of FLIP-FLOPs. In a simple 8 bit register, there will be 8 flip-flops which will be holding the data , the design of the flip-flops are based on the manufactures specification. Several registers are present in the CPU. Such as MAR → used to store the address of the particular data. MDR → which is used to store the data of a particular address stored in MAR. PC → is used to point the next instruction to be executed.

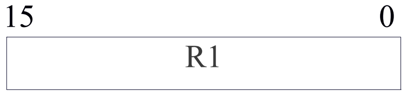
The following Diagrams are the different representations of the Register.

Register with a the name inside: Register R1

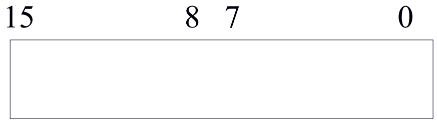
Register with the no of bits showing inside:



Register with Name and no of bits showing:



Register with bits divided into parts:



Next we will see how the Register transfer actually works. We will take the example syntax and then see how the register transfer takes place.

Ex: if(P==1) then (R2 ← R1)

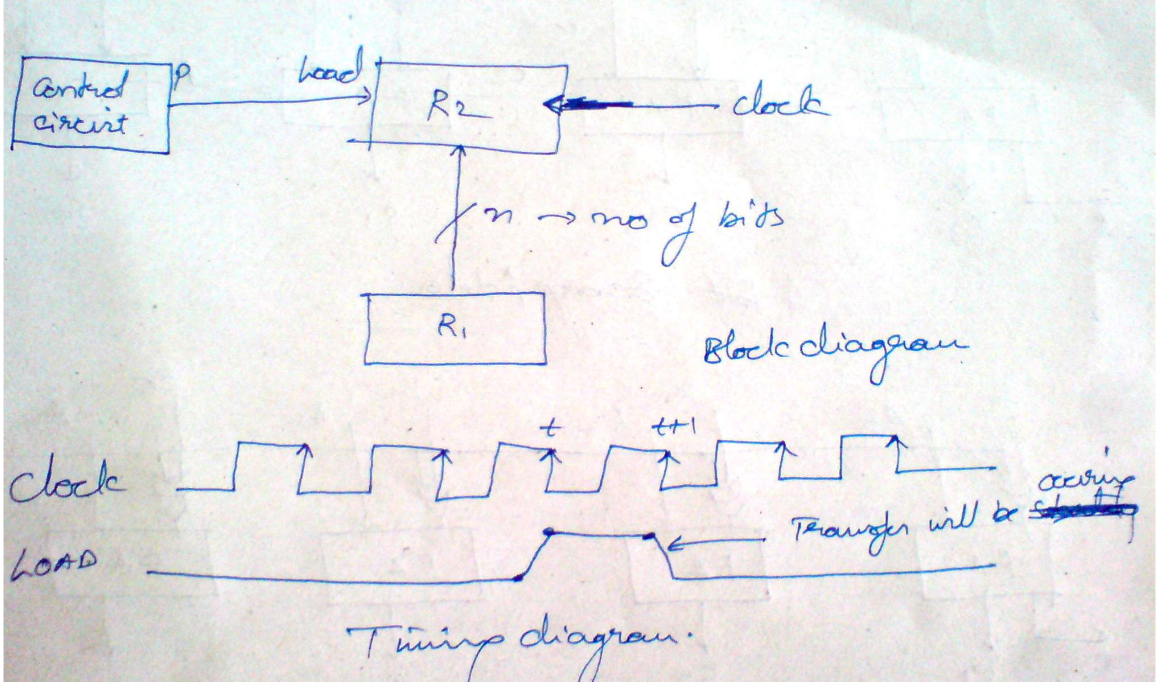
Here "P" is the control signal generated by the control unit which should be true if the data transfer from R1 to R2 should take place.

Steps of execution:

- First n outputs of R1 is connected with n inputs of R2.
- Register R2 has a Load Input which is activated when the control signal P==1;

- Here the control variable is synchronized with the clock. The type of the register here we are considering is the Positive edge trigger flip flop. Therefore the data storage and the transfer process will begin only in the positive edge of the clock.
- After acquiring the positive edge of the clock, the control signal P is active i.e.., P=1 and then the Register R2 finds the Load input active and the data inputs from R1 are loaded into R2 in parallel.
- After the data transfer P will go back to 0, because if P is not going back to 0, it will always be 1 and at every positive edge of the clock, the data transfer from R1 to R2 will be taking place.

The complete scenario is depicted in the diagram below:



Basic symbols for Register Transfers:

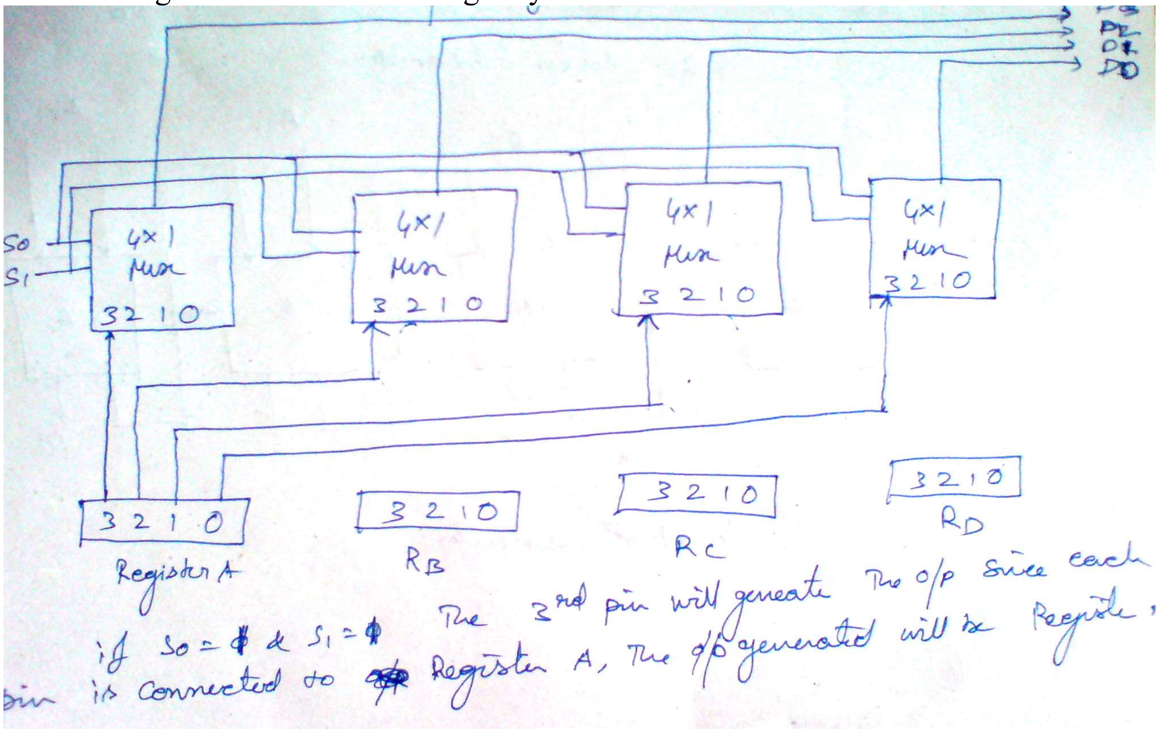
Symbols	Description	Examples
Letters	Denotes a Register	MAR, PC, IR, MDR, R1, R2 etc..,
Parentheses ( )	Denotes a part of the register	R2(0 – 7)
Arrow ← or →	Denotes transfer of information	R2 ← R1
Comma,	Separates two Micro-operations	R2 ← R1, R3 ← R1

Bus and Memory Transfers:

In a digital computer all the devices are inter connected using a bus. All the devices will be using these bus paths as the data paths through which the data will be exchanged. So if we are constructing a system which has several devices(Registers) and each register has to be having its own path means, there will be a lot of bus(set of wires). This is not a good design, because providing a communication path for each and every device means the time taken to exchange information will be long if the bus path is very long or else the bus management will be very difficult. To avoid this kind of problems we use "Common Bus Structure". In this all the bus available will be shared among all the devices. Therefore only two devices can communicate at a time using the common bus structure in-order to avoid data inconsistency.

To solve this problem we have discussed the method of buffers. But this solution also solves the problem only to a certain point. Consider the following problem. You have a system which has 16 bit bus lines, through which you want to send 16 signals of 16 bit data. That is total of 256 bit of data has to be sent from 16 registers. So in this type of problem, we use multiplexers.

Multiplexers are the one which convert the no of input data signals into a small no of output signals. Ex: 16x1 multiplexer can convert 16 signals into one signal basing on the selection modes. The diagram below will represent how 4x1 multiplexers are used to transfer 4 signals of 4 bits each using only 4 bit bus.

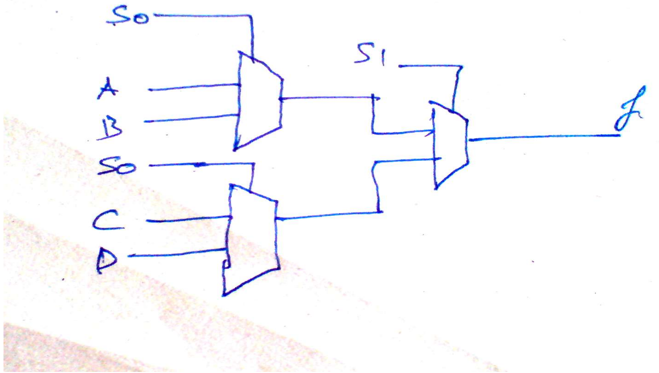


if $S_0 = \phi$ & $S_1 = \phi$ The 3rd pin will generate the o/p since each pin is connected to Register A, The o/p generated will be Register,

The description below explains how the 4x1 multiplexer is used in the above scenario. The above setup has 2 select signals named "S0" and "S1", along with 4 sets of registers which are 4 bits long(stores 4 bits of data). In this diagram we can see that the first multiplexer has inputs from the first bits of all the respective registers. Like wise the second multiplexer will have all the inputs from the second bit of all the registers. Likewise third and fourth multiplexers have the third bit and fourth bit input from the respective registers.

So in this scenario two select signals will generate four combinations. Such as (0,0),(0,1),(1,1),(1,0). so for every select signals combination each respective bit of the multiplexer will be activated. For example if we are considering the select signal (0,0) notation is active then, let us assume that the third bit of each multiplexer is in on state which will then generate the register D at the output. Thus the data in the Register D will occupy the bus when the select signals are in (0,0) combination. Thus the total amount of bus will be shared among all the available registers.

The general symbol and the structure of 4X1 multiplexer is given below:



The truth table representing the output of the above 4x1 multiplexer circuit:

Signal S0	Signal S1	Output generated
0	0	A
0	1	B
1	0	C
1	1	D

So whenever we are representing that a data transfer is taking place from R1 → R2, this internally means that R1 → BUS and BUS → R2. If we know that the bus is definitely present in the system then we can skip the BUS variable and we can just write R1 → R2.

This is how a multiple data paths are achieved by using a single bus, by implementing Multiplexers.

Three State Bus Buffers:

Without using multiplexers also we can construct a bus system. These can be achieved using three state gates.

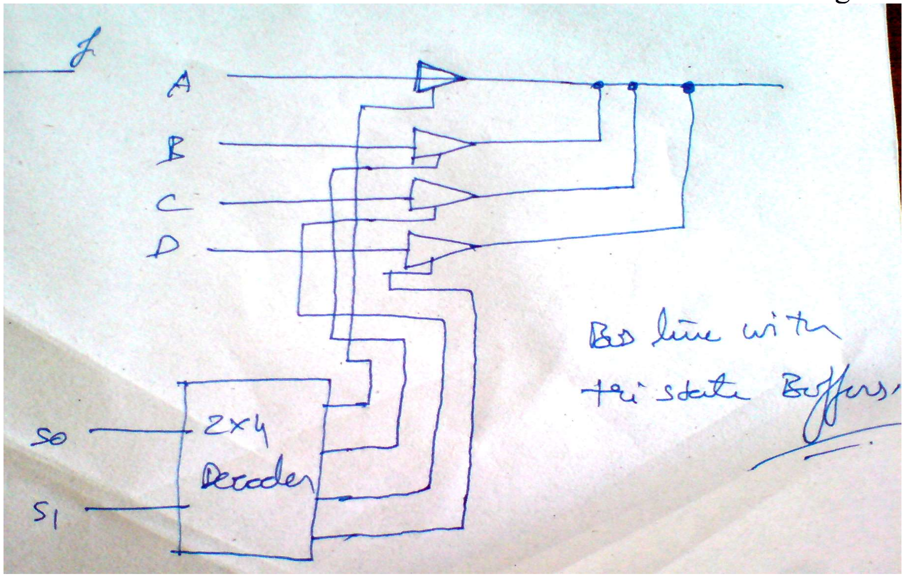
The three state gate can be defined as the device which have a “Data input line”, “Data output line” and “A control signal Line”. Therefore the behavior of the Three state gate will be as follows:

- If the control signal is in high state then the device either produced 1 or 0 basing on the input provided.

- If the control signal is in low state(== 0) then the device is in high impedance state.

The three states are “1”, “0”, “High Impedance state”.

So how can we use this to create a bus structure is defined below in the diagram.



The below diagram uses a 2 to 4 Decoder. The decoder is a device which will produce the output signals basing on the input signals. It naturally decodes the data using select signals as well as Enable. The truth table of the 2 to 4 decoder is given below:

Select Signal S0	Select Signal S1	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	1
0	1	0	0	1	0
1	1	0	1	0	0
1	0	1	0	0	0

So as per the truth table we can see that when the select signals S0 and S1 both are equal to 0, Bit 0 is having the output of 1. Therefore the Three state gate 1 will be active and the signal A0 will be transferred to the bus line for bit 0. Like wise when S0 and S1 both equal to 1, Bit 3 is having the output of 1. Therefore the Three state gate 4 will be active and the signal D0 will be transferred to the bus line for bit 0. This is how the parallel data transfer will be used. This is the process of implementing the three state gates also called as three state bus buffers. These are called as Three state bus buffers because when the control signal is equal to 1, simple the data input will be sent as the data output, nothing but it stores and sends the output signal. Since a temporary data storing is occurring it is called as Buffers. So in this type of scenario the only measure we should take is that if one control signal is in on position, then definitely all the remaining buffer's control signals must be in off position otherwise the buffer should be in high impedance state. This scenario can be achieved using a decoder circuit.

Note: Impedance can be defined as the opposition to the flow of the Alternate Current.

Memory Transfer:

Memory Data Transfers are of two types. Read operation and Write operation. So a Read Operation can be defined as the transfer of data from the memory to the outside world. The data that is being transferred will be in the form of word (set of bits). The memory word will be represented by Capital Letter M.

For example: Read: DR ← M[AR]. This statement is used to transfer the data word that is present at the particular AR → Address Register to the destination register DR → Data Register.

A Write operation can be defined as the transfer of data from the outside world to the Memory. This operation also will be performed as the transfer of word (Set of Bits).
For Example: Write: M[AR] ← R1. In this statement the data that is present in R1 will be transferred to the Address that is represented by the AR → Address Register. M[AR] represents the data that is to be written a the particular memory location.

Arithmetic Micro-operations:

Micro operation can be defined as the operation that is being performed on the binary data that is present inside the registers. The most commonly described micro operations are of four types.
They are:

- Register transfer micro operations:
  - Which performs all the data transfer from one register to another register.
- Arithmetic micro-operations performed on the binary data stored in the registers.
  - Which performs all the arithmetic operations such as "Addition", "Subtraction", "Multiplication", "Division".
- Logic Micro-operations perform bit manipulation operations on binary data stored in the registers.
  - Such as Logical AND,OR,NOT, Comparisons etc..,
- Shift Micro-operations which perform data shifting as per bit wise.
  - Such as Left Shift and Right Shift operations.

In the previous scenarios we saw that the Register transfers are used to transfer the data between one register to another register with out modifying the data. But as computer systems are not only designed to copy the data, but also modify the data, this generally incorporates the following things:

- Registers to hold the data
- Arithmetic circuits which perform the respective operations such as Add, Subtraction, etc..,

The following syntax will show how to perform an Addition micro operation

R3 ← R1 + R2

So in the syntax above the contents of the register R1 and R2 are added using some device and then finally the result that is acquired from that addition will be stored in the register R3.

If a subtract command has to be performed, it will use the technique of two's complement as discussed in the before chapters:

R3 ← R1 + R2`+1

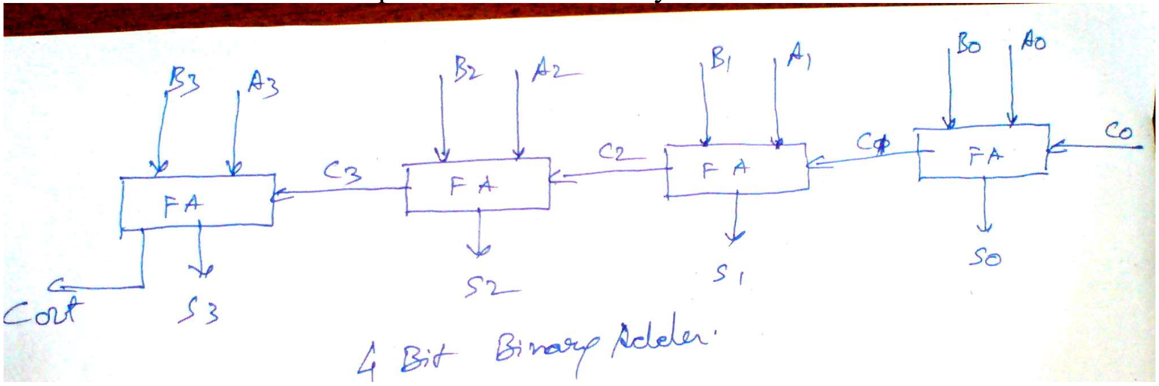
This syntax will achieve the subtraction of register R1 to R2 using two's complement technique.

The following table indicates different micro operations performed:

Symbolic Designation	Description
R3 ← R1+R2	Contents of R1 plus R2 transferred to R3
R3 ← R1-R2	Contents of R1 minus R2 transferred to R3
R2 ← R2'	Complement the contents of R2(1`s complement)
R2 ← R2' + 1	2's complement the contents of R2
R3 ← R1 + R2'+1	Subtraction operation achieved using the 2's complement addition of the second number
R1 ← R1+1	Increment by 1
R1 ← R1-1	Decrement by 1

Binary Adder:

In order to implement binary adder we need to consider the implementation of full adder circuit. Binary adder circuit will be constructed basing on the cascaded connections of Full adder circuits. The implementation of binary adder is shown below.

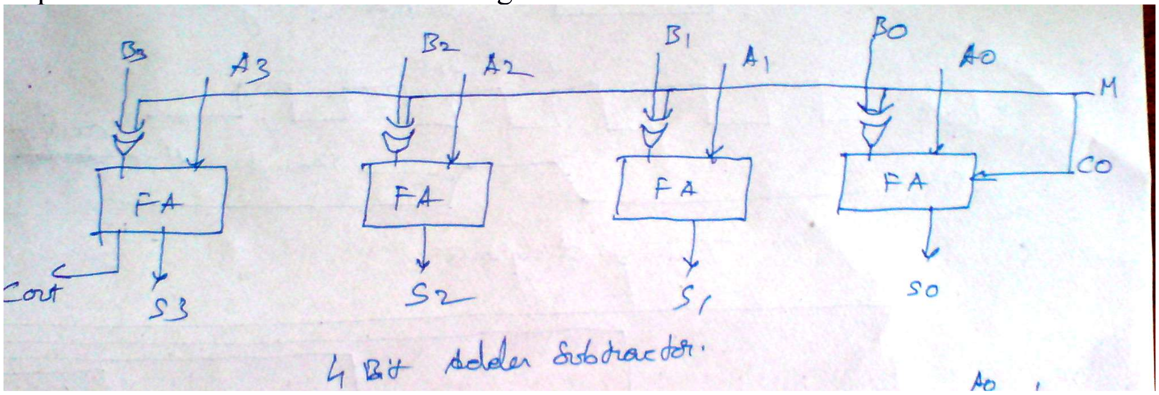


4 Bit Binary Adder.

In the figure above, you can see that initially the first full adder is having a carry signal C0, after that each of the bits of the data to be added will be feed to the circuit bit by bit. In this two bits are added and if any carry is generated then it will be forwarded to the next Full adder. Therefore a “n-bit” adder will be having “n” Full adder circuits.

Binary Adder – Subtracter:

This is a special kind of binary adder circuit implementation in which both addition and subtraction can be performed basing on the control signal. The circuit implementation will be in the following form.



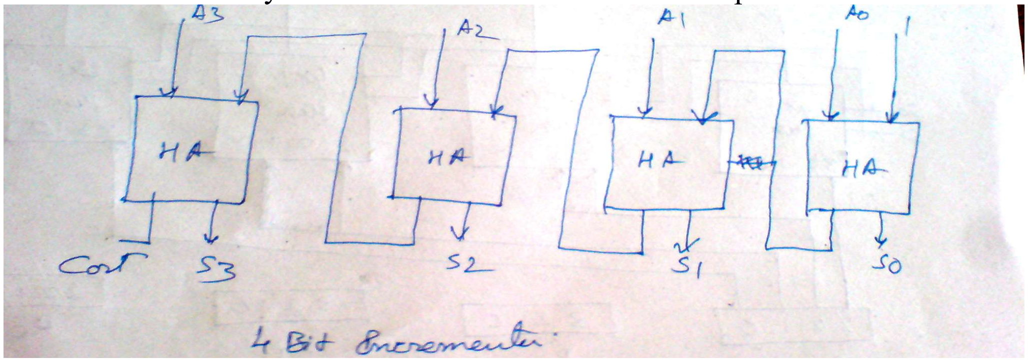
4 Bit Adder Subtractor.

In the circuit above if the control signal M=0 then the circuit becomes an Adder Circuit and if the signal M=1, then the circuit will compute subtraction using 2's complement addition.

If the value of M=0, then in the circuit xor gates will generate the output as the same input. i.e.., if B0 signal is transferred through the xor gate then if the other input is 0, then the output generated by the xor will be B0 signal only. Like wise if B1 signal is passing through the xor signal then the output generated will be B1 only. Since M=0, C0 which is the first carry will be 0, then the whole circuit will be converted as Binary Adder. If M=1 then the output of xor will generate the 1's complement, since M=1 the carry C0 also equal to 1. If we are adding 1 to 1's complement no, then that will become 2's complement. Then if we add 2's complement to the original no, it will be the subtraction operation. Like this we can accomplish both addition and subtraction operation basing on the value of M.

Binary Incrementer:

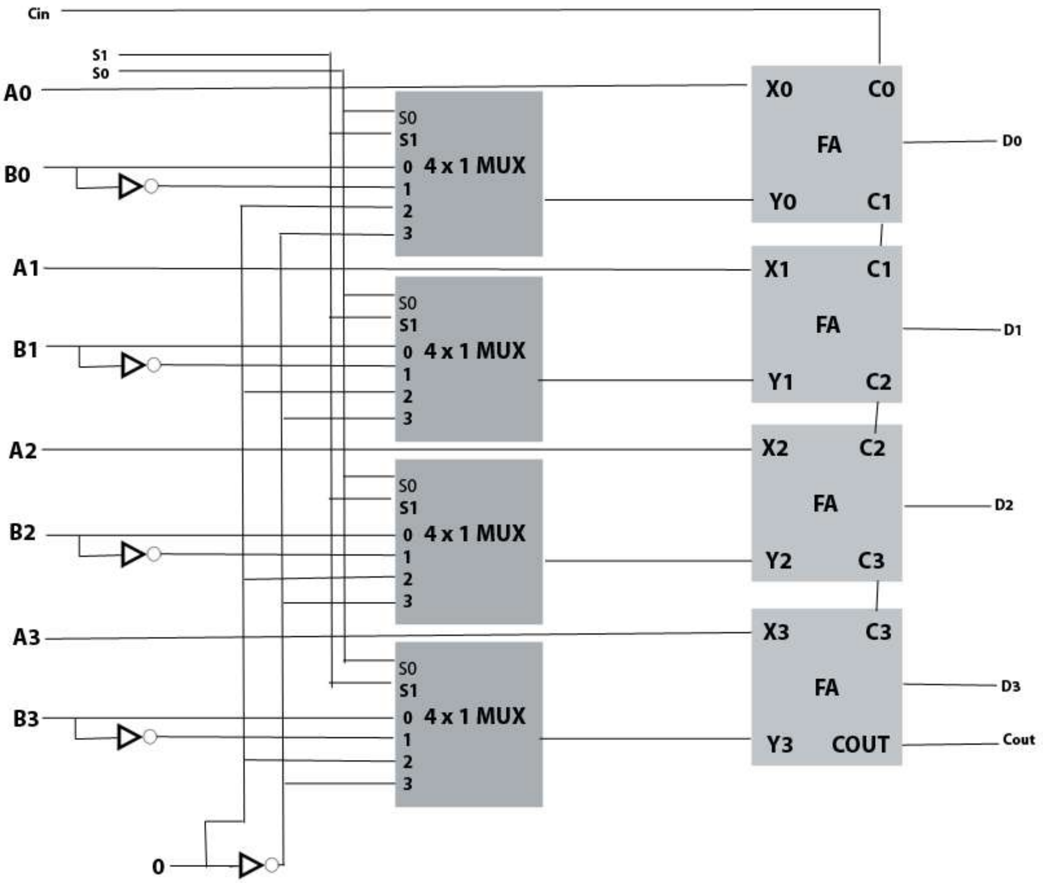
The purpose of the incrementer circuit is to add 1 to the original data. The 4 bit combinational circuit incrementer is shown in the following figure. In this the first half adder will be having one input as 1, then automatically the bit will be added to the current position and then the carry will be incremented to the next bit position.



4 Bit Incrementer.

Arithmetic circuit:

The arithmetic circuit is a special type of circuit in which the above discussed circuits, I.e.., Binary adder, Binary Subtracter and Binary incrementer will be implemented in a single circuit using a combination of the select signals and other signal combinations. By controlling the data inputs to the arithmetic circuit, we can control the output being generated. The circuit diagram of the arithmetic circuit is given below.



Working principle of the Arithmetic Circuit:

The arithmetic circuit will be having the following input signals. Cin signal is the default carry signal inputted to the circuit. S0, S1 signals are the select signals which will be forming the different combinations of the select signals for the multiplexers. The input of the multiplexers will be the select signals S0, S1, Bbit and B'bit. After these inputs are fed to the multiplexer circuits basing on the select signals the output will be generated and these outputs will be fed to the Full Adder circuits, after which they will be added to the Abit Signals. By controlling these input signals we can generate different combinations of the output which will implement different arithmetic operations mentioned in the table below.

Arithmetic Circuit Function Table

Select	Input Y	Output D=A+Y+Cin	Micro operation		
S1	S0	Cin			
0	0	0	B	D=A+B	Add
0	0	1	B	D=A+B+1	Add with carry
0	1	0	B`	D=A+B`	Subtract with borrow
0	1	1	B`	D=A+B`+1	Subtract
1	0	0	0	D=A	Transfer A
1	0	1	0	D=A+1	Increment A
1	1	0	1	D=A-1	Decrement A
1	1	1	1	D=A	Transfer A

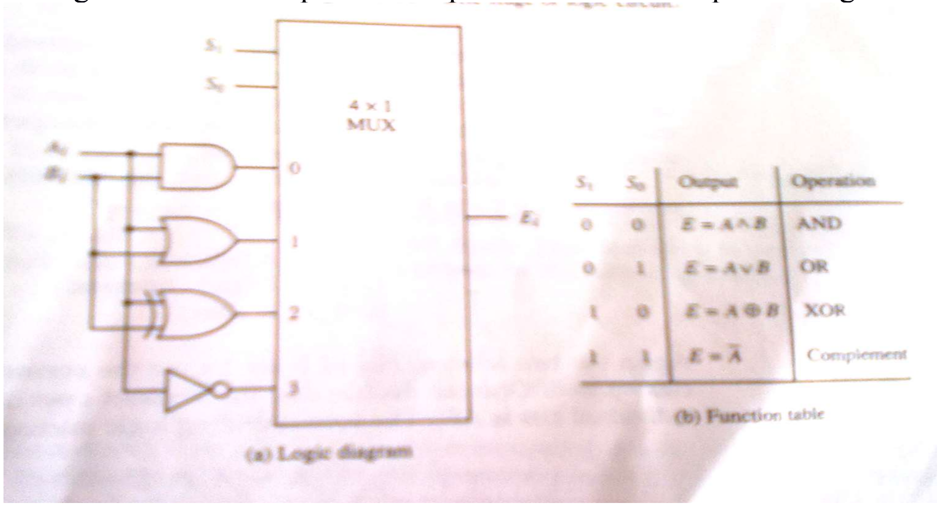
By using this arithmetic circuit we can have all the arithmetic operations implemented in only one circuit.

Logic Microoperations:

x	y	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		Clear	AND	A^B`	A	A`^B	B	A xor B	OR	Nor	XNOR	B`	Av B`	A`	A`v B	NAND	Set 1

Hardware Implementation:

The hardware implementation of the above discussed logic operations are based on the primitive gates such as AND, OR, XOR, NOT gates. Using the combinations or by using select signals with the help of a multiplexer we will be implementing the circuit.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

Selective set: The selective set operation sets to 1 the bits in register A where there are corresponding 1's in register B. it does not affect bit positions that have 0's in B. the following numerical example clarifies this operation:

1010 A Before
1100 B (Logic operand)

1110 A After

Selective complement: The selective complement operation complements bits in A where there are corresponding 1's in B. It does not affect bit positions that have 0's in B. For example:

1010 A Before
1100 B (Logic operand)

0110 A After

Selective clear: The selective clear operation clears to 0 the bits in A only where there are corresponding 1's in B. For example:

1010 A Before
1100 B (Logic operand)

0010 A After

Masking: The mask operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding 0's in B. The mask operation is an AND micro operation as seen from the following numerical example:

1010 A Before
1100 B (Logic operand)

```
------
1000    A After
```

Inserting: The insert operation inserts a new value into a group of bits. The is done by first masking the bits and then ORing them with the required value. For example, suppose that an A register contains eight bits, 01101010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

```
0110 1010      A Before
0000 1111      B (Logic operand)
------
0000 1010      A After
```

and then insert the new value:

```
0110 1010      A Before
1001 0000      B (Logic operand)
------
1001 1010      A After
```

Shift Micro operations:

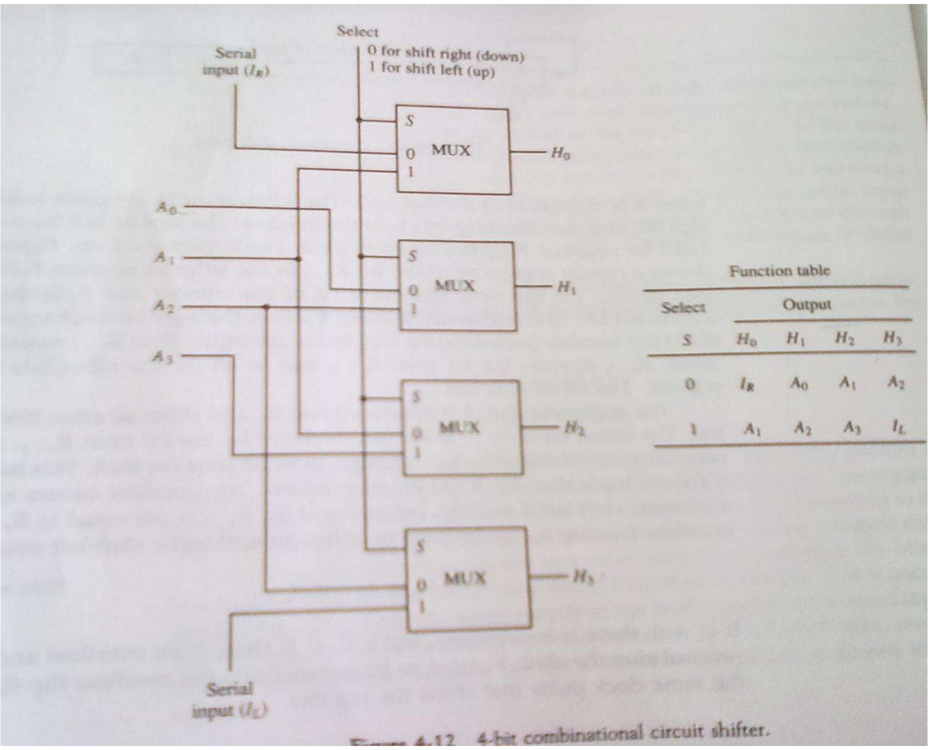
Logical shift: A logical shift is one that transfers 0 through the serial input. We will adopt the symbols shl and shr for logical shift-left and shift-right micro operations.

R1 ← shl R1
R2 ← shr R2

The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

Circular Shift: The circular shift(also known as a rotate operation) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. We will use the symbols cil and cir for the circular shift left and right, respectively. The symbolic notation for the shift micro operations is shown in the following figure;

Symbolic Designation	Description
R ← shl R	Shift left register R
R ← Shr R	Shift right register R
R ← cil R	Circular Shift left Register R
R ← cil R	Circular Shift Right Register R
R ← ashl R	Arithmetic shift left R
R ← ashr R	Arithmetic Shift Right R



Function table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Figure 4-12 4-bit combinational circuit shifter.

Arithmetic Logic Shift Unit:

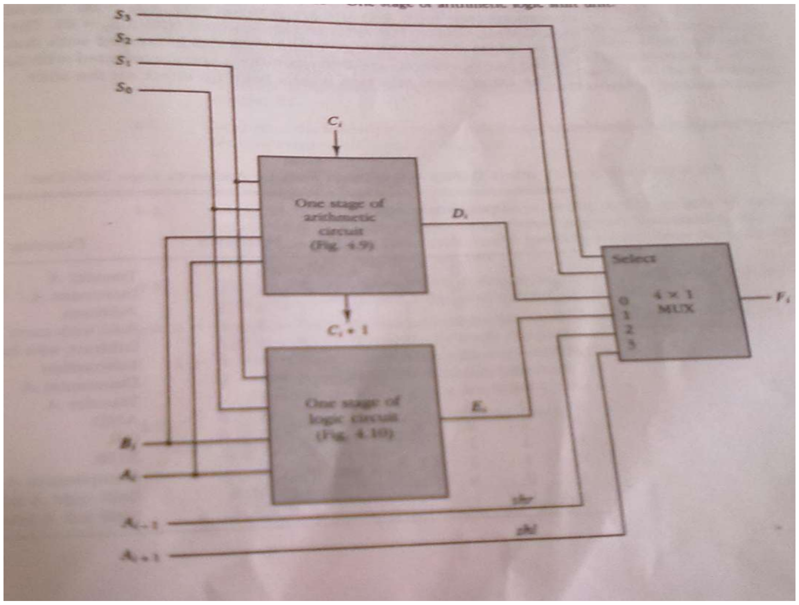
In a computer system, if each and every register performs its own operation, then the designing of the computer system will be very much harder and also many devices are needed to create the computer. Normally all the devices such as arithmetic circuit, logic circuit and shift circuit will be combined together to form a single unit called as ALU stands for Arithmetic Logic Unit. In the modern computers the shift unit is also combined with the main ALU unit. The below diagram represents the ALU Circuit implementation in a single circuit.

It is controlled by a set of 4 Select signals named as S0, S1, S2, S3. Along with the data lines A and B and a carry in signal C_{in}. This has three parts,

- One stage of Arithmetic circuit
- One stage of Logic Circuit
- 4X1 Mux.

By using the combinations of the select signals the ALU will perform all the different operations that are listed below in the table.

Operation Select						
S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	F=A	Transfer A
0	0	0	0	1	F=A+1	Increment A
0	0	0	1	0	F= A+B	Addition
0	0	0	1	1	F= A+B+1	Add with Carry
0	0	1	0	0	F=A+B`	Subtract with Borrow
0	0	1	0	1	F= A+B`+1	Subtraction
0	0	1	1	0	F= A -1	Decrement A
0	0	1	1	1	F= A	Transfer A
0	1	0	0	X	F = A and B	AND
0	1	0	1	X	F = A v B	OR
0	1	1	0	X	F= A xor B	XOR
0	1	1	1	X	F= A`	Complement A
1	0	X	X	X	F= shr A	Shift right A into F
1	1	X	X	X	F= shl A	Shift left A into F



Instruction Code

An instruction code is a group of bits that instruct the computer to perform a specific operation.

Operation Code

The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement. The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer. The operation code must consist of at least n bits for a given 2^n (or less) distinct operations.

Accumulator (AC)

Computers that have a single-processor register usually assign to it the name accumulator (AC) accumulator and label it AC. The operation is performed with the memory operand and the content of AC.

Stored Program Organization

- The simplest way to organize a computer is to have one processor register and an instruction code format with two parts.
- The first part specifies the operation to be performed and the second specifies an address.
- The memory address tells the control where to find an operand in memory.
- This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.
- The following figure 2.1 shows this type of organization.

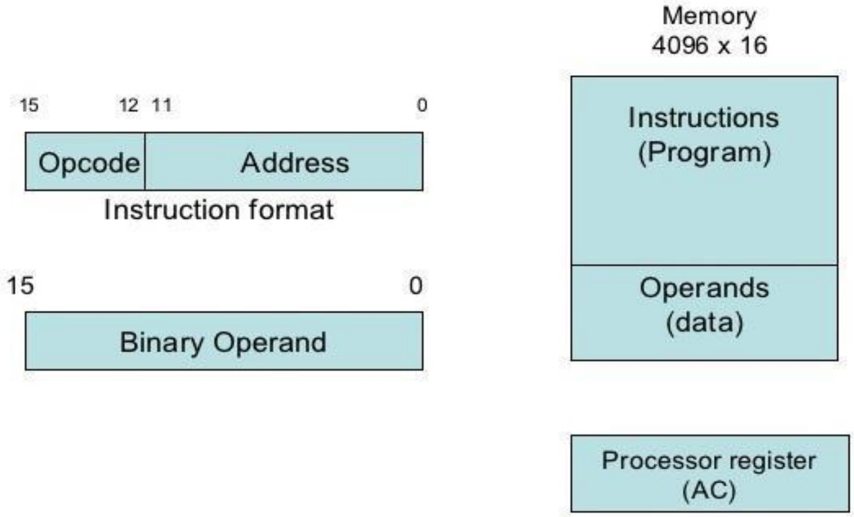


Figure 2.1: Stored Program Organization

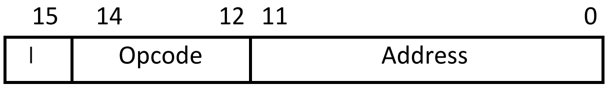
- Instructions are stored in one section of memory and data in another.
- For a memory unit with 4096 words, we need 12 bits to specify an address since 2^{12} = 4096.

- If we store each instruction code in one 16-bit memory word, we have available four bits for operation code (abbreviated opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand.
- The control reads a 16-bit instruction from the program portion of memory.
- It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory.
- It then executes the operation specified by the operation code.
- Computers that have a single-processor register usually assign to it the name accumulator and label it AC.
- If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes.
- For example, operations such as clear AC, complement AC, and increment AC operate on data stored in the AC register. They do not need an operand from memory. For these types of operations, the second part of the instruction code (bits 0 through 11) is not needed for specifying a memory address and can be used to specify other operations for the computer.

Direct and Indirect addressing of basic computer.

- The second part of an instruction format specifies the address of an operand, the instruction is said to have a **direct address**.
- In **Indirect address**, the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found.
- One bit of the instruction code can be used to distinguish between a direct and an indirect address.
- It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by I.
- The mode bit is 0 for a direct address and 1 for an indirect address.
- A direct address instruction is shown in Figure 2.2. It is placed in address 22 in memory.
- The I bit is 0, so the instruction is recognized as a direct address instruction.
- The opcode specifies an ADD instruction, and the address part is the binary equivalent of 457.
- The control finds the operand in memory at address 457 and adds it to the content of AC.
- The instruction in address 35 shown in Figure 2.3 has a mode bit I = 1, recognized as an indirect address instruction.
- The address part is the binary equivalent of 300.
- The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of AC.

- The indirect address instruction needs two references to memory to fetch an operand.
  1. The first reference is needed to read the address of the operand
  2. Second reference is for the operand itself.
- The memory word that holds the address of the operand in an indirect address instruction is used as a pointer to an array of data.



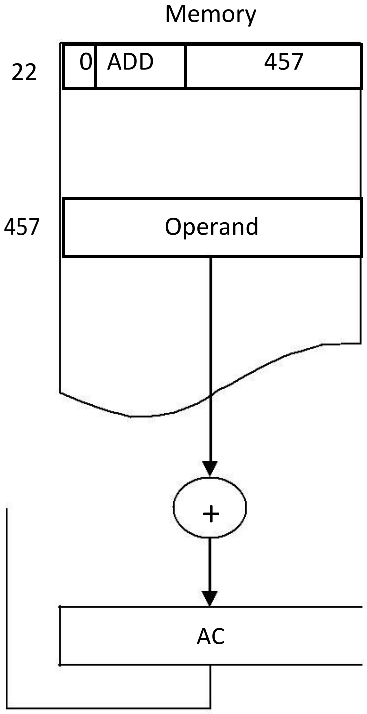


Figure 2.2: Direct Address

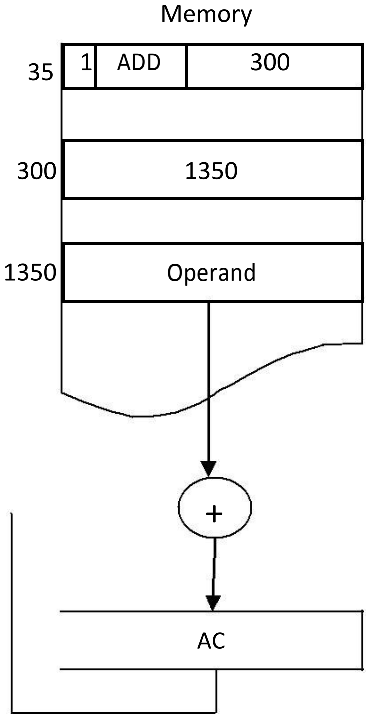


Figure 2.3: Indirect Address

Direct Address	Indirect Address
When the second part of an instruction code specifies the address of an operand, the instruction is said to have a direct address.	When the second part of an instruction code specifies the address of a memory word in which the address of the operand, the instruction is said to have a direct address.
For instance the instruction MOV R0 00H. R0, when converted to machine language is the physical address of register R0. The instruction moves 0 to R0.	For instance the instruction MOV @R0 00H, when converted to machine language, @R0 becomes whatever is stored in R0, and that is the address used to move 0 to. It can be whatever is stored in R0.

Registers of basic computer

- It is necessary to provide a register in the control unit for storing the instruction code after it is read from memory.
- The computer needs processor registers for manipulating data and a register for holding a memory address.
- These requirements dictate the register configuration shown in Figure 2.4.

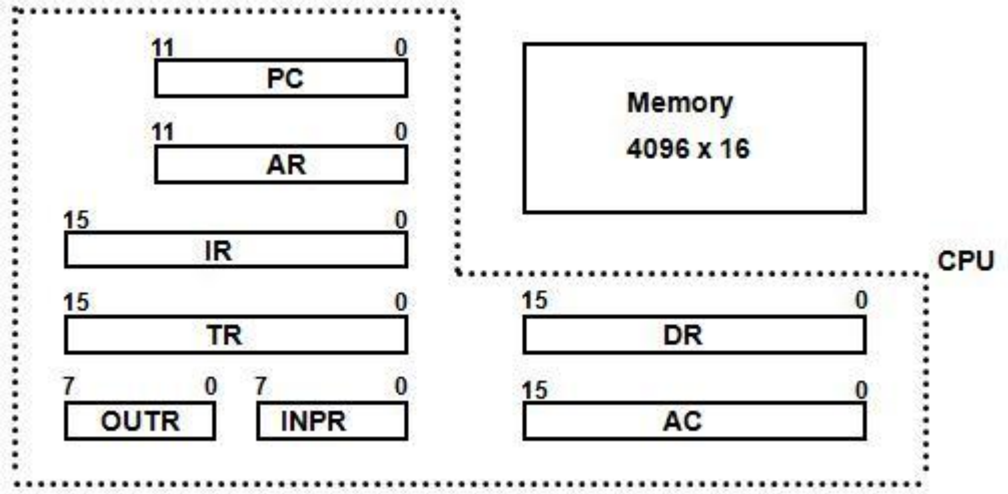


Figure 2.4: Basic Computer Register and Memory

- The data register (DR) holds the operand read from memory.
- The accumulator (AC) register is a general purpose processing register.
- The instruction read from memory is placed in the instruction register (IR).
- The temporary register (TR) is used for holding temporary data during the processing.
- The memory address register (AR) has 12 bits.
- The program counter (PC) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed.
- Instruction words are read and executed in sequence unless a branch instruction is encountered. A branch instruction calls for a transfer to a nonconsecutive instruction in the program.
- Two registers are used for input and output. The input register (INPR) receives an 8-bit character from an input device. The output register (OUTR) holds an 8-bit character for an output device.

Register Symbol	Bits	Register Name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

Table 2.1: List of Registers for Basic Computer

Common Bus System for basic computer register.

What is the requirement of common bus System?

- The basic computer has eight registers, a memory unit and a control unit.
- Paths must be provided to transfer information from one register to another and between memory and register.
- The number of wires will be excessive if connections are between the outputs of each register and the inputs of the other registers. An efficient scheme for transferring information in a system with many register is to use a common bus.
- The connection of the registers and memory of the basic computer to a common bus system is shown in figure 2.5.
- The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S2, S1, and S0.
- The number along each output shows the decimal equivalent of the required binary selection.
- The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and S2 S1 S0 = 1 1 1.
- Four registers, DR, AC, IR, and TR have 16 bits each.
- Two registers, AR and PC, have 12 bits each since they hold a memory address.
- When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to 0's. When AR and PC receive information from the bus, only the 12 least significant bits are transferred into the register.
- The input register INPR and the output register OUTR have 8 bits each and communicate with the eight least significant bits in the bus. INPR is connected to provide information to the bus but OUTR can only receive information from the bus.

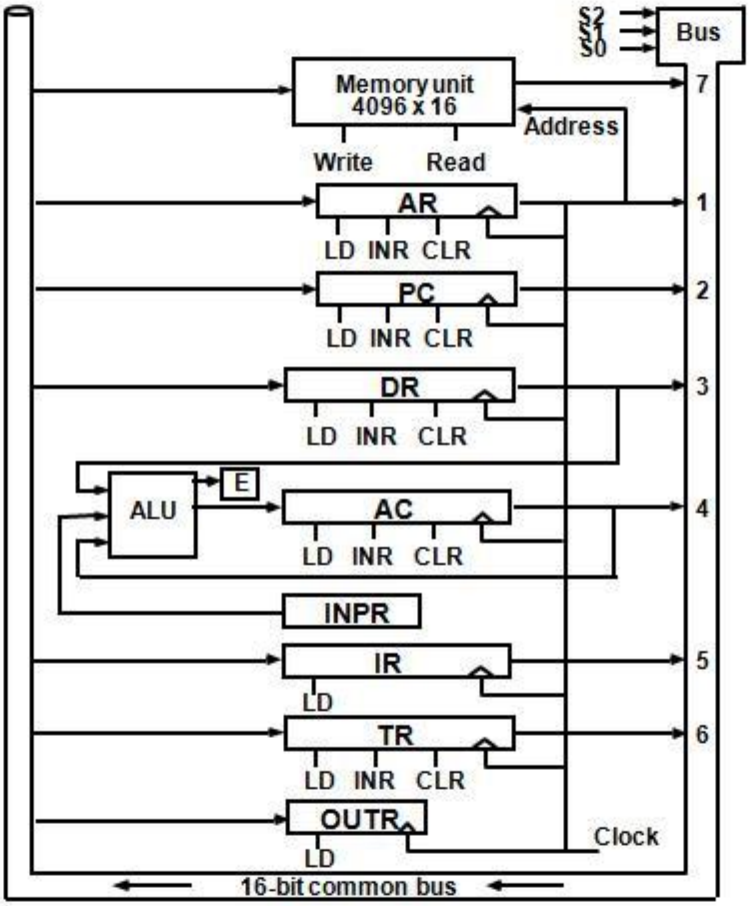


Figure 2.5: Basic computer registers connected to a common bus

- Five registers have three control inputs: LD (load), INR (increment), and CLR (clear). Two registers have only a LD input.
- AR must always be used to specify a memory address; therefore memory address is connected to AR.
- The 16 inputs of AC come from an adder and logic circuit. This circuit has three sets of inputs.
  1. Set of 16-bit inputs come from the outputs of AC.
  2. Set of 16-bits come from the data register DR.
  3. Set of 8-bit inputs come from the input register INPR.
- The result of an addition is transferred to AC and the end carry-out of the addition is transferred to flip-flop E (extended AC bit).
- The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into AC.

Instruction Format with its types.

- The basic computer has three instruction code formats, as shown in figure 2.6.

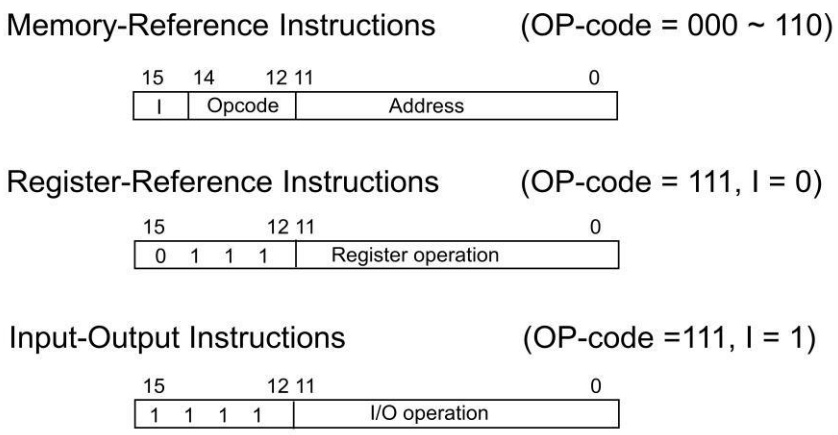


Figure 2.6: Basic computer instruction format

- Each format has 16 bits.
- The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.
- A **memory-reference instruction** uses 12 bits to specify an address and one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirect address.
- The **register reference instructions** are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of the instruction. A register-reference instruction specifies an operation on or a test of the AC register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed.
- An **input-output instruction** does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation or test performed.

Control Unit with timing diagram.

- The block diagram of the control unit is shown in figure 2.7.
- Components of Control unit are
  1. Two decoders
  2. A sequence counter
  3. Control logic gates
- An instruction read from memory is placed in the instruction register (IR). In control unit the IR is divided into three parts: I bit, the operation code (12-14)bit, and bits 0 through 11.
- The operation code in bits 12 through 14 are decoded with a 3 X 8 decoder.

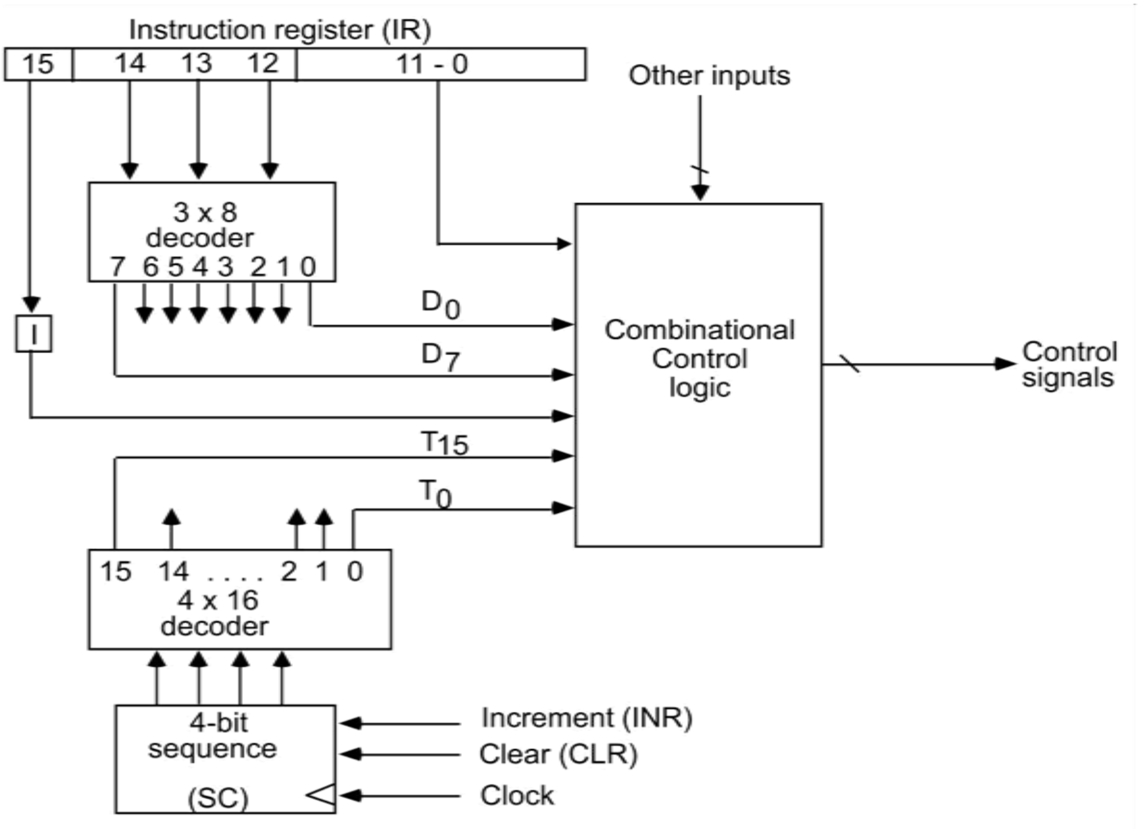


Figure 2.7: Control unit of basic computer

- Bit-15 of the instruction is transferred to a flip-flop designated by the symbol I.
- The eight outputs of the decoder are designated by the symbols D0 through D7. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15.The outputs of counter are decoded into 16 timing signals T0 through T_{15}.
- The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of 4 X 16 decoder. Once in awhile, the counter is cleared to 0, causing the next timing signal to be T0.
- As an example, consider the case where SC is incremented to provide timing signals T0, T_1, T_2, T_3 and T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D3 is active. This is expressed symbolically by the statement

$$D_3T_4: SC \leftarrow 0$$

Timing Diagram:

- The timing diagram figure2.8 shows the time relationship of the control signals.
- The sequence counter SC responds to the positive transition of the clock.
- Initially, the CLR input of SC is active.
- The first positive transition of the clock clears SC to 0, which in turn activates the timing T_0 out of the decoder. T_0 is active during one clock cycle. The positive clock transition

labeled T_0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T_0.

- SC is incremented with every positive clock transition, unless its CLR input is active.
- This procedures the sequence of timing signals T_0, T_1, T_2, T_3 and T_4, and so on. If SC is not cleared, the timing signals will continue with T_5, T_6, up to T_{15} and back to T_0.

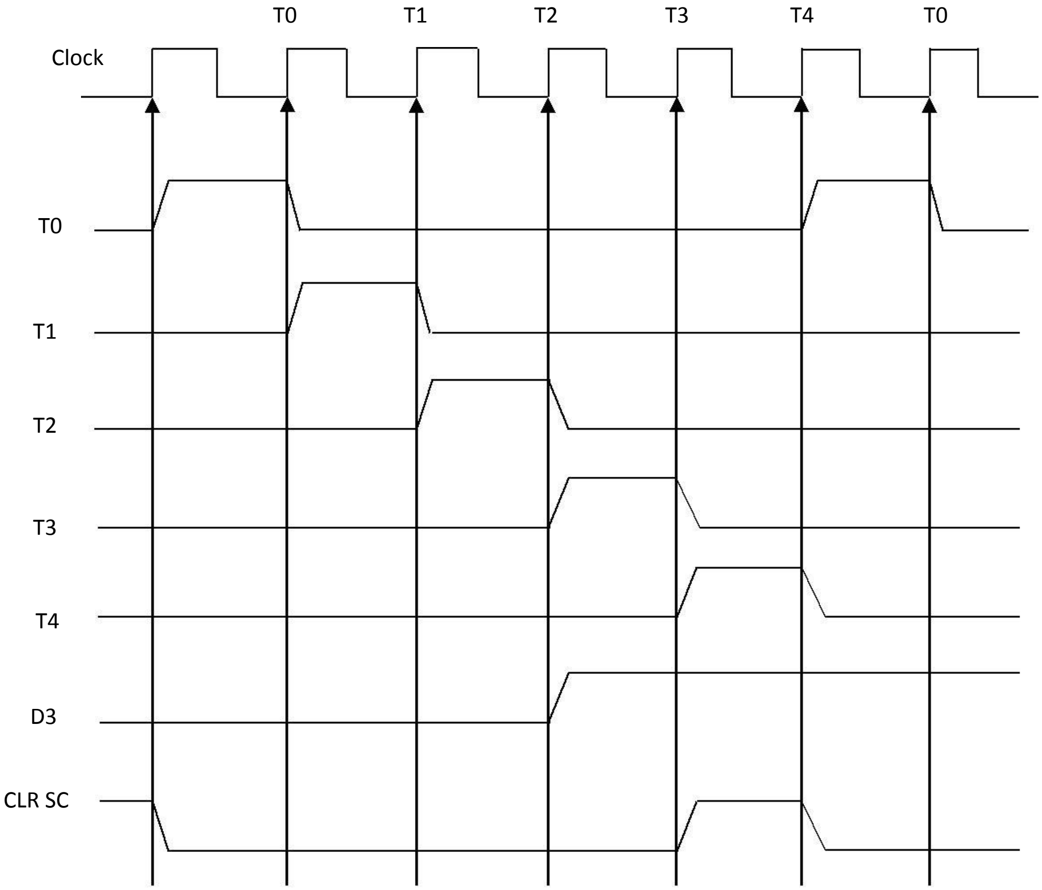


Figure 2.8: Example of control timing signals

- The last three waveforms shows how SC is cleared when D3T4 = 1. Output D3 from the operation decoder becomes active at the end of timing signal T_2. When timing signal T_4 becomes active, the output of the AND gate that implements the control function D_3T_4 becomes active.
- This signal is applied to the CLR input of SC. On the next positive clock transition the counter is cleared to 0. This causes the timing signal T_0 to become active instead of T_5 that would have been active if SC were incremented instead of cleared.

Instruction cycle

- A program residing in the memory unit of the computer consists of a sequence of instructions. In the basic computer each instruction cycle consists of the following phases:
  1. Fetch an instruction from memory.
  2. Decode the instruction.
  3. Read the effective address from memory if the instruction has an indirect address.
  4. Execute the instruction.
- After step 4, the control goes back to step 1 to fetch, decode and execute the nex instruction.
- This process continues unless a HALT instruction is encountered.

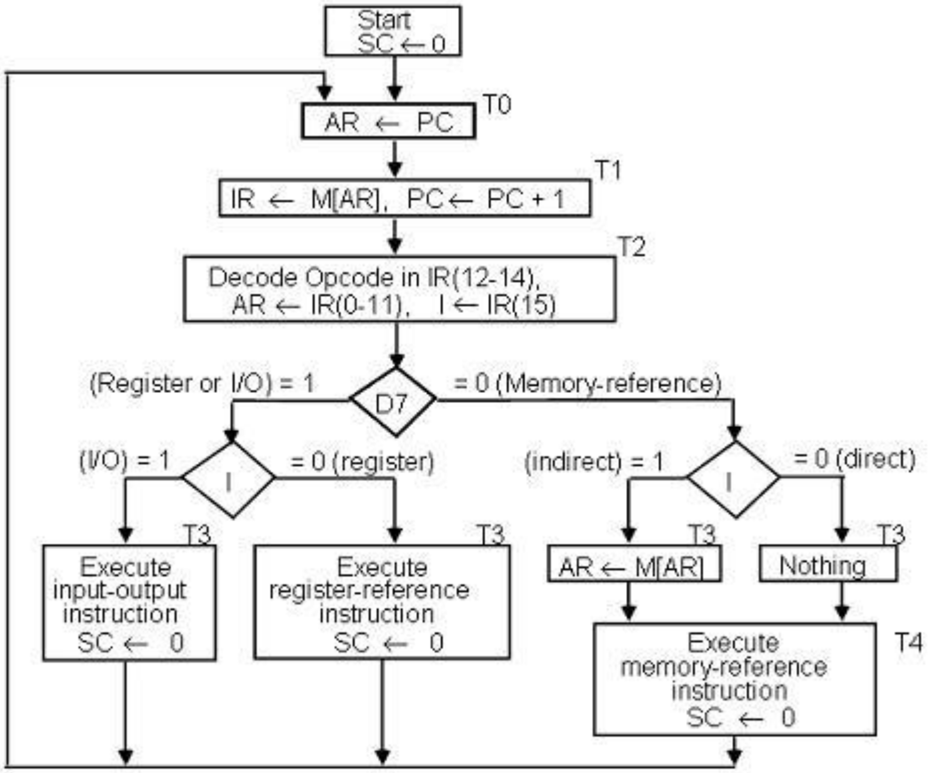

Start
SC ← 0
AR ← PC
T0
IR ← M[AR], PC ← PC + 1
T1
Decode Opcode in IR(12-14),
AR ← IR(0-11), I ← IR(15)
T2
(Register or I/O) = 1
D7
= 0 (Memory-reference)
(I/O) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T3
Execute input-output instruction SC ← 0
T3
Execute register-reference instruction SC ← 0
T3
AR ← M[AR]
T3
Nothing
T4
Execute memory-reference instruction SC ← 0

Figure 2.9: Flowchart for instruction cycle (initial configuration)

- The flowchart presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding.
- If D7 = 1, the instruction must be register-reference or input-output type. If D7 = 0, the operation code must be one of the other seven values 110, specifying a memory-reference instruction. Control then inspects the value of the first bit of the instruction, which now available in flip-flop I.
- If D7 = 0 and I = 1, we have a memory-reference instruction with an indirect address. It is then necessary to read the effective address from memory.
- The three instruction types are subdivided into four separate paths. The selected

operation is activated with the clock transition associated with timing signal T_3.This can be symbolized as follows:

D’7 I T3: AR ← M [AR]
D’7 I’ T3: Nothing
D7 I’ T3: Execute a register-reference instruction
D7 I T3: Execute an input-output instruction

- When a memory-reference instruction with I = 0 is encountered, it is not necessary to do anything since the effective address is already in AR.
- However, the sequence counter SC must be incremented when D’7 I T3 = 1, so that the execution of the memory-reference instruction can be continued with timing variable T4.
- A register-reference or input-output instruction can be executed with the click associated with timing signal T3. After the instruction is executed, SC is cleared to 0 and control returns to the fetch phase with T0 =1. SC is either incremented or cleared to 0 with every positive clock transition.

Register reference instruction.

- When the register-reference instruction is decoded, D7 bit is set to 1.
- Each control function needs the Boolean relation D7 I' T3

15 12	11 0
0 1 1 1	Register Operation

- There are 12 register-reference instructions listed below:

	r:	SC←0	Clear SC
CLA	rB_{11}:	AC ← 0	Clear AC
CLE	rB_{10}:	E ← 0	Clear E
CMA	rB_{9}:	AC ← AC’	Complement AC
CME	rB_{8}:	E ← E’	Complement E
CIR	rB_{7}:	AC ← shr AC, AC(15) ← E, E ← AC(0)	Circular Right
CIL	rB_{6}:	AC ← shl AC, AC(0) ← E, E ← AC(15)	Circular Left
INC	rB_{5}:	AC ← AC + 1	Increment AC
SPA	rB_{4}:	if (AC(15) = 0) then (PC ← PC+1)	Skip if positive
SNA	rB_{3}:	if (AC(15) = 1) then (PC ← PC+1	Skip if negative
SZA	rB_{2}:	if (AC = 0) then (PC ← PC+1)	Skip if AC is zero
SZE	rB_{1}:	if (E = 0) then (PC ← PC+1)	Skip if E is zero
HLT	rB_{0}:	S ← 0 (S is a start-stop flip-flop)	Halt computer

- These 12 bits are available in IR (0-11). They were also transferred to AR during time T_2.
- These instructions are executed at timing cycle T_3.
- The first seven register-reference instructions perform clear, complement, circular shift, and increment microoperations on the AC or E registers.
- The next four instructions cause a skip of the next instruction in sequence when

condition is satisfied. The skipping of the instruction is achieved by incrementing PC.

- The condition control statements must be recognized as part of the control conditions. The AC is positive when the sign bit in AC(15) = 0; it is negative when AC(15) = 1. The content of AC is zero (AC = 0) if all the flip-flops of the register are zero.
- The HLT instruction clears a start-stop flip-flop S and stops the sequence counter from counting. To restore the operation of the computer, the start-stop flip-flop must be set manually.

Memory reference instructions

- When the memory-reference instruction is decoded, D7 bit is set to 0.

15	14 12	11 0
I	000~110	Address

- The following table lists seven memory-reference instructions.

Symbol	Operation Decoder	Symbolic Description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR]$, $E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC$, $PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, if $M[AR] + 1 = 0$ then $PC \leftarrow PC+1$

- The effective address of the instruction is in the address register AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I = 1.
- The execution of the memory-reference instructions starts with timing signal T_4.

AND to AC

This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address. The result of the operation is transferred to AC.

D_0T_4: $DR \leftarrow M[AR]$

D_0T_5: $AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$

ADD to AC

This instruction adds the content of the memory word specified by the effective address to the value of AC. The sum is transferred into AC and the output carry C_{out} is transferred to the E (extended accumulator) flip-flop.

D_1T_4: $DR \leftarrow M[AR]$

D_1T_5: $AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$

LDA: Load to AC

This instruction transfers the memory word specified by the effective address to AC.

D_2T_4: DR ← M[AR]

D_2T_5: AC ← DR, SC ← 0

STA: Store AC

This instruction stores the content of AC into the memory word specified by the effective address.

D_3T_4: M[AR] ← AC, SC ← 0

BUN: Branch Unconditionally

This instruction transfers the program to instruction specified by the effective address. The BUN instruction allows the programmer to specify an instruction out of sequence and the program branches (or jumps) unconditionally.

D_4T_4: PC ← AR, SC ← 0

BSA: Branch and Save Return Address

This instruction is useful for branching to a portion of the program called a subroutine or procedure. When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in PC) into a memory location specified by the effective address.

M[AR] ← PC, PC ← AR + 1

M[135] ←21, PC ←135 + 1 = 136

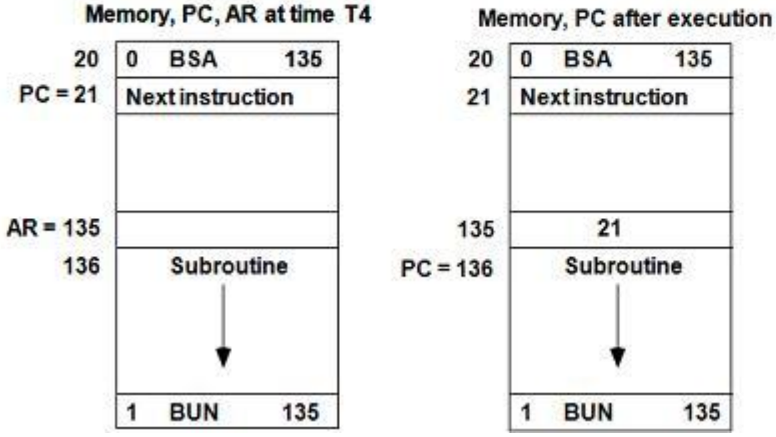


Figure2.10: Example of BSA instruction execution

It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer. To use the memory and the bus properly, the BSA instruction must be executed with a sequence of two microoperations:

D_5T_4: M[AR] ← PC, AR ← AR + 1

D_5T_5: PC ← AR, SC ← 0

ISZ: Increment and Skip if Zero

These instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1. Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory.

D_6T_4: DR ← M[AR]

D_6T_5: DR ← DR + 1

D_6T_4: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

Control Flowchart

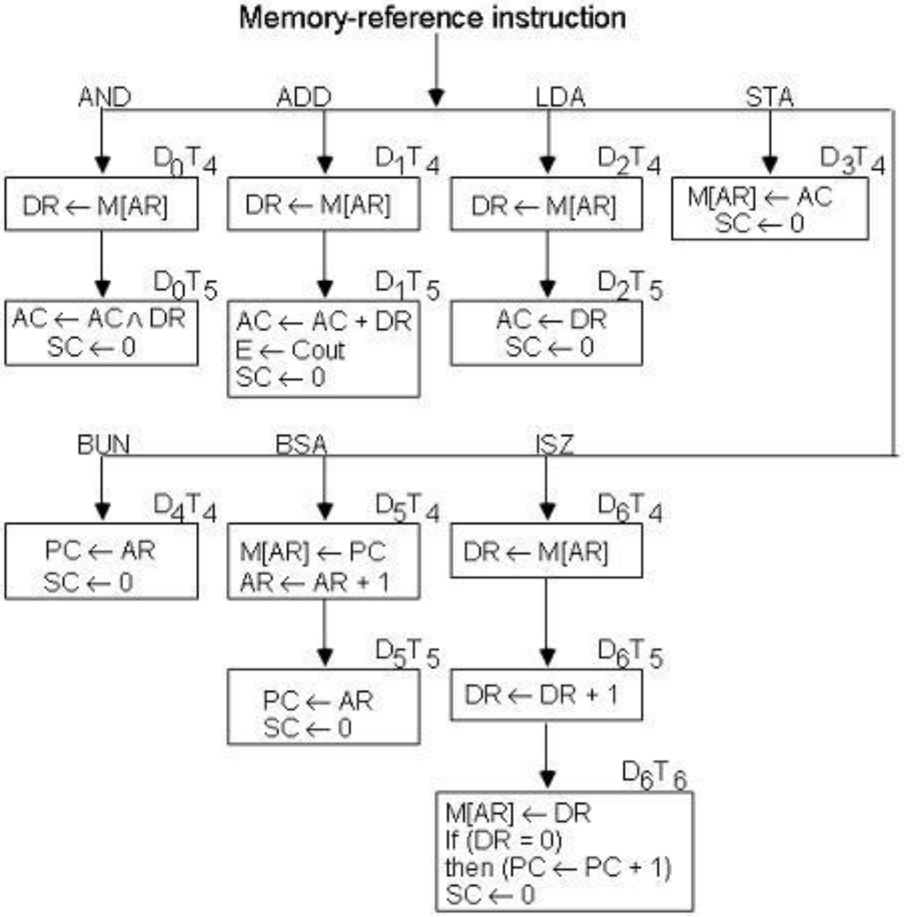


Figure 2.11: Flowchart for memory-reference instructions

Input-output configuration of basic computer

- A computer can serve no useful purpose unless it communicates with the external environment.
- To exhibit the most basic requirements for input and output communication, we will use a terminal unit with a keyboard and printer.

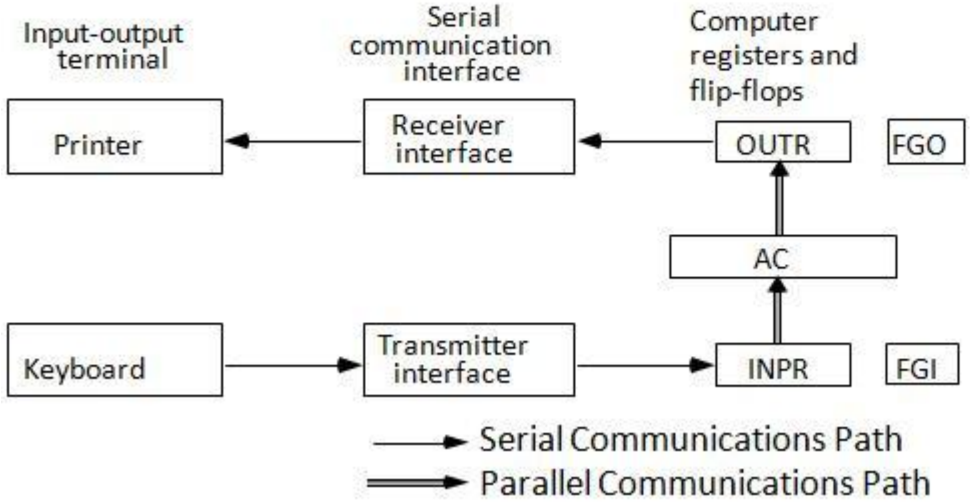

Input-output terminal
Serial communication interface
Computer registers and flip-flops
Printer
Receiver interface
OUTR
FGO
AC
Keyboard
Transmitter interface
INPR
FGI
Serial Communications Path
Parallel Communications Path

Figure 2.12: Input-output configuration

- The terminal sends and receives serial information and each quantity of information has eight bits of an alphanumeric code.
- The serial information from the keyboard is shifted into the input register INPR.
- The serial information for the printer is stored in the output register OUTR.
- These two registers communicate with a communication interface serially and with the AC in parallel.
- The transmitter interface receives serial information from the keyboard and transmits it to INPR. The receiver interface receives information from OUTR and sends it to the printer serially.
- The 1-bit input flag FGI is a control flip-flop. It is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer.
- The flag is needed to synchronize the timing rate difference between the input device and the computer.
- The process of information transfer is as follows:

The process of input information transfer:

- Initially, the input flag FGI is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1.
- As long as the flag is set, the information in INPR cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0.

- Once the flag is cleared, new information can be shifted into INPR by striking another key.

The process of outputting information:

- The output register OUTR works similarly but the direction of information flow is reversed.
- Initially, the output flag FGO is set to 1. The computer checks the flag bit; if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets FGO to 1.
- The computer does not load a new character into OUTR when FGO is 0 because this condition indicates that the output device is in the process of printing the character.

Input-Output instructions

- Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility.
- Input-output instructions have an operation code 1111 and are recognized by the control when D7 = 1 and I = 1.
- The remaining bits of the instruction specify the particular operation.
- The control functions and microoperations for the input-output instructions are listed below.

INP	AC(0-7) ← INPR, FGI ← 0	Input char. to AC
OUT	OUTR ← AC(0-7), FGO ← 0	Output char. from AC
SKI	if(FGI = 1) then (PC ← PC + 1)	Skip on input flag
SKO	if(FGO = 1) then (PC ← PC + 1)	Skip on output flag
ION	IEN ← 1	Interrupt enable on
IOF	IEN ← 0	Interrupt enable off

Table 2.2: Input Output Instructions

- The INP instruction transfers the input information from INPR into the eight low-order bits of AC and also clears the input flag to 0.
- The OUT instruction transfers the eight least significant bits of AC into the output register OUTR and clears the output flag to 0.
- The next two instructions in Table 2.2 check the status of the flags and cause a skip of the next instruction if the flag is 1.
- The instruction that is skipped will normally be a branch instruction to return and check the flag again.
- The branch instruction is not skipped if the flag is 0. If the flag is 1, the branch instruction is skipped and an input or output instruction is executed.
- The last two instructions set and clear an interrupt enable flip-flop IEN. The purpose of IEN is explained in conjunction with the interrupt operation.

Interrupt Cycle

The way that the interrupt is handled by the computer can be explained by means of the flowchart shown in figure 2.13.

- An interrupt flip-flop R is included in the computer.
- When R = 0, the computer goes through an instruction cycle.
- During the execute phase of the instruction cycle IEN is checked by the control.
- If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle.
- If IEN is 1, control checks the flag bits.
- If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information.
- In this case, control continues with the next instruction cycle. If either flag is set to 1 while IEN = 1, flip-flop R is set to 1.
- At the end of the execute phase, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.

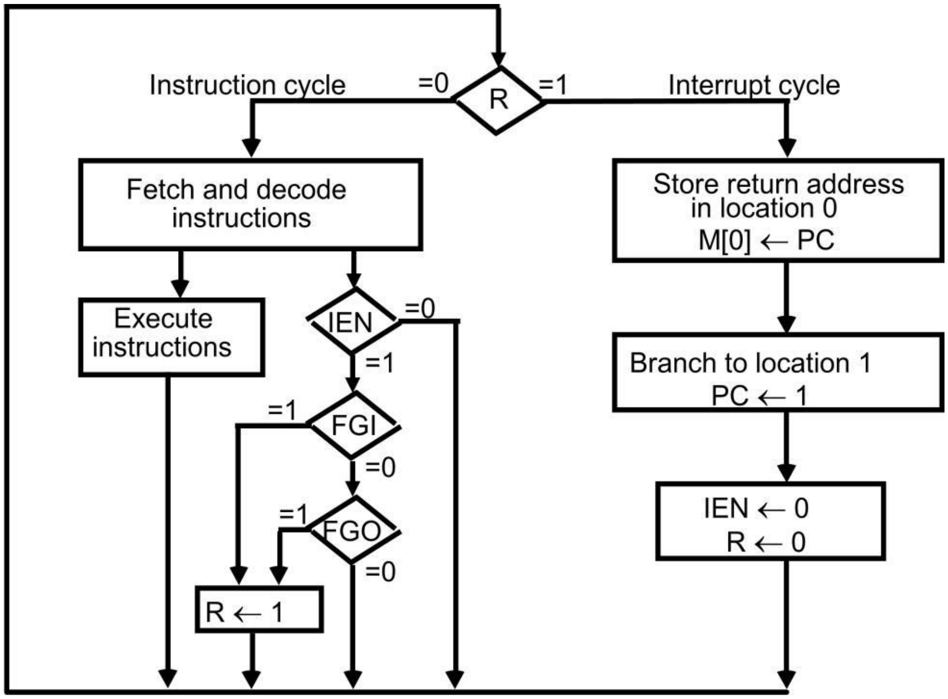


Figure 2.13: Flowchart for interrupt cycle

Interrupt Cycle

- The interrupt cycle is a hardware implementation of a branch and save return address operation.
- The return address available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted. This location may be a processor register, a memory stack, or a specific memory location.
- Here we choose the memory location at address 0 as the place for storing the return

address.

- Control then inserts address 1 into PC and clears IEN and R so that no more interruptions can occur until the interrupt request from the flag has been serviced.
- An example that shows what happens during the interrupt cycle is shown in Figure 2.14:

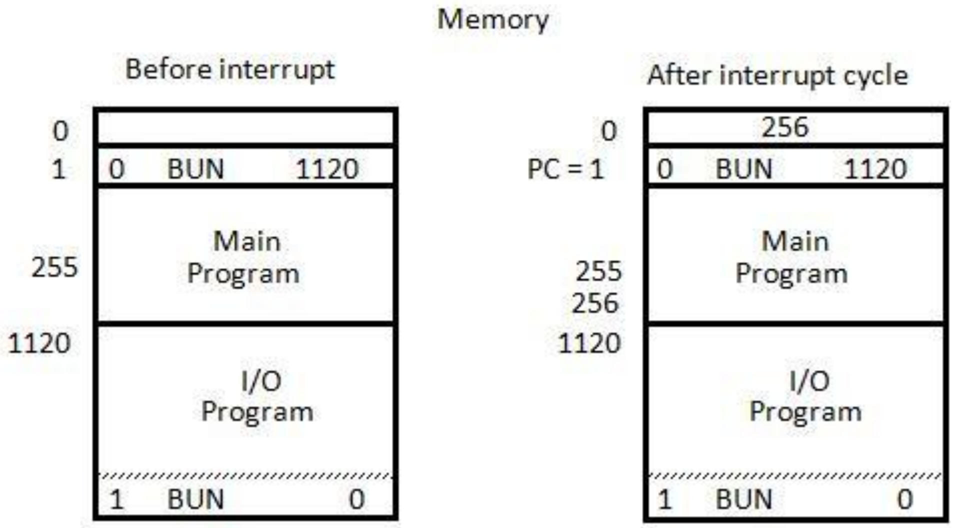


Figure 2.14: Demonstration of the interrupt cycle

- Suppose that an interrupt occurs and R = 1, while the control is executing the instruction at address 255. At this time, the return address 256 is in PC.
- The programmer has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1.
- The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0.
- At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of PC. The branch instruction at address 1 causes the program to transfer to the input-output service program at address 1120.
- This program checks the flags, determines which flag is set, and then transfers the required input or output information. Once this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted.
- The instruction that returns the computer to the original place in the main program is a branch indirect instruction with an address part of 0. This instruction is placed at the end of the I/O service program.
- The execution of the indirect BUN instruction results in placing into PC the return address from location 0.

Register transfer statements for the interrupt cycle

- The flip-flop is set to 1 if IEN = 1 and either FGI or FGO are equal to 1. This can happen with any clock transition except when timing signals T_0, T_1 or T_2 are active.
- The condition for setting flip-flop R= 1 can be expressed with the following register transfer statement:

$$T_0'T_1'T_2' \text{ (IEN) (FGI + FGO): } R \leftarrow 1$$

- The symbol + between FGI and FGO in the control function designates a logic OR operation. This is AND with IEN and $T_0'T_1' T_2'$.
- The fetch and decode phases of the instruction cycle must be modified and Replace T_0, T_1, T_2 with $R'T_0$, $R'T_1$, $R'T_2$
- Therefore the interrupt cycle statements are :
  RT_0: AR ← 0, TR ← PC
  RT_1: M[AR] ← TR, PC ← 0
  RT_2: PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0
- During the first timing signal AR is cleared to 0, and the content of PC is transferred to the temporary register TR.
- With the second timing signal, the return address is stored in memory at location 0 and PC is cleared to 0.
- The third timing signal increments PC to 1, clears IEN and R, and control goes back to T_0 by clearing SC to 0.
- The beginning of the next instruction cycle has the condition RT_0 and the content of PC is equal to 1. The control then goes through an instruction cycle that fetches and executes the BUN instruction in location 1.

Flow chart for computer operation.

- The final flowchart of the instruction cycle, including the interrupt cycle for the basic computer, is shown in Figure 2.15.
- The interrupt flip-flop R may be set at any time during the indirect or execute phases.
- The control returns to timing signal T_0 after SC is cleared to 0.
- If R = 1, the computer goes through an interrupt cycle. If R = 0, the computer goes through an instruction cycle.
- If the instruction is one of the memory-reference instructions, the computer first checks if there is an indirect address and then continues to execute the decoded instruction according to the flowchart.
- If the instruction is one of the register-reference instructions, it is executed with one of the microoperations register reference.
- If it is an input-output instruction, it is executed with one of the microoperation's input-output reference.

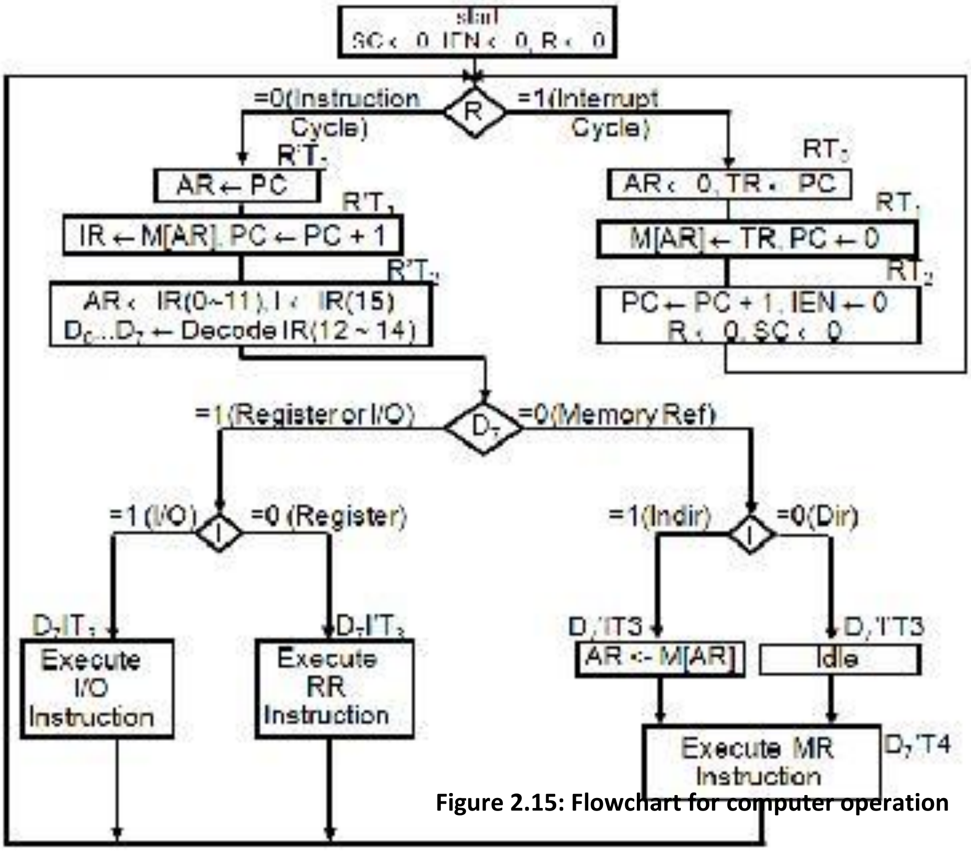


Figure 2.15: Flowchart for computer operation

REFERENCE :

1. COMPUTER SYSTEM ARCHITECTURE, MORRIS M. MANO, 3RD EDITION, PRENTICE HALL INDIA.

Unit 2 – Microprogrammed Control

Hardwired Control Unit:

When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.

Micro programmed control unit:

A control unit whose binary control variables are stored in memory is called a micro programmed control unit.

Dynamic microprogramming:

A more advanced development known as *dynamic* microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing.

Control Memory:

Control Memory is the storage in the microprogrammed control unit to store the microprogram.

Writeable Control Memory:

Control Storage whose contents can be modified, allow the change in microprogram and Instruction set can be changed or modified is referred as Writeable Control Memory**.**

Control Word:

The control variables at any given time can be represented by a control word string of 1 's and 0's called a control word.

Microoperation, Microinstruction, Micro program, Microcode.

Microoperations:

- In computer central processing units, micro-operations (also known as a micro-ops or μops) are detailed low-level instructions used in some designs to implement complex machine instructions (sometimes termed macro-instructions in this context).

Micro instruction:

- A symbolic microprogram can be translated into its binary equivalent by means of an assembler.
- Each line of the assembly language microprogram defines a symbolic microinstruction.
- Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD.

Micro program:

- A sequence of microinstructions constitutes a microprogram.
- Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM).
- ROM words are made permanent during the hardware production of the unit.
- The use of a micro program involves placing all control variables in words of ROM for use by the control unit through successive read operations.
- The content of the word in ROM at a given address specifies a microinstruction.

Microcode:

- Microinstructions can be saved by employing subroutines that use common sections of microcode.
- For example, the sequence of micro operations needed to generate the effective address of the operand for an instruction is common to all memory reference instructions.
- This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

Organization of micro programmed control unit

- The general configuration of a micro-programmed control unit is demonstrated in the block diagram of Figure 4.1.
- The control memory is assumed to be a ROM, within which all control information is permanently stored.

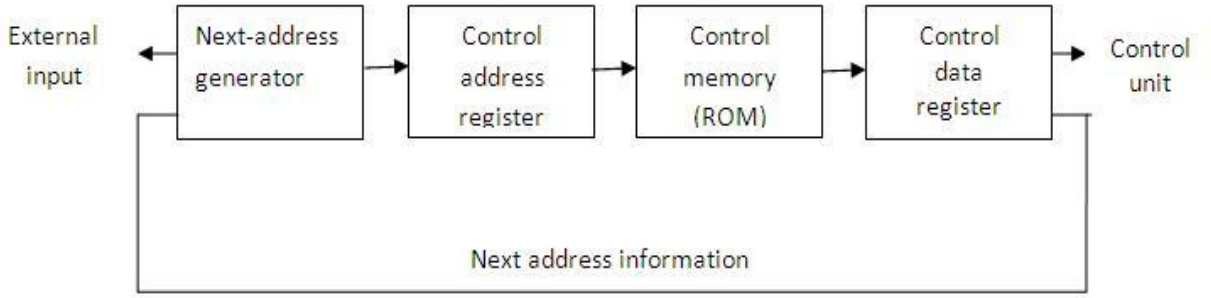


figure 4.1: Micro-programmed control organization

- The control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory.
- The microinstruction contains a control word that specifies one or more microoperations for the data processor. Once these operations are executed, the control must determine the next address.
- The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory.

- While the microoperations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.
- Thus a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.
- The next address generator is sometimes called a ***micro-program sequencer***, as it determines the address sequence that is read from control memory.
- Typical functions of a micro-program sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations.
- The control data register holds the present microinstruction while the next address is computed and read from memory.
- The data register is sometimes called a ***pipeline register***.
- It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next microinstruction.
- This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.
- The main advantage of the micro programmed control is the fact that once the hardware configuration is established; there should be no need for further hardware or wiring changes.
- If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory.

Address Sequencing

- Microinstructions are stored in control memory in groups, with each group specifying a ***routine***.
- To appreciate the address sequencing in a micro-program control unit, let us specify the steps that the control must undergo during the execution of a single computer instruction.

Step-1:

- An initial address is loaded into the control address register when power is turned on in the computer.
- This address is usually the address of the first microinstruction that activates the instruction fetch routine.
- The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions.
- At the end of the fetch routine, the instruction is in the instruction register of the computer.

Step-2:

- The control memory next must go through the routine that determines the effective address of the operand.
- A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers.
- The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction.
- When the effective address computation routine is completed, the address of the operand is available in the memory address register.

Step-3:

- The next step is to generate the microoperations that execute the instruction fetched from memory.
- The microoperation steps to be generated in processor registers depend on the operation code part of the instruction.
- Each instruction has its own micro-program routine stored in a given location of control memory.
- The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a ***mapping*** process.
- A mapping procedure is a rule that transforms the instruction code into a control memory address.

Step-4:

- Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register.
- Micro-programs that employ subroutines will require an external register for storing the return address.
- Return addresses cannot be stored in ROM because the unit has no writing capability.
- When the execution of the instruction is completed, control must return to the fetch routine.
- This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine.

In summary, the address sequencing capabilities required in a control memory are:

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.

selection of address for control memory

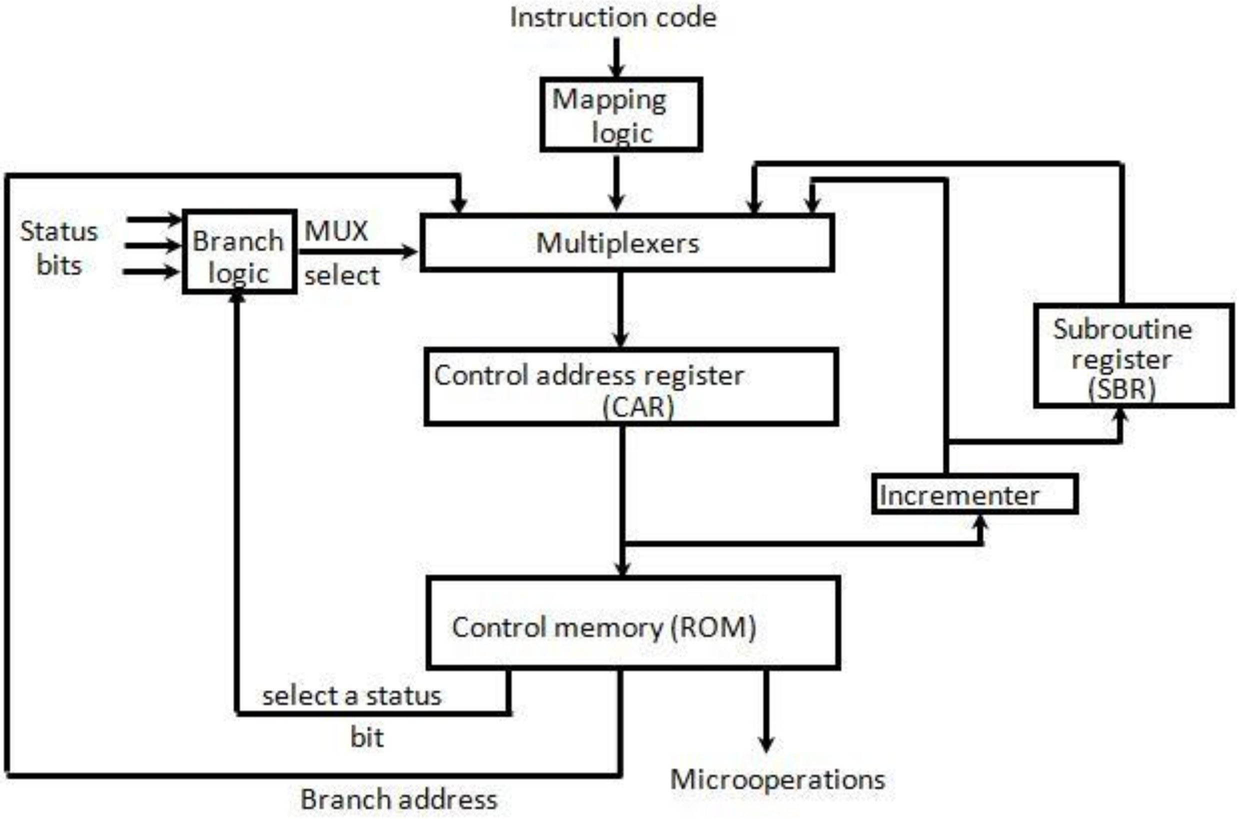


Figure 4.2: Selection of address for control memory

- Above figure 4.2 shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.
- The microinstruction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained.
- The diagram shows four different paths from which the control address register (CAR) receives the address.
- The incrementer increments the content of the control address register by one, to select the next microinstruction in sequence.
- Branching is achieved by specifying the branch address in one of the fields of the microinstruction.
- Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.
- An external address is transferred into control memory via a mapping logic circuit.
- The return address for a subroutine is stored in a special register whose value is then used when the micro-program wishes to return from the subroutine.

- The branch logic of figure 4.2 provides decision-making capabilities in the control unit.
- The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions.
- The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.
- A 1 output in the multiplexer generates a control signal to transfer the branch address from the microinstruction into the control address register.
- A 0 output in the multiplexer causes the address register to be incremented.

Mapping of an Instruction

- A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located.
- The status bits for this type of branch are the bits in the operation code part of the instruction.

  For example, a computer with a simple instruction format as shown in figure 4.3 has an operation code of four bits which can specify up to 16 distinct instructions.
- Assume further that the control memory has 128 words, requiring an address of seven bits.
- One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in figure 4.3.
- This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register.
- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.
- If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111. If it uses fewer than four microinstructions, the unused memory locations would be available for other routines.

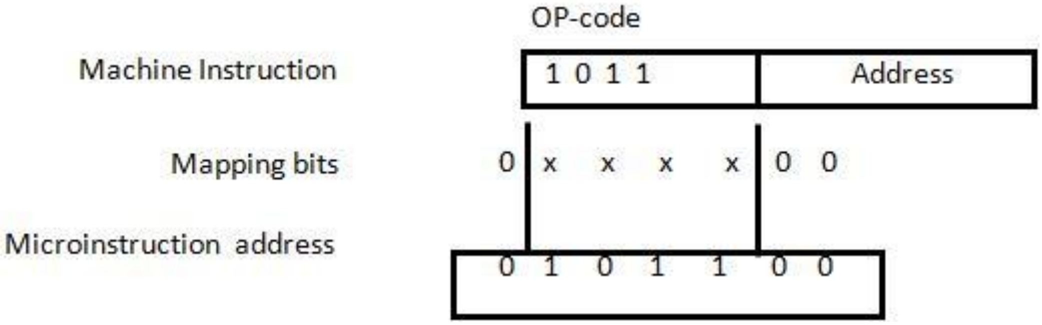


Figure 4.3: Mapping from instruction code to microinstruction address

- One can extend this concept to a more general mapping rule by using a ROM to specify the mapping function.
- The contents of the mapping ROM give the bits for the control address register.

- In this way the microprogram routine that executes the instruction can be placed in any desired location in control memory.
- The mapping concept provides flexibility for adding instructions for control memory as the need arises.

Computer Hardware Configuration

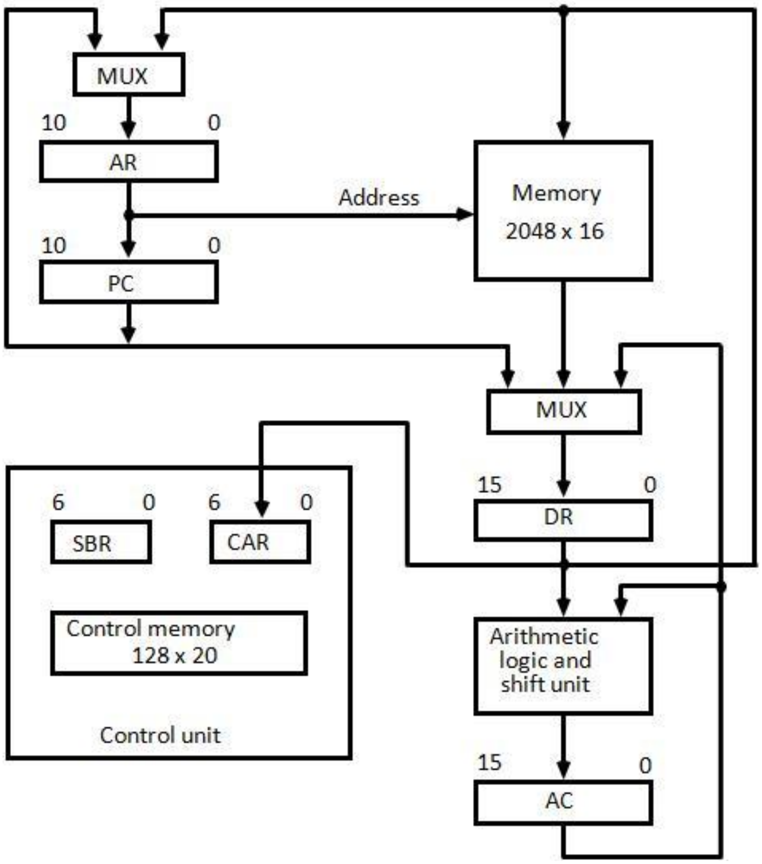


Figure 4.4: Computer hardware configuration

The block diagram of the computer is shown in Figure 4.4. It consists of

1. Two memory units:

   Main memory -> for storing instructions and data, and
   Control memory -> for storing the microprogram.
2. Six Registers:

   Processor unit register: AC(accumulator),PC(Program Counter), AR(Address Register), DR(Data Register)

   Control unit register: CAR (Control Address Register), SBR(Subroutine Register)
3. Multiplexers:

   The transfer of information among the registers in the processor is done through multiplexers rather than a common bus.
4. ALU:

   The arithmetic, logic, and shift unit performs microoperations with data from AC and DR and places the result in AC.

- DR can receive information from AC, PC, or memory.
- AR can receive information from PC or DR.
- PC can receive information only from AR.
- Input data written to memory come from DR, and data read from memory can go only to DR.

Microinstruction Format

The microinstruction format for the control memory is shown in figure 4.5. The 20 bits of the microinstruction are divided into four functional parts as follows:

1. The three fields F1, F2, and F3 specify microoperations for the computer.

   The microoperations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct microoperations. This gives a total of 21 microoperations.
2. The CD field selects status bit conditions.
3. The BR field specifies the type of branch to be used.
4. The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 2^7 words.

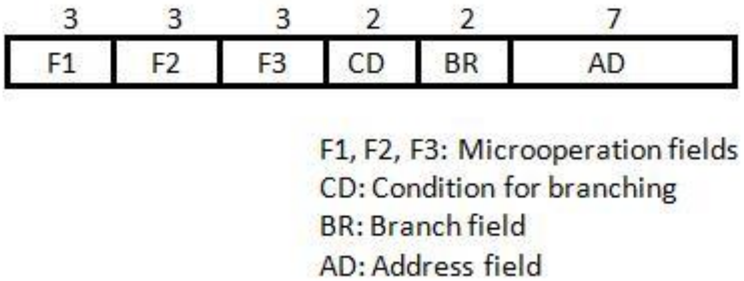
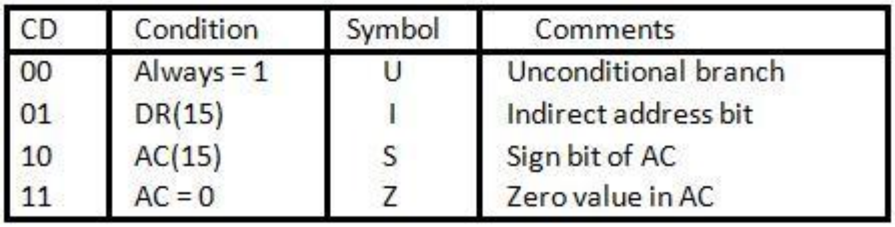

3
3
3
2
2
7
F1
F2
F3
CD
BR
AD
F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

Figure 4.5: Microinstruction Format

- As an example, a microinstruction can specify two simultaneous microoperations from F2 and F3 and none from F1.

  DR □M[AR] with F2 = 100

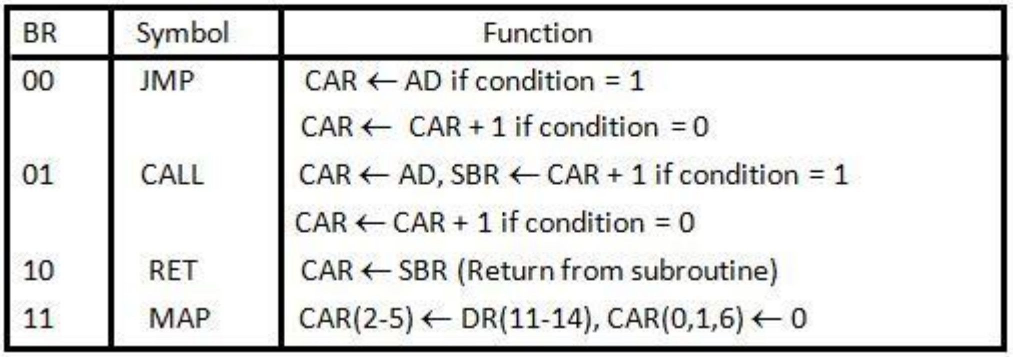
  PC □ PC + 1 with F3 = 101
- The nine bits of the microoperation fields will then be 000 100 101.
- The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table 4.1.



CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

Table 4.1: Condition Field

- The BR (branch) field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction shown in Table 4.2.



BR	Symbol	Function

00	JMP	CAR ← AD if condition = 1
		CAR ← CAR + 1 if condition = 0

01	CALL	CAR ← AD, SBR ← CAR + 1 if condition = 1
		CAR ← CAR + 1 if condition = 0

10	RET	CAR ← SBR (Return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

Table 4.2: Branch Field

Symbolic Microinstruction.

- Each line of the assembly language microprogram defines a symbolic microinstruction.
- Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD. The fields specify the following Table 4.3.

1.	Label	The label field may be empty or it may specify a symbolic address. A label is terminated with a colon (:).
2.	Microoperations	It consists of one, two, or three symbols, separated by commas, from those defined in Table 5.3. There may be no more than one symbol from each F field. The NOP symbol is used when the microinstruction has no microoperations. This will be translated by the assembler to nine zeros.
3.	CD	The CD field has one of the letters U, I, S, or Z.
4.	BR	The BR field contains one of the four symbols defined in Table 5.2.
5.	AD	The AD field specifies a value for the address field of the microinstruction in one of three possible ways: i. With a symbolic address, this must also appear as a label. ii. With the symbol NEXT to designate the next address in sequence. iii. When the BR field contains a RET or MAP symbol, the AD field is left empty and is converted to seven zeros by the assembler.

Table 4.3: Symbolic Microinstruction

Micro programmed sequencer for a control memory

Microprogram sequencer:

- The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address.
- The address selection part is called a microprogram sequencer.
- A microprogram sequencer can be constructed with digital functions to suit a particular application.
- To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applications.
- The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.
- Commercial sequencers include within the unit an internal register stack used for temporary storage of addresses during microprogram looping and subroutine calls.
- Some sequencers provide an output register which can function as the address register for the control memory.
- The block diagram of the microprogram sequencer is shown in figure 4.6.
- There are two multiplexers in the circuit.
- The first multiplexer selects an address from one of four sources and routes it into a control address register CAR.
- The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit.
- The output from CAR provides the address for the control memory.
- The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine registers SBR.
- The other three inputs to multiplexer 1 come from the address field of the present microinstruction, from the output of SBR, and from an external source that maps the instruction.
- Although the figure 4.6 shows a single subroutine register, a typical sequencer will have a register stack about four to eight levels deep. In this way, a number of subroutines can be active at the same time.
- The CD (condition) field of the microinstruction selects one of the status bits in the second multiplexer.
- If the bit selected is equal to 1, the T (test) variable is equal to 1; otherwise, it is equal to 0.
- The T value together with the two bits from the BR (branch) field goes to an input logic circuit.
- The input logic in a particular sequencer will determine the type of operations that are available in the unit.

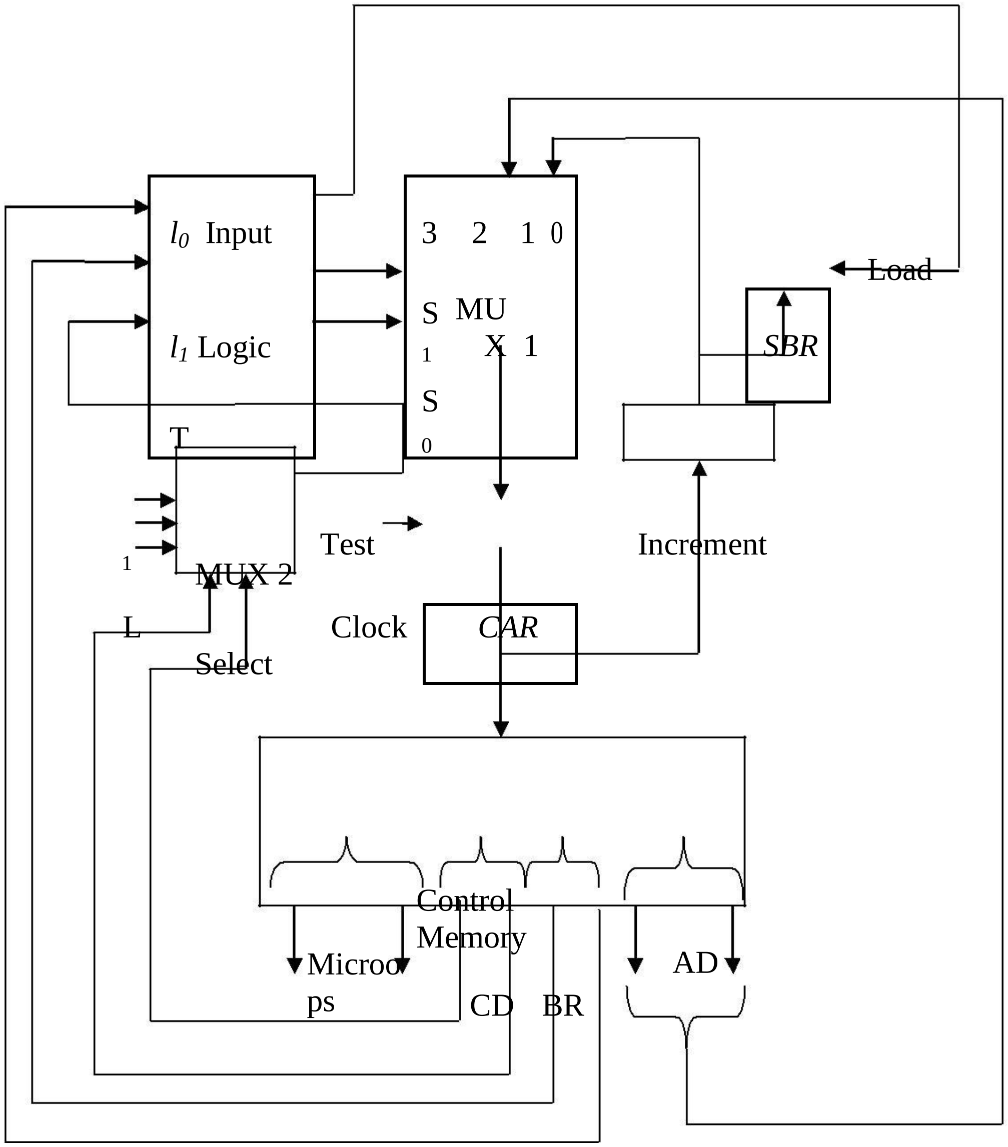


Figure 4.6: Microprogram Sequencer for a control memory

Input Logic : Truth Table

BR	Input			MUX 1		Load SBR
	I1	**I0**	**T**	**S1**	**S0**	**L**
0 0	0	0	0	0	0	0
0 0	0	0	1	0	1	0
0 1	0	1	0	0	0	0
0 1	0	1	1	0	1	1
1 0	1	0	X	1	0	0
1 1	1	1	X	1	1	0

Table 4.4: Input Logic Truth Table for Microprogram Sequencer

Boolean Function:

S0 = I0
S1 = I0I1 + I0’T
L = I0’I1T

- Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations.
- With three inputs, the sequencer can provide up to eight address sequencing operations.
- Some commercial sequencers have three or four inputs in addition to the T input and thus provide a wider range of operations.

Chapter – 2
Central Processing Unit

The part of the computer that performs the bulk of data processing operations is called the Central Processing Unit (CPU) and is the central component of a digital computer. Its purpose is to interpret instruction cycles received from memory and perform arithmetic, logic and control operations with data stored in internal register, memory words and I/O interface units. A CPU is usually divided into two parts namely processor unit (Register Unit and Arithmetic Logic Unit) and control unit.

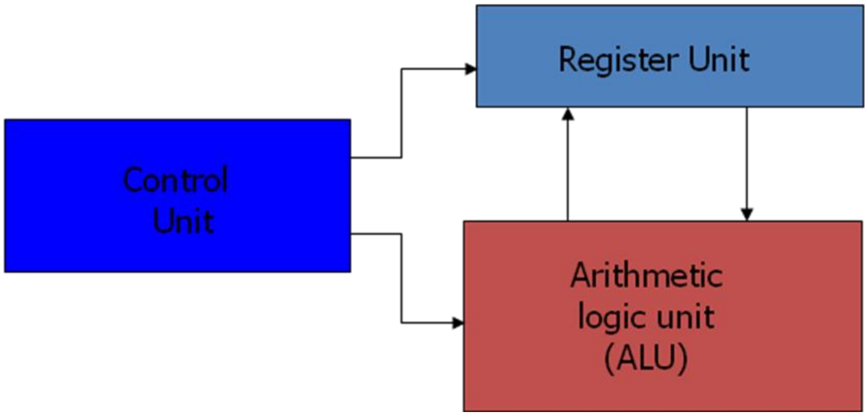


Fig: Components of CPU

Processor Unit:
The processor unit consists of arithmetic unit, logic unit, a number of registers and internal buses that provides data path for transfer of information between register and arithmetic logic unit. The block diagram of processor unit is shown in figure below where all registers are connected through common buses. The registers communicate each other not only for direct data transfer but also while performing various micro-operations.

Here two sets of multiplexers select register which perform input data for ALU. A decoder selects destination register by enabling its load input. The function select in ALU determines the particular operation that to be performed.

For an example to perform the operation: $R_3 \leftarrow R_1 + R_2$

1. MUX A selector (SELA): to place the content of R_1 into bus A.
2. MUX B selector (SELB): to place the content of R_2 into bus B.
3. ALU operation selector (OPR): to provide arithmetic addition A + B.
4. Decoder destination selector (SELD): to transfer the content of the output bus into R_3.

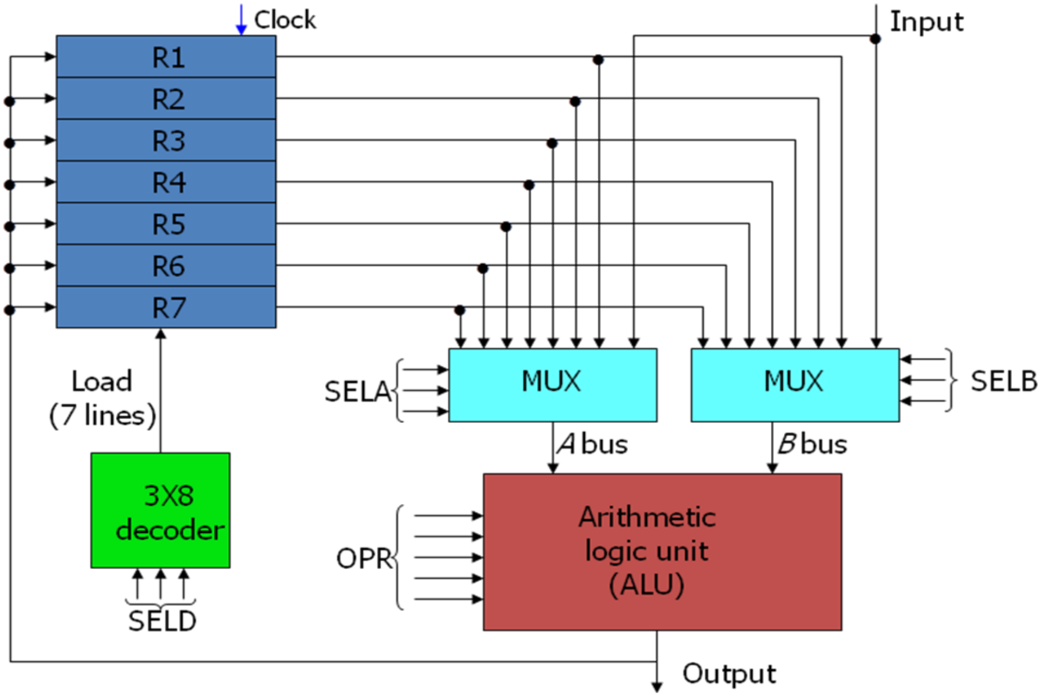


Fig: Processor Unit

Control unit:
The control unit is the heart of CPU. It consists of a program counter, instruction register, timing and control logic. The control logic may be either hardwired or micro-programmed. If it is a hardwired, register decodes and a set of gates are connected to provide the logic that determines the action required to execute various instructions. A micro-programmed control unit uses a control memory to store micro instructions and a sequence to determine the order by which the instructions are read from control memory.

The control unit decides what the instructions mean and directs the necessary data to be moved from memory to ALU. Control unit must communicate with both ALU and main memory and coordinates all activities of processor unit, peripheral devices and storage devices. It can be characterized on the basis of design and implementation by:

- Defining basic elements of the processor
- Describing the micro-operation that processor performs
- Determining the function that the control unit must perform to cause the micro-operations to be performed.

Control unit must have inputs that allow determining the state of system and outputs that allow controlling the behavior of system.

The input to control unit are:

- Flag: flags are headed to determine the status of processor and outcome of previous ALU operation.

- Clock: All micro-operations are performed within each clock pulse. This clock pulse is also called as processor cycle time or clock cycle time.
- Instruction Register: The op-code of instruction determines which micro-operation to perform during execution cycle.
- Control signal from control bus: The control bus portion of system bus provides interrupt, acknowledgement signals to control unit.

The outputs from control unit are:

- Control signal within processor: These signals causes data transfer between registers, activate ALU functions.
- Control signal to control bus: These are signals to memory and I/O module. All these control signals are applied directly as binary inputs to individual logic gate.

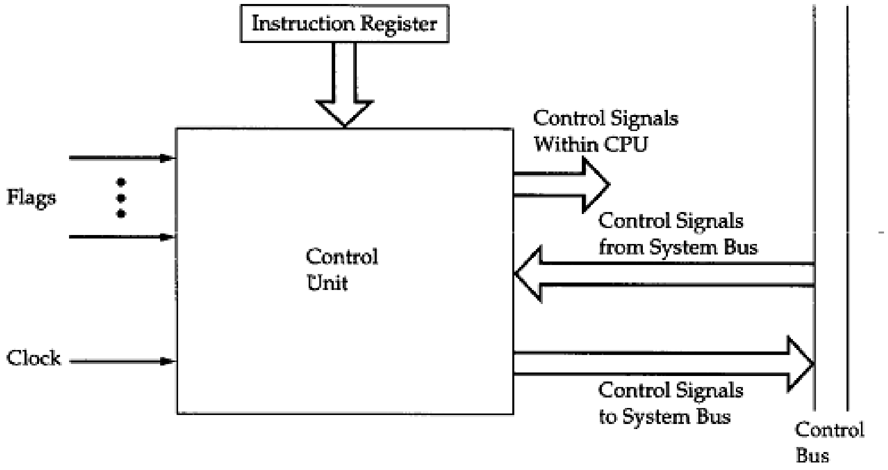

Instruction Register
Control Signals Within CPU
Flags
Control Signals from System Bus
Control Unit
Clock
Control Signals to System Bus
Control Bus

Fig: Control Unit

2.1 CPU Structure and Function

Processor Organization

- Things a CPU must do:
  - Fetch Instructions
  - Interpret Instructions
  - Fetch Data
  - Process Data
  - Write Data

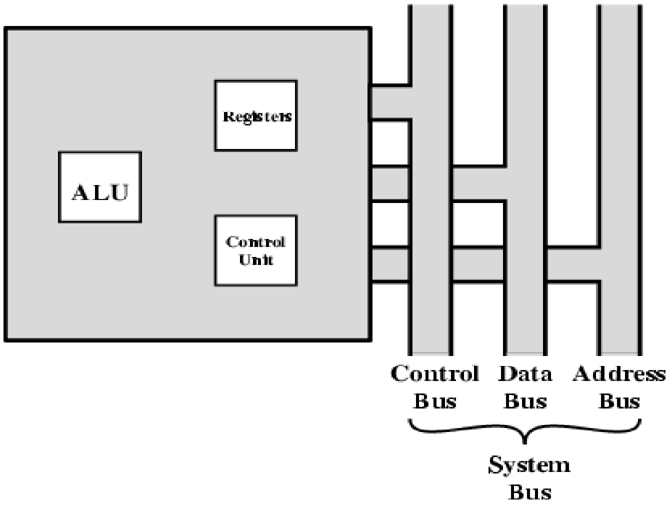


Fig: The CPU with the System Bus

- A small amount of internal memory, called the registers, is needed by the CPU to fulfill these requirements

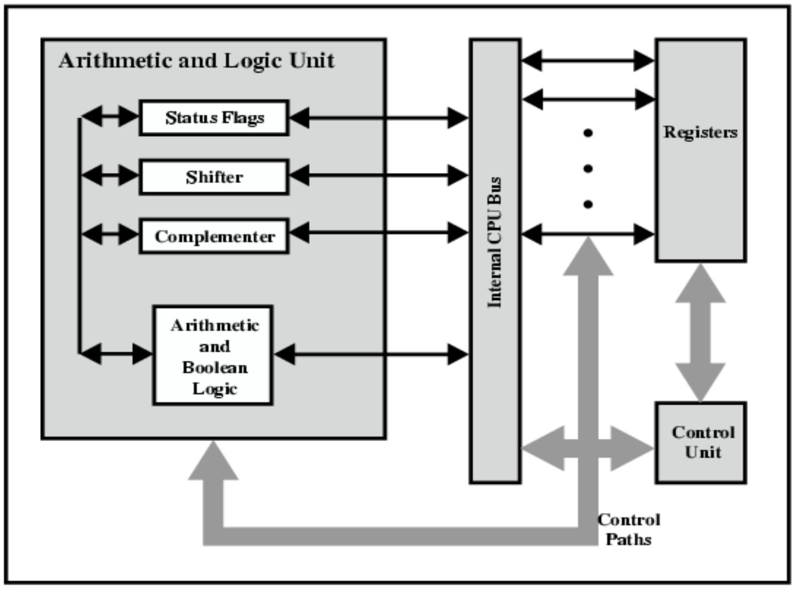


Fig: Internal Structure of the CPU

- Components of the CPU
  - Arithmetic and Logic Unit (ALU): does the actual computation or processing of data
  - Control Unit (CU): controls the movement of data and instructions into and out of the CPU and controls the operation of the ALU.

Register Organization

- Registers are at top of the memory hierarchy. They serve two functions:
  1. User-Visible Registers - enable the machine- or assembly-language programmer to minimize main-memory references by optimizing use of registers
  2. Control and Status Registers - used by the control unit to control the operation

of the CPU and by privileged, OS programs to control the execution of programs

User-Visible Registers

Categories of Use

- General Purpose registers - for variety of functions
- Data registers - hold data
- Address registers - hold address information
- Segment pointers - hold base address of the segment in use
- Index registers - used for indexed addressing and may be auto indexed
- Stack Pointer - a dedicated register that points to top of a stack. Push, pop, and other stack instructions need not contain an explicit stack operand.
- Condition Codes (flags)

Design Issues

- Completely general-purpose registers or specialized use?
  - Specialized registers save bits in instructions because their use can be implicit
  - General-purpose registers are more flexible
  - Trend is toward use of specialized registers
- Number of registers provided?
  - More registers require more operand specifier bits in instructions
  - 8 to 32 registers appears optimum (RISC systems use hundreds, but are a completely different approach)
- Register Length?
  - Address registers must be long enough to hold the largest address
  - Data registers should be able to hold values of most data types
  - Some machines allow two contiguous registers for double-length values
- Automatic or manual save of condition codes?
  - Condition restore is usually automatic upon call return
  - Saving condition code registers may be automatic upon call instruction, or may be manual

Control and Status Registers

- Essential to instruction execution
  - Program Counter (PC)
  - Instruction Register (IR)
  - Memory Address Register (MAR) - usually connected directly to address lines of bus
  - Memory Buffer Register (MBR) - usually connected directly to data lines of bus
- Program Status Word (PSW) - also essential, common fields or flags contained include:
  - Sign - sign bit of last arithmetic operation
  - Zero - set when result of last arithmetic operation is 0
  - Carry - set if last op resulted in a carry into or borrow out of a high-order bit
  - Equal - set if a logical compare result is equality
  - Overflow - set when last arithmetic operation caused overflow
  - Interrupt Enable/Disable - used to enable or disable interrupts
  - Supervisor - indicates if privileged ops can be used

- Other optional registers
  - Pointer to a block of memory containing additional status info (like process control blocks)
  - An interrupt vector
  - A system stack pointer
  - A page table pointer
  - I/O registers
- Design issues
  - Operating system support in CPU
  - How to divide allocation of control information between CPU registers and first part of main memory (usual tradeoffs apply)

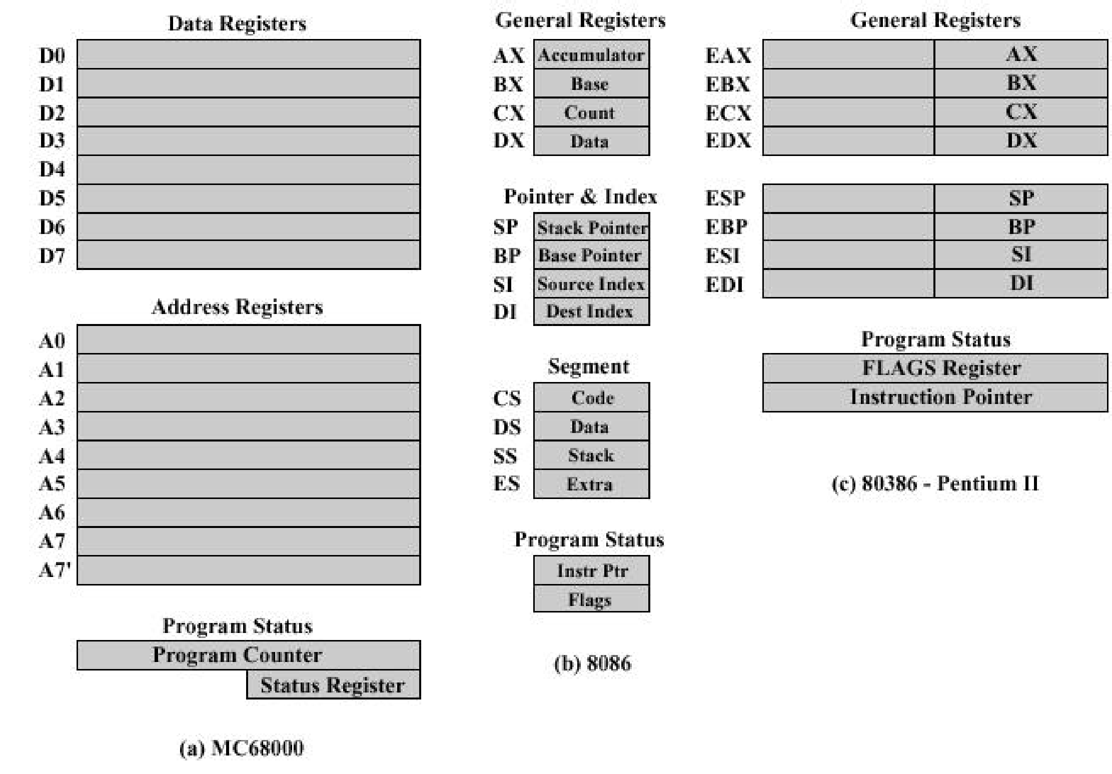


Fig: Example Microprocessor Register Organization

The Instruction Cycle

Basic instruction cycle contains the following sub-cycles.

- Fetch - read next instruction from memory into CPU
- Execute - Interpret the opcode and perform the indicated operation
- Interrupt - if interrupts are enabled and one has occurred, save the current process state and service the interrupt

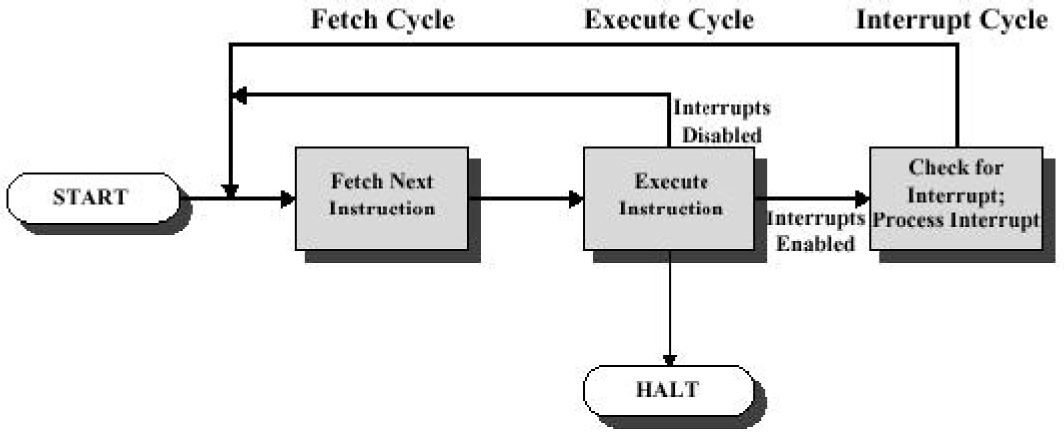


Fig: Instruction Cycles

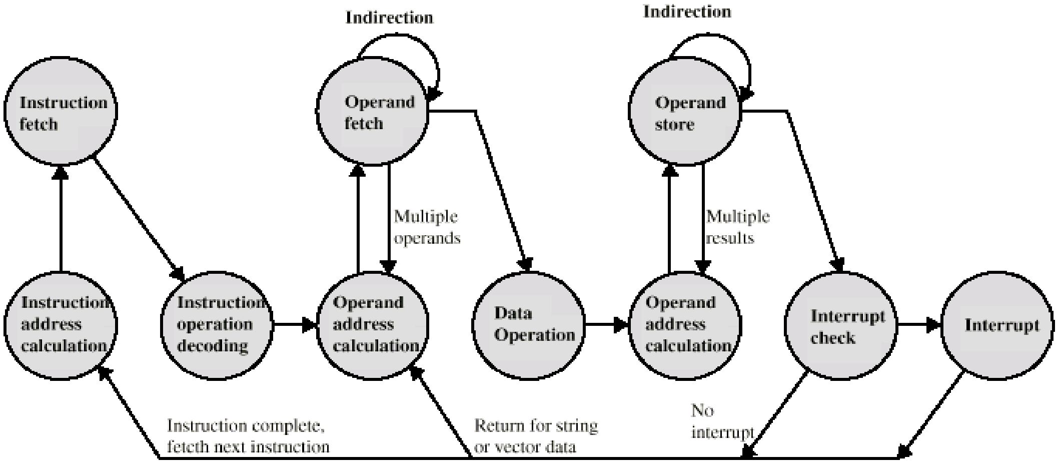

Indirection
Indirection
Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data Operation
Operand address calculation
Interrupt check
Interrupt
Instruction complete, fetcth next instruction
Return for string or vector data
No interrupt

Fig: Instruction Cycle State Diagram

The Indirect Cycle

- Think of as another instruction sub-cycle
- May require just another fetch (based upon last fetch)
- Might also require arithmetic, like indexing

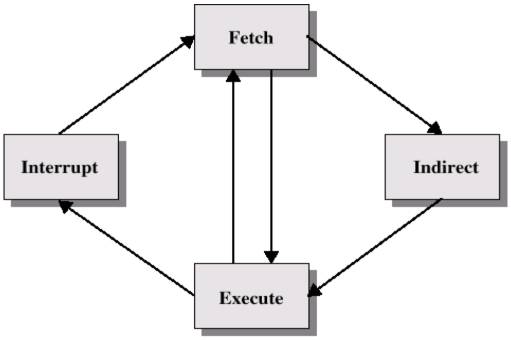


Fig: Instruction Cycle with Indirect

Data Flow

- Exact sequence depends on CPU design
- We can indicate sequence in general terms, assuming CPU employs:
  - a memory address register (MAR)
  - a memory buffer register (MBR)
  - a program counter (PC)
  - an instruction register (IR)

Fetch cycle data flow

- PC contains address of next instruction to be fetched
- This address is moved to MAR and placed on address bus
- Control unit requests a memory read
- Result is
  - placed on data bus
  - result copied to MBR
  - then moved to IR
- Meanwhile, PC is incremented

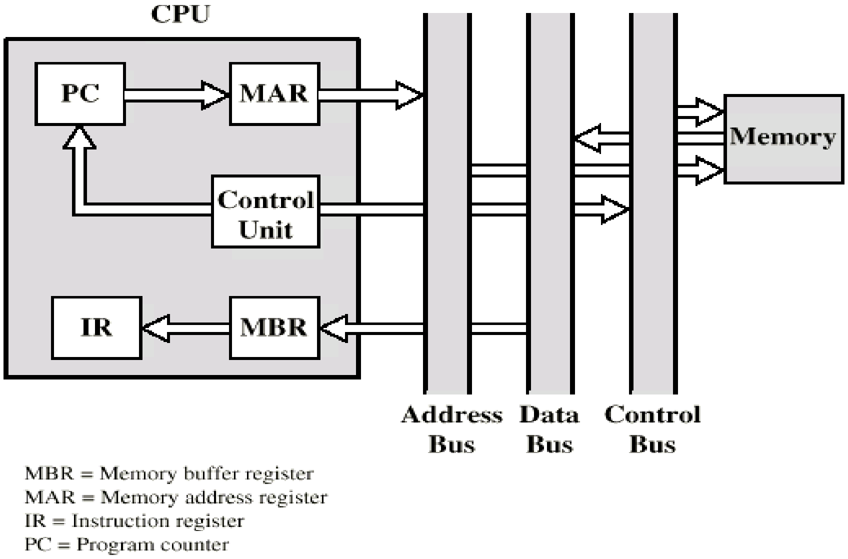

CPU
PC
MAR
Control Unit
IR
MBR
Memory
Address Bus
Data Bus
Control Bus
MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Fig: Data flow, Fetch Cycle

t1: MAR ← (PC)
t2: MBR ← Memory
PC ← PC + 1
t3: IR(Address) ← (MBR(Address))

Indirect cycle data flow

- Decodes the instruction
- After fetch, control unit examines IR to see if indirect addressing is being used. If so:
- Rightmost n bits of MBR (the memory reference) are transferred to MAR
- Control unit requests a memory read, to get the desired operand address into the MBR

t1: MAR ← (IR(Address))
t2: MBR ← Memory
t3: IR(Address) ← (MBR(Address))

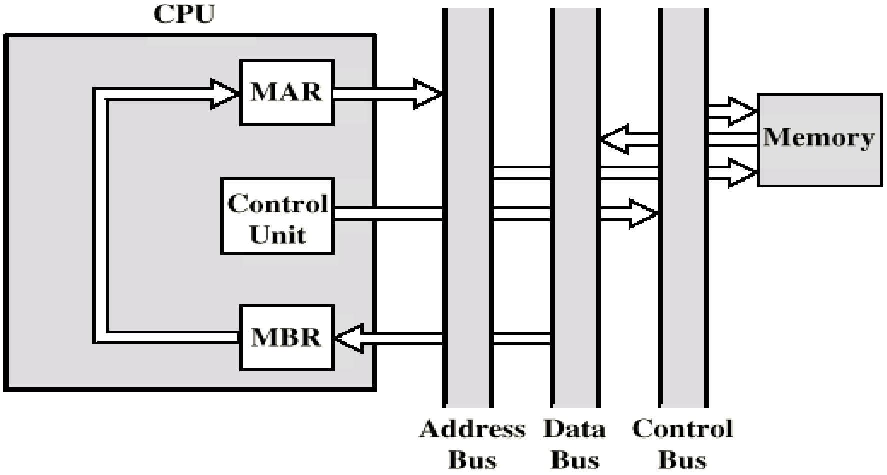


Fig: Data Flow, Indirect Cycle

Execute cycle data flow

- Not simple and predictable, like other cycles
- Takes many forms, since form depends on which of the various machine instructions is in the IR
- May involve
  - transferring data among registers
  - read or write from memory or I/O
  - invocation of the ALU

For example: ADD R_1, X
t1: MAR ← (IR(Address))
t2: MBR ← Memory
t3: R_1 ← (R_1) + (MBR)

Interrupt cycle data flow

- Current contents of PC must be saved (for resume after interrupt), so PC is transferred to MBR to be written to memory
- Save location's address (such as a stack ptr) is loaded into MAR from the control unit
- PC is loaded with address of interrupt routine (so next instruction cycle will begin by fetching appropriate instruction)

t1: MBR ← (PC)
t2: MAR ← save_address
PC ← Routine_address
t3: Memory ← (MBR)

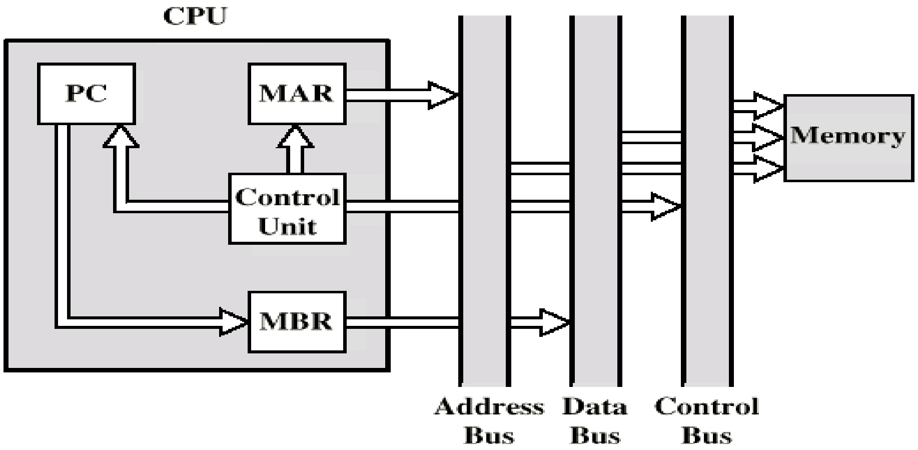
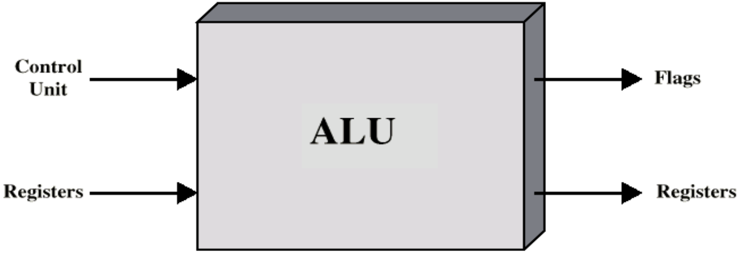


Fig: Data Flow, Interrupt Cycle

2.2 Arithmetic and Logic Unit

ALU is the combinational circuit of that part of computer that actually performs arithmetic and logical operations on data. All of the other elements of computer system- control unit, registers, memory, I/O are their mainly to bring data into the ALU for it to process and then to take the result back out. An ALU & indeed all electronic components in computer are based on the use of simple digital logic device that can store binary digit and perform simple Boolean logic function. Figure indicates in general in general term how ALU is interconnected with rest of the processor.



Data are presented to ALU in register and the result of operation is stored in register. These registers are temporarily storage location within the processor that are connected by signal path to the ALU. The ALU may also set flags as the result of an operation. The flags values are also stored in registers within the processor. The control unit provides signals that control the operation of ALU and the movement of data into an out of ALU.

The design of ALU has three stages.

1. **Design the arithmetic section**
   The basic component of arithmetic circuit is a parallel adder which is constructed with a number of full adder circuits connected in cascade. By controlling the data inputs to the parallel adder, it is possible to obtain different types of arithmetic operations. Below figure shows the arithmetic circuit and its functional table.

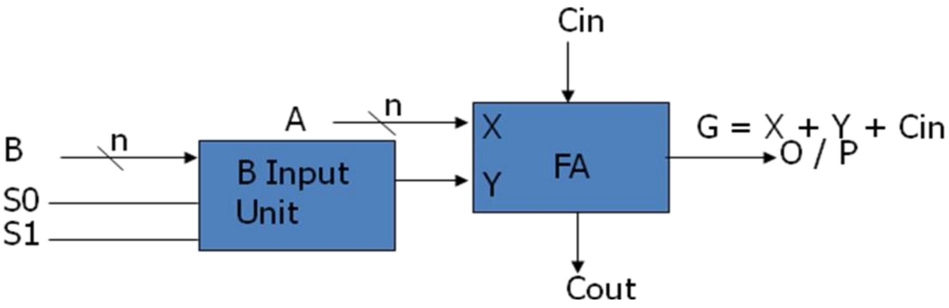


Fig: Block diagram of Arithmetic Unit

Functional table for arithmetic unit:

Select		Input	Output		Microoperation	
S_1	S_0	Y	Cin = 0	Cin = 1	Cin = 0	Cin = 1
0	0	0	A	A+1	Transfer A	Increment A
0	1	B	A+B	A+B+1	Addition	Addition with carry
1	0	B’	A+B’	A+B’+1	Subtraction with borrow	Subtraction
1	1	-1	A-1	A	Decrement A	Transfer A

2. **Design the logical section**

The basic components of logical circuit are AND, OR, XOR and NOT gate circuits connected accordingly. Below figure shows a circuit that generates four basic logic micro-operations. It consists of four gates and a multiplexer. Each of four logic operations is generated through a gate that performs the required logic. The two selection input S1 and S0 choose one of the data inputs of the multiplexer and directs its value to the output. Functional table lists the logic operations.

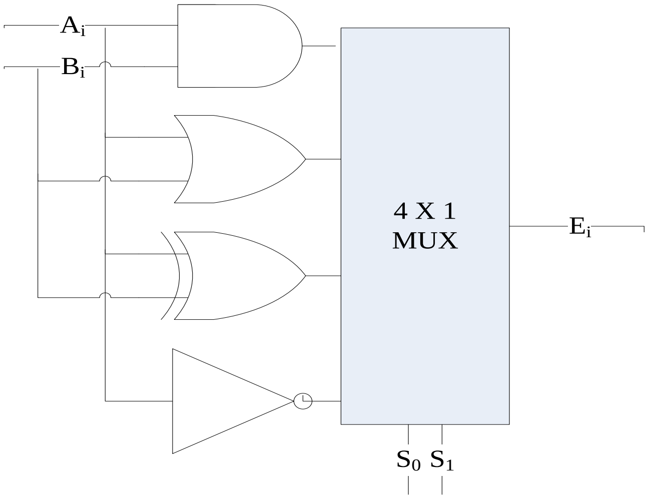


Fig: Block diagram of Logic Unit

Functional table for logic unit:

S1	S0	output	Microoperation
0	0	Ai && Bi	AND
0	1	Ai || Bi	OR
1	0	Ai XOR Bi	XOR
1	1	Ai’	NOT

3. **Combine these 2 sections to form the ALU**

Below figure shows a combined circuit of ALU where n data input from A are combined with n data input from B to generate the result of an operation at the G output line. ALU has a number of selection lines used to determine the operation to be performed. The selection lines are decoded with the ALU so that selection lines can specify distinct operations. The mode select S_2 differentiate between arithmetic and logical operations. The two functions select S_1 and S_0 specify the particular arithmetic and logic operations to be performed. With three selection lines, it is possible to specify arithmetic operation with S_2 at 0 and logical operation with S_2 at 1.

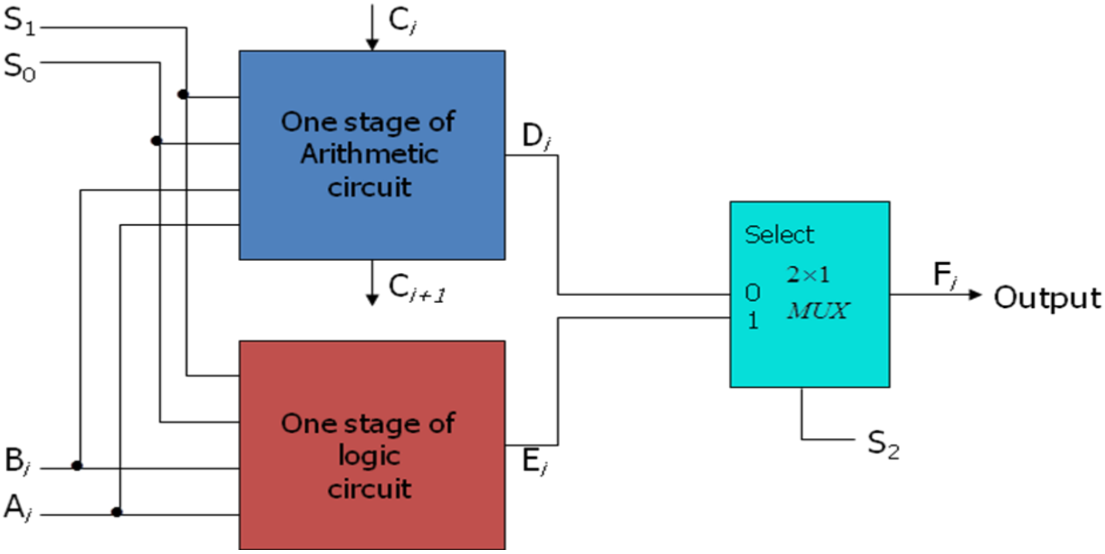
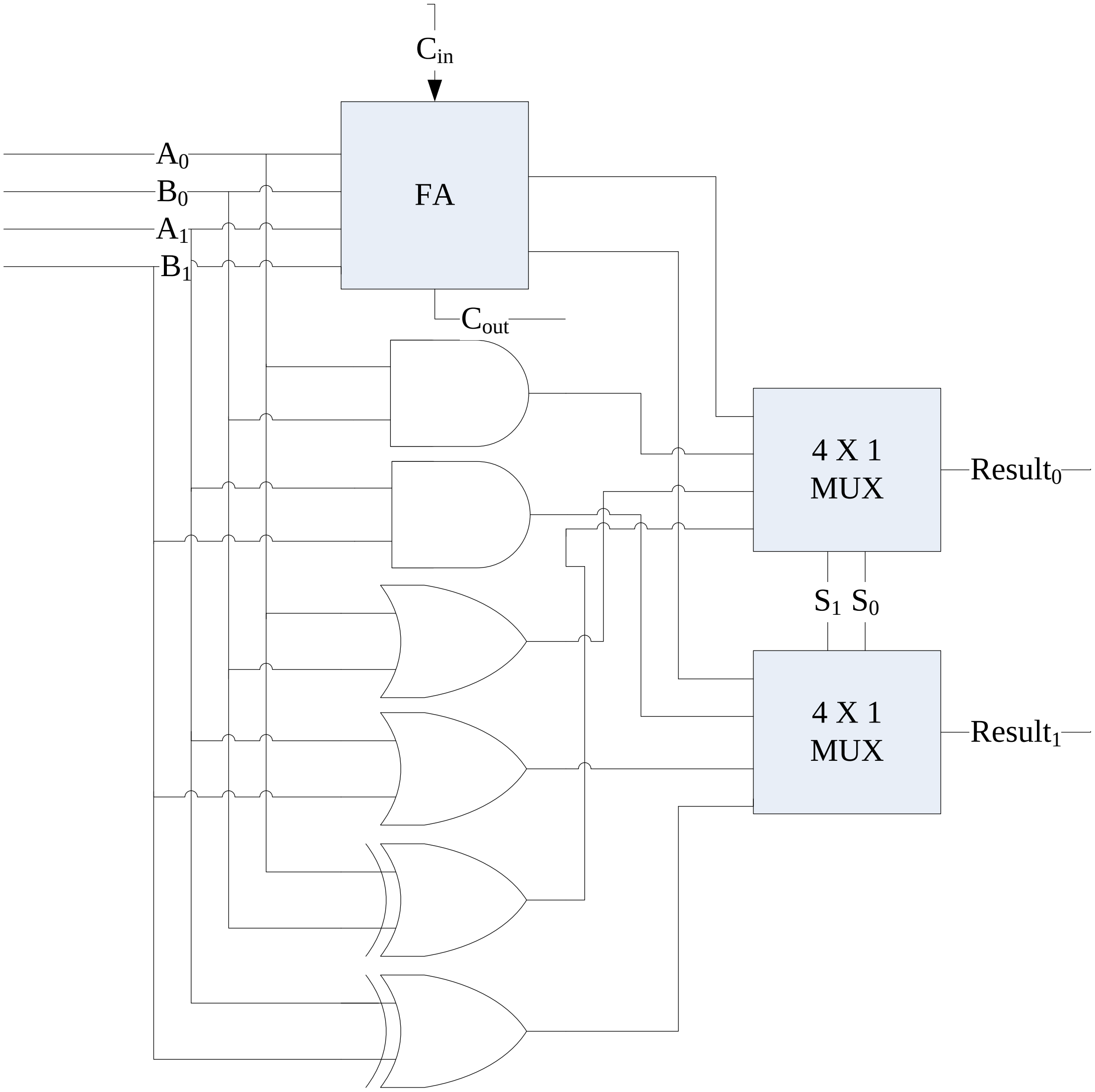


Fig: Block diagram of ALU

Example: Design a 2-bit ALU that can perform addition, AND, OR, & XOR.



2.3 Instruction Formats

The computer can be used to perform a specific task, only by specifying the necessary steps to complete the task. The collection of such ordered steps forms a ‘program’ of a computer. These ordered steps are the instructions. Computer instructions are stored in central memory locations and are executed sequentially one at a time. The control reads an instruction from a specific address in memory and executes it. It then continues by reading the next instruction in sequence and executes it until the completion of the program.

A computer usually has a variety of Instruction Code Formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction. An n bit instruction that k bits in the address field and m bits in the operation code field come addressed 2^k location directly and specify 2^m different operation.

- The bits of the instruction are divided into groups called fields.
- The most common fields in instruction formats are:
  - An **Operation code** field that specifies the operation to be performed.
  - An **Address field** that designates a memory address or a processor register.
  - A **Mode field** that specifies the way the operand or the effective address is determined.

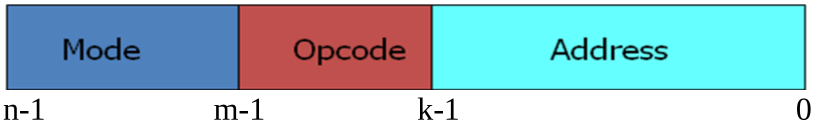


Fig: Instruction format with mode field

The operation code field (Opcode) of an instruction is a group of bits that define various processor operations such as add, subtract, complement, shift etcetera. The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address. Operation specified by an instruction is executed on some data stored in the processor register or in the memory location. Operands residing in memory are specified by their memory address. Operands residing in processor register are specified with a register address.

Types of Instruction

- Computers may have instructions of several different lengths containing varying number of addresses.
- The number of address fields in the instruction format of a computer depends on the internal organization of its registers.
- Most computers fall into one of 3 types of CPU organizations:

Single accumulator organization:- All the operations are performed with an accumulator register. The instruction format in this type of computer uses one address field. For example: ADD X, where X is the address of the operands .

General register organization:- The instruction format in this type of computer needs three register address fields. For example: ADD R1,R2,R3

Stack organization:- The instruction in a stack computer consists of an operation code with no address field. This operation has the effect of popping the 2 top numbers from the stack, operating the numbers and pushing the sum into the stack. For example: ADD

Computers may have instructions of several different lengths containing varying number of addresses. Following are the types of instructions.

1. **Three address Instruction**

   With this type of instruction, each instruction specifies two operand location and a result location. A temporary location T is used to store some intermediate result so as not to alter any of the operand location. The three address instruction format requires a very complex design to hold the three address references.

   Format: Op X, Y, Z; X ← Y Op Z

Example: ADD X, Y, Z; X ← Y + Z

- ADVANTAGE: It results in short programs when evaluating arithmetic expressions.
- DISADVANTAGE: The instructions requires too many bits to specify 3 addresses.

2. **Two address instruction**
Two-address instructions are the most common in commercial computers. Here again each address field can specify either a processor register, or a memory word. One address must do double duty as both operand and result. The two address instruction format reduces the space requirement. To avoid altering the value of an operand, a MOV instruction is used to move one of the values to a result or temporary location T, before performing the operation.
Format: Op X, Y; X ← X Op Y
Example: SUB X, Y; X ← X - Y

3. **One address Instruction**
It was generally used in earlier machine with the implied address been a CPU register known as accumulator. The accumulator contains one of the operand and is used to store the result. One-address instruction uses an implied accumulator (Ac) register for all data manipulation. All operations are done between the AC register and a memory operand. We use LOAD and STORE instruction for transfer to and from memory and Ac register.
Format: Op X; Ac ← Ac Op X
Example: MUL X; Ac ← Ac * X

4. **Zero address Instruction**
It does not use address field for the instruction like ADD, SUB, MUL, DIV etc. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The name "Zero" address is given because of the absence of an address field in the computational instruction.
Format: Op; TOS ← TOS Op (TOS – 1)
Example: DIV; TOS ← TOS DIV (TOS – 1)

Example: To illustrate the influence of the number of address on computer programs, we will evaluate the arithmetic statement X=(A+B)*(C+D) using Zero, one, two, or three address instructions.

1. Three-Address Instructions:
ADD R1, A, B; R1 ← M[A] + M[B]
ADD R2, C, D; R2 ← M[C] + M[D]
MUL X, R1,R2; M[X] ← R1 * R2

It is assumed that the computer has two processor registers R1 and R2. The symbol M[A] denotes the operand at memory address symbolized by A.

2. Two-Address Instructions:
MOV R1, A; R1 ← M[A]
ADD R1, B; R1 ← R1 + M[B]

```
MOV R2, C;   R2 ← M[C]
ADD R2, D;   R2 ← R2 + M[D]
MUL R1, R2;  R1 ← R1 * R2
MOV X, R1;   M[X] ← R1
```

3. One-Address Instruction:

```
LOAD A;      Ac ← M[A]
ADD B;       Ac ← Ac + M[B]
STORE T;     M[T] ← Ac
LOAD C;      Ac ← M[C]
ADD D;       Ac ← Ac + M[D]
MUL T;       Ac ← Ac * M[T]
STORE X;     M[X] ← Ac
```

Here, T is the temporary memory location required for storing the intermediate result.

4. Zero-Address Instructions:

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

2.4 Addressing Modes

- Specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.
- Computers use addressing mode techniques for the purpose of accommodating the following purposes:-
  - To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data and various other purposes.
  - To reduce the number of bits in the addressing field of the instructions.
    - Other computers use a single binary for operation & Address mode.
    - The mode field is used to locate the operand.
    - Address field may designate a memory address or a processor register.
    - There are 2 modes that need no address field at all (Implied & immediate modes).

Effective address (EA):

- The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode.
- The effective address is the address of the operand in a computational-type instruction.

The most well known addressing mode are:

- Implied Addressing Mode.
- Immediate Addressing Mode
- Register Addressing Mode
- Register Indirect Addressing Mode
- Auto-increment or Auto-decrement Addressing Mode
- Direct Addressing Mode
- Indirect Addressing Mode
- Displacement Address Addressing Mode
- Relative Addressing Mode
- Index Addressing Mode
- Stack Addressing Mode

Implied Addressing Mode:

- In this mode the operands are specified implicitly in the definition of the instruction.
  For example:- CMA - "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied-mode instructions.

Instruction
Opcode

Advantage: no memory reference. Disadvantage: limited operand

Immediate Addressing mode:

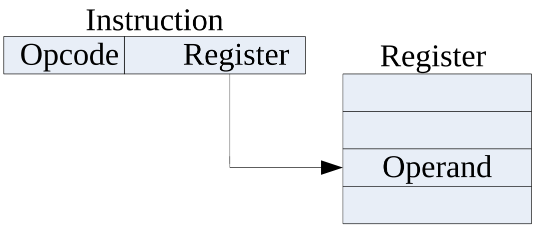
- In this mode the operand is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address field.
- This instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction.
- These instructions are useful for initializing register to a constant value;
  For example MVI B, 50H

Instruction	
Opcode	Operand

It was mentioned previously that the address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in register-mode.
Advantage: no memory reference. Disadvantage: limited operand

Register direct addressing mode:

- In this mode, the operands are in registers that reside within the CPU.
- The particular register is selected from the register field in the instruction.
  For example MOV A, B



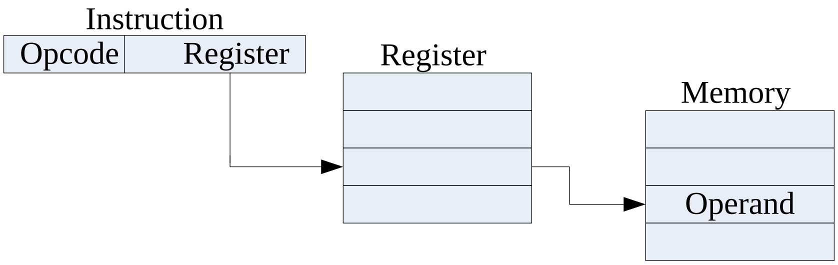
Effective Address (EA) = R

Advantage: no memory reference. Disadvantage: limited address space

Register indirect addressing mode:

- In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in the memory.
- In other words, the selected register contains the address of the operand rather than the operand itself.
- Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction.

  For example LDAX B



Effective Address (EA) = (R)

Advantage: Large address space.

The address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

Disadvantage: Extra memory reference

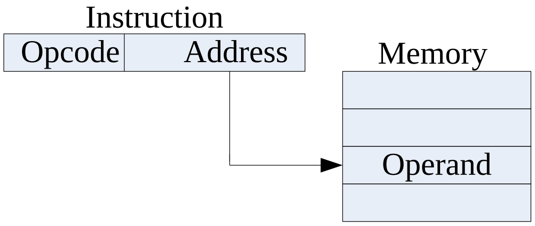
Auto increment or Auto decrement Addressing Mode:

- This is similar to register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.
- When the address stored in the registers refers to a table of data in memory, it is necessary to increment or decrement the registers after every access to the table.
- This can be achieved by using the increment or decrement instruction. In some computers it is automatically accessed.
- The address field of an instruction is used by the control unit in the CPU to obtain the operands from memory.
- Sometimes the value given in the address field is the address of the operand, but sometimes it is the address from which the address has to be calculated.

Direct Addressing Mode

- In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction.

  For example LDA 4000H

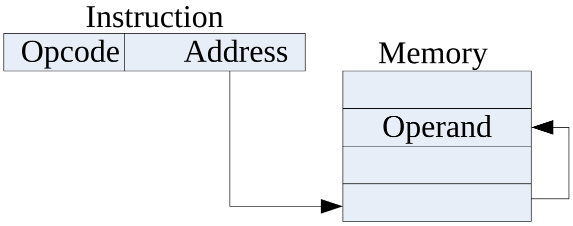


Effective Address (EA) = A

Advantage: Simple. Disadvantage: limited address field

Indirect Addressing Mode

- In this mode the address field of the instruction gives the address where the effective address is stored in memory.
- Control unit fetches the instruction from the memory and uses its address part to access memory again to read the effective address.

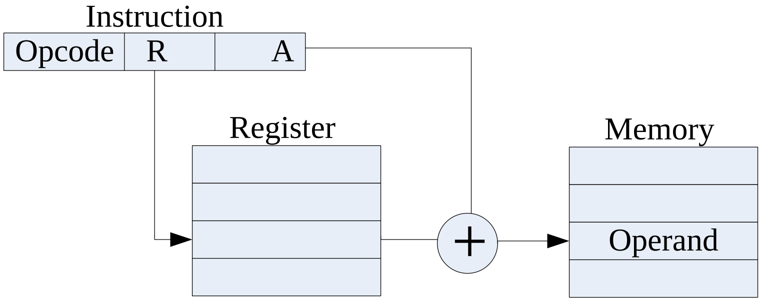


Effective Address (EA) = (A)

Advantage: Flexibility. Disadvantage: Complexity

Displacement Addressing Mode

- A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing.
- The address field of instruction is added to the content of specific register in the CPU.



Effective Address (EA) = A + (R)

Advantage: Flexibility. Disadvantage: Complexity

Relative Addressing Mode

- In this mode the content of the program counter (PC) is added to the address part of the instruction in order to obtain the effective address.
- The address part of the instruction is usually a signed number (either a +ve or a –ve number).
- When the number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.
  Effective Address (EA) = PC + A

Indexed Addressing Mode

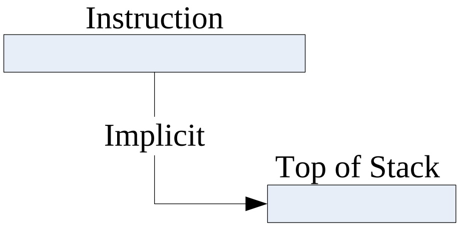
- In this mode the content of an index register (XR) is added to the address part of the instruction to obtain the effective address.
- The index register is a special CPU register that contains an index value.
- Note: If an index-type instruction does not include an address field in its format, the instruction is automatically converted to the register indirect mode of operation.
  Effective Address (EA) = XR + A

Base Register Addressing Mode

- In this mode the content of a base register (BR) is added to the address part of the instruction to obtain the effective address.
- This is similar to the indexed addressing mode except that the register is now called a base register instead of the index register.
- The base register addressing mode is used in computers to facilitate the relocation of programs in memory i.e. when programs and data are moved from one segment of memory to another.
  Effective Address (EA) = BR + A

Stack Addressing Mode

- The stack is the linear array of locations. It is some times referred to as push down list or last in First out (LIFO) queue. The stack pointer is maintained in register.



Effective Address (EA) = TOS

Let us try to evaluate the addressing modes with as example.

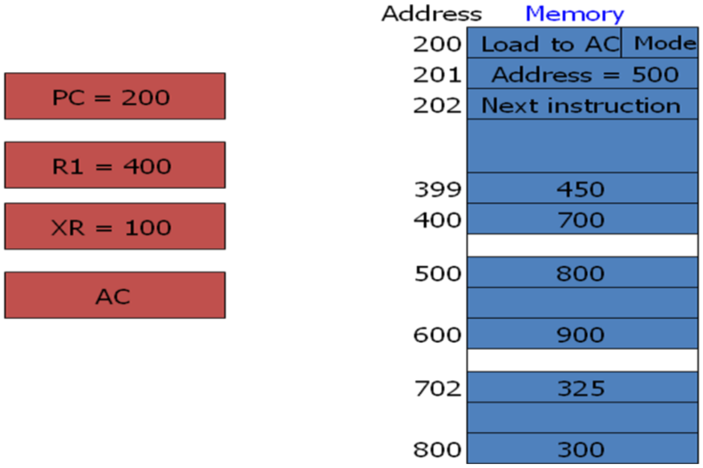
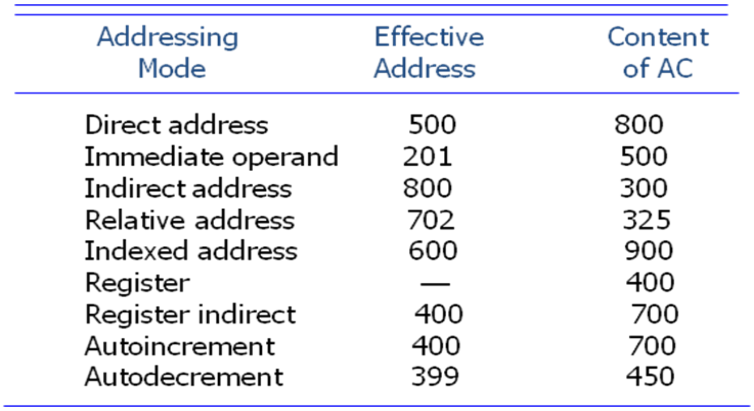


Fig: Numerical Example for Addressing Modes



Addressing Mode	Effective Address	Content of AC
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

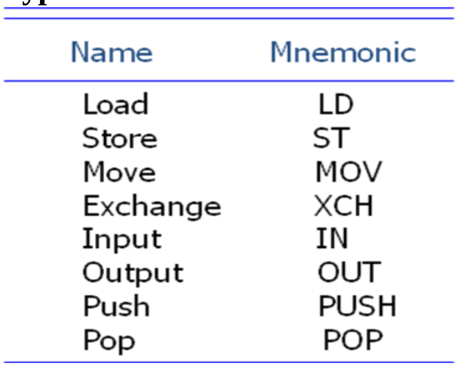
Fig: Tabular list of Numerical Example

2.5 Data Transfer and Manipulation

Data transfer instructions cause transfer of data from one location to another without changing the binary information. The most common transfer are between the

- Memory and Processor registers
- Processor registers and input output devices
- Processor registers themselves

Typical Data Transfer Instructions

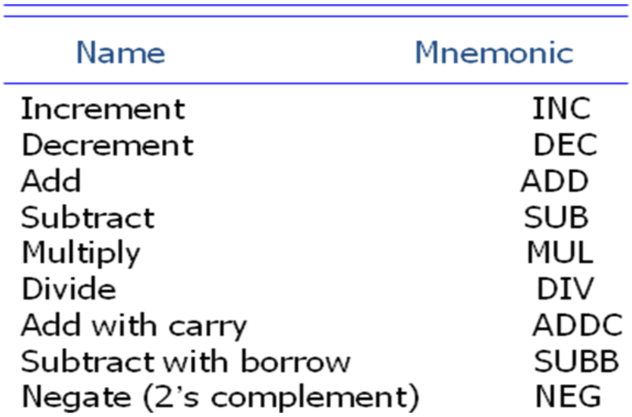


Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Data manipulation Instructions

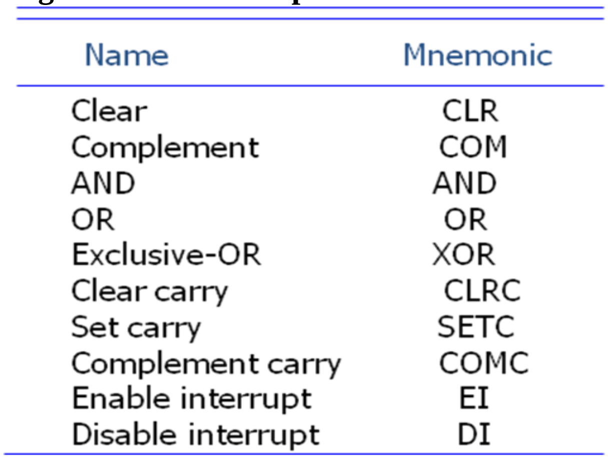
Data manipulation instructions perform operations on data and provide the computational capabilities for the computer. These instructions perform arithmetic, logic and shift operations.

Arithmetic Instructions



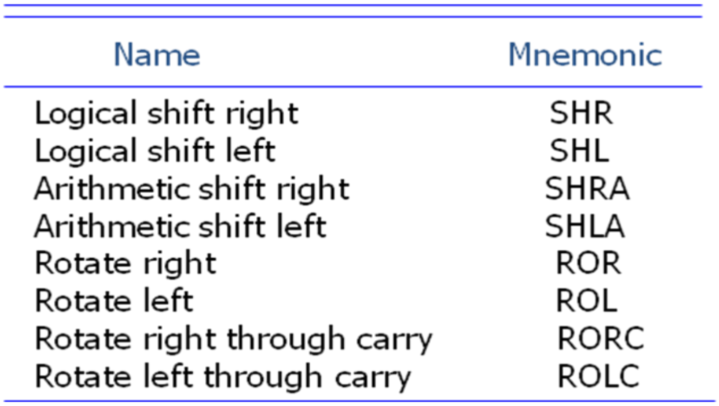
Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

Logical and Bit Manipulation Instructions



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

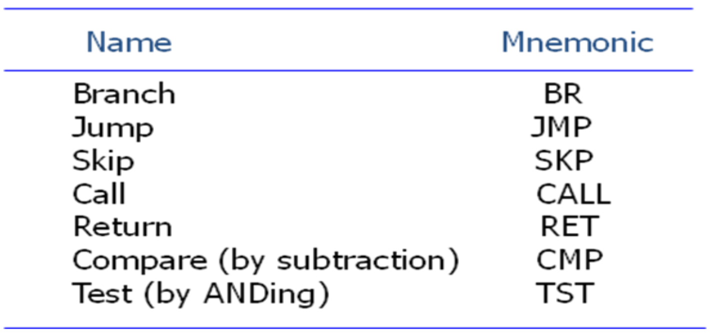
Shift Instructions



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

Program Control Instructions

The program control instructions provide decision making capabilities and change the path taken by the program when executed in computer. These instructions specify conditions for altering the content of the program counter. The change in value of program counter as a result of execution of program control instruction causes a break in sequence of instruction execution. Some typical program control instructions are:



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

Subroutine call and Return

A subroutine call instruction consists of an operation code together with an address that specifies the beginning of the subroutine. The instruction is executed by performing two tasks:

- The address of the next instruction available in the program counter (the return address) is stored in a temporary location (stack) so the subroutine knows where to return.
- Control is transferred to the beginning of the subroutine.

The last instruction of every subroutine, commonly called return from subroutine; transfer the return address from the temporary location into the program counter. This results in a transfer of program control to the instruction where address was originally stored in the temporary location.

Interrupt

The interrupt procedure is, in principle, quite similar to a subroutine call except for three variations:

- The interrupt is usually initiated by an external or internal signal rather than from execution of an instruction.
- The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction.
- An interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter.

UNIT – III

Data Representation: Data types, Complements, Fixed Point Representation, Floating Point Representation. **Computer Arithmetic:** Addition and subtraction, multiplication Algorithms, Division Algorithms, Floating – point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations.

Data Representation:

Digital computers store and process information in binary form as digital logic has only two values "1" and "0" or in other words "True or False" or also said as "ON or OFF". This system is called radix 2. We human generally deal with radix 10 i.e. decimal. As a matter of convenience there are many other representations like Octal (Radix 8), Hexadecimal (Radix 16), Binary coded decimal (BCD), Decimal etc.

Every computer's CPU has a width measured in terms of bits such as 8 bit CPU, 16 bit CPU, 32 bit CPU etc. Similarly, each memory location can store a fixed number of bits and is called memory width. Given the size of the CPU and Memory, it is for the programmer to handle his data representation. Most of the readers may be knowing that 4 bits form a Nibble, 8 bits form a byte. The word length is defined by the Instruction Set Architecture of the CPU. The word length may be equal to the width of the CPU.

The memory simply stores information as a binary pattern of 1's and 0's. It is to be interpreted as what the content of a memory location means. If the CPU is in the Fetch cycle, it interprets the fetched memory content to be instruction and decodes based on Instruction format. In the Execute cycle, the information from memory is considered as data. As a common man using a computer, we think computers handle English or other alphabets, special characters or numbers. A programmer considers memory content to be data types of the programming language he uses. Now recall figure 1.2 and 1.3 of chapter 1 to reinforce your thought that conversion happens from computer user interface to internal representation and storage.

Data Representation in Computers

Information handled by a computer is classified as instruction and data. A broad overview of the internal representation of the information is illustrated in figure 3.1. No matter whether it is data in a numeric or non-numeric form or integer, everything is internally represented in Binary. It is up to the programmer to handle the interpretation of the binary pattern and this interpretation is called *Data Representation*. These data representation schemes are all standardized by international organizations.

Choice of Data representation to be used in a computer is decided by

- The number types to be represented (integer, real, signed, unsigned, etc.)
- Range of values likely to be represented (maximum and minimum to be represented)

- The Precision of the numbers i.e. maximum accuracy of representation (floating point single precision, double precision etc)
- If non-numeric i.e. character, character representation standard to be chosen. ASCII, EBCDIC, UTF are examples of character representation standards.
- The hardware support in terms of word width, instruction.

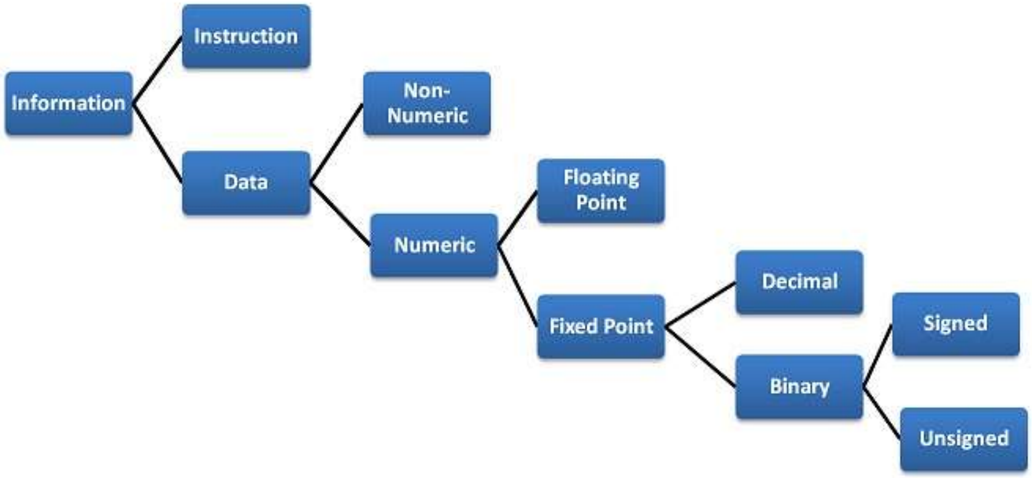


Figure.3.1 Typical Internal data representation types

Before we go into the details, let us take an example of interpretation. Say a byte in Memory has value "0011 0001". Although there exists a possibility of so many interpretations as in figure 3.2, the program has only one interpretation as decided by the programmer and declared in the program.

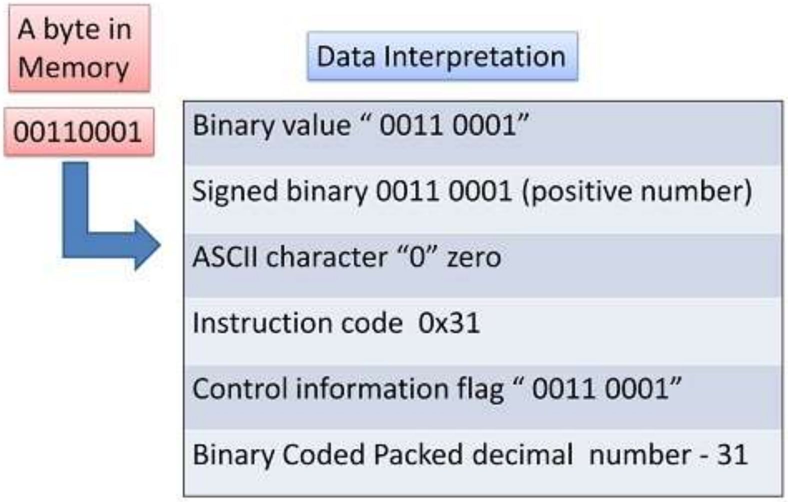

A byte in Memory
00110001
Data Interpretation
Binary value " 0011 0001"
Signed binary 0011 0001 (positive number)
ASCII character "0" zero
Instruction code 0x31
Control information flag " 0011 0001"
Binary Coded Packed decimal number - 31

Figure.3.2 Data Interpretation

Fixed point Number Representation

Fixed point numbers are also known as whole numbers or Integers. The number of bits used in representing the integer also implies the maximum number that can be represented in the system hardware. However for the efficiency of storage and operations, one may choose to represent the integer with one Byte, two Bytes, Four bytes or more. This space allocation is translated from the definition used by the programmer while defining a variable as integer short or long and the Instruction Set Architecture.

In addition to the bit length definition for integers, we also have a choice to represent them as below:

- **Unsigned Integer**: A positive number including zero can be represented in this format. All the allotted bits are utilised in defining the number. So if one is using 8 bits to represent the unsigned integer, the range of values that can be represented is 28 i.e. "0" to "255". If 16 bits are used for representing then the range is 216 i.e. "0 to 65535".
- **Signed Integer**: In this format negative numbers, zero, and positive numbers can be represented. A sign bit indicates the magnitude direction as positive or negative. There are three possible representations for signed integer and these are **Sign Magnitude format, 1's Compliment format and 2's Complement format**.

Signed Integer – Sign Magnitude format: Most Significant Bit (MSB) is reserved for indicating the direction of the magnitude (value). A "0" on MSB means a positive number and a "1" on MSB means a negative number. If n bits are used for representation, n-1 bits indicate the absolute value of the number. Examples for n=8:

Examples for n=8:

0010 1111 = + 47 Decimal (Positive number)

1010 1111 = - 47 Decimal (Negative Number)

0111 1110 = +126 (Positive number)

1111 1110 = -126 (Negative Number)

0000 0000 = + 0 (Postive Number)

1000 0000 = - 0 (Negative Number)

Although this method is easy to understand, Sign Magnitude representation has several shortcomings like

- Zero can be represented in two ways causing redundancy and confusion.
- The total range for magnitude representation is limited to 2n-1, although n bits were accounted.
- The separate sign bit makes the addition and subtraction more complicated. Also, comparing two numbers is not straightforward.

Signed Integer – 1's Complement format: In this format too, MSB is reserved as the sign bit. But the difference is in representing the Magnitude part of the value for negative numbers (magnitude) is inversed and hence called 1's Complement form. The positive numbers are represented as it is in binary. Let us see some examples to better our understanding.

Examples for n=8:

0010 1111 = + 47 Decimal (Positive number)

1101 0000 = - 47 Decimal (Negative Number)

0111 1110 = +126 (Positive number)

1000 0001 = -126 (Negative Number)

0000 0000 = + 0 (Postive Number)

1111 1111 = - 0 (Negative Number)

Converting a given binary number to its 2's complement form

Step 1. -x = x' + 1 where x' is the one's complement of x.

Step 2 Extend the data width of the number, fill up with sign extension i.e. MSB bit is used to fill the bits.

Example: **-47 decimal over 8bit representation**

Binary equivalent of + 47 is	0010 1111
Binary equivalent of - 47 is	1010 1111 (Sign Magnitude Form)
1's complement equivalent is	1101 0000
2's complement equivalent is	1101 0001

As you can see zero is not getting represented with redundancy. There is only one way of representing zero. The other problem of the complexity of the arithmetic operation is also eliminated in 2's complement representation. Subtraction is done as Addition.

More exercises on number conversion are left to the self-interest of readers.

Floating Point Number system

The maximum number at best represented as a whole number is 2^n. In the Scientific world, we do come across numbers like Mass of an Electron is 9.10939 x 10-31 Kg. Velocity of light is 2.99792458 x 108 m/s. Imagine to write the number in a piece of paper without exponent and converting into binary for computer representation. Sure you are tired!!. It makes no sense to write a number in non- readable form or non- processible form. Hence we write such large or small numbers using exponent and mantissa. This is said to be Floating Point representation or real number representation. he real number system could have infinite values between 0 and 1.

Representation in computer

Unlike the two's complement representation for integer numbers, Floating Point number uses Sign and Magnitude representation for both *mantissa* and *exponent*. In the number 9.10939 x 1031, in decimal form, +31 is Exponent, 9.10939 is known as *Fraction*. Mantissa, Significand and fraction are synonymously used terms. In the computer, the representation is binary and the binary point is not fixed. For example, a number, say, 23.345 can be written as 2.3345 x 101 or 0.23345 x 102 or 2334.5 x 10-2. The representation 2.3345 x 101 is said to be in normalised form.

Floating-point numbers usually use multiple words in memory as we need to allot a sign bit, few bits for exponent and many bits for mantissa. There are standards for such allocation which we will see sooner.

IEEE 754 Floating Point Representation

We have two standards known as Single Precision and Double Precision from IEEE. These standards enable portability among different computers. Figure 3.3 picturizes Single precision while figure 3.4 picturizes double precision. Single Precision uses 32bit format while double precision is 64 bits word length. As the name suggests double precision can represent fractions with larger accuracy. In both the cases, MSB is sign bit for the mantissa part, followed by Exponent and Mantissa. The exponent part has its sign bit.

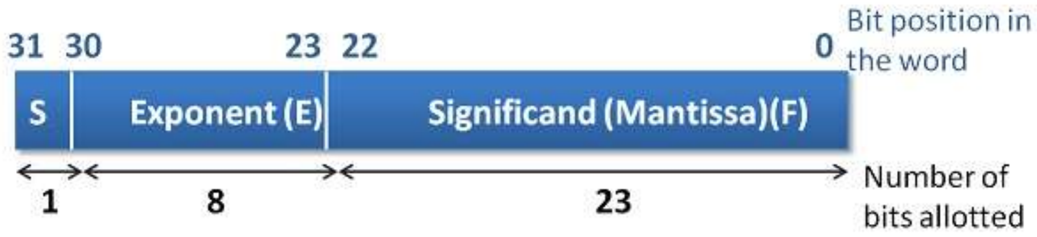


Figure.3.3 IEEE 754 Single Precision Floating Point representation Standard

It is to be noted that in Single Precision, we can represent an exponent in the range -127 to +127. It is possible as a result of arithmetic operations the resulting exponent may not fit in. This situation is called ***overflow*** in the case of positive exponent and ***underflow*** in the case of negative exponent. The Double Precision format has 11 bits for exponent meaning a number as large as -1023 to 1023 can be represented. The programmer has to make a choice between Single Precision and Double Precision declaration using his knowledge about the data being handled.

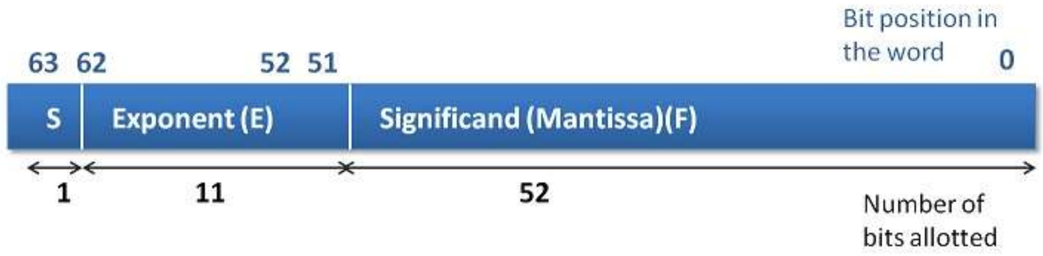


Figure 3.4 IEEE 754 Double Precision Floating Point representation Standard

The Floating Point operations on the regular CPU is very very slow. Generally, a special purpose CPU known as Co-processor is used. This Co-processor works in tandem with the main CPU. The programmer should be using the float declaration only if his data is in real number form. Float declaration is not to be used generously.

UNIT-III

COMPUTER ARITHMETIC: Addition and subtraction, multiplication algorithms, Division Algorithms, Floating point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations.

COMPUTER ARITHMETIC:

Addition, subtraction, multiplication are the four basic arithmetic operations. Using these operations other arithmetic functions can be formulated and scientific problems can be solved by numerical analysis methods.

Arithmetic Processor:

It is the part of a processor unit that executes arithmetic operations. The arithmetic instructions definitions specify the data type that should be present in the registers used . The arithmetic instruction may specify binary or decimal data and in each case the data may be in fixed-point or floating point form.

Fixed point numbers may represent integers or fractions. The negative numbers may be in signed-magnitude or signed- complement representation. The arithmetic processor is very simple if only a binary fixed point add instruction is included. It would be more complicated if it includes all four arithmetic operations for binary and decimal data in fixed and floating point representations.

Algorithm:

Algorithm can be defined as a finite number of well defined procedural steps to solve a problem. Usually, an algorithm will contain a number of procedural steps which are dependent on results of previous steps. A convenient method for presenting an algorithm is a flowchart which consists of rectangular and diamond –shaped boxes. The computational steps are specified in the rectangular boxes and the decision steps are indicated inside diamond-shaped boxes from which 2 or more alternate path emerge.

Addition and Subtraction:

3 ways of representing negative fixed point binary numbers:

1. Signed-magnitude representation---- used for the representation of mantissa for floating point operations by most computers.
2. Signed-1's complement
3. Signed -2's complement—Most computers use this form for performing arithmetic operation with integers

Addition and subtraction algorithm for signed-magnitude data

Let the magnitude of two numbers be A & B. When signed numbers are added or subtracted, there are 4 different conditions to be considered for each addition and subtraction depending on the sign of the numbers. The conditions are listed in the table below. The table shows the operation to be performed with magnitude(addition or subtraction) are indicated for different conditions.

Sl.No	Operation	Add Magnitudes	Subtract magnitudes		
			When A> B	When A< B	When A=B
1	(+A) + (+B)	+ (A + B)			
2	(+A) + (-B)		+(A-B)	-(B-A)	+(A-B)
3	(-A) + (+B)		-(A-B)	+(B-A)	+(A-B)
4	(-A) + (-B)	- (A + B)			
5	(+A) - (+B)		+(A-B)	-(B-A)	+(A-B)
6	(+A) - (-B)	+ (A + B)			
7	(-A) - (+B)	- (A + B)			
8	(-A) - (-B)		-(A-B)	+(B-A)	+(A-B)

The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

The algorithm for addition and subtraction (from the table above):

Addition Algorithm:

When the signs of A and B are identical, add two magnitudes and attach the sign of A to the result. When the sign of A and B are different, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A>B or the complement of sign of A if A < B. If the two magnitudes are equal, subtract B from A and make te sign of the result positive.

Subtraction algorithm:

When the signs of A and B are different, add two magnitudes and attach the sign of A to the result. When the sign of A and B are identical, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A>B or the complement of sign of A if A < B. If the two magnitudes are equal, subtract B from A and make te sign of the result positive.

Hardware Implementation:

Let A and B are two registers that hold the numbers.

A_S and B_S are 2, flip-flops that hold sign of corresponding numbers. The result is stored In A and A_S .and thus they form Accumulator register.

We need to perform micro operation, A+ B and hence a parallel adder.

A comparator is needed to establish if A> B, A=B, or A<B.

We need to perform micro operations A-B and B-A and hence two parallel subtractor.

An exclusive OR gate can be used to determine the sign relationship, that is, equal or not.

Thus the hardware components required are a magnitude comparator, an adder, and two subtractors.

Reduction of hardware by using different procedure:

1. We know subtraction can be done by complement and add.
2. The result of comparison can be determined from the end carry after the subtraction. We find An adder and a complementer can do subtraction and comparison if 2's complement is used for subtraction.

Hardware forsigned-magnitude addition and subtraction:

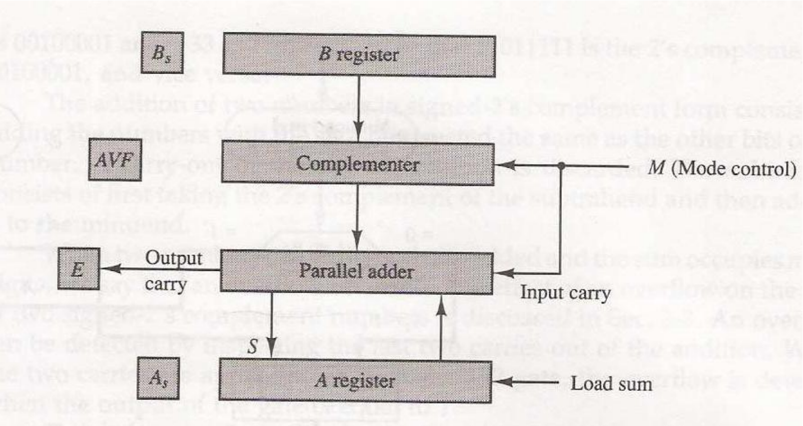


Figure 10-1 Hardware for signed-magnitude addition and subtraction.

AVF Add overflow flip flop. It hold the overflow bit when A & B are added.

Flip flop E—Output carry is transferred to E. It can be checked to see the relative magnitudes of the two numbers.

A-B = A +(-B)= Adding a and 2's complement of B.

The A register provides other micro operations that may be needed when the sequence of steps in the algorithm is specified.

The complementer Passes the contents of B or the complement of B to the Parallel Adder depending on the state of the mode control B. It consists of EX-OR gates and the parallel adder consists of full adder circuits. The M signal is also applied to the input carry of the adder.

When input carry M=0, the sum of full adder is A +B. When M=1, S = A + B' +1= A – B

Hardware algorithm:

Flow Chart for Add and Subtract operations:

The EX-OR gate provides 0 as output when the signs are identical. It is 1 when the signs are different.

A + B is computed for the following and the sum is stored in EA:

1. When the signs are same and addition operation is required.
2. When the signs are different and subtract operation is required.

   The carry in E after addition indicates an overflow if it is 1 and it is transferred to AVF, the addoverflow flag

   A-B = A+ B'+1 computed for the following:

   1. When the signs are different and addition operation is required.
   2. When the signs are same and subtract operation is required.
      No overflow can occur if the numbers are subtracted and hence AVF is cleared to Zero.

      [the subtraction of 2 n-digit un signed numbers M-N (N≠0) in base r can be done as follows:

      1. Add minuend M to thee r's complement of the subtrahend N. This performs M-N $+r^n$.
      2. If M ≥ N, The sum will produce an end carry r^nwhich is discarded, and what is left is the result M-N.
      3. If M< N, the sum does not produce an end carry and is equal to r^n–(N-M), which is the r's complement of the sum and place a negative sign in front.]

         A 1 in E indicates that A ≥ B and the number in A is the correct result.
         If this number in A is zero, the sign A_S must be made positive to avoid a negative zero.
         A 0 in E indicates that A< B. For this case it is necessary to take the 2's complement of the value in A.
         In the algorithm shown in flow chart, it is assumed that A register has circuits for micro operations complement and increment. Hence two complement of value in A is obtained in 2, micro operations. In other paths of the flow chart , the sign of the result is the same as the sign of A, so no change in A_S is required.

However When A < B, the sign of the result is the complement of original sign of A.

Hence The complement of A_S stored in A_S.

Final Result: A_S A

Flow chart for ADD and Subtract operations:

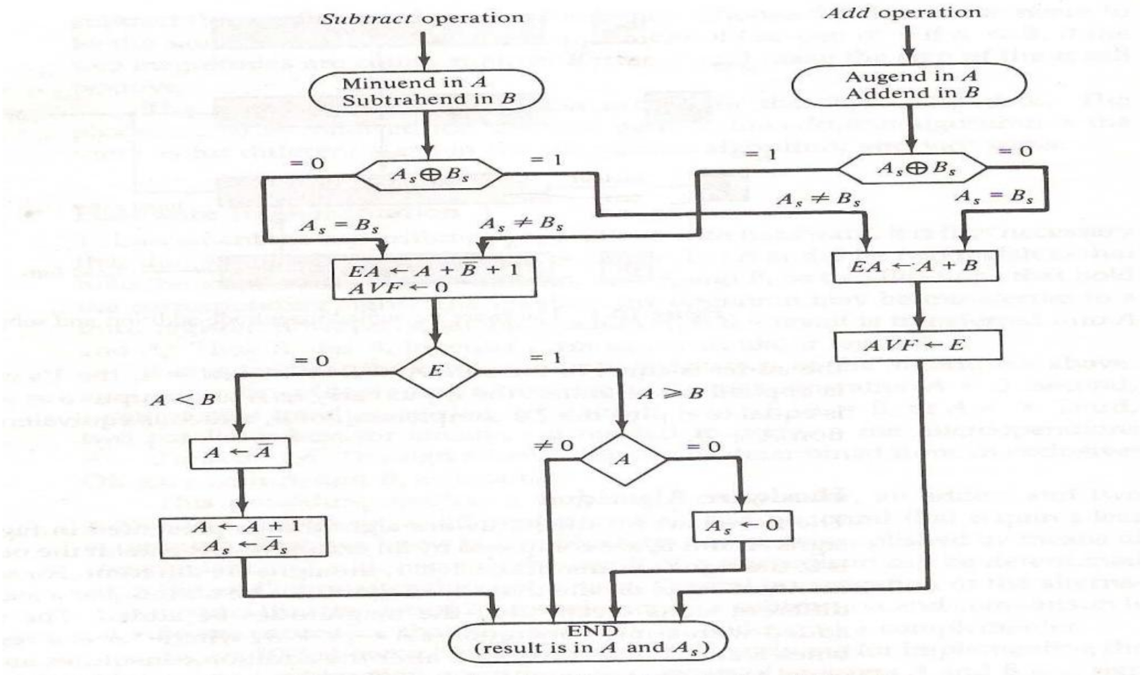


Figure 10-2 Flowchart for add and subtract operations.

Addition and Subtraction with signed-2's complement Data.:

Arithmetic Addition:

This method does not need a comparison or subtraction but only addition and complementation. The procedure is as below:

1. **Represent the negative numbers in 2's complement form.**
2. **Add the two numbers including the sign bits and discard any carry out of sign bit position.**
3. **The overflow bit V is set to 1 if there is a carry into sign bit and no carry out of sign bit or if there is a no carry into sign bit and a carry out of sign bit. Otherwise it is set to zero.**
4. **If the result is negative, take the 2's complement of the result to get a correct negative result.**

Arithmetic Subtraction:

1. **Represent the negative numbers in 2's complement form.**

2. **Take the 2's complement of the subtrahend including the sign bit and add it to the minuend including the sign bit.**
3. **The overflow bit V is set to 1 if there is a carry into sign bit and no carry out of sign bit or if there is a no carry into sign bit and a carry out of sign bit. Otherwise it is set to zero.**
4. **Discard the carry out of the sign bit position.**

Note: A subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed.

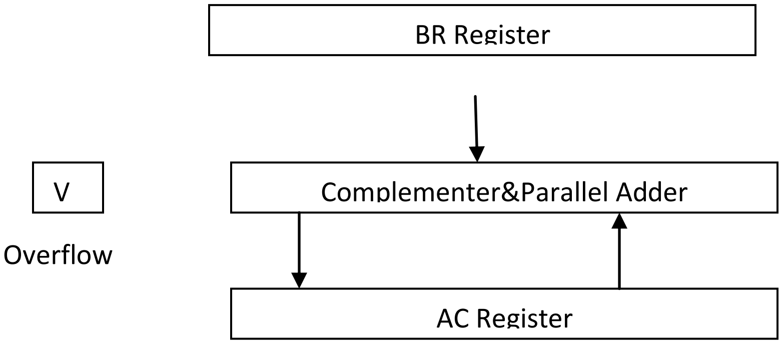


Fig: Hardware for Signed 2/s complement for addition/ subtractioin.

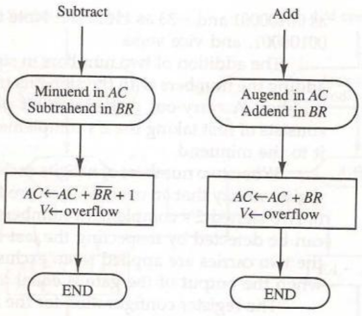


Figure 10-4 Algorithm for adding and subtracting numbers in signed-2's complement representation.

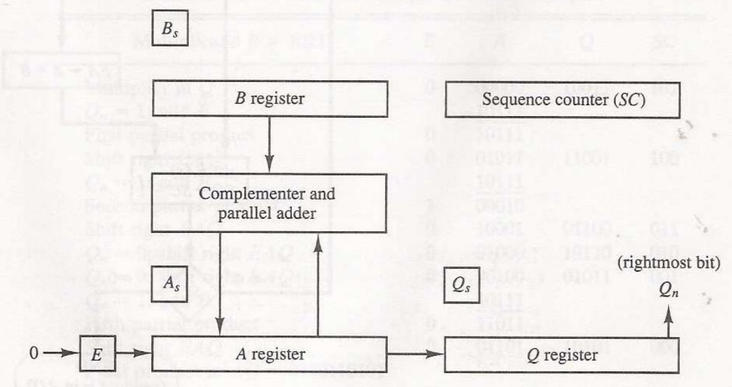
Multiplication Algorithm:

Hardware implementation of multiplication of numbers in signed – magnitude form:

1. A adder is provided to add two binary numbers and the partial product is accumulated in a register.
2. Instead of shifting the multiplicand to the left, the partial product is shifted to the right, which result in leaving the partial product and the multiplicand in the required relative positions.
3. When the corresponding bit of the multiplier is zero, there is no need to add all zeros to the partial product, since it will not alter it's value.

The hardware consists of 4 flipflops, 3 registers, one sequence counter , an adder and complementer.

Figure 10-5 Hardware for multiply operation.



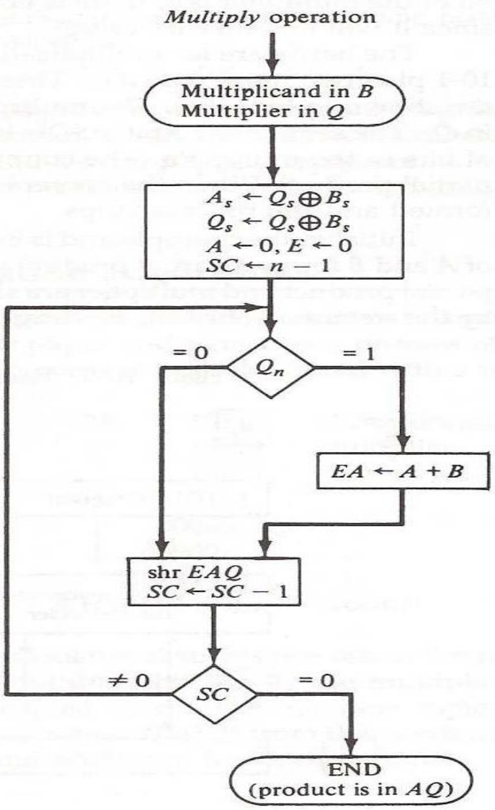
Q register&Q_S flip flop : contains multiplier & Its sign
Sequence counter : It is set to a value equal to the number of bits in the multiplier
B Register& B_S flipflop : It contains the multiplicand,& its sign
A Register, E Flip flop : Initialized to ' 0'. A_S denotes sign of partial product
EA Register : hold partial product, with carry generated in addition being shifted to E .
Qn : Rightmost bit of the multiplier; AQ : will contain the final product.

As **AQ** represent product register, both A_S $Q_S$represent the sign of the partial product or product.
The number to be multiplied are stores in memory as n bit sign magnitude numbers and when transferred to register msb bit go to sign flipflop and remaining n-1 bits go to registers. Hence SC is initially set to n-1.
Let the lower order bit of the multiplier in $Q_n$tested.
If it is 1, the multiplicand in B is added to the present partial product in A.
If it is a '0', nothing is done. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and it's new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when SC = 0.

The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

Flowchart for multiply operation:

Figure 10-6 Flowchart for multiply operation.



Numerical Example for the above algorithm:

Multiplicand B= 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
Q_n =1;add B		10111		
First Partial Product	0	10111		
Shift Right EAQ	0	01011	11001	100
Q_n =1;add B		10111		
Second Partial Product	1	00010		

Shift Right EAQ	0	10001	01100	011
Q_n =0; Shift Right EAQ	0	01000	10110	010
Q_n =0; Shift Right EAQ	0	00100	01011	001
Q_n =1;add B		10111		
Fifth Partial Product	0	11011		
Shift Right EAQ	0	01101	10101	000
Final Product in AQ AQ = 0110110101				

Booth Multiplication Algorithm:

Multiplication of signed- 2's complement integers:

This algorithm uses the following facts.

1. A string of 0's in the multiplier requires no addition but just shifting.
2. A string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as 2^{k+1} - $2^{m.}$

Example: Consider the binary number: 001110(+14)

The number has a string of 1's from 2^3 to 2^1. Hence k = 3 and m= 1. As other bits are 0's, the number can be represented as $2^{k+1} - 2^m = 2^4 – 2^1 = 16-2 = 14$. Therefore the multiplication M * 14 , where M is the multiplicand and 14 the multiplier can be done as Mx 2^4 –M x $2^{1.}$

This can be achieved by shifting binary multiplicand M four times to the left and subtracting M shifted left once which is equal to (Mx 2^4 –M x $2^{1.}$).

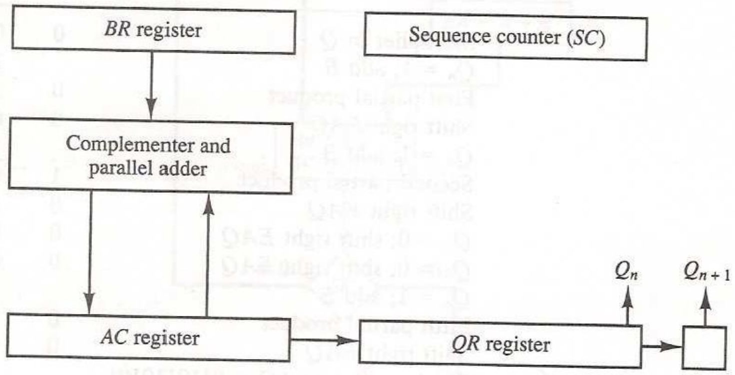
Shifting and addition/subtraction rules for multiplicand in Booth's Algorithm:

1. The multiplicand is subtracted from the partial product upon encountering the first least significand 1 in a string of I'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 Booth Algorithm:

Figure 10-7 Hardware for Booth algorithm.



Note: Sign bit is not separated from register. QR register contains the multiplier register and $Q_n$represent the least significant bit of the multiplier in QR. Q_{n+1} is an extra flip flop appended to QR to facilitate a double bit inspection of the multiplier.

AC register and appended Q_{n+1} are initially cleared to 0.

Sequence counter Sc is set to the number n which is equal to the number of bits of bits In the multiplier.

Q_nQ_{n+1} are to successive bits in the multiplier

Example for multiplication using Boot h algorithm:

Q_nQ_{n+1}	BR = 1011 ,BR'+1 = 01001	AC	QR	Q_{n+1}	SC
10	Initial Subtract BR ashr	00000 01001 01001 00100	10011 11001	0 1	101 100
11	ashr	00010	01100	1	011
01	Add BR ashr	10111 11001 11100	 10110	 0	 010
00	ashr	11110	01011	0	001
10	Subtract BR	01001			

	Ashr	00111 00011	10101	1	000

Algorithm in flowchart for multiplication of signed 2's complement numbers.

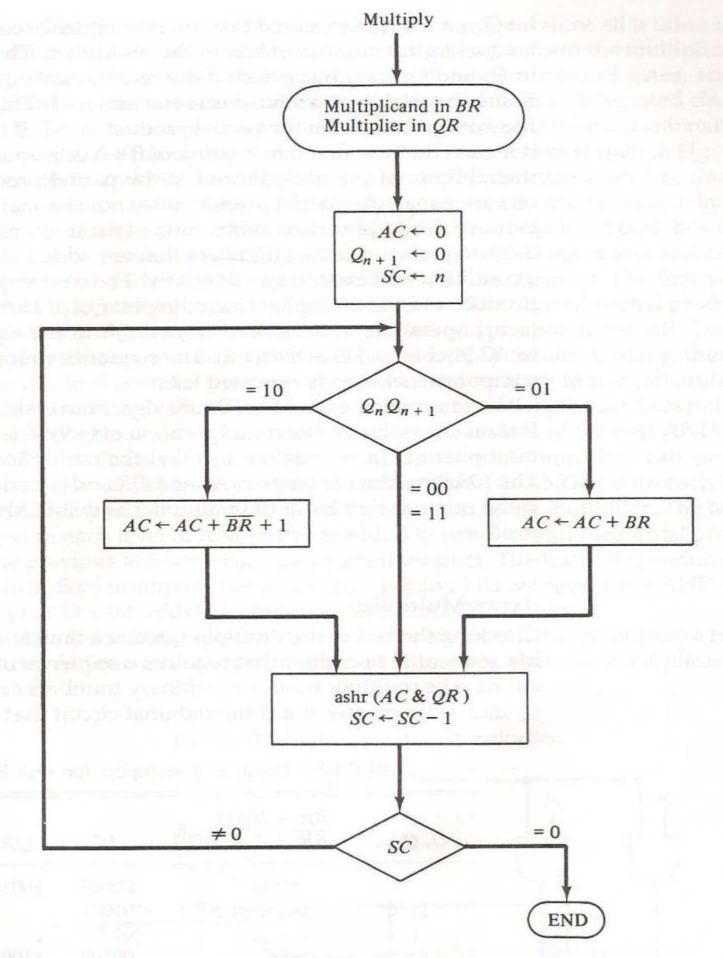


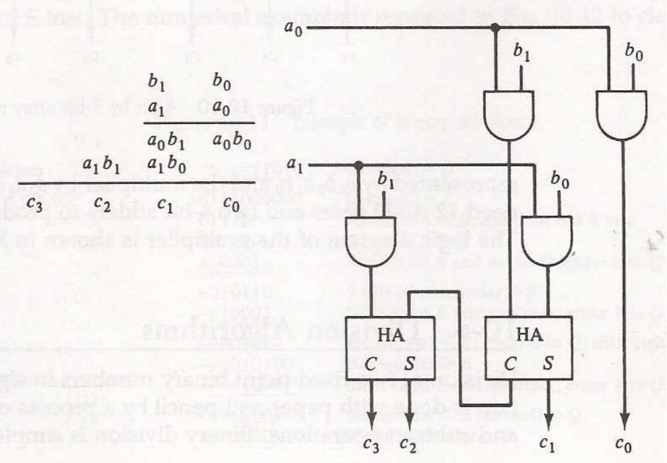
Figure 10-8 Booth algorithm for multiplication of signed-2's complement numbers.

Array Multiplier:

2 -bit by 2- bit Array Multiplier:

Multiplicand bits are b_1 and b_0 .Multiplier bits are a_1 and a_0 The first partial product is obtained by multiplying a_0 by b_1 b_0 . The bit multiplication is implemented by AND gate. First partial product is made by two AND gates. Second partial product is made by two AND gates. The two partial products are added with two half adder circuits.

Figure 10-9 2-bit by 2-bit array multiplier.



Combinational circuit binary multiplier:

A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there bits in the multiplier. The binary output in each level of the AND gates is added in parallel with the partial product of the previous level to form a ne partial product. The last level produces the product. For j multiplier and k multiplicand bits, we need j*k AND Gates and (j-1)*k bit adders to ptoduce a product of j+k bits.

4- bit by 3-bit Array Multiplier:

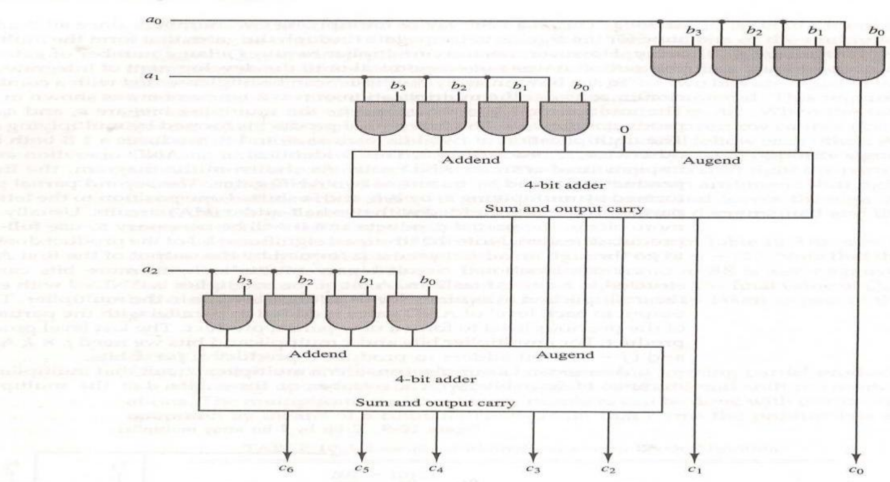


Figure 10-10 4-bit by 3-bit array multiplier.

Division Algorithms:

Division Process for division of fixed point binary number in signed –magnitude representation:

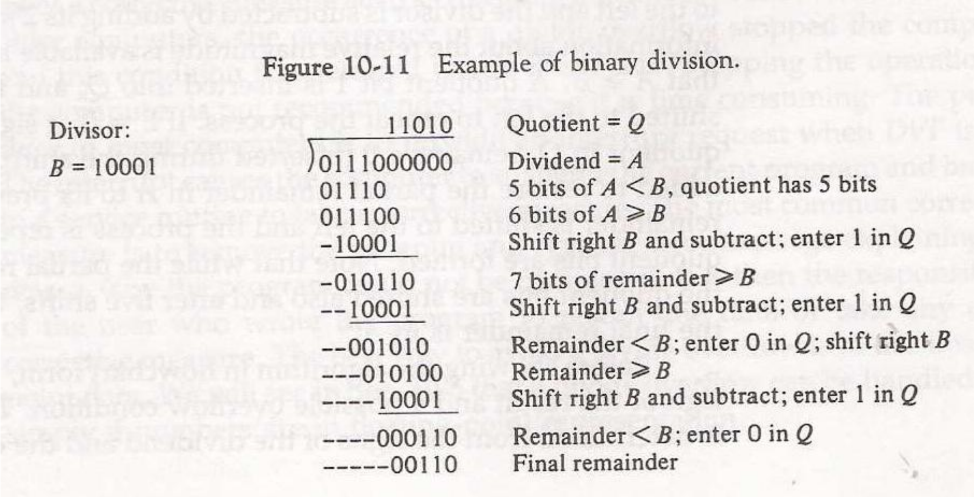


Figure 10-11 Example of binary division.

```
Divisor:          11010     Quotient = Q
B = 10001    )0111000000    Dividend = A
              01110         5 bits of A < B, quotient has 5 bits
              011100        6 bits of A ≥ B
              -10001        Shift right B and subtract; enter 1 in Q
              -010110       7 bits of remainder ≥ B
              --10001       Shift right B and subtract; enter 1 in Q
              --001010      Remainder < B; enter 0 in Q; shift right B
              ---010100     Remainder ≥ B
              ----10001     Shift right B and subtract; enter 1 in Q
              ----000110    Remainder < B; enter 0 in Q
              -----00110    Final remainder
```

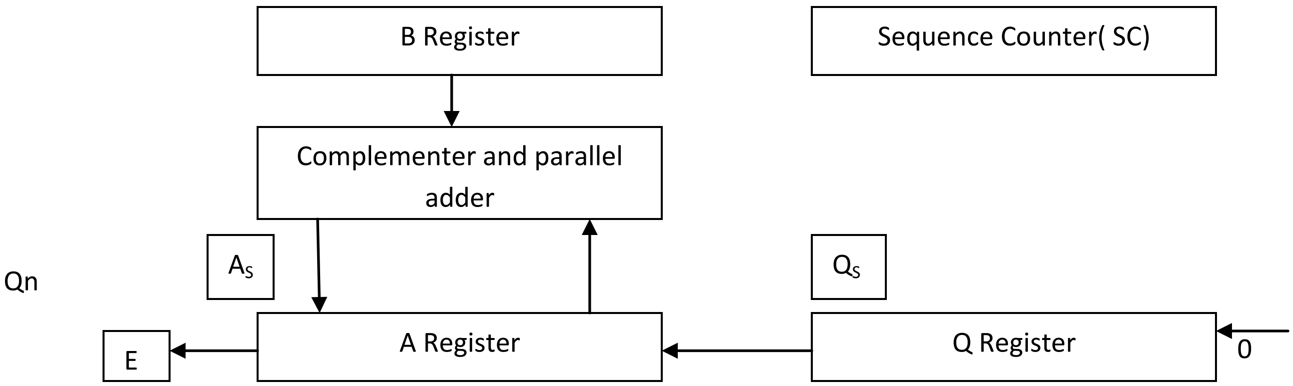
Let dividend A consists of 10 bits and divisor B consists of 5 bits.

1. Compare the 5 most significant bits of the dividend with that of divisor.
2. If the 5 bit number is smaller than divisor B, then take 6 bits of the dividend and compare with the 5 bit divisor.
3. The 6 bit number is greater than divisor B. Hence place a 1 for the quotient bit in the sixth position above the dividend. Shift the divisor once to the right and subtracted from the dividend. The difference is called partial remainder.
4. Repeat the process with the partial remainder and divisor. If the partial remainder is equal or greater than or equal to the divisor, the quotient bit is equal to 1.The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is small than the divisor, then the quotient bit is zero and no subtraction is needed. The divisor is shifted once to the right in any case,.

Hardware Implementation of division for signed magnitude fixed point numbers:

To implement division using a digital computer, the process is changed slightly for convenience.

1. Instead of shifting the divisor to the right, the dividend or the partial remainder, is shifted to the left so as to leave the two numbers in the required relative position.
2. Subtraction may be achieved by adding A (dividend)to the 2's complement of B(divisor). The information about the relative magnitude is then available from end carry.
3. Register EAQ is now shifted to the left with 0 inserted into Q_n and the previous value of E is lost..
4. The divisor is stored in B register and the double length dividend is stored in registers A and Q.
5. The dividend is shifted to the left and the divisor is subtracted by adding it's 2's complement value.
6. If E= 1, it signifies that A ≥ B.A quotient bit is inserted into $Q_n$and the partial remainder is shifted to the left to repeat the process.
7. If E = 0, it signifies that A < B so the quotient Q_n remains 0(inserted during the shift). The value of B is then added to restore the partial remainder in A to its previous value. The partial remainder is shifted to the left and the process is repeated again until all 5 quotient bits are formed.
8. At the end Q contains the quotient and A the remainder. If the sign of dividend and divisor are alike, the quotient is positive and if unalike, it is negative. The sign of the remainder is the same as dividend.



Hardware for implementing division of fixed point signed- Magnitude Numbers

Example of Binary division with digital hardware: Divisor B = 10001, $\overline{B}$ + 1 = 01111

		E	A	Q	SC
	Dividend:		01110	00000	5
	Shl EAQ		11100	00000	
	Add **,** $\overline{B}$ **+ 1**		01111		
	E = 1	1	01011		
	Set Q_n= 1	1	01011	00001	4
	Shl EAQ	0	10110	00010	
	Add **, B + 1**		01111		
	E = 1	1	00101		
	Set Q_n= 1	1	00101	00011	3
	Shl EAQ	0	01010	00110	
	Add **, B + 1**		01111		
	E= 0; Leave Q_n= 0	0	11001	00110	
	Add B		10001		

	Restore remainder	1	01010		2
	Shl EAQ	0	10100	01100	
	Add , **B +1**		01111		
	E = 1	1	00011		
	Set Q_n= 1	1	00011	01101	1
	Shl EAQ	0	00110	11010	
	Add , **B +1**		01111		
	E= 0; Leave Q_n= 0	0	10101	11010	
	Add B		10001		
	Restore remainder	1	00110	11010	0
	Neglect E				
	Remainder in A		00110	11010	
	Quotient in Q				

Divide overflow:

When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows:

A divide-overflow condition occurs if the higher order half bits of the dividend constitute a number greater than or equal to the divisor. If the divisor is zero, then the dividend will definitely be greater than or equal to divisor. Hence divide overflow condition occurs and hence the divide-overflow –flip flop will be set. Let the flip flop be called DVF.

Handling DVF:

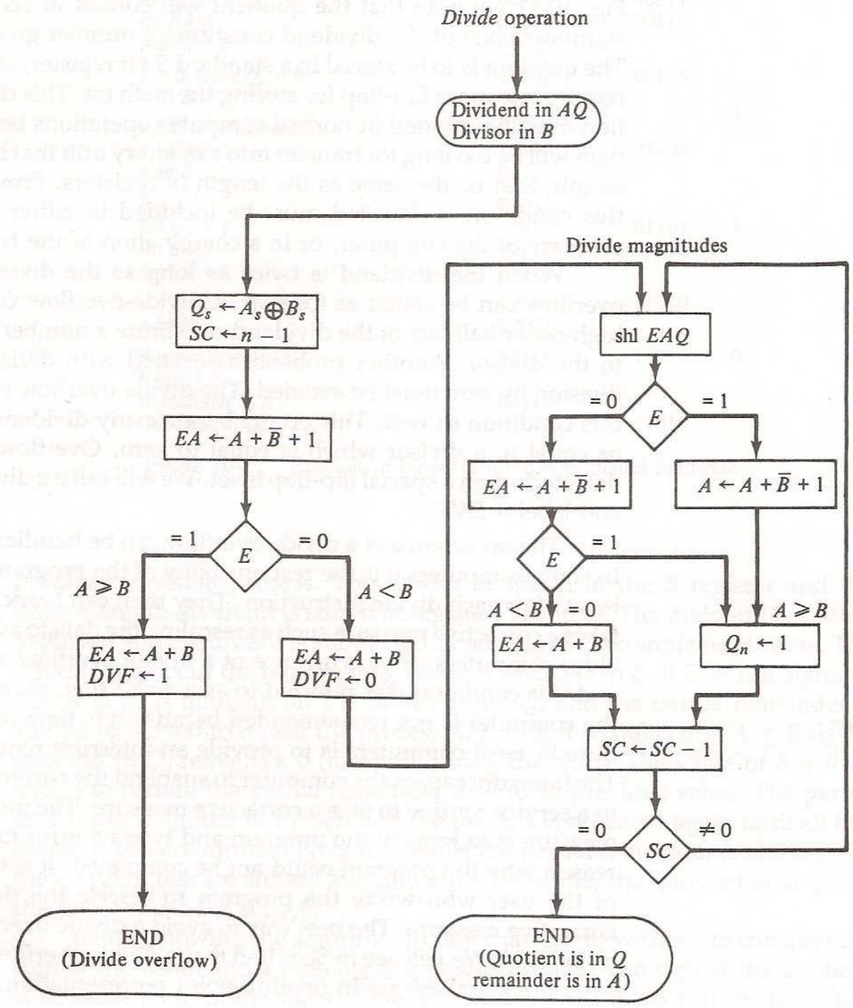
1. Check if DVF is set after each divide instruction. If DVF is set, then the program branches to a subroutine that takes corrective measures such as rescaling the data to avoid overflow.
2. An interrupt is generated if DVF is set. The interrupt causes the processor to suspend the current program and branch to interrupt service routine to take corrective measure. The most

common corrective measure is to remove the program and type an error message that explains the reasons.

3. The divide overflow can be handled very simply if the numbers are represented in floating point representation.

Flow chart for divide operation:

Figure 10-13 Flowchart for divide operation.



Assumption:

Operands are transferred from memory to registers as n bit words.n-1 bit form magnitude and 1 bit shows the sign.

A divide overflow condition is tested by subtracting the divisor in B from half of the bits of dividend stored in A. If vA ≥ B, the DVF is set and the operation is terminated prematurely. If A < B, no DVF occurs and so the value of dividend is restored by adding B to A.

The division of the magnitudes starts by shifting the dividend in AQ to the left, with the higher order bit shifted into E. If the bit shifted into E is 1, we know that EA is greater than B because EA consists of a 1 followed by n-1 bits while B consists of only n-1 bits. In this case, B must be subtracted from EA and 1 inserted into Q_n for the quotient bit. Since register A is missing the higher order bit of the dividend (which is in E), it's value is EA – 2^{n-1} . Adding to this value the 2's complement of B results in

$(EA-2^{n-1}) + (2^{n-1}–B)= E-B$. The carry from the addition is not transferred to E if we want E to remain a 1.

If the shift left operation inserts a zero into E, the divisor is subtracted by adding it's 2's complement value and the carry is transferred into E. If E = 1, it signifies that A ≥ B and hence Q_n is set to 1. If E = 0, it signifies that A < B and the original number is restored by adding B to A. In the latter case we leave a 0 in $Q_{n\ .}$(0 was inserted during the shift).

This process is repeated again with register A holding the partial remainder. After n-1 times, the quotient magnitude is formed in the register Q and the remainder is found in register A.

10-5 Floating-Point Arithmetic Operations

integer declaration statement

Many high-level programming languages have a facility for specifying floating-point numbers. The most common way is to specify them by a *real* declaration statement as opposed to fixed-point numbers, which are specified by an *integer* declaration statement. Any computer that has a compiler for such high-level programming language must have a provision for handling floating-point arithmetic operations. The operations are quite often included in the internal hardware. If no hardware is available for the operations, the compiler must be designed with a package of floating-point software subroutines. Although the hardware method is more expensive, it is so much more efficient than the software method that floating-point hardware is included in most computers and is omitted only in very small ones.

Basic Considerations

Floating-point representation of data was introduced in Sec. 3-4. A floating-point number in computer registers consists of two parts: a mantissa m and an exponent e. The two parts represent a number obtained from multiplying m times a radix r raised to the value of e; thus

$$m \times r^e$$

The mantissa may be a fraction or an integer. The location of the radix point and the value of the radix r are assumed and are not included in the registers. For example, assume a fraction representation and a radix 10. The decimal number 537.25 is represented in a register with $m = 53725$ and $e = 3$ and is interpreted to represent the floating-point number

$$.53725 \times 10^3$$

A floating-point number is normalized if the most significant digit of the mantissa is nonzero. In this way the mantissa contains the maximum possible number of significant digits. A zero cannot be normalized because it does not have a nonzero digit. It is represented in floating-point by all 0's in the mantissa and exponent.

Floating-point representation increases the range of numbers that can be accommodated in a given register. Consider a computer with 48-bit words. Since one bit must be reserved for the sign, the range of fixed-point integer numbers will be $\pm(2^{47} - 1)$, which is approximately $\pm 10^{14}$. The 48 bits can be used to represent a floating-point number with 36 bits for the mantissa and 12 bits for the exponent. Assuming fraction representation for the mantissa and taking the two sign bits into consideration, the range of numbers that can be accommodated is

$$\pm(1 - 2^{-35}) \times 2^{2047}$$

This number is derived from a fraction that contains 35 1's, an exponent of 11 bits (excluding its sign), and the fact that $2^{11} - 1 = 2047$. The largest number that can be accommodated is approximately 10^{615}, which is a very large number. The mantissa can accommodate 35 bits (excluding the sign) and if considered as an integer it can store a number as large as $(2^{35} - 1)$. This is approximately equal to 10^{10}, which is equivalent to a decimal number of 10 digits.

Computers with shorter word lengths use two or more words to represent a floating-point number. An 8-bit microcomputer may use four words to represent one floating-point number. One word of 8 bits is reserved for the exponent and the 24 bits of the other three words are used for the mantissa.

Arithmetic operations with floating-point numbers are more complicated than with fixed-point numbers and their execution takes longer and requires more complex hardware. Adding or subtracting two numbers requires first an alignment of the radix point since the exponent parts must be made equal before adding or subtracting the mantissas. The alignment is done by shifting one mantissa while its exponent is adjusted until it is equal to the other exponent. Consider the sum of the following floating-point numbers:

$$\begin{array}{r} .5372400 \times 10^{2} \\ +\ .1580000 \times 10^{-1} \end{array}$$

It is necessary that the two exponents be equal before the mantissas can be added. We can either shift the first number three positions to the left, or shift the second number three positions to the right. When the mantissas are stored in registers, shifting to the left causes a loss of most significant digits. Shifting to the right causes a loss of least significant digits. The second method is preferable because it only reduces the accuracy, while the first method may cause an error. The usual alignment procedure is to shift the mantissa that has

2.

the smaller exponent to the right by a number of places equal to the difference between the exponents. After this is done, the mantissas can be added:

$$\begin{array}{r} .5372400 \times 10^2 \\ +.0001580 \times 10^2 \\ \hline .5373980 \times 10^2 \end{array}$$

When two normalized mantissas are added, the sum may contain an overflow digit. An overflow can be corrected easily by shifting the sum once to the right and incrementing the exponent. When two numbers are subtracted, the result may contain most significant zeros as shown in the following example:

$$\begin{array}{r} .56780 \times 10^5 \\ -.56430 \times 10^5 \\ \hline .00350 \times 10^5 \end{array}$$

A floating-point number that has a 0 in the most significant position of the mantissa is said to have an *underflow*. To normalize a number that contains an underflow, it is necessary to shift the mantissa to the left and decrement the exponent until a nonzero digit appears in the first position. In the example above, it is necessary to shift left twice to obtain $.35000 \times 10^3$. In most computers, a normalization procedure is performed after each operation to ensure that all results are in a normalized form.

Floating-point multiplication and division do not require an alignment of the mantissas. The product can be formed by multiplying the two mantissas and adding the exponents. Division is accomplished by dividing the mantissas and subtracting the exponents.

The operations performed with the mantissas are the same as in fixed-point numbers, so the two can share the same registers and circuits. The operations performed with the exponents are compare and increment (for aligning the mantissas), add and subtract (for multiplication and division), and decrement (to normalize the result). The exponent may be represented in any one of the three representations: signed-magnitude, signed-2's complement, or signed-1's complement.

A fourth representation employed in many computers is known as a *biased* exponent. In this representation, the sign bit is removed from being a separate entity. The bias is a positive number that is added to each exponent as the floating-point number is formed, so that internally all exponents are positive. The following example may clarify this type of representation. Consider an exponent that ranges from −50 to 49. Internally, it is represented by two digits (without a sign) by adding to it a bias of 50. The exponent register contains the number $e + 50$, where e is the actual exponent. This way, the exponents are represented in registers as positive numbers in the range of 00 to 99. Positive exponents in registers have the range of numbers from 99 to 50. The subtraction of 50 gives the positive values from 49 to 0. Negative exponents are represented in registers in the range from 49 to 00. The subtraction of 50 gives the negative values in the range of −1 to −50.

The advantage of biased exponents is that they contain only positive numbers. It is then simpler to compare their relative magnitude without being concerned with their signs. As a consequence, a magnitude comparator can be used to compare their relative magnitude during the alignment of the mantissa. Another advantage is that the smallest possible biased exponent contains all zeros. The floating-point representation of zero is then a zero mantissa and the smallest possible exponent.

In the examples above, we used decimal numbers to demonstrate some of the concepts that must be understood when dealing with floating-point numbers. Obviously, the same concepts apply to binary numbers as well. The algorithms developed in this section are for binary numbers. Decimal computer arithmetic is discussed in the next section.

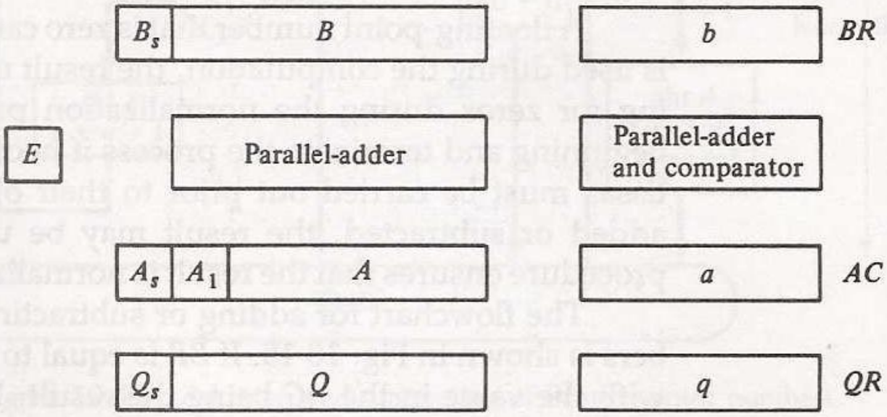
Register Configuration

The register configuration for floating-point operations is quite similar to the layout for fixed-point operations. As a general rule, the same registers and adder used for fixed-point arithmetic are used for processing the mantissas. The difference lies in the way the exponents are handled.

The register organization for floating-point operations is shown in Fig. 10-14. There are three registers, *BR*, *AC*, and *QR*. Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part uses the corresponding lowercase letter symbol.

It is assumed that each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus the *AC* has a mantissa

Figure 10-14 Registers for floating-point arithmetic operations.



whose sign is in A_s and a magnitude that is in A. The exponent is in the part of the register denoted by the lowercase letter symbol a. The diagram shows explicitly the most significant bit of A, labeled by A_1. The bit in this position must be a 1 for the number to be normalized. Note that the symbol AC represents the entire register, that is, the concatenation of A_s, A, and a.

Similarly, register BR is subdivided into B_s, B, and b, and QR into Q_s, Q, and q. A parallel-adder adds the two mantissas and transfers the sum into A and the carry into E. A separate parallel-adder is used for the exponents. Since the exponents are biased, they do not have a distinct sign bit but are represented as a biased positive quantity. It is assumed that the floating-point numbers are so large that the chance of an exponent overflow is very remote, and for this reason the exponent overflow will be neglected. The exponents are also connected to a magnitude comparator that provides three binary outputs to indicate their relative magnitude.

The number in the mantissa will be taken as a *fraction*, so the binary point is assumed to reside to the left of the magnitude part. Integer representation for floating-point causes certain scaling problems during multiplication and division. To avoid these problems, we adopt a fraction representation.

The numbers in the registers are assumed to be initially normalized. After each arithmetic operation, the result will be normalized. Thus all floating-point operands coming from and going to the memory unit are always normalized.

Addition and Subtraction

During addition or subtraction, the two floating-point operands are in AC and BR. The sum or difference is formed in the AC. The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

A floating-point number that is zero cannot be normalized. If this number is used during the computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary. The alignment of the mantissas must be carried out prior to their operation. After the mantissas are added or subtracted, the result may be unnormalized. The normalization procedure ensures that the result is normalized prior to its transfer to memory.

The flowchart for adding or subtracting two floating-point binary numbers is shown in Fig. 10-15. If BR is equal to zero, the operation is terminated, with the value in the AC being the result. If AC is equal to zero, we transfer

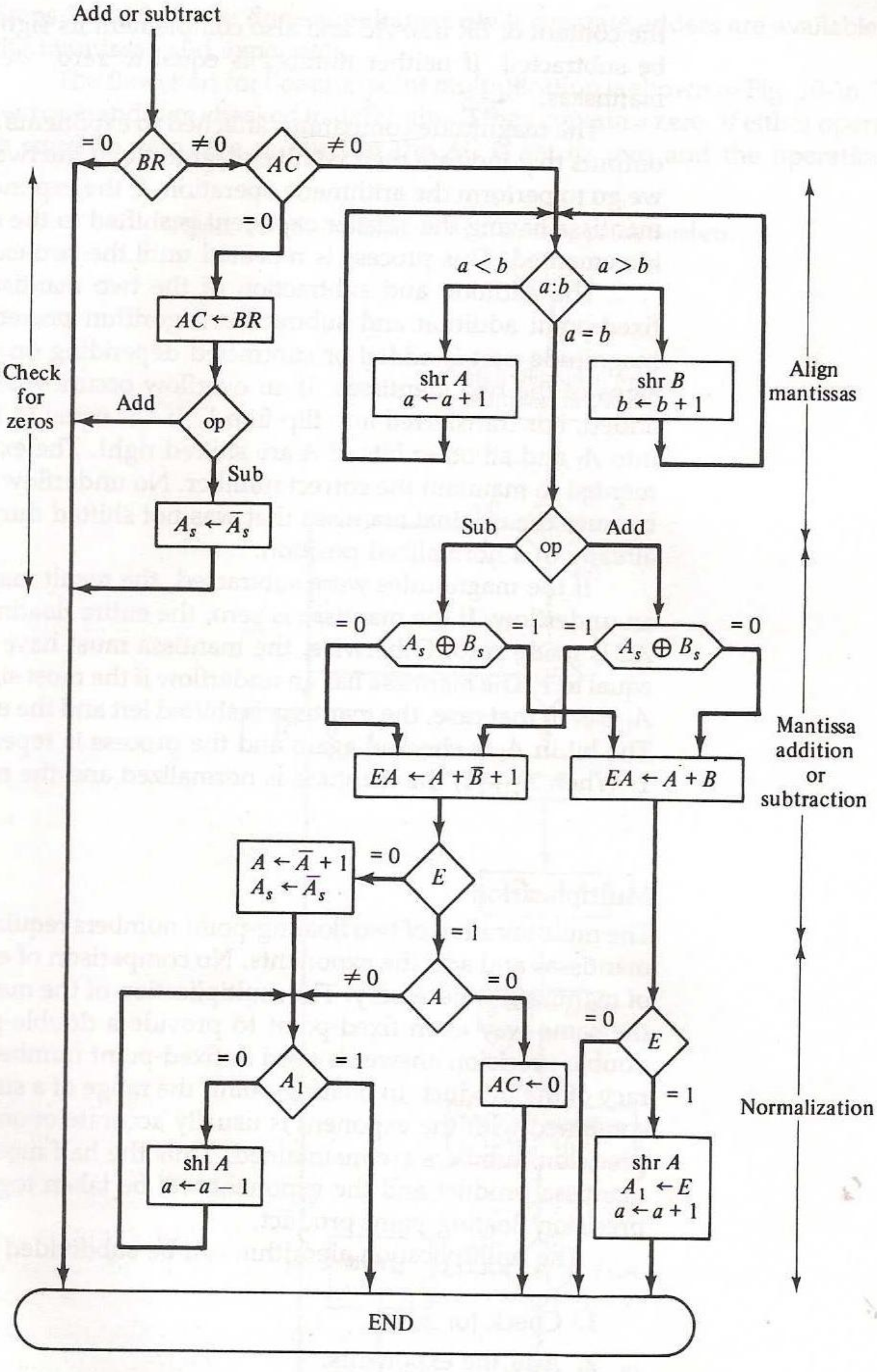


Figure 10-15 Addition and subtraction of floating-point numbers.

the content of *BR* into *AC* and also complement its sign if the numbers are to be subtracted. If neither number is equal to zero, we proceed to align the mantissas.

The magnitude comparator attached to exponents *a* and *b* provides three outputs that indicate their relative magnitude. If the two exponents are equal, we go to perform the arithmetic operation. If the exponents are not equal, the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until the two exponents are equal.

The addition and subtraction of the two mantissas is identical to the fixed-point addition and subtraction algorithm presented in Fig. 10-2. The magnitude part is added or subtracted depending on the operation and the signs of the two mantissas. If an overflow occurs when the magnitudes are added, it is transferred into flip-flop *E*. If *E* is equal to 1, the bit is transferred into A_1 and all other bits of *A* are shifted right. The exponent must be incremented to maintain the correct number. No underflow may occur in this case because the original mantissa that was not shifted during the alignment was already in a normalized position.

If the magnitudes were subtracted, the result may be zero or may have an underflow. If the mantissa is zero, the entire floating-point number in the *AC* is made zero. Otherwise, the mantissa must have at least one bit that is equal to 1. The mantissa has an underflow if the most significant bit in position A_1 is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit in A_1 is checked again and the process is repeated until it is equal to 1. When $A_1 = 1$, the mantissa is normalized and the operation is completed.

Multiplication

The multiplication of two floating-point numbers requires that we multiply the mantissas and add the exponents. No comparison of exponents or alignment of mantissas is necessary. The multiplication of the mantissas is performed in the same way as in fixed-point to provide a double-precision product. The double-precision answer is used in fixed-point numbers to increase the accuracy of the product. In floating-point, the range of a single-precision mantissa combined with the exponent is usually accurate enough so that only single-precision numbers are maintained. Thus the half most significant bits of the mantissa product and the exponent will be taken together to form a single-precision floating-point product.

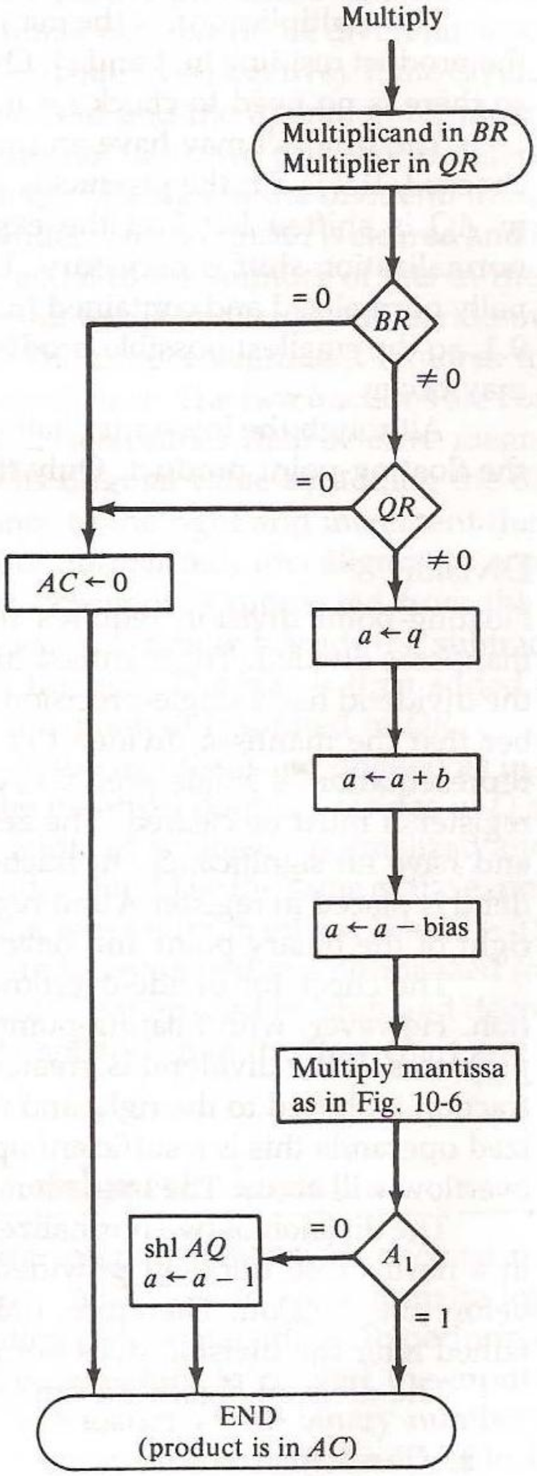
The multiplication algorithm can be subdivided into four parts:

1. Check for zeros.
2. Add the exponents.
3. Multiply the mantissas.
4. Normalize the product.

Steps 2 and 3 can be done simultaneously if separate adders are available for the mantissas and exponents.

The flowchart for floating-point multiplication is shown in Fig. 10-16. The two operands are checked to determine if they contain a zero. If either operand is equal to zero, the product in the *AC* is set to zero and the operation is

Figure 10-16 Multiplication of floating-point numbers.



terminated. If neither of the operands is equal to zero, the process continues with the exponent addition.

The exponent of the multiplier is in q and the adder is between exponents a and b. It is necessary to transfer the exponents from q to a, add the two exponents, and transfer the sum into a. Since both exponents are biased by the addition of a constant, the exponent sum will have double this bias. The correct biased exponent for the product is obtained by subtracting the bias number from the sum.

The multiplication of the mantissas is done as in the fixed-point case with the product residing in A and Q. Overflow cannot occur during multiplication, so there is no need to check for it.

The product may have an underflow, so the most significant bit in A is checked. If it is a 1, the product is already normalized. If it is a 0, the mantissa in AQ is shifted left and the exponent decremented. Note that only one normalization shift is necessary. The multiplier and multiplicand were originally normalized and contained fractions. The smallest normalized operand is 0.1, so the smallest possible product is 0.01. Therefore, only one leading zero may occur.

Although the low-order half of the mantissa is in Q, we do not use it for the floating-point product. Only the value in the AC is taken as the product.

Division

Floating-point division requires that the exponents be subtracted and the mantissas divided. The mantissa division is done as in fixed-point except that the dividend has a single-precision mantissa that is placed in the AC. Remember that the mantissa dividend is a fraction and not an integer. For integer representation, a single-precision dividend must be placed in register Q and register A must be cleared. The zeros in A are to the left of the binary point and have no significance. In fraction representation, a single-precision dividend is placed in register A and register Q is cleared. The zeros in Q are to the right of the binary point and have no significance.

The check for divide-overflow is the same as in fixed-point representation. However, with floating-point numbers the divide-overflow imposes no problems. If the dividend is greater than or equal to the divisor, the dividend fraction is shifted to the right and its exponent incremented by 1. For normalized operands this is a sufficient operation to ensure that no mantissa divide-overflow will occur. The operation above is referred to as a *dividend alignment*.

dividend alignment

The division of two normalized floating-point numbers will always result in a normalized quotient provided that a dividend alignment is carried out before the division. Therefore, unlike the other operations, the quotient obtained after the division does not require a normalization.

The division algorithm can be subdivided into five parts:

1. Check for zeros.
2. Initialize registers and evaluate the sign.

3. Align the dividend.
4. Subtract the exponents.
5. Divide the mantissas.

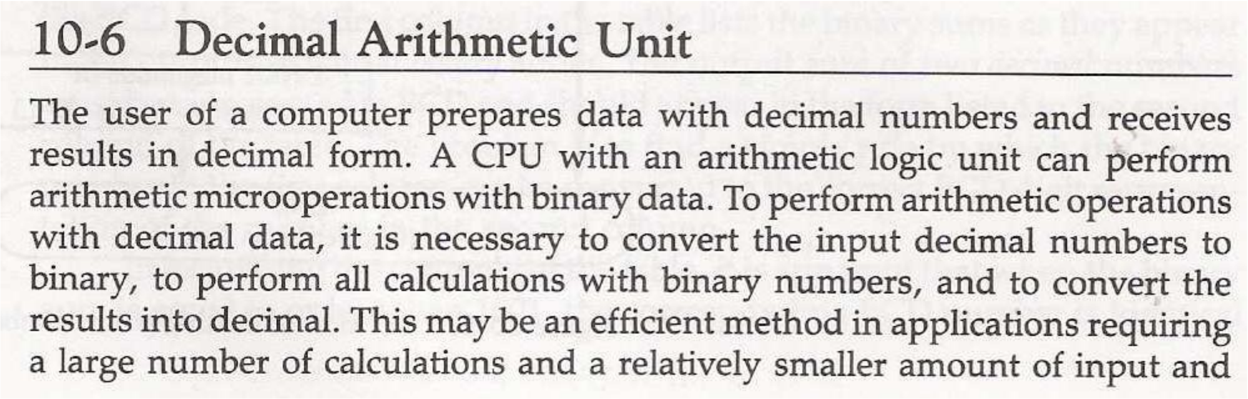
The flowchart for floating-point division is shown in Fig. 10-17. The two operands are checked for zero. If the divisor is zero, it indicates an attempt to divide by zero, which is an illegal operation. The operation is terminated with an error message. An alternative procedure would be to set the quotient in QR to the most positive number possible (if the dividend is positive) or to the most negative possible (if the dividend is negative). If the dividend in AC is zero, the quotient in QR is made zero and the operation terminates.

If the operands are not zero, we proceed to determine the sign of the quotient and store it in Q_s. The sign of the dividend in A_s is left unchanged to be the sign of the remainder. The Q register is cleared and the sequence counter SC is set to a number equal to the number of bits in the quotient.

The dividend alignment is similar to the divide-overflow check in the fixed-point operation. The proper alignment requires that the fraction dividend be smaller than the divisor. The two fractions are compared by a subtraction test. The carry in E determines their relative magnitude. The dividend fraction is restored to its original value by adding the divisor. If $A \geq B$, it is necessary to shift A once to the right and increment the dividend exponent. Since both operands are normalized, this alignment ensures that $A < B$.

Next, the divisor exponent is subtracted from the dividend exponent. Since both exponents were originally biased, the subtraction operation gives the difference without the bias. The bias is then added and the result transferred into q because the quotient is formed in QR.

The magnitudes of the mantissas are divided as in the fixed-point case. After the operation, the mantissa quotient resides in Q and the remainder in A. The floating-point quotient is already normalized and resides in QR. The exponent of the remainder should be the same as the exponent of the dividend. The binary point for the remainder mantissa lies $(n - 1)$ positions to the left of A_1. The remainder can be converted to a normalized fraction by subtracting $n - 1$ from the dividend exponent and by shift and decrement until the bit in A_1 is equal to 1. This is not shown in the flow chart and is left as an exercise.



10-6 Decimal Arithmetic Unit

The user of a computer prepares data with decimal numbers and receives results in decimal form. A CPU with an arithmetic logic unit can perform arithmetic microoperations with binary data. To perform arithmetic operations with decimal data, it is necessary to convert the input decimal numbers to binary, to perform all calculations with binary numbers, and to convert the results into decimal. This may be an efficient method in applications requiring a large number of calculations and a relatively smaller amount of input and

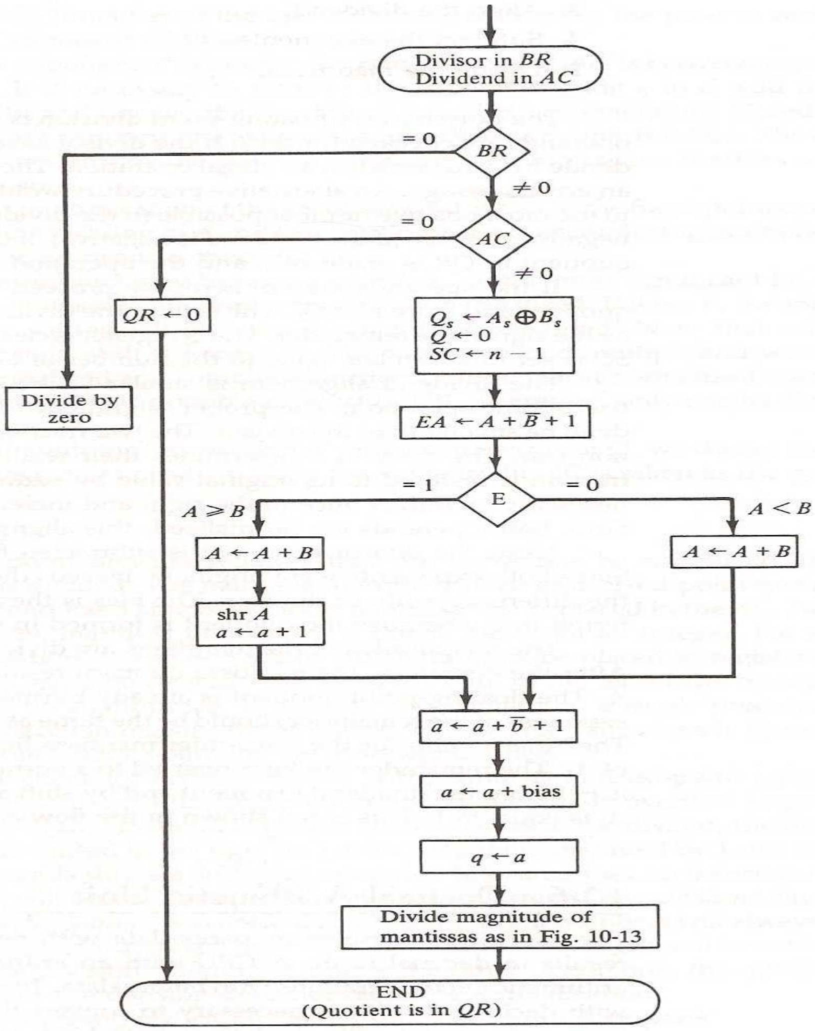


Figure 10-17 Division of floating-point numbers.

output data. When the application calls for a large amount of input–output and a relatively smaller number of arithmetic calculations, it becomes convenient to do the internal arithmetic directly with the decimal numbers. Computers capable of performing decimal arithmetic must store the decimal data in binary-coded form. The decimal numbers are then applied to a decimal arithmetic unit capable of executing decimal arithmetic microoperations.

Electronic calculators invariably use an internal decimal arithmetic unit since inputs and outputs are frequent. There does not seem to be a reason for converting the keyboard input numbers to binary and again converting the displayed results to decimal, since this process requires special circuits and also takes a longer time to execute. Many computers have hardware for arithmetic calculations with both binary and decimal data. Users can specify by programmed instructions whether they want the computer to perform calculations with binary or decimal data.

A decimal arithmetic unit is a digital function that performs decimal microoperations. It can add or subtract decimal numbers, usually by forming the 9's or 10's complement of the subtrahend. The unit accepts coded decimal numbers and generates results in the same adopted binary code. A single-stage decimal arithmetic unit consists of nine binary input variables and five binary output variables, since a minimum of four bits is required to represent each coded decimal digit. Each stage must have four inputs for the augend digit, four inputs for the addend digit, and an input-carry. The outputs include four terminals for the sum digit and one for the output-carry. Of course, there is a wide variety of possible circuit configurations dependent on the code used to represent the decimal digits.

BCD Adder

Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than $9 + 9 + 1 = 19$, the 1 in the sum being an input-carry. Suppose that we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in *binary* and produce a result that may range from 0 to 19. These binary numbers are listed in Table 10-4 and are labeled by symbols K, Z_8, Z_4, Z_2, and Z_1. K is the carry and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code. The first column in the table lists the binary sums as they appear in the outputs of a 4-bit *binary* adder. The output sum of two *decimal* numbers must be represented in BCD and should appear in the form listed in the second column of the table. The problem is to find a simple rule by which the binary number in the first column can be converted to the correct BCD digit representation of the number in the second column.

In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical

3.

TABLE 10-4 Derivation of BCD Adder

Binary Sum					BCD Sum					
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	Decimal
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain a nonvalid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output-carry as required.

One method of adding decimal numbers in BCD would be to employ one 4-bit binary adder and perform the arithmetic operation one digit at a time. The low-order pair of BCD digits is first added to produce a binary sum. If the result is equal or greater than 1010, it is corrected by adding 0110 to the binary sum. This second operation will automatically produce an output-carry for the next pair of significant digits. The next higher-order pair of digits, together with the input-carry, is then added to produce their binary sum. If this result is equal to or greater than 1010, it is corrected by adding 0110. The procedure is repeated until all decimal digits are added.

The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry $K = 1$. The other six combinations from 1010 to 1111 that need a correction have a 1 in position Z_8. To distinguish them from binary 1000 and 1001 which also have a 1 in position Z_8, we specify further that either Z_4

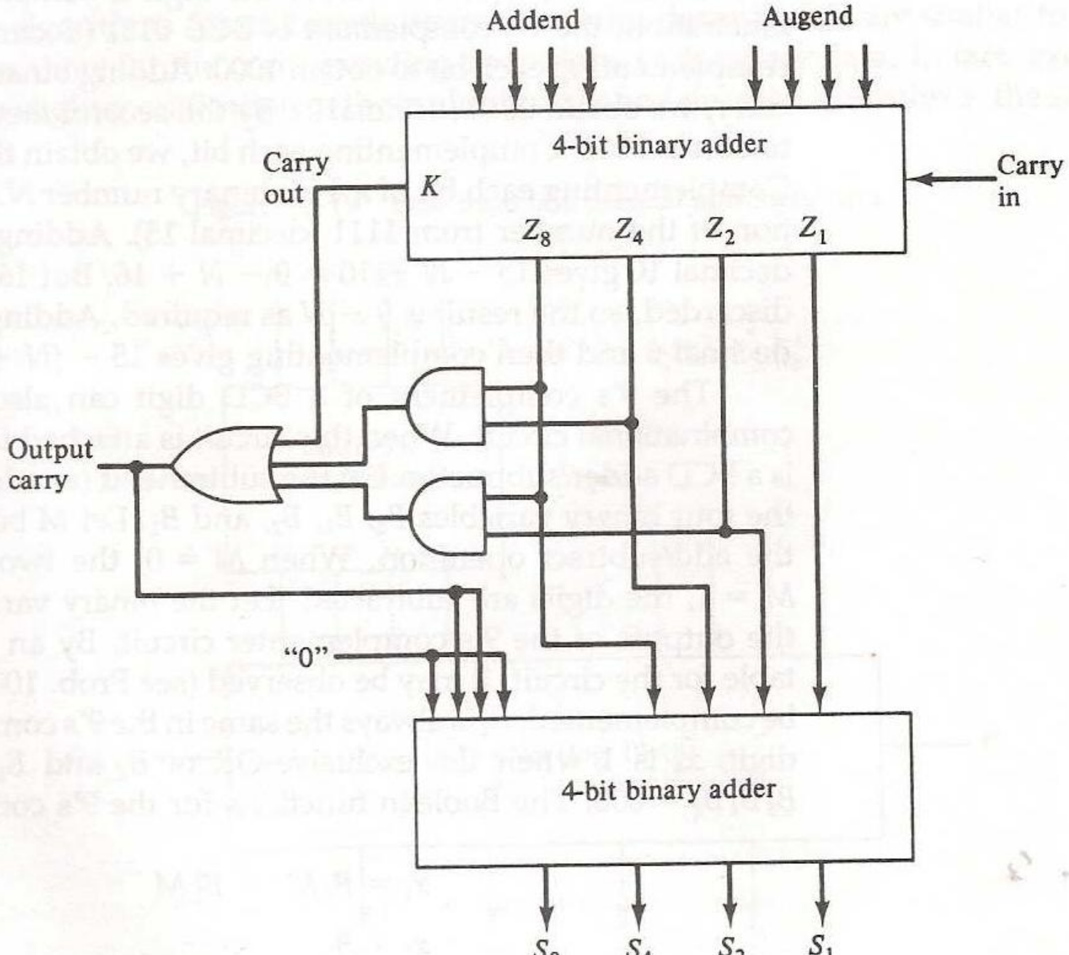
or Z_2 must have a 1. The condition for a correction and an output-carry can be expressed by the Boolean function

$$C = K + Z_8 Z_4 + Z_8 Z_2$$

When $C = 1$, it is necessary to add 0110 to the binary sum and provide an output-carry for the next stage.

A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder as shown in Fig. 10-18. The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output-carry is equal to 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output-carry generated from the bottom binary adder may be ignored, since it supplies information already available in the output-carry terminal.

Figure 10-18 Block diagram of BCD adder.



A decimal parallel-adder that adds *n* decimal digits needs *n* BCD adder stages with the output-carry from one stage connected to the input-carry of the next-higher-order stage. To achieve shorter propagation delays, BCD adders include the necessary circuits for carry look-ahead. Furthermore, the adder circuit for the correction does not need all four full-adders, and this circuit can be optimized.

BCD Subtraction

A straight subtraction of two decimal numbers will require a subtractor circuit that will be somewhat different from a BCD adder. It is more economical to perform the subtraction by taking the 9's or 10's complement of the subtrahend and adding it to the minuend. Since the BCD is not a self-complementing code, the 9's complement cannot be obtained by complementing each bit in the code. It must be formed by a circuit that subtracts each BCD digit from 9.

The 9's complement of a decimal digit represented in BCD may be obtained by complementing the bits in the coded representation of the digit provided a correction is included. There are two possible correction methods. In the first method, binary 1010 (decimal 10) is added to each complemented digit and the carry discarded after each addition. In the second method, binary 0110 (decimal 6) is added before the digit is complemented. As a numerical illustration, the 9's complement of BCD 0111 (decimal 7) is computed by first complementing each bit to obtain 1000. Adding binary 1010 and discarding the carry, we obtain 0010 (decimal 2). By the second method, we add 0110 to 0111 to obtain 1101. Complementing each bit, we obtain the required result of 0010. Complementing each bit of a 4-bit binary number *N* is identical to the subtraction of the number from 1111 (decimal 15). Adding the binary equivalent of decimal 10 gives $15 - N + 10 = 9 - N + 16$. But 16 signifies the carry that is discarded, so the result is $9 - N$ as required. Adding the binary equivalent of decimal 6 and then complementing gives $15 - (N + 6) = 9 - N$ as required.

The 9's complement of a BCD digit can also be obtained through a combinational circuit. When this circuit is attached to a BCD adder, the result is a BCD adder/subtractor. Let the subtrahend (or addend) digit be denoted by the four binary variables B_8, B_4, B_2, and B_1. Let M be a mode bit that controls the add/subtract operation. When $M = 0$, the two digits are added; when $M = 1$, the digits are subtracted. Let the binary variables x_8, x_4, x_2, and x_1 be the outputs of the 9's complementer circuit. By an examination of the truth table for the circuit, it may be observed (see Prob. 10-30) that B_1 should always be complemented; B_2 is always the same in the 9's complement as in the original digit; x_4 is 1 when the exclusive-OR of B_2 and B_4 is 1; and x_8 is 1 when $B_8 B_4 B_2 = 000$. The Boolean functions for the 9's complementer circuit are

$$x_1 = B_1 M' + B_1' M$$

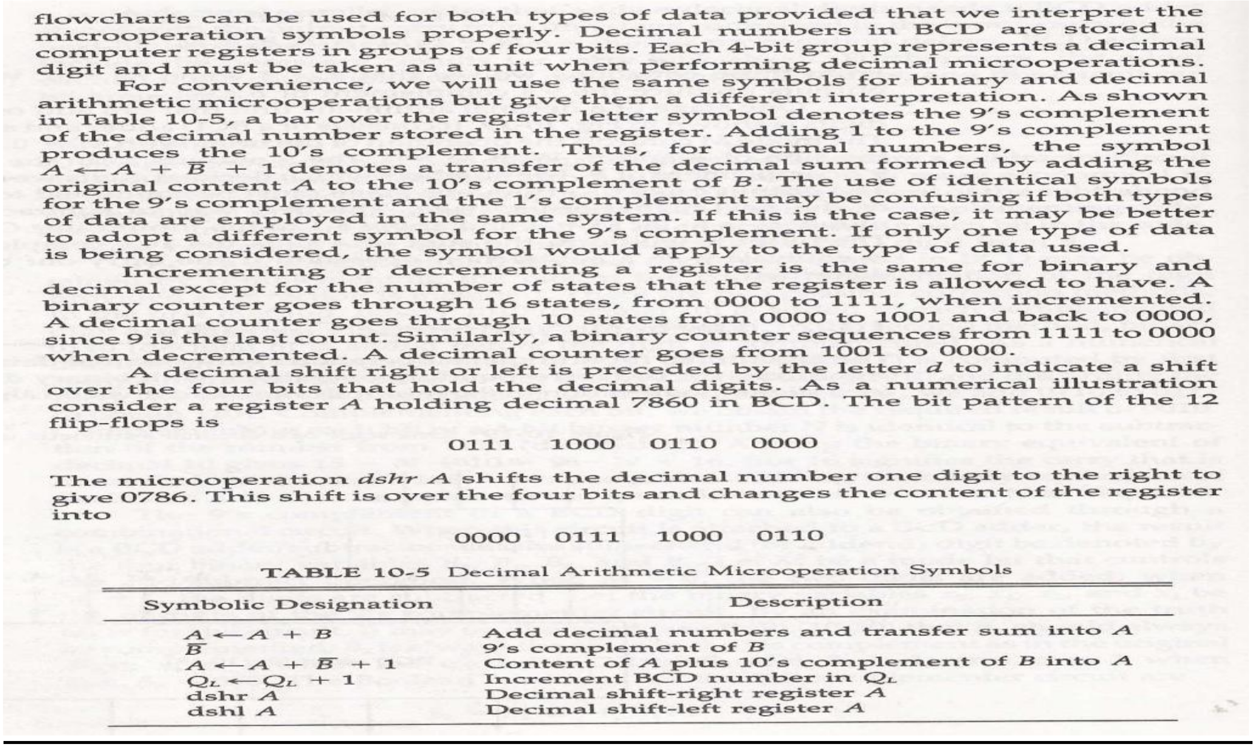
$$x_2 = B_2$$

$$x_4 = B_4 M' + (B_4' B_2 + B_4 B_2')M$$

$$x_8 = B_8 M' + B_8' B_4' B_2' M$$

From these equations we see that $x = B$ when $M = 0$. When $M = 1$, the x outputs produce the 9's complement of B.

One stage of a decimal arithmetic unit that can add or subtract two BCD digits is shown in Fig. 10-19. It consists of a BCD adder and a 9's complementer. The mode M controls the operation of the unit. With $M = 0$, the S outputs form the sum of A and B. With $M = 1$, the S outputs form the sum of A plus the 9's complement of B. For numbers with n decimal digits we need n such stages. The output carry C_{i+1} from one stage must be connected to the input carry C_i of the next-higher-order stage. The best way to subtract the two decimal numbers is to let $M = 1$ and apply a 1 to the input carry C_1 of the first stage. The outputs will form the sum of A plus the 10's complement of B, which is equivalent to a subtraction operation if the carry-out of the last stage is discarded.



flowcharts can be used for both types of data provided that we interpret the microoperation symbols properly. Decimal numbers in BCD are stored in computer registers in groups of four bits. Each 4-bit group represents a decimal digit and must be taken as a unit when performing decimal microoperations.

For convenience, we will use the same symbols for binary and decimal arithmetic microoperations but give them a different interpretation. As shown in Table 10-5, a bar over the register letter symbol denotes the 9's complement of the decimal number stored in the register. Adding 1 to the 9's complement produces the 10's complement. Thus, for decimal numbers, the symbol $A \leftarrow A + \overline{B} + 1$ denotes a transfer of the decimal sum formed by adding the original content A to the 10's complement of B. The use of identical symbols for the 9's complement and the 1's complement may be confusing if both types of data are employed in the same system. If this is the case, it may be better to adopt a different symbol for the 9's complement. If only one type of data is being considered, the symbol would apply to the type of data used.

Incrementing or decrementing a register is the same for binary and decimal except for the number of states that the register is allowed to have. A binary counter goes through 16 states, from 0000 to 1111, when incremented. A decimal counter goes through 10 states from 0000 to 1001 and back to 0000, since 9 is the last count. Similarly, a binary counter sequences from 1111 to 0000 when decremented. A decimal counter goes from 1001 to 0000.

A decimal shift right or left is preceded by the letter d to indicate a shift over the four bits that hold the decimal digits. As a numerical illustration consider a register A holding decimal 7860 in BCD. The bit pattern of the 12 flip-flops is

$$0111 \quad 1000 \quad 0110 \quad 0000$$

The microoperation *dshr A* shifts the decimal number one digit to the right to give 0786. This shift is over the four bits and changes the content of the register into

$$0000 \quad 0111 \quad 1000 \quad 0110$$

TABLE 10-5 Decimal Arithmetic Microoperation Symbols

Symbolic Designation	Description
$A \leftarrow A + B$	Add decimal numbers and transfer sum into A
$\overline{B}$	9's complement of B
$A \leftarrow A + \overline{B} + 1$	Content of A plus 10's complement of B into A
$Q_L \leftarrow Q_L + 1$	Increment BCD number in Q_L
dshr A	Decimal shift-right register A
dshl A	Decimal shift-left register A

Addition and Subtraction

The algorithm for addition and subtraction of binary signed-magnitude numbers applies also to decimal signed-magnitude numbers provided that we interpret the microoperation symbols in the proper manner. Similarly, the algorithm for binary signed-2's complement numbers applies to decimal signed-10's complement numbers. The binary data must employ a binary adder and a complementer. The decimal data must employ a decimal arithmetic unit capable of adding two BCD numbers and forming the 9's complement of the subtrahend as shown in Fig. 10-19.

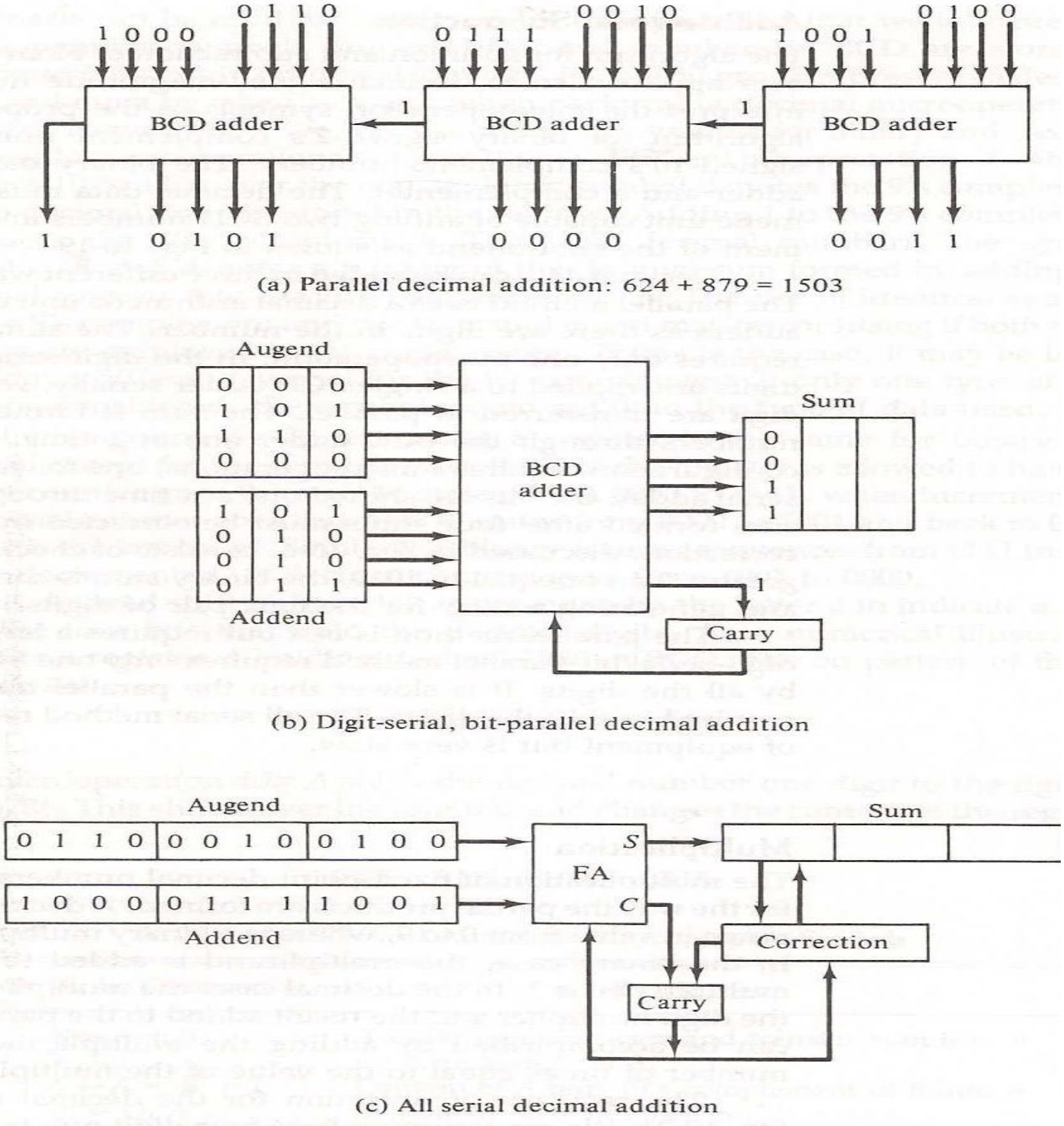
Decimal data can be added in three different ways, as shown in Fig. 10-20. The parallel method uses a decimal arithmetic unit composed of as many BCD adders as there are digits in the number. The sum is formed in parallel and requires only one microoperation. In the digit-serial bit-parallel method, the digits are applied to a single BCD adder serially, while the bits of each coded digit are transferred in parallel. The sum is formed by shifting the decimal numbers through the BCD adder one at a time. For *k* decimal digits, this configuration requires *k* microoperations, one for each decimal shift. In the all serial adder, the bits are shifted one at a time through a full-adder. The binary sum formed after four shifts must be corrected into a valid BCD digit. This correction, discussed in Sec. 10-6, consists of checking the binary sum. If it is greater than or equal to 1010, the binary sum is corrected by adding to it 0110 and generating a carry for the next pair of digits.

The parallel method is fast but requires a large number of adders. The digit-serial bit-parallel method requires only one BCD adder, which is shared by all the digits. It is slower than the parallel method because of the time required to shift the digits. The all serial method requires a minimum amount of equipment but is very slow.

Multiplication

The multiplication of fixed-point decimal numbers is similar to binary except for the way the partial products are formed. A decimal multiplier has digits that range in value from 0 to 9, whereas a binary multiplier has only 0 and 1 digits. In the binary case, the multiplicand is added to the partial product if the multiplier bit is 1. In the decimal case, the multiplicand must be multiplied by the digit multiplier and the result added to the partial product. This operation can be accomplished by adding the multiplicand to the partial product a number of times equal to the value of the multiplier digit.

The registers organization for the decimal multiplication is shown in Fig. 10-21. We are assuming here four-digit numbers, with each digit occupying four bits, for a total of 16 bits for each number. There are three registers, *A*, *B*, and *Q*, each having a corresponding sign flip-flop A_s, B_s, and Q_s.



(a) Parallel decimal addition: 624 + 879 = 1503

(b) Digit-serial, bit-parallel decimal addition

(c) All serial decimal addition

Figure 10-20 Three ways of adding decimal numbers.

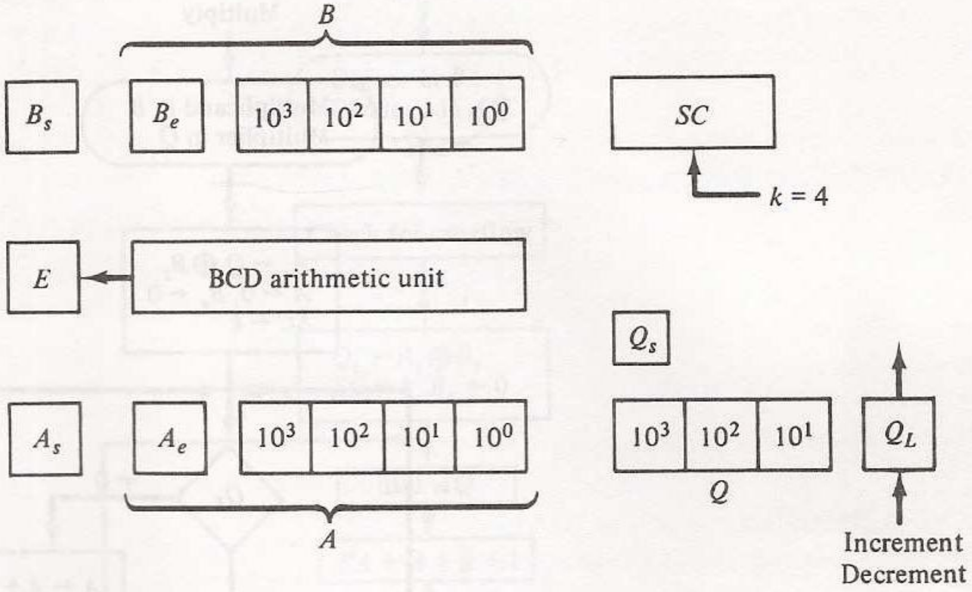


Figure 10-21 Registers for decimal arithmetic multiplication and division.

Registers A and B have four more bits designated by A_e and B_e that provide an extension of one more digit to the registers. The BCD arithmetic unit adds the five digits in parallel and places the sum in the five-digit A register. The end-carry goes to flip-flop E. The purpose of digit A_e is to accommodate an overflow while adding the multiplicand to the partial product during multiplication. The purpose of digit B_e is to form the 9's complement of the divisor when subtracted from the partial remainder during the division operation. The least significant digit in register Q is denoted by Q_L. This digit can be incremented or decremented.

A decimal operand coming from memory consists of 17 bits. One bit (the sign) is transferred to B_s and the magnitude of the operand is placed in the lower 16 bits of B. Both B_e and A_e are cleared initially. The result of the operation is also 17 bits long and does not use the A_e part of the A register.

The decimal multiplication algorithm is shown in Fig. 10-22. Initially, the entire A register and B_e are cleared and the sequence counter SC is set to a number k equal to the number of digits in the multiplier. The low-order digit of the multiplier in Q_L is checked. If it is not equal to 0, the multiplicand in B is added to the partial product in A once and Q_L is decremented. Q_L is checked again and the process is repeated until it is equal to 0. In this way, the multiplicand in B is added to the partial product a number of times equal to the multiplier digit. Any temporary overflow digit will reside in A_e and can range in value from 0 to 9.

Next, the partial product and the multiplier are shifted once to the right. This places zero in A_e and transfers the next multiplier quotient into Q_L. The process is then repeated k times to form a double-length product in AQ.

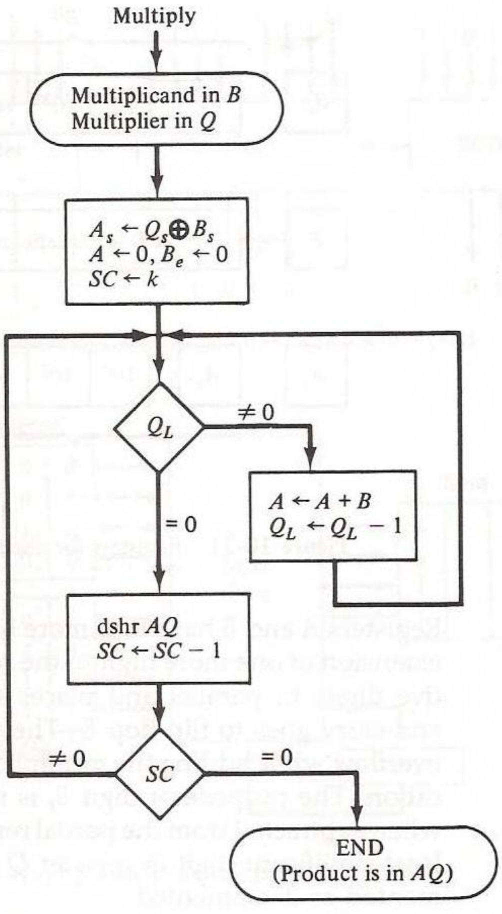


Figure 10-22 Flowchart for decimal multiplication.

Division

Decimal division is similar to binary division except of course that the quotient digits may have any of the 10 values from 0 to 9. In the restoring division method, the divisor is subtracted from the dividend or partial remainder as many times as necessary until a negative remainder results. The correct remainder is then restored by adding the divisor. The digit in the quotient reflects the number of subtractions up to but excluding the one that caused the negative difference.

The decimal division algorithm is shown in Fig. 10-23. It is similar to the algorithm with binary data except for the way the quotient bits are formed. The dividend (or partial remainder) is shifted to the left, with its most significant digit placed in A_e. The divisor is then subtracted by adding its 10's complement value. Since B_e is initially cleared, its complement value is 9 as required. The carry in E determines the relative magnitude of A and B. If $E = 0$, it signifies

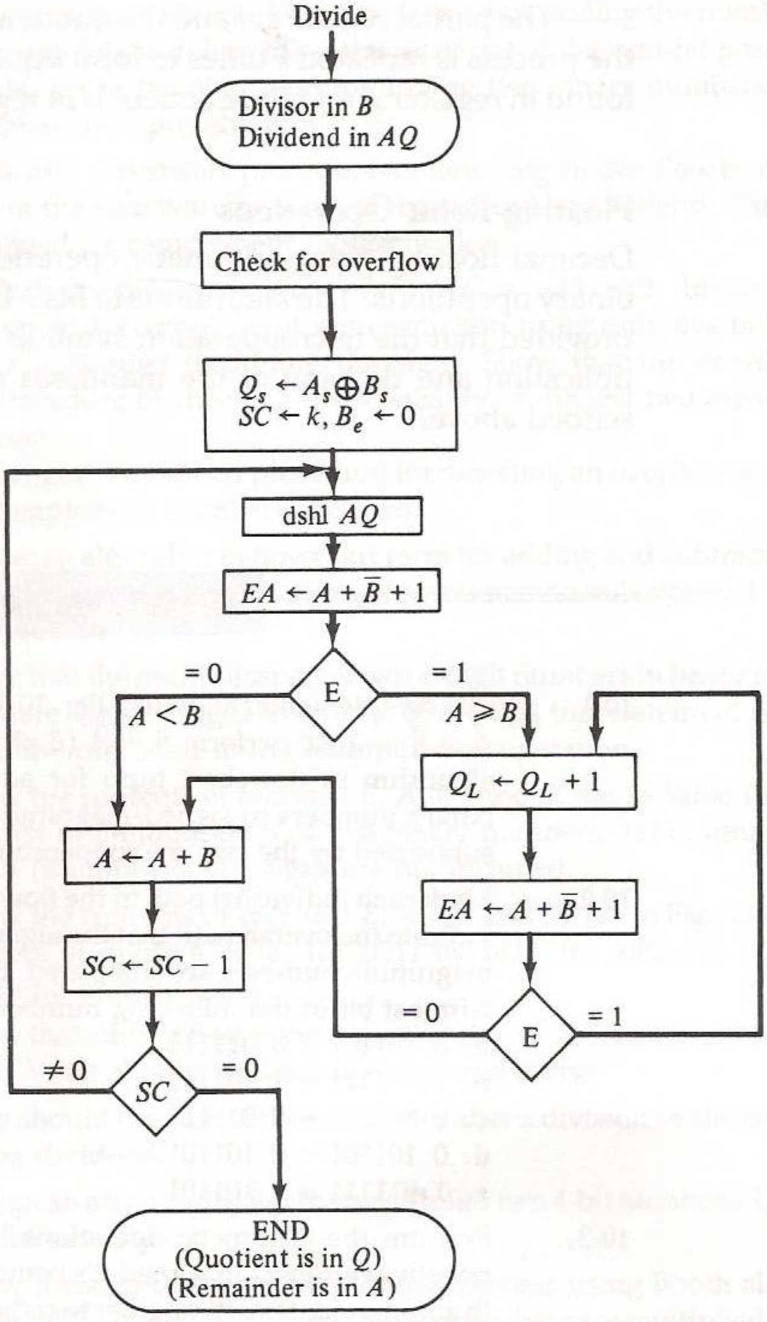


Figure 10-23 Flowchart for decimal division.

that $A < B$. In this case the divisor is added to restore the partial remainder and Q_L stays at 0 (inserted there during the shift). If $E = 1$, it signifies that $A \geq B$. The quotient digit in Q_L is incremented once and the divisor subtracted again. This process is repeated until the subtraction results in a negative difference which is recognized by E being 0. When this occurs, the quotient digit is not incremented but the divisor is added to restore the positive remainder. In this way, the quotient digit is made equal to the number of times that the partial remainder "goes" into the divisor.

The partial remainder and the quotient bits are shifted once to the left and the process is repeated k times to form k quotient digits. The remainder is then found in register A and the quotient is in register Q. The value of E is neglected.

Floating-Point Operations

Decimal floating-point arithmetic operations follow the same procedures as binary operations. The algorithms in Sec. 10-5 can be adopted for decimal data provided that the microoperation symbols are interpreted correctly. The multiplication and division of the mantissas must be done by the methods described above.

UNIT-IV

INPUT/OUTPUT ORGANIZATION AND MEMORY ORGANIZATION

Input-Output Organization: Input-Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt - Direct memory Access.

Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory.

Input-Output Organization: Input-Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt - Direct memory Access.

Input/output Organization:

- **Input-Output Interface**
- **Asynchronous Data Transfer**
- **Modes of Transfer**
- **Priority Interrupt**
- **Direct Memory Access (DMA).**

INPUT-OUTPUT INTERFACE

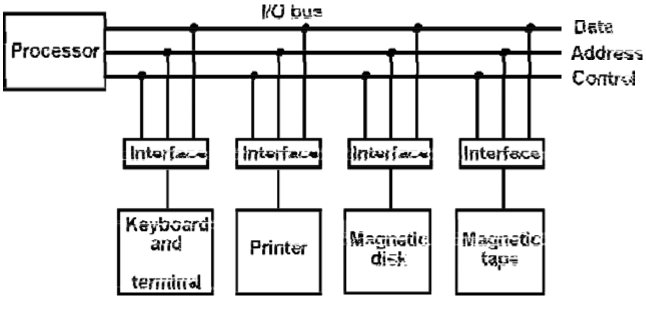
- Input-output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The purpose of the communication link is to resolve the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

  1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.
  2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be need.
  3. Data codes and formats in peripherals differ from the word format in the CPU and memory.
  4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

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

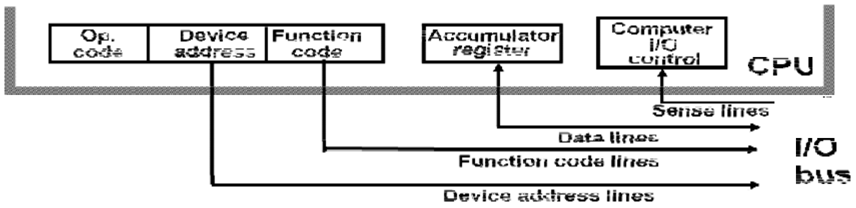
I/O BUS AND INTERFACE MODULES

- A typical communication link between the processor and several peripherals is shown in Fig.
- The I/O bus consists of data lines, address lines, and control lines.
- Each peripheral device has associated with it an interface unit. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller. It also synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller that operates the particular electromechanical device.

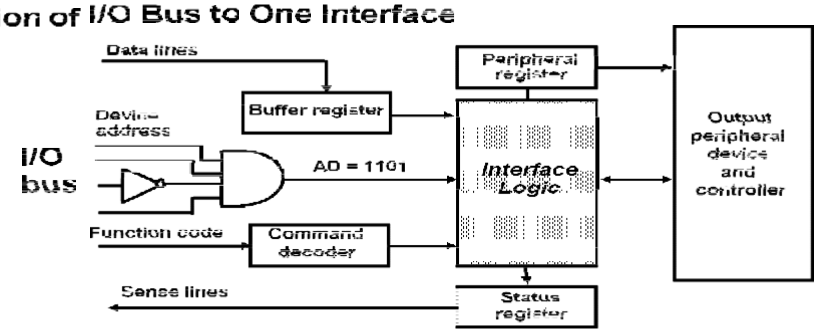


- To communicate with a particular device, the processor places a device address on the address lines. Each interface attached to the I/O bus contains an address decoder that monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the device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

Connection of I/O Bus to CPU



Connection of I/O Bus to One Interface



- At the same time the processor provides a function code in the control lines.
- There are four types of commands that an interface may receive. They are classified as control, status, data output, and data input.
- A control command is issued to activate the peripheral and to inform it what to do.
- A status command is used to test various status conditions in the interface and the peripheral.
- A data output command causes the interface to respond by transferring data from the bus into one of its registers.
- The data input command is the opposite of the data output. In this case the interface receives an item of data from the peripheral and places it in its buffer register.

I/O VERSUS MEMORY BUS

- In addition to communicating with I/O, the processor must communicate with the memory unit. Like the I/O bus, the memory bus contains data, address, and read/write control lines. There are three ways that computer buses can be used to communicate with memory and I/O:

1. Use two separate buses, one for memory and the other for I/O.
2. Use one common bus for both memory and I/O but have separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

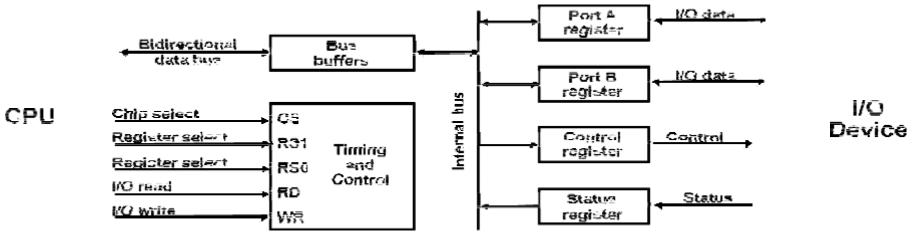
In the first method, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for I/O. This is done in computers that provide a separate I/O processor (IOP) in addition to the central processing unit (CPU). The memory communicates with both the CPU and the IOP through a memory bus. The IOP communicates also with the input and output devices through a separate I/O bus with its own address, data and control lines. The purpose of the IOP is to provide an independent pathway for the transfer of information between external devices and internal memory

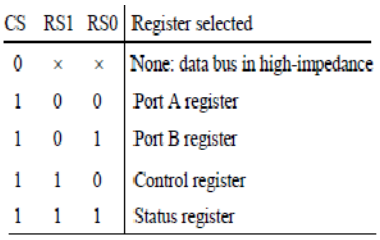
ISOLATED VERSUS MEMORY-MAPPED I/O

- Many computers use one common bus to transfer information between memory or I/O and the CPU. The distinction between a memory transfer and I/O transfer is made through separate read and write lines. The CPU specifies whether the address on the address lines is for a memory word or for an interface register by enabling one of two possible read or write lines. The I/O read and I/O write control lines are enabled during an I/O transfer. The memory read and memory write control lines are enabled during a memory transfer.
- In the **isolated I/O** configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/O read (for input) or I/O write (for output) control line. This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word. On the other hand, when the CPU is fetching an instruction or an operand from memory, it places the memory address on the address lines and enables the memory read or memory write control line. This informs the external components that the address is for a memory word and not for an I/O interface.
- The other alternative is to use the same address space for both memory and I/O. This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as **memory mapped I/O**. The computer treats an interface register as being part of the memory system.
- In a memory-mapped I/O organization there is no specific input or output instructions. The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words. Each interface is organized as a set of registers that respond to read and write requests in the normal address space. Typically, a segment of the total address space is reserved for interface registers, but in general, they can be located at any address as long as there is not also a memory word that responds to the same address.
- Computers with memory-mapped I/O can use memory-type instructions to access I/O data. It allows the computer to use the same instructions for either input-output transfers or for memory transfers.
- The advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from I/O registers.
- In a typical computer, there are more memory-reference instructions than I/O instructions. With memory mapped I/O all instructions that refer to memory are also available for I/O. .

EXAMPLE OF I/O INTERFACE

- An example of an I/O interface unit is shown in block diagram





CS	RS1	RS0	Register selected
0	×	×	None: data bus in high-impedance
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

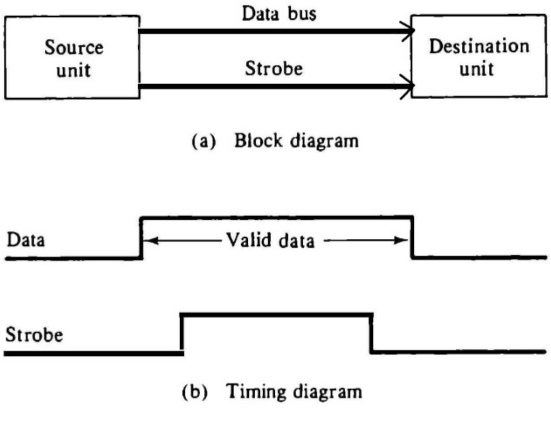
- It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus.
- The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively.
- The four registers communicate directly with the I/O device attached to the interface. The I/O data to and from the device can be transferred into either port A or Port B.
- The interface may operate with an output device or with an input device, or with a device that requires both input and output..
- A command is passed to the I/O device by sending a word to the appropriate interface register. In a system like this, the function code in the I/O bus is not needed because control is sent to the control register, status information is received from the status register, and data are transferred to and from ports A and B registers. Thus the transfer of data, control, and status information is always via the common data bus.
- The distinction between data, control, or status information is determined from the particular register with which the CPU communicates.
- The control register receives control information from the CPU. By loading appropriate bits into the control register, the interface and the I/O device attached to it can be placed in a variety of operating modes.
- The interface registers communicate with the CPU through the bidirectional data bus.
- The address bus selects the interface unit through the chip select and the two register select inputs. A circuit must be provided externally (usually, a decoder) to detect the address assigned to the interface registers. This circuit enables the chip select (CS) input when the interface is selected by the address bus. The two register select inputs RS1 and RS0 are usually connected to the two least significant lines of the lines address bus. These two inputs select one of the four registers in the interface as specified in the table accompanying the diagram.
- The content of the selected register is transfer into the CPU via the data bus when the I/O read signal is enabled. The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

ASYNCHRONOUS DATA TRANSFER

- The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator.
- If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous.
- In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other.
- Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted.
- One way of achieving this is by means of a **strobe pulse** supplied by one of the units to indicate to the other unit when the transfer has to occur. Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another control signal to acknowledge receipt of the data. This type of agreement between two independent units is referred to as **handshaking.**

STROBE CONTROL

- The strobe control method of asynchronous data transfer employs a single control line to time each transfer. The strobe may be activated by either the source or the destination unit.
- The data bus carries the binary information from source unit to the destination unit. Typically, the bus has multiple lines to transfer an entire byte or word. The strobe is a single line that informs the destination unit when a valid data word is available in the bus.
- As shown in the timing diagram the source unit first places the data on the data bus. After a brief delay to ensure that the data settle to a steady value, the source activates the strobe pulse.



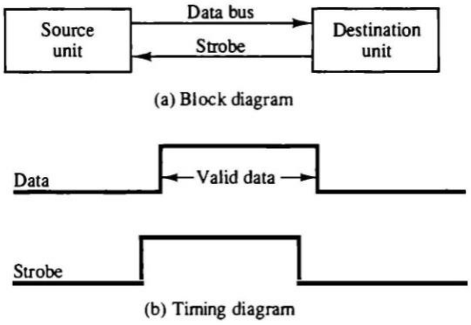
(a) Block diagram

(b) Timing diagram

Source-initiated strobe for data transfer.

- The information on the data bus and the strobe signal remain in the active state for a sufficient time period to allow the destination unit to receive the data. Often, the destination unit uses the falling edge of the strobe pulse to transfer the contents of the data bus into one of its internal registers.

- The following Figure shows a data transfer initiated by the destination unit.
- In this case the destination unit activates the strobe pulse, informing the source to provide the data. The source unit responds by placing the requested binary information on the data bus. The data must be valid and remain in the bus long enough for the destination unit to accept it. The falling edge of the strobe pulse can be used again to trigger a destination register. The destination unit then disables the strobe



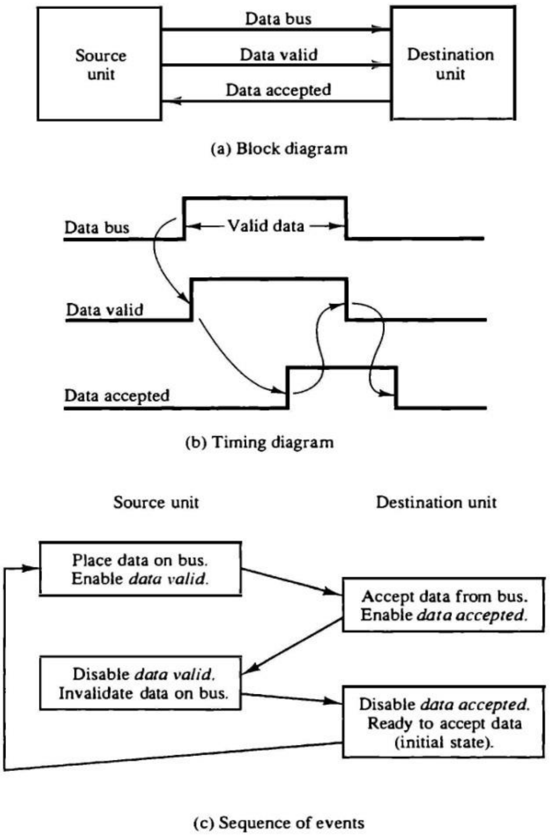
Destination-initiated strobe for data transfer.

- The transfer of data between the CPU and an interface unit is similar to the strobe transfer. Data transfer between an interface and an I/O device is commonly controlled by a set of handshaking lines

HANDSHAKING

- The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus. The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that initiates the transfer.
- The basic principle of the handshaking method of data transfer is as follows. One control line is in the same direction as the data flow in the bus from the source to the destination. It is used by the source unit to inform the destination unit whether there are valued data in the bus.
- The other control line is in the other direction from the destination to the source. It is used by the destination unit to inform the source whether it can accept data. The sequence of control during the transfer depends on the unit that initiates the transfer.
- The two handshaking lines are **data valid,** which is generated by the source unit, and **data accepted,** generated by the destination unit.
- The timing diagram shows the exchange of signals between the two units. In the destination-initiated transfer the source does not place data on the bus until after it receives the ready for data signal from the destination unit.
- The handshaking scheme provides a high degree of flexibility and reality because the successful completion of a data transfer relies on active participation by both units. If one unit is faulty, the data transfer will not be completed. Such an error can be detected by means of a timeout mechanism, which produces an alarm if the data transfer is not completed within a predetermined time. The timeout is implemented by means of an internal clock that starts

counting time when the unit enables one of its handshaking control signals. If the return handshake signal does not respond within a given time period, the unit assumes that an error has occurred


Source unit
Data bus
Data valid
Data accepted
Destination unit
Data bus
Valid data
Data valid
Data accepted
Source unit
Destination unit
Place data on bus. Enable *data valid*.
Accept data from bus. Enable *data accepted*.
Disable *data valid*. Invalidate data on bus.
Disable *data accepted*. Ready to accept data (initial state).

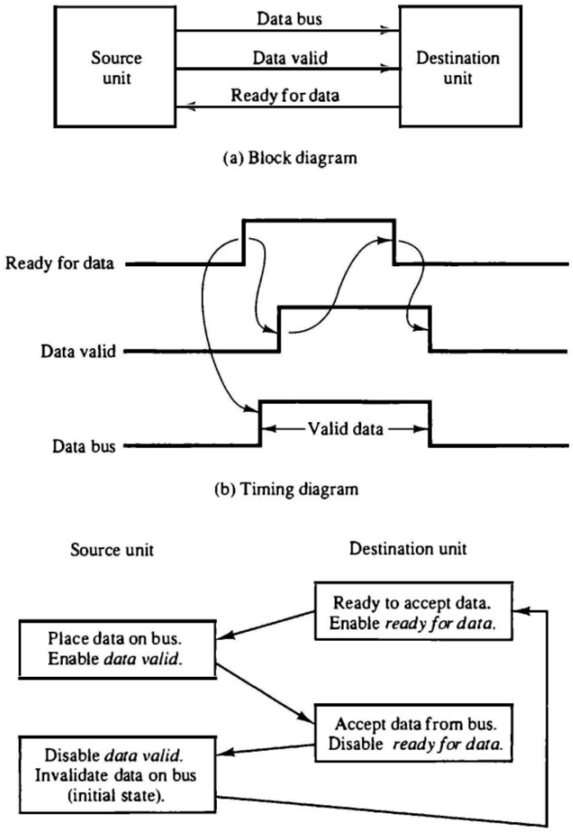
(a) Block diagram

(b) Timing diagram

(c) Sequence of events

Source-initiated transfer using handshaking.

Destination-initiated transfer using handshaking.


Source unit
Data bus
Data valid
Ready for data
Destination unit
Ready for data
Data valid
Data bus
Valid data
Source unit
Destination unit
Ready to accept data. Enable *ready for data*.
Place data on bus. Enable *data valid*.
Accept data from bus. Disable *ready for data*.
Disable *data valid*. Invalidate data on bus (initial state).

(a) Block diagram

(b) Timing diagram

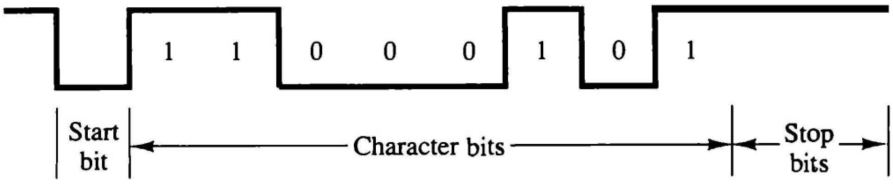
(c) Sequence of events

ASYNCHRONOUS SERIAL TRANSFER

- The transfer of data between two units may be done in parallel or serial. In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time.
- In serial data transmission, each bit in the message is sent in sequence one at a time.
- Parallel transmission is faster but requires many wires. It is used for short distances and where speed is important. Serial transmission is slower but is less expensive since it requires only one pair of conductors.
- Serial transmission can be synchronous or asynchronous. In synchronous transmission, the two units share a common clock frequency and bits are transmitted continuously at the rate dictated by the clock pulses. In long-distant serial transmission, each unit is driven by a separate clock of the same frequency. Synchronization signals are transmitted periodically between the two units to keep their clocks in step with each other.

- In asynchronous transmission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted.
- Serial asynchronous data transmission technique used in many interactive terminals employs special bits that are inserted at both ends of the character code. With this technique, each character consists of three parts: a start bit, the character bits, and stop bits.
- The convention is that the transmitter rests at the 1-state when no characters are transmitted. The first bit, called the start bit, is always a 0 and is used to indicate the beginning of a character. The last bit called the stop bit is always a 1.
- An example of this format is shown in Fig.

Asynchronous serial transmission.



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

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

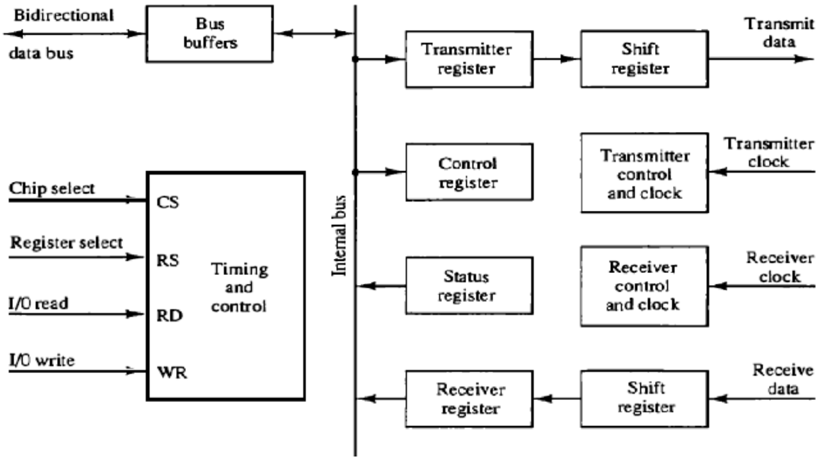
  2. The initiation of a character transmission is detected from the start bit, which is always 0.

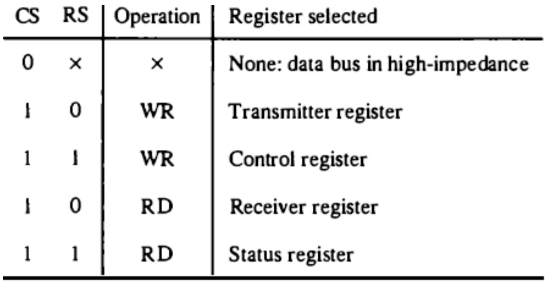
  3. The character bits always follow the start bit.

  4. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.
- Using these rules, the receiver can detect the start bit when the line gives from 1 to 0. A clock in the receiver examines the line at proper bit times. The receiver knows the transfer rate of the bits and the number of character bits to accept. After the character bits are transmitted, one or two stop bits are sent. The stop bits are always in the 1-state and frame the end of the character to signify the idle or wait state.
- At the end of the character the line is held at the 1-state for a period of at least one or two bit times so that both the transmitter and receiver can resynchronize. The length of time that the line stays in this state depends on the amount of time required for the equipment to resynchronize.
- Some older electromechanical terminals use two stop bits, but newer terminals use one stop bit.
- The line remains in the 1-state until another character is transmitted. The stop time ensures that a new character will not follow for one or two bit times.

Asynchronous Communication Interface

- The block diagram of an asynchronous communication interface is shown in Fig.





CS	RS	Operation	Register selected
0	×	×	None: data bus in high-impedance
1	0	WR	Transmitter register
1	1	WR	Control register
1	0	RD	Receiver register
1	1	RD	Status register

Block diagram of a typical asynchronous communication interface.

- It functions as both a transmitter and a receiver. The interface is initialized for a particular mode of transfer by means of a control byte that is loaded into its control register. The transmitter register accepts a data byte from the CPU through the data bus. This byte is transferred to a shift register for serial transmission. The receiver portion receives serial information into another shift register, and when a complete data byte is accumulated, it is transferred to the receiver register. The CPU can select the receiver register to read the byte through the data bus.
- The bits in the status register are used for input and output flags and for recording certain errors that may occur during the transmission. The CPU can read the status register to check the status of the flag bits and to determine if any errors have occurred.
- The chip select and the read and write control lines communicate with the CPU. The chip select (CS) input is used to select the interface through the address bus. The register select (RS) is associated with the read (RD) and write (WR) controls. Two registers are write-only and two are read-only. The register selected is a function of the RS value and the RD and WR status, as listed in the table accompanying the diagram.
- The operation of the asynchronous communication interface is initialized by the CPU by sending a byte to the control register. The initialization procedure places the interface in a specific mode of operation as it defines certain parameters such as the baud rate to use, how many bits are in each character, whether to generate and check parity, and how many stop bits

are appended to each character. Two bits in the status register are used as flags. One bit is used to indicate whether the transmitter register is empty and another bit is used to indicate whether the receiver register is full.

- The operation of the transmitter portion of the interface is as follows. The CPU reads the status register and checks the flag to see if the transmitter register is empty. If it is empty, the CPU transfers a character to the transmitter register and the interface clears the flag to mark the register full. The first bit in the transmitter shift register is set to 0 to generate a start bit. The character is transferred in parallel from the transmitter register to the shift register and the appropriate number of stop bits are appended into the shift register. The transmitter register is then marked empty. The character can now be transmitted one bit at a time by shifting the data in the shift register at the specified baud rate. The CPU can transfer another character to the transmitter register after checking the flag in the status register. The interface is said to be double buffered because a new character can be loaded as soon as the previous one starts transmission.
- The operation of the receiver portion of the interface is similar. The receive data input is in the 1-state when the line is idle. The receiver control monitors the receive-data line for a 0 signal to detect the occurrence of a start bit. Once a start bit has been detected, the incoming bits of the character are shifted into the shift register at the prescribed baud rate. After receiving the data bits, the interface checks for the parity and stop bits. The character without the start and stop bits is then transferred in parallel from the shift register to the receiver register. The flag in the status register is set to indicate that the receiver register is full. The CPU reads the status register and checks the flag, and if set, it reads the data from the receiver register. The interface checks for any possible errors during transmission and sets appropriate bits in the status register. The CPU can read the status register at any time to check if any errors have occurred. Three possible errors that the interface checks during transmission are parity error, framing error, and overrun error. Parity error occurs if the number of l's in the received data is not the correct parity. A framing error occurs if the right number of stop bits is not detected at the end of the received character. An overrun error occurs if the CPU does not read the character from the receiver register before the next one becomes available in the shift register. Overrun error results in a loss of characters in the received data stream.

First-In, First-Out Buffer

A first-in, first-out (FIFO) buffer is a memory unit that stores information in such a manner that the item first in is the item first out. A FIFO buffer comes with separate input and output terminals. The important feature of this buffer is that it can input data and output data at two different rates and the output data are always in the same order in which the data entered the buffer. When placed between two units, the FIFO can accept data from the source unit at one rate of transfer and deliver the data to the destination unit at another rate. If the source unit is slower than the destination unit, the buffer can be filled with data at a slow rate and later emptied at the higher rate. If the source is faster than the destination, the FIFO is useful for those cases where the source data arrive in bursts that fill out the buffer but the time between bursts is long enough for the destination unit to empty some or all the information from the buffer. Thus a FIFO buffer can be useful in some applications when data are transferred asynchronously. It piles up data as they come in and gives them away in the same order when the data are needed.

The logic diagram of a typical 4 × 4 FIFO buffer is shown in Fig. 11-9. It consists of four 4-bit registers RI, $I = 1, 2, 3, 4$, and a control register with

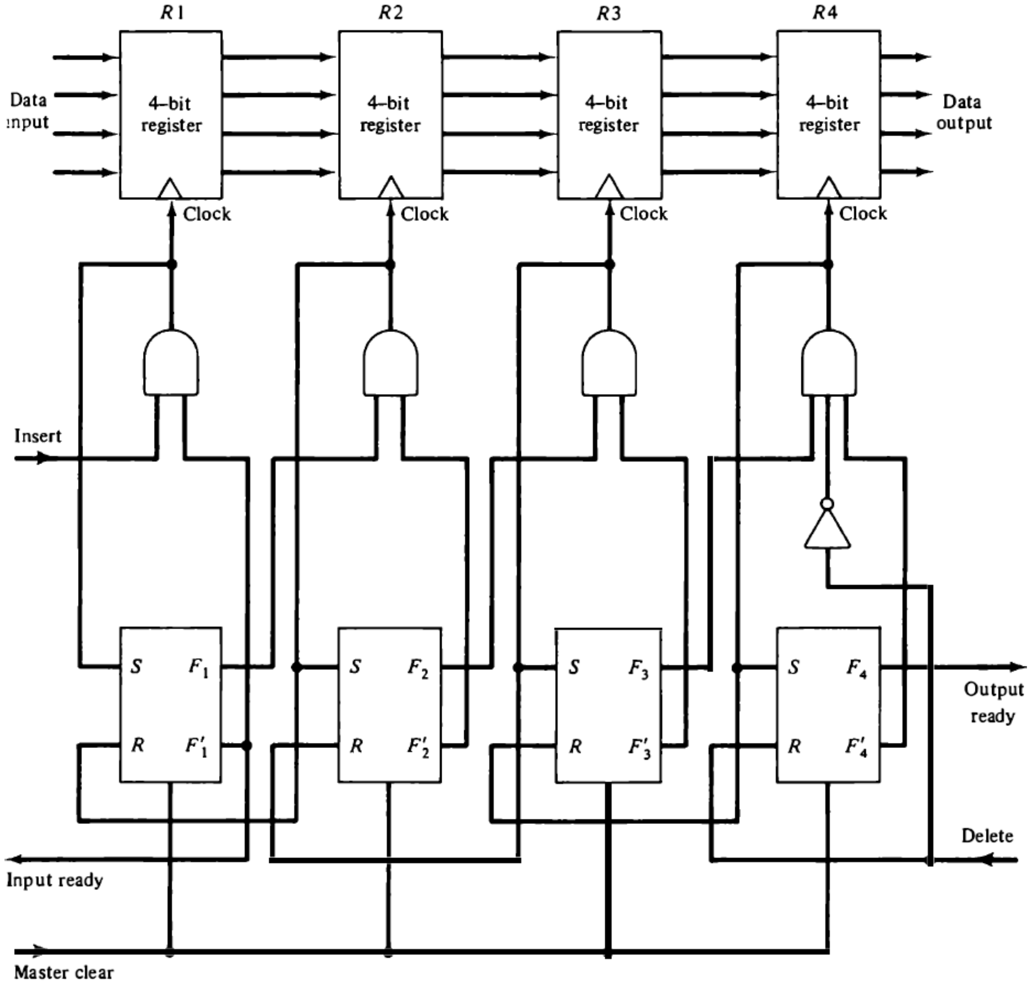


Figure 11-9 Circuit diagram of 4 × 4 FIFO buffer.

flip-flops F_i, $i = 1, 2, 3, 4$, one for each register. The FIFO can store four words of four bits each. The number of bits per word can be increased by increasing the number of bits in each register and the number of words can be increased by increasing the number of registers.

A flip-flop F_i in the control register that is set to 1 indicates that a 4-bit data word is stored in the corresponding register *RI*. A 0 in F_i indicates that the corresponding register does not contain valid data. The control register directs the movement of data through the registers. Whenever the F_i bit of the control register is set ($F_i = 1$) and the F_{i+1} bit is reset ($F'_{i+1} = 1$), a clock is generated causing register $R(I + 1)$ to accept the data from register *RI*. The same clock transition sets F_{i+1} to 1 and resets F_i to 0. This causes the control flag to move one position to the right together with the data. Data in the registers move down the FIFO toward the output as long as there are empty locations ahead of it. This ripple-through operation stops when the data reach a register *RI* with the next flip-flop F_{i+1} being set to 1, or at the last register *R*4. An overall master clear is used to initialize all control register flip-flops to 0.

Data are inserted into the buffer provided that the *input ready* signal is enabled. This occurs when the first control flip-flop F_1 is reset, indicating that register *R*1 is empty. Data are loaded from the input lines by enabling the clock in *R*1 through the *insert* control line. The same clock sets F_1, which disables the *input ready* control, indicating that the FIFO is now busy and unable to accept more data. The ripple-through process begins provided that *R*2 is empty. The data in *R*1 are transferred into *R*2 and F_1 is cleared. This enables the *input ready* line, indicating that the inputs are now available for another data word. If the FIFO is full, F_1 remains set and the *input ready* line stays in the 0 state. Note that the two control lines *input ready* and *insert* constitute a destination-initiated pair of handshake lines.

The data falling through the registers stack up at the output end. The *output ready* control line is enabled when the last control flip-flop F_4 is set, indicating that there are valid data in the output register *R*4. The output data from *R*4 are accepted by a destination unit, which then enables the *delete* control signal. This resets F_4, causing *output ready* to disable, indicating that the data on the output are no longer valid. Only after the *delete* signal goes back to 0 can the data from *R*3 move into *R*4. If the FIFO is empty, there will be no data in *R*3 and F_4 will remain in the reset state. Note that the two control lines *output ready* and *delete* constitute a source-initiated pair of handshake lines.

MODES OF TRANSFER

- Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit.
- Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; other transfer the data directly to and from the memory unit.
- Data transfer to and from peripherals may be handled in one of three possible modes:
  1. Programmed I/O
  2. Interrupt-initiated I/O
  3. Direct memory access (DMA)
- Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device.
- In the programmed I/O method, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly. It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. In the meantime the CU can proceed to execute another program. The interface meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing Transfer of data under programmed I/O is between CPU and peripheral.
- In direct memory access (DMA), the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. When the transfer is made, the DMA requests memory cycles through the memory bus. When the request is granted by the memory controller, the DMA transfers the data directly

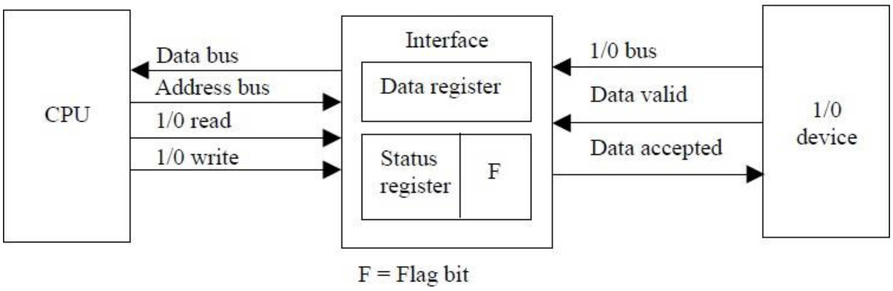
into memory. The CPU merely delays its memory access operation to allow the direct memory I/O transfer. Since peripheral speed is usually slower than processor speed, I/O-memory transfers are infrequent compared to processor access to memory.

- Many computers combine the interface logic with the requirements for direct memory access into one unit and call it an I/O processor (IOP). The IOP can handle many peripherals through a DMPA and interrupt facility. In such a system, the computer is divided into three separate modules: the memory unit, the CPU, and the IOP.

EXAMPLE OF PROGRAMMED I/O

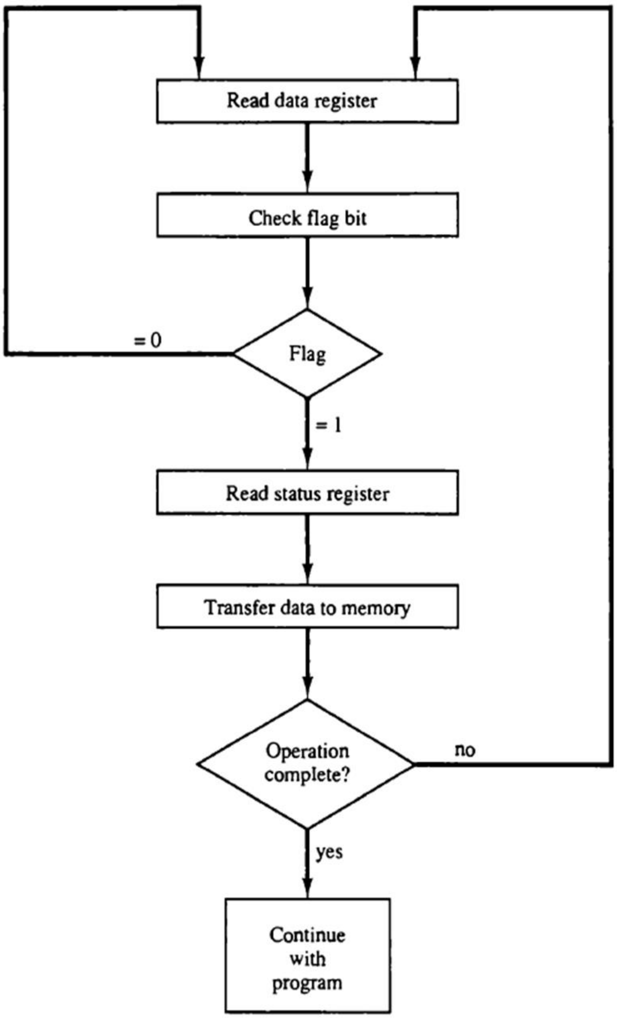
- In the programmed I/O method, the I/O device does not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU, and a store instruction to transfer the data from the CPU to memory. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.
- An example of data transfer from an I/O device through an interface into the CPU is shown in Fig.

Data transfer form I/O device to CPU



- The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a bit in the status register that we will refer to as an F or “flag” bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface.
- A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device. This is done by reading the status register into a CPU register and checking the value of the flag bit. If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

- A flowchart of the program that must be written for the CPU is shown in Fig.



Flowchart for CPU program to input data.

It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

1. Read the status register.

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

3. Read the data register.

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

INTERRUPT-INITIATED I/O

- An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility. While the CPU is running a program, it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set.
- The CPU deviates from what it is doing to take care of the input or output transfer. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.
- The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the

required I/O transfer. The way that the processor chooses the branch address of the service routine varies from tone unit to another. In principle, there are two methods for accomplishing this. One is called vectored interrupt and the other, no vectored interrupt. In a non vectored interrupt, the branch address is assigned to a fixed location in memory. In a vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the interrupt vector. In some computers the interrupt vector is the first address of the I/O service routine. In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored.

SOFTWARE CONSIDERATIONS

- The previous discussion was concerned with the basic hardware needed to interface I/O devices to a computer system. A computer must also have software routines for controlling peripherals and for transfer of data between the processor and peripherals. I/O routines must issue control commands to activate the peripheral and to check the device status to determine when it is ready for data transfer. Once ready, information is transferred item by item until all the data are transferred. In some cases, a control command is then given to execute a device function such as stop tape or print characters. Error checking and other useful steps often accompany the transfers.
- In interrupt-controlled transfers, the I/O software must issue commands to the peripheral to interrupt when ready and to service the interrupt when it occurs. In DMA transfer, the I/O software must initiate the DMA channel to start its operation.
- Software control of input-output equipment is a complex undertaking. For this reason I/O routines for standard peripherals are provided by the manufacturer as part of the computer system. They are usually included within the operating system. Most operating systems are supplied with a variety of I/O programs to support the particular line of peripherals offered for the computer. I/O routines are usually available as operating system procedures and the user refers to the established routines to specify the type of transfer required without going into detailed machine language programs.

PRIORITY INTERRUPT

- Data transfer between the CPU and an I/O device is initiated by the CPU. The CPU cannot start the transfer unless the device is ready to communicate with the CPU. The readiness of the device can be determined from an interrupt signal.
- Numbers of I/O devices are attached to the computer; several sources will request service simultaneously. The first task of the interrupt system is to identify the source of the interrupt and decide which device to service first
- A priority interrupts is a system to determine which interrupt is to be served first when two or more requests are made simultaneously. Also determines which interrupts are permitted to interrupt the computer while another is being serviced. Higher priority interrupts can make requests while servicing a lower priority interrupt
- Establishing the priority of simultaneous interrupts can be done by software or hardware.
- Priority Interrupt by Software(Polling)

- Priority is established by the order of polling the devices(interrupt sources)

- Flexible since it is established by software

- Low cost since it needs a very little hardware

- Very slow

- Priority Interrupt by Hardware

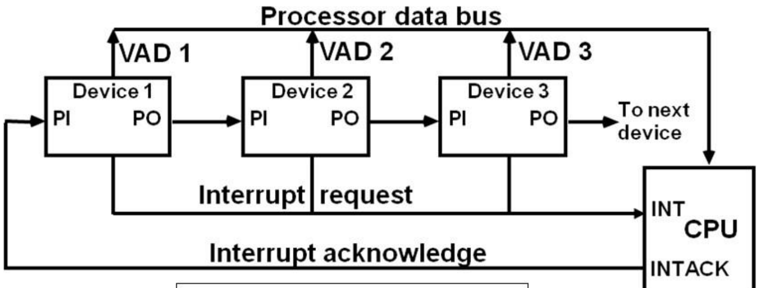
- Require a priority interrupt manager which accepts all the interrupt requests to determine the highest priority request

- Fast since identification of the highest priority interrupt request is identified by the hardware. Each interrupt source has its own interrupt vector to access directly to its own service routine

- The hardware priority function can be established by either a serial or a parallel connection of interrupt lines. The serial connection is also known as the daisy chaining method.

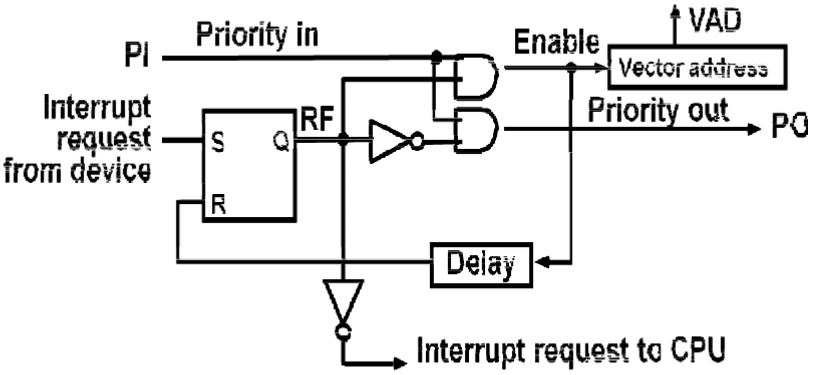
DAISY-CHAINING PRIORITY

- The daisy-chaining method of establishing priority consists of a serial connection of all devices that request an interrupt. The device with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain.



DAISY-CHAIN PRIORITY INTERRUPT

- The interrupt request line is common to all devices and forms a wired logic connection. If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU. When no interrupts are pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU.
- The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its PI (priority in) input. The acknowledge signal passes on to the next device through the PO (priority out) output only if device 1 is not requesting an interrupt.
- If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output. It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.
- The device with PI = 1 and PO = 0 is the one with the highest priority that is requesting an interrupt, and this device places its VAD on the data bus.
- The following figure shows the internal logic that must be included with in each device when connected in the daisy-chaining scheme.



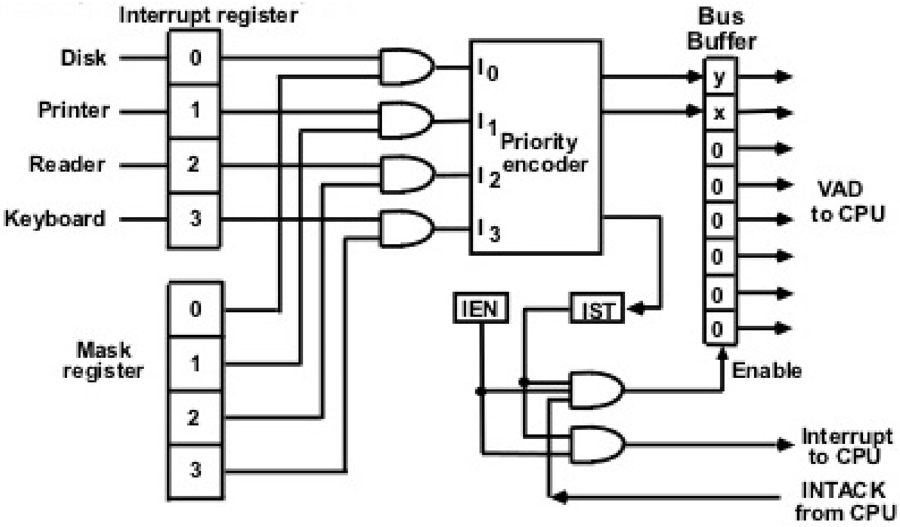
PI	RF	PO	Enable
0	0	0	0
0	1	0	0
1	0	1	0
1	1	1	1

Fig: One stage of the daisy- chain priority arrangement

- The device sets its RF flip-flop when it wants to interrupt the CPU. The output of the RF flip-flop goes through an open-collector inverter, a circuit that provides the wired logic for the common interrupt line.
- If PI = 0, both PO and the enable line to VAD are equal to 0, irrespective of the value of RF.
- If PI = 1 and RF = 0, then PO = 1 and the vector address is disabled. This condition passes the acknowledge signal to the next device through PO.
- The device is active when PI = 1 and RF = 1. This condition places a 0 in PO and enables the vector address for the data bus. It is assumed that each device has its own distinct vector address.
- The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

PARALLEL PRIORITY INTERRUPT

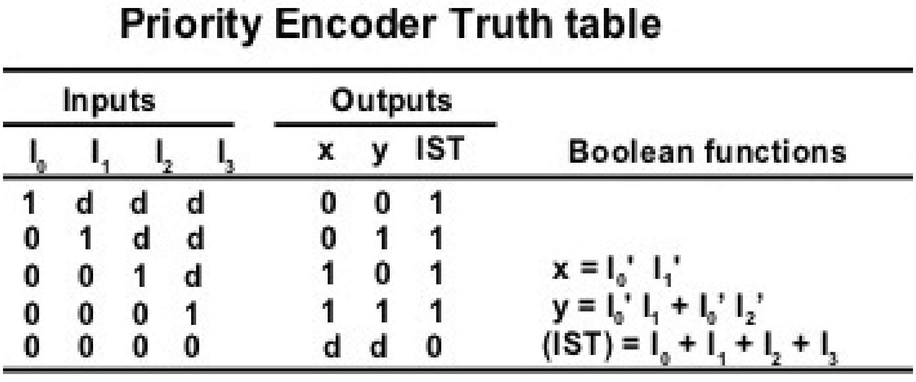
- The parallel priority interrupt method uses a register whose bits are set separately by the interrupt signal from each device.
- Priority is established according to the position of the bits in the register. In addition to the interrupt register the circuit may include a mask register whose purpose is to control the status of each interrupt request.
- The mask register can be programmed to disable lower-priority interrupts while a higher-priority device is being serviced. It can also provide a facility that allows a high-priority device to interrupt the CPU while a lower-priority device is being serviced.
- The priority logic for a system of four interrupt sources is shown in Fig.



- It consists of an interrupt register whose individual bits are set by external conditions and cleared by program instructions.
- The mask register has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register.
- Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way an interrupt is recognized only if its corresponding mask bit is set to 1 by the program.
- The priority encoder generates two bits of the vector address, which is transferred to the CPU.
- Another output from the encoder sets an interrupt status flip-flop IST when an interrupt that is not masked occurs.
- The interrupt enable flip-flop IEN can be set or cleared by the program to provide an overall control over the interrupt system.
- The outputs of IST ANDed with IEN provide a common interrupt signal for the CPU.
- The interrupt acknowledge INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus.

Priority Encoder

- The priority encoder is a circuit that implements the priority function. The logic of the priority encoder is such that if two or more inputs arrive at the same time, the input having the highest priority will take precedence.



Priority Encoder Truth table

Inputs				Outputs			
I_0	I_1	I_2	I_3	x	y	IST	Boolean functions
1	d	d	d	0	0	1	
0	1	d	d	0	1	1	
0	0	1	d	1	0	1	$x = I_0' \; I_1'$
0	0	0	1	1	1	1	$y = I_0' I_1 + I_0' I_2'$
0	0	0	0	d	d	0	$(IST) = I_0 + I_1 + I_2 + I_3$

Interrupt Cycle

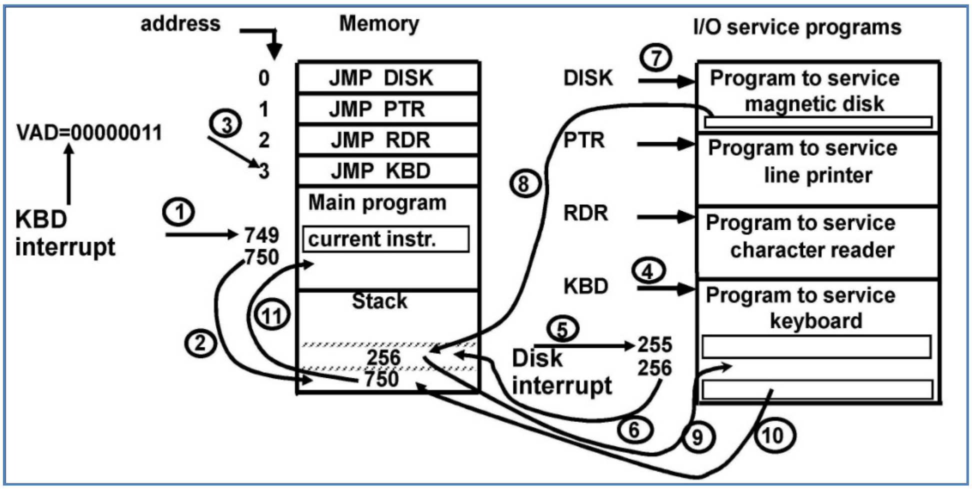
- The interrupt enable flip-flop IEN can be set or cleared by program instructions.
- When IEN is cleared, the interrupt request coming from IST is neglected by the CPU.
- The program-controlled IEN bit allows the programmer to choose whether to use the interrupt facility. If an instruction to clear IEN has been inserted in the program, it means that the user does not want his program to be interrupted. An instruction to set IEN indicates that the interrupt facility will be used while the current program is running.
- Most computers include internal hardware that clears IEN to 0 every time an interrupt is acknowledged by the processor
- At the end of each instruction cycle the CPU checks IEN and the interrupt signal from IST. If either is equal to 0, control continues with the next instruction.
- If both IEN and IST are equal to 1, the CPU goes to an interrupt cycle.
- During the interrupt cycle the CPU performs the following sequence of microoperations:

SP ← SP-1	Decrement stack pointer
M[SP] ←PC	Push PC into stack
INTACK ←1	Enable interrupt acknowledge
PC ← VAD	Transfer vector address to PC
lEN ←0	Disable further interrupts

Go to fetch next instruction

Software Routines

- A priority interrupt system is a combination of hardware and software techniques
- The following figure shows the programs that must reside in memory for handling the interrupt system.



Initial and Final Operations

- Each interrupt service routine must have an initial and final set of operations for controlling the registers in the hardware interrupt system
- **Initial Sequence**

  [1] Clear lower level Mask reg. bits

  [2] IST ←0

  [3] Save contents of CPU registers

  [4] IEN←1

  [5] Go to Interrupt Service Routine
- **Final Sequence**

  [1] IEN ←0

  [2] Restore CPU registers

  [3] Clear the bit in the Interrupt Reg

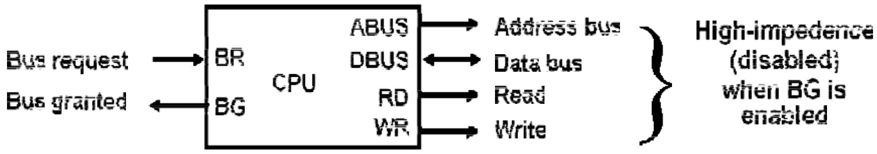
  [4] Set lower level Mask reg. bits

  [5] Restore return address into PC, and IEN ←1
- The initial and final operations are referred to as **overhead operations** or **housekeeping chores**. They are not part of the service program proper but are essential for processing interrupts.
- All overhead operations can be implemented by software. This is done by inserting the proper instructions at the beginning and at the end of each service routine. Some of the overhead operations can be done automatically by the hardware

DIRECT MEMORY ACCESS (DMA):

- The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This transfer technique is called **direct memory access (DMA)**.
- During DMA transfer, the CPU is idle and has no control of the memory buses.
- A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.
- The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals.

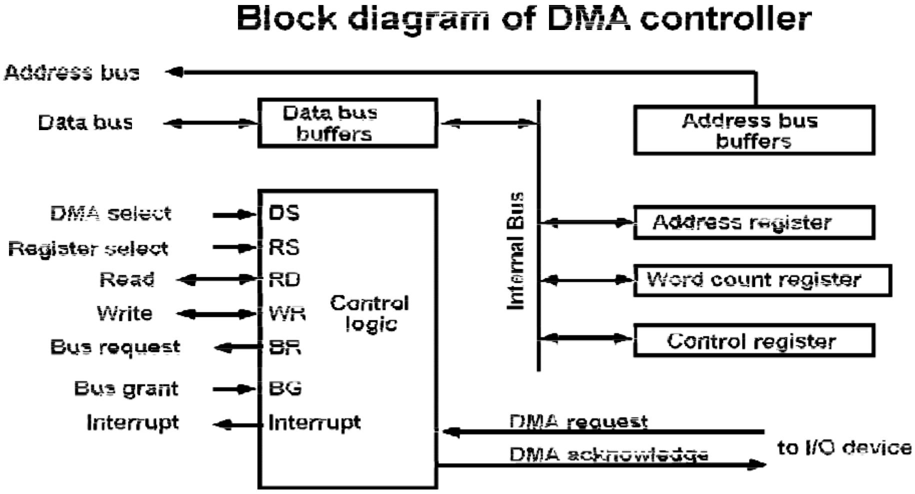
CPU bus signals for DMA transfer



- The bus request (BR) input is used by the DMA controller to request the CPU to cease control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance.
- The CPU activates the Bus grant (BG) output to inform the external DMA that the buses are in the high-impedance state. The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention. When the DMA terminates the transfer, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation.
- When the DMA takes control of the bus system, it communicates directly with the memory. The transfer can be made in several ways. In DMA **burst transfer**, a block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses. This mode of transfer is needed for fast devices such as magnetic disks, where data transmission cannot be stopped or slowed down until an entire block is transferred.
- An alternative technique called **cycle stealing** allows the DMA controller to transfer one data word at a time after which it must return control of the buses to the CPU. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to “steal” one memory cycle.

DMA CONTROLLER

- The following figure shows the block diagram of a typical DMA controller



- The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs. The RD (read) and WR (write) inputs are bidirectional.
- When the BG (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers.
- When BG = 1, the CPU has relinquished(ceased) the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control.
- The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.
- The DMA controller has three registers: an address register, a word count register, and a control register. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory.
- The word count register is incremented after each word that is transferred to memory. The word count register holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero.
- The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus the CPU can read from or write into the DMA registers under program control via the data bus.
- The DMA is first initialized by the CPU. After that, the DMA starts and continues to transfer data between memory and peripheral unit until an entire block is transferred. The initialization process is essentially a program consisting of I/O instructions that include the address for

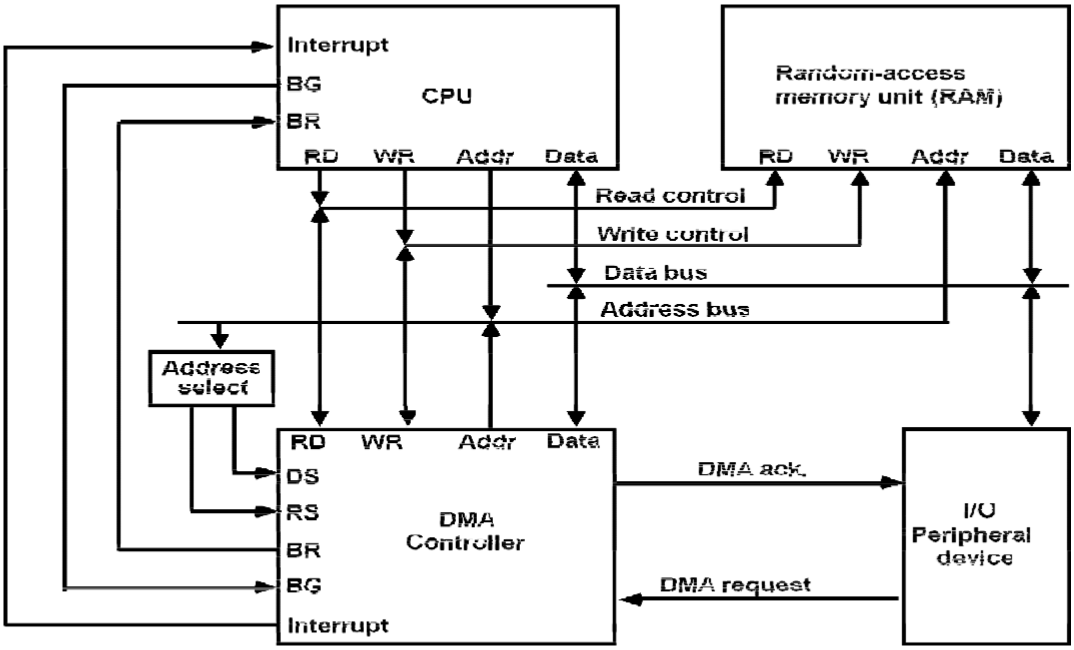
selecting particular DMA registers. The CPU initializes the DMA by sending the following information through the data bus:

1. The starting address of the memory block where data are available (for read) or where data are to be stored (for write)
2. The word count, which is the number of words in the memory block
3. Control to specify the mode of transfer such as read or write
4. A control to start the DMA transfer

- The starting address is stored in the address register. The word count is stored in the word count register, and the control information in the control register.
- Once the DMA is initialized, the CPU stops communicating with the DMA unless it receives an interrupt signal or if it wants to check how many words have been transferred.

DMA Transfer

- The position of the DMA controller among the other components in a computer system is illustrated in following fig.



- The CPU communicates with the DMA through the address and data buses as with any interface unit. The DMA has its own address, which activates the DS and RS lines.
- The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory.
- When the peripheral device sends a DMA request, the DMA controller activates the BR line, informing the CPU to relinquish the buses. The CPU responds with its BG line, informing the DMA that its buses are disabled.
- The DMA then puts the current value of its address register into the address bus, initiates the RD or WR signal, and sends a DMA acknowledge to the peripheral device.

- Note that the RD and WR lines in the DMA controller are bidirectional. The direction of transfer depends on the status of the BG line. When BG = 0, the RD and WR are input lines allowing the CPU to communicate with the internal DMA registers. When BG = 1, the RD and WR and output lines from the DMA controller to the random-access memory to specify the read or write operation for the data.
- When the peripheral device receives a DMA acknowledge, it puts a word in the data bus (for write) or receives a word from the data bus (for read). Thus the DMA controls the read or write operations and supplies the address for the memory.
- The peripheral unit can then communicate with memory through the data bus for direct transfer between the two units while the CPU is momentarily disabled.
- For each word that is transferred, the DMA increments its address register and decrements its word count register. If the word count does not reach zero, the DMA checks the request line coming from the peripheral.
- For a high-speed device, the line will be active as soon as the previous transfer is completed. A second transfer is then initiated, and the process continues until the entire block is transferred.
- If the peripheral speed is slower, the DMA request line may come somewhat later. In this case the DMA disables the bus request line so that the CPU can continue to execute its program. When the peripheral requests a transfer, the DMA requests the buses again.
- If the word count register reaches zero, the DMA stops any further transfer and removes its bus request. It also informs the CPU of the termination by means of an interrupt.
- When the CPU responds to the interrupt, it reads the content of the word count register. The zero value of this register indicates that all the words were transferred successfully. The CPU can read this register at any time to check the number of words already transferred.
- A DMA controller may have more than one channel. In this case, each channel has a request and acknowledges pair of control signals which are connected to separate peripheral devices. Each channel also has its own address register and word count register within the DMA controller.
- A priority among the channels may be established so that channels with high priority are serviced before channels with lower priority.
- DMA transfer is very useful in many applications.
- It is used for fast transfer of information between magnetic disks and memory.
- It is also useful for updating the display in an interactive terminal.

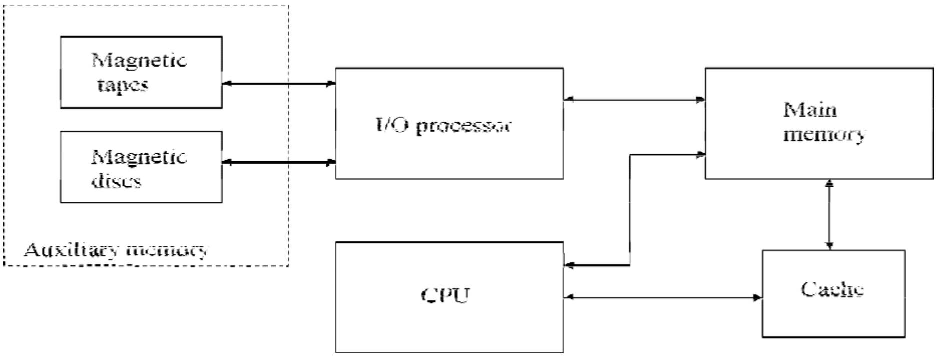
Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory.

Memory Organization:

- **Memory Hierarchy**
- **Main Memory**
- **Auxiliary Memory**
- **Associative Memory**
- **Cache Memory**
- **Virtual Memory.**

MEMORY HIERARCHY

- ✓ The memory unit is an essential component in any digital computer since it is needed for storing programs and data. A very small computer with a limited application may be able to fulfill its intended task without the need of additional storage capacity.
- ✓ Most general-purpose computers would run more efficiently if they were equipped with additional storage beyond the capacity of the main memory.
- ✓ It is more economical to use low-cost storage devices to serve as a backup for storing the information that is not currently used by the CPU.
- ✓ The memory unit that communicates directly with the CPU is called the **main memory**. Devices that provide backup storage are called **auxiliary memory**. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. They are used for storing system programs, large data files, and other backup information. Only programs and data currently needed by the processor reside in main memory. All other information is stored in auxiliary memory and transferred to main memory when needed.
- ✓ The memory hierarchy system consists of all storage devices employed in a computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high-speed processing logic.



Memory hierarchy in computer system

- ✓ The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor.
- ✓ When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.
- ✓ A special very-high speed memory called a **cache** is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic.

- ✓ CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory.
- ✓ A technique used to compensate for the mismatch in operating speeds is to employ in extremely fast, small cache between the CPU and main memory whose access time is close to processor logic clock cycle time.
- ✓ The reason for having two or three levels of memory hierarchy is economics.
- ✓ As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer.
- ✓ The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system.
- ✓ Auxiliary and cache memories are used for different purposes. The cache holds those parts of the program and data that are most heavily used, while the auxiliary memory holds those parts that are not presently used by the CPU. Moreover, the CPU has direct access to both cache and main memory but not to auxiliary memory. The transfer from auxiliary to main memory is usually done by means of direct memory access of large blocks of data. The typical access time ratio between cache and main memory is about 1 to 7. For example, a typical cache memory may have an access time of 100ns, while main memory access time may be 700ns. Auxiliary memory average access time is usually 1000 times that of main memory. Block size in auxiliary memory typically ranges from256 to 2048 words, while cache block size is typically from 1 to 16 words.
- ✓ Many operating systems are designed to enable the CPU to process a number of independent programs concurrently. This concept, called **multiprogramming**, refers to the existence of two or more programs in different parts of the memory hierarchy at the same time.
- ✓ In a multiprogramming system, when one program is waiting for input or output transfer, there is another program ready to utilize the CPU.
- ✓ Computer programs are sometimes too long to be accommodated in the total space available in main memory.
- ✓ When the program or a segment of the program is to be executed, it is transferred to main memory to be executed by the CPU.
- ✓ It is the task of the operating system to maintain in main memory a portion of this information that is currently active.
- ✓ The part of the computer system that supervises the flow of information between auxiliary memory and main memory is called the **memory management system.**

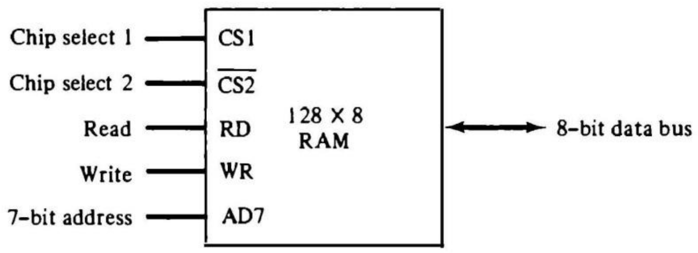
MAIN MEMORY

- ✓ The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation.
- ✓ The principal technology used for the main memory is based on semiconductor integrated circuits.
- ✓ Integrated circuit RAM chips are available in two possible operating modes, **static** and **dynamic.** The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to unit. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charge on the capacitors tends to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory.
- ✓ The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip.
- ✓ The static RAM is easier to use and has shorted read and write cycles.
- ✓ Most of the main memory in a general-purpose computer is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips.
- ✓ RAM refers to a random-access memory, but it is used to designate a read/write memory to distinguish it from a read-only memory, although ROM is also random access.
- ✓ RAM is used for storing the bulk of the programs and data that are subject to change. ROM is used for storing programs that are permanently resident in the computer
- ✓ The ROM portion of main memory is needed for storing an initial program called a **bootstrap loader**. The bootstrap loader is a program whose function is to start the computer software operating when power is turned on.
- ✓ Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is turned off and on again.
- ✓ The startup of a computer consists of turning the power on and starting the execution of an initial program. Thus when power is turned on, the hardware of the computer sets the program counter to the first address of the bootstrap loader. The bootstrap program loads a portion of the operating system from disk to main memory and control is then transferred to the operating system, which prepares the computer for general use.
- ✓ RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size. Ex: 1024 × 8 memory can be constructed with 128 × 8 RAM chips and 512 × 8 ROM chips.

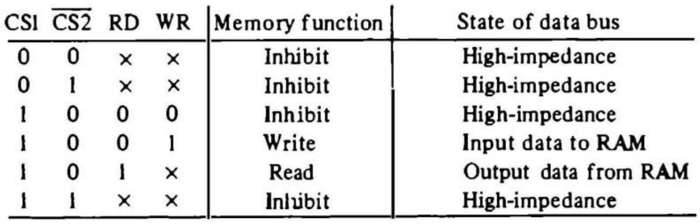
RAM AND ROM CHIPS

- ✓ A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation or from CPU to memory during a write operation.
- ✓ A bidirectional bus can be constructed with three-state buffers.
- ✓ The block diagram of a RAM chip is shown in Fig.

Typical RAM chip.



(a) Block diagram



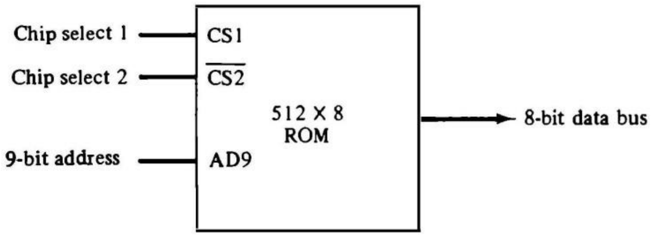
CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

- ✓ The capacity of the memory is 128 words of eight bits (one byte) per word. This requires a 7-bit address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor. The availability of more than one control input to select the chip facilitates the decoding of the address lines when multiple chips are used in the microcomputer.
- ✓ The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations or read or write.
- ✓ The unit is in operation only when CS1 = 1 and $\overline{CS2}$ = 0.
- ✓ If the chip select inputs are not enabled, or if they are enabled but the read but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state.
- ✓ When CS1 = 1 and $\overline{CS2}$ = 0, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines.

✓ When the RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.

✓ A ROM chip is organized externally in a similar manner. ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in Fig.



Typical ROM chip.

✓ The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The two chip select inputs must be CS1 = 1 and $\overline{CS2}$ = 0 for the unit to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because the unit can only read. Thus when the chip is enabled by the two select inputs, the byte selected by the address lines appears on the data bus.

MEMORY ADDRESS MAP

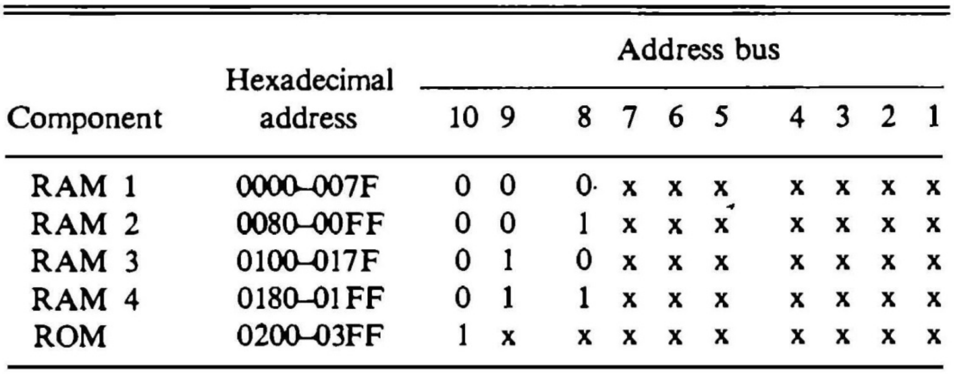
✓ The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM. The interconnection between memory and processor is then established form knowledge of the size of memory needed and the type of RAM and ROM chips available.

✓ A memory address map, is a pictorial representation of assigned address space for each chip in the system.

✓ To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM.

✓ The memory address map for this configuration is shown in Table.

Memory Address Map for Microprocomputer

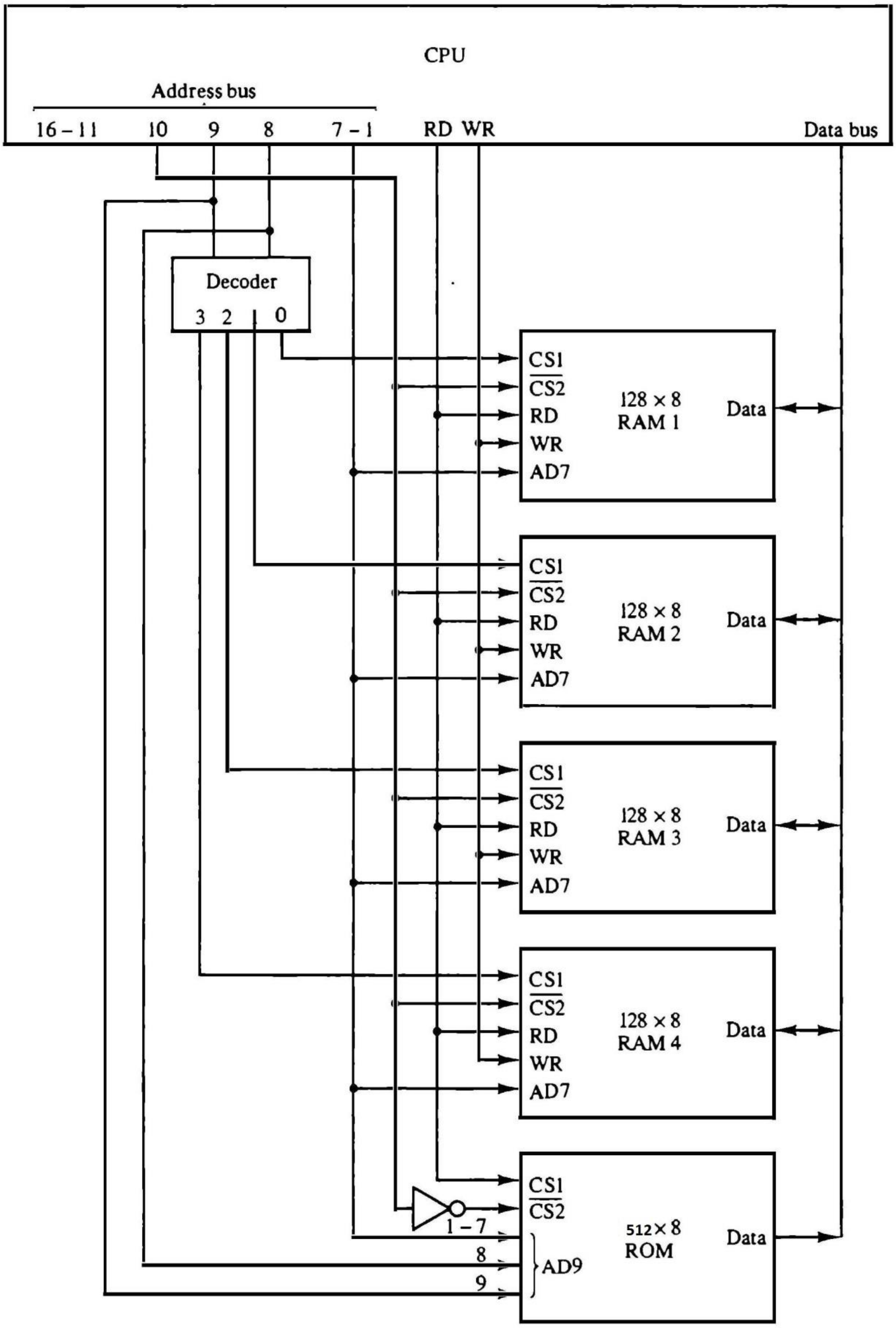


Component	Hexadecimal address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000–007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080–00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100–017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180–01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200–03FF	1	x	x	x	x	x	x	x	x	x

- ✓ The small x’s under the address bus lines designate those lines that must be connected to the address inputs in each chip.
- ✓ The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines.
- ✓ It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations.
- ✓ The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.
- ✓ The first hexadecimal digit represents lines 13 to 16 and is always 0. The next hexadecimal digit represents lines 9 to 12, but lines 11 and 12 are always 0. The range of hexadecimal addresses for each component is determined from the x’s associated with it. These x’s represent a binary number that can range from an all-0’s to an all-1’s value.

MEMORY CONNECTION TO CPU

- ✓ RAM and ROM chips are connected to a CPU through the data and address buses.
- ✓ The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs.
- ✓ The connection of memory chips to the CPU is shown in Fig. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM.
- ✓ Each RAM receives the seven low-order bits of the address bus to select one of 128 possible bytes. The particular RAM chip selected is determined from lines 8 and 9 in the address bus. This is done through a 2 × 4 decoder whose outputs go to the CS1 input in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected. When 01, the second RAM chip is selected, and so on.
- ✓ The RD and WR outputs from the microprocessor are applied to the inputs of each RAM chip.
- ✓ The selection between RAM and ROM is achieved through bus line 10. The RAMs are selected when the bit in this line is 0, and the ROM when the bit is 1. The other chip select input in the ROM is connected to the RD control line for the ROM chip to be enabled only during a read operation.
- ✓ Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. This assigns addresses 0 to 511 to RAM and 512 to 1023 to ROM.
- ✓ The data bus of the ROM has only an output capability, whereas the data bus connected to the RAMs can transfer information in both directions.



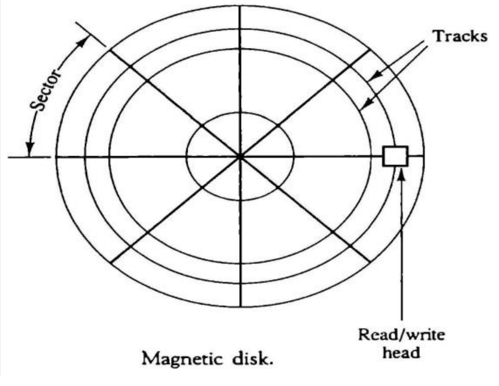
Memory connection to the CPU.

AUXILIARY MEMORY:

- ✓ The most common auxiliary memory devices used in computer systems are magnetic disks and magnetic tapes. Other components used, but not as frequently, are magnetic drums, magnetic bubble memory, and optical disks.
- ✓ The important characteristics of any device are its access mode, access time, transfer rate, capacity, and cost.
- ✓ The average time required to reach a storage location in memory and obtain its contents is called the access time. The access time consists of a seek time required to position the read-write head to a location and a transfer time required to transfer data to or from the device.
- ✓ Auxiliary storage is organized in records or blocks. A record is a specified number of characters or words. Reading or writing is always done on entire records. The transfer rate is the number of characters or words that the device can transfer per second, after it has been positioned at the beginning of the record.
- ✓ Magnetic drums and disks are quite similar in operation. Both consist of high-speed rotating surfaces coated with a magnetic recording medium. The rotating surface of the drum is a cylinder and that of the disk, a round flat plate. Bits are recorded as magnetic spots on the surface as it passes a stationary mechanism called a write head. Stored bits are detected by a change in magnetic field produced by a recorded spot on the surface as it passes through a read head.

MAGNETIC DISKS

- ✓ A magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material. Often both sides of the disk are used and several disks may be stacked on one spindle with read/write heads available on each surface.
- ✓ All disks rotate together at high speed and are not stopped or started from access purposes.
- ✓ Bits are stored in the magnetized surface in spots along concentric circles called tracks. The tracks are commonly divided into sections called sectors. In most systems, the minimum quantity of information which can be transferred is a sector.
- ✓ Some units use a single read/write head from each disk surface. The track address bits are used by a mechanical assembly to move the head into the specified track position before reading or writing.



Magnetic disk.

- ✓ In other disk systems, separate read/write heads are provided for each track in each surface. The address can then select a particular track electronically through a decoder circuit. This type of unit is more expensive and is found only in very large computer systems.
- ✓ A disk system is addressed by address bits that specify the disk number, the disk surface, the sector number and the track within the sector.
- ✓ After the read/write heads are positioned in the specified track, the system has to wait until the rotating disk reaches the specified sector under the read/write head.
- ✓ Information transfer is very fast once the beginning of a sector has been reached.
- ✓ Disks may have multiple heads and simultaneous transfer of bits from several tracks at the same time.
- ✓ A track in a given sector near the circumference is longer than a track near the center of the disk. If bits are recorded with equal density, some tracks will contain more recorded bits than others. To make all the records in a sector of equal length, some disks use a variable recording density with higher density on tracks near the center than on tracks near the circumference. This equalizes the number of bits on all tracks of a given sector.
- ✓ Disks that are permanently attached to the unit assembly and cannot be removed by the occasional user are called hard disks. A disk drive with removable disks is called a floppy disk.
- ✓ The disks used with a floppy disk drive are small removable disks made of plastic coated with magnetic recording material. There are two sizes commonly used, with diameters of 5.25 and 3.5 inches. The 3.5-inch disks are smaller and can store more data than can the 5.25-inch disks.

MAGNETIC TAPE

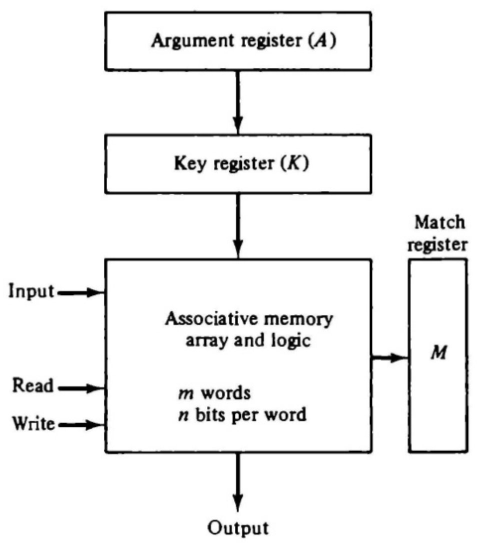
- ✓ The Magnetic tape itself is a strip of plastic coated with a magnetic recording medium. Bits are recorded as magnetic spots on the tape along several tracks. Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit.
- ✓ Read/write heads are mounted one in each track so that data can be recorded and read as a sequence of characters.
- ✓ Magnetic tape units can be stopped, started to move forward or in reverse, or can be rewound.
- ✓ Gaps of unrecorded tape are inserted between records where the tape can be stopped. The tape starts moving while in a gap and attains its constant speed by the time it reaches the next record.
- ✓ Each record on tape has an identification bit pattern at the beginning and end. By reading the bit pattern at the beginning, the tape control identifies the record number. By reading the bit pattern at the end of the record, the control recognizes the beginning of a gap. A tape unit is addressed by specifying the record number of characters in the record. Records may be of fixed or variable length.

ASSOCIATIVE MEMORY

- ✓ Many data-processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent.
- ✓ The number of accesses to memory depends on the location of the item and the efficiency of the search algorithm. Many search algorithms have been developed to minimize the number of accesses while searching for an item in a random or sequential access memory.
- ✓ The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address.
- ✓ A memory unit accessed by content is called an ***associative memory or content addressable memory (CAM).***
- ✓ When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locates all words which match the specified content and marks them for reading.
- ✓ An associative memory is more expensive than a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument. For this reason, associative memories are used in applications where the search time is very critical and must be very short.

HARDWARE ORGANIZATION

Block diagram of associative memory.



- ✓ The block diagram of an associative memory is shown in Fig.
- ✓ It consists of a memory array and logic for *m* words with *n* bits per word. The argument register A and key register K each have n bits, one for each bit of a word. The match register M has m bits, one for each memory word.
- ✓ Each word in memory is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register.
- ✓ After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched.
- ✓ Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

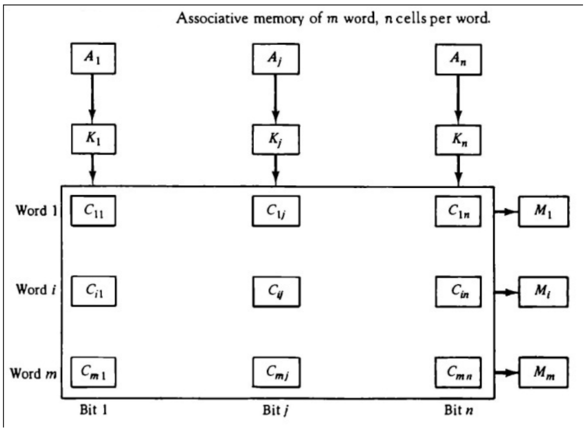
- ✓ The key register provides a mask for choosing a particular field or key in the argument word. The entire argument is compared with each memory word if the key register contains all 1's. Otherwise, only those bits in the argument that have 1's in their corresponding position of the key register are compared.
- ✓ To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has 1's in these positions.

  A 101 111100

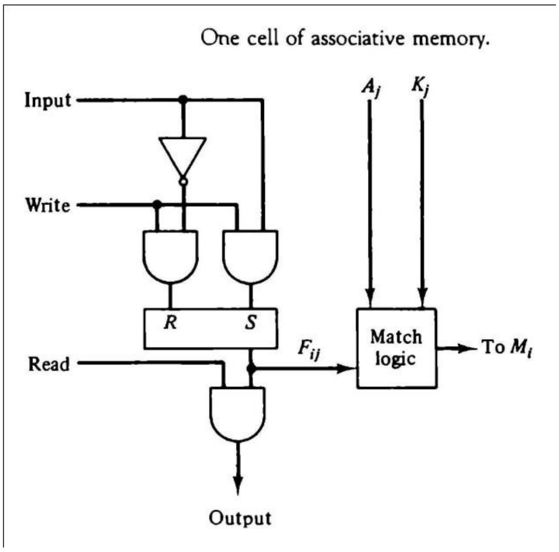
  K 111 000000

  Word 1 100 111100 no match

  Word 2 101 000001 match
- ✓ The relation between the memory array and external registers in an associative memory is shown in Fig.
- ✓ The cells in the array are marked by the letter C with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word. Thus cell Cij is the cell for bit j in word i.
- ✓ A bit Aj in the argument register is compared with all the bits in column j of the array provided that Kj = 1. This is done for all columns j = 1, 2,…,n.


Associative memory of m word, n cells per word.

- ✓ If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit Mi in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, Mi is cleared to 0
- ✓ The internal organization of a typical cell Cij is shown in Fig.
- ✓ It consists of a flipflop storage element Fij and the circuits for reading, writing, and matching the cell.
- ✓ The input bit is transferred into the storage cell during a write operation. The bit stored is read out during a read operation.
- ✓ The match logic compares the content of the storage cell with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in Mi.


One cell of associative memory.

MATCH LOGIC

- The match logic for each word can be derived from the comparison algorithm for two binary numbers. First, we neglect the key bits and compare the argument in A with the bits stored in the cells of the words. Word i is equal to the argument in A if Aj = Fij for j = 1, 2,…, n. Two bits are equal if they are both 1 or both 0. The equality of two bits can be expressed logically by the Boolean function

$$xj = Aj\ Fij + A'jF'ij$$

where xj = 1 if the pair of bits in position j are equal; otherwise, xj = 0.

- For a word i to be equal to the argument in A we must have all xj variables equal to 1.
- This is the condition for setting the corresponding match bit Mi to 1. The Boolean function for this condition is

$$Mi = x1\ x2\ x3 \ldots xn$$

- Include the key bit Kj in the comparison logic. The requirement is that if Kj = 0,the corresponding bits of Aj and Fij need no comparison. Only when Kj = 1 must they be compared. This requirement is achieved by ORing each term with K’j , thus:

$$xj + K'j = \begin{cases} xj & \text{if } Kj=1 \\ 1 & \text{if } Kj=0 \end{cases}$$

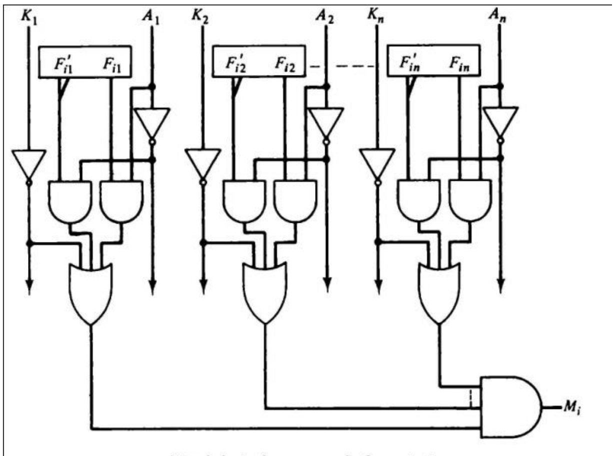
- When Kj = 1, we have Kj’ = 0 and xj + 0 = xj. When Kj = 0, then Kj’ = 1 xj + 1 = 1. A term (xj +Kj’) will be in the 1 state if its pair of bitsis not compared. This is necessary because each term is ANDed with all other terms so that an output of 1 will have no effect. The comparison of the bits has an effect only when Kj = 1.
- The match logic for word i in an associative memory can now be expressed by the following Boolean function:

$$Mi = (x1 + K'1)\ (x2 + K'2)\ (x3 + K'3) \ldots (xn + K'n)$$

- Each term in the expression will be equal to 1 if its corresponding Kj = 0. If Kj = 1, the term will be either 0 or 1 depending on the value of xj. A match will occur and Mi will be equal to 1 if all terms are equal to 1.
- If we substitute the original definition of xj. the Boolean function above can be expressed as follows:

$$Mi = \prod_{j=1}^{n} (Aj\ Fij + A'jF'ij + K'j)$$

- The circuit for matching one word is shown in Fig. Each cell requires two AND gates and one OR gate. The inverters for Aj and Kj are needed once for each column and are used for all bits in the column. The output of all OR gates in the cells of the same word go to the input of a common AND gate to generate the match signal for Mi. Mi will be logic 1 if a match occurs and 0 if no match occurs. Note that if the key register contains all 0’s, output Mi will be a 1 irrespective of the value of A or the word. This occurrence must be avoided during normal operation.



Match logic for one word of associative memory.

READ OPERATION

✓ If more than one word in memory matches the unmasked argument field, all the matched words will have 1's in the corresponding bit position of the match register. It is then necessary to scan the bits of the match register one at a time. The matched words are read in sequence by applying a read signal to each word line whose corresponding Mi bit is a 1.

✓ In most applications, the associative memory stores a table with no two identical items under a given key. In this case, only one word may match the unmasked argument field. By connecting output Mi directly to the read line in the same word position (instead of the M register), the content of the matched word will be presented automatically at the output lines and no specialread command signal is needed.

WRITE OPERATION

✓ An associative memory must have a write capability for storing the information to be searched.

✓ Writing in an associative memory can take different forms, depending on the application. If the entire memory is loaded with new information at once prior to a search operation then the writing can be done by addressing each location in sequence. This will make the device a random-access memory for writing and a content addressable memory for reading. The advantage here is that the address for input can be decoded as in a random-access memory. Thus instead of having m address lines, one for each word in memory, the number of address lines can be reduced by the decoder to d lines, where $m = 2^d$.

✓ If unwanted words have to be deleted and new words inserted one at a time, there is a need for a special register to distinguish between active and inactive words. This register, sometimes called a *tag register*.

✓ For every active word stored in memory, the corresponding bit in the tag register is set to 1. A word is deleted from memory by clearing its tag bit to 0.

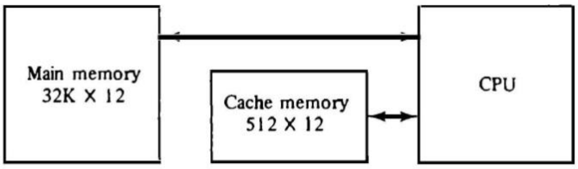
✓ Words are stored in memory by scanning the tag register until the first 0 bit is encountered. This gives the first available inactive word and a position for writing a new word. After the new word is stored in memory it is made active by setting its tag bit to 1. An unwanted word when deleted from memory can be cleared to all 0's if this value is used to specify an empty location.

CACHE MEMORY

- Locality of Reference: The references to memory at any given time interval tends to be confined within a localized area.
- When a program loop is executed, the CPU repeatedly refers to the set of instructions in memory that constitute the loop.
- Every time a given subroutine is called, its set of instructions is fetched from memory. Thus loops and subroutines tend to localize the references to memory for fetching instructions.
- Iterative procedures refer to common memory locations and array of numbers are confined within a local portion of memory
- If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a cache memory. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.
- When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory
- The performance of cache memory is frequently measured in terms of a quantity called **hit ratio**. When the CPU refers to memory and finds the word in cache, it is said to produce a hit. If the word is not found in cache, it is in main memory and it counts as a miss. *The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio.*
- The average memory access time of a computer system can be improved considerably by use of a cache.
- The transformation of data from main memory to cache memory is referred to as a mapping process. Three types of mapping procedures are :

2. Associative mapping
3. Direct mapping
4. Set-associative mapping.

- Consider the following memory organization:

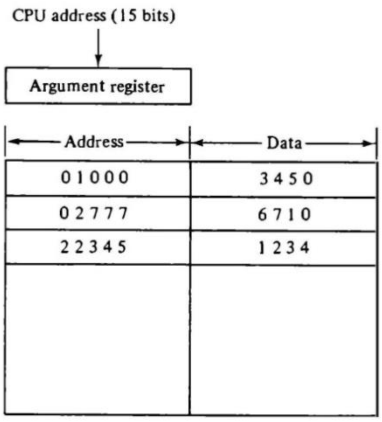


Example of cache memory.

ASSOCIATIVE MAPPING

- The faster and most flexible cache organization use an associative memory. The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory.
- A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found, the corresponding 12-bit data is read and sent to the CPU.
- If no match occurs, the main memory is accessed for the word. The address-data pair is then transferred to the associative cache memory. If the cache is full, an address–data pair must be displaced to make room for a pair that is needed and not presently in the cache.

Associative mapping cache (all numbers in octal).



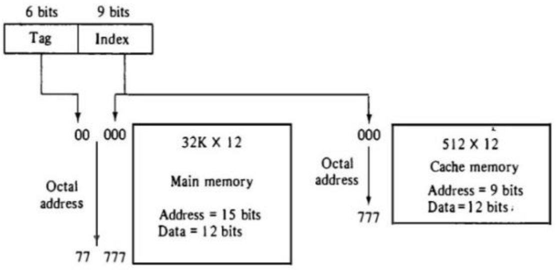
Address	Data
01000	3450
02777	6710
22345	1234

- The decision as to what pair is replaced is determined from the replacement algorithm that the designer chooses for the cache. A simple procedure is to replace cells of the cache in round-robin order whenever a new word is requested from main memory. This constitutes a first-in first-out (FIFO) replacement policy.

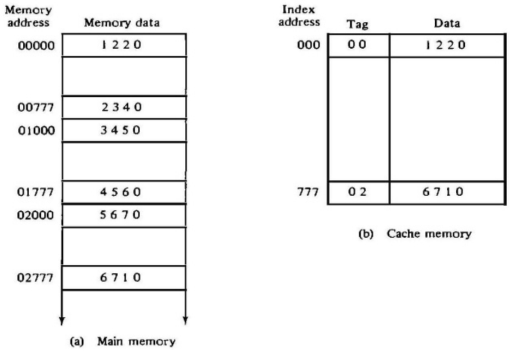
DIRECT MAPPING

- Associative memories are expensive compared to random-access memories because of the added logic associated with each cell.
- Direct mapping uses RAM instead of CAM.
- The n-bit memory address is divided into two fields: k bits for the index field and n-k bits for the tag field. The direct mapping cache organization uses the n-bit address to access the main memory and the k-bit index to access the cache.

Addressing relationships between main and cache memories.



- The internal organization of the words in the cache memory is as shown in Fig
- Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the index field is used for the address to access the cache.



Memory address	Memory data
00000	1220
00777	2340
01000	3450
01777	4560
02000	5670
02777	6710

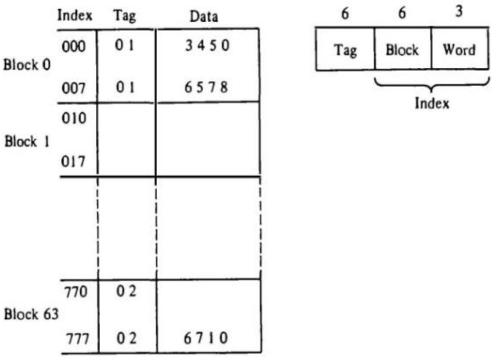
(a) Main memory

Index address	Tag	Data
000	00	1220
777	02	6710

(b) Cache memory

Direct mapping cache organization.

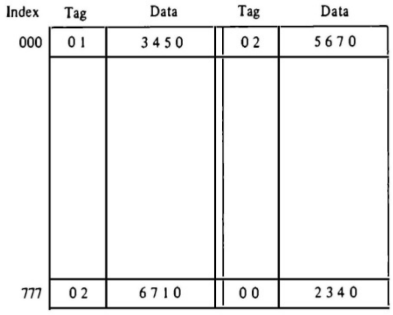
- The tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value.
- The disadvantage of direct mapping is that the hit ratio can drop considerably if two or more words whose addresses have the same index but different tags are accessed repeatedly.
- Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is sued to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 5670.
- The direct-mapping uses a block size of one word. The same organization but using a block size of 8 words is shown in Fig.
- The index field is now divided into two parts: the block field and the word field. The tag field stored within the cache is common to all eight words of the same block.
- Every time a miss occurs, an entire block of eight words must be transferred from main memory to cache memory. Although this takes extra time, the hit ratio will most likely improve with a larger block size because of the sequential nature of computer programs.



Direct mapping cache with block size of 8 words.

SET-ASSOCIATIVE MAPPING

- Set-associative mapping is an improvement over the direct-mapping organization in that each word of cache can store two or more words of memory under the same index address.
- Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form a set.
- Each index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the word length is 2(6 + 12) = 36 bits. An index address of nine bits can accommodate 512 words. Thus the size of cache memory is 512 × 36. It can accommodate 1024



Two-way set-associative mapping cache.

- The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777.

- When the CPU generates a memory request, the index value of the address is used to access the cache. The tag field of the CPU address is then compared with both tags in the cache to determine if a match occurs.
- The hit ratio will improve as the set size increases because more words with the same index but different tags can reside in cache.
- When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value. The most common replacement algorithms used are: random replacement, first-in first out (FIFO), and least recently used (LRU).

WRITING INTO CACHE

- An important aspect of cache organization is concerned with memory write requests. If the operation is a write, there are two ways that the system can proceed.
- The simplest and most commonly used procedure is to up data main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address. This is called the ***write-through method***. This method has the advantage that main memory always contains the same data as the cache,. This characteristic is important in systems with direct memory access transfers.
- The second procedure is called the ***write-back method***. In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the words are removed from the cache it is copied into main memory. The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times; however, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache. It is only when the word is displaced from the cache that an accurate copy need be rewritten into main memory.

CACHE INITIALIZATION

- The cache is initialized when power is applied to the computer or when the main memory is loaded with a complete set of programs from auxiliary memory. After initialization the cache is considered to be empty, built in effect it contains some non-valid data. It is customary to include with each word in cache a valid bit to indicate whether or not the word contains valid data.
- The cache is initialized by clearing all the valid bits to 0. The valid bit of a particular cache word is set to 1 the first time this word is loaded from main memory and stays set unless the cache has to be initialized again. The introduction of the valid bit means that a word in cache is initialization condition has the effect of forcing misses from the cache until it fills with valid data.

Pipelining and Vector Processing

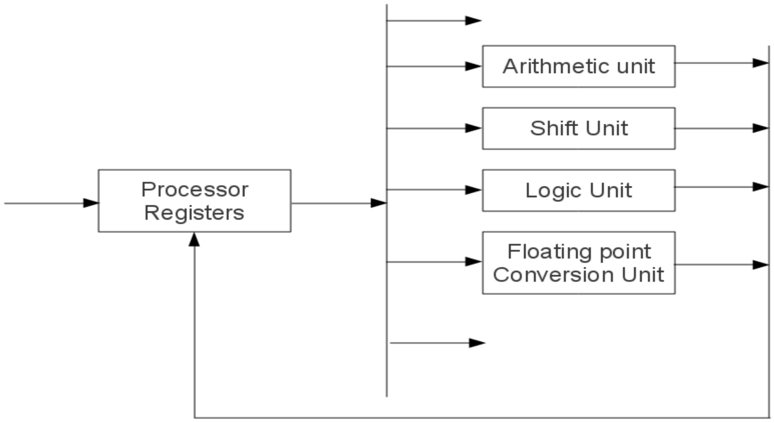
Parallel Processing:

The term parallel processing indicates that the system is able to perform several operations in a single time. Now we will elaborate the scenario, in a CPU we will be having only one Accumulator which will be storing the results obtained from the current operation. Now if we are giving only one command such that "a+b" then the CPU performs the operation and stores the result in the accumulator. Now we are talking about parallel processing, therefore we will be issuing two instructions "a+b" and "c-d" in the same time, now if the result of "a+b" operation is stored in the accumulator, then "c-d" result cannot be stored in the accumulator in the same time. Therefore the term parallel processing in not only based on the Arithmetic, logic or shift operations. The above problem can be solved in the following manner. Consider the registers R1 and R2 which will be storing the operands before operation and R3 is the register which will be storing the results after the operations. Now the above two instructions "a+b" and "c-d" will be done in parallel as follows.

- Values of "a" and "b" are fetched in to the registers R1 and R2
- The values of R1 and R2 will be sent into the ALU unit to perform the addition
- The result will be stored in the Accumulator
- When the ALU unit is performing the calculation, the next data "c" and "d" are brought into R1 and R2.
- Finally the value of Accumulator obtained from "a+b" will be transferred into the R3
- Next the values of C and D from R1 and R2 will be brought into the ALU to perform the "c-d" operation.
- Since the accumulator value of the previous operation is present in R3, the result of "c-d" can be safely stored in the Accumulator.

This is the process of parallel processing of only one CPU. Consider several such CPU performing the calculations separately. This is the concept of parallel processing.

Concept of Parallel Processing



In the above figure we can see that the data stored in the processor registers is being sent to separate devices basing on the operation needed on the data. If the data inside the processor registers is requesting for an arithmetic operation, then the data will be sent to the arithmetic unit and if in the same time another data is requested in the logic unit, then the data will be sent to logic unit for logical operations. Now in the same time both arithmetic operations and logical operations are executing in parallel. This is called as parallel processing.

Instruction Stream: The sequence of instructions read from the memory is called as an Instruction Stream

Data Stream: The operations performed on the data in the processor is called as a Data Stream.

The computers are classified into 4 types based on the Instruction Stream and Data Stream. They are called as the Flynn's Classification of computers.

Flynn's Classification of Computers:

- Single Instruction Stream and Single Data Stream (SISD)
- Single Instruction Stream and Multiple Data Stream (SIMD)
- Multiple Instruction Stream and Single Data Stream (MISD)
- Multiple Instruction Stream and Multiple Data Stream (MIMD)

SISD represents the organization of a single computer containing a control unit, a processor unit and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.

MISD structure is only of theoretical interest since no practical system has been constructed using this organization because Multiple instruction streams means more no of instructions, therefore we have to perform multiple instructions on same data at a time. This is practically impossible.

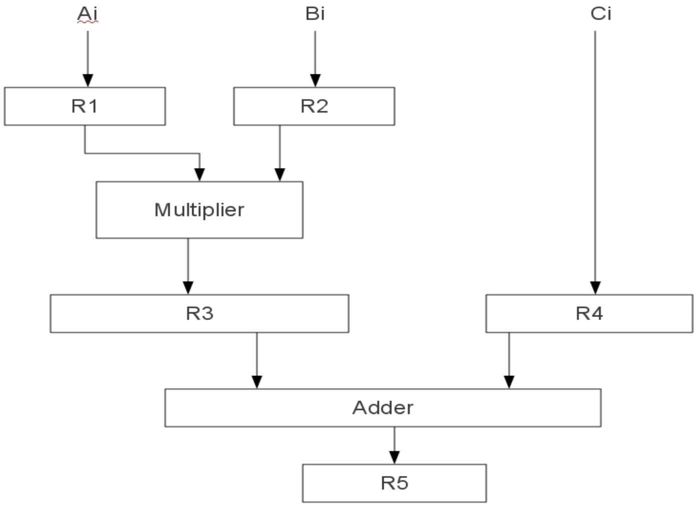
MIMD structure refers to a computer system capable of processing several programs at the same time operating on different data.

Pipelining: Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments. We can consider the pipelining concept as a collection of several segments of data processing programs which will be processing the data and sending the results to the next segment until the end of the processing is reached. We can visualize the concept of pipelining in the example below.
Consider the following operation: Result=(A+B)*C

- First the A and B values are Fetched which is nothing but a “Fetch Operation”.
- The result of the Fetch operations is given as input to the Addition operation, which is an Arithmetic operation.
- The result of the Arithmetic operation is again given to the Data operand C which is fetched from the memory and using another arithmetic operation which is Multiplication in this scenario is executed.
- Finally the Result is again stored in the “Result” variable.

In this process we are using up-to 5 pipelines which are the
→ Fetch Operation (A)| Fetch Operation(B) | Addition of (A & B) | Fetch Operation(C) | Multiplication of ((A+B), C) | Load ((A+B)*C), Result);

Pipelining



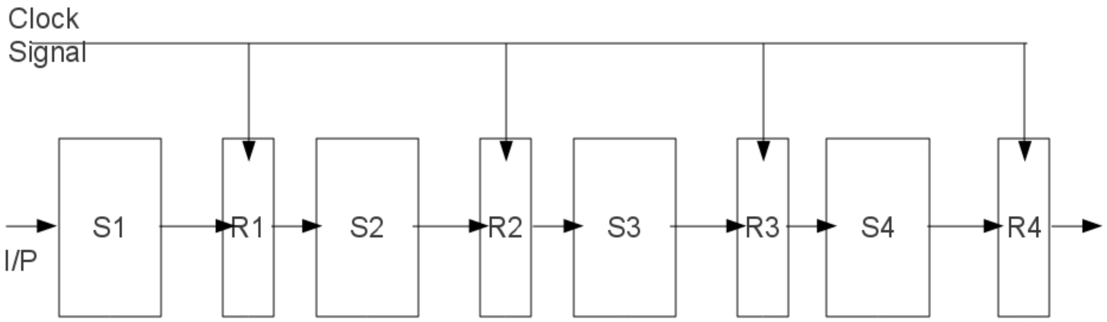
The contents of the Registers in the above pipeline concept are given below. We are considering the implementation of A[7] array with B[7] array.

Clock Pulse Number	Segment1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1	-	-	-
2	A2	B2	A1*B1	C1	-
3	A3	B3	A2*B2	C2	A1*B1+C1
4	A4	B4	A3*B3	C3	A2*B2+C2
5	A5	B5	A4*B4	C4	A3*B3+C3
6	A6	B6	A5*B5	C5	A4*B4+C4
7	A7	B7	A6*B6	C6	A5*B5+C5
8			A7*B7	C7	A6*B6+C6
9					A7*B7+C7

If the above concept is executed with out the pipelining, then each data operation will be taking 5 cycles, totally they are 35 cycles of CPU are needed to perform the operation. But if are using the concept of pipeline, we will be cutting off many cycles. Like given in the table below when the values of A1 and B1 are coming into the registers R1 and R2, the registers R3, R4 and R5 are empty. Now in the second cycle the multiplication of A1 and B1 is transferred to register R3, now in this point the contents of the register R1 and R2 are empty. Therefore the next two values A2 and B2 can be brought into the registers. Again in the third cycle after fetching the C1 value the operation (A1*B1)+C1 will be performed. So in this way we can achieve the total concept in only 9 cycles. Here we are assuming that the clock cycle timing is fixed. This is the concept of pipelining.

Below is the diagram of 4 segment pipeline.

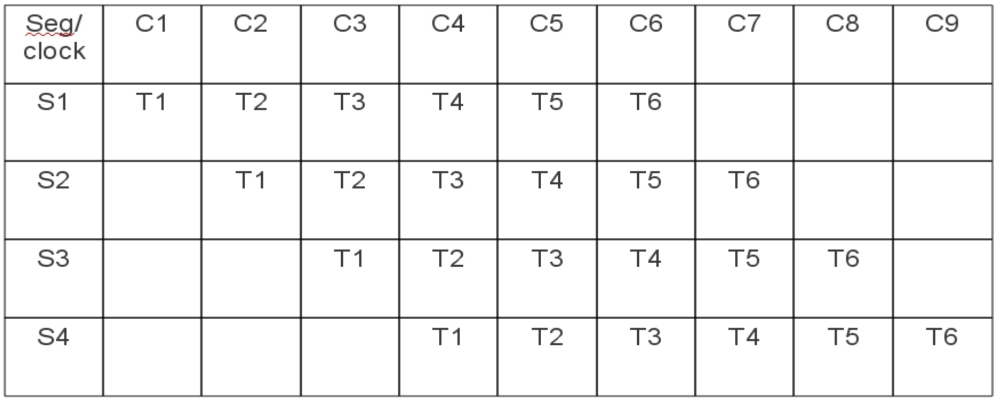
Segment Representation



4 Segment pipeline

The below table is the space time diagram for the execution of 6 tasks in the 4 segment pipeline.

Space and Time Diagram

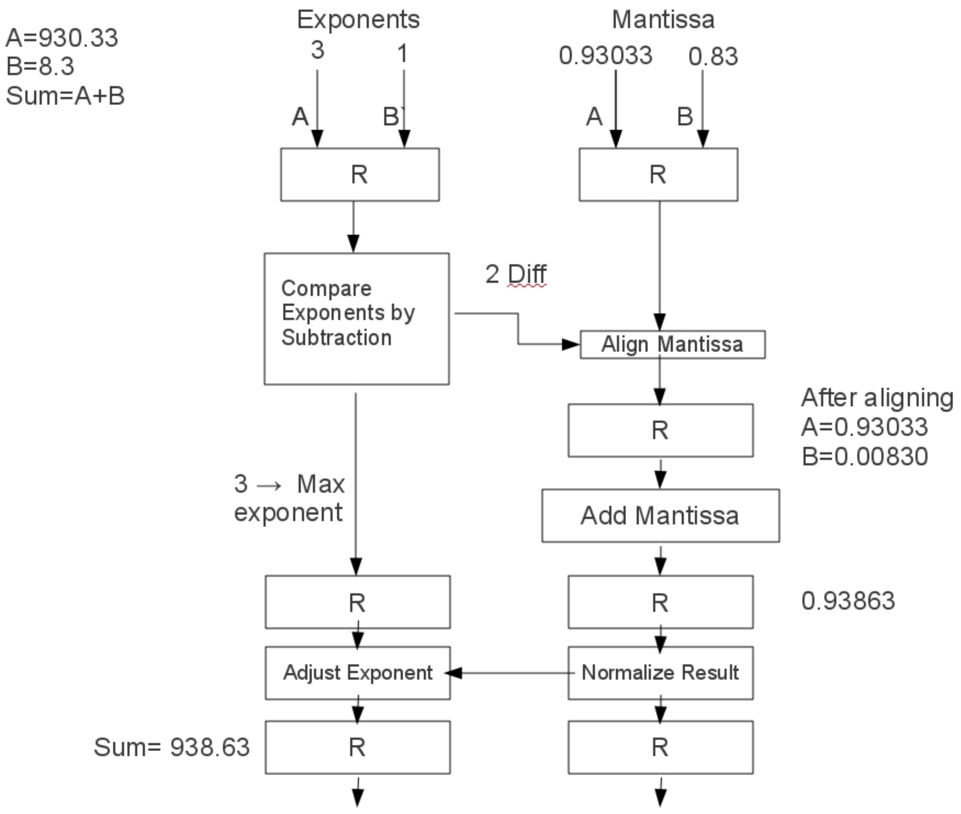


Seg/ clock	C1	C2	C3	C4	C5	C6	C7	C8	C9
S1	T1	T2	T3	T4	T5	T6			
S2		T1	T2	T3	T4	T5	T6		
S3			T1	T2	T3	T4	T5	T6	
S4				T1	T2	T3	T4	T5	T6

S= nTn/(K+n-1)*tp

<u>Arithmetic pipeline:</u>

ARITHMETIC PIPELINING



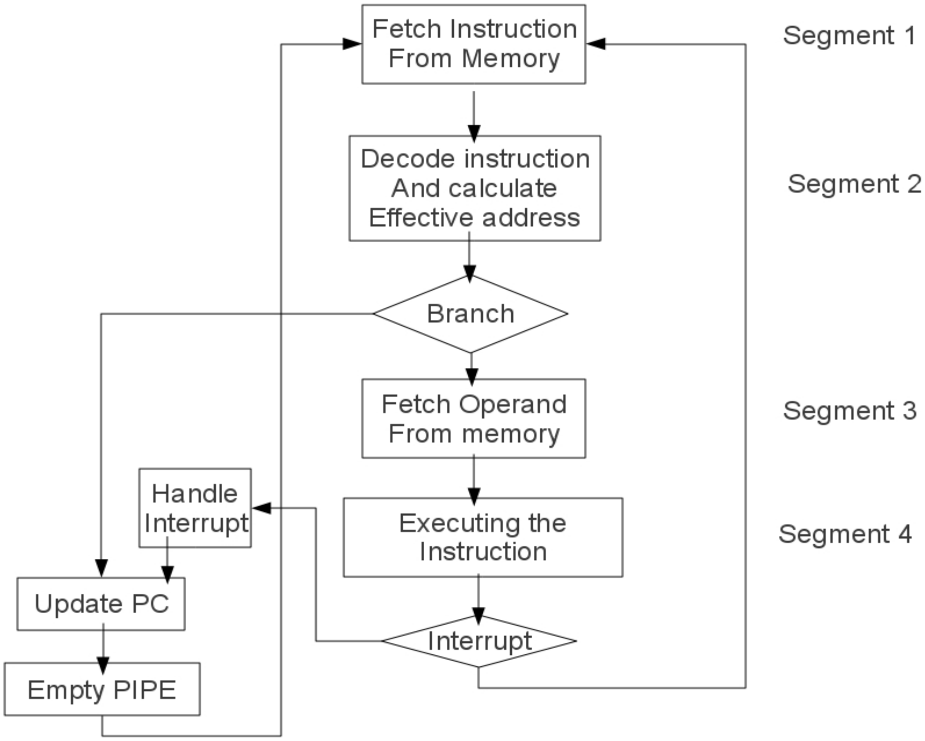
The above diagram represents the implementation of arithmetic pipeline in the area of floating point arithmetic operations. In the diagram, we can see that two numbers A and B are added together. Now the values of A and B are not normalized, therefore we must normalize them before start to do any operations. The first thing is we have to fetch the values of A and B into the registers. Here R denote a set of registers. After that the values of A and B are normalized, therefore the values of the exponents will be compared in the comparator. After that the alignment of mantissa will be taking place. Finally, we will be performing addition, since an addition is happening in the adder circuit. The source registers will be free and the second set of values can be brought. Like wise when the normalizing of the result is taking place, addition of the new values will be added in the adder circuit and when addition is going on, the new data values will be brought into the registers in the start of the implementation. We can see how the addition is being performed in the diagram.

Instruction Pipeline: Pipelining concept is not only limited to the data stream, but can also be applied on the instruction stream. The instruction pipeline execution will be like the queue execution. In the queue the data that is entered first, will be the data first retrieved. Therefore when an instruction is first coming, the instruction will be placed in the queue and will be executed in the system. Finally the results will be passing on to the next instruction in the queue. This scenario is called as Instruction pipelining. The instruction cycle is given below

- Fetch the instruction from the memory
- Decode the instruction
- calculate the effective address
- Fetch the operands from the memory
- Execute the instruction
- Store the result in the proper place.

In a computer system each and every instruction need not necessary to execute all the above phases. In a Register addressing mode, there is no need of the effective address calculation. Below is the example of the four segment instruction pipeline.

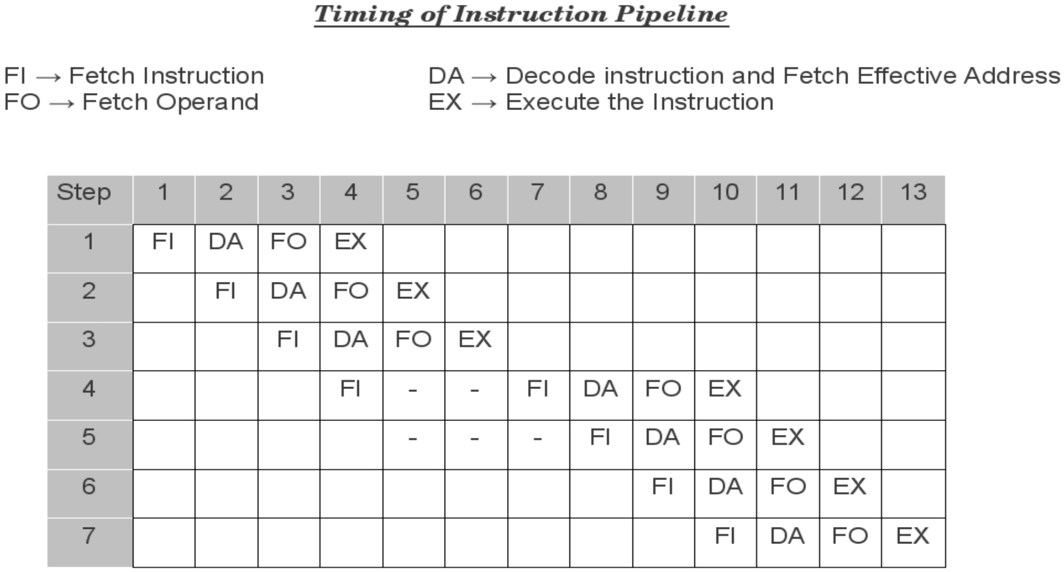
Instruction pipelining


Fetch Instruction From Memory
Segment 1
Decode instruction And calculate Effective address
Segment 2
Branch
Fetch Operand From memory
Segment 3
Handle Interrupt
Executing the Instruction
Segment 4
Update PC
Interrupt
Empty PIPE

In the above diagram we can see that the instruction which is first executing has to be fetched from the memory, there after we are decoding the instruction and we are calculating the effective address. Now we have two ways to execute the instruction. Suppose we are using a normal instruction like ADD, then the operands for that instruction will be fetched and the instruction will be executed. Suppose we are executing an instruction such as Fetch command. The fetch command itself has internally three more commands which are like ACTDR, ARTDR etc.., therefore we have to jump to that particular location to execute the command, so we are using the branch operation. So in a branch operation, again other instructions will be executed. That means we will be updating the PC value such that the instruction can be executed. Suppose we are fetching the operands to perform the original operation such as ADD, we need to fetch the data. The data can be fetched in two ways, either from the main memory or else from an input output devices. Therefore in order to use the input output devices, the devices must generate the interrupts which should be handled by the CPU. Therefore the handling of interrupts is also a kind of program execution. Therefore we again have to start from the starting of the program and execute the interrupt cycle.

The different instruction cycles are given below:

- FI → FI is a segment that fetches an instruction
- DA → DA is a segment that decodes the instruction and identifies the effective address.
- FO → FO is a segment that fetches the operand.
- EX → EX is a segment that executes the instruction with the operand.



Timing of Instruction Pipeline

FI → Fetch Instruction
FO → Fetch Operand
DA → Decode instruction and Fetch Effective Address
EX → Execute the Instruction

Step	1	2	3	4	5	6	7	8	9	10	11	12	13
1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
3			FI	DA	FO	EX							
4				FI	-	-	FI	DA	FO	EX			
5					-	-	-	FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

Pipelining Conflicts: There are different conflicts that are caused by using the pipeline concept. They are

- Resource Conflicts: These are caused by access to memory by two or more segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories
- Data Dependency: These conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
- Branch difficulties: These difficulties arise from branch and other instructions that change the value of PC.

Data Dependency Conflict: The data dependency conflict can be solved by using the following methods.

- Hardware Interlocks: The most straight forward method is to insert hardware interlocks. An interlock is a circuit that detects instructions whose source operands are destination of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict. This approach maintains the program sequence by using hardware to insert the required delay.
- Operand Forwarding: Another technique called operand forwarding uses special hardware to detect a conflict and avoid the conflict path by using a special path to forward the values between the pipeline segments.
- Delayed Load: The delayed load operation is nothing but when executing an instruction in the pipeline, simply delay the execution starting of the instruction such that all the data that is needed for the instruction can be successfully updated before execution.

Branch Conflicts:

The following are the solutions for solving the branch conflicts that are obtained in the pipelining concept.

- Pre-fetch Target Instruction: In this the branch instructions which are to be executed are pre-fetched to detect if any errors are present in the branch before execution.
- Branch Target Buffer: BTB is the associative memory implementation of the branch conditions.
- Loop buffer: The loop buffer is a very high speed memory device. Whenever a loop is to be executed in the computer. The complete loop will be transferred in to the loop buffer memory and will be executed as in the cache memory.

- Branch Prediction: The use of branch prediction is such that, before a branch is to be executed, the instructions along with the error checking conditions are checked. Therefore we will not be going into any unnecessary branch loops.
- Delayed Branch: The delayed branch concept is same as the delayed load process in which we are delaying the execution of a branch process, before all the data is fetched by the system for beginning the CPU.

RISC Pipeline:

The ability to use the instruction pipelining concept in the RISC architecture is very efficient. The simplicity of the instruction set can be utilized to implement an instruction pipeline using a small number of sub operations, with each being executed in one clock cycle. Due to fixed length instruction format, the decoding of the operation can occur at the same time as the register selection. Since the arithmetic, logic and shift operations are done on register basis, there is no need for extra fetching or effective address decoding steps to perform the operation. So pipelining concept can be effectively used in this scenario. Therefore the total operations can be categorized as one segment will be fetching the instruction from program memory, the other segment executes the instruction in the ALU and the third segment may be used to store the result of the ALU operation in a destination register. The data transfer instructions in RISC are limited to only Load and Store instructions. To prevent conflicts in data transfer, we will be using two separate buses one for storing the instructions and other for storing the data.

Example of three segment instruction pipeline:
We want to perform a operation in which there is some arithmetic, logic or shift operations. Therefore as per the instruction cycle, we will be having the following steps:

- I: Instruction Fetch
- A: ALU Operation
- E: Execute Instruction.

The I segment will be fetching the instruction from program memory. The instruction is decoded and an ALU operation is performed in the A segment. In the A segment the ALU operation instruction will be fetched and the effective address will be retrieved and finally in the E segment the instruction will be executed.

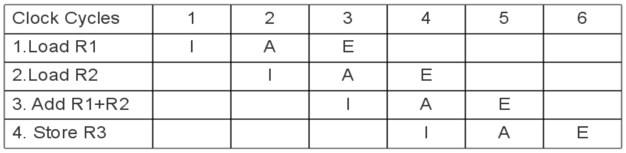
Delayed Load:

Consider the following instructions:

1. LOAD: R1 ← M[address 1]
2. LOAD: R2 ← M[address 2]
3. ADD: R3 ← R1 + R2
4. STORE: M[address 3] ← R3

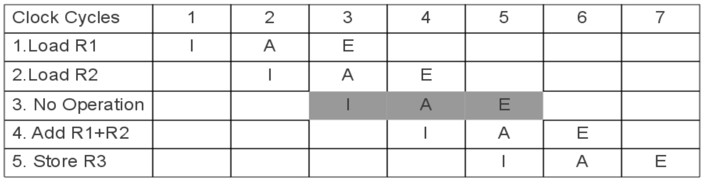
The below tables will be showing the pipelining concept with the data conflict and without data conflict.

Pipeline timing with data conflict



Clock Cycles	1	2	3	4	5	6
1.Load R1	I	A	E			
2.Load R2		I	A	E		
3. Add R1+R2			I	A	E	
4. Store R3				I	A	E

Pipeline timing with delayed load



Clock Cycles	1	2	3	4	5	6	7
1.Load R1	I	A	E				
2.Load R2		I	A	E			
3. No Operation			I	A	E		
4. Add R1+R2				I	A	E	
5. Store R3					I	A	E

Vector Processing:

Normal computational systems are not enough in some special processing requirements. Such as, in special processing systems like artificial intelligence systems and some weather forecasting systems, terrain analysis, the normal systems are not sufficient. In such systems the data processing will be involving on very high amount of data, we can classify the large data as a very big arrays. Now if we want to process this data, naturally we will need new methods of data processing. The vectors are considered as the large one dimensional array of data. The term vector processing involves the data processing on the vectors of such large data.

The vector processing system can be understand by the example below.

Consider a program which is adding two arrays A and B of length 100;

Machine level program

```
        Initialize I=0
20      Read A(I)
        Read B(I)
        Store C(I)=A(I)+B(I)
        Increment I=I+1
        If I<=100 go to 20
        continue
```

so in this above program we can see that the two arrays are being added in a loop format. First we are starting from the value of 0 and then we are continuing the loop with the addition operation until the I value has reached to 100. In the above program there are 5 loop statements which will be executing 100 times. Therefore the total cycles of the CPU taken is 500 cycles. But if we use the concept of vector processing then we can reduce the unnecessary fetch cycles, since the fetch cycles are used in the creation of the vector. The same program written in the vector processing statement is given below.

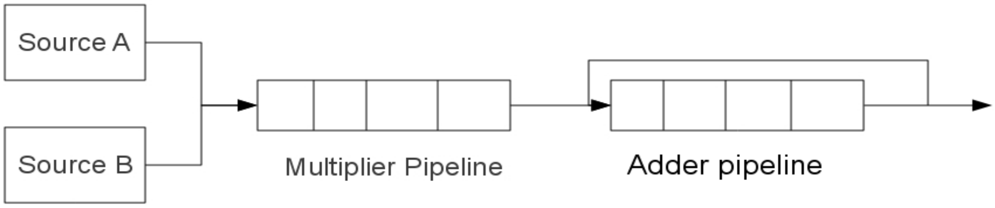
C(1:100)=A(1:100)+B(1:100)

In the above statement, when the system is creating a vector like this the original source values are fetched from the memory into the vector, therefore the data is readily available in the vector. So when a operation is initiated on the data, naturally the operation will be performed directly on the data and will not wait for the fetch cycle. So the total no of CPU Cycles taken by the above instruction is only 100.

Operation Code	Base Address SRC 1	Base Address SRC 2	Base Address DST	Vector length

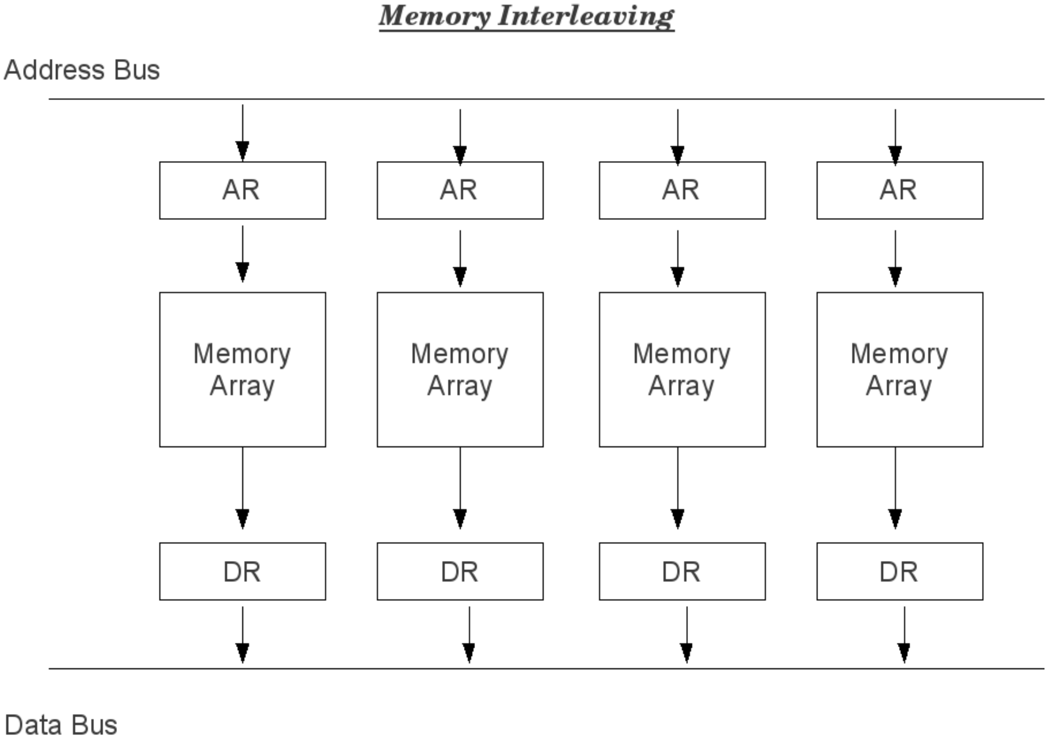
Instruction format of Vector Instruction

Below we can see the implementation of the vector processing concept on the following matrix multiplication. In the matrix multiplication, we will be multiplying the row of A matrix with the column of the B matrix elements individually finally we will be adding the results.



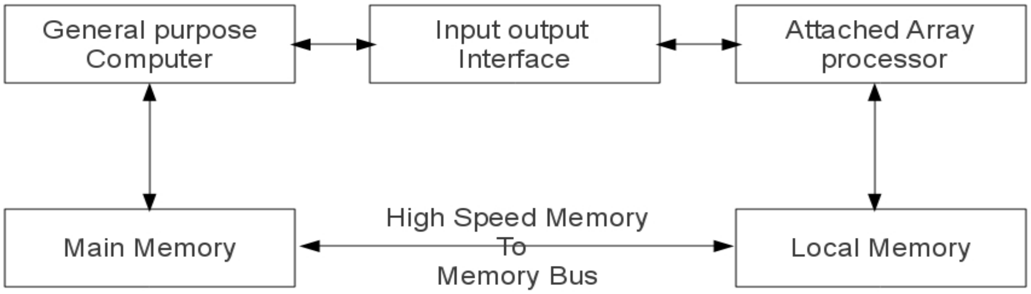
In the above diagram we can see that how the values of A vector and B Vector which represents the matrix are being multiplied. Here we will be considering a 4x4 matrix A and B. Now the from the source A vector we will be taking the first 4 values and will be sending to the multiplier pipeline along with the 4 values from the vector B. The resultant 1 value is stored in the adder pipeline. Like wise remaining values from a row and column multiplication will be brought into the adder pipeline, which will be performing the addition of all the things finally we will have the result of one row to column multiplication. When addition operation is taking place in the adder pipeline the next set of values will be brought into the multiplier pipeline, so that all the operations can be performed simultaneously using the parallel processing concepts by the implementation of pipeline.

Memory Interleaving:



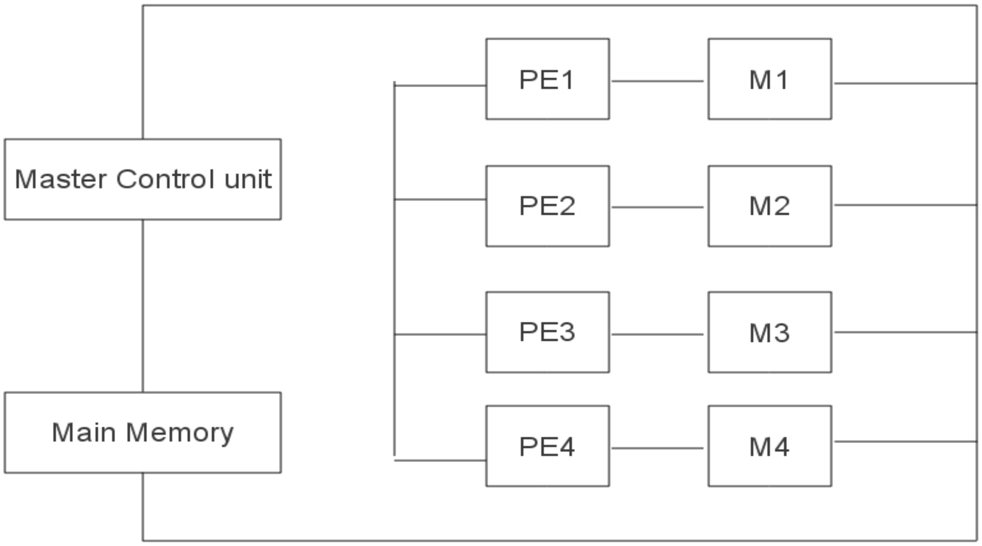
Pipelining and vector processing naturally requires the several data elements for processing. So instead of using the same memory and selecting one at a time, we will be using several modules of the memory such that we can have separate data for each processing unit. As we can see in the above in the diagram each memory array is designed independently of the next memory array. Such that when the data needed for a operation is stored in the first memory array, another data for another operation can be safely stored in the next memory array, so that the operations can be performed concurrently. This process is called as memory interleaving.

Array Processors: In a distributed computing we will be having several computers working on the same task such that their processing power will be shared among all the systems so that they can perform the task fast. But the disadvantage of the distributed computing is that we have to give separate resources for each system and every system need to be controlled by a task initiating system or can be called as a central control unit. The management of this kind of systems is very hard. In order to perform a specific operation involving a large processing there is no need of distributed computing. The alternate for this kind of scenarios is array processors or attached array processors. The simplest is the SIMD Attached array processor.



Attached Array processor

The above diagram shows that the system is attached a separate processor which will be used for operation specific purpose. If the array processor is designed for solving floating point arithmetic, then it will only perform that operations. The detailed figure of the attached array processor is given in the diagram below. This will be having the SIMD architecture. In this we will be having a master control unit which will be coordinating all the process in the array processor. Each processing unit in the array processor is having a local memory unit as in the memory interleaving concept on which it performs the operations. Finally we will be having a main memory in which the original source data and the results that are obtained from the array processor will be stored. This is the working principle of the SIMD array processor technology.



<u>SIMD Array Processor Technology</u>

UNIT V MULTIPROCESSORS

Characteristics of multiprocessors – Interconnection structures – Inter processor arbitration – Inter processor communication and synchronization – Cache coherence

5.1 Multiprocessor:

- A set of processors connected by a communications network

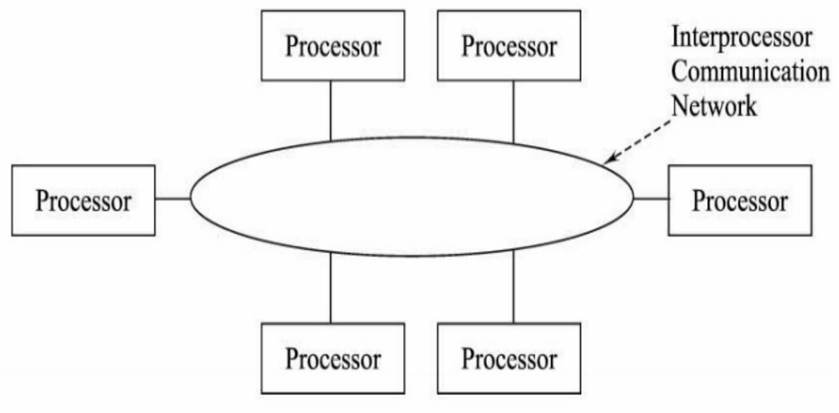


Fig. 5.1 Basic multiprocessor architecure

- A multiprocessor system is an interconnection of two or more CPU's with memory and input-output equipment.
- Multiprocessors system are classified as multiple instruction stream, multiple data stream systems(MIMD).
- There exists a distinction between multiprocessor and multicomputers that though both support concurrent operations.
- In multicomputers several autonomous computers are connected through a network and they may or may not communicate but in a multiprocessor system there is a single OS Control that provides interaction between processors and all the components of the system to cooperate in the solution of the problem.
- VLSI circuit technology has reduced the cost of the computers to such a low Level that the concept of applying multiple processors to meet system performance requirements has become an attractive design possibility.

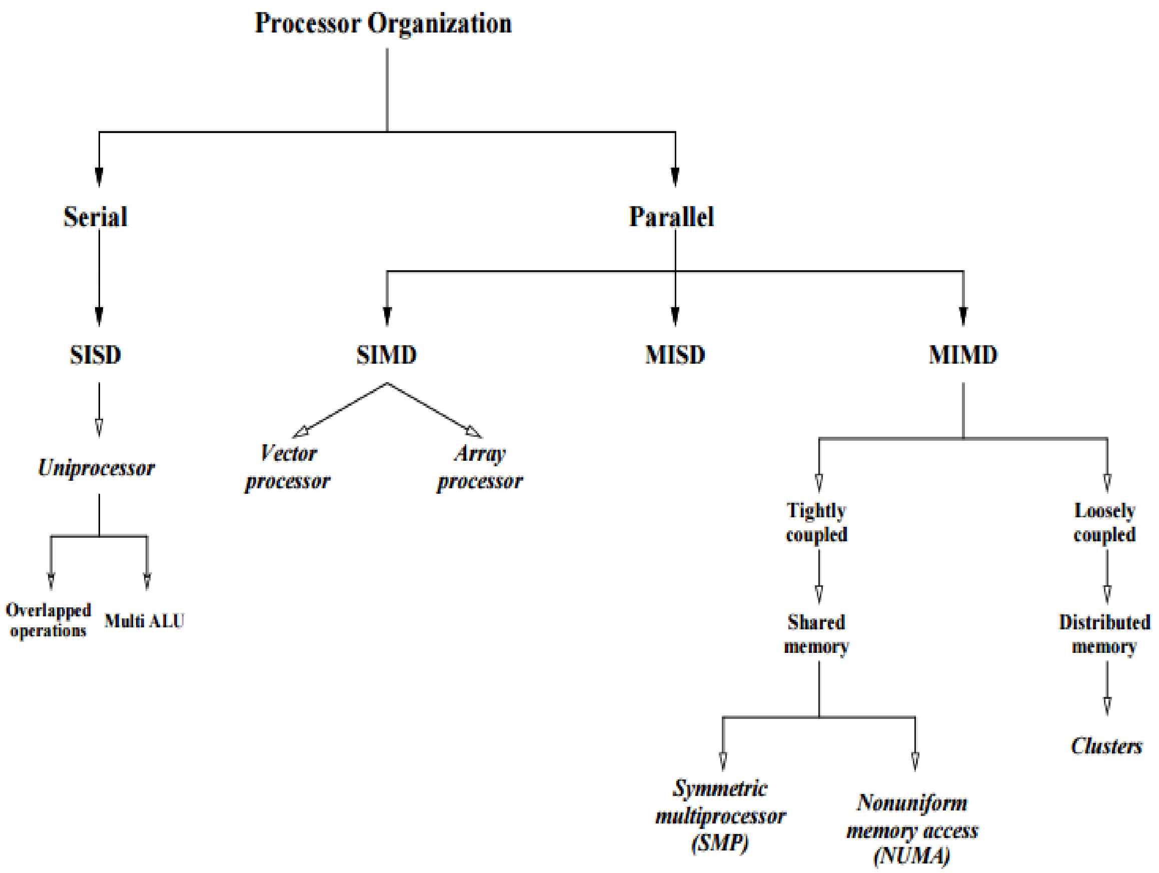


Fig. 5.2 Taxonomy of mono- mulitporcessor organizations

Characteristics of Multiprocessors:

Benefits of Multiprocessing:

1. Multiprocessing increases the reliability of the system so that a failure or error in one part has limited effect on the rest of the system. If a fault causes one processor to fail, a second processor can be assigned to perform the functions of the disabled one.

2. Improved System performance. System derives high performance from the fact that computations can proceed in parallel in one of the two ways:

a) Multiple independent jobs can be made to operate in parallel.

b) A single job can be partitioned into multiple parallel tasks.

This can be achieved in two ways:

- The user explicitly declares that the tasks of the program be executed in parallel

- The compiler provided with multiprocessor s/w that can automatically detect parallelism in program. Actually it checks for Data dependency

COUPLING OF PROCESSORS

Tightly Coupled System/Shared Memory:

- Tasks and/or processors communicate in a highly synchronized fashion
- Communicates through a common global shared memory
- Shared memory system. This doesn’t preclude each processor from having its own local memory(cache memory)

Loosely Coupled System/Distributed Memory

- Tasks or processors do not communicate in a synchronized fashion.
- Communicates by message passing packets consisting of an address, the data content, and some error detection code.
- Overhead for data exchange is high
- Distributed memory system

Loosely coupled systems are more efficient when the interaction between tasks is minimal, whereas tightly coupled system can tolerate a higher degree of interaction between tasks.

Shared (Global) Memory

- A Global Memory Space accessible by all processors
- Processors may also have some local memory

Distributed (Local, Message-Passing) Memory

- All memory units are associated with processors
- To retrieve information from another processor's memory a message must be sent there

Uniform Memory

- All processors take the same time to reach all memory locations

Non-uniform (NUMA) Memory

- Memory access is not uniform

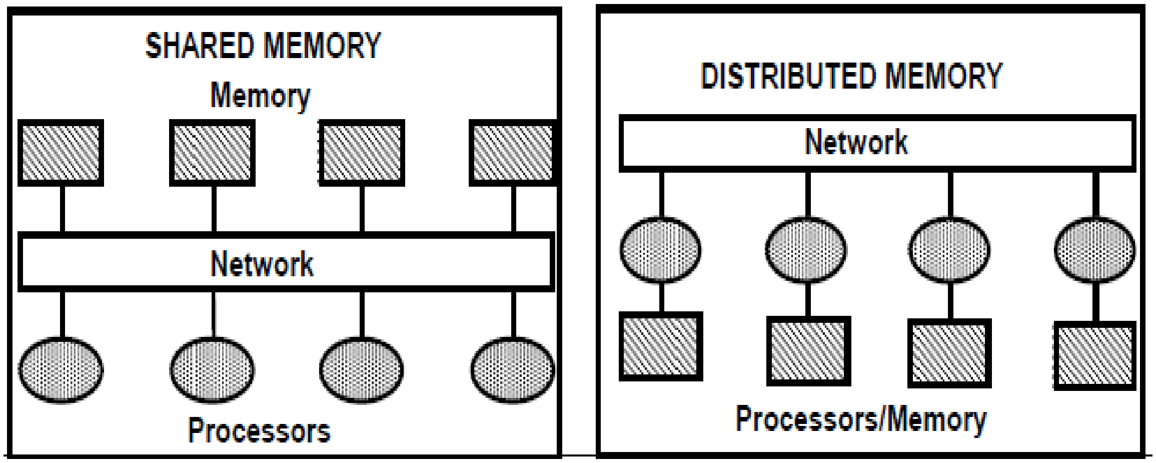


Fig. 5.3 Shared and distributed memory

Shared memory multiprocessor:

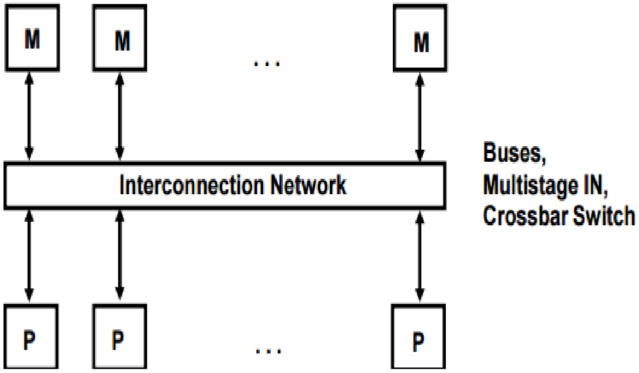


Fig 5.4 Shared memory multiprocessor

Characteristics

- All processors have equally direct access to one large memory address space

Limitations

- Memory access latency; Hot spot problem

5.2 Interconnection Structures:

The interconnection between the components of a multiprocessor System can have different physical configurations depending n the number of transfer paths that are available between the processors and memory in a shared memory system and among the processing elements in a loosely coupled system.

Some of the schemes are as:

- Time-Shared Common Bus
- Multiport Memory
- Crossbar Switch
- Multistage Switching Network
- Hypercube System

a. Time shared common Bus

- All processors (and memory) are connected to a common bus or busses
- Memory access is fairly uniform, but not very scalable
- A collection of signal lines that carry module-to-module communication
- Data highways connecting several digital system elements
- Operations of Bus

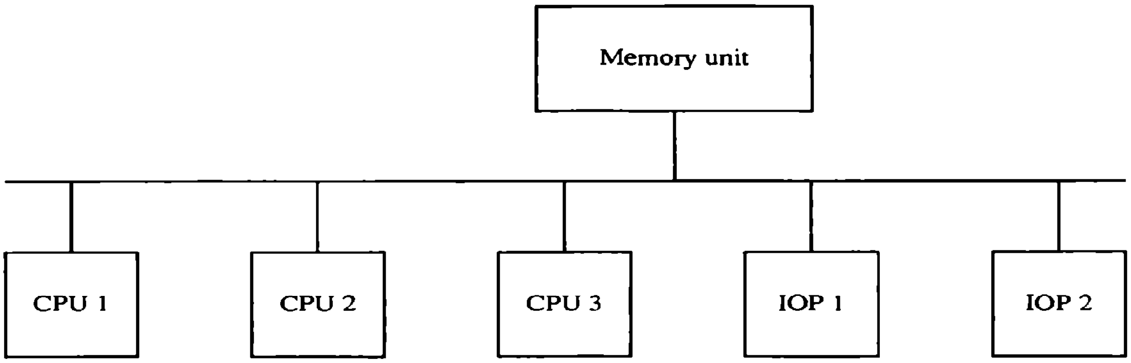


Fig. 5.5 Time shared common bus organization

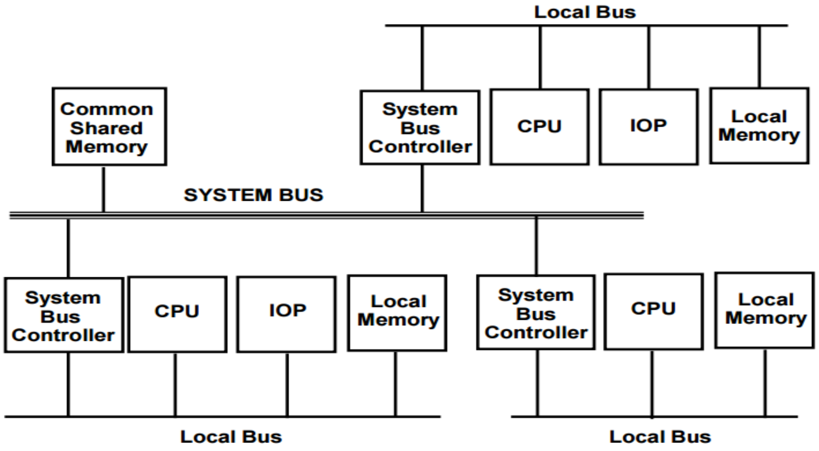


Fig. 5.6 system bus structure for multiprocessor

In the above figure we have number of local buses to its own local memory and to one or more processors. Each local bus may be connected to a CPU, an IOP, or any combinations of processors. A system bus controller links each local bus to a common system bus. The I/O devices connected to the local IOP, as well as the local memory, are available to the local processor. The memory connected to the common system bus is shared by all processors. If an IOP is connected directly to the system bus the I/O devices attached to it may be made available to all processors

Disadvantage.:

- Only one processor can communicate with the memory or another processor at any given time.
- As a consequence, the total overall transfer rate within the system is limited by the speed of the single path

b. Multiport Memory:

Multiport Memory Module

- Each port serves a CPU

Memory Module Control Logic

- Each memory module has control logic
- Resolve memory module conflicts Fixed priority among CPUs

Advantages

- The high transfer rate can be achieved because of the multiple paths.

Disadvantages:

- It requires expensive memory control logic and a large number of cables and connections

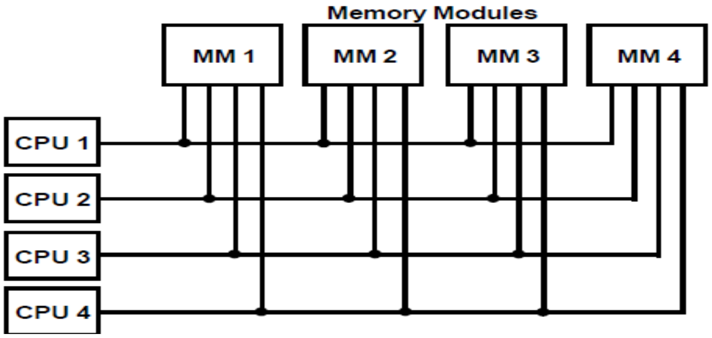
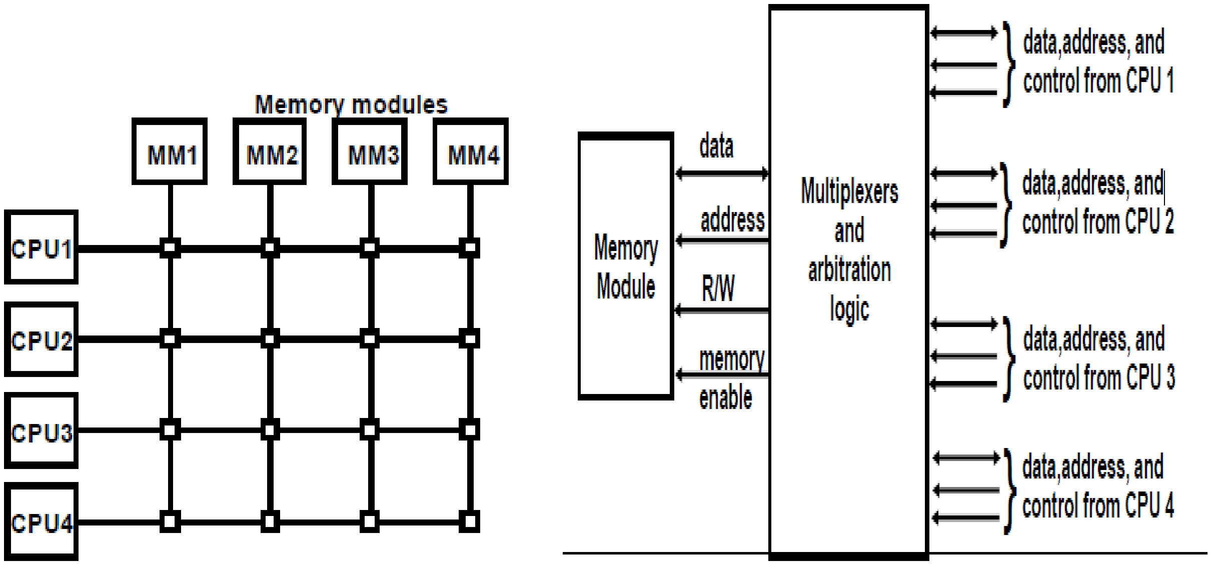


Fig. 5.7 Multiport memory

c. Crossbar switch:

- Each switch point has control logic to set up the transfer path between a processor and a memory.
- It also resolves the multiple requests for access to the same memory on the predetermined priority basis.
- Though this organization supports simultaneous transfers from all memory modules because there is a separate path associated with each Module.
- The H/w required to implement the switch can become quite large and complex


Memory modules
MM1
MM2
MM3
MM4
CPU1
CPU2
CPU3
CPU4
Memory Module
data
address
R/W
memory enable
Multiplexers and arbitration logic
data,address, and control from CPU 1
data,address, and control from CPU 2
data,address, and control from CPU 3
data,address, and control from CPU 4

a) b)

Fig. 5.8 a) cross bar switch b) Block diagram of cross bar switch

Advantage:

- Supports simultaneous transfers from all memory modules

Disadvantage:

- The hardware required to implement the switch can become quite large and complex.

d. Multistage Switching Network:

- The basic component of a multi stage switching network is a two-input, two-output interchange switch.

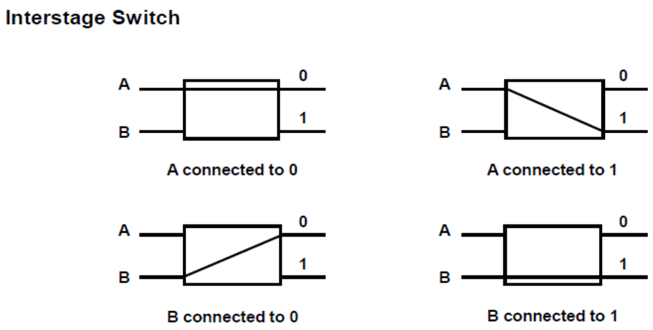
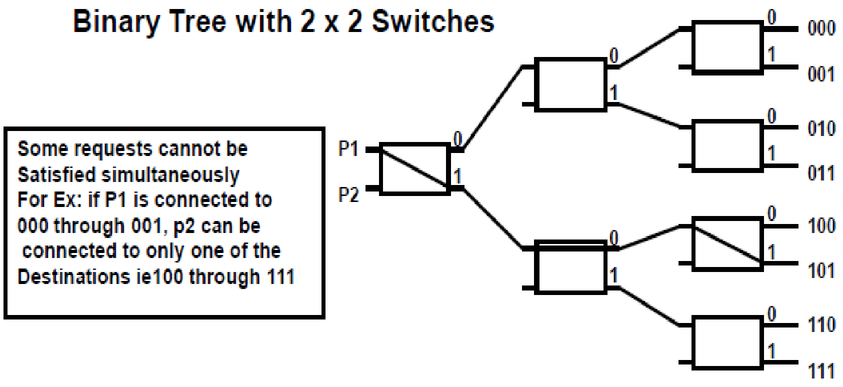


Fig. 5.9 operation of 2X2 interconnection switch

Using the 2x2 switch as a building block, it is possible to build a multistage network to control the communication between a number of sources and destinations.

- To see how this is done, consider the binary tree shown in Fig. below.
- Certain request patterns cannot be satisfied simultaneously.

i.e., if P1 → 000~011, then P2 → 100~111



Some requests cannot be Satisfied simultaneously For Ex: if P1 is connected to 000 through 001, p2 can be connected to only one of the Destinations ie100 through 111

Fig 5.10 Binary tree with 2x2 switches

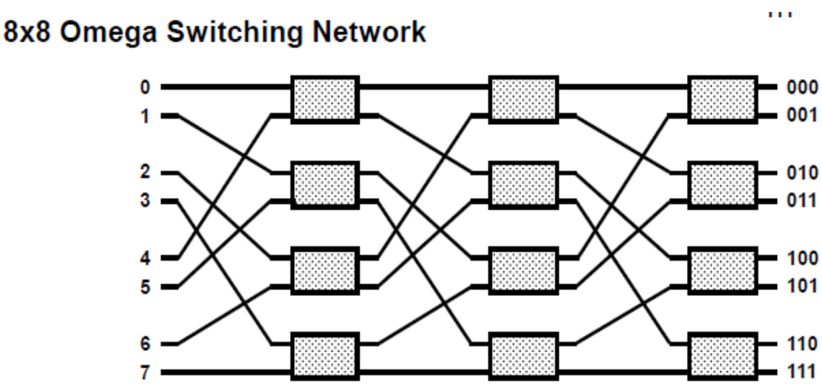


Fig. 5.11 8X8 Omega switching network

- Some request patterns cannot be connected simultaneously. i.e., any two sources cannot be connected simultaneously to destination 000 and 001
- In a tightly coupled multiprocessor system, the source is a processor and the destination is a memory module.
- Set up the path → transfer the address into memory → transfer the data
- In a loosely coupled multiprocessor system, both the source and destination are Processsing elements.

e. Hypercube System:

The hypercube or binary n-cube multiprocessor structure is a loosely coupled system composed of N=2n processors interconnected in an n-dimensional binary cube.

- Each processor forms a node of the cube, in effect it contains not only a CPU but also local memory and I/O interface.
- Each processor address differs from that of each of its n neighbors by exactly one bit position.
- Fig. below shows the hypercube structure for n=1, 2, and 3.
- Routing messages through an *n*-cube structure may take from one to *n* links from a source node to a destination node.
- A routing procedure can be developed by computing the exclusive-OR of the source node address with the destination node address.
- The message is then sent along any one of the axes that the resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ.
- A representative of the hypercube architecture is the Intel iPSC computer complex.
- It consists of 128(*n*=7) microcomputers, each node consists of a CPU, a floating point processor, local memory, and serial communication interface units

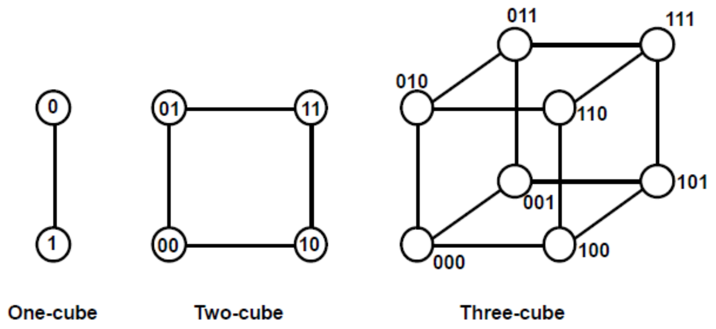


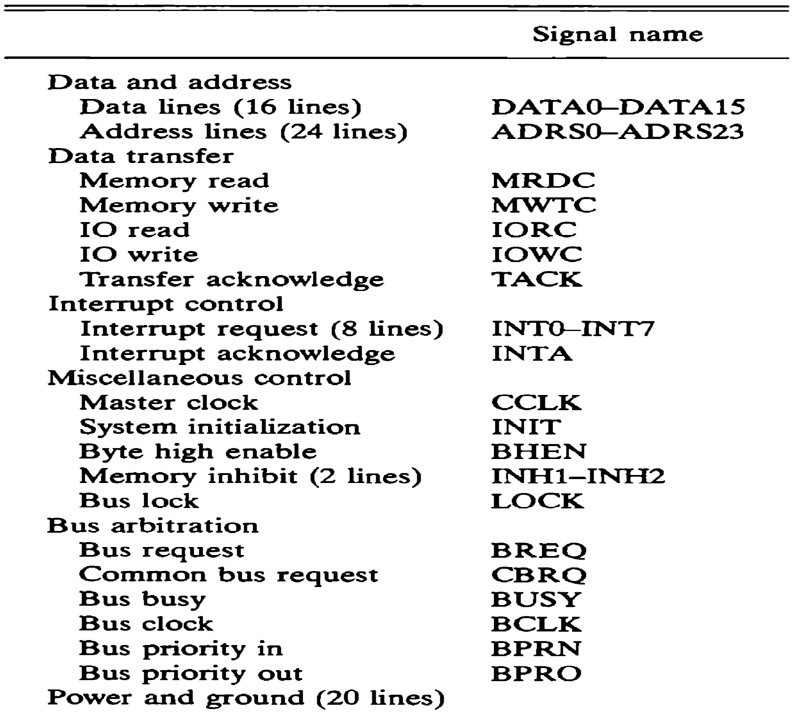
Fig. 5.12 Hypercube structures for n=1,2,3

5.3 Inter-processor Arbitration

- Only one of CPU, IOP, and Memory can be granted to use the bus at a time
- Arbitration mechanism is needed to handle multiple requests to the shared resources to resolve multiple contention
- SYSTEM BUS:
  - A bus that connects the major components such as CPU's, IOP's and memory
  - A typical System bus consists of 100 signal lines divided into three functional groups: data, address and control lines. In addition there are power distribution lines to the components.
- Synchronous Bus
  - Each data item is transferred over a time slice
  - known to both source and destination unit
  - Common clock source or separate clock and synchronization signal is transmitted periodically to synchronize the clocks in the system
- Asynchronous Bus
  - Each data item is transferred by Handshake mechanism
    - Unit that transmits the data transmits a control signal that indicates the presence of data
    - Unit that receiving the data responds with another control signal to acknowledge the receipt of the data

- Strobe pulse -supplied by one of the units to indicate to the other unit when the data transfer has to occur

Table 5.1 IEEE standard 796 multibus signals



	Signal name
Data and address	
Data lines (16 lines)	DATA0–DATA15
Address lines (24 lines)	ADRS0–ADRS23
Data transfer	
Memory read	MRDC
Memory write	MWTC
IO read	IORC
IO write	IOWC
Transfer acknowledge	TACK
Interrupt control	
Interrupt request (8 lines)	INT0–INT7
Interrupt acknowledge	INTA
Miscellaneous control	
Master clock	CCLK
System initialization	INIT
Byte high enable	BHEN
Memory inhibit (2 lines)	INH1–INH2
Bus lock	LOCK
Bus arbitration	
Bus request	BREQ
Common bus request	CBRQ
Bus busy	BUSY
Bus clock	BCLK
Bus priority in	BPRN
Bus priority out	BPRO
Power and ground (20 lines)	

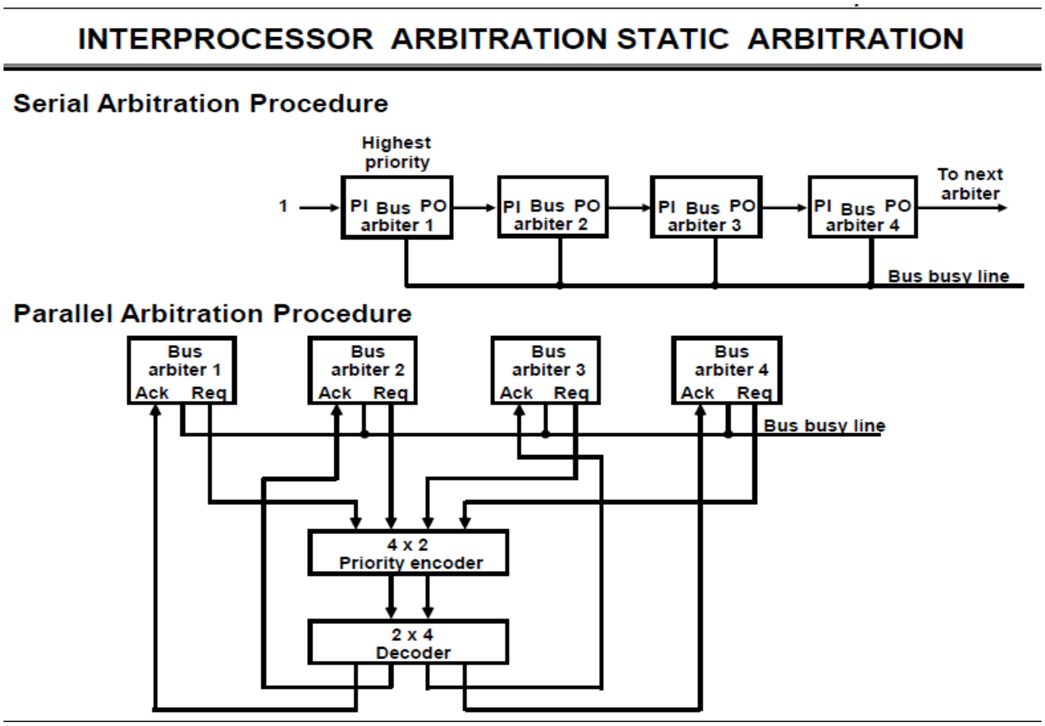


Fig. 5.13 Inter-processor arbitration static arbitration

Interprocessor Arbitration Dynamic Arbitration

- Priorities of the units can be dynamically changeable while the system is in operation
- Time Slice
  - Fixed length time slice is given sequentially to each processor, round-robin fashion
- Polling
  - Unit address polling -Bus controller advances the address to identify the requesting unit. When processor that requires the access recognizes its address, it activates the bus busy line and then accesses the bus. After a number of bus cycles, the polling continues by choosing a different processor.
- LRU
  - The least recently used algorithm gives the highest priority to the requesting device that has not used bus for the longest interval.
- FIFO
  - The first come first serve scheme requests are served in the order received. The bus controller here maintains a queue data structure.
- Rotating Daisy Chain
  - Conventional Daisy Chain -Highest priority to the nearest unit to the bus controller
  - Rotating Daisy Chain –The PO output of the last device is connected to the PI of the first one. Highest priority to the unit that is nearest to the unit that has most recently accessed the bus(it becomes the bus controller)

<u>5.4 Inter processor communication and synchronization:</u>

- The various processors in a multiprocessor system must be provided with a facility for *communicating* with each other.
  - A communication path can be established through *a portion of memory* or *a common input-output channels*.

- The sending processor structures a request, a message, or a procedure, and places it in the memory mailbox.
    - *Status bits* residing in common memory
    - The receiving processor can check the mailbox *periodically*.
    - The response time of this procedure can be time consuming.
- A more efficient procedure is for the sending processor to alert the receiving processor directly by means of an *interrupt signal*.
- In addition to shared memory, a multiprocessor system may have other shared resources.
    - e.g., a magnetic disk storage unit.
- To prevent conflicting use of shared resources by several processors there must be a provision for assigning resources to processors. i.e., operating system.
- There are three organizations that have been used in the design of operating system for multiprocessors: *master-slave configuration*, *separate operating system*, and *distributed operating system*.
- In a master-slave mode, one processor, master, always executes the operating system functions.
- In the separate operating system organization, each processor can execute the operating system routines it needs. This organization is more suitable for *loosely coupled systems*.
- In the distributed operating system organization, the operating system routines are distributed among the available processors. However, each particular operating system function is assigned to only one processor at a time. It is also referred to as a *floating operating system*.

Loosely Coupled System

- There is *no shared memory* for passing information.
- The communication between processors is by means of message passing through *I/O channels*.
- The communication is initiated by one processor calling a *procedure* that resides in the memory of the processor with which it wishes to communicate.

- The communication efficiency of the interprocessor network depends on the *communication routing protocol, processor speed, data link speed, and the topology of the network*.

Interprocess Synchronization

- The instruction set of a multiprocessor contains basic instructions that are used to implement communication and synchronization between cooperating processes.
    - Communication refers to the exchange of data between different processes.
    - Synchronization refers to the special case where the data used to communicate between processors is control information.
- Synchronization is needed to enforce the *correct sequence of processes* and to ensure *mutually exclusive access* to shared writable data.
- Multiprocessor systems usually include various mechanisms to deal with the synchronization of resources.
    - Low-level primitives are implemented directly by the hardware.
    - These primitives are the basic mechanisms that enforce mutual exclusion for more complex mechanisms implemented in software.
    - A number of hardware mechanisms for mutual exclusion have been developed.
        - A binary semaphore

Mutual Exclusion with Semaphore

- A properly functioning multiprocessor system must provide a mechanism that will guarantee orderly access to shared memory and other shared resources.
    - Mutual exclusion: This is necessary to protect data from being changed simultaneously by two or more processors.
    - Critical section: is a program sequence that must complete execution before another processor accesses the same shared resource.
- A *binary variable* called a *semaphore* is often used to indicate whether or not a processor is executing a critical section.

- Testing and setting the semaphore is itself a critical operation and must be performed as a single indivisible operation.
- A semaphore can be initialized by means of a *test and set instruction* in conjunction with a hardware *lock* mechanism.
- The instruction TSL SEM will be executed in two memory cycles (the first to read and the second to write) as follows:

R ← M[*SEM*], M[*SEM*] ← 1

5.5 Cache Coherence

cache coherence is the consistency of shared resource data that ends up stored in multiple local **caches**. When clients in a system maintain **caches** of a common memory resource, problems may arise with inconsistent data, which is particularly the case with CPUs in a multiprocessing system.

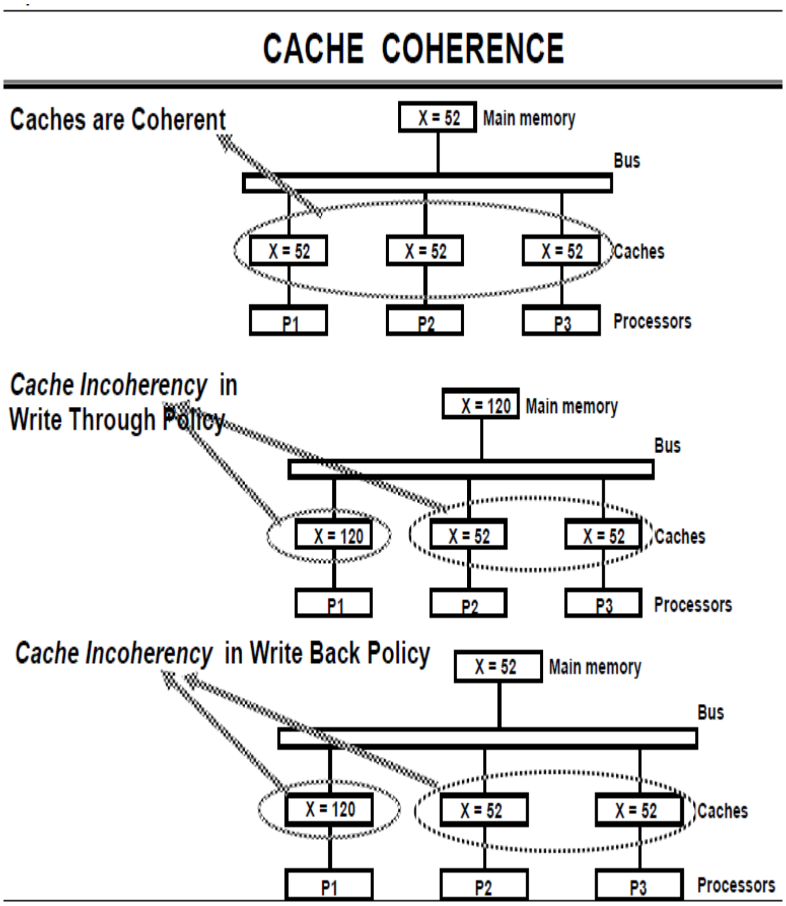


Fig. 5.14 cache coherence

Shared Cache

-Disallow private cache

-Access time delay

Software Approaches

* Read-Only Data are Cacheable

- Private Cache is for Read-Only data
- Shared Writable Data are not cacheable
- Compiler tags data as cacheable and noncacheable
- Degrade performance due to software overhead

* Centralized Global Table

- Status of each memory block is maintained in CGT: RO(Read-Only); RW(Read and Write)
- All caches can have copies of RO blocks
- Only one cache can have a copy of RW block
- Hardware Approaches

* Snoopy Cache Controller

- Cache Controllers monitor all the bus requests from CPUs and IOPs
- All caches attached to the bus monitor the write operations
- When a word in a cache is written, memory is also updated (write through)
- Local snoopy controllers in all other caches check their memory to determine if they have a copy of that word; If they have, that location is marked invalid(future reference to this location causes cache miss)