UNIT-I

DEFINITIONS FOR DIGITAL SIGNALS

Analog versus Digital

- Electronic circuits and systems can be conveniently divided into two broad categories generally referred to as *analog* and *digital.*
- Analog circuits, designed for use with small signals, can be made to work in a linear fashion. An operational amplifier (op amp) connected as an amplifier with a voltage gain of 10 is an analog circuit.
- The output voltage for this circuit will be faithfully amplified version of any signal presented at its input till saturation is reached. This is *linear operation.*
- Digital circuits are generally used with *large signals* and are considered nonlinear.

- Any quantity that changes with time either can be represented as an analog signal or it can be treated as a digital signal.
- For example, place a container of water at room temperature on a stove and apply heat. The measurable quantity of interest here is the change in water temperature. The temperature is recorded *continuously,* and it changes smoothly from 20°C to 80°C.
- While being heated, the water temperature passes through every possible value between 20°C and 80°C. This is an example signal of an analog.

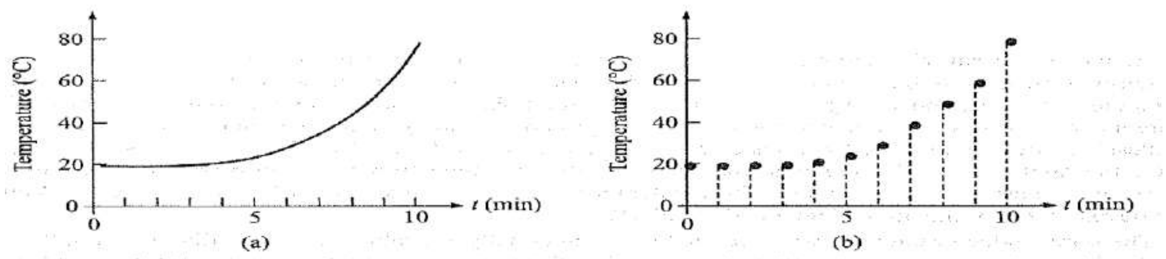


Fig. 1.1 (a) An analog (continuous) signal, (b) A digital (discrete) signal

- In this case, the recorded temperature is *not* continuous. Rather, it *jumps* from point to point, and there are only a finite number of values between 20°C and 80°C say, at an increment of 1°C like 20°C, 21 °C, 22°C, and so on. There are exactly 11 values in this case.
- When a quantity is recorded as a series of distinct (discrete) points, it is said to be *sampled.* This is an example of a digital signal.

- *Digital signals* represent only a finite number of discrete values, Digital circuits and systems can used be to process both analog signals and digital signals.

Binary System

- Digital electronics today involves circuits that have exactly two possible states. A system having only two states is said to be binary *(bi* means "two").
- The *binary number system* has exactly two symbols----0 and 1.The binary number system is used in digital electronics. Electronic circuit can be described in terms of its voltage levels.
- Digital circuit, there are only two The more positive voltage is the *high (HJ* level, and the other is the *low (L)* level. This is immediately related to the binary number system by assigning *L* = 0 and *H* = 1.
- Many functions performed by digital circuits are *logical operations*, and thus the terms true(T)

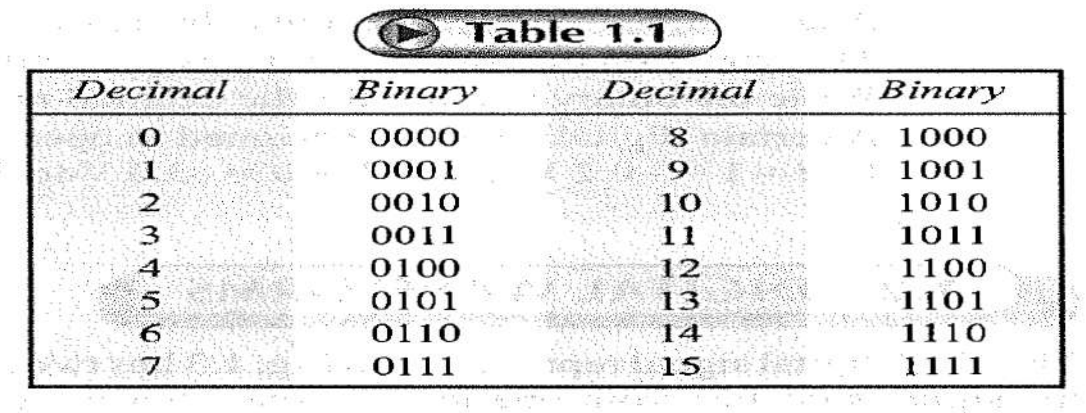


Table 1.1

Decimal	Binary	Decimal	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

and false *(F)* are often used. Choosing *H* = 1 = *T* and *L* = 0 = *F* is called *positive logic.*

- The majority of digital systems utilize positive logic. Note it that is also possible to construct a *negative logic* system by choosing *H* =.O = *F* and *L* = l = *T.*

- Today the majority of digital circuit families utilize a single +5 Vdc power supply, and the two voltage levels used are +5 Vdc and O Vdc. Here is a summary of the two binary states (levels) in this positive logic system.

  +5 V de = *H* = I = *T*

  0Vdc =*L=0=F*

- You can no doubt see how to extend these definitions to include terms such as *on-off go-no go, yes-no,* and so on.
- A lamp or a *light-emitting* diode (LED) is frequently used to indicate digital signal. *On* (illuminated) represents 1, and *off*(extinguished) represents 0.

- As an example, the four LEDs in Fig. 1.2 are indicating the binary number 0101, which is equivalent to decimal 5.

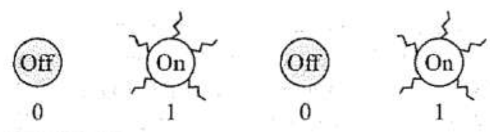


Fig. 1.2 The binary number 0101 (decimal 5)

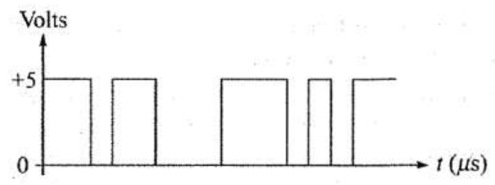


Fig. 1.3 An ideal digital signal

Ideal Digital Signals

- The voltage levels in an *ideal* digital circuit will have values of either + 5 V de or O V de. Furthermore, when the voltages change (switch) between values, they do so in zero time

DIGITAL WAVEFORMS

- The ideal digital signal represented in Fig. 1.3 has two precise voltage levels-+ 5 V de and OV de. Furthermore, the signal switches from one level to the other in zero time. In reality, modern digital circuits can produce signals that approach, but do not quite attain, this ideal behavior.

Voltage levels

- First of all, the output voltage level of any digital circuit depends somewhat on its load, as illustrated in Fig. 1.4a below. When V_0 is high, the voltage should be +5 Vdc.
- In this case, the digital circuit must act as a *current source* to deliver the current *1* to the load. However, the circuit may not be capable of delivering the necessary current *1* while maintaining +5 V de.
- To account for this, it is agreed that any output voltage close to +5 Vdc within a certain range will be considered high. This is illustrated in Fig. 1.4c, where any output voltage level between +5 V de and V H,min is defined as *H* = 1 = *T*. The term V B,min stands for the minimum value of the output voltage when high.
- As we will see, one popular transistor-transistor logic (TTL) family of digital circuits allows V_0 H,min = +3.5 Vdc. In this case, any voltage level between +5 Vdc and +3.5 Vdc is *H*= 1.

- In Fig. 1.4b.v_0 is low, and the digital circuit must act as a *current sink*. That is, it must be capable of accepting a current I_0 from the load and delivering it to ground. In this situation, ~, should be 0 Vdc, but the

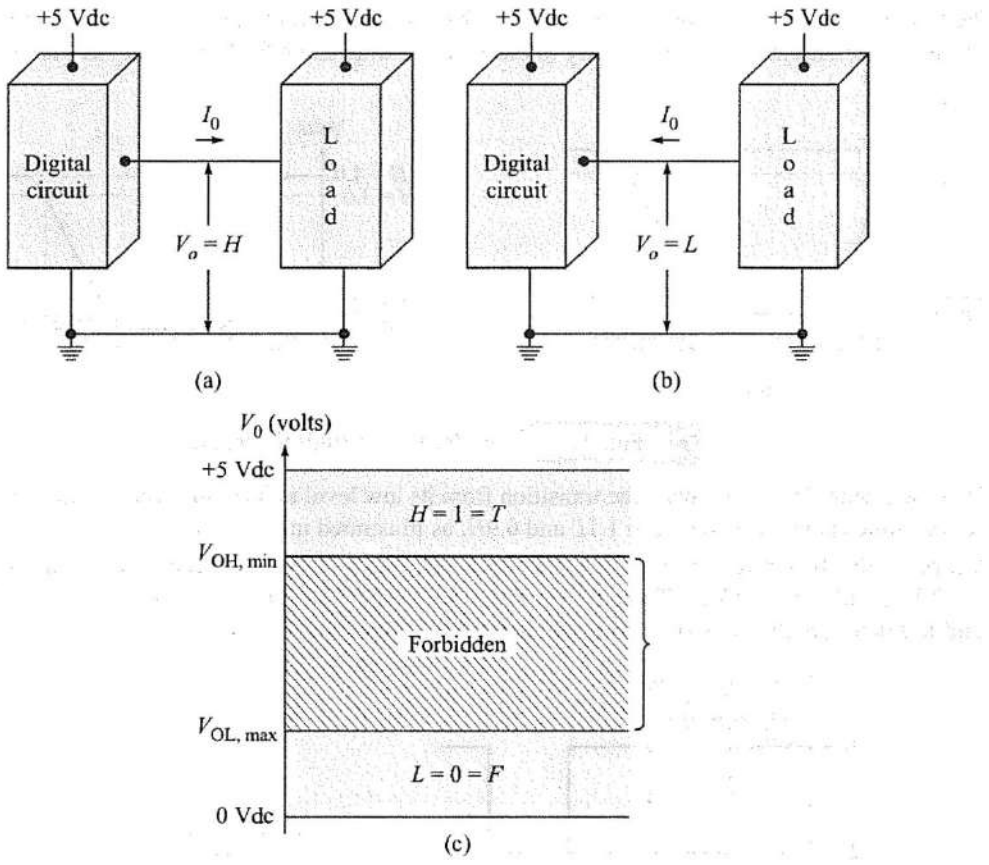


Fig. 1.4 Loading of digital circuit

and V_0 L,max' is defined as $L = 0 = F$. The term V_0 L,max stands for the *maximum* value of the output voltage when low.

- Again, the popular TTL family mentioned above allows V_0 L,max = +0.1 Vdc. Thus, any voltage level between +0.1 Vdc and O Vdc is $L = 0$.

Switching Time

- If the digital circuit were *ideal*, it would change from high to low, or from low to high, in zero time.
- Thus, the output voltage would never have a value in the forbidden range. In reality, it does, in fact, require a finite amount of time for *V* to make the transition (switch) between levels.

- The time required for V_0 , to make the transition from its high level to its low level is defined as *fall time t,*.
- For ease of measurement, it is customary to measure fall time using *0.9H* and 1. IL .

- The time required for V_0 to make the transition from its low level to its high level is defined as *rise time t,.*.Again, rise time is measured between *1.1 L* and *0.9H.*

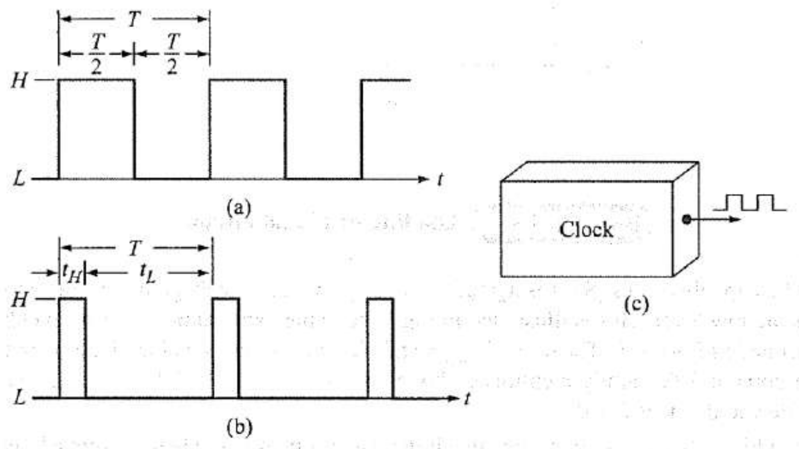


Fig. 1.6 (a) Symmetrical signal with period T, (b) Asymmetrical signal with period T, (c) System clock

Period and Frequency

- There are many occasions where a symmetrical digital signal will be used (clock and counter circuits for instance). The period *T* of this waveform is shown: This is the time over which the signal repeats itself.
- A rectangular waveform such as this can be produced by adding together an infinite (or at least a large number) of sinusoidal waveforms of different frequencies and amplitudes.
- Even though this digital signal is not sinusoidal, it is convenient to define the frequency as/= 1/T. As an example, if the period of this square wave is I µs, then its frequency is found as

$$f = \frac{1}{T} = \frac{1}{1\,\mu s} = \frac{1}{10^{-6}} = 10^6 = 1\text{ MHz}$$

- A symmetrical signal is frequently used as the basis for timing all operations in a digital system. As such, it is called the *clock signal.*
- The electronic circuit used to generate this square wave is referred to as the *system clock.*
- A system clock is simply an oscillator circuit having a very precise frequency. Frequency stability is provided by using a crystal as the frequency-determining element.

Duty Cycle

- *Duty cycle* is a convenient measure of how symmetrical or how unsymmetrical a waveform is. For the waveform there are two possible definitions for duty cycle.

$$\text{Duty cycle } H = \frac{t_H}{T}$$
$$\text{Duty cycle } L = \frac{t_L}{T}$$

- The first definition is the fraction of time the signal is high, and the second is the fraction of time the signal is low. Either definition is acceptable, provided you clearly define which you are using. To express as a percentage, simply multiply by 100.

- Note that the duty cycle for a syletrical wave

$$\text{Duty cycle } H = \text{Duty cycle } L = \frac{T/2}{T} = 0.5 \text{ or } (50\%)$$

DIGITAL LOGIC

Generating Logic Levels

- The digital voltage levels described in Fig. 1.4 can be produced using switches on the next page. The switch is down and $V_0 = L = 0 = 0$ V de. When the switch is up, as $V_0 = H = 1 = +5$ V de.
- A switch is easy to use and easy to understand, but it must be operated by hand.

- A *relay* is a switch that is actuated by applying a voltage Vi to a coil. The coil current develops a magnetic field that moves the switch an n from one contact to the other.
- This is indicated with the dashed line drawn between the coil and the relay arm.
- For this particular relay, $V_0 = L = 0$ Vdc when $v;$ = 0 V de. Applying a voltage $v;$ will actuate the relay, and then $V_0 = H = +5$ V de. This relay could of course be connected so that its output is low when actuated.
- Switches and relays were useful in the construction of early machines used for calculation and/or logic operations.
- In fact, they are still used to a limited extent in modem computer systems where humans must interact with a system. For instance, on-off power switches, reset, start-stop, and load-unload are functions that might require human initiation.

- On the other hand, modem computers are capable of performing billions of switching operations every second! Switches and relays are clearly not capable of this performance, and they have therefore been replaced by transistors (bipolar and/or MOSFET).

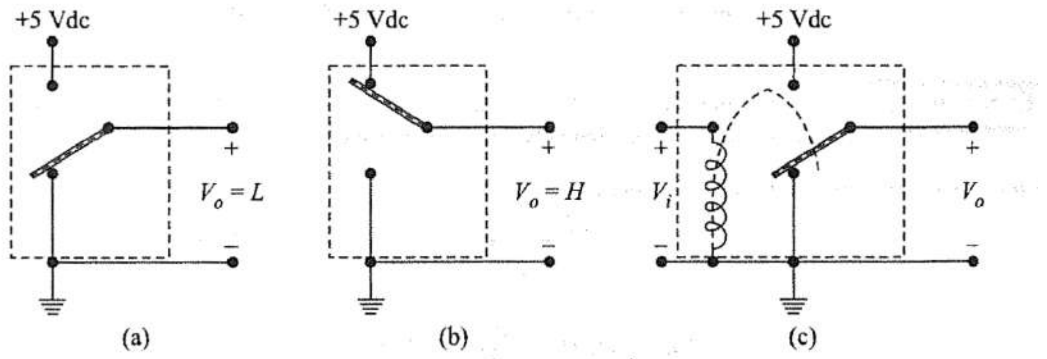
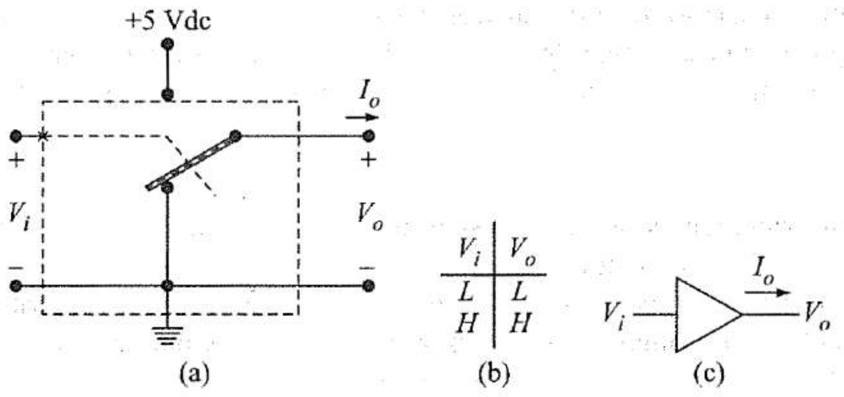


Fig. 1.7 (a) Switch, (b) Switch, (c) Normally low relay

The Buffer

- In order to deliver the necessary load current *1* in digital IC called a *buffer* might be used. A buffer can be thought of as an electronic switch.
- The switch is actuated by the input voltage *v;*. Its operation is similar to the relay. When *v;* is low, the switch is down, and V_0 is low. On the other hand, when *v;* is high, the switch moves up and V_0 is high.
- Operation of this IC is summarized by using the truth table, or table of combinations. There are only two possible input voltage levels *(L* and *JI)*, and the truth table shows the value of V_0 in each case.



V_i	V_o
L	L
H	H

Fig. 1.8 (a) Buffer amplifier model, (b) Truth table, (c) Symbol

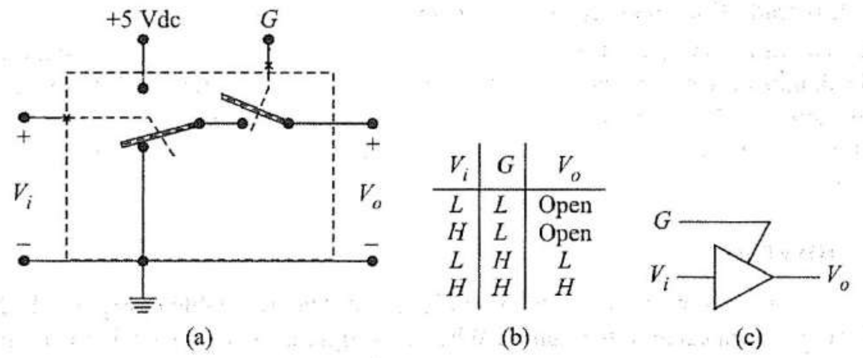
Since the buffer is capable of delivering additional current to a load, it is often called a *buffer amplifier*.

- The traditional amplifier symbol (a triangle) shown in Fig. l.8c is used on schematic diagrams. If you're interested in an actual IC buffer, look in the standard TTL logic family. The 5407 or 7407 is a 14-pin IC that contains six buffers.

The Tri-State Buffer

- At the input of a digital system, there may be more than one input signal of interest. Generally speaking, however, it will be necessary to connect only one signal at a time, and thus there is a requirement to connect or disconnect (switch) input signals electronically.
- Similarly, the output of a digital system may need to be directed to more than one destination, one at a time.

- The logic circuit is a simple buffer with an additional switch controlled by an input labeled *G.* When *G* is low, this switch is open and the output is "disconnected" from the buffer. When *G* is high, the switch is closed and the output follows the input.
- That is, the circuit behaves as an ordinary buffer amplifier. In effect, the control signal *G* connects the buffer to the load or disconnects the buffer from the load.



V_i	G	V_o
L	*L*	Open
H	*L*	Open
L	*H*	*L*
H	*H*	*H*

Fig. 1.9 **A tri-state buffer: (a) Model, (b) Truth table, (c) Symbol**

- When G is high, v_0 is either high or low (two states). However, when *G* is low, the output is in effect an *open circuit* (a third state).
- Since there are *three* possible states for V_0, this circuit is called a *tri-state bzttfer. (Tri* stands for "three", and thus the term *three-state buffer* is often used.)

The Inverter

- One of the most basic operations in a digital system is *inversion,* or *negation.* This requires a circuit that will invert a digital level. This logic circuit is called an *inverter*, or sometimes a *NOT circuit.*
- The switch arrangement in is an inverter. When the input to this circuit is low, the switch remains up and the output is high.
- When the input is high, the switch moves down and the output is low. The truth table for the inverter. Ob Clearly the output is the negative, or the inverse, of the input.

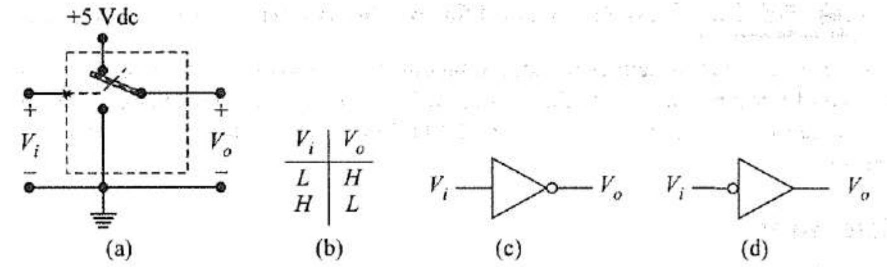


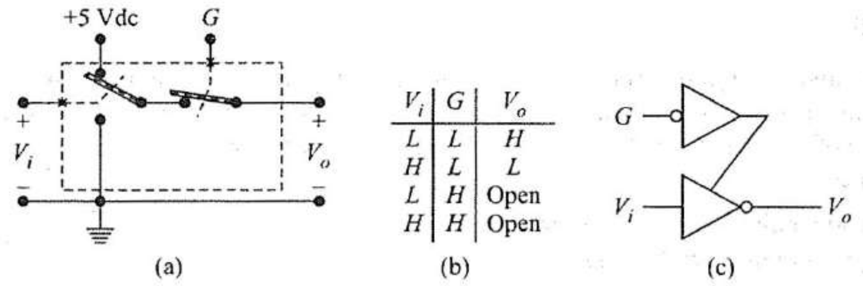
Fig. 1.10 **Digital inverter: (a) Model, (b) Truth table, (c) Symbol, (d) Another symbol**

- When the inverter is used as a logic circuit, *His* often defined as the "true" state, while *L* is defined as the "false" state. In this sense, the inverter will always provide at its output a signal that is the inverse, or complement, of the signal at its input.
- It is thus called a *negation* or NOT circuit. This makes sense, since there are only two possible states, and therefore *NOT H* must be *L* and NOT *L* must be *H.*
- The inverse complement or of a signal is shown by writing a bar above the symbol. For instance, the complement of *A* is written as $\overline{A}$ or *A'* and this is read as *"A* bar" A logic expression for the inverter.

The Tri-state Inverter

- A *tri-state inverter* is easy to construct, that when *G* is low, the inverter to is connected the output. When G is high, the enable switch opens, and the output is disconnected from the inverter.

- The standard logic symbol for this tri-state inverter is given in Fig. 1.11 c. The inverting amplifier symbol indicates that *V_0*, is the inverse of *v;* (the small circle is at the amplifier output).



V_i	G	V_o
L	L	H
H	L	L
L	H	Open
H	H	Open

Fig. 1.11 **Inverting tri-state buffer: (a) Model, (b) Truth table, (c) Symbol**

The AND Gate

- An *AND gate* is a digital circuit having two or more inputs and a single output. The inputs to this gate are labeled *V_1, V_2, V,, ... Vn*, (there are n inputs), and the output is labeled *V_0*.
- The operation of an AND gate can be expressed in a number of different, but equivalent, ways. For instance.

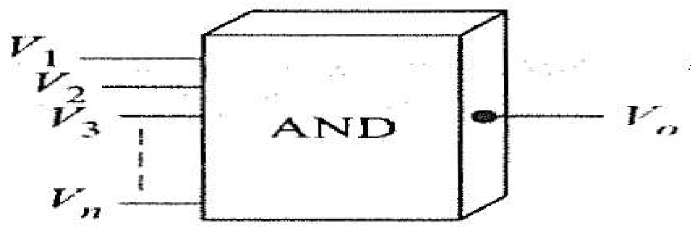


Fig. 1.12 **AND gate**

1. If *any* input is low, V_o will be low.
2. V_o will be high only when all inputs are high.
3. $V_o = H$ only if $V_1 = H$, and $V_2 = H$, and $V_3 = H$, . . . and $V_n = H$.

This last statement leads to the designation *AND gate*, since V_1 *and* V_2, *and* V_3, . . . *and* V_n must all be high in order for V_o to be high.

- A model for an AND gate having 2 inputs. This gate can be used to make "logical" decisions; for example, "If V_1 and V_2, then V_0,." As a result, it is referred to as a digital logic circuit, as are all AND gates.
- From the model, it is seen that $V_1 = H$ closes the upper switch, and $V_2 = H$ closes the lower switch. Clearly, $V_0 = H$ only when both V_1 and V_2 are high. This can be expressed in the form of a logic equation written as

$$V_o = V_1 \text{ AND } V_2$$

The OR Gate

- An *OR gate* is also a digital circuit having 2 or more inputs and a single output as indicated in

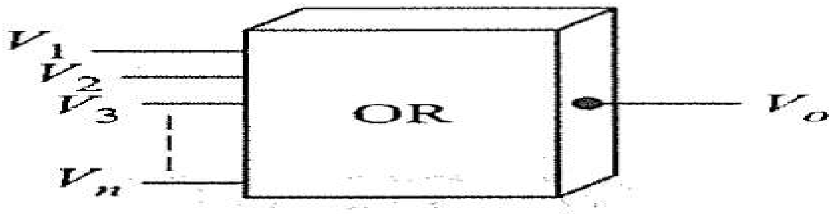


Fig. 1.14 OR gate

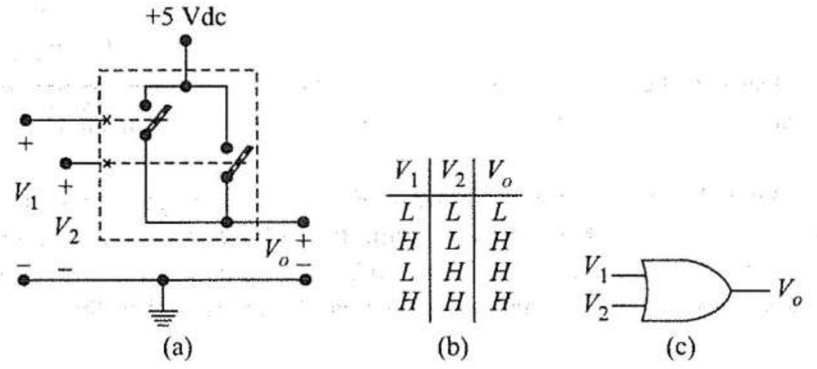
1. V_o will be low only when all inputs are low.
2. If any input is high, V_o will be high.
3. $V_o = H$ if V_1, *or* V_2 *or* V_3, . . . *or* $V_n = H$.

This last statement leads to the designation OR gate, since $V_o = H$ only if V_1 *or* V_2, *or* V_3, . . . $V_n = H$.

- A model for an OR gate having 2 inputs is shown in Fig. 1.15a. This gate can be used to make "logical"decisions; for example, "If V_1 or V_2, then V_0." As a result, it is referred to as a digital logic circuit, as are all OR gates.
- From the model, it is seen that $V_1 = H$ closes the upper switch, and $V_2 = H$ closes the lower switch. Clearly, $V: = H$ if either V_1 or V_2 is high. This can be expressed in the form of a "logic" equation written as

$$V_o = V_1 \text{ OR } V_2$$

- The operation is summarized in the truth table in Fig. 1.15b. The symbol for a 2-input OR gate is shown in Fig. 1.15c. Thus, OR is a logic operation which is realized here through gate a logic gate.



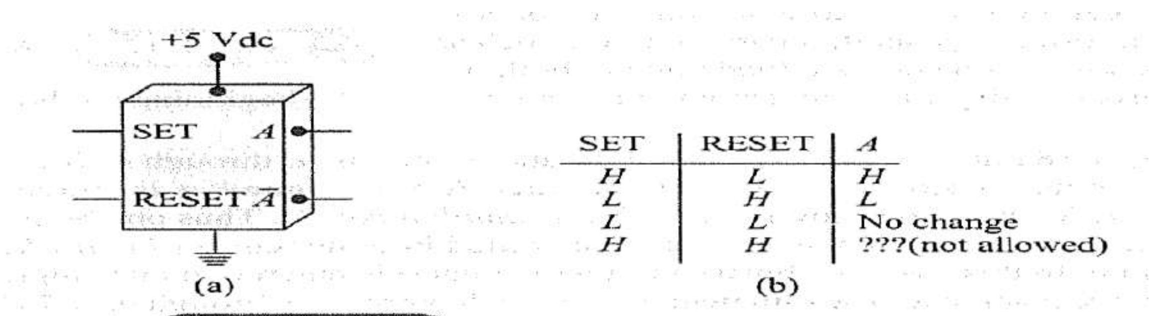
V_1	V_2	V_o
L	L	L
H	L	H
L	H	H
H	H	H

Fig. 1.15 **Two-input OR gate: (a) Model, (b) Truth table, (c) Symbol**

MOVING AND STORING DIGITAL INFORMATION

Memory Elements

- A *digital memory element* is a device or perhaps a circuit that will maintain a desired logic level at its output till it is changed by changing the input condition.
- The switch is placed such that its output is low, and it will remain low without any further action.
- Thus, it will "remember" that $V_0 = L$. Since $L = 0 = 0$ Vdc, the switch can be thought of as "holding" or "storing" a logic 0. In Fig. I. 7b, $V_0 = H$, and it will remain high without any further action. The switch remembers that $V_0 = H$.
- In this case, the switch is holding or storing a logic 1, since H = I = +5 V de. It is easy to see that this switch can be used to store a digital level, and it will remember the stored level indefinitely.
- The simplest electronic circuit used as a memory element is called a *flip:flop.* Since a flip-flop is constructed using transistors, its operation depends upon de supply voltage(s).



SET	RESET	*A*
H	*L*	*H*
L	*H*	*L*
L	*L*	No change
H	*H*	???(not allowed)

Fig. 1.16 (a) A flip-flop. (b) Truth table

- The flip-flop can be used to store a logic level (high or low), and it will retain a stored level indefinitely provided the de supply voltage is maintained. An interruption in the de supply voltage will result in loss of the stored logic level.
- When power is first applied to a flip-flop (turning the system on first thing in the morning), it will store either a high or a low. This is a "random" result, and it must be accounted for in any digital system. Generally, a signal such as MASTER RESET or power-on reset will be used to *initialize* all storage elements.

- The two inputs are SET and RESET, and the output is *A.* The output labeled *A* is simply the *inverse* of *A.* Here's how it works:

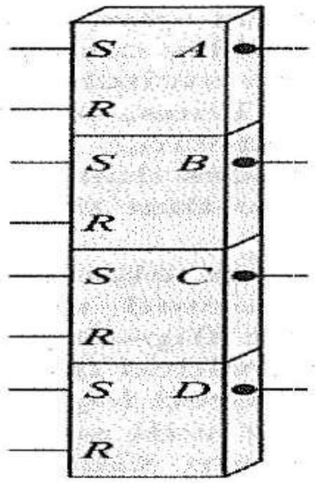
1. When SET= Hand RESET = *L,* the flip-flop is *set,* and *A* = *H.*
2. When SET = *L* and RESET = *H,* the flip-flop is reset, and *A* = *L.*
3. Holding SET= Land RESET= *L* disables the flip-flop and its output
remains unchanged.
4. Applying SET= Hand RESET= Hat the same time is not allowed, since this is a request to *set* and *reset* at the same time-an impossible request!

- To summarize, the flip-flop is SET, it stores a high (a logic 1). When it is RESET, it stores a low (a logic 0).
- A simple flip-flop such as this is often called a *latch,* since its operation is similar to a switch. A 7475 is an IC in the TTL family that contains four similar flip-flops.

Registers

- A group of flip-flops can be connected together to store more than a single logic level. For instance, the four flip-flops can be used to store four logic levels.

- As such, they could be used to store any of the ten binary numbers given in Table. As an example, if *A* is SET, *B* is RESET, C is SET, and *D* is RESET, this will store the binary number *DCBA* = *LHLH* = 0101, which is equivalent to decimal 5.
- When we speak of decimal numbers, in each position a number is called a *decimal digit,* or simply a digit. For example, the decimal number 847 has three digits.
- When we speak of binary numbers, each position in the number is called a *binary digit,* or *bit.* (The term "binary digit!" has been shortened to "bit.")
- For example, the binary number 0101 is composed of four bits; It is a 4-bit binary number. The four flip-flops can be used to store any 4-bit binary number.
- A group of flip-flops used to store a binary number is called a *register,* or sometimes a *storage register.* The register is a 4-bit register.
- There are eight flip-flops in an 8-bit register, and so on. In the TTL family, the 7 4198 is an 8-bit register. Clearly a register can be used to store decimal numbers in their binary equivalent form.
- In general, binary numbers such as this are referred to as *data.* A register is a fundamental building block in a microprocessor or digital computer, and you can now see the beginnings of how these systems are used for computation.



S = SET R = RESET

Fig. 1.17 A four-bit register

- The register in Fig. 1.18a has 8 inputs, 1 through 8, and 8 outputs, *a* through *h.* It is constructed using eight flip-flops and some additional electronic circuits.

- A binary number is stored in this register by applying the appropriate level (high or low) at each input *simultaneously.* Thus one bit is "shifted" into each flip-flop in the register.
- The binary number is said to be shifted into the register *in parallel,* since all bits are entered at the same time. In this case, binary the number (or data) is entered in one single operation.
- Once a number is stored in this register, it appears immediately at the 8 outputs, *a* through *h.* A 74198 is an example of an 8-bit *parallel register.*
- The register in Fig. 1.18b has a single input and a single output. It is also constructed using eight flip-flops and some additional electronic circuits.
- It will store an 8-bit binary number, but the number must be entered into the register one bit at a time at the input. It thus requires eight operations to store an 8-bit number. This is how it is done.
- The first bit of the binary number is entered in flip-flop *A* at the input. The second bit is

then entered into flip-flop *A,* and at the same time the first bit in flip-flop *A* is passed along (shifted) to flip-flop *B.*

- When the third bit enters *A*, the bit in *A* goes to *B* and the bit in *B* goes to C. This shift right process is repeated, and after eight operations, the 8-bit number will be stored in the register. Since the bits are entered one after the other in a serial fashion, this is called a *serial register*.

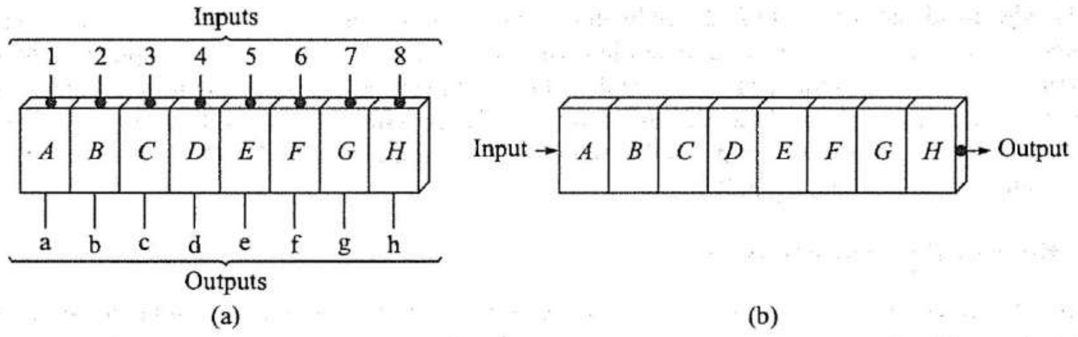


Fig. 1.18 **(a) An 8-bit parallel register, (b) An 8-bit serial register**

- For a stored number to be extracted from this register, the bits must he shifted through the flip-flops from left to right.

- The stored number will then appear at the output, one bit at a time. It requires eight operations, or eight right shifts, to extract the stored number.
  A 74164 TTL is an example of an 8-bit serial register (this particular IC
  also provides parallel outputs).

Transferring Digital Data

- A register is used to enter data (binary numbers) into a microprocessor or computer. A register is also used to extract data from a computer and direct it to an external destination are Wire cables generally the means for connecting systems.
- If a parallel register is used, the data is said to be shifted in parallel. The connector in this case must have one pin for each bit, and the cable must have at least one wire for each bit.
- An 8-bit register requires a cable having at least 8 wires, a 16-bit register must have at least 16 wires, and so on.
- Data are also transferred (shifted) between registers within a digital system. Instead of drawing all 8 (or16 or 32) wires on a schematic, it is common practice to use an arrow between the registers.
- The number 8 in parentheses means that there are eight wires. In this case, there are eight connections used to transfer 8 bits of data in parallel from register *A* to register *B.* The eight wires represented by this arrow are called a *data bus.*

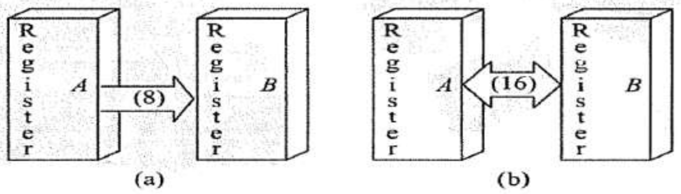


Fig. 1.19 (a) An 8-bit data bus, (b) A 16-bit, bidirectional data bus

- The double arrow shown in Fig. 1.19b means 16 bits of data can be shifted in parallel from *A* to *B* or from *B* to *A.* This is a 16-bit *bidirectional data bus.*

- On the other hand, data can be shifted serially into or out of a serial register, and only one wire (connection) is required for the data.
- Clearly, parallel operation will transfer data into or out of a computer system much more rapidly than serial operation.
- The price paid for this gain in speed is an increase complexity, in in terms of both the electronic circuits and the increased number of connections (wires in the cable).
- The computer connector where data is entered or extracted is frequently called a *port.* Nearly all computer systems have available both a serial port and a parallel port.

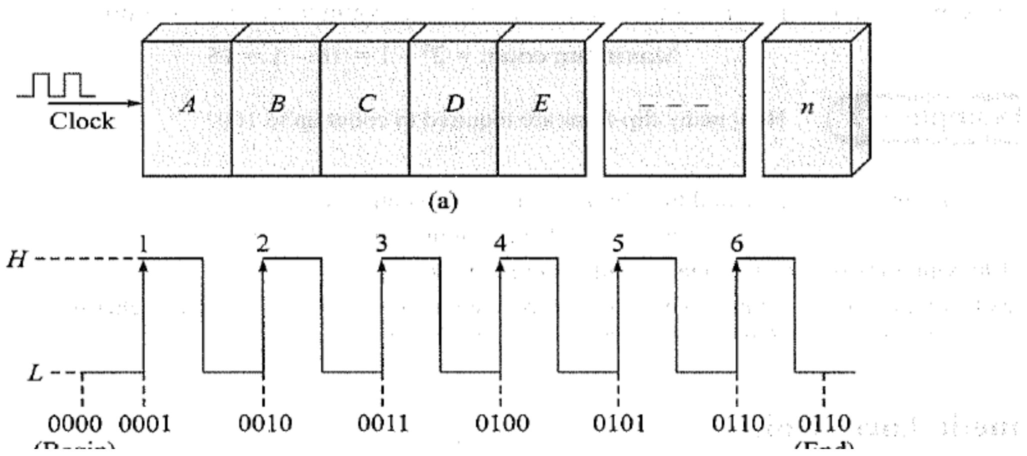
Magnetic and Optical Memory

- Any memory element must be capable of storing or retaining only two logic levels, and there are numerous devices with the appropriate electronic circuits used for this purpose.
- One of the most common systems for memory makes use of the fact that a magnetic material can be magnetized with two different orientations.
- Thus, magnetizing spots on a strip of magnetic tape, or on a hard disk with a magnetic coating, or on a magnetic floppy disk are well known used and widely memory systems.

DIGITAL OPERATIONS

Counters

- It was mentioned previously that counting is an operation easily performed by a digital circuit.
- A digital circuit designed to keep track of a number of events, or to count, is called a *counter:* The counter is constructed using a number of flip-flops *(n)* and additional electronic circuits.
- It is similar to a storage register, since it is capable of storing a binary number. The input to this counter is the rectangular waveform labeled *clock.* Each time the clock signal changes state from low to high, the counter will add one (1) to the number stored in its flip-flops.



- In other words, this counter will count the number of clock transitions from low to high. A clock having a small circle (bubble) in the input side would count clock transitions from high to low. This is the concept of *active* is, an action occurs when the input is low.

- A four-flip-flop counter can count decimal numbers from 0 to 15. To count higher, it is necessary to add more flip-flops. It is easy to determine the maximum decimal count in terms of the number of flip-flops using the following relation

Maximum count = 2" - 1 (1.1)

where *n* = number of flip-flops.

- The term 2" means 2 raised to the nth power, that is, 2 multiplied by itself *n* times. For example,

2^2 = 2 x2 = 4

2^3 = 2 X 2 X 2 = 8

2^4 = 2 X 2 X 2 X 2 '"-' 16

2^5 = 2 X 2 X 2 X 2 X 2 = 32

2^6 = 2 X 2 X 2 X 2 X 2 X 2 = 64

2^7 = 128

2^8 = 256

2^9 = 512

2^{10} = 1024

Arithmetic logic Unit

- An *arithmetic logic unit (ALU)* is a digital circuit capable ofperfonning both *arithmetic* and *logic* operations.
- The basic arithmetic operations performed by an ALU are addition (+) and subtraction(-:). Multiplication (*) and division (/) of digital numbers are accomplished with other digital circuits.
- Logic operations will usually include inversion (NOT), AND, and OR. The ALU represented has two data inputs; the

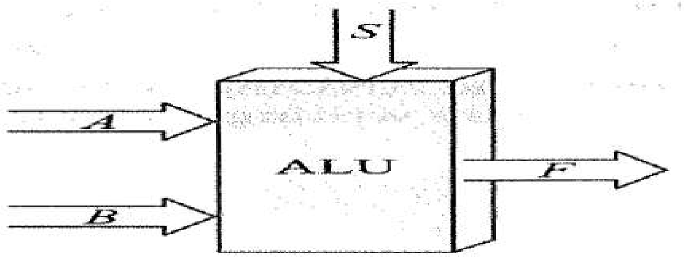
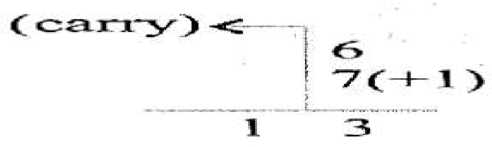


Fig. 1.22 An arithmetic logic unit (ALU)

Addition and Subtraction:

- If the proper digital levels are applied to the inputs of the S bus, the ALU can be used to add two digital numbers.

- The two numbers to be added are represented by the proper logic levels at *A* and *B*, and the SUM of these two numbers will appear at output *F*.
- In event the sum of the two numbers generates a carry, an *H* will appear at the CARRY OUT. To illustrate, suppose we wish to perform the addition 6 + 7 = 13. Here's how we might do it with decimal numbers.



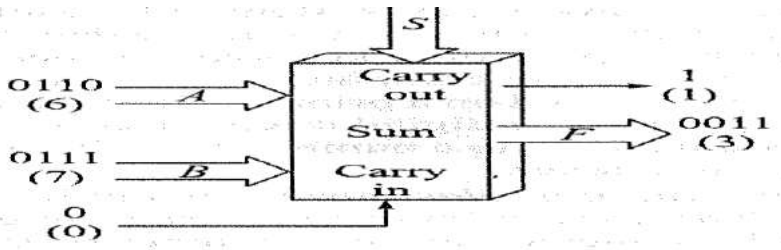


Fig. 1.23 An ALU used for addition

- The digital levels illustrated result in the addition of these two numbers. The equivalent numbers decimal are shown in parentheses. Notice that the CARRY IN allows this ALU to add two numbers, plus a carry.
- By changing the control levels at the S bus, this ALU will determine the difference *A -B* (subtraction).
- In this case, the digital levels at the *F* bus represent the DIFFERENCE, rather than the SUM.

Logic Functions

- By changing the digital levels at the S bus, the ALU can be used to perform a number of different logic functions relative to the two digital inputs.
- The desired function appears at the *F* bus. Here are some of the possibilities:

$F = \overline{A}$

$F = \overline{B}$

F=A AND B

F=A OR B

- The operations are carried out "bit by bit." For example, If A= 1010 then F= $\bar{A}$= 0101

If A= 1010 and if B = 0ll0, then

F= A AND B = 1010 AND 0110 = 0010

- In this case, the AND ing is done on the corresponding bit of each input. There are tour AND operations.

- It's easier to see by writing the data as follows:

```
A = 1010
      | | | |
B = 0110
F = 0010
```

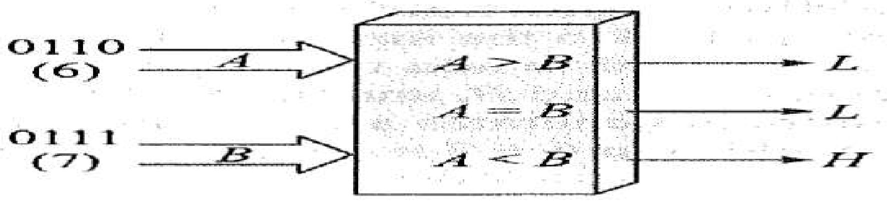


Fig. 1.24 A comparator

The "vertical lines" between A and B show which bits are AND.

Comparison

- Comparing the magnitude of two numbers is an important logical operation. The circuit is a *comparator*. It is capable of comparing two digital numbers and indicating whether the magnitude of one is greater than, less than, or equal to the other.
- For example, if A= 0110 (decimal 6) and B = 0111 (decimal 7), then the output $A < B$ will be high.
- The other two outputs will be low. A 7485 in the TTL family is a 4-bit comparator similar. Also, the 74181 ALU can be used with the same results.

Input/Output

- In order for any digital system to be useful, there must be some provision for entering data into the system and also some method of extracting data from the system.
- In the case of a computer, information is frequently entered by typing on a keyboard or perhaps by using a magnetic floppy disk. Useful information can be obtained from the computer by examining the visual displays on a cathode-ray tube (CRT) or by reading material produced on a printer.
- Clearly there is a requirement to connect multiple input devices, one at a time, to the system. The digital circuit used for this operation is a multiplexer.
- Likewise, there is a need to connect the system output to a number of different destinations, one at a time. The digital circuit used for this purpose is a demultiplexer.

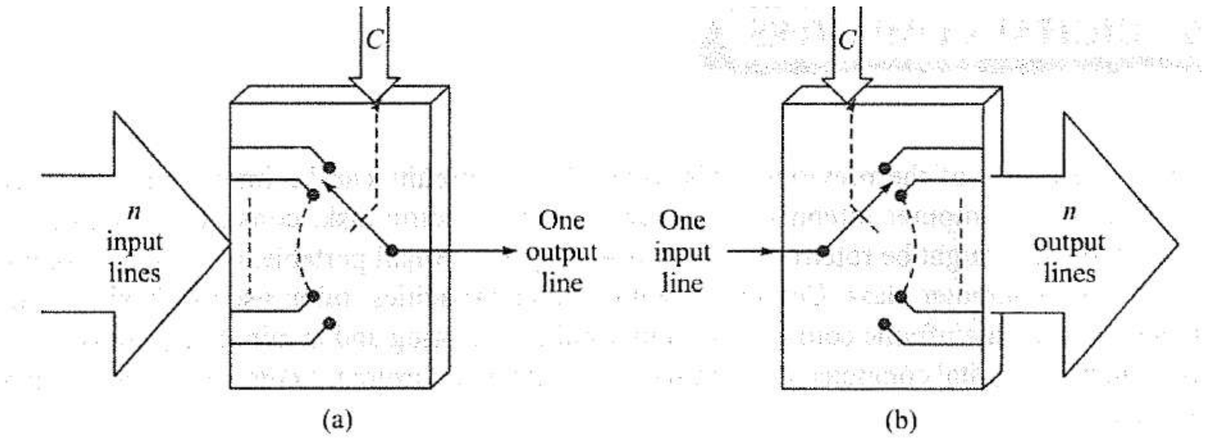


Fig. 1.25 (a) A multiplexer (MUX), (b) A demultiplexer (DEMUX)

- The term *multiplex* means "many into one." A *multiplexer (MUX)* can be represented. There are *n* input lines. Each line is used to shift digital data serially.
- Thus, data from multiple sources can be connected to a single input port, one at a time. An example ofa MUX is the 74150 in the TTL family. It has 16 input lines and a single output line.

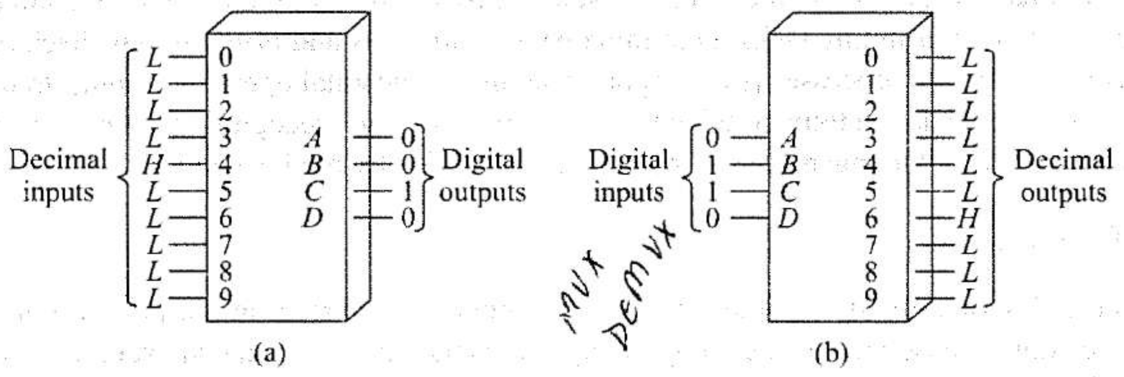


Fig. 1.26 (a) An encoder, (b) A decoder

- The opposite of multiplex is *demultiplex,* "onewhich means into many." A *demultiplexer (DEMUX)* can be represented. This digital circuit simply connects the single data input line to one of the *n* output lines, one at a time, according to the levels on the *C* bus.
- Thus serial data from the computer output port can be directed to different destinations, one at a time. An example of a DEMUX is the TTL 74154, which can be used to connect a single input to any one of 16 outputs, one at a time.
- Any information entered into a digital system must be in the form of a digital number. A circuit that changes data into the required digital form is called an *encoder.*
- The encoder on the next page will change a decimal number into its binary equivalent. It may be used with a keyboard.
- For instance, depressing the number 4 key on a keyboard will cause input line 4 to this encoder to be high (the other inputs are all low). The result will be decimal 4, binary 0100, at the encoder output as shown.

- Taking digital information from the output of a computer and changing it into another form is accomplished with a *decoder,* for example, changing the digital number O110 (decimal 6) into its decimal form.

DIGITAL COMPUTERS

Terms

- Digital circuits can be interconnected to construct a digital computer.
- A computer intended to perform a very specific task, constructed with a minimum number of components, might be referred to as a *microcomputer*.
- Small portable, or desktop, computers are usually in the microcomputer class. Computers with greater capacities, often used in business, are called *minicomputers.*
- A large mainframe computer system capable of storing and manipulating massive quantities of data, for example, a digital computer system used by a bank or an insurance company, might then be called *maxi computer*.

Uses

- The inclusion of an ALU with additional logic circuits provides arithmetic capabilities (addition, subtraction, multiplication, division).
- The logic portion of the ALU means the computer can be used to make logical decisions. Beyond these basic functions, a digital computer can be used to process data (balance bank accounts), to rapidly perform otherwise time-consuming tasks (determine payroll amounts and print out paychecks), to precisely monitor and control intricate processes (life support systems in a hospital operating room).

Basic Configurations

- A microcomputer designed to control a given machine, process, or system might be represented in fig. The control produced signals by the computer appear as the output bus and are sent to an output device.
- Here, the signals are properly conditioned and sent to the mechanism being controlled. The controlled entity must then send signals indicating its present condition back to the computer via an input device and via the input bus.
- The computer analyzes these present condition signals, determines any necessary action, and sends required correction signals to out the system.

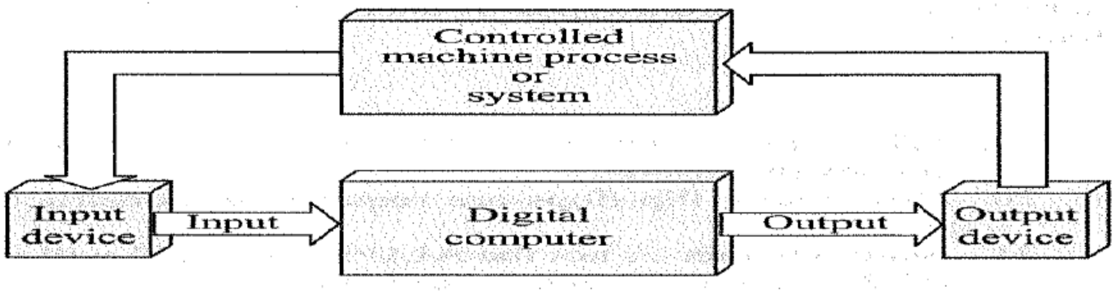


Fig. 1.27 A digital computer based system

A microcomputer system might be designed to irrigate the lawn area of a park. Watering is to be done only at night, when the soil moisture falls below a given value.

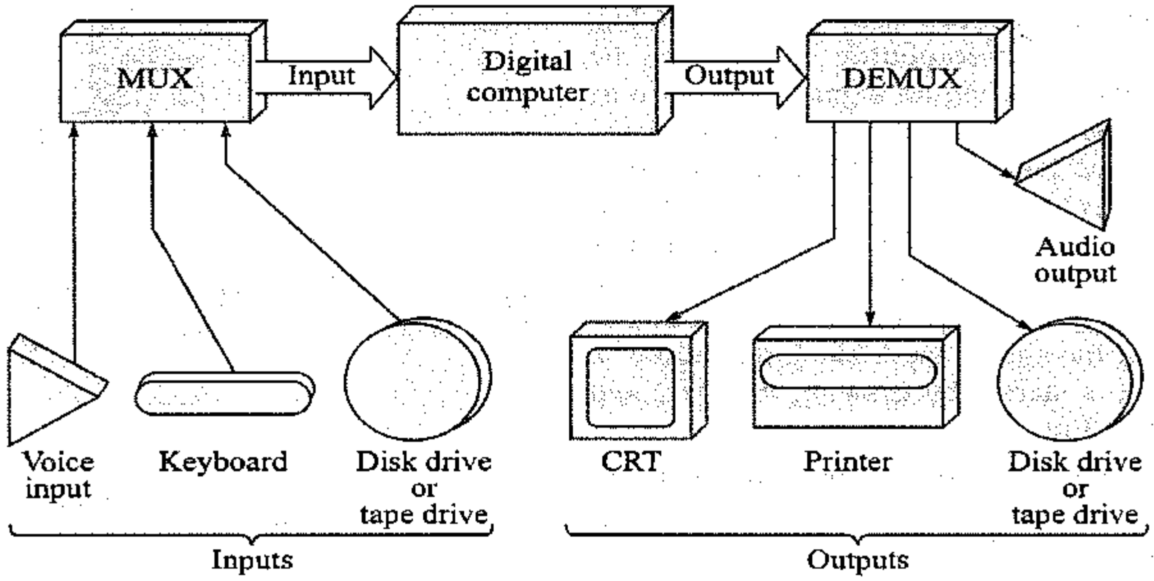


Fig. 1.28 Block diagram of a computer system

- This allows the connection of a number of different input devices:

A keyboard for typewritten entry of alphanumeric informationA disk drive or tape drive for entering data stored in magnetic form A microphone for voice input

- The DEMUX on the output bus allows numerous possibilities for receiving information from the computer: Familiar The CRT for a visual display.A printer to provide printed material (called *hard copy).*

- A disk or tape drive to record data in magnetic form Perhaps a speaker for audio information
- A minicomputer such as this can be used for many different tasks. It can be used as a word processor, for data processing, for communication via telephone (both voice and fax), for training in an educational setting, for computer games, and so on!
- For instance, a maxi computer will likely have more than one printer, and perhaps even different types of printers.
- It will generally have a large number of users, all of whom desire access to the system at the same time.
- One workstation must then be provided for each user.
- A keyboard and a CRT are the minimum components required at each workstation. The digital circuits used to construct maxi computer systems are necessarily more complicated than minicomputer systems, and they may operate at a much faster rate.

Basic Computer Architecture

- The *central processing unit (CPU)* is the brain of a digital computer. It is constructed using an ALU along with a number of registers and counters.
- The CPU is therefore the primary center for computation and decision making. All the operations within the CPU, and indeed within the computer itself, must be carefully coordinated.
- A digital signal refered to as the *system clock* is used as a reference to time when specific operations take place. The clock signal is usually a periodic, rectangular waveform.
- Using a crystal in the clock circuit allows the accuracy and stability of the clock frequency, to be controlled with great precision.
- The clock provides a "heartbeat" for the computer. A block diagram of a digital computer is started by drawing the CPU and clock.
- The CPU is capable of computation and decision, but it must have specific instructions telling it exactly ·*what* to do and *when* to do it. This set of instructions is called a *program.*

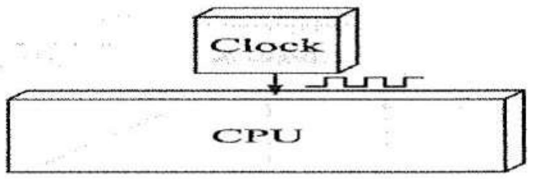


Fig. 1.29 The "heart" and "brain" of a digital computer

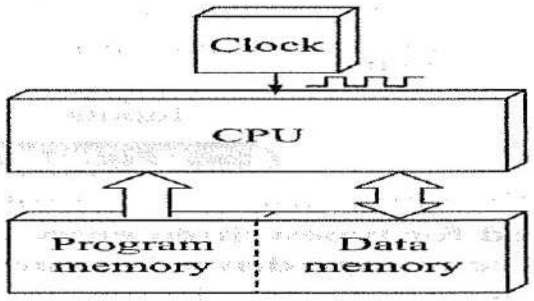


Fig. 1.30 CPU with memory block

- A program is a detailed list of operations written by a human programmer. The programmer decides what the computer is to do and when it should be done, and then writes a list of instructions to be carried out in the proper order.
- The program is entered into the computer, using perhaps a keyboard, and stored in the computer *memory*.
- The CPU can then "fetch" from memory one instruction at a time, in the given order. It will execute the instruction and then fetch the next instruction. With this repeated fetch and-execute cycle, the CPU will accomplish the desired task. A memory block used for program storage has been added.

- A portion of the memory block is labeled *data*. This is the area where the information being processed by the computer is stored. In addition, this is where the CPU stores the results of computations and/or decisions made.
- Since the CPU takes ("reads") data from memory, as well as returns ("writes") data into memory, the memory data bus is bidirectional. By contrast, the program data

bus is not bidirectional, since information on this bus is always from memory to CPU.

- The CPU communicates with the "outside world" by means of the input encoders and the output decoders.
- The ability to multiplex inputs and de multiplex outputs may also be included in the input/output blocks that have been added. This configuration is sometimes quite inefficient, since all information entering or exiting the computer must pass through the CPU.
- The CPU operates at a much faster rate than most external devices, and it must *wait* while data are being entered or exited: A *direct memory access (DMA)* block is generally included to alleviate this problem.
- The DMA allows information to move directly from an input device into memory or from memory directly to an output device.
- While information is being transferred via the DMA, the CPU is free to carry on its computational or logical operations. This greatly improves system efficiency as well as speed of operation.

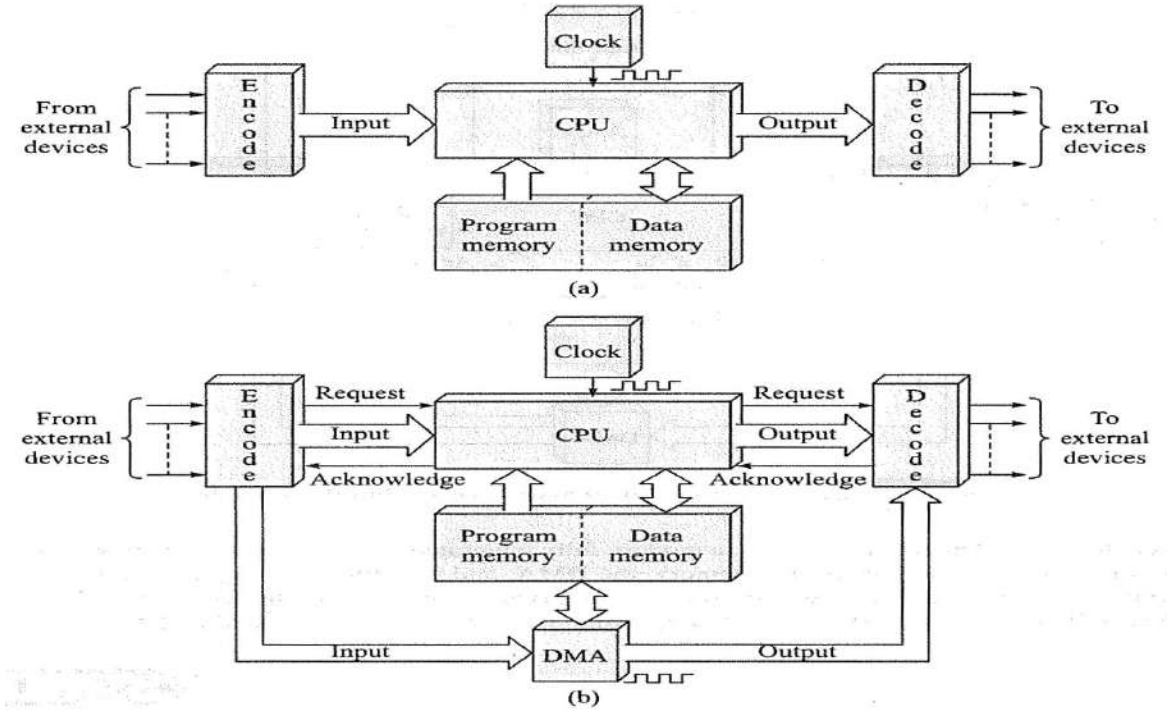


Fig. 1.31 CPU, memory with input/output block

- Before data can be entered into the computer, a signal on the input request line asks the computer for "permission" to input information.
- For instance, depressing the enter key on a keyboard will generate an input request signal. When the CPU is ready, a signal is generated on the acknowledge line, and data will be entered via the DMA into the memory. This request-acknowledge sequence is often called *handshaking*.
- A similar handshaking must occur when the CPU is ready to deliver data to an external device. However, in this case, the CPU makes an output request, and the external device gives permission.

- All of these blocks are operated in synchronism with the clock, but additional direction must be provided.
- The *controller* is the unit that decides which block "goes first" (establishes priorities), decides the order in which external devices are serviced, routes data along the various buses such that no conflicts occur, and controls the overall operation of the system.

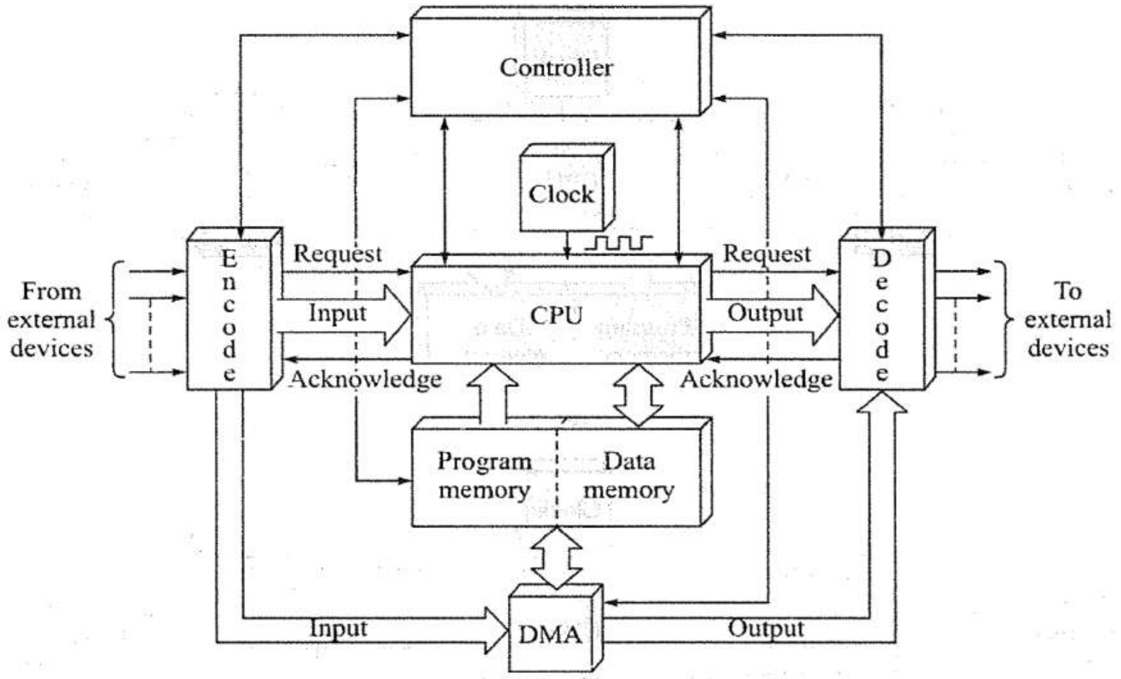


Fig. 1.32 A microprocessor-based digital computer

- *A microprocessor is* often used as the basic IC around which a micro computer or minicomputer is constructed. Numerous computers have been designed, beginning with the 8080 microprocessor. Improvements to this basic IC have led to the development of a family of microprocessor units including the 8085 and the 8086.

DIGITAL INTEGRATED CIRCUITS

- A digital IC is constructed by an interconnection of resistors, transistors, and perhaps capacitors, small all of which have been formed on the surface of a semiconductor wafer. The entire circuit resides on a tiny piece of semiconductor material called a *chip.*
- The semiconductor wafer is typically a slice of mono crystal line about silicon 0.2 mm thick and perhaps 8 to 15 cm in diameter. The wafer is divided checkerboard fashion into 1000 or so rectangular areas.

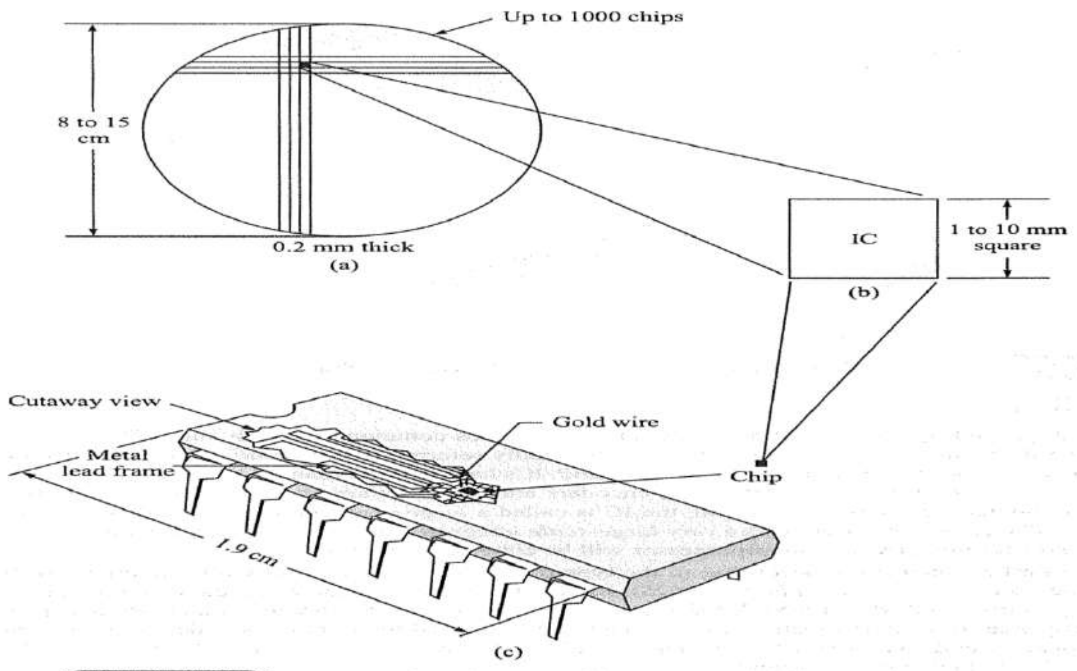


Fig. 1.33 (a) A wafer, (b) One chip, (c) Dual-inline package (DIP)

- Each area will become a single chip. The resistors and transistors necessary for each digital circuit are then formed on each chip by a series of semiconductor processing steps. In this fashion, identical digital circuits are manufactured simultaneously on the same silicon wafer.-

- After the processing steps are completed, the wafer is separated into individual chips. Each chip is a digital circuit, for example, an inverter or an AND gate.
- An individual digital circuit may have only a few components, but some circuits have a few hundred components!
- Each chip is then mounted in a suitable package, as shown in Fig. 1.33c. The package illustrated here is a 14-pin *dual-inline package (DIP).*

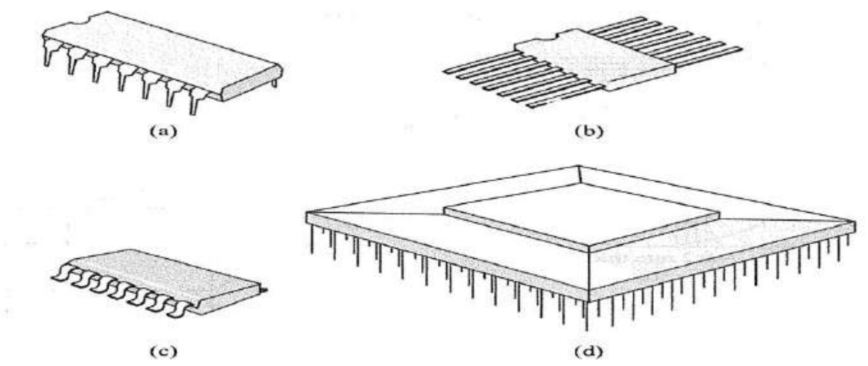


Fig. 1.34 Some IC packages: (a) DIP, (b) Flat pack, (c) Surface mount, (d) Pin-grid array

IC Families

- ICs are categorized by size according to the number of gates contained on each chip. There is no absolute rule, but an IC having fewer than 10 or 12 gates is usually referred to as a *small-scale integration (SSI) IC.*
- For instance, a 7404 has six inverters in a 14-pin DIP. ICs having more than 12 but fewer than 100 gates are called *medium-scale integration (MS!) !Cs;* encoders and decoders are examples of MSI ICs.
- If there are more than 100 gates but fewer than 1000, the IC is called a *large-scale integration (LSI) JC.* An IC having more greater than 1000 is referred to as a *ve,y large-scale integration (VLSI) JC.*
- A large complex system such as semiconductor memory or a microprocessor will be either LSI or VLSI.

- ICs are further categorized according to the type of transistors used. The two basic transistor types are *bipolar* and *metal-oxide-semiconductor* (MOS).
- Bipolar technology is faster but requires more power, and is generally preferred for SSI and MSI. MOS is slower, but requires much less power occupies and also a much smaller chip area for a given function.
- MOS is therefore preferred for LSI and is widely used in applications such as pocket calculators, wristwatches, hearing aids, and so on. For the moment, let's consider the overall characteristics of each digital IC family.

Bipolar Transistors

There are two important digital circuit families constructed bipolar using transistors:

- **Transistor-Transistor Logic** (TTL)

- **Emitter-Coupled Logic** (ECL)

Transistor-Transistor Logic

- TTL was first introduced by Texas Instruments in 1964 using the numbers 54XX and 74XX. These two families are now widely available from a number of different manufacturers.
- The 74XX ICs operate over a temperature range of 0°C to 75°C. The 54XX devices are more rugged; they operate over a temperature range of -55°C to+ I25°C. As you might expect, the 54XX devices are more expensive.

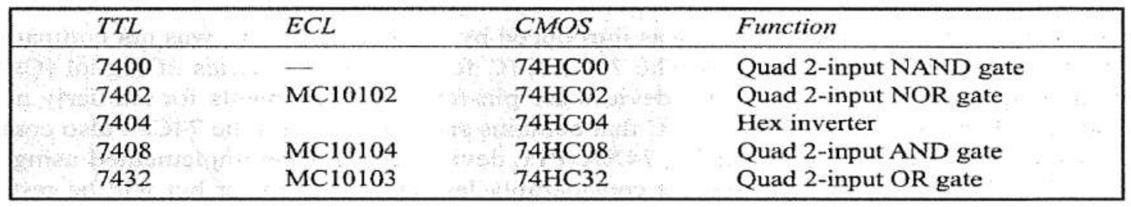


Table 1.2 Some Standard Digital Circuits

TTL	*ECL*	*CMOS*	*Function*
7400	—	74HC00	Quad 2-input NAND gate
7402	MC10102	74HC02	Quad 2-input NOR gate
7404		74HC04	Hex inverter
7408	MC10104	74HC08	Quad 2-input AND gate
7432	MC10103	74HC32	Quad 2-input OR gate

- Otherwise, the logical operations of these two families are the same. In each case, the XX portion of the part number refers to a specific device.
- For instance, "04" stands for inverter, and a 7404 is a TTL inverter. A 7411 is a TTL AND gate, and so on. When there is no danger of confusion, it is common practice to shorten the description by omitting the first two digits.
- In the interest of higher operating speed, the 74XX family was improved with the introduction of the 74HXX (where the H stands for high speed) family of devices.
- The price paid for increased speed was an increase in power required to operate each gate. This led to another family of devices designed to minimize power requirements-the 74LXX (where the L stands for low power) series.
- A major improvement in the TTL series came with the development of a special transistor arrangement called a *Schottky transistor.*
- Using this device, the 74SXX (S for Schottky) family came into being. These devices greatly improved operating speed, but again at the cost of increased power consumption. At this point, a family of devices designated 74LSXX (low-power Schottky) was developed.
- The 74LSXX family offers high-speed operation with minimal power consumption and today is preferred in most designs. The original 7400 also remains popular.
- There are two additional families, 74ASXX (advanced Schottky) and 74ALSXX (advanced low-power Schottky), available.

Emitter-Coupled Logic

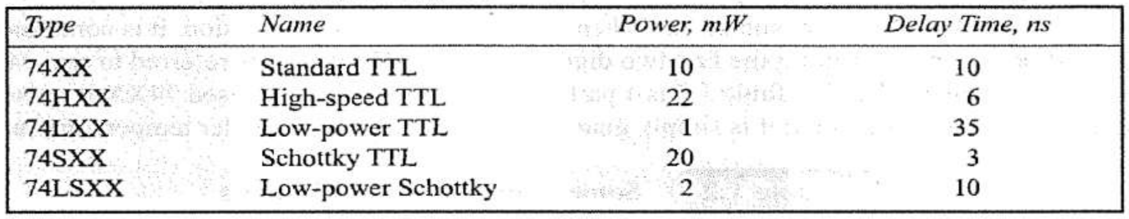
- Emitter-coupled logic (ECL) is considerably faster than any of the TTL families, but the power required for each gate is also much higher.
- With a propagation delay of only 2 ns, the industry standard for ECL circuits is 10,000 ECL, abbreviated 10K. The l00K (100,000) series is even faster, with a delay time of only 1 ns. Motorola markets a family of devices designated MECL 1OK and MECL 1OKH (Motorola Emitter coupled Logic).

MOS Transistors

- Three digital logic families constructed using MOS field-effect transistors (MOSFETs):

- **PMOS** Using p-channel MOSFETs
- **NMOS** Using n-channel MOSFETs
- **CMOS** Using both n-channel and p-channel MOSFETs

Table 1.3 TTL Power-Delay Values



Type	*Name*	*Power, mW*	*Delay Time, ns*
74XX	Standard TTL	10	10
74HXX	High-speed TTL	22	6
74LXX	Low-power TTL	1	35
74SXX	Schottky TTL	20	3
74LSXX	Low-power Schottky	2	10

- PMOS, the slowest and oldest type, is nearly obsolete today.
- NMOS dominates the LSI field and is widely used in semiconductor memories micro processors.
- CMOS is preferred where individual logic circuits are used and where very low power consumption is required.

Digital logic Symbols

- The Institute of Electrical and Electronics Engineers (IEEE) along with the American National Standards Institute (ANSI) have developed a new symbolic language and set of symbols to be used with digital logic circuits.
- These new symbols are now being used on manufacturers' data sheets along with traditional symbols. The most recent revision of *IEEE Standard Graphic Symbols for Logic Functions*, ANSI/IEEE Std 91-1984, provides for two different types of symbols.
- Symbols of the first type, called *distinctive-shape symbols,* are exactly as have been shown throughout this chapter.

- The second system, which is called the *rectangular-shape system,* uses a rectangular box with a special symbol for each type of gate. The IEEE standard does not express a preference for either shape.
- A rectangular box is used for the gate, the input is labeled *A*, and the output is labeled *Y*.
- The triangle on the output line signifies that the output is active when low. Thus, when the input is active (high), the output will be active (low).

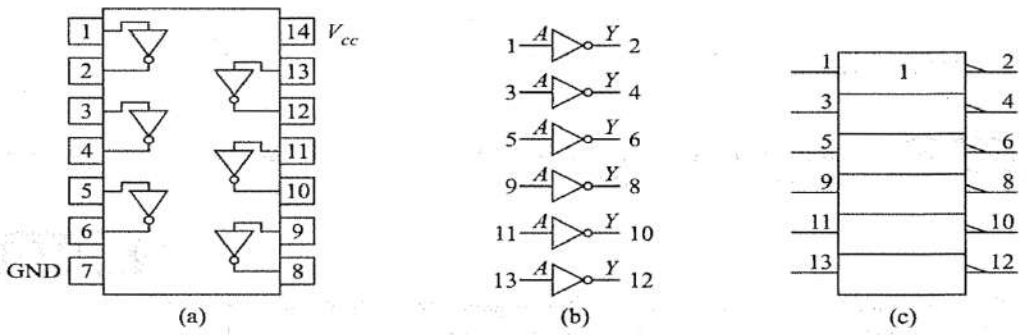
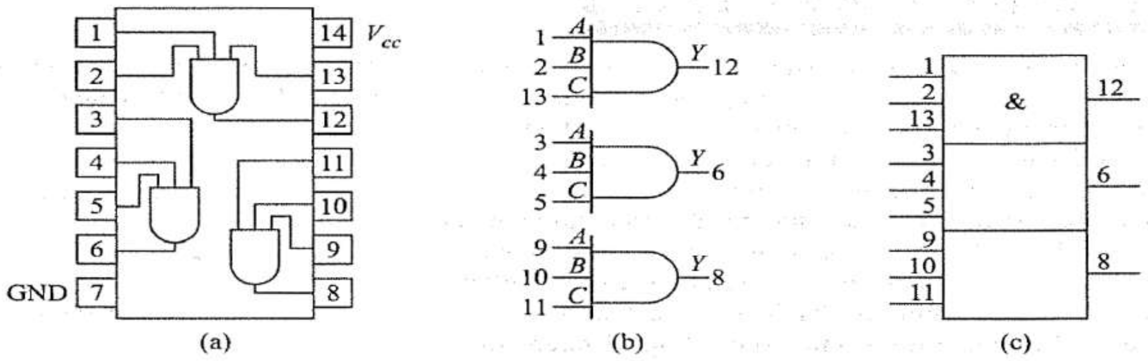


Fig. 1.36 **Hex inverter, 7404: (a) Pin configuration, (b) Logic symbol, (c) Logic symbol (IEEE)**

- The 7404 is a hex inverter; that is, it is an IC that contains six inverters. The DIP package for this device is shown in Fig. 1.36a, along with the proper pin numbers on the package. Figure 1.36b shows the six standard logic symbols for the inverters.



Triple three-input AND gate, 7 411: (a) Pin configuration, (b) Logic symbol,

(c) Logic symbol (IEEE)

- The pin out and symbols for the 7411 quad 2-input. The new IEEE symbol for the AND gate is a rectangle with the ampersand (&) symbol written in it; is used in Fig. 1.37c to show the three AND gates in the 7411.

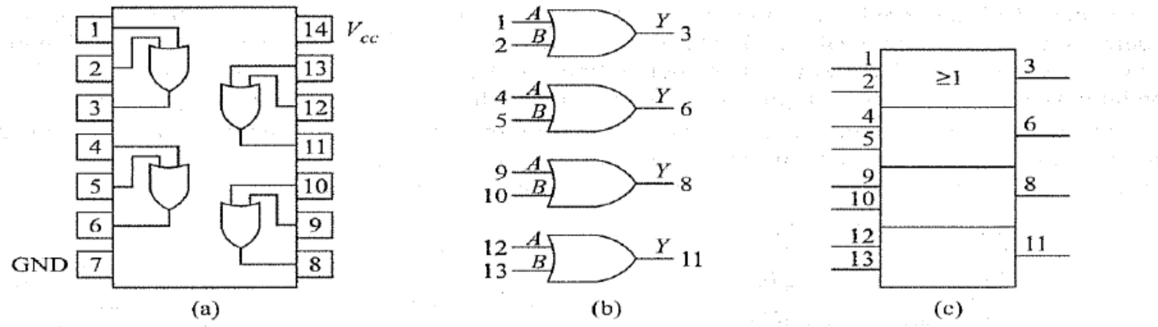


Fig. 1.38 Quad two-input OR gate, 7432: (a) Pin configuration, (b) Logic symbol, (c) Logic symbol (IEEE)

DIGITAL LOGIC

- A digital circuit having one or more input signals but only one output signal is called a *gate* .the most basic gates-the NOT gate (inverter), the OR gate and the AND gate-were introduced.
- Connecting the basic gates in different ways makes it possible to produce circuits that perform arithmetic and other functions associated with the human brain (an ALU). Because they simulate mental processes, gates are often called *logic circuits.*

A discussion of both positive and negative logic leads to the important concept of *assertion-/eve/ logic.*

- Hardware description languages (HDL) are an alternative way of describing logic circuits. This uses a set of textual codes that is machine (computer) readable.

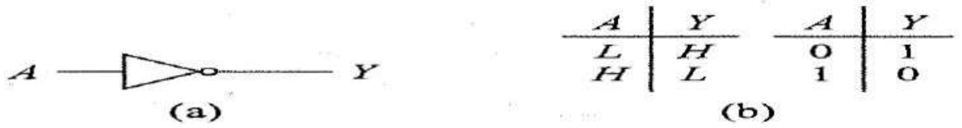
THE BASIC GATES-NOT, OR, AND

- Three logic circuits: the *inverter,* the *OR gate,* and the *AND gate,* can be used to produce any digital system.

The Inverter (NOT Gate)

- In one truth table, the symbols Hand *L* are used, while the binary numbers O and 1 are used .in the other. The information in each table is identical, however, since we know *L* = 0 and *H* = 1.

- In this text, both symbols are used, hence since there is no chance for confusion. You will find both symbols used in other texts, as well as in manufacturers' data sheets.



A	Y
L	H
H	L

A	Y
0	1
1	0

Fig. 2.1 (a) Inverter symbol, (b) Truth tables

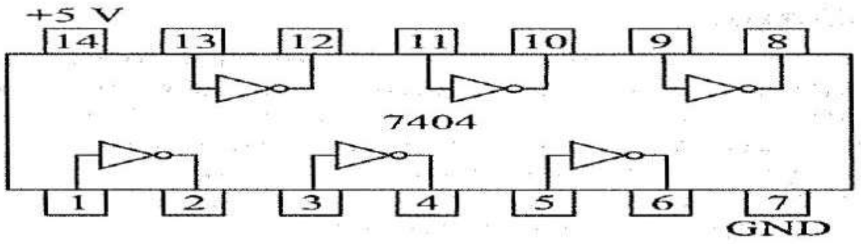
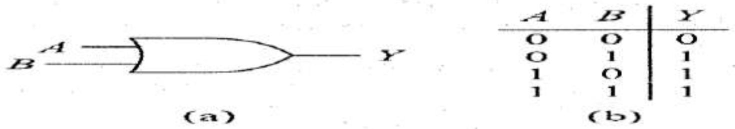


Fig. 2.2 Pinout diagram of a 7404

- The important idea is that there are only two possible voltage levels (low and high) associated with a digital circuit. This fits nicely with the binary number system, since it has only two values (0 and 1). This is often referred to as *two-state operation.*
- By definition, this is *positive logic,* since the *higher* voltage level is assigned binary 1. In n*egative logic,* where the *higher* voltage level is assigned binary (zero).

OR Gates

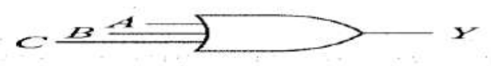
- An OR gate has two or more input signals but only one output signal. It is called an OR gate because the output voltage is high if any or all of the input voltages are high. For instance, the output of a 2-input OR gate is high if either or both inputs are high.



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 2.4 (a) OR gate, (b) Truth table

- For example, the first *ABC* entry is 000, the next is 001, then 010, and so on, up to the final entry of 111. Since all binary numbers are present, all input possibilities are included.



(a)

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(b)

Fig. 2.5 (a) Three-input OR gate, (b) Truth table

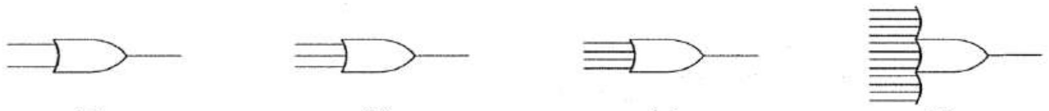


Fig. 2.6 OR gate symbols: (a) Two-input, (b) Three-input, (c) Four input, (d) Twelve-input

- Incidentally, the number of rows in a truth table equals 2", where *n* is the number of inputs. For a 2-input OR gate, the truth table has 22, or 4 rows. A 3-input OR gate has a truth table with 2^3, or 8 rows, while a 4-input OR gate results in 24, or 16 rows, and so on.

- **An OR gate can have** as many inputs as desired. No matter how many inputs, the action of any OR gate is summarized like this: One or more high inputs produce a high output.

logic Symbols

- Figure 2.6a shows the symbol for a 2-input OR gate of any design. Whenever you see this symbol, remember the output is high if either input is high.
- The logic symbol for a 3-input OR gate. Figure 2.6c is the symbol for a 4-input OR gate. For these gates, the output is high when any input is high. The only way to get output allow is by having all inputs low.
- When there are many input signals, it's common drafting practice to extend the input side as needed to allow sufficient space between the input lines.
- It is the symbol for a 12-input OR gate. The same idea applies to any type of gate; extend the input side when necessary to accommodate a large number of input signals.

- **TTL OR Gates** Figure 2.7 shows the pinout diagram of a 7432, a TTL quad 2-input OR gate. This digital IC contains four 2-input OR gates inside a 14-pin DIP. After connecting a supply voltage of +5 V to pin 14 to and a ground pin 7, you can connect one or more of the OR gates to other TTL devices

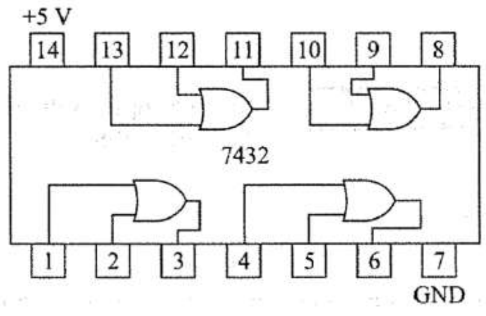


Fig. 2.7 Pinout diagram of a 7432

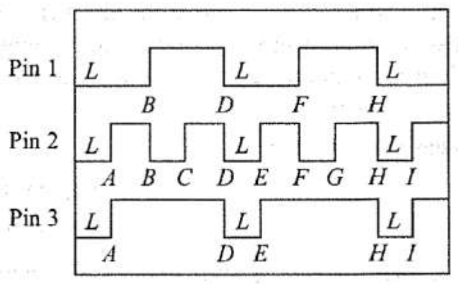


Fig. 2.8 Timing diagram

- **Timing Diagram Fi**gure 2.8 shows an example of a timing diagram for a 2-input OR gate. The input voltages drive pins 1 and 2 of7432a. Notice that the output (pin 3) is low only when both inputs are low. The output is high the rest of the time because one or more input pins are high.

AND Gates

- The AND gate has a high output only when all inputs are high. Figure 2.1a shows a 2-input AND gate. The truth table (Fig. 2.10 b) summarizes all input-output possibilities for a 2-input AND gate.
- Examine this table carefully and remember the following: the AND gate has a high output only when *A* and *B* are high. In other words, the AND gate is an all-or-nothing gate; a high output occurs only when all inputs are high. This truth table uses 1s and 0s, where 1 = Hand 0 = *L*.

- In Boolean equation form =*A* AND *B*, i.e. $Y=A.B$ or $Y=AB$

$Y = 0.0 = 0$, $Y = 0.1 = 0$, $Y = 1.0 = 0$and $Y = 1.1 = 1$

- The '.' sign here represents logic AND operation and not multiplication operation of basic arithmetic though the result for are same both.
- **TTL AND Gates** Figure 2.13 shows the pinout diagram of a 7408, a TTL quad 2-input AND gate. This digital IC contains four 2-input AND gates.
- After connecting a supply voltage of +5V to pin 14 and a ground to pin 7, you can connect one or more of the AND gates to other TTL devices.
- TTL AND gates are also available in triple 3-input and dual 4-input packages.

- **Timing Diagram** Figure 2.14 shows an example of a timing diagram for a 2-input AND gate. The input voltages drive pins 1 and 2 of a 7408. Notice that the output (pin 3) is high only when both inputs are high (between C and *D*, *G* and *H*, etc.). The output is low the rest of the time.

UNIVERSAL LOGIC GATES-NOR, NAND

- Gates, which can perform this task, are called universal logic gates. Clearly, basic gates like AND, OR and NOT don't fit into this category for the simple reason that conversion among themselves itself are not possible. As for example, one cannot gate OR operation by using number any or combination of AND gates.

- Two universal logic gates NOR and NAND.

NOR Gates

- The logic circuit used to be called a NOT-OR gate because the output is

$Y=A+B$

- Read this as *"Y* equals NOT *A* OR B" or *"Y* equals the complement of *A OR B."* Because the circuit is an OR gate followed by an inverter, the only way to get a high output is to have both inputs low, as shown in the truth table of Table 2.1.

NOR Gate Symbol

- The bubble (small circle) on the output is a reminder of the inversion that takes place after the ORing. Furthermore, the words NOT-OR are contracted to the word NOR.

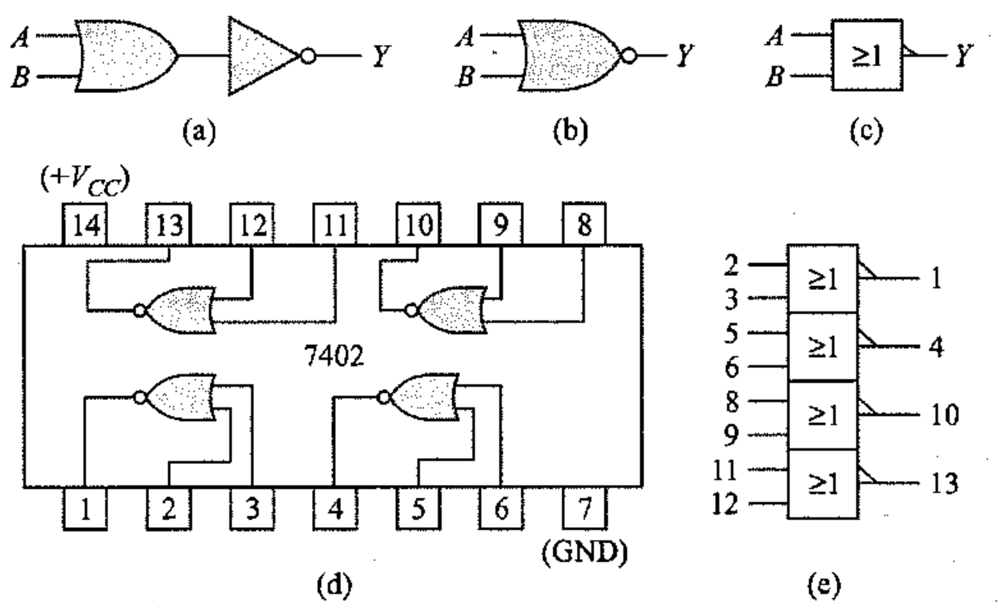


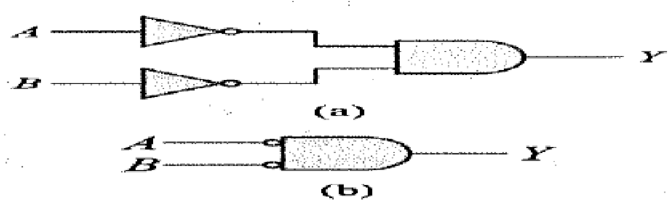
Fig. 2.19 NOR logic gate

NOR logic gate

- The new IEEE rectangular symbol for the NOR gate. The small triangle on the output is equivalent to the bubble used on the standard symbol. The indicator ≥ inside the box means "if one or more of the inputs are high, the output is high."

Table 2.1 NOR Gate

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0



:0 (a) AND gate with inverted inputs, (b) Equivalent symbol

Bubbled AND Gate

- The bubbles on the inputs are a reminder of the inversion that takes place before the AND operation. It is *bubbled AND gate.* We find that output *Y* and inputs *A*, *B* are identical for bubbled AND gate and NOR gate.
- Therefore, these two circuits and are equivalent thus interchangeable. Given any logic circuit with NOR gates, we can replace it by bubbled AND gates and converse is also true.

De Morgan'sFirst Theorem

The Boolean equation for Fig. 2.19b is *Y=A+B*

The Boolean equation for Fig. 2.20b is *Y=AB*

- The first equation describes a NOR gate, and the second equation a bubbled AND gate. Since the outputs same inputs, we can equate the right~hand members to get.

$$\overline{A+B} = \overline{A}\,\overline{B}$$

- This identity is known as De Morgan'sFirst Theorem. In words, it says the complement of a sum equals the product of the complements. This can also be proved by comparing the truth tables shown in Fig. 2.4(b) and NOR gate truth table of Table 2.1. A similar exercise that compares truth tables of three input NOR gate and three input bubbled AND gate show they are identical and we can write, $(A+ B + C)' = A'B'C'$. Note that this equivalence can be extended to gates or circuits for larger number of inputs, too.

Universality of NOR Gate

- Figure 2.21 shows how all other logic gates can be obtained from NOR gates. To get a NOT gate we tie inputs of NOR gate together (Fig. 2.21 a) so that there is only one input to the circuit. If input is 0, then both

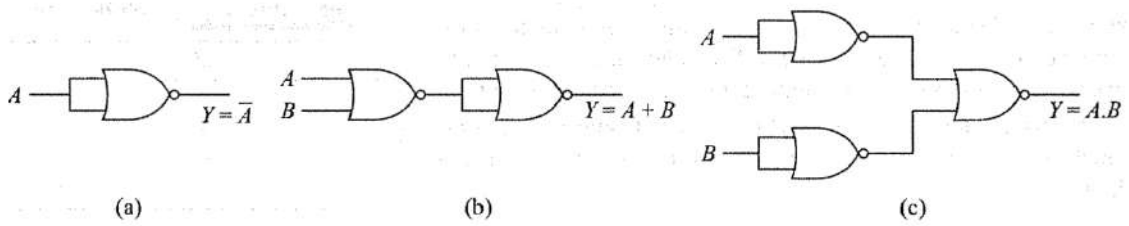


Fig. 2.21 Universality of NOR gate (a) NOT from NOR, (b) OR from NOR, (c) AND from NOR

- The inputs to NOR gate are 0. Following NOR gate truth table (Table 2see.1) we output now is l. Similarly, if input is 1, both the inputs to NOR gate are 1 that gives output 0. Therefore output of circuit, shown in Fig. 2.2 la is complement of its input and thus gives NOT operation.

Eye of the Beholder

- Truth tables, logic circuits, and Boolean equations are different ways of looking at the same thing. Whatever we learn from one viewpoint applies to the other two. If we prove that truth tables are identical, this immediately tells us the corresponding logic circuits are interchangeable, and their Boolean equations are equivalent.

NAND Gates

- Originally, the logic circuit was called NOT-AND gate because the output is $Y=AB$

- Read this as "Y equals NOT *A* AND *B"* or "Y equals the complement of *A* AND *B."* Because the circuit is an AND gate followed the by an inverter, only way to get a low output is for both inputs to be high, as shown in the truth table of Table.

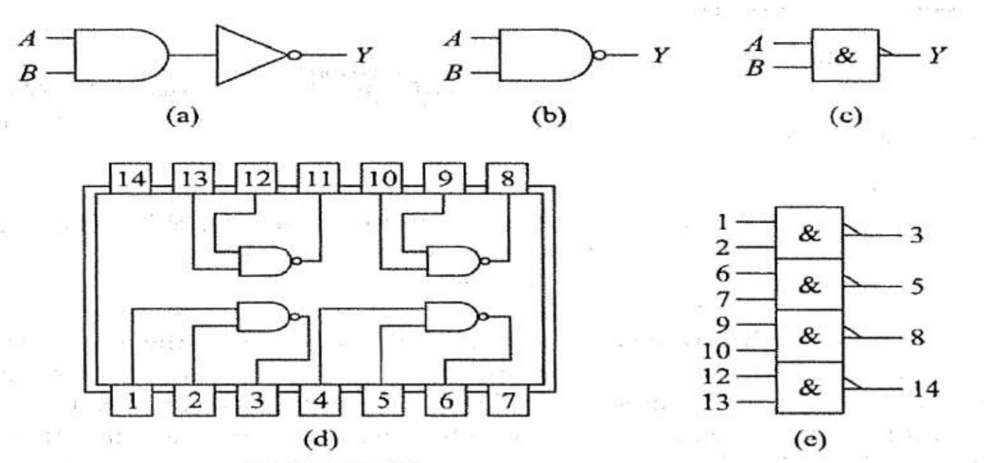


Fig. 2.25 NAND logic gate

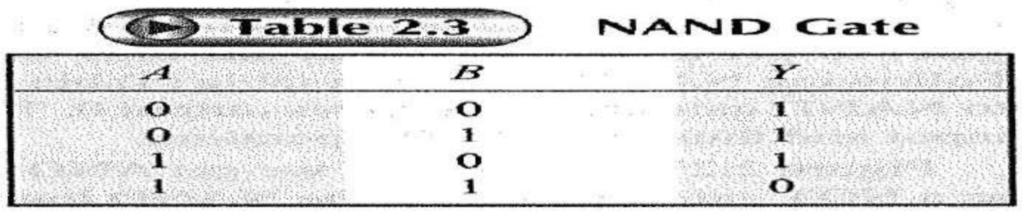


Table 2.3 NAND Gate

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NAND-Gate Symbol

- The logic circuit has become so popular that the abbreviated symbol is used for it. The bubble on the output reminds us of the inversion after the ANDing.
- Also, the words NOT-AND are contracted to NAND. Whenever you see this symbol, remember that the output is NOT the AND of the inputs. With a NAND gate, all inputs must be high to get a low output. If any input is low, the output is high.
- The new IEEE rectangular symbol for the NAND gate. The small triangle on the output is equivalent to the bubble used on the standard symbol. The indicator"&" inside the box means "the output is high only when *all* inputs are high."

Bubbled OR Gate

- Inverters on the input lines of an OR gate. The circuit is often drawn in the abbreviated form, where the bubbles represent inversion. We will refer to the abbreviated drawing of as a *bubbled OR gate.*

De Morgan'sSecond Theorem

The Boolean equation for *Y=AB*

The Boolean equation for *Y=X +B*

- The first equation describes a NAND gate, and the second equation a bubbled OR gate. Since the outputs are equal for the same inputs, we can equate the right-hand members to get

$$B = A + B \qquad 2.2)$$

- This identity is known as *De Morgans second theorem.* It says the complement of a product equals the sum of the complements. This can also be proved by comparing the truth tables shown in Fig. 2.3(b) and NAND gate truth table of Table 2.2.
- A similar exercise that compares truth tables of three input NAND gate and three input bubbled OR gate show they are identical and we can write, *(A.B.C)' =A'+ B'* + C'. Note that this equivalence can be extended to gates or circuits with any number of inputs.

Universality of NANO Gate

- How all other logic gates can be obtained from NAND gates and why it is called a universal logic gate. Figure 2.27a shows how we tie inputs of NAND gate together (as we had done in case of NOR gate) to get a NOT gate that has only one input.
- If input is 0, then both the inputs to NAND gate are. 0. Following NAND gate trnth table (Table 2.3) we see output now isl. Similarly, if input is **1,** both the inputs to NAND gate are 1 that gives output 0. Therefore output of circuit, shown in Fig. 2.27a is complement of its input and thus gives NOT operation.

2.3 AND-OR-INVERT GATES

Figure 2.31a shows an AND-OR circuit. Figure 2.31b shows the De Morgan equivalent circuit, a NAND-NAND network. In either case, the Boolean equation is *Y=AB+C*

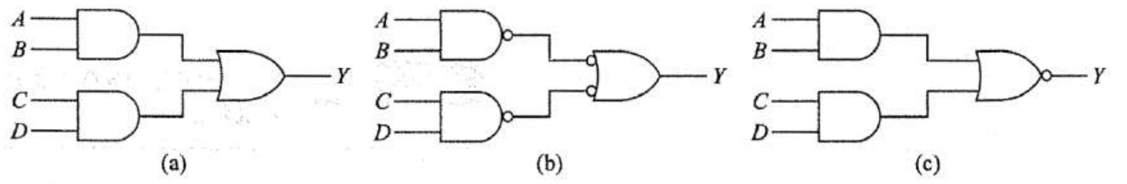


Fig. 2.31 (a) AND-OR circuit, (b) NAND-NAND circuit, (c) AND-OR-INVERT circuit

- Since NAND gates are the preferred TTL gates, we would build the circuit of Fig. 2.31 b. As you know, NAND-NAND circuits like this are important because with them you can build any desired logic circuit.

TTL Devices

- AND-OR circuits are not easily derived from the basic NAND-gate design. But it is easy to get an AND-OR-INVERT circuit as in Fig. 2.3 lc. A variety of circuits like this are available as TTL chips. Because of the inversion, the output has the equation shown below.

$$Y = \overline{AB + CD}$$

- Table 2.6 lists the AND-OR-INVERT gates available in the 7400 series. In this table, *2-wide* means two AND gates across, *4-wide* means four AND gates across, and so on.
- For instance, the 7454 is a 2-input 4-wide AND-OR-INVERT gate ; each AND gate has two inputs (2-input), and there are four AND gates (4-wide). Figure 2.32b shows the 7464; it is a 2-2-3-4-input 4-wide AND-OR-INVERT gate.

Table 2.6 AND-OR-INVERT Gates

Device	Description
7451	Dual 2-input 2-wide
7454	2-input 4-wide
7459	Dual 2-3-input 2-wide
7464	2-2-3-4-input 4-wide

- Connecting the output of a 2-input 2-wide AND-OR-INVERT gate to an inverter will give us the same output as an AND-OR circuit.

Expandable AND-OR-INVERT Gates

- The widest AND-OR-INVERT gate available in the 7400 series is 4-wide. What do we do when we need a 6- or 8-wide circuit? One solution is to use an *expandable* AND-OR-INVERT gate.

- Figure 2.33 shows the logic symbol for an expandable AND-OR-INVERT gate. There are two additional inputs, labeled *bubble* and *arrow*. Table 2.7 lists the expandable AND-OR-INVERT gates in the 7400 series.

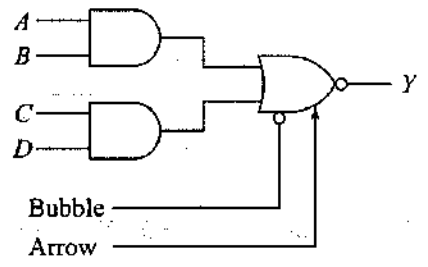


Fig. 2.33 Expandable AND-OR-INVERT gate

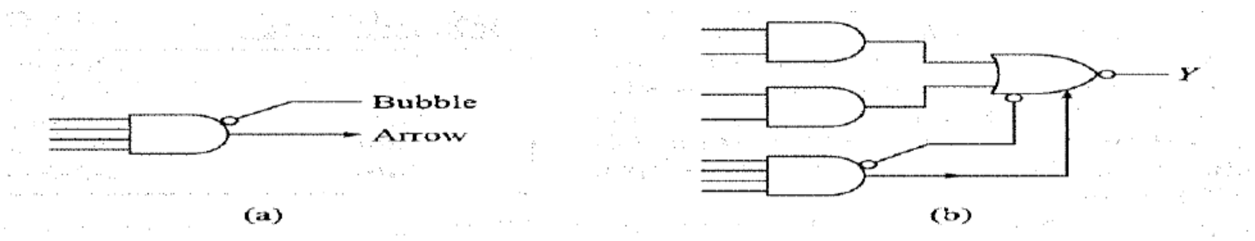
Table 2.7 Expandable AND-OR-INVERT Gates

Device	Description
7450	Dual 2-input 2-wide
7453	2-input 4-wide
7455	4-input 2-wide

Expanders

- Connect bubble to bubble and arrow to arrow.
- Visualize the outputs of Fig. 2.34a connected to the arrow and bubble inputs of Fig. 2.33. Figure 2.34b shows the logic circuit. This means that the expander outputs are being ORed with the

signals of the AND-OR-INVERT gate. In other words, Fig. 2.34b is equivalent to the AND-OR-INVERT circuit of Fig. 2.34c.



- We can connect Figure more expanders. It shows two expanders driving the expandable gate. Now we have a 2-2-4-4-input 4-wide AND-OR-INVERT circuit.
- The 7460 is a dual 4-input expander. The 7450, a dual expandable AND-OR-INVERT gate, is designed for use with up to four 7460 expanders. This means that we can add two more expanders in Fig. 2-34d to get a 2-2-4-4-4-4-input 6-wide AND-OR-INVERT circuit.

POSITIVE AND NEGATIVE LOGIC

- Up to now, we have used a binary O for low voltage and a binary 1 for high voltage. This is called *positive logic*.
- People are comfortable with positive logic because it feels right. But there is another code known as *negative logic* where binary O stands for high voltage and binary 1 for low voltage. Even though it seems unnatural, negative logic has many uses.

Positive and Negative Gates

- An OR gate in a positive logic an system becomes AND gate in a negative logic system.. That is, if either input is high in Fig. 2.35, the output is high.

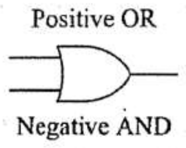


Fig. 2.35 Meaning of symbol depends on whether you use positive or negative logic

Table 2.8

A	B	Y
Low	Low	Low
Low	High	High
High	Low	High
High	High	High

- In a positive logic system, binary O stands for low and binary 1 for high. So, we can convert Table 2.8 to Table 2.9. Note that *Y* is a 1 if either *A* or *B* is 1. This sounds like an OR gate.
- And it is, because we are using positive logic. To avoid ambiguity, a positive OR gate because it performs the OR function with positive logic. (Some data sheets describe gates as positive OR gate, positive AND gate, etc.)

- In a negative logic system, binary 1 stands for low and binary O for high. With this code. Now, watch what happens. The output *Y* is a 1 only when both *A* and *B* are 1.
- This sounds like an AND gate! And it is, because we are now using negative logic. In other words, gates are defined by the way they process the binary Os and 1s. If you use binary 1 for low voltage and binary O for high voltage, then you liave to refer to Fig. 2.35 as a negative AND gate.

Table 2.9

A	*B*	*Y*
0	0	0
0	1	1
1	0	1
1	1	1

Table 2.10

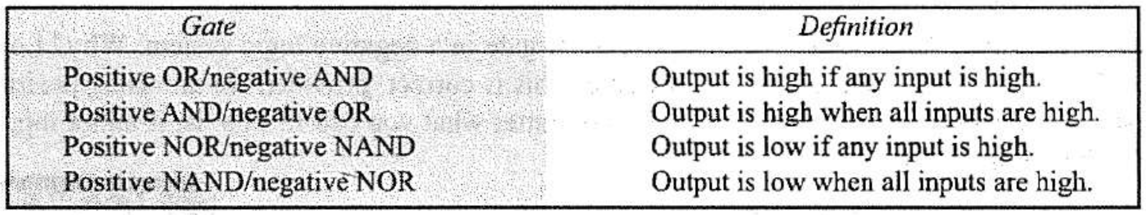
A	*B*	*Y*
1	1	1
1	0	0
0	1	0
0	0	0

- In a similar way, we can show the truth table of other gates with positive or negative logic. By analyzing the inputs and outputs in terms of Os and ls, you find these equivalences between the positive and negative logic:

Positive OR	↔ negative AND
Positive AND	↔ negative OR
Positive NOR	↔ negative NAND
Positive NAND	↔ negative NOR

- These definitions are always valid. If you get confused from time to time, refer to Table 2.11 to get back to the ultimate meaning of the basic gates.

Table 2.11 Voltage Definitions of Basic Gates



Gate	Definition
Positive OR/negative AND	Output is high if any input is high.
Positive AND/negative OR	Output is high when all inputs are high.
Positive NOR/negative NAND	Output is low if any input is high.
Positive NAND/negative NOR	Output is low when all inputs are high.

Assertion-level logic

- Why do we even bother with negative logic? The reason is related to the concept of *active-low signals.* For instance, the 74150 multiplexer has an active-low input strobe; this input turns on the chip only when it is low (negative true).
- This is an active-low signal; it causes something to happen when it is low, rather than high. As another example, the 74154 decoder has 16 output lines; the decoded output signal is low (negative true).
- In other words, all output lines have a high voltage, except the decoded output line. Besides TTL devices, microprocessor chips like the 8085 have a lot of active-low input and output signals.
- Many designers draw their logic circuits with bubbles on all pins with active-low signals and omit bubbles on all pins with active-high signals. This use of bubbles with active-low signals is called *assertion-level logic* that.
- It means you draw chips with the kind of input that causes something to happen, or with the kind of output that indicates something has happened. If a low input signal turns on a chip, you show a bubble on that input.
- If a low output is a sign of chip action, you draw a bubble on that output. Once you get used to assertion-level you may prefer drawing logic circuits this way.
- Sometimes you hear expressions such as "The inputs are asserted" or "What happens when the inputs are asserted?" An input is *asserted* when it is active. This means it may be low or high, depending on whether it is an active-low or active-high input.
- For instance, given a positive AND gate, all inputs must be asserted (high) to get a high output. As another example, the STROBE input of a TTL multiplexer must be asserted (low) to turn on

the multiplexer. In short, you can equate the word *assert* with *activate*. You assert, or activate, the inputs of a gate or device to get something to happen.

UNIT -II

COMBINATIONAL LOGIC CIRCUIT

- This chapter discusses Boolean algebra and several simplification techniques. After learning the laws and theorems of Boolean algebra, you can rearrange Boolean equations to arrive at simpler logic circuits.
- An alternative method of simplification is based on the Kanaugh map. In this approach, geometric rather than algebraic techniques are used to simplify logic circuits.
- Quine-McClusky tabular method provides a more systematic reduction technique, which is preferred when a large number of variables are in consideration.

  There are two fundamental approaches in logic design:

1. The sum-of-products method
2. product-of sum method

- The sum-of-products solution results in an AND-OR or NAND-NAND network,
- while the product-of-sums solution results in an OR-AND or NOR-NOR network.

BOOLEAN LAWS & THEOREMS

Basic laws

The commutative laws are

$$A + B = B + A \qquad (3.1)$$
$$AB = BA \qquad (3.2)$$

- These two equations indicate that the order of a logical operation is unimportant because the same answer is arrived at either way. As far as logic circuits are concerned. Figure 3.la shows how to visualize Eq. (,.1).
- All it amounts to is realizing that the inputs to an OR gate can be transposed without changing the output. Likewise, Fig. 3.lb is a graphical equivalent for Eq. (3.2).

The associative laws are

$$A + (B + C) = (A + B) + C \qquad (3.3)$$
$$A(BC) = (AB)C \qquad (3.4)$$

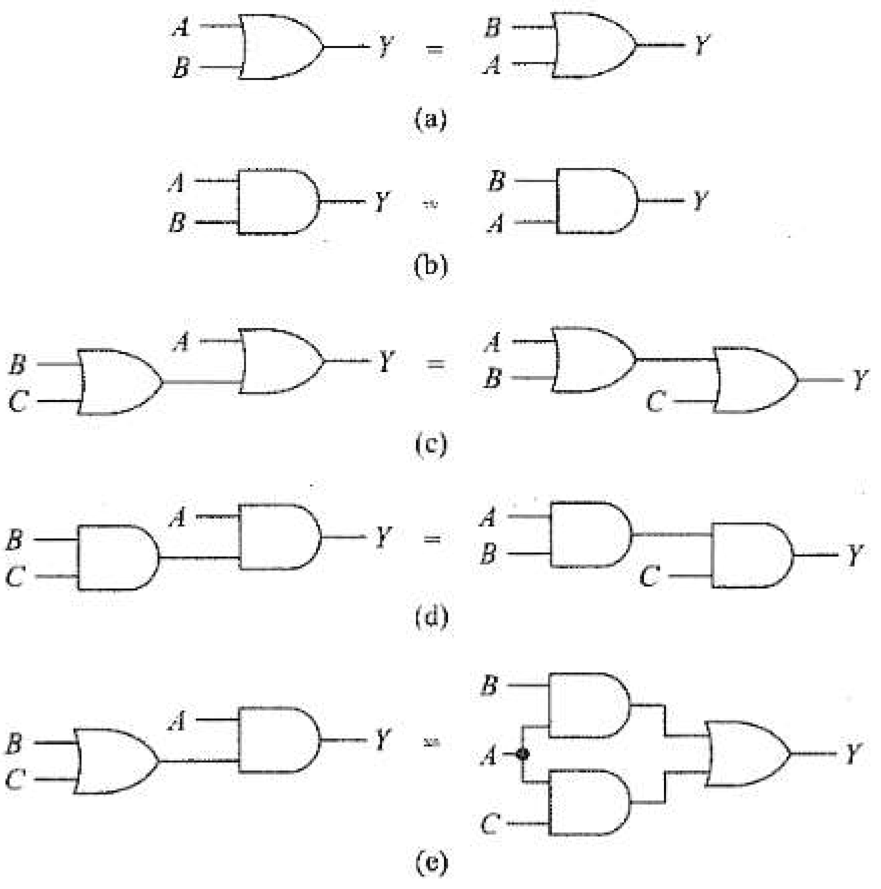


Fig. 3.1 **Commutative, associative, and distributive laws**

These laws show that the order of combining variables has no effect on the final answer. In terms of logic circuits , Eq Fig..3.1c (3.3) while 3.1d represents illustrates Eq. (3.4).

The distribute law is

$$A(B + C) = AB + AC$$

- ➢ This law is easy to remember because it is identical to ordinary algebra. Figure 3 .1 e shows the corresponding logic equivalence. The distributive law gives you a hint about the value of Boolean algebra.
- ➢ If you can rearrange a Boolean expression, the corresponding logic circuit may be simpler.
- The first five laws present no difficulties because they are identical to ordinary algebra. You can use these laws to simplify complicated and Boolean expressions arrive at simpler logic circuits.
- But before you begin, you have to learn other Boolean laws and theorems.

OR Operations

The next four Boolean relations are about OR operations. Here is the first:

$$A+O=A$$

This says that a variable ORed with O equals the variable. If you think about it, makes perfect sense. When *A* is 0,

$$O+O=O$$

And when *A* is 1,

$$1+0=1$$

In either Eq. case, (3.6) is true.

Another Boolean relation is

$$A+A=A$$

Again, you can see right through this by substituting the two possible values of *A*. First when *A* = 0, Eq.(3.7) gives

$$O+O=O$$

which is true. Next, *A* = I results in

$$1+1=1$$

which is also true because 1 ORed with 1 produces 1. Therefore, any variable ORed with itself equals the variable.

Another Boolean rule worth knowing is

$$A+I=1$$

Why is this valid? When *A*= 0, Eq. (3.8) gives

$$0+1=1$$

which is true. Also. *A* = 1 gives

$$1+1=1$$

This is correct because the plus sign implies OR addition, not ordinary addition. In summary,

The next four Boolean relations are about OR operations. Here is the first:

$$A+O=A$$

This says that a variable ORed with O equals the variable. If you think about it, makes perfect sense. When A is 0,

$$O+O=O$$

And when A is 1,

$$1+0=1$$

In eitherEq. case, (3.6) is true.

Another Boolean relation is

$$A+A=A$$

Again, you can see right through this by substituting the two possible values of A. First when $A = 0$, Eq.

(3.7) gives $O+O=O$

which is true. Next, A = I results in

$$1+1=1$$

The next four Boolean relations are about OR operations. Here is the first:

$$A+O=A$$

This says that a variable ORed with O equals the variable. If you think about it, makes perfect sense. When A is 0,

$$O+O=O$$

And when A is 1,

$$1+0=1$$

In eitherEq. case, (3.6) is true.

Another Boolean relation is

$$A+A=A$$

Again, you can see right through this by substituting the two possible values of *A*. First when *A* = 0, Eq.

(3.7) gives

O+O =O

which is true. Next, *A* = I results in

$$1 + 1 = 1$$

This is correct because the plus sign implies OR addition, not ordinary addition. In summary,

Eq. (3.8) says this,input if oneto an OR gate is high, the output is high no matter what the other input.

Finally, we have

$$A+ A= I$$

You should see this in a flash. If *A* is 0, *A* is 1 and the equation is true. Conversely if *A* is 1, *A* is O and The equation agree still.In short , a variable Red with its complement always equals 1.

AND Operations

Here are three AND relations

$$A\cdot 1 =A$$

$$A \cdot A =A$$

$$A \cdot O =O$$

When *A* is 0, all the foregoing are true. Likewise, when *A* is 1, each is true.

Therefore, the three equations

are valid and can be used to simplify Boolean equations.

One more AND formula is

$$A\cdot A \ = O$$

This one easy to understand because you get either

$$0 \cdot l \ = 0$$

or

$$I\cdot 0 = 0$$

for the two possible values of *A.* **In** words, Eq. (3.13) indicates that a variable ANDed with its complement always equals zero.

Double Inversion and De Morgan's Theorems

The *double-inversion nde* is

$$A \ = A$$

which shows that the double complement of a variable equals the variable. Finally, there are the De Morgan theorems discussed in Chapter 2:

$$A+B = AB \tag{3.15}$$

$$AB = A + B \tag{3.16}$$

You already know how important these are. The first says a NOR gate and a bubbled AND gate are equivalent. The second says a NAND gate and a bubbled OR gate are equivalent.

Duality Theorem

The *duality theorem* is one of those elegant theorems proved in advanced mathematics. We will state the theorem without proof. Here is what the duality theorem says. Starting with a Boolean relation, you can derive another Boolean relation by

- Changing each OR sign to an AND sign
- Changing each AND sign to an OR sign
- Complementing any O or 1 appearing in the expression

For instance, Eq. (3.6) says that

$$A+O=A$$

The dual relation is

$$A\cdot 1 = A$$

This dual property is obtained by changing the OR sign to an AND sign, and by complementing the O to get a 1.

The duality theorem is useful because it sometimes produces a new Boolean relation. For example, Eq. (3 .5) states that

$$A(B+C) = AB+AC$$

By changing each OR and AND operation, we get relation the dual

$$A + BC = (A + B)(A + C)$$

17)

Covering and Combination

- The covering rule, where one term covers the condition of the other term so that the other term becomes redundant, can be represented in dual form as

$$A + AB = A \qquad (3.18)$$
$$A(A + B) = A \qquad (3.19)$$

$$A + AB = A \qquad (3.18)$$
$$A(A + B) = A \qquad (3.19)$$

$$A + AB = A \qquad (3.18)$$

$$A(A + B) = A \qquad (3.19)$$

And

The above can be easily proved from basic laws because,

$$AB + AB = A$$

$$(A + B)(A + B) = A$$

Eq. (3.20) can easily be proved $B + B = 1$

Expanding left hand side of Eq. (3.21)

$$\begin{aligned} A \cdot A + A \cdot B + A \cdot \overline{B} + B \cdot \overline{B} &= A + A(B + \overline{B}) + 0 \\ &= A + A \cdot 1 = A + A = A = \text{right hand side} \end{aligned}$$

Table 3.1 Fundamental Products for Two Inputs

A	*B*	*Fundamental Product*
0	0	$\overline{A}\,\overline{B}$
0	1	$\overline{A}B$
1	0	$A\overline{B}$
1	1	AB

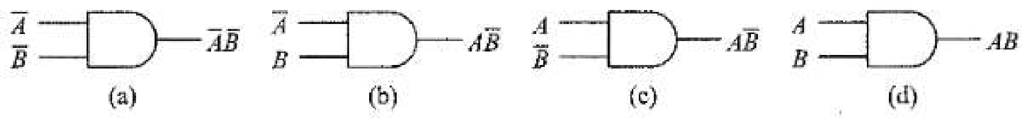


Fig. 3.3 ANDing two variables and their complements

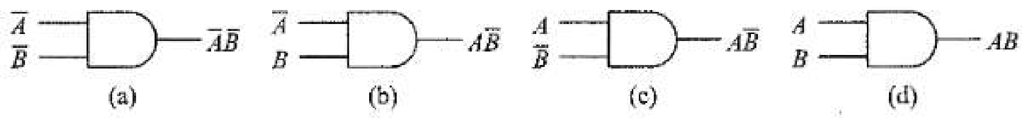


Fig. 3.3 ANDing two variables and their complements

$$\overline{A}\overline{B}\overline{C}, \overline{A}\overline{B}C, \overline{A}B\overline{C}, \overline{A}BC, A\overline{B}\overline{C}, A\overline{B}C, AB\overline{C}, ABC$$

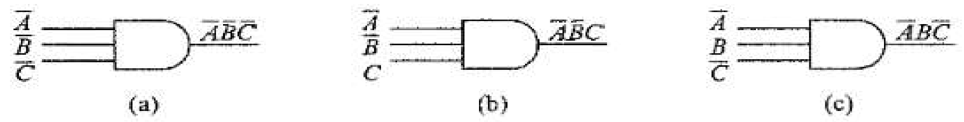


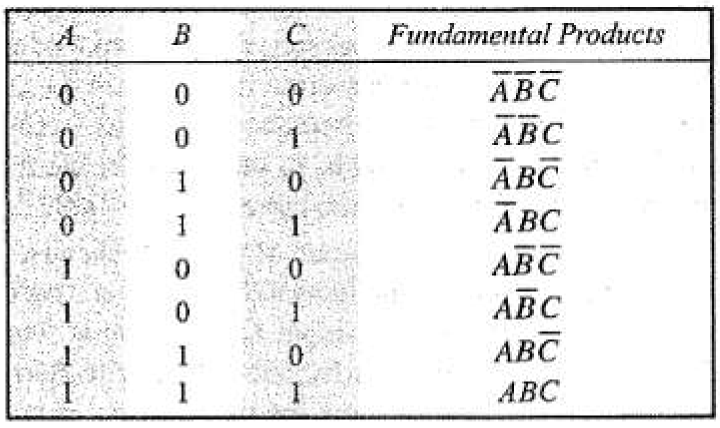
Fig. 3.4 Examples of ANDing three variables and their complements

above three variable minterms can alterna-tively be represented by *mo,*
m1, 1112, 1113, 1114, 1115, in an output of

$$Y = A\overline{B}\overline{C} = 1\cdot\overline{0}\cdot\overline{0} = 1$$

Sum of products Equation

Table 3.2 Fundamental Products for Three Inputs



A	B	C	*Fundamental Products*
0	0	0	$\overline{A}\overline{B}\overline{C}$
0	0	1	$\overline{A}\overline{B}C$
0	1	0	$\overline{A}B\overline{C}$
0	1	1	$\overline{A}BC$
1	0	0	$A\overline{B}\overline{C}$
1	0	1	$A\overline{B}C$
1	1	0	$AB\overline{C}$
1	1	1	ABC

It follows the equations

The sum of equations represents

$$Y = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC \qquad (3.24)$$

lternate representation of Table 3.3,

$$Y = F(A, B, C) = \Sigma m\ (3, 5, 6, 7)$$

$$Y = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC \qquad (3$$

lternate representation of Table 3.3,

$$Y = F(A, B, C) = \Sigma m\ (3, 5, 6, 7)$$

$$Y = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

Alternate representation of Table 3.3,

$$Y = F(A, B, C) = \Sigma m\ (3, 5, 6, 7)$$

Table 3.3 **Design Truth Table**

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 3.4 Fundamental Products for Table 3.3

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	$1 \rightarrow \overline{A}BC$
1	0	0	0
1	0	1	$1 \rightarrow A\overline{B}C$
1	1	0	$1 \rightarrow AB\overline{C}$
1	1	1	$1 \rightarrow ABC$

logic Circuit

After you have a sum-of-products equation, you can derive the co1Tesponding logic circuit by drawing a what amounts to the same thing, a NAND-NAND network. In Eq. (3.24) each product is the output o a 3-i

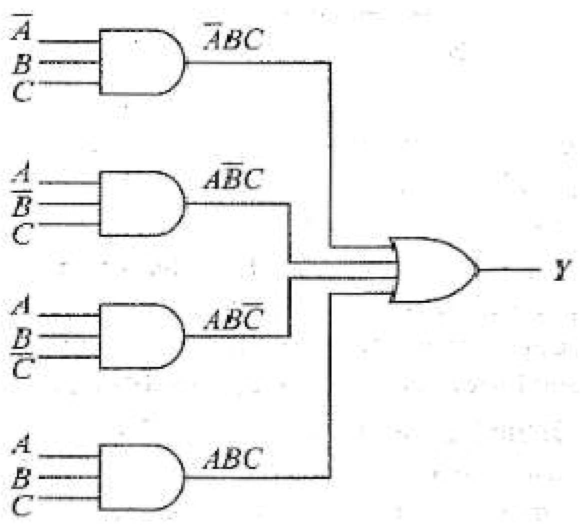


Fig. 3.5 AND-OR solution

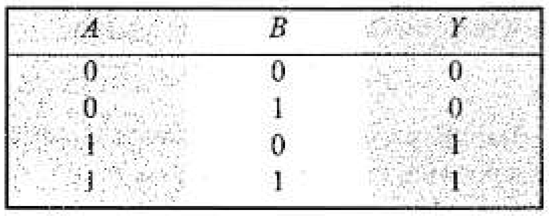
In Fig. 3.6, the bus has six wires with logic signals A, B, C, and their complements. Microcomputers are bus-organized, meaning that the input and output signals of the logic circuits are connected to buses.

TRUTH TABLE TO KARNAUGH MAP

- A *Karnaugh map* is a visual display of the fundamental products needed for a sum-of-products solution. For instance here is how to convert Table 3.5 into its Karnaugh map.
- Begin by drawing Fig. 3.7a. Note the variables and complements:

the vertical column has *A* followed by *A*, and the horizontal row has *B* followed by *B*. The first output 1 appears for *A* = 1 and *B* = 0. The fundamental product for this input condition is AB.

Table 3.5



A	*B*	*Y*
0	0	0
0	1	0
1	0	1
1	1	1

In terms of decimal equivalence each position of Karnaugh map can be drawn as shown in Fig. 3.7b. Note that, Table 3.5 can be written using minterms as

Y= *L m(2,* 3) and Fig. 3.7e represents

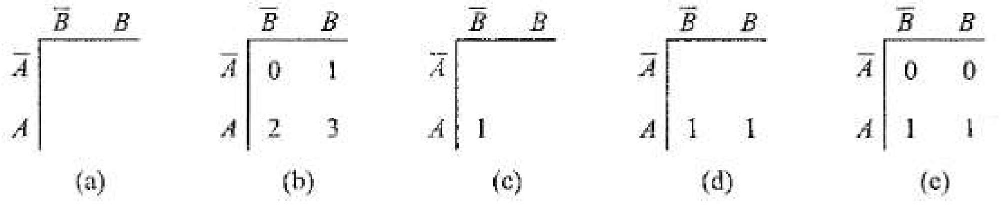


Fig. 3.7 Constructing a Karnaugh map

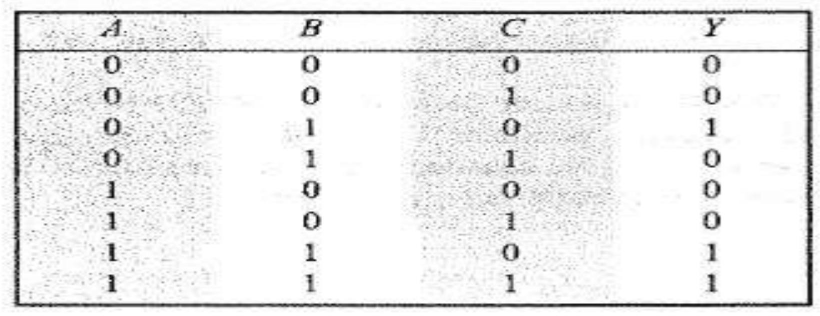
Three-Variable Maps

- Here is how to draw a Karnaugh map for Table 3.6 or for logic equation, *Y* = *F(A*, *B*, C) =

Lm(2,6,7). First, draw the blank map of Fig. 3.8a. T

- The vertical column is labeled *AB*, *AB*, *AB*, and *AB*.
- With this order, only one variable changes from complemented to uncomplemented form (or vice versa) as you move downward.
- In terms of decimal equivalence of each position the Karnaugh map is as shown in Fig. 3.8b. Note how mintenns in the equation gets mapped into corresponding positions in the map.

Table 3.6



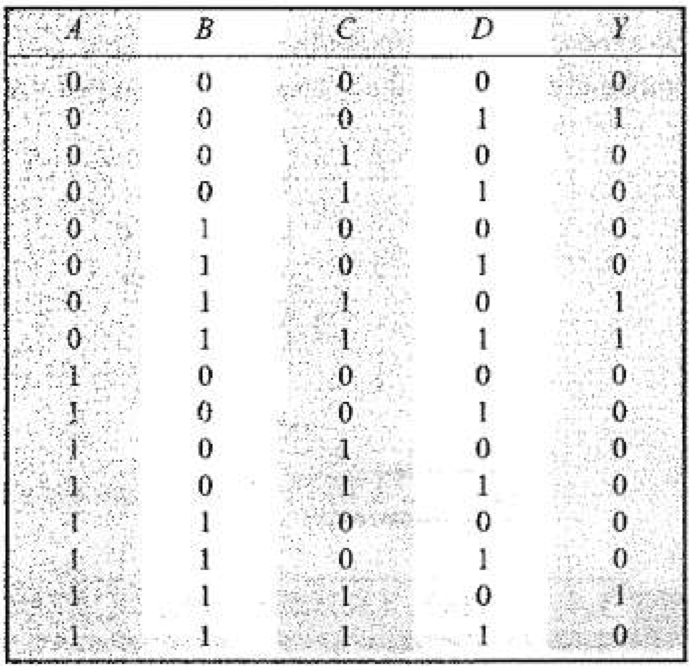
A	*B*	*C*	*Y*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Four-Variable Maps

- Many digital computers and systems process 4-bit numbers. For instance, some digital chips wil work with nibbles like 0000, 0001, 0010, and so on. For this reason, logic circuits are ofte designed to handle four input variables (or their complements). This is why you must know how to draw a four-variable Kamaugh map.
- Here is an example. Suppose you have a truth table like Table 3.7. Start by drawing a blank ma
- like Fig. 3.9a. Notice the order. The vertical column is *AB, AB, AB,*and *AB.*The horizontal row is *CD, CD, CD,* and *CD.* In terms of decimal equivalence of each position the Kam ugh map is as shownin Fig. 3.9b.

- In Table 3.7, you have output ls appearing for *ABCD* inputs of 0001, OllO, 0111, and 1110. The fundamental products for these input conditions are *ABCD,ABCD,ABCD,* and *ABCD.* After enteri g ls on the Karnaugh map, you have Fig. 3.9c. The final step of filling in Os results in the complete ma p of

Table 3.7



A	*B*	*C*	*D*	*Y*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

Fig. 3.9d

Entered Variable Map

As the name suggests, in *entered variable map* one of the input variable is placed inside Kamaugh map. his is done separately noting how it is related with output. This reduces the Karnaugh map size by one degr e,

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$				
$\overline{A}B$				
AB				
$A\overline{B}$				

(a)

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	0	1	3	2
$\overline{A}B$	4	5	7	6
AB	12	13	15	14
$A\overline{B}$	8	9	11	10

(b)

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	0	1	3	2
$\overline{A}B$	4	5	7	6
AB	12	13	15	14
$A\overline{B}$	8	9	11	10

(c)

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	0	1	0	0
$\overline{A}B$	0	0	1	1
AB	0	0	0	1
$A\overline{B}$	0	0	0	0

(d)

Fig. 3.9 Constructing a four-variable Karnaugh map

i.e. a three variable problem that requires 2^3 = 8 locations in Karnaugh map will require i ^{3}-ll = 4 locations in entered variable map. This technique is particularly useful for mapping problems with more than four input variables.

We illustrate the technique by taking a three variable example, truth table of which is shown in Table 3.6. Let's choose C as *map entered variable* and see how output *Y* varies with C for different combinations of other two variables *A* and *B.*

- Fig. 3.1 Oa shows the relation drawn from Table 3.6. For *AB*= 00 we find *Y* = 0 and is not dependent on C. For *AB*= 01 we find *Y* is complement of C thus we can write *Y* = C'. Similarly, for *AB*= 10, *Y* = 0 and for *AB*= 11, *Y*= 1.
- The corresponding entered variable map is shown in Fig. 3.10b. If we choose *A* as map entered variable we have table shown in Fig. 3 .1 Oc showing relation with *Y* for various combinations of *BC;* the corresponding entered variable map is shown in Fig. 3.10d.

A	B	Y
0	0	0
0	1	$\overline{C}$
1	0	0
1	1	1

(a)

	$\overline{B}$	B
$\overline{A}$	0	$\overline{C}$
A	0	1

(b)

B	C	Y
0	0	0
0	1	0
1	0	1
1	1	A

(c)

	$\overline{C}$	C
$\overline{B}$	0	0
B	1	A

(d)

Fig. 3.10 Entered variable map of truth table shown in Table 3.6

PAIRS, QUADS, AND OCTETS

Look at Fig. 3.lla. The map contains a pair of ls that are horizontally adjacent (next to each other). The first l represents the product *ABCD;* the second l stands for the product *ABC I5.* As we move from the first 1 to the second 1, only one variable goes from uncomplemented to complemented form (D to *D);* the other variables don't change form (A, Band C remain uncomplemented). Whenever this happens, you can *eliminate the variable that changes form.*

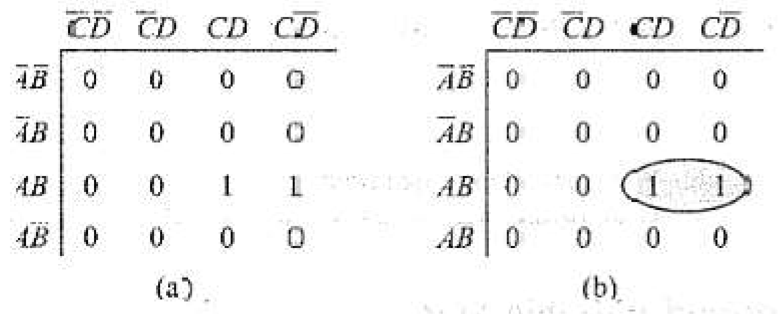


Fig. 3.11 **Horizontally adjacent 1s**

Proof

The sum-of-products equation corresponding to Fig. 3.1 la is

$$Y=ABCD + ABCD$$

which factors into

$$y = ABC(D + D)$$

Since *D* is ORed with its complement, the equation simplifies to

$$Y=ABC$$

- ➢ In general, a pair of horizontally adjacent Is like those of Fig. 3.11 a means the sum-of-products equation will have a variable and a complement that drop out as shown above.
- ➢ For easy identification, we will encircle two adjacent Is as shown in Fig. 3 .11 b. Two adjacent 1s such as these are called a pair. In this way, we can tell at a glance that one variable and its complement will drop out of

- The c01Tesponding Boolean equationother.In words, an encircled pair of ls like those of Fig. 3.1 lb no longer stand for the ORing of two separate products, *ABCD* and *ABCD.* Rather, the encircled pair is visualized as representing a single reduced product *ABC.*
- Here is another example. Figure 3.12a shows a pair of 1s that are vertically adjacent. These ls correspond to *ABC D* and *AB CD.* Notice that only one variable changes from uncomplemented to complemented form (B to *B).* Therefore, *B* and *B* can be factored and eliminated algebraically, leaving a reduced product of*ACD.*

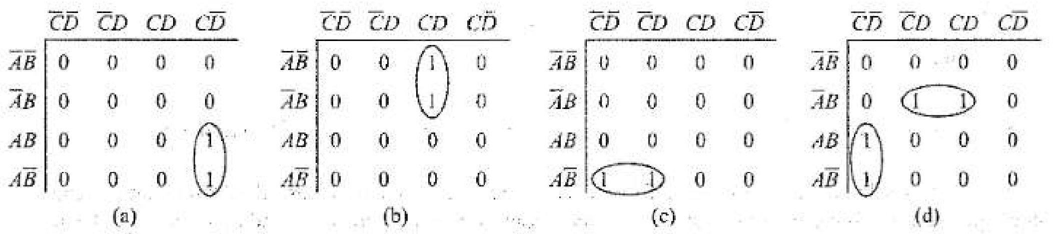


Fig. 3.12 Examples of pairs

More .Examples

- Whenever you see a pair of horizontally or vertically adjacent 1s, you can eliminate the variable that appears in both complemented and uncomplemented form. The remaining variables (or their complements) will be the only ones appearing in the single-product tenn corresponding to the pair of ls.

$$Y= ACD + ABD$$

The Quad

- **A** *quad* is a group of four ls that are horizontally or vertically adjacent. The ls may be end-to-end, as shown in Fig. 3.13a, or in the form of a square, as in Fig. 3.13b.
- When you see a quad, always encircle it because it leads to a simpler product. In fact, a quad eliminates *two variables and their complements.*

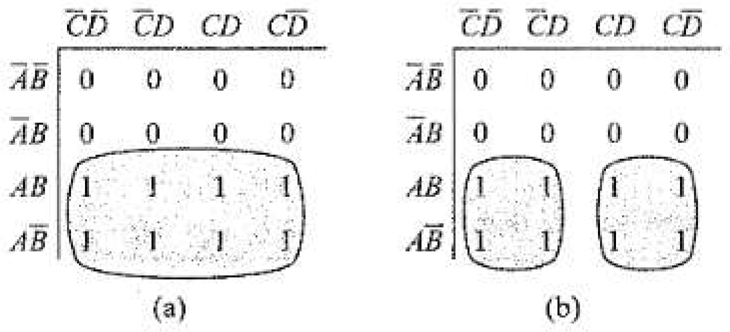


Fig. 3.14 Example of octet

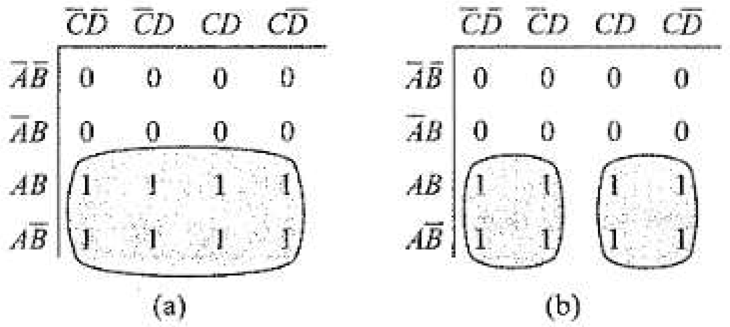


Fig. 3.14 Example of octet

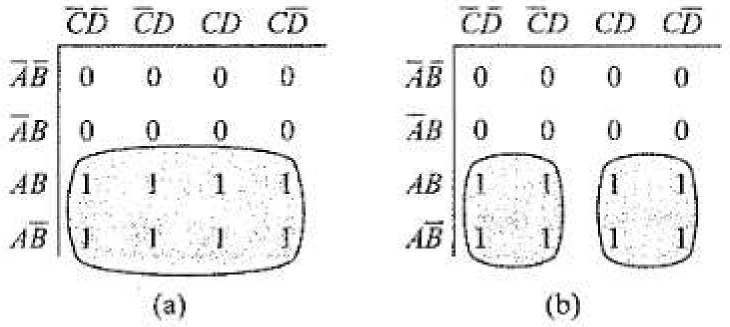


Fig. 3.14 Example of octet

$$Y = AB\overline{C} + ABC$$

This factors into

$$Y = AB(\overline{C} + C)$$

which reduces to

$$Y = AB$$

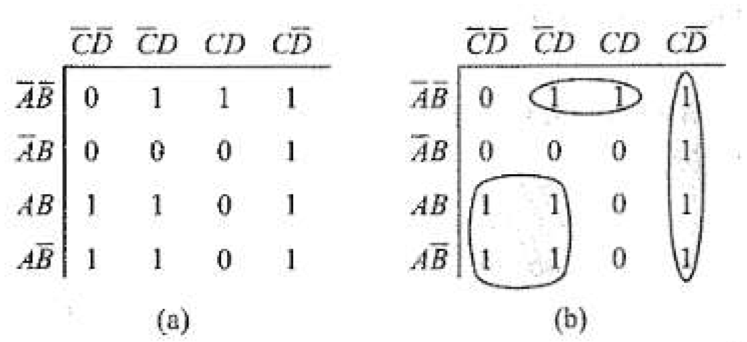
- So the octet of Fig. 3.14a means three variables and their complements drop out of the corresponding product.
- A similar proof applies to any octet. From now on don't bother with the algebra.

- Merely step through the ls of the octet and determine which three variables change fom1. These are the variables that drop out.

KARNAUGH SIMPLIFICATIONS

- A pair eliminates one variable and its complement, a quad eliminates two variables and their complements, and an octet eliminates three variables and their complements.

Because of this, after you draw a karnaugh map,encircle the octets first,the quads, second and the pair lost. In this way the greatest simplification results



(a)

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	1	1
$\bar{A}B$	0	0	0	1
AB	1	1	0	1
$A\bar{B}$	1	1	0	1

(b)

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	1	1
$\bar{A}B$	0	0	0	1
AB	1	1	0	1
$A\bar{B}$	1	1	0	1

Fig. 3.15 Encircling octets, quads and pairs

Karnaugh map:

$$Y = \bar{A}\bar{B}D + A\bar{C} + C\bar{D} \qquad (3.25)$$

Overlapping Groups

The fundamental product *ABC D* is part of the pair and part of the octet. The simplified equation for the

$$Y = A + B\bar{C}D \qquad (3.26)$$

It is valid to encircle the ls as shown in Fig.

3. l 6b, but then the isolated 1 results in a more com-

plicated equation:

$Y = A + \overline{A}B\overline{C}D$

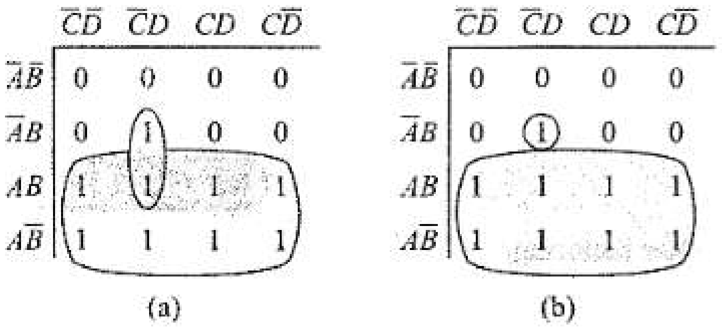


Fig. 3.16 Overlapping groups

Rolling the Map

Another thing to know about is rolling. Look at Fig. 3.17a on the next page. The pairs result in this equation

$$Y = B\overline{C}\overline{D} + BC\overline{D} \qquad (3.27)$$

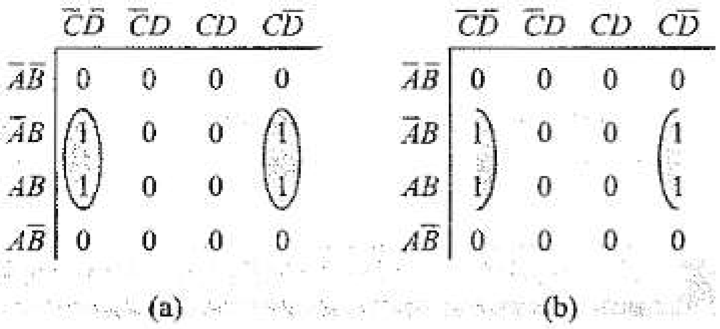


Fig. 3.17 Rolling the Karnaugh map

- Visualize picking up the Kamaugh map and rolling it so that the left side touches the right side. If you are visualizing correctly, you will realize the two pairs actually form a quad.

To indicate this, draw half circles around each pair, as shown in Fig. 3: 17b. From this viewpoint, the uad of Fig. 3.17b has the equation

$$Y = B\overline{D}$$

Why is rolling valid? Because Eq. (3.27) can be algebraically simplified to Eq. (3.28). The proof s arts with Eq. (3.27):

This factors into

$$Y = B\overline{D}(\overline{C} + C)$$

which reduces to

$$Y = B\overline{D}$$

$$Y = B\overline{C}\overline{D} + BC\overline{D}$$

But this final equation is the one that represents a rolled quad like Fig. 3.17b.

Therefore, ls on the edges of Kanaugh map can be grouped with ls on opposite edges.

Eliminating Redundant Groups

- After you have finished encircling groups, eliminate any *redundant group.* This is a group who e ls are already used by other groups. Here is an example. Given Fig. 3.20a, encircle the quad to get Fig. 3.20b. Next, group the remaining 1s into pairs by overlapping (Fig. 3 .20c). In Fig. 3 .20c, all t e 1s of the quad are used by the pairs.
- Because of this, the quad is redundant and can be eliminated to get Fig. 3.20d. As you see, all t e ls are covered by the pairs. Figure 3.20d contains one less product than Fig. 3.20c; therefore, Fig. 3 20d is the most efficient way to group the ls.

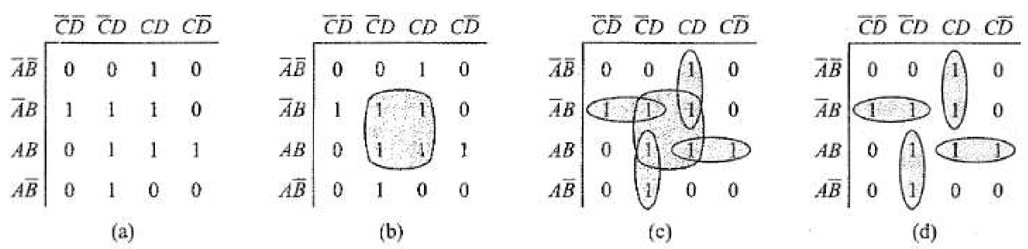


Fig. 3.20 Eliminating an unnecessary group

Conclusion

- Here is a summary of the Karnaugh-map method for simplifying Boolean equations:

- Enter a 1 on the Kamaugh map for each fundamental product that produces a I output in the truth table. Enter Os elsewhere.
- Encircle the octets, quads, and pairs. Remember to roll and overlap to get the largest groups possible.
- If any isolated ls remain, encircle each.
- Eliminate any redundant group.
- Write the Boolean equation by ORing the products corresponding to the encircled groups.

Simplification of Entered Variable Map

- This is similar to Kamaugh map method. Refer to entered variable maps shown in Fig. 3.10. The groupings for these are as shown in Fig. 3.21a and Fig. 3.21b.
- Note that in Fig. 3.21a C' is grouped with 1 to get a larger group as I can be written as 1 = 1 + C'. Similarly *A* is grouped with 1 in Fig. 3.21b.

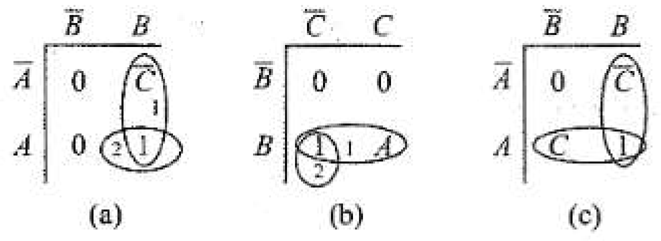


Fig. 3.21 Simplification of entered variable map

This is because one can write 1 = *C* + C' and *C* is included in one group while C' in other. The output of this map can be written as *Y =AC+ BC'.*

Y=F(A,B,C,D)=I:.m(7,9, 10, 11, 12, 13, 14, 15)

DON'T-CARE CONDITIONS

- In some digital systems, certain input conditions never occur during normal operation; therefore the corresponding output never appears. Since the output never appears, it is indicated by an *X* in the truth table.
- For instance, Table 3.8 on the next page shows a truth table where the output is low for all input entries from 0000 to 1000, high for input entry 1001, and an *X* for **l** O10 through 1111. The *X* is called a *don 't-care condition.*
- Whenever you see anX in a truth table, you can let it equal either O or 1, whichever produces a simpler logic circuit.

Table 3.8 **Truth Table with Don't-Care Conditions**

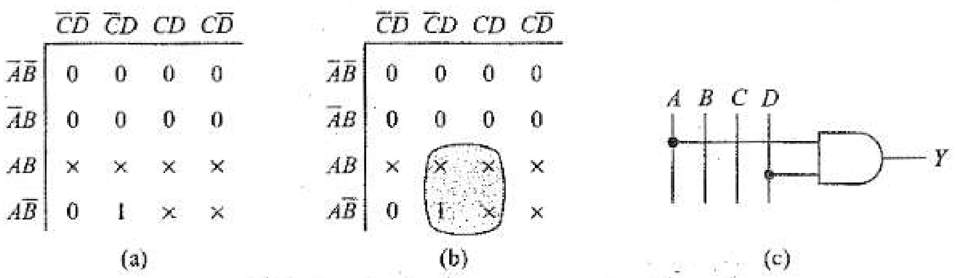
A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

Figure 3.23a shows the Karnaugh map

- 3.8 with don't-cares for all inputs from 1010 to 1111. These don't-cares are like wild cards in p oker because you can let them stand for whatever you like.
  - Figure 3.23b shows the most efficient way to encircle the l. Notice two crucial ideas. irst, the 1 is included in aquad, the largest group you can find if you visualize all *X's* as ls. Second, fter the 1 has been encircled, all X's outside the quad are visualized as Os.
  - In this way, the *Xs* are used to the best possible advantage. As
  - already mentioned, you are free to do this because don't-cares coITespond to input condit ons that never ap-pear.

The quad of Fig. 3.23b results in a Boolean equation of

$$Y=AD$$



$\overline{C}\overline{D}$ $\overline{C}D$ CD $C\overline{D}$
$\overline{A}\overline{B}$ 0 0 0 0
$\overline{A}B$ 0 0 0 0
AB × × × ×
$A\overline{B}$ 0 1 × ×
(a)
$\overline{C}\overline{D}$ $\overline{C}D$ CD $C\overline{D}$
$\overline{A}\overline{B}$ 0 0 0 0
$\overline{A}B$ 0 0 0 0
AB × × × ×
$A\overline{B}$ 0 1 × ×
(b)
A B C D
Y
(c)

Fig. 3.23 Don't-care conditions

Remember these idea about don't-care conditions:

- Given the truth table, draw a Kamaugh map with Os, Is, and don't-cares.
- Encircle the actual ls on the Kamaugh map in the largest groups you can find by treating the don't-cares as ls.
- After the actual ls have been included in groups, disregard the remaining don't cares by visualizing them as Os.

PRODUCT-Of-SUMS METHOD

- With the sum-of-products method the design starts with a truth table that summarizes the desired input-output conditions. The next step is to convert the truth table into an equivale t sum-of-products equationfinal.
- The step is to draw the AND-OR network or its NAND-NAND equivalent.
- The product-of-sums method is similar. Given a truth table, you identify the fundamental ums needed for a logic design. Then by ANDing these sums, you get the product-of-sums equ tion corresponding to the truth table.
- But there are some differences between the two approaches. With the sum-of-pro ucts method, the fundamental product produces an output l for the corresponding input conditi n.
- But with the product-of-sums method, the fundamental sum produces an output O fo the corresponding input condition. The best way to understand this distinction is with an exam le.
- Suppose you are given a truth table like Table 3.9 and you want to get the product-of-sum equation. What you have to do is locate each output O in the truth table and write down its fundamental sum. **In** Table 3.9, the
- first output O appears for *A*= 0, *B* = 0, and *C* = 0. The fundamental sum for these inputs is A+ *B* + C. Why?

This Because produces an output zero for the corresponding input condition:

$$Y=A+B+C=O+O+O=O$$

Table 3.9

A	B	C	Y	*Maxterm*
0	0	0	$0 \rightarrow A+B+C$	M_0
0	0	1	1	M_1
0	1	0	1	M_2
0	1	1	$0 \rightarrow A+\overline{B}+\overline{C}$	M_3
1	0	0	1	M_4
1	0	1	1	M_5
1	1	0	$0 \rightarrow \overline{A}+\overline{B}+C$	M_6
1	1	1	1	M_7

The second output 0 appears for the input condition of $A = 0$, $B = 1$, and $C = 1$. The fundamental sum for this is $A + \overline{B} + \overline{C}$. Notice that B and C are complemented because this is the only way to get a logical sum of 0 for the given input conditions:

$$Y = A + \overline{B} + \overline{C} = 0 + \overline{1} + \overline{1} = 0 + 0 + 0 = 0$$

Similarly, the third output 0 occurs for $A = 1$, $B = 1$, and $C = 0$; therefore, its fundamental sum is $\overline{A} + \overline{B} + C$:

$$Y = \overline{A} + \overline{B} + C = \overline{1} + \overline{1} + 0 = 0 + 0 + 0 = 0$$

$$Y = (A + B + C)(A + \overline{B} + \overline{C})(\overline{A} + \overline{B} + C) \qquad (3.31)$$

$$Y = F(A, B, C) = \Pi M(0, 3, 6)$$

logic Circuit

- Each sum represents the output of a 3-input OR gate. Furthem10re, the logical product *Y* i the output of a 3-input AND gate. Therefore, you can draw the logic circuit as shown in Fig. 3.26.
- A 3-input OR gate is not available as a TTL chip. So, the circuit of Fig. 3.26 is not practical. ith De Morgan's first theorem, however, you can replace the OR-AND circuit of Fig. 3.26 b the NOR-NOR circuit ofFig. 3.27.

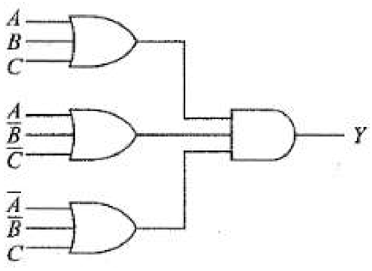


Fig. 3.26 Product-of-sums circuit

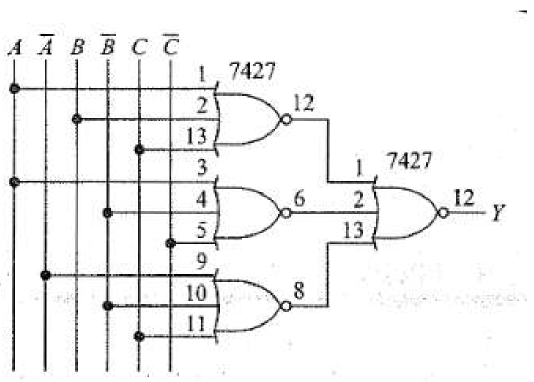


Fig. 3.27

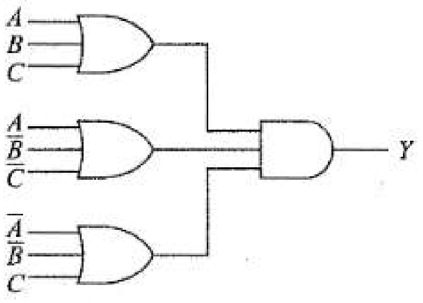
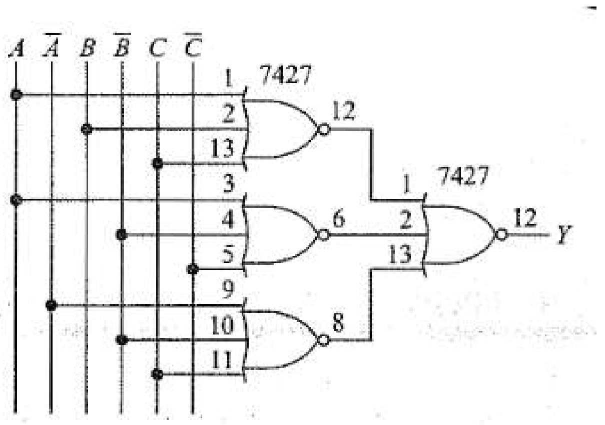



Fig. 3.26 Product-of-sums circuit

Fig. 3.27

Conversion between SOP and POS

- We have seen that SOP representation is obtained by considering ones in a truth table while POS comes considering zeros. In SOP, each one at output gives one AND tem1 which is finally ORed.
- In POS, each zero gives onewhichORterm is finally ANDed. Thus SOP and POS occupy complementary locations in a tmth table and one representation can be obtained from the other by identifying complementary locations,
- changing mintenn to maxtenn or reverse, and finally changing summation by product or reverse.

Thus Table 3.9 can be represented as

$$Y = F(A, B, C) = \Pi M(0, 3, 6) = \Sigma m(1, 2, 4, 5, 7)$$

Similarly Table 3.4 can be represented as

$$Y = F(A, B, C) = \Sigma m(3, 5, 6, 7) = \Pi M(0, 1, 2, 4)$$

This is also known as *conversion between canonical forms.*

PRODUCT-Of-SUMS SIMPLIFICATION

- After you write a product-of-sums equation, you can simplify it with Boolean algebra. Alternatively, you may prefer simplification based on the Kamaugh map. There are several ways of using the Kanaugh map.
- One can use a similar technique as followed in SOP representation but by forming largest group of zeros and then replacing each group by a sum term. The variable going in the formation of sum term is inverted if it remains constant with a value 1 in the group and it is not inverted if that value is 0. Finally, all the sum terms are ANDed to get simplest POS fonn. We illustrate this in Examples 3.11 and 3.12this.In section we also present an interesting alternative to above technique.

Sum-of-Products Circuit

Suppose the design starts with a truth table like Table 3.10. The first thing to dotheis to draw Kanaugh map in the usual way to get Fig. 3.28a. The encircled groups allow us to write a sum-of-products equation:

$$Y = \overline{A}\,\overline{B} + AB + AC$$

Figure 3.28b shows the corresponding NAND-NAND circuit.

Complementary Circuit

- To get a product-of-sums circuit, begin by comple-menting each O and 1 on the Kamaugh map of Fig. 3.28a. This results in the complemented map shown in Fig. 3.28c. The encircled ls allow us to write the following sum-of-products equation:

$$\overline{Y} = \overline{A}B + A\overline{B}\,\overline{C}$$

Finding the NOR-NOR Circuit

What we want to do next is to get the product-of-sums solution, the NOR-NOR circuit that produces the

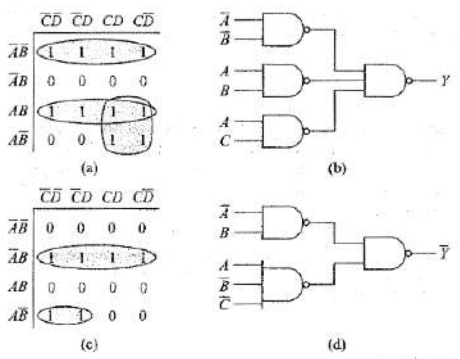


Fig. 3.28 Deriving the sum-of-products circuit

Table 3.10

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

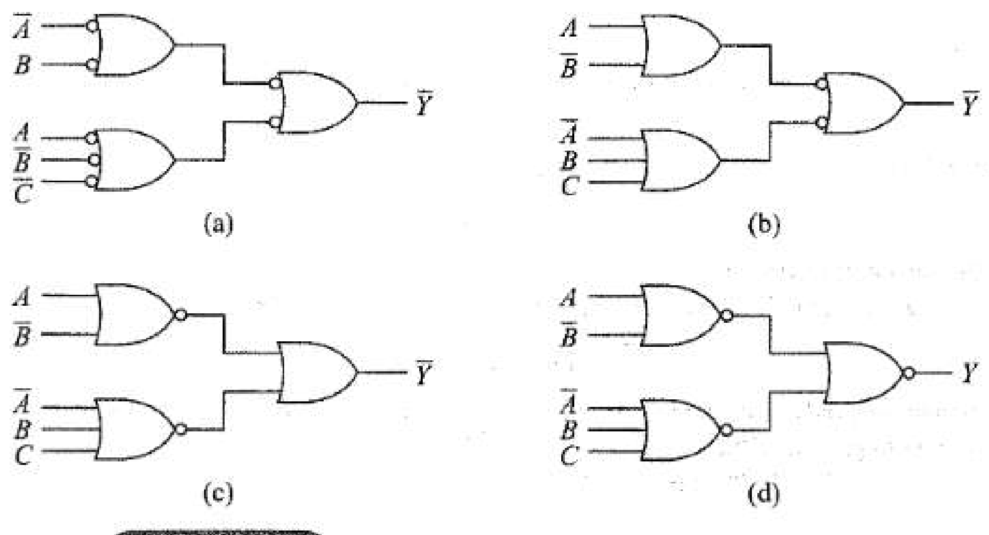


Fig. 3.29 Deriving the product-of-sums circuit

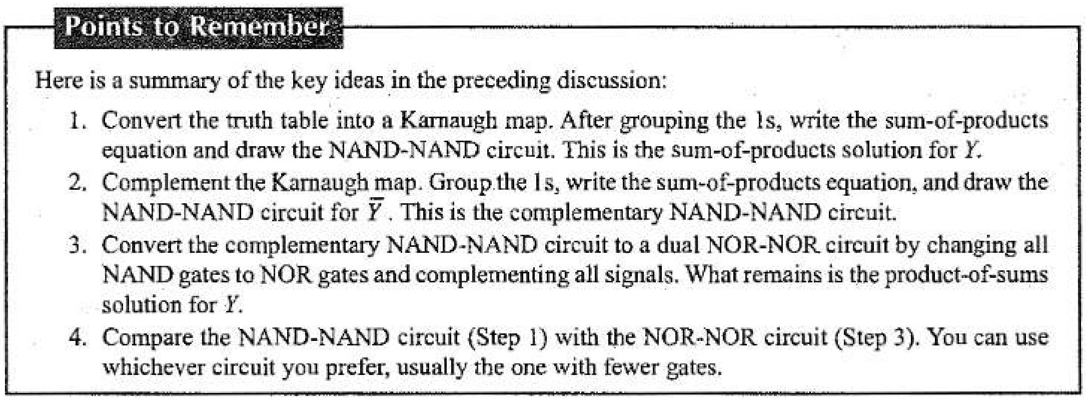
Duality

- An earlier section introduced the duality theorem of Boolean algebra. Now we are ready to apply this theo-rem to logic circuits. Given a logic circuit, we can find its dual circuit as follows:
- Change each AND gate to an OR gate, change each OR gate to an AND gate, and complement all input-output signals. An equivalent statement of duality is this: Change each NAND gate to a NOR gate, change each NOR gate to a NAND gate, and complement all input-output signals.

Here is a summary of the key ideas in the preceding discussion:

- Convert the truth table into a Karnaugh map. After grouping the ls, write the sum-of-products equation and draw the NAND-NAND circuit. This is the sum-of-products solution for *Y*.
- Complement the Karnaugh map. Group the 1s, write the sum-of-products equation, and draw the NAND-NAND circuit for *Y*. This is the complementary NAND-NAND circuit.

- Convert the complementary NAND-NAND circuit to a dual NOR-NOR circuit by changing all NAND gates to NOR gates and complementing all signals. What remains is the product-of-sums solution for *Y*.
- Compare the NAND-NAND circuit (Step 1) with the NOR-NOR circuit (Step 3). You can use whichever circuit you prefer, usually the one with fewer gates.



Points to Remember

Here is a summary of the key ideas in the preceding discussion:

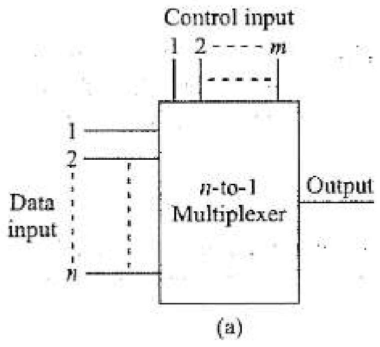
1. Convert the truth table into a Karnaugh map. After grouping the 1s, write the sum-of-products equation and draw the NAND-NAND circuit. This is the sum-of-products solution for *Y*.
2. Complement the Karnaugh map. Group the 1s, write the sum-of-products equation, and draw the NAND-NAND circuit for $\overline{Y}$. This is the complementary NAND-NAND circuit.
3. Convert the complementary NAND-NAND circuit to a dual NOR-NOR circuit by changing all NAND gates to NOR gates and complementing all signals. What remains is the product-of-sums solution for *Y*.
4. Compare the NAND-NAND circuit (Step 1) with the NOR-NOR circuit (Step 3). You can use whichever circuit you prefer, usually the one with fewer gates.

Data-Processing Circuits

- This chapter is about logic circuits that process binary data. We begin with a discussion of multiplexers, which are circuits that can select one of many inputs.
- Then you will see how multiplexers are used as a design alternative to the sum-of-products solution. This will be followed by an examination of a variety of circuits, such as demultiplexers, decoders, encoders, exclusive-OR gates, parity checkers, magnitude comparator, and read-only memories.
- The chapter ends with a discussion of programmable logic arraysrelevantand HDL concepts.

MULTIPLEXERS

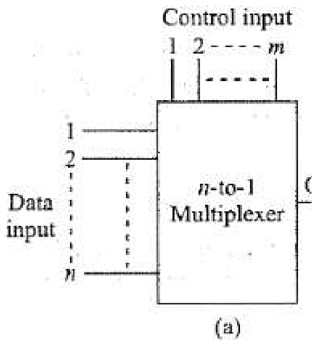
- *Multiplex* means *many into one.* A *multiplexer* is a circuit with many inputs but only one output. By applying control signals, we can steer any input to the output. Thus it is also called a *data selector* and control inputs are termed select inputs.
- Figure 4.la illustrates the general idea. The circuit has *n* input signals, *m* control signals and 1 output signal. Note that, *m* control signals can select at the most 2^{111} input signals thus $n \sim 2^{111}$ •
- The circuit diagram of a 4-to- l multiplexer is shown in Fig. 4.1 c and its truth table in Fig. 4.1 b. Depending on control inputs *A,* Bone of the four inputs *Do* to D_3 is steered to output *Y.*
- Let us write the logic equation of this circuit. Clearly, it will give a SOP representation, each AND gate



A	B	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

(b)

Fig. 4.1 (a) Multi
(c) Its log



A	B	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

(b)

Fig. 4.1 (a)
(c)

16-to-1 Multiplexer

Figure shows a 16-to-l multiplexer. The input bits are labeled *Do* to D_{15} . Only one of these is transmitted to the output. Which one depends on the value of *ABCD,* the control input. For instance, when *ABCD=OOOO*

the upper AND gate is enabled while all other AND gates are disabled. Therefore, data bit *Do* is transmitted to the output, giving *Y=Do*

If *Do* is low, *Y* is low; if *Do* is high, *Y* is high.is The point that *Y* depends only on the value of *Do.* lf the control nibble (group of 4-bits) is changed to

ABCD=

1111

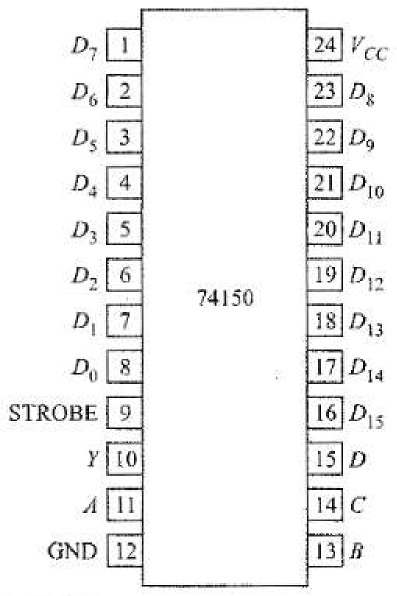
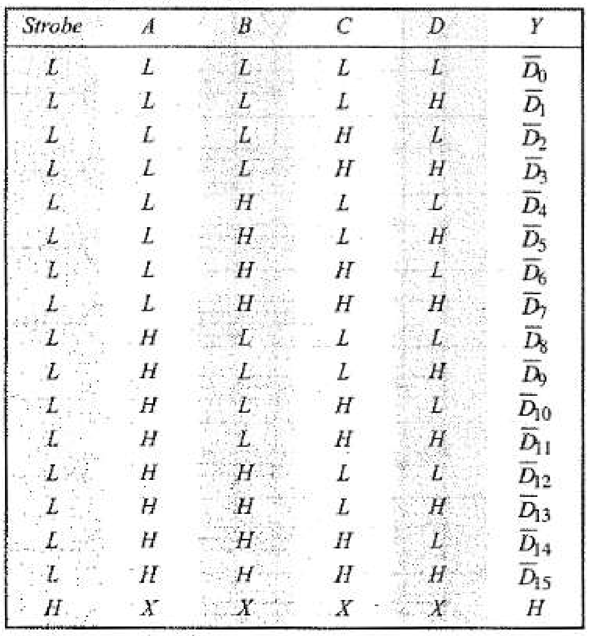


Fig. 4.3 Pinout diagram of 74150

Table 4.1 74150 Truth Table



Strobe	A	B	C	D	Y
L	L	L	L	L	$\overline{D_0}$
L	L	L	L	H	$\overline{D_1}$
L	L	L	H	L	$\overline{D_2}$
L	L	L	H	H	$\overline{D_3}$
L	L	H	L	L	$\overline{D_4}$
L	L	H	L	H	$\overline{D_5}$
L	L	H	H	L	$\overline{D_6}$
L	L	H	H	H	$\overline{D_7}$
L	H	L	L	L	$\overline{D_8}$
L	H	L	L	H	$\overline{D_9}$
L	H	L	H	L	$\overline{D_{10}}$
L	H	L	H	H	$\overline{D_{11}}$
L	H	H	L	L	$\overline{D_{12}}$
L	H	H	L	H	$\overline{D_{13}}$
L	H	H	H	L	$\overline{D_{14}}$
L	H	H	H	H	$\overline{D_{15}}$
H	X	X	X	X	H

all gates are disabled except the bottom AND gate. In this case, D $_{15}$ is the only bit transmitted to the output, and *Y=D1s*

As you can see, the control nibble determines which of the input data bits is transmitted to the output. Thus we can write output as

$$Y = A'B'C'D'.D_0 + A'B'C'D.D_1 + A'B'CD'.D_2 + \dots + ABCD'.D_{14} + ABCD.D_{15}$$

$$Y = A'B'C'.D_0 + A'B'C.D_1 + A'BC'.D_2 + A'BC.D_3 + AB'C'.D_4 + AB'C.D_5 + ABC'.D_6 + ABC.D_7$$

The 74150

- Try to visualize the 16-input OR gate of Fig. 4.2 changed to a NOR gate. What effect does this have on the operation of the circuit? Almost none.
- All that happens is we get the complement of the selected data bit rather than the data bit itself. For instance, when *ABCD* = 0111,is the output

Y= D7

- This is the Boolean equation for a typical transistor-transistor logic (TTL) multiplexer because it has an
- inverter on the output that produces the complement of the selected data bit.
- The 74150 is a 16-to-l TTL multiplexer with the pin diagram shown in Fig. 4.3. Pins 1 to 8 and 16 to 23 are
- for the input data bits *D0* to *D 15* . Pins 11, 13, 14, and 15 are for the control bits *ABCD.* Pin 10 is the output;

. **Multiplexer logic**

- Digital design usually begins with a truth table. The problem is to come up with a logic circuit that has the same truth table. In Chapter 3, you saw two standard methods for implementing a truth table: the sum-of-products and the product-of-sums solutions.
- The third method is the *multiplexer solution.* For example, to use a 74150 to implement Table 4.2. Complement each Youtput to get the corresponding data input:

$$D_0 = \overline{1} = 0$$
$$D_1 = \overline{0} = 1$$
$$D_2 = \overline{1} = 0$$

ınd so forth, up to

$$D_{15} = \overline{1} = 0$$

Bubbles on Signal lines

Data sheets often show inversion bubbles on some of the signal lines. For instance, notice the bubble on pin 10, the output of Fig. 4.4.

This bubble is a reminder that the output is the complement of the selected data bit.

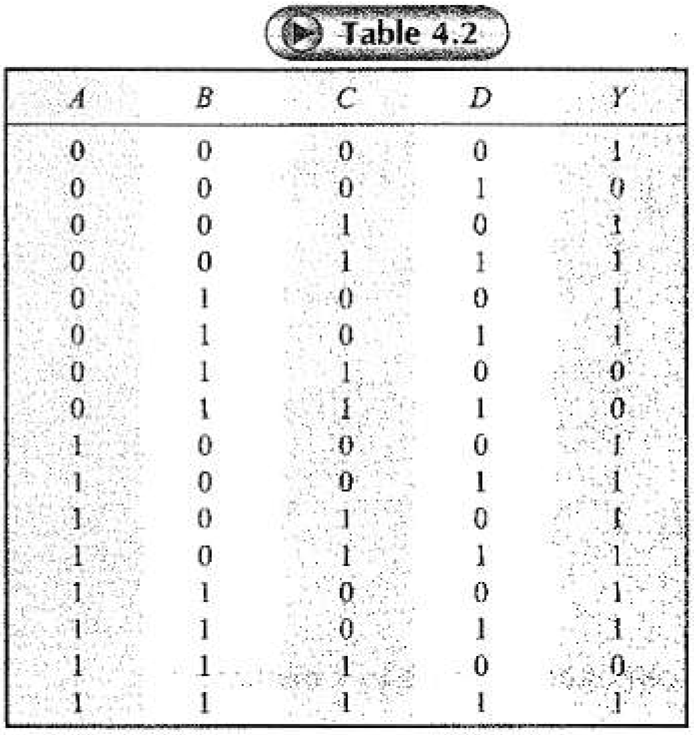


Table 4.2

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

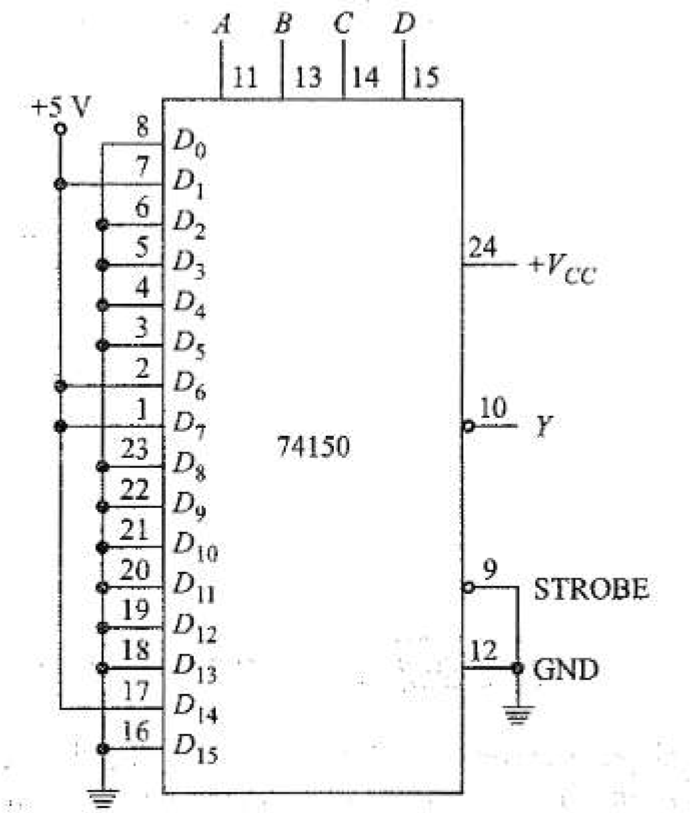
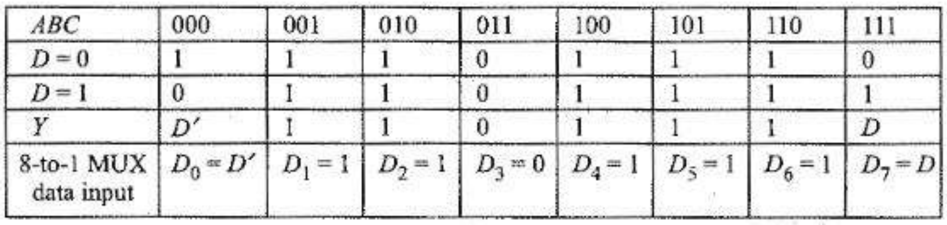


Fig. 4.4 Using a 74150 for multiplexer logic

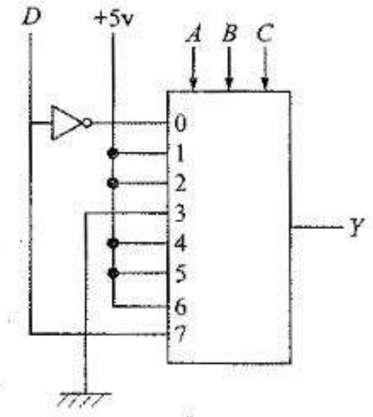
Universal logic Circuit

- Multiplexer sometimes is called *universal logic circuit* because a *2n-to-l* multiplexer can be used as a design solution for any *n* variable truth table. Thisseenwe have for realization of a 4 variable truth table by 16-to-l multiplexer in Fig. 4.5. Here, we show how this truth table can be realized using an 8-to-l multiplexer.
- Let's consider *A*,*B* and *C* variables to be fed as select inputs. The fourth variable *D* then has to be present as data input. The method is shown in Fig. 4.5a. The first three rows map the truth table in a different way, similar to the procedure we adopted in entered variable map (Section 3.3). We write all the combinations of3 select inputs in first row along different columns. Now corresponding to each value of 4th variable *D,* truth table



ABC	000	001	010	011	100	101	110	111
$D=0$	1	1	1	0	1	1	1	0
$D=1$	0	1	1	0	1	1	1	1
Y	D'	1	1	0	1	1	1	D
8-to-1 MUX data input	$D_0=D'$	$D_1=1$	$D_2=1$	$D_3=0$	$D_4=1$	$D_5=1$	$D_6=1$	$D_7=D$

(a)



(b)

Fig. 4.5 A four variable truth table realization using 8-to-1 multiplexer

Nibble Multiplexers

Sometimes we want to select one of two input nibbles. In this case, we can use a nibble multiplexer like the one shown in Fig. 4.6. The input nibble on the left is *A3*A_2*A1Ao* and the one on the right is $B_3B_2B_1B_0$.

The control signal labeled *SELECT* determines which input nibble is transmitted to the output. When SELECT is low, the four NAND gates on the left are activated; therefore,

Y3 Y2Y1 Yo= A3A2A1Ao

When SELECT is high, the four NAND gates on the right are active, and

Y3 Y2Y1 Yo =B3B2B1Bo

Figure 4.7a on the next page shows the pinout diagram of a 74157, a nibble multiplexer with a SELECT described inputas. previously When SELECT is low, the left nibble is steered to the output.

When SELECT ,

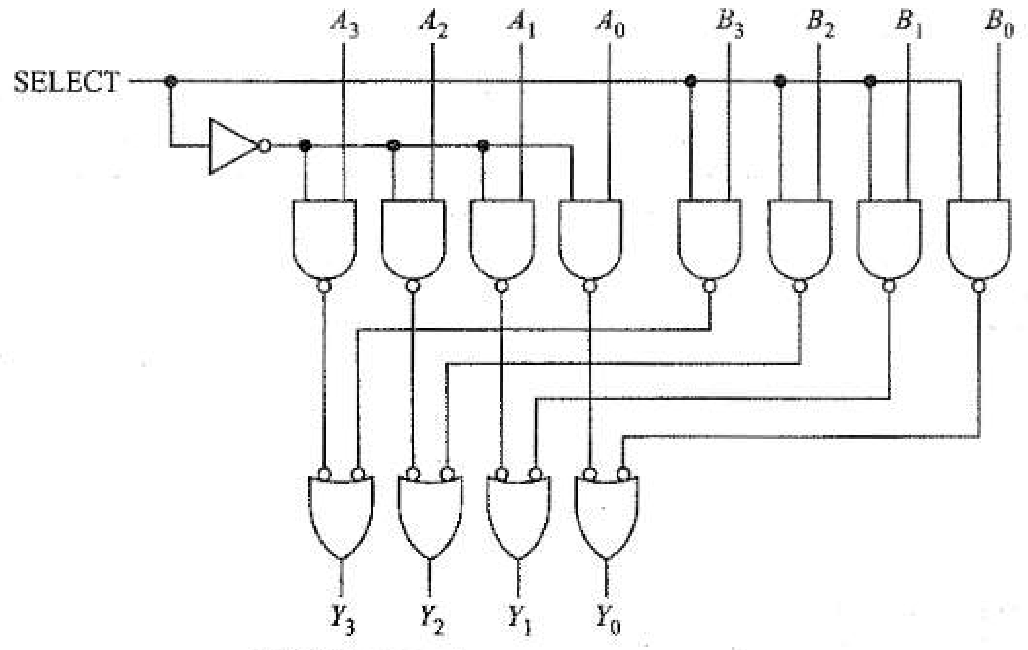


Fig. 4.6 Nibble multiplexer

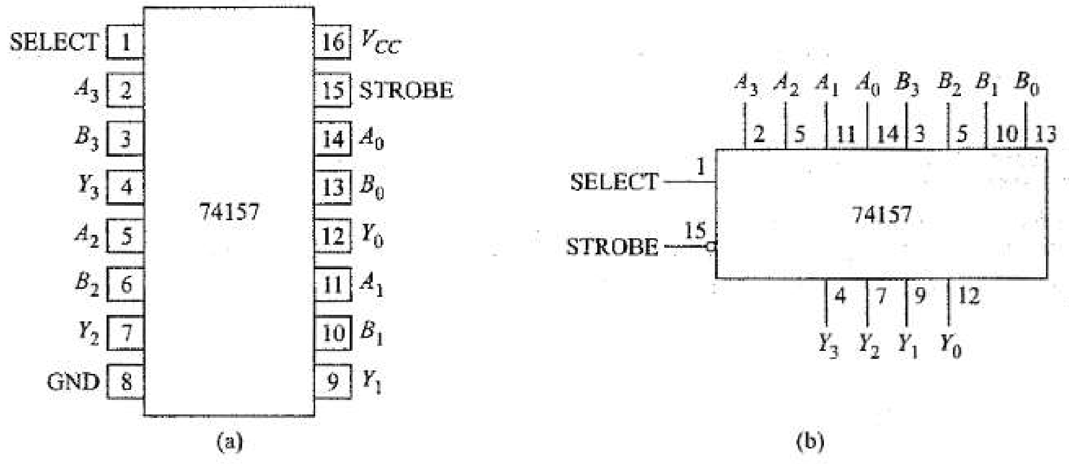


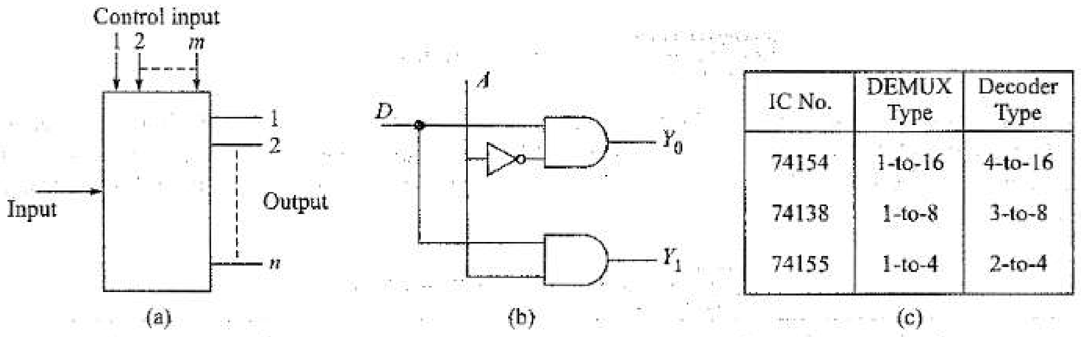
Fig. 4.7 Pinout diagram of 74157

DEMULTIPLEXERS

- *Demultiplex* means *one into many.* A *demultiplexer* is awithlogic circuit one input and many outputs. By applying control signals, we can steer the input signal to one of the output lines. Figure 4.9a illustrates the general idea.
- The circuit has 1 input signal, *m* control or select signals and *n* output signals where *n* :::; 2^{111} • Figure 4.9b shows the circuit diagram of a 1-to-2 demultiplexer. Note the similarity of multiplexer

and demultiplexer circuits in generating different combinations of control variables through a bank ofAND gates.

- Figure 4.9c lists some of the commercially available demultiplexer I Cs. Note that a demultiplexer IC can also behave like a decoder. More about this will be discussed in next section.



IC No.	DEMUX Type	Decoder Type
74154	1-to-16	4-to-16
74138	1-to-8	3-to-8
74155	1-to-4	2-to-4

Fig. 4.9 **(a) Demultiplexer block diagram, (b) Logic circuit of 1-to-2 demultiplexer, (c) Few commercially available ICs**

1-to-16 Demultiplexer

- Figure 4.10 shows a l-to-16 demultiplexer. The input bit is labeled *D.* This data bit (D) is transmitted to the data bit of the output lines. But which one? Again, this depends on the value of *ABCD*, the control input. When *ABCD* = 0000, the upper AND gate is enabled while all other AND gates are disabled.
- Therefore, data bit *Dis* transmitted only to the *Yo* output, giving *Yo*= *D.* If *Dis* low, *Yo* is low. If*D* is high, *Yo* is high. As you can see, the value of Y_0 depends on the value of *D.* All other outputs are in the low state. If the control nibble is changed to *ABCD* = 1111, all gates are disabled except the bottom AND gate. Then, *D* is transmitted only to the Y1s output, and Y1s = *D.*

The 74154

The 74154 is a l-to-16demultiplexer with the pin diagram ofFig. 4.11. Pin 18 is for the input DATAD, and pins 20 to 23 are for the control bits *ABCD.*

Pins l to 11 and 13 to 17 are for the output bits *Yo* to Y_{15} • Pin 19 is for the STROBE, again an active-low input. Finally, pin 24 is for *V*cc and pin 12 for ground.

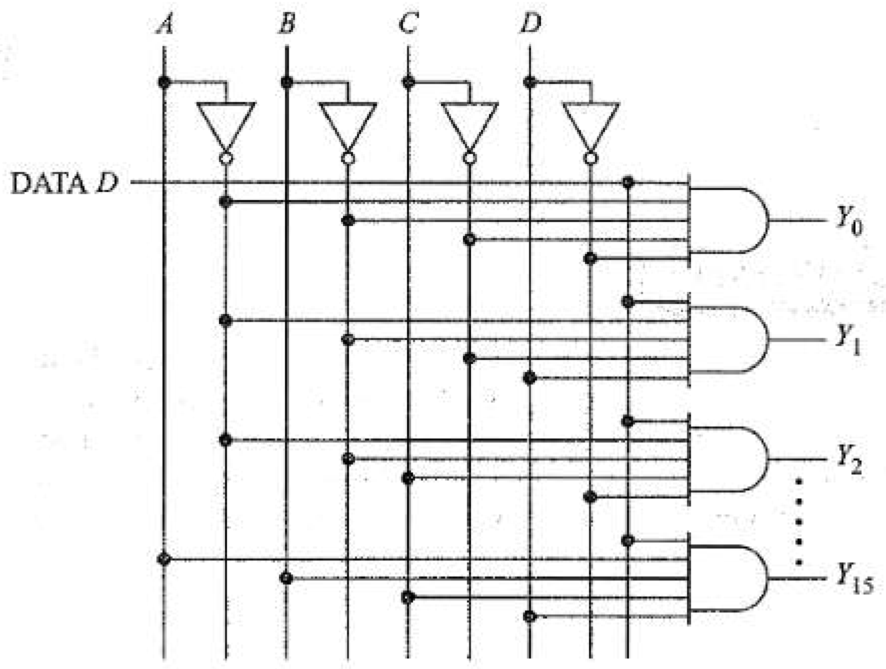
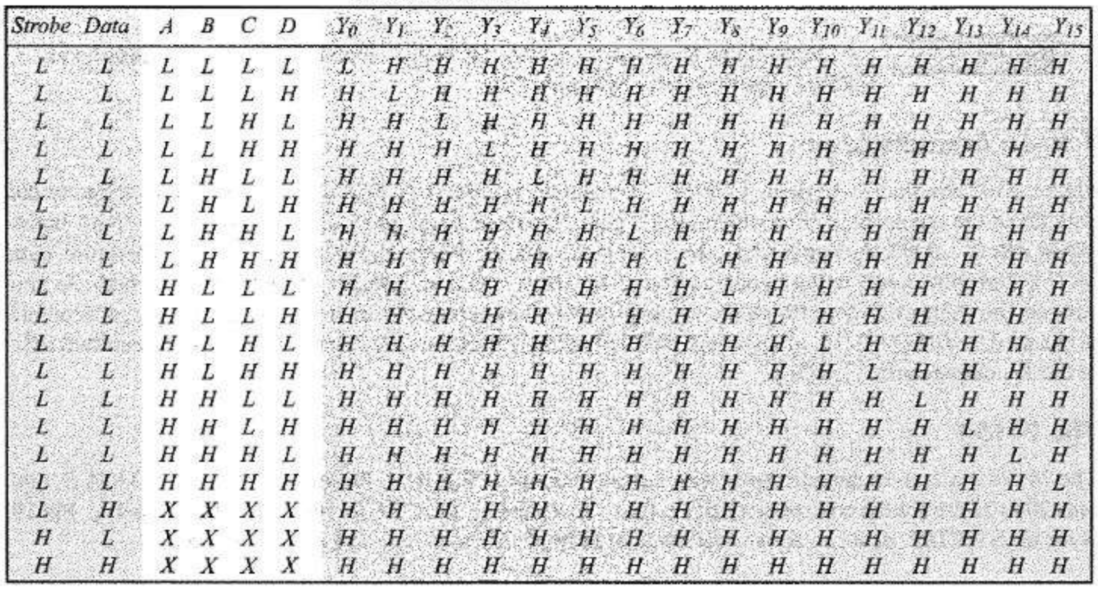


Fig. 4.10 **1-to-16 demultiplexer**

shows the truth table of a 74154. First, notice the STROBE input. It must be low to activate the 74154. When the STROBE is low, the control inputABCD determines which output lines are low when the DATA input is low.

When the DATA input is high, all output lines are high. And, when the STROBE is high, all output lines are high.

Table 4.3 **74154 Truth Table**



Strobe	*Data*	*A*	*B*	*C*	*D*	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	Y_9	Y_{10}	Y_{11}	Y_{12}	Y_{13}	Y_{14}	Y_{15}
L	L	L	L	L	L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	H	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	H	L	L	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H
L	L	L	H	L	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H
L	L	L	H	H	L	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H
L	L	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H
L	L	H	L	L	L	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H
L	L	H	L	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H
L	L	H	L	H	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H
L	L	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H
L	L	H	H	L	L	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H
L	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H
L	L	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H
L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L
L	H	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	L	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	H	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H

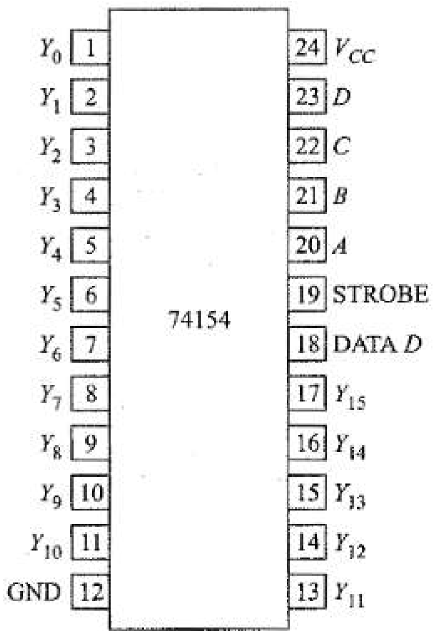


Fig. 4.11 Pinout diagram of 74154

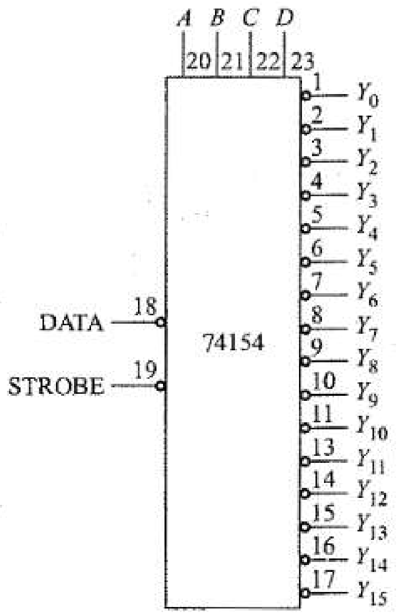


Fig. 4.12 Logic diagram of 74154

BCD-TO-DECIMAL DECODERS

BCD is an abbreviation for *binary-coded decimal.* The BCD code expresses each digit in a decimal number by its nibble equivalent. For instance, decimal number 429 is changed to its BCD form as follows:

$$\begin{array}{ccc} 4 & 2 & 9 \\ \downarrow & \downarrow & \downarrow \\ 0100 & 0010 & 1001 \end{array}$$

To anyone using the BCD code, 0100 0010 1001 is equivalent to 429.

As another example, here is how to convert the decimal number 8963 to its BCD form:

$$\begin{array}{cccc} 8 & 9 & 6 & 3 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 1000 & 1001 & 0110 & 0011 \end{array}$$

Again, we have changed each decimal digit to its binary equivalent.

Some early computers processed BCD numbers. This means that the decimal numbers were changed into BCD numbers, which the computer then added, subtracted, etc. The final answer was converted from BCD back to decimal numbers.

$$\begin{array}{ccc} 0101 & 0111 & 1000 \\ \downarrow & \downarrow & \downarrow \\ 5 & 7 & 8 \end{array}$$

One final point should be considered. Notice that BCD digits are from 0000 to 1001. All combinations above this (1010 to 1111) cannot exist in the BCD code because the digit highest decimal being coded is 9.

BCD-to-Decimal Decoder

The circuit of Fig. 4.18 is called a *l-of-10 decoder* because only 1 of the 10 output lines is high. For instance,

when *ABCD* is 0011, only the *Y3* AND gate has all high inputs; therefore, only the Y output is high, If*ABCD* changes to 1000, only the Y AND gate hasinputs;allhigh as a result, only the Y output goes high. 8 8

BCD-to-decimal converter.

The 7445

Typically, you would not build a decoder with separate inverters and AND gates, as shown in Fig. 4.18. Instead, you would use a TTL IC like the 7445 ofFig. 4.19.

Pin 16 connects to the supply voltage *V*cc and pin 8 is grounded. Pins 12 to 15 are for the BCD input *(ABCD),* while pins I to 7 and 9 to 11 are for the outputs.

This IC is functionally equivalent to the one in Fig. 4.18, except that the active output line is in the low state. All other output lines are in the high state, as shown in Table 4.4.

Notice that an invalid BCD input (1010 to 1111) forces all output lines into the high state.

ENCODERS

An encoder converts an active input signal into a coded output signal. Figure 4.24 illustrates the general idea. There are *n* input lines, only one of which is active.

Internal logic within the encoder converts this active input to a coded binary output with *m* bits.

Decimal-to-BCD Encoder

high inputs, therefore, the output is

$$ABCD = 0011$$

If button 5 is pressed, the output becomes

$$ABCD = 0101$$

When switch 9 is pressed,

$$ABCD = 1001$$

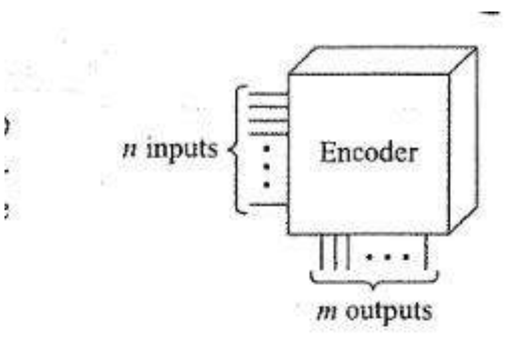


Fig. 4.24 Encoder

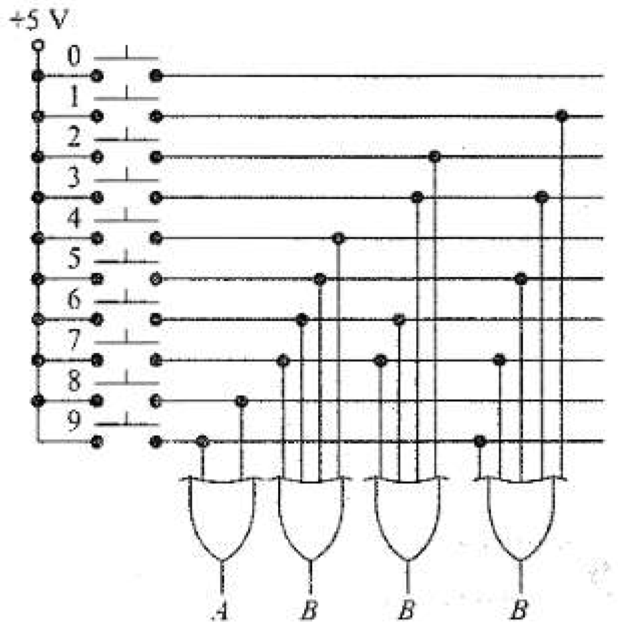


Fig. 4.25 Decimal-to-BCD encoder

The 74147

- Figure 4.26a is the pinout diagram for a 74147, a decimal-to-BCD encoder. The decimal input, *X $_1$* to *X9*, connect to pins 1 to 5, and 10 to 13. The BCD output comes from pins 14, 6, 7, and 9. Pin 16 is for the supply voltage, and pin 8 is grounded. The label NC on pin 15 means *no connection* (the pin is not used).
- Figure 4.26b shows how to draw a 74147 on a schematic diagram. As usual, the bubbles indicate active-low inputs and outputs.
- Table 4.5 is the truth table ofa 74147. Notice the following. When allXinputs are high, all outputs are high. When *X $_9$* is low, the *ABCD* output is *LHHL* (equivalent to 9 if you complement the bits). When X_8 is the only low input, *ABCD* is *LHHH*

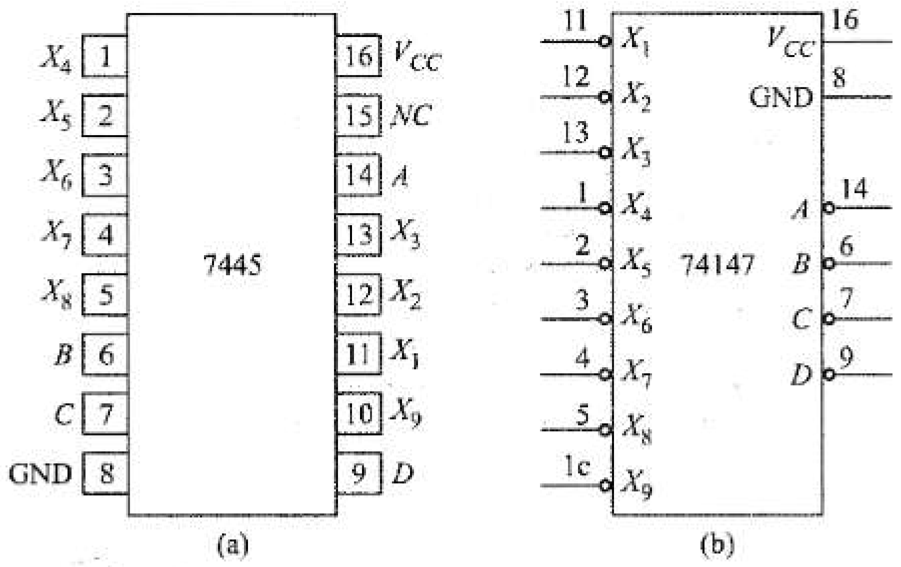


Fig. 4.26 (a) Pinout diagram of 74147, (b) Logic diagram

Table 4.5 74147 Truth Table

Inputs									Outputs			
X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	A	B	C	D
H	H	H	H	H	H	H	H	H	H	H	H	H
X	X	X	X	X	X	X	X	L	L	H	H	L
X	X	X	X	X	X	X	L	H	L	H	H	H
X	X	X	X	X	X	L	H	H	H	L	L	L
X	X	X	X	X	L	H	H	H	H	L	L	H
X	X	X	X	L	H	H	H	H	H	L	H	L
X	X	X	L	H	H	H	H	H	H	L	H	H
X	X	L	H	H	H	H	H	H	H	H	L	L
X	L	H	H	H	H	H	H	H	H	H	L	H
L	H	H	H	H	H	H	H	H	H	H	H	L

EXCLUSIVE-OR GATES

- The *exclusive-OR gate* has a high output only when an odd number of inputs is high. Figure 4.29 shows how to build an exclusive-OR gate.
- The upper AND gate forms the product *AB,* while the lower one produces *AB.* There- fore, the output of the OR gate is

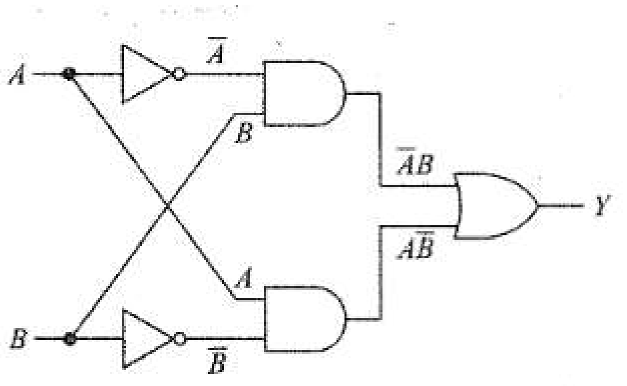


Fig. 4.29 Exclusive-OR gate

$$Y = \overline{A}B + A\overline{B}$$

Table 4.6 shows the truth table for a 2-input exclusive-OR gate. The output is high when *A* or Bis high, but not when both high are. This is why the circuit is known as an exclusive-OR gate. In other words, the output is a 1 *only* when the inputs are different

Table 4.6 Exclusive-OR Truth Table

A	*B*	*Y*
0	0	0
0	1	1
1	0	1
1	1	0

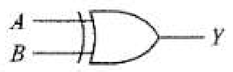


Fig. 4.30 Logic symbol for exclusive-OR gate

Four Inputs

- Figure 4.3 la shows a pair of exclusive-OR gates driving an exclusive-OR gate. If all inputs *(A* to *D)* are low, the input gates have low outputs, so the fi-nal gate has a low output. If *A* to C are low and *D* is high, the upper gate has a low output, the lower gate has a high output, and the output gate has a high output.

- If we continue analyzing the circuit operation for the remaining input possibilities, we can work out Table 4. 7. Here is an important prope1ty of this truth table. Each *ABCD* input with an odd number of ls produces an output I.

- For instance, the first *ABCD* entry to produce an output 1 is 000 l; it has an odd

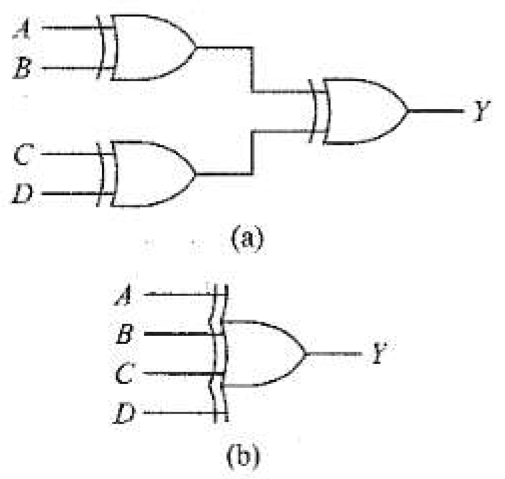


Fig. 4.31 Four-input exclusive OR gate

Table 4.7 4-Input Exclusive-OR Gate

Comment	*A*	*B*	*C*	*D*	*Y*
Even	0	0	0	0	0
Odd	0	0	0	1	1
Odd	0	0	1	0	1
Even	0	0	1	1	0
Odd	0	1	0	0	1
Even	0	1	0	1	0
Even	0	1	1	0	0
Odd	0	1	1	1	1
Odd	1	0	0	0	1
Even	1	0	0	1	0
Even	1	0	1	0	0
Odd	1	0	1	1	1
Even	1	1	0	0	0
Odd	1	1	0	1	1
Odd	1	1	1	0	1
Even	1	1	1	1	0

Any Number of Inputs

- Using 2-input exclusive-OR gates as building blocks, you can produce exclusive-OR gates with any number of inputs. For example, Fig. 4.32a shows a pair of exclusive-OR gates. There are 3 inputs and l output.
- If you analyze this circuit, you will find it produces an output 1 only when the 3-bit input has an odd number of ls. Figure 4.32b shows an abbreviated symbol for a 3-input exclusive-OR gate.

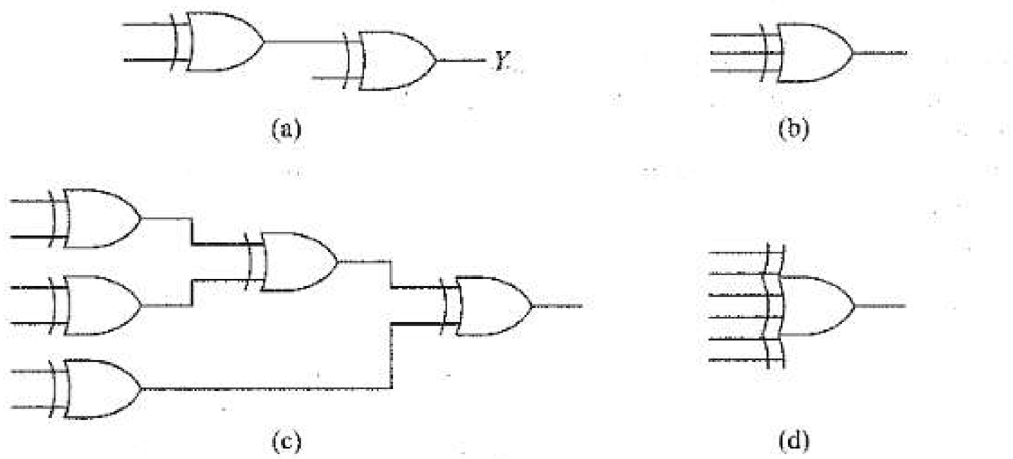


Fig. 4.32 Exclusive-OR gate with several inputs

PARITY GENERATORS AND CHECKERS

Even parity means an n-bit input has an even number of ls. For instance, 110011 has even parity because it contains four ls.

Odd parity means an n-bit input has an odd number of ls. For example, 110001 has odd parity because it contains three ls.

Here are two more examples:

1111 0000 1111 0011 even parity
1111 0000 1111 0111 odd parity

Parity Checker

- Exclusive-OR gates are ideal for checking the parity of a binary number because they produce an output 1 when the input has an odd number of 1s. Therefore, an even-parity in-put to an exclusive-OR gate produces a low output, while an odd-parity input produces a high output.

- For instance, Fig. 4.33 shows a 16-input exclusive-OR gate. A 16-bit number drives the input. The exclusive- OR gate produces an output 1 because the input has odd parity (an odd number of ls). If the 16-bit input
changes to another.

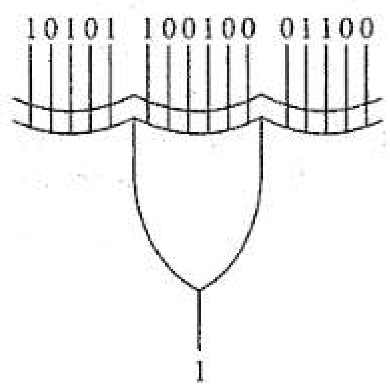


Fig. 4.33 **Exclusive-OR gate with 16 inputs**

Parity Generation

- In a computer, a binary number may represent an instruction that tells the computer to add, subtract, and so on; or the binary number may represent data to be processed like a number, letter, etc.
- In either case, you sometimes will see an extra bit added to the original binary number to produce a new binary number with even or odd parity.

For instance, Fig. 4.34 shows this 8-bit binary number:

$$X_7X_6X_5X_4 \quad X_3X_2X_1X_0$$

Suppose this number equals 0100 0001. Then, the number has even parity, which means the exclusive-OR gate produces an output of 0. Because of the inverter,

$$X_8 = 1$$

and the final 9-bit output is 1 01000001. Notice that this has odd parity.

Suppose we change the 8-bit input to O110 0001. Now, it has odd parity. In this case, the exclusive-OR gate produces an output 1. But the inve1ter produces a 0, so that the final 9-bit output is O O110 0001. Again, the final output has odd parity.

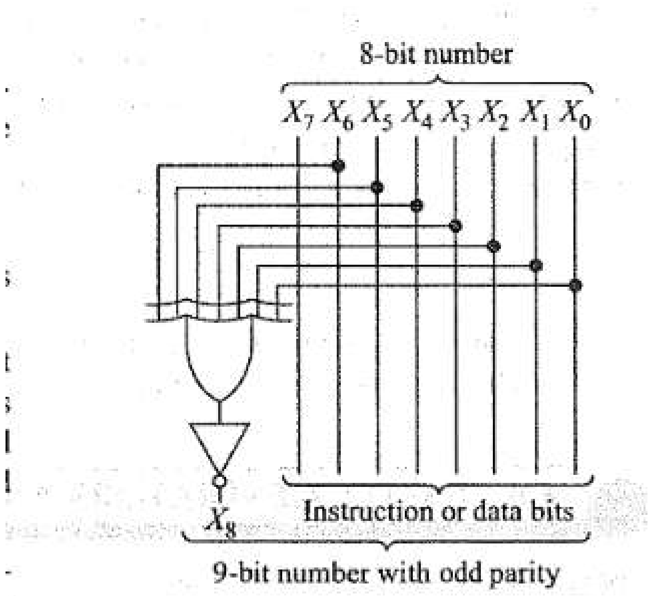

8-bit number
X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0
X_8
Instruction or data bits
9-bit number with odd parity

Fig. 4.34 Odd-parity generation

Application

- What is the practical application of parity generation and checking? Because of transients, noise, and other disturbances, 1-bit errors sometimes occur when binary data is transmitted over telephon,' lines or other communication paths. One way to check for errors is to use an odd-parity generator at the iansmitting end and an odd-parity checker at the receiving end. If no 1-bit errors occur in transmission, the received data will have odd parity. But if one of the transmitted bits is changed by noise or any other disturbance, the received data will have even parity.

For instance, suppose we want to send 0100 0011. With an odd-parity generator like Fig. 4.34, the data to be transmitted will be O O100 0011.

This data can be sent over telephone lines to some destination. If no errors occur in transmission, the odd-parity checker at the receiving end will produce a high output, meaning the received numqer has odd parity. On the other hand, if a 1-bit error does creep into the transmitted data, the odd-parity checker will have a low output, indicating the received data is invalid.

One final point should be made. Errors are rare to begin with. When they do occur, they are usually 1-bite1rnrs. This is why the method describedalmosthere catches all of the errors that occur in transmitted data.

The 74180

- Figure 4.35 shows the pinout diagram for a 74180, which is a TTL parity generator-checker. The input data bits are X_7 to X_0; these bits may have even or odd parity. The even input (pin 3) and the odd input (pin 4) control the operation of the chip as shown in Table 4.8. The symbol I stands for *summation*
- . In the left input column of Table 4.8, I of *H's* (highs) refers to the parity of the input dataX_7 toX_0. Depending on how you set up the values of the even and odd inputs, the I even and I odd outputs may be low or high.
- For instance, suppose even input is high and odd input is low. When the input data has even parity (the
- first entry of Table 4.8), the I even output is high and the I odd output is low. Whendatathe input has odd parity, the I even output is low and the 1: odd output is high.

generator,the delete inverter.)

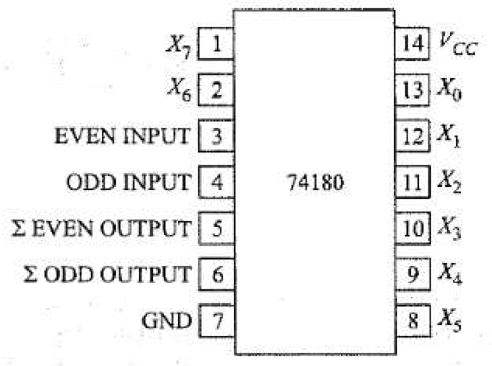
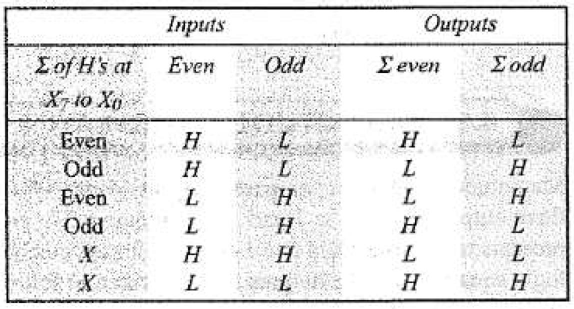


Fig. 4.35 Pinput diagram of 74180

Table 4.8 74180 Truth Table



Inputs			Outputs	
Σ of H's at X_7 to X_0	*Even*	*Odd*	*Σ even*	*Σ odd*
Even	*H*	*L*	*H*	*L*
Odd	*H*	*L*	*L*	*H*
Even	*L*	*H*	*L*	*H*
Odd	*L*	*H*	*H*	*L*
X	*H*	*H*	*L*	*L*
X	*L*	*L*	*H*	*H*

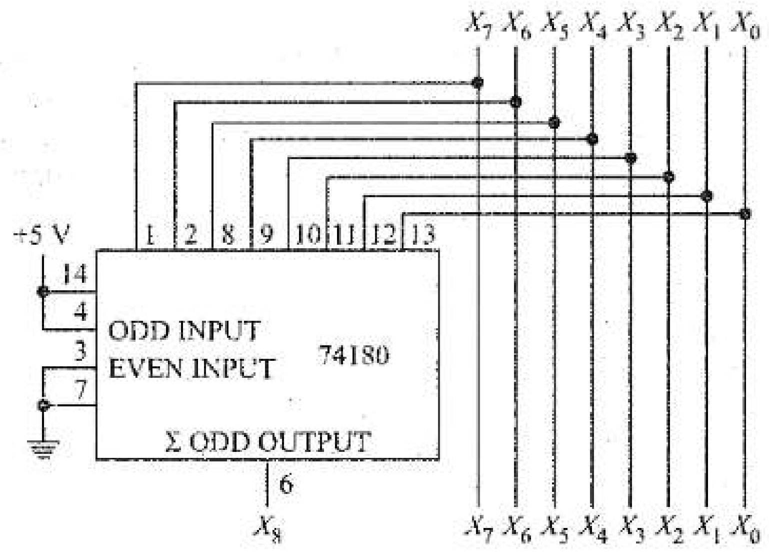


Fig. 4.36 Using a 74180 to generate odd parity

UNIT –III

NUMBER SYSTEM & CODE

BINARY NUMBER SYSTEM

- The *binary number system* is a system that uses only the digits 0 and 1 as codes. All other digits (2 to 9) are thrown away.
- To represent decimal numbers and letters of the alphabet with the binary code, you have to use different strings of binary digits number for each or letter. The idea is similar to the Morse code, where strings of dots and dashes are used to code all numbers and letters. What follows is a discussion of decimal and binary counting.

Decimal Odometer

- To understand how to count with binary numbers, it helps to review how an odometer (miles indicator of a car) counts with decimal numbers.
- When a car is new, its odometer starts with00000 After 1 km the reading becomes 00001

  Successive kms produce 00002, 00003, and so on, up to 00009
- A familiar thing happens at the end of the tenth km. When the units wheel turns from 9 back to 0, a tab on this wheel forces the tens wheel to advance by 1. This is why the numbers change to00010

Reset-and-Carry

- The units wheel has reset to 0 and sent a carry to the tens wheel. Let's call this familiar action *reset and carry* The other wheels of an odometer also reset and carry. For instance, after 999 kms the odometer shows 00999

- What does the next km do? The units wheel resets and carries, the tens wheel resets and carries, the hundreds wheel resets and carries, and the thousands wheel advances by 1, to get 01000

Binary Odometer

- Visualize a binary odometer as a device whose wheels have only two digits, 0 and 1. When each wheel turns, it displays 0, then 1, then back to 0, and the cycle repeats. A four-digit binary odometer starts with

  0000 (zero)

  After 1 mile, it indicates

  0001 (one)

  The next mile forces the units wheel to reset and carry, so the numbers change to

  0010 (two)

The third mile results in0011 (three)

After 4 miles, the units wheel resets and carries, the second wheel resets d carries, and third the wheel advances by 1:

100 (four)

Table 5.1 shows all the binary numbers from OOOO to 1111, equivalent to decimal O to 15 The word *bit* is the abbreviation for binary digit. Table 5.1 is a list of 4-bit number from 0000 to 1111. When a binary number has 4 bits, it is sometimes called a *nibble.* Table 5.1 shows 16 nibbles (0000 to llll).

- A binary number with is 8 bits known as a *byte;* this has become the basic unit of data used in computers.For now memorise these definitions:

bit =X

nibble = XXXX

byte = XXXXXXXX

where the X may be a 0 or a 1.

BINARY-TO-DECIMAL CONVERSION

Positional Notation and Weights

- We can express any decimal *integer* (a whole number) in units, tens, hundreds, thousands, and so on. For instance, decimal number 2945 may be written as

$$2945 = 2000 + 900 + 40 + 5$$

- In powers of 10, this becomes

$$2945 = 2(10^3) + 9(10^2) + 4(10^1) + 5(10^\circ)$$

- The decimal number system is an example of *positional notation*, each digit position has a *weight* or value. With decimal numbers, the weights are units, tens, hundreds, thousands, so and on. The sum of all the digits multiplied by their weights gives the total amount being represented.
- In the foregoing example, the 2 is multiplied by a weight of 1000, the 9 by a weight of 100, the 4 by a weight of 10, and the 5 by weight of 1; the total is

$$2000 + 900 + 40 + 5 = 2945$$

Binary Weights

- For instance, binary number 111 becomes

$$111=100+10+1 \qquad (5.1)$$

In decimal numbers, this may be rewritten as

$$7=4+2+1 \qquad (5.2)$$

- Writing a binary number as shown in Eq. (5 .1) is the same as splitting its decimal equivalent into units, 2s, and 4s as indicated by Eq. (5.2).
- In other words, each digit position in a binary number has a weight. The least significant digit (the one on the right) has a weight of 1.
  - The second position from the right has a weight of 2; the next, 4; and then 8, 16, 32, and so forth. These weights are in ascending powers of 2; therefore, we can write the foregoing equation as

$$7= 1(2^2)+ 1(2^1)+ 1(2^\circ)$$

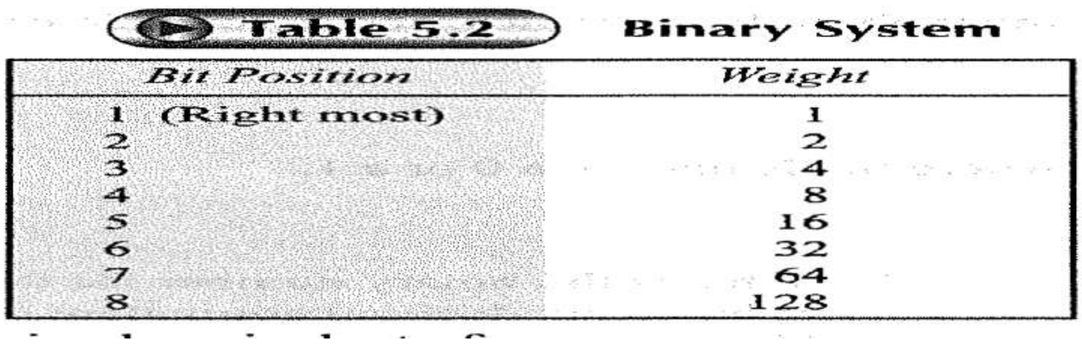


Table 5.2 Binary System

Bit Position	*Weight*
1 (Right most)	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128

- Whenever you look at a binary number, you can find its decimal equivalent as follows:

1. When there is a 1 in a digit position, add the weight of that position.

2. When there is a 0 in a digit position, disregard the weight of that position.

- For example, binary number 101 has a decimal equivalent of

  4+0+1=5

- As another example, binary number 1101 is equivalent to

  8+4+0+1=13

- Still another example is 1100 l, which is equivalent to

  16 + 8 + 0 + 0 + 1 = 25

Streamlined Method

- We can streamline binary-byte-decimal conversion the following procedure:

1. Write the binary number.

2. Directly under the binary number write 1, 2, 4, 8, 16 ... , working from right to left.
3. If a zero appears in a digit position, cross out the decimal weight for that position.

4. Add the remaining weights to obtain the decimal equivalent.

- As an example of this approach, let us convert binary 101 to its decimal equivalent:

STEP 1 101

STEP 2 4 2 1

STEP 3 4 2 1

STEP4 4 + 1 = 5

- As another example, notice how quickly 10101 is converted to its decimal equivalent:

1 0 1 0 1

16 8 4 2 1 $\rightarrow$ 21

Fractions

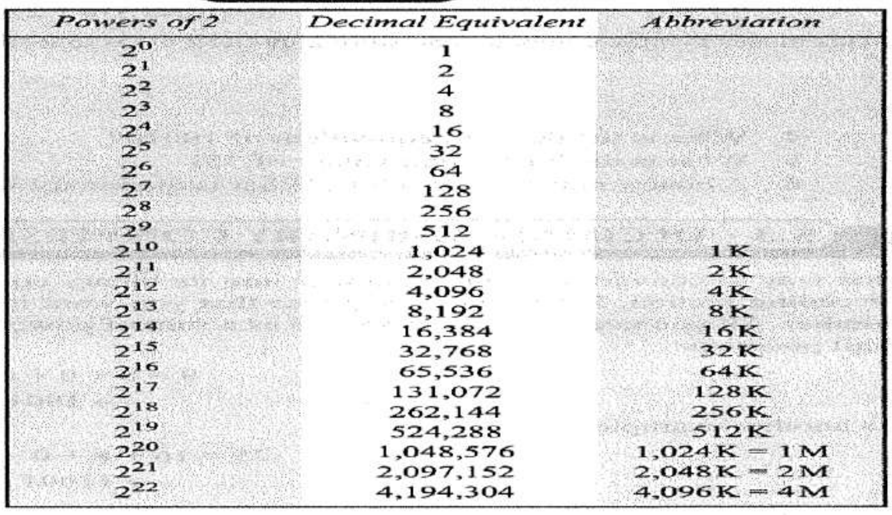
- For instance, what is the decimal equivalent of0.101? In this case, the weights of digit positions to the right of the binary point are given by ½,1/4,1/8, 1/16 and so on. In powers of 2, the weights are

2^{-1} 2^{-2} 2^{-3} 2^{-4} etc.

or in decimal form:

0.5 0.25 0.125 0.0625 etc.

Table 5.3 Powers of 2



Powers of 2	*Decimal Equivalent*	*Abbreviation*
2^0	1	
2^1	2	
2^2	4	
2^3	8	
2^4	16	
2^5	32	
2^6	64	
2^7	128	
2^8	256	
2^9	512	
2^{10}	1,024	1K
2^{11}	2,048	2K
2^{12}	4,096	4K
2^{13}	8,192	8K
2^{14}	16,384	16K
2^{15}	32,768	32K
2^{16}	65,536	64K
2^{17}	131,072	128K
2^{18}	262,144	256K
2^{19}	524,288	512K
2^{20}	1,048,576	1,024K = 1M
2^{21}	2,097,152	2,048K = 2M
2^{22}	4,194,304	4,096K = 4M

- Here is an example. Binary fraction 0.101 has a decimal equivalent of 0.1 0

  10.5 + 0+ 0.125 = 0625

- Another example, the decimal equivalent of 0.1101 is

0.1 1 0 1 0.5 + 0.25 + 0 + 0.0625 = 0.8125

Mixed Numbers

- For *mixed* numbers (numbers with an integer and a fractional part), handle each part according to the rules just developed.
- The weights for a mixed number are

etc. 2^3 2^2 2^1 2^0 . 2^{-1} 2^{-2} 2^{-3} etc.

↑

Binary point

- For future reference, Table 5.3 lists powers of 2 and their decimal Equivalents and the numbers of K and M. The abbreviation K stands for 1024.Therefore, lK means 1024. 2K stands for 2048, 4K represents 4096, and so on. The abbreviation M stands for 1,048,576, which is equivalent to 1024K (1024 x 1024 = 1,048,576).
- A memory chip that stores 4096 bits is called a "4K memory." A digital device might have a memory capacity of 4,194,304 bytes. This would be referred to as a "4-megabyte (Mb) memory."

DECIMAL-TO-BINARY CONVERSION

- One way to convert a decimal number into its binary equivalent is to reverse the process described in the preceding section. For instance, suppose that you want to convert decimal 9 into the corresponding binary number.
- All you need to do is express 9 as a sum of powers of 2, and then write 1s and Os in the appropriate digit positions:

$$9=8+0+0+1 \rightarrow 1001$$

- As another example:

$$25 = 16 + 8 + 0 + 0 + 1 \rightarrow 11001$$

Double Dabble

- A popular way to convert decimal numbers to binary numbers is the *double-dabble method.*
- In the double-dabble method you progressively divide the decimal number by 2, writing down the remainder after each division. The remainders, taken in reverse order, form the binary number.
- The best way to understand the method is to go through an example step by is step. Here how to convert decimal 13 to its binary equivalent

Step 1 Divide 13 by 2, writing your work like this:

$$\begin{array}{rl} 6 & \\ 2\overline{)13} & 1 \rightarrow \text{(first remainder)} \end{array}$$

The quotient is 6 with a remainder of 1.

Step 2 Divide 6 by 2 to get

$$\begin{array}{rl} 3 & \\ 2\overline{)6} & 0 \rightarrow \text{(second remainder)} \\ 2\overline{)13} & 1 \rightarrow \text{(first remainder)} \end{array}$$

This division gives 3 with a remainder of 0.

Step 3 Again you divide by 2:

$$\begin{array}{rl} 1 & \\ 2\overline{)3} & 0 \rightarrow \text{(second remainder)} \\ 2\overline{)6} & 1 \rightarrow \text{(first remainder)} \\ 2\overline{)13} & 1 \rightarrow \text{(first remainder)} \end{array}$$

Here you get a quotient of 1 with a remainder of 1.

Step 4 One more division gives

$$\begin{array}{rcl} 0 & & \\ 2\overline{)1} & 1 & \rightarrow \text{(fourth remainder)} \\ 2\overline{)3} & 1 & \\ 2\overline{)6} & 0 & \text{Read down} \\ 2\overline{)13} & 1 & \downarrow \end{array}$$

- In this final division 2 does not divide into 1; thus, the quotient is 0 with a remainder of 1.

- Whenever you arrive at a quotient of O with a remainder of 1, the conversion is finished. The remainders when read downward give the binary equivalent. In this example, binary 1101 is equivalent to decimal 13.

$$\begin{array}{rcl} 0 & 1 & \\ 1 & 1 & \\ 3 & 0 & \text{Read down} \\ 6 & 1 \downarrow & \\ 2\overline{)13} & & \end{array}$$

Fractions

- As far as fractions are concerned, you *multiply* by 2 and record a carry in the integer position. The carries read downward are the binary fraction. As an example, 0.85 converts to binary as follows:

$0.85 \times 2 = 1.7 = 0.7$ with a carry of 1
$0.7 \times 2 = 1.4 = 0.4$ with a carry of 1
$0.4 \times 2 = 0.8 = 0.8$ with a carry of 0 Read down
$0.8 \times 2 = 1.6 = 0.6$ with a carry of 1
$0.6 \times 2 = 1.2 = 0.2$ with a carry of 1
$0.2 \times 2 = 0.4 = 0.4$ with a carry of 0 $\downarrow$

- Reading the carries downward gives binary fraction 0.110110. In this case, we stopped the conversion process after getting six binary digits.
- Because of this, the answer is an approximation. If more accuracy is needed, continue multiplying by 2 until you have as many digits as necessary for your application.

Useful Equivalents

- The table has an important property that you should be aware of. Whenever a binary number has all 1s (consists of only 1s), you can find its decimal equivalent with this formula:

$$\text{Decimal} = 2^{n} - 1$$

where *n* is the number of bits. For instance, 1111 has 4 bits; therefore, its decimal equivalent is

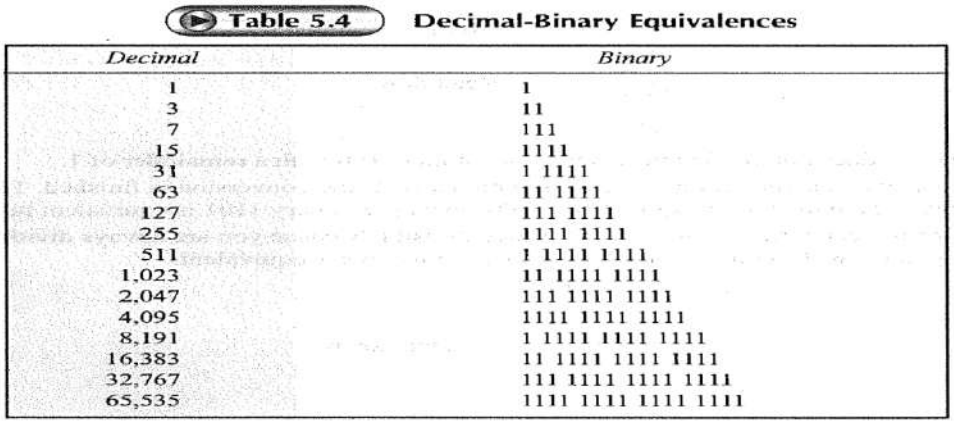


Table 5.4 Decimal-Binary Equivalences

Decimal	*Binary*
1	1
3	11
7	111
15	1111
31	1 1111
63	11 1111
127	111 1111
255	1111 1111
511	1 1111 1111
1,023	11 1111 1111
2,047	111 1111 1111
4,095	1111 1111 1111
8,191	1 1111 1111 1111
16,383	11 1111 1111 1111
32,767	111 1111 1111 1111
65,535	1111 1111 1111 1111

- As another example, 1111 1111 has 8 bits, so

$$\text{Decimal} = 2^{8} - 1 = 256 - 1 = 255$$

BCD-8421 and BCD-2421 Code

- Binary Coded Decimal (BCD) refers to representation of digits 0-9 in decimal system by 4-bit unsigned binary numbers.

- The usual method is to follow 8421 encoding which employs conventional route of weight placements like 8 representing the weight of the 4th. place (as2^{4-1}= 8), 4, i.e. 2^{3-1} of the 3rd place, 2, i.e. 2^{2-1} of the 2nd place and 1, i.e. 2^{1-1} of the 1st place.
- The 2421 code is similar to 8421 code except for the fact that the weight

assigned to 4th place is 2 and not 8. The decimal numbers 0-9 in these two codes then can be represented as shown in Table 5.5.

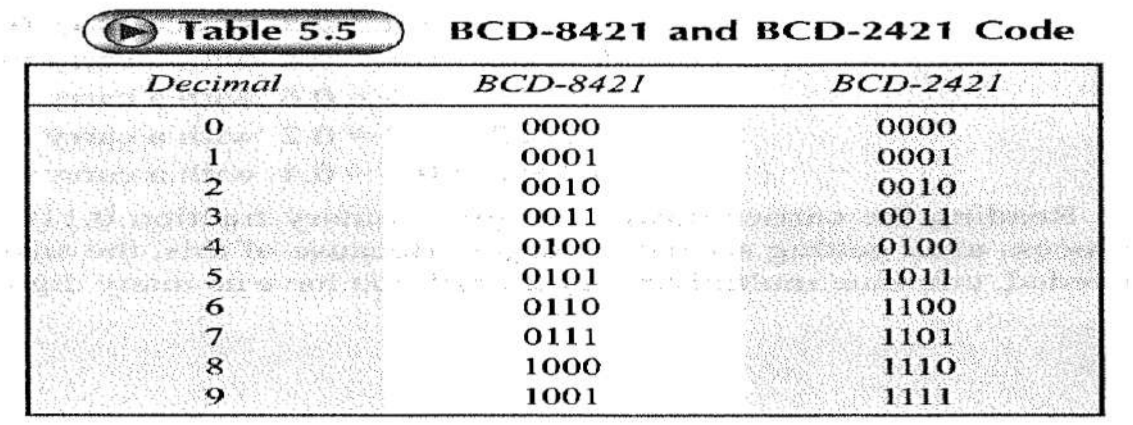


Table 5.5 BCD-8421 and BCD-2421 Code

Decimal	BCD-8421	BCD-2421
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	1011
6	0110	1100
7	0111	1101
8	1000	1110
9	1001	1111

- As an example,number decimal 29 in BCD-8421 is written as 00101001 (0010 representing 2 and 1001 representing 9) while in BCD-2421, it is written as 00101111 (0010 representing 2 and 1111 representing 9).

5.4 OCTAL NUMBERS

- The base of a number system equals the number of digits it uses. The decimal number system has a base of 10 because it uses the digits O to 9.
- The binary number system has a base of 2 because it uses only the digit
  0 and 1. The *octal number system* has a base of 8.
- Although we can use any eight digits, it is customary to use the first eight decimal digits:

0, 1,2,3,4,5,6, 7

- (There is no 8 or 9 in the octal number code.) These digits, 0 through 7, have exactly the same physical meaning as decimal symbols; that is, 2 stands for.., 5 symbolizes ....., and so on.

Octal Odometer

- The easiest way to learn how to count in octal numbers is to use an *octal odometer*. This hypothetical device is similar to the odometer of a car, except that each display wheel contains only eight digits, numbered 0 to 7.

- When a wheel turns from 7 back to 0, it sends a carry to the next-higher wheel.
  Initially, an octal odometer shows

0000	(zero)

- The next 7 kms produces readings of

0001	(one)
0002	(two)
0003	(three)
0004	(four)
0005	(five)
0006	(six)
0007	(seven)

- At this point, the least-significant wheel has run out of digits. Therefore, the next km forces a reset and carry to obtain

0010	(eight)

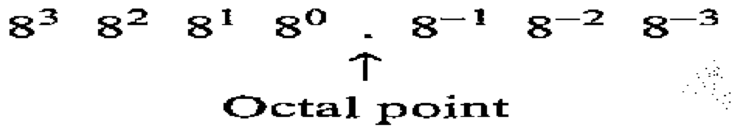
- The next 7 kms produces these readings: 0011, 0012, 0013, 0014, 0015, 0016, and 0017. Once again, the least-significant wheel has run out of digits. So the next km results in a reset and carry:

0020	(sixteen)

- Subsequent kms produce readings of 0021, 0022, 0023, 0024, 0025, 0026, 0027, 0030, 0031, and so on.

Octal-to-Decimal Conversion

- How do we convert octal numbers to decimal numbers? In the octal number system each digit position corresponds to a power of 8 as follows:



- Therefore, to convert from octal to decimal, multiply each octal digit by its weight and add the resulting products. Note that s^0 = I.
- For instance, octal 23 converts to decimal like this:

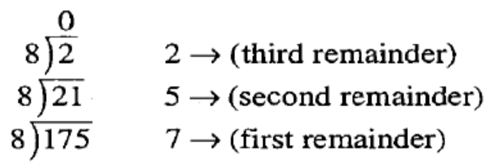
$2(8^{1}) + 3(8°) = 16 + 3 = 19$

- Here is another example. Octal 257 converts to

$2(8^{1})+5(8^{1})+7(8°)= 128+40+7= 175$

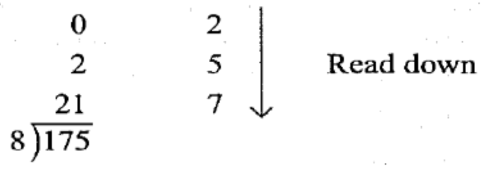
Decimal-to-Octal Conversion

- How do you convert direction, in the opposite that is, from decimal to octal? *Octal dabble,* a method similar to double dabble, is used with octal numbers. Instead of dividing by 2 (the base of binary numbers), you divide by 8 (the base of octal numbers) writing down the remainders after each division.
- The remainders in reverse order form the octal number. As an example, convert decimal 17 5 as follows:



0
8)2 2 → (third remainder)
8)21 5 → (second remainder)
8)175 7 → (first remainder)

You can condense these steps by writing



Thus decimal 175 is equal to octal 257.

Fractions

- With decimal fractions, multiply instead of divide, writing the carry into the integer position. An example of this is to convert decimal 0.23 into an octal fraction.

$$\begin{array}{lll} 0.23 \times 8 = 1.84 = 0.84 & \text{with a carry of } 1 & \\ 0.84 \times 8 = 6.72 = 0.72 & \text{with a carry of } 6 & \text{Read down} \\ 0.72 \times 8 = 5.76 = 0.76 & \text{with a carry of } 5 \downarrow & \\ \quad\quad\quad \text{etc.} & & \end{array}$$

- The carries read downward give the octal fraction 0.165. We terminated after three places; for more accuracy, we would continue multiplying to obtain more octal digits.

Octal-to-Binary Conversion

- Because 8 (the base of octal numbers) is the third power of 2 (the base of binary numbers), you can convert from octal to binary as follows: change each octal digit to its binary equivalent.
- For instance, change octal 23 its to binary equivalent as follows:

$$\begin{array}{cc} 2 & 3 \\ \downarrow & \downarrow \\ 010 & 011 \end{array}$$

- Here, each octal digit converts to its binary equivalent (2 becomes 010, and 3 becomes 011). The binary equivalent of octal 23 is 010 011, or 010011. Often, a space is left between groups of 3 bits; this makes it easier to read the binary number.

As another example, octal 3574 converts to binary as follows:

$$\begin{array}{cccc} 3 & 5 & 7 & 4 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 011 & 101 & 111 & 100 \end{array}$$

Hence binary 011101111100 is equivalent to octal 3574. Notice how much easier the binary number is to read if we leave a space between groups of 3 bits: 011 101 111 100.

Mixed octal numbers are no problem. Convert each octal digit to its equivalent binary value. Octal 34.562 becomes

$$\begin{array}{cccccc} 3 & 4 & . & 5 & 6 & 2 \\ \downarrow & \downarrow & & \downarrow & \downarrow & \downarrow \\ 011 & 100 & . & 101 & 110 & 010 \end{array}$$

Binary-to-Octal Conversion

- Conversion from binary to octal is a reversal of the foregoing procedures. Simply remember to group the bits in threes, starting at the binary point; then convert each group of three to its octal equivalent (Os are added at each end, if necessary). For instance, binary number 1011.01101 converts as follows:

$$\begin{array}{lcccc} 1011.01101 \rightarrow & 001 & 011. & 011 & 010 \\ & \downarrow & \downarrow & \downarrow & \downarrow \\ & 1 & 3 & 3 & 2 \end{array}$$

- Start at the binary point and, working both ways, separate the bits into groups of three.
- When necessary, as in this case, add 0s to complete the outside groups. Then convert each group of three into its binary equivalent. Therefore:

1011.01101 = 13.32

- The simplicity of converting octal to binary and vice versa has many advantages in digital work.
- For one thing, getting information into and out of a digital system requires less circuitry because it is easier to read and print out octal numbers than binary numbers.
- Another advantage is that large decimal numbers are more easily converted to binary if first converted to octal and then to binary.

5.5 HEXADECIMAL NUMBERS

- *Hexadecimal numbers* are used extensively in microprocessor work. To begin with, they are much shorter than binary numbers. This makes them easy to write and remember. Furthermore, you can mentally convert them to binary whenever necessary.

- The hexadecimal number system has a base of 16. Although any 16 digits may be used, everyone uses 0 to 9 and A to F as shown in Table 5.6. In other words, after reaching 9 in the hexadecimal system, you continue counting as follows:

A,B,C,D,E,F

Hexadecimal Odometer

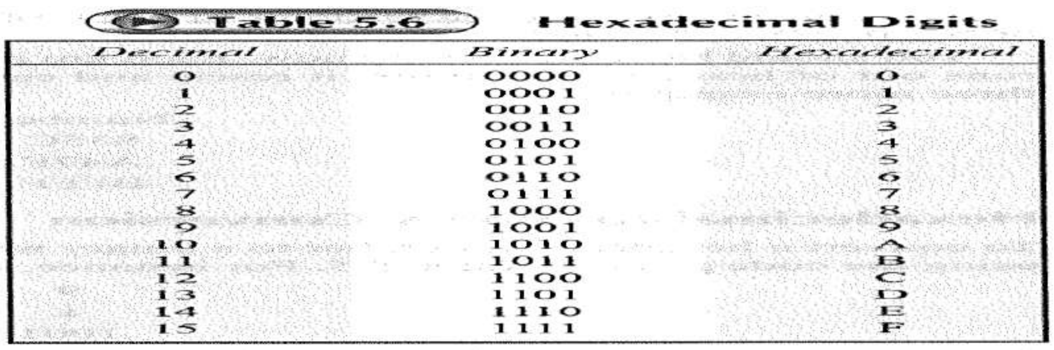


Table 5.6 Hexadecimal Digits

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

- The easiest way to learn how to count in hexadecimal numbers is to use a *hexadecimal odometer*.
- This hypothetical device is similar to the odometer of a car, except that each display wheel has 16 digits, numbered 0 to F. When a wheel turns from F back to 0, it sends a carry to the wheel next. Higher Initially, a hexadecimal odometer shows

0000 (zero)

The next 9 kms produces readings of

0001	(one)
0002	(two)
0003	(three)
0004	(four)
0005	(five)
0006	(six)
0007	(seven)
0008	(eight)
0009	(nine)

The next 6 kms gives

000A	(ten)
000B	(eleven)
000C	(twelve)
000D	(thirteen)
000E	(fourteen)
000F	(fifteen)

- At this point, the least-significant wheel has run out of digits. Therefore, the next km forces a reset and carry to obtain

  0010 (sixteen)

- The next 15 kms produces these readings: 0011, 0012, 0013, 0014, 0015, 0016, 0017, 0018, 0019, 001A,00IB, 00lC, 001D, 00lE, and 00lF. Once again, the least significant wheel has run out of digits. So, the nextkm results in reset a and carry:

  0020 (thirty-two)

- Subsequent kms produce readings of 0021,0022,0023,0024,0025,0026,0027,0028,0029,002A,002B, 002C, 002D, 002E, and 002F.
- For instance, here are three more examples:

Number	Next number
835C	835D
A47F	A480
BFFF	cooo

Hexadecimal-to-Binary Conversion

➢ To convert a hexadecimal number to a binary number, convert each hexadecimal digit to its 4-bit equivalent using the code given in Table 5.5. For instance, here's how to 9AF converts binary:

$$\begin{array}{ccc} 9 & A & F \\ \downarrow & \downarrow & \downarrow \\ 1001 & 1010 & 1111 \end{array}$$

As another example, C5E2 converts like this:

$$\begin{array}{cccc} C & 5 & E & 2 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 1100 & 0101 & 1110 & 0010 \end{array}$$

Binary-to-Hexadecimal Conversion

➢ To convert in the opposite direction, from binary to hexadecimal, again use the code from Table 5.5. Here are two examples. Binary 1000 1100 converts as follows:

$$\begin{array}{cc} 1000 & 1100 \\ \downarrow & \downarrow \\ 8 & C \end{array}$$

Binary 1110 1000 1101 0110 converts like this:

$$\begin{array}{cccc} 1110 & 1000 & 1101 & 0110 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ E & 8 & D & 6 \end{array}$$

➢ In both these conversions, we start with a binary number and wind up with the equivalent hexadecimal number.

Hexadecimal-to-Decimal Conversion

➢ How do we convert hexadecimal numbers to decimal numbers? In the hexadecimal number system each digit position corresponds to a power of 16. The weights of the digit positions in a hexadecimal number are as follows

$$16^3 \quad 16^2 \quad 16^1 \quad 16^0 \quad \underset{\substack{\uparrow \\ \text{Hexadecimal point}}}{.} \quad 16^{-1} \quad 16^{-2} \quad 16^{-3}$$

- Therefore, to convert from hexadecimal to decimal, multiply each hexadecimal digit by its weight and add the resulting products. Note that 16° = 1.

- Here's an example. Hexadecimal F8E6to.39 converts decimal as follows:

F8E6 = F(16 ^{3}) + 8(16^2) + E(16 ^{1}) + 6(16°) + 3(16-^{1}) + 9(16-^{2})

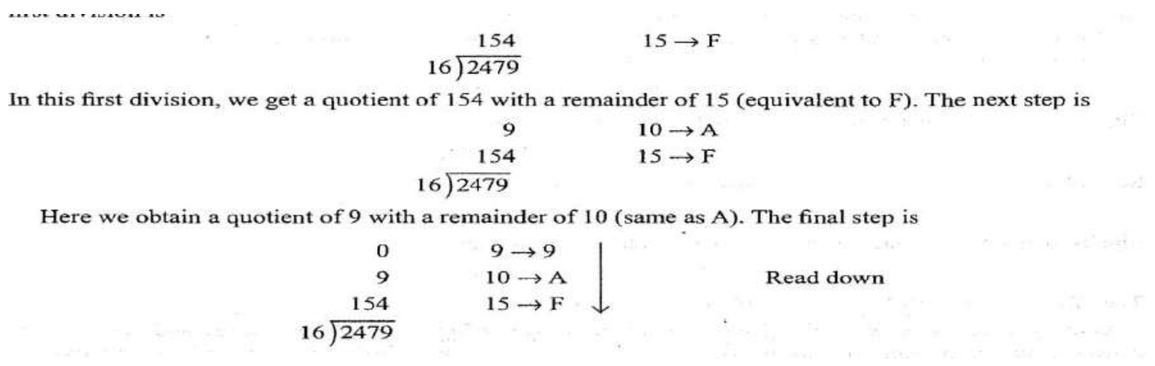
= 15(16^3) + 8(16^2) + 14(16^1) + 6(16°) + 3(16-^{1}) + 9(16-^{2})

= 61,440 + 2048 + 224 + 6 + 0.1875 + 0.0352

= 63,718.2227

Decimal-to-Hexadecimal Conversion

- One way to convert from decimal to hexadecimal is the *hex dabble.* The idea to is divide successively by 16, writing down the remainders. Here's a sample of how it's done. To convert decimal 2479 to hexadecimal, the first division is



first division is

$$\begin{array}{rl} 154 & \quad 15 \rightarrow F \\ 16\overline{)2479} & \end{array}$$

In this first division, we get a quotient of 154 with a remainder of 15 (equivalent to F). The next step is

$$\begin{array}{rl} 9 & \quad 10 \rightarrow A \\ 154 & \quad 15 \rightarrow F \\ 16\overline{)2479} & \end{array}$$

Here we obtain a quotient of 9 with a remainder of 10 (same as A). The final step is

$$\begin{array}{rll} 0 & \quad 9 \rightarrow 9 & \big\downarrow \\ 9 & \quad 10 \rightarrow A & \qquad \text{Read down} \\ 154 & \quad 15 \rightarrow F & \\ 16\overline{)2479} & & \end{array}$$

- Therefore, hexadecimal 9AF is equivalent to decimal 2479.

- Notice how similar hex dabble is to double dabble. Notice also that remainders greater than 9 have to be changed to hexadecimal digits (10 becomes A, 15 becomes F, etc.).

Using Appendix 1 *

- A typical microcomputer can store up to 65,535 bytes. The decimal addresses of these bytes are from 0 to 65,535. The equivalent binary addresses are from

0000 0000 0000 0000

```
111
1    1111    1111    1111
```

- The first 8 bits are called the *upper byte,* and the second 8 bits are the *lower byte.*

- If you have to do many conversions between binary, hexadecimal, and decimal, learn to use Appendix 1. It has four headings: *binary, hexadecimal, upper byte,* and *lower byte.*

- For any decimal number between O and 255, you would use the binary, hexadecimal, and lower byte columns. Here is the recommended way to use Appendix 1. Suppose you want to convert binary 0001 1000 to its decimal equivalent. First, mentally convert to hexadecimal:

0001 1000 $\rightarrow$ 18 (mental conversion)

- Next, look up hexadecimal 18 in Appendix 1 and read the corresponding decimal value from the lower-byte column:

18 $\rightarrow$ 24 (look up in Appendix 1)

For another example, binary 1111 0000 converts like this:

1111 0000 $\rightarrow$ F0 $\rightarrow$ 240

- The reason for mentally converting from binary to hexadecimal is that you can more easily locate a hexadecimal number in Appendix 1 than a binary number. Once have you the hexadecimal equivalent, you can read the lower-byte column to find the decimal equivalent.

- When the decimal number is greater than 255, you have to use both the upper byte and the lower byte in

- Appendix 1. For instance, suppose you want to convert this binary number to its decimal equivalent:

1110 1001 0 ll l 0100

- First, convert the upper byte to its decimal equivalent as follows:

$$1110\ 1001 \rightarrow \text{E9} \rightarrow 59{,}648 \quad \text{(upper byte)}$$

Second, convert the lower byte to its decimal equivalent:

$$0111\ 0100 \rightarrow 74 \rightarrow 116 \quad \text{(lower byte)}$$

Finally, add the upper and lower bytes to obtain the total decimal value:

$$59{,}648 + 116 = 59{,}764$$

- Therefore, binary 1110 1001 0111 0100 is equivalent to decimal 59,764.

THE ASCII CODE:

- To get information into and out of a computer, we need to use some kind of *alphanumeric* code (one for letters, numbers, and other symbols).
- At one time, manufacturers used their own alphanumeric codes, which led to all kinds of confusion. Eventually, industry settled on an input-output code known as the *American Standard Code for Information Interchange* (ASCII, pronounced ask'-ee).
- This code allows manufacturers to standardize computer hardware such as keyboards, printers, and video displays.

Using the Code

- The ASCII code is a 7-bit code whose format is

$$X_6X_5X_4X_3X_2X_1X_0$$

where each *Xis* a O or a 1. Use Table 5 .8 to find the ASCII code for the uppercase and lowercase letters of the alphabet and some of the most commonly used symbols. For example, the table shows that the capital letter A has an $X_6X_5X_4$ of 100 and *an* $X_3X_2X_1X_0$ ofOOOl. The ASCII code for A is, therefore,

1000001

For easier reading, we can leave a space as follows:

100 0001 (A)

The letter a is coded as

110 0001 (a)

More examples are

110 0010	(b)
1100011	(c)
110 0100	(d)

so and on.

- Also, study the punctuation and mathematical symbols. Some examples are

010 0100	($)
010 1011	(+)
011 1101	(=)

In Table 5.7, SP stands for space (blank). Hitting the space bar of an ASCII keyboard sends this into a microcomputer:

010 0000 (space)

Parity Bit

- The ASCII code is used for sending digital data over telephone lines. As mentioned in the preceding chapter, 1-bit errors may occur in transmitted data.
- To catch these errors, a parity bit is usually transmitted along with the original bits. Then a parity checker at the receiving end can test for even or odd parity, whichever parity has been prearranged between the sender and the receiver.
- Since ASCII code uses 7 bits, the addition of a parity bit to the transmitted data produces an 8-bit number in this format:

$$\underset{\substack{\uparrow \\ \text{Parity bit}}}{X_7}X_6X_5X_4 \quad X_3X_2X_1X_0$$

- This is an ideal length because most digital equipment is set up to handle bytes of data.

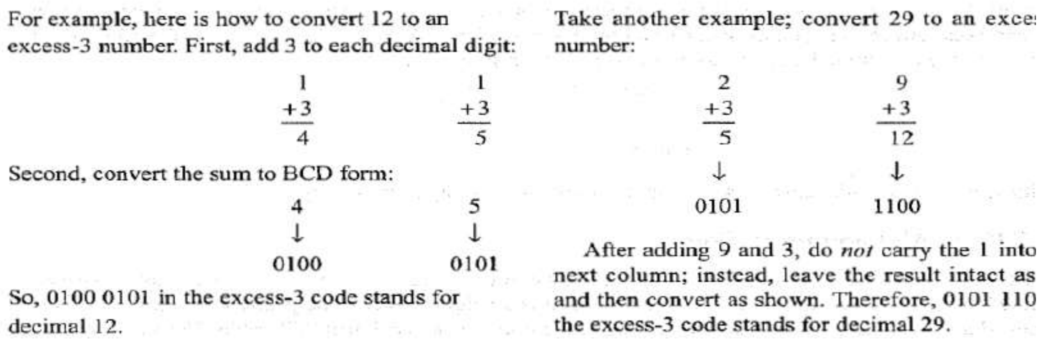
EBCDIC as Alphanumeric Code

- There exists few others but relatively less used alphanumeric codes. The EBCDIC is an abbreviation of Extended Binary Coded Decimal Interchange Code.

- It is an eight-bit code and primarily used in IBM make devices. Here, the binary codes ofletters and numerals come as an extension of BCD code. The bit assignments of EBCDIC are different from the ASCII but the character symbols are the same.

5.7 THE EXCESS-3 CODE

- The *excess-3 code* is an important 4-bit code sometimes used with binary-coded decimal (BCD) numbers. To convert any decimal number into its excess-3 fonn, add 3 to each decimal digit, and then convert the sum to a BCD number.



For example, here is how to convert 12 to an excess-3 number. First, add 3 to each decimal digit:

$$\begin{array}{r} 1 \\ +3 \\ \hline 4 \end{array} \qquad \begin{array}{r} 1 \\ +3 \\ \hline 5 \end{array}$$

Second, convert the sum to BCD form:

$$\begin{array}{c} 4 \\ \downarrow \\ 0100 \end{array} \qquad \begin{array}{c} 5 \\ \downarrow \\ 0101 \end{array}$$

So, 0100 0101 in the excess-3 code stands for decimal 12.

Take another example; convert 29 to an exce
number:

$$\begin{array}{r} 2 \\ +3 \\ \hline 5 \\ \downarrow \\ 0101 \end{array} \qquad \begin{array}{r} 9 \\ +3 \\ \hline 12 \\ \downarrow \\ 1100 \end{array}$$

After adding 9 and 3, do *not* carry the 1 into next column; instead, leave the result intact as and then convert as shown. Therefore, 0101 110 the excess-3 code stands for decimal 29.

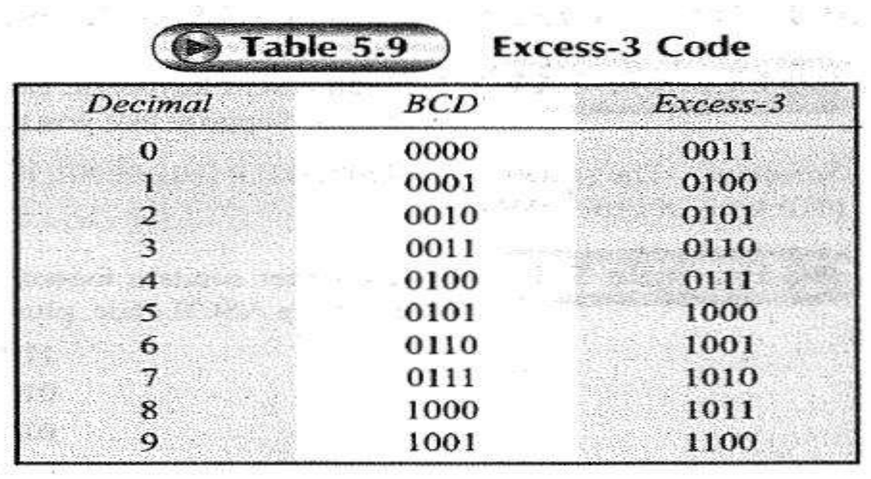


Table 5.9 Excess-3 Code

Decimal	*BCD*	*Excess-3*
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

- Table 5.9 shows the excess-3 code: In each case, the excess-3 code number is 3 greater than the BCD equivalent.
- Such coding helps in BCD arithmetic as 9's complement of any excess-3 coded number can be obtained simply by complementing each bit. Take for example decimal number 2.
- Its 9's complement is 9- 2 = 7. Excess-3 code of2 is 0101. Complementing each bit we get 1010 and its decimal equivalent is 7. To convert BCD to excess-3 we need an adder and for the reverse we need a subtractor.

- Incidentally, if you need an integrated circuit (IC) that converts from excess 3 to decimal, look at the data sheet of a 7443. This transistor-transistor logic (TTL) chip has four input lines for the excess-3 input and IO output lines for the decoded decimal output.

THE GRAY CODE

- The advantage of such coding will be understood from this example. Let an object move along a track and move from one zone to another. Let the presence of the object in one zone is sensed by sensors *ABC.*
- If consecutive zones are binary coded zone -then0 is represented by *ABC*= 000, zone- I by *ABC*= 001, zone-2 by *ABC*= 010 and so on, as shown in Fig. 5.la. Now consider, the object moves from zone-1 to zone-2.
- Both *BC* has to change to sense that movement. Suppose, sensor *B* (may be an electro-mechanical switch) reacts slightly late than sensor C.
- Then, initially *ABC*= 000 is sensed as if the object has moved in the other direction from zone-1 to zone-0. This problem can be more prominent if the object moves from zone-3 *(ABC*=011) to zone-4 *(ABC*= 100) when all three sensors has to change its value.
- Note that, if zones are gray coded (Fig. 5.lb) such problem does not appear as between two consecutive zones only one sensor changes its value.

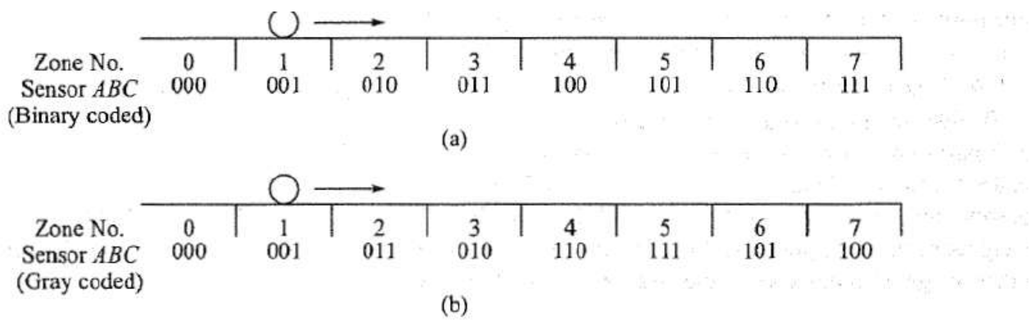


Fig. 5.1 Object moving along a track with sensors: (a) Binary coded, (b) Gray coded

- The disadvantage with gray code is that it is not good for arithmetic operation.
- However, comparing truth tables of binary coded numbers and gray coded numbers (Table 5.18) we can design binary to gray converter as shown in Fig. 5.2a and gray to binary converter as shown in Fig. 5.2b. Let's see how these circuits work by taking one example each.

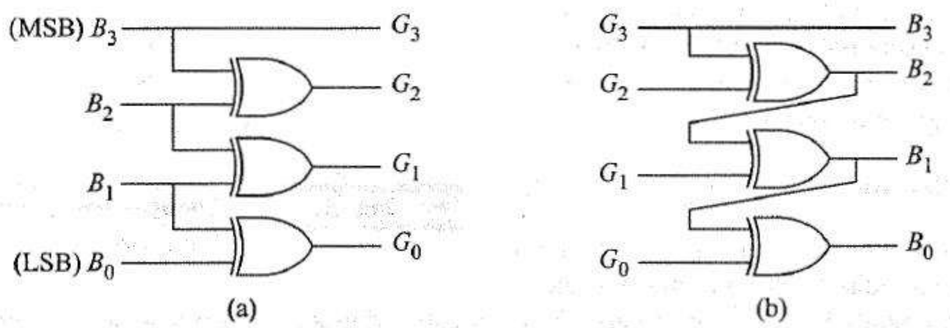


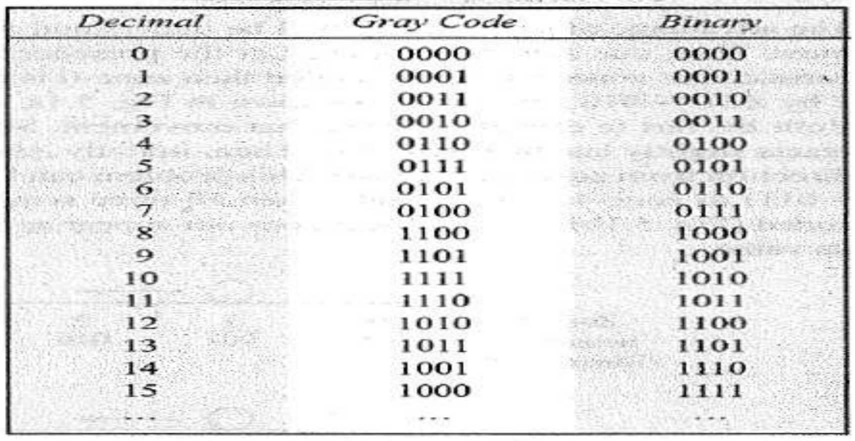
Fig. 5.2 (a) Binary to Gray converter, (b) Gray to Binary converter

Consider, a binary number $B_3B_2B_1B_0 = 1011$. Following the relation shown in Fig. 5.2a we get, $G_3 = B_3 = 1$, $G_2 = B_3 \oplus B_2 = 1 \oplus 0 = 1$, $G_1 = B_2 \oplus B_1 = 0 \oplus 1 = 1$ and $G_0 = B_1 \oplus B_0 = 1 \oplus 1 = 0$, i.e. $G_3G_2G_1G_0 = 1110$ and we can verify the same from truth table.

Similarly, for a gray coded number say, $G_3G_2G_1G_0 = 0111$ from Fig. 5.2b we get, $B_3 = G_3 = 0$, $B_2 = G_3 \oplus G_2 = 0 \oplus 1 = 1$, $B_1 = B_2 \oplus G_1 = 1 \oplus 1 = 0$ and $B_0 = B_1 \oplus G_0 = 0 \oplus 1 = 1$, i.e. $B_3B_2B_1B_0 = 0101$. Again

- Again this conversion can be verified from Table 5.10 that shows the *Gray code*, along with the corresponding binary numbers.
- Each Gray-code number differs from any adjacent number by a single bit. For instance, in going from decimal 7 to 8, the Gray-code numbers change from 0100 to 1100; these numbers differ only in the most significant bit. As another example, decimal numbers 13 and 14 are represented by Gray-code numbers 1011 and 1001; these numbers differ in only one digit position (the second position from the right).
- So, it is with the entire Gray code; every number dif-fers by only 1 bit from the preceding number.

Table 5.10 Gray Code



Decimal	Gray Code	Binary
0	0000	0000
1	0001	0001
2	0011	0010
3	0010	0011
4	0110	0100
5	0111	0101
6	0101	0110
7	0100	0111
8	1100	1000
9	1101	1001
10	1111	1010
11	1110	1011
12	1010	1100
13	1011	1101
14	1001	1110
15	1000	1111
...	...	...

- Besides the excess-3 and Gray codes, there are other binary-type codes. Appendix 5 lists some of these codes for future reference.
- Incidentally, the BCD code is sometimes referred to as the *8421 code* because the weights of the digit positions from left to right are 8, 4, 2, and 1. As shown in Appendix 5, there are many other weighted codes such as the 7421, 6311, 5421, and so on.

ARITHMETIC CIRCUITS

BINARY ADDITION

- Numbers represent physical quantities. Table 6.1 shows the decimal digits and the corresponding amount of pebbles.
- Digit 2 stands for two pebbles (..),five5for pebbles (.....), and so on. Addition represents the combining of physical quantities. For instance:

2+3=5

symbolizes the combining of two pebbles with three pebbles to obtain a total of five pebbles.
Symbolically, this is expressed

- + ••• = •••••

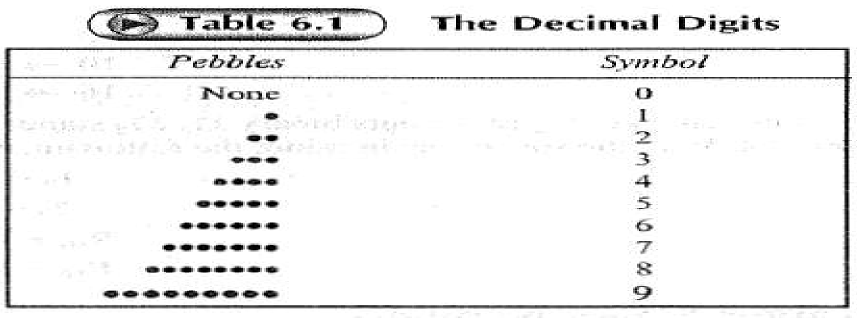


Table 6.1 The Decimal Digits

Pebbles	Symbol
None	0
•	1
••	2
•••	3
••••	4
•••••	5
••••••	6
•••••••	7
••••••••	8
•••••••••	9

Four Cases to Remember

- Computer circuits don't process decimal numbers;process they numbers binary. Before you can understand how a computer perfonns arithmetic, you have to learn how to numbers add binary.
- Binary addition is the key to binary subtraction, multiplication, and division. So, let's begin with the four most basic cases of binary addition:

$$0+0=0 \qquad (6.1)$$

$$0+1=1 \qquad (6.2)$$

$$1+0=1 \qquad (6.3)$$

$$1+1=10 \qquad (6.4)$$

Equation (6.1) is obvious; so are Eqs. (6.2) and (6.3) because they are identical to decimal addition.

The fourth case, however, may bother you. If so, you don't understand what Eq. (6.4) represents in the physical world.

Equation (6.4) represents the combining of one pebble and one pebble to obtain a total of two pebbles:

•+• = ••

- Since binary 10 stands for ••**,** the binary equation

$1 + 1 = 10$

makes perfect sense. From now on, remember that numbers, whether binary, decimal, octal, or hexadecimal are codes for physical amounts.

Subscripts

- The foregoing discussion brings up the idea of *subscripts.* Since we already have discussed four kinds of numbers (decimal, binary, octal, and hexadecimal), we have four different ways to code physical quantities. How do we know which code is being used? In other words, how can we tell when l O is a decimal, binary, octal, or hexadecimal number?

- On the other hand, if a discussion uses more than one type of number, it may be helpful for to use subscripts the base as follows:

$$\begin{array}{rl} 2 & \rightarrow \text{binary} \\ 8 & \rightarrow \text{octal} \\ 10 & \rightarrow \text{decimal} \\ 16 & \rightarrow \text{hexadecimal} \end{array}$$

  - For instance, 11_2 represents binary 11,23_8 stands for octal 23, 45_{10} for decimal 45, and $F4_{16}$ for hexadecimal F4. With the subscripts in mind, the following equations should make perfect sense:

$$\begin{array}{r} 1_2 + 1_2 = 10_2 \\ 7_8 + 1_8 = 10_8 \\ 9_{10} + 1_{10} = 10_{10} \\ F_{16} + 1_{16} = 10_{16} \end{array}$$

larger Binary Numbers

  - Column-by-column addition applies to binary as well as decimal numbers. For example, suppose you have this problem in binary addition:

$$\begin{array}{r} 11100 \\ +\,11010 \\ \hline ? \end{array}$$

Start by adding the least-significant column to get

$$\begin{array}{r} 11100 \\ +\,11010 \\ \hline 0 \end{array}$$

Next, add the bits in the second column as follows:

$$\begin{array}{r} 11100 \\ +\,11010 \\ \hline 10 \end{array}$$

The third column gives

$$\begin{array}{r} 11100 \\ +\,11010 \\ \hline 110 \end{array}$$

The fourth column results in

$$\begin{array}{rr} \text{Carry} \rightarrow & 1 \\ & 11100 \\ & +\,11010 \\ \hline & 0110 \end{array}$$

  - Notice the carry into the final column; this carry occurs because 1 + I = 10. As in decimal addition, you write down the O and carry the I to the next-higher column.

Finally, the last column gives

$$\begin{array}{rr} \text{Carry} \rightarrow & 1 \\ & 11100 \\ + & 11010 \\ \hline & 110110 \end{array}$$

In the last column, 1 + 1 + 1 = 10 + 1 = 11.

8-Bit Arithmetic

- That's all there is to binary addition. If you can remember the four basic rules, you can add column by column to find the sum of two binary numbers, regardless of how long they may be. In first-generation microcomputers (Apple II, TRS-80, etc.), addition is done on two 8-bit numbers such as

$$\begin{array}{rcc} & A_7A_6A_5A_4 & A_3A_2A_1A_0 \\ + & B_7B_6B_5B_4 & B_3B_2B_1B_0 \\ \hline & \multicolumn{2}{c}{?} \end{array}$$

- The most-significant bit (MSB) of each number is on the left, and the least-significant bit (LSB) is on the right. For the first number, A_7 is the MSB and *Ao* is the LSB. For the second number, B_7 is the MSB and *Bo* is the LSB.

BINARY SUBTRACTION

Let's begin with four basic cases of binary subtraction:

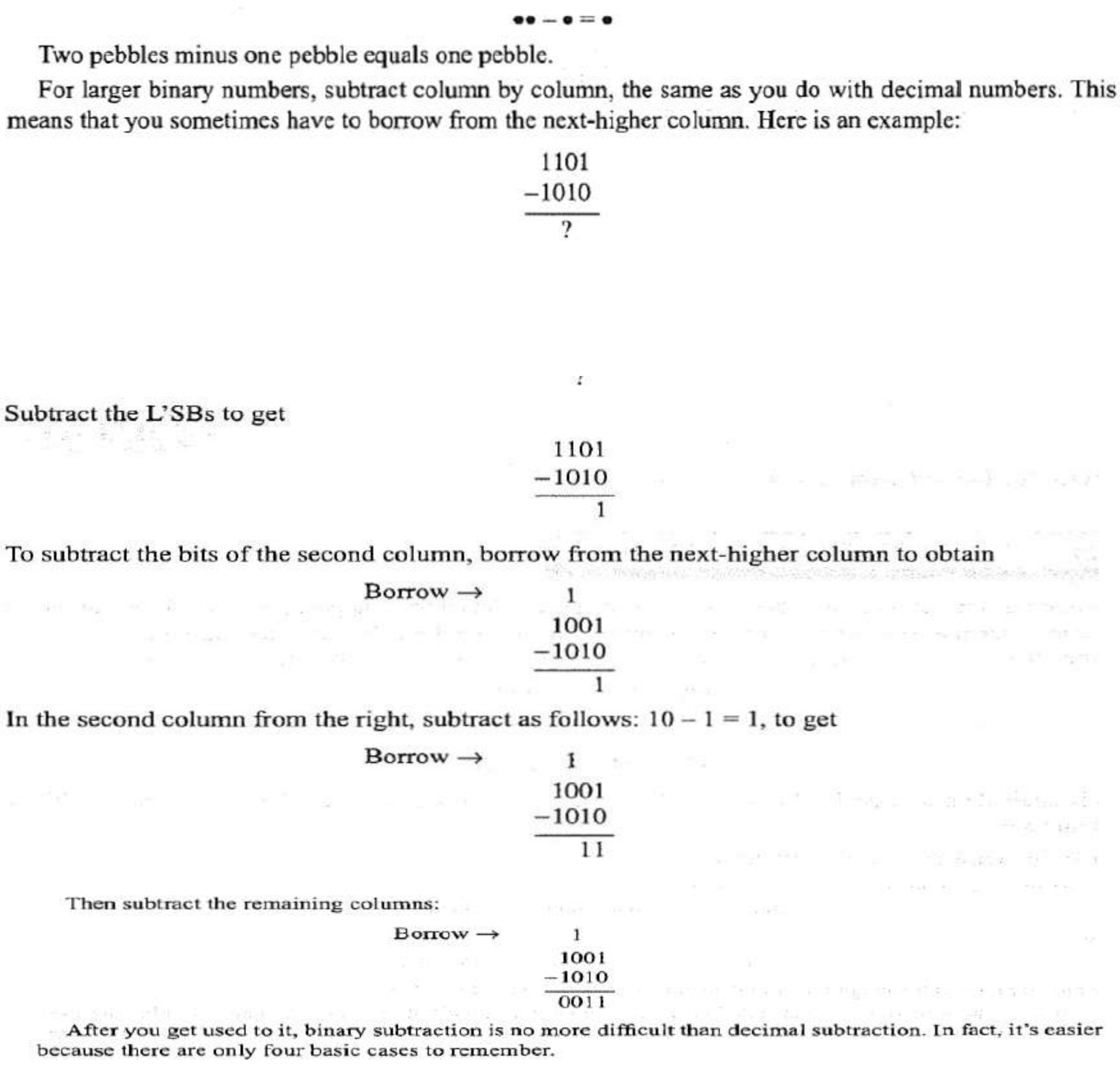
0-0 =O (6.5)

1-0 = 1 (6.6)

1-1 =O (6.7)

10-1=1 (6.8)

- Equations (6.5) to (6.7) are easy to understand because they are identical to decimal subtraction. The fourth case will disturb you if you have lost sight of what it really means. Back in the physical world, Eq (6.4) represents


$$\bullet\bullet - \bullet = \bullet$$

Two pebbles minus one pebble equals one pebble.

For larger binary numbers, subtract column by column, the same as you do with decimal numbers. This means that you sometimes have to borrow from the next-higher column. Here is an example:

$$\begin{array}{r} 1101 \\ -1010 \\ \hline ? \end{array}$$

Subtract the L'SBs to get

$$\begin{array}{r} 1101 \\ -1010 \\ \hline 1 \end{array}$$

To subtract the bits of the second column, borrow from the next-higher column to obtain

$$\begin{array}{lr} \text{Borrow} \rightarrow & 1 \\ & 1001 \\ & -1010 \\ \hline & 1 \end{array}$$

In the second column from the right, subtract as follows: 10 − 1 = 1, to get

$$\begin{array}{lr} \text{Borrow} \rightarrow & 1 \\ & 1001 \\ & -1010 \\ \hline & 11 \end{array}$$

Then subtract the remaining columns:

$$\begin{array}{lr} \text{Borrow} \rightarrow & 1 \\ & 1001 \\ & -1010 \\ \hline & 0011 \end{array}$$

After you get used to it, binary subtraction is no more difficult than decimal subtraction. In fact, it's easier because there are only four basic cases to remember.

6.3 UNSIGNED BINARY NUMBERS

- All data is either positive or negative. When this happens, you can forget about + and - signs, and concentrate on the *magnitude* (absolute value) of numbers.
- For instance, the smallest 8-bit num-ber is 0000 0000, and the largest is 1111 1111. Therefore, the total range of 8-bit numbers is

0000	0000	(OOH)
To		
1111	1111	(FFH)

- This is equivalent to a decimal Oto 255. As you can see, orwe are not including+ - signs with these decimal numbers.

With 16-bit numbers, the total range is

0000	0000 0000 0000	(OOOOH)

To

1111 1111 1111 1111 (FFFFH)

which represents the magnitude of all decimal numbers from O to 65,535.

- Data of the foregoing type is called *unsigned binary* because all of the bits in a binary number are used to represent the magnitude of the corresponding decimal number.
- You can unsigned add and subtract binary numbers, provided certain conditions are satisfied. The following examples will tell you more about unsigned binary numbers.

limits

- First-generation microcomputers can process only 8 bits at a time. For this reason, there are certain restrictions you should be aware of. With 8-bit unsigned arithmetic, all magnitudes must be between O and 255.
- Therefore, each number being added or subtracted must be between O and 255. Also, the answer must fall in the range ofO to 255. If any magnitudes are greater than 255, you should use 16-bit arithmetic, which means operating on the lower 8 bits first, then on the upper 8 bits (see Example 6.3).

Overflow

- In 8-bit arithmetic, addition of two unsigned numbers whose sum is greater than 255 causes an *ove1ifow*, a carry into the ninth column.
- Most microprocessors have a logic circuit called a *carryfiag;* this circuit detects a carry into the ninth column and warns you that the 8-bit answer is invalid.

6.4 SIGN-MAGNITUDE NUMBERS

- What do we do when the data has positive and negative values? The answer is important because it determines how complicated the arithmetic circuits must be. The negative decimal numbers are -1, -2, -3, and so on.
- The magnitude of these numbers is 1, 2, 3, and so forth. One way to code these as binary numbers is to convert the magnitude to its binary equivalent and prefix the sign.
- With this approach, the sequence-I, -2, and-3 becomes-001, -010, and-011. Since everything has to be coded as strings ofOs and ls, the+ and- signs also have to be represented in binary form.

- For reasons given soon, 0 is used for the+ sign and 1 for the - sign. Therefore, -001, -010, and-011 are coded as 1001, 1010, and l0ll.

- The foregoing numbers contain a sign bit followed by magnitude bits. Numbers in this form are called *sign-magnitude numbers.*
- For larger decimal numbers, you need more than 4 bits. But the idea is still the same: the MSB always represents the sign, and the remaining bits always stand for the magnitude. Here are

some examples of converting sign-magnitude numbers:

$+7 \rightarrow 0000\ \ 0111$

$-16 \rightarrow 1001\ \ 0000$

$+25 \rightarrow 0000\ \ 0000\ \ 0001\ \ 1001$

$-128 \rightarrow 1000\ \ 0000\ \ 1000\ \ 0000$

Range of Sign-Magnitude Numbers

- As you know, the unsigned 8-bit numbers cover the decimal range ofO to 255. When you use sign-magnitude numbers, the you reduce largest magnitude from 255 to 127 because you need to represent both positive and negative quantities. For instance, the negative numbers are

```
                 1000  0001      (−1)
to
                 1111  1111      (−127)
The positive numbers are
                 0000  0001      (+1)
to
                 0111  1111      (+127)
```

- The largest magnitude is 127, approximately half of what is for unsigned binary numbers. As long as your input data is in the range of -127 to + 127, you can use 8-bit arithmetic. The programmer still must check sums for an overflow because all 8-bit answers are between -127 and+ 127.

- If the data has magnitudes greater than 127, then 16-bit arithmetic may work. With 16-bit numbers, the negative numbers are from

```
                 1000  0000  0000  0001      (−1)
to
                 1111  1111  1111  1111      (−32, 767)
and the positive numbers are from
                 0000  0000  0000  0001      (+1)
to
                 0111  1111  1111  1111      (+32, 767)
```

- The main advantage of sign-magnitude numbers is their simplicity. Negative numbers are identical to positive numbers, except for the sign bit.
- Because of this, you can easily find the magnitude by deleting the sign bit and converting the remaining bits to their decimal equivalents.
- Unfortunately, sign-magnitude numbers have limited use because they require complicated arithmetic circuits. If you don't have to add or subtract the data, sign-magnitude numbers are acceptable.
- For instance, sign-magnitude numbers are often used in *analog-to-digital* (AID) conversions.

2'S COMPLEMENT REPRESENTATION

- There is a rather unusual number system that leads to the simplest logic circuits for performing arithmetic. Known as , 2's *complement representation,* this system dominates microcomputer architecture and programming.

1 's Complement

- The I's complement of a binary number is the number that results when we complement each bit.
- Figure 6.1 shows how to produce the I's complement with logic circuits. Since each bit drives an inverter, the 4-bit output is the l's complement of the 4-bit input. For in-stance, if the input is

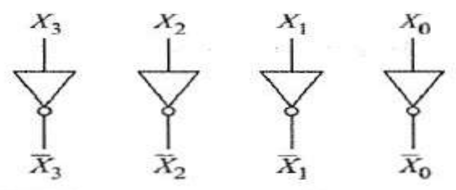


Fig. 6.1 **Inverters produce the 1's complement.**

The same principle applies to binary numbers of any length: complement each bit to obtain the l's complement. More examples of l's complements are

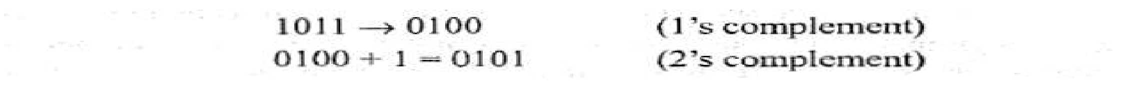
$$1010 \rightarrow 0101$$
$$1110\ \ 1100 \rightarrow 0001\ \ 0011$$
$$0011\ \ 1111\ \ 0000\ \ 0110 \rightarrow 1100\ \ 0000\ \ 1111\ \ 1001$$

2's Complement

- The 2's complement is the binary number that results when we add l to the l's complement. As a formula:

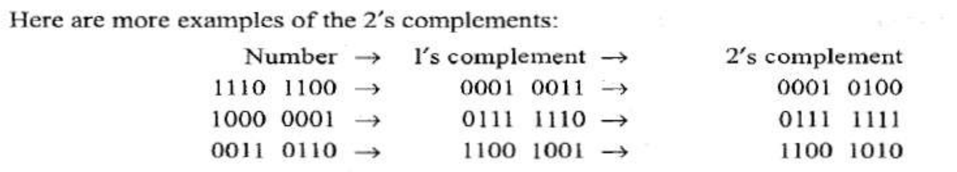
2's complement = l's complement + 1

- For instance, to find the 2's complement of 1011, proceed like this:



$1011 \rightarrow 0100$	(1's complement)
$0100 + 1 = 0101$	(2's complement)

- Instead of adding 1, you can visualize the next reading on a binary odometer. So, after obtaining the l's complement O100, ask yourself what comes next on a binary odometer. The answer is 0101.



Here are more examples of the 2's complements:

Number →	1's complement →	2's complement
1110 1100 →	0001 0011 →	0001 0100
1000 0001 →	0111 1110 →	0111 1111
0011 0110 →	1100 1001 →	1100 1010

Back to the Odometer

- The binary odometer is a marvelous way to understand 2's complement representation.
- By examining the numbers of a binary odometer, we can see how the typical microcomputer represents positive and negative numbers. With a binary odometer, all bits eventually reset to Os.
- Some readings before and after a complete reset look like this:

1000	(-8)
1001	(-7)
1010	(-6)
1011	(-5)
1100	(-4)
1101	(-3)

1110	(-2)
1111	(-1)
0000	(0)
0001	(+1)
0010	(+2)
0011	(+3)
0100	(+4)
0101	(+5)
0110	(+6)
0111	(+7)

- Binary 1101 is the reading 3 miles before reset, 1110 occurs 2 miles before reset, and 1111 indicates 1 mile before reset.
- Then, 0001 is the reading 1 mile after reset, 0010 occurs 2 miles after reset, and OOH indicates 3 miles after reset.

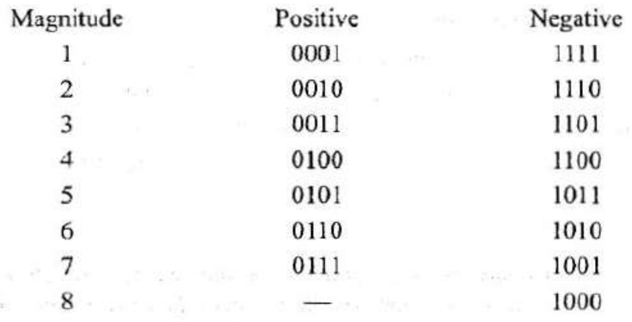
Positive and Negative Numbers

- "Before" and "after" are synonymous with "negative" and "positive." Figure 6.2 illustrates with this idea the number line of basic algebra: 0 marks the origin, positive numbers are on the right, and negative numbers are on the left.
- The odometer readings are the binary equivalents of decimal numbers: 1000 is the binary equivalent of -8, 1001 stands for - 7, 1010 stands for -6, and so on.

- The odometer readings in Fig. 6.2 demonstrate how positive and negative numbers are stored in a typical microcomputer.

Here are two important ideas to notice about these odometer readings.

- First, the MSB is the sign bit: 0 represents a + sign, and 1 stands for a - sign.
- Second, the negative numbers in Fig. 6.2 are the 2's Complements positive of the numbers, as you can see in the following:



Magnitude	Positive	Negative
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8	—	1000

Except for the last entry, the positive and negative numbers are 2′s complements of each other.

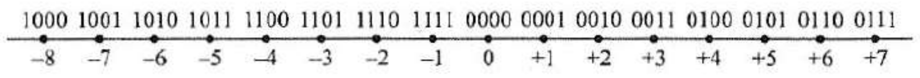
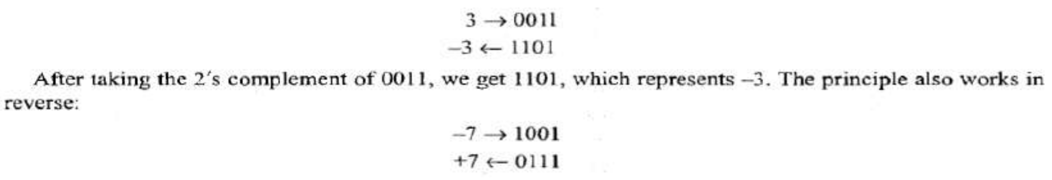


Fig. 6.2 Representing decimal numbers as 2′s complements

- In other words, you can take the 2's complement of a positive binary number to find the corresponding negative binary number. For instance:



$3 \rightarrow 0011$

$-3 \leftarrow 1101$

After taking the 2′s complement of 0011, we get 1101, which represents −3. The principle also works in reverse:

$-7 \rightarrow 1001$

$+7 \leftarrow 0111$

- After taking the 2's complement of 1001, we obtain O111, which represents +7.

- What does the foregoing mean? It means that taking the 2's complement is equivalent to *negation,* changing the sign of the number. Why is this important? Because it's easy to build a logic circuit that produces the 2's complement. Whenever this circuit takes the 2's complement, the output is the negative of the input. This key idea leads to an incredibly simple arithmetic circuit that can add and subtract.

- In summary, here are the things to remember about 2's complement representation:

I. Positive numbers always have a sign bit of 0, and negative numbers always have a sign bit of 1.

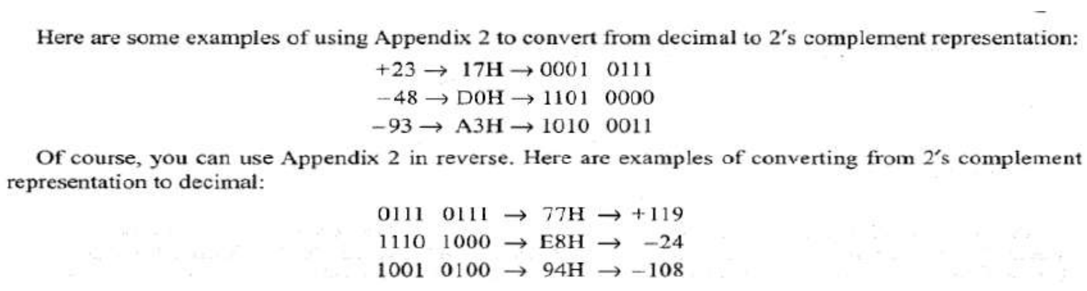
- Positive numbers are stored in sign-magnitude form.

- Negative numbers are stored as 2's complements.

- Taking the 2's complement is equivalent to a sign change.

Converting to and from 2'sComplement Representation

- We need a fast way to express numbers in 2's complement representation. Appendix 2 lists all 8-bit numbers in positive and negative form.



Here are some examples of using Appendix 2 to convert from decimal to 2's complement representation:

$+23 \rightarrow 17H \rightarrow 0001\ \ 0111$

$-48 \rightarrow D0H \rightarrow 1101\ \ 0000$

$-93 \rightarrow A3H \rightarrow 1010\ \ 0011$

Of course, you can use Appendix 2 in reverse. Here are examples of converting from 2's complement representation to decimal:

$0111\ \ 0111 \rightarrow 77H \rightarrow +119$

$1110\ \ 1000 \rightarrow E8H \rightarrow -24$

$1001\ \ 0100 \rightarrow 94H \rightarrow -108$

- A final point. Look at the last two entries in Appendix 2. As you see, + 127 is the largest positive number in 2's complement representation, and-128 is the largest negative number.
- Similarly, in the 4-bit odometer discussed earlier, + 7 was the largest positive number, and -8 was the largest negative number. The largest negative number has a magnitude that is one greater than the largest positive number. This *slight asymmetry* of 2's complement representation has no particular meaning.

2'S. COMPLEMENT ARITHMETIC

- Early computers used sign-magnitude numbers for positive and negative values. This led to complicated arithmetic circuits.
- Then an engineer discovered that 2's complement representation could simplify arithmetic *hardware.* (This refers to the electronic, magnetic, and mechanical devices of a computer.)

- Since then, 2'scomplement representation has become a universal code for processing positive negative and numbers.

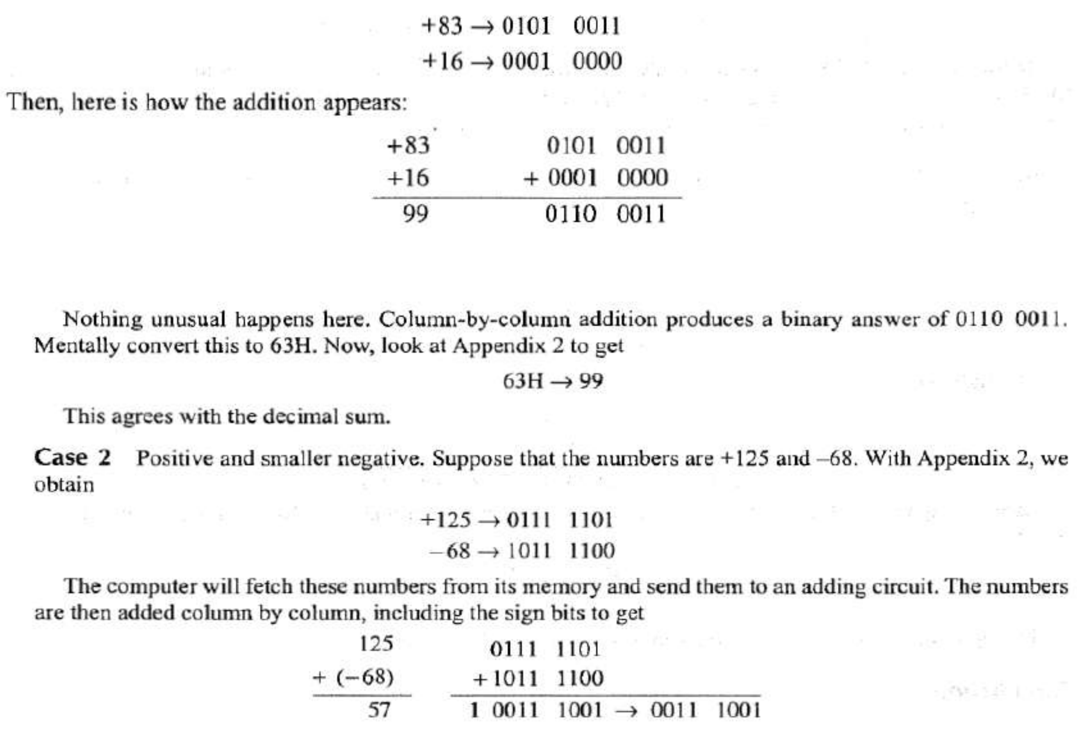
Help from the Binary Odometer

- Addition and subtraction can be visualized in terms of a binary odometer. When you add a positive number, this is equivalent to advancing the odometer reading. When you add a negative number, this has the effect of turning the odometer backward.
- Likewise, subtraction of a positive number reverses the odometer, but subtraction of a negative number advances it.

Addition

- Let us take a look at how binary numbers are added. There are four possible cases: both numbers positive, a positive number and a smaller negative number, a negative number and a smaller positive number, and both numbers negative.
- Let us go through all four cases for a complete coverage of what happens when a computer adds numbers.

Case 1 Both positive. Suppose that the numbers are +83 and + 16. With Appendix 2, these numbers are converted as follows:



$$+83 \rightarrow 0101\ \ 0011$$
$$+16 \rightarrow 0001\ \ 0000$$

Then, here is how the addition appears:

```
 +83          0101  0011
 +16        + 0001  0000
 ---        ------------
  99          0110  0011
```

Nothing unusual happens here. Column-by-column addition produces a binary answer of 0110 0011. Mentally convert this to 63H. Now, look at Appendix 2 to get

$$63H \rightarrow 99$$

This agrees with the decimal sum.

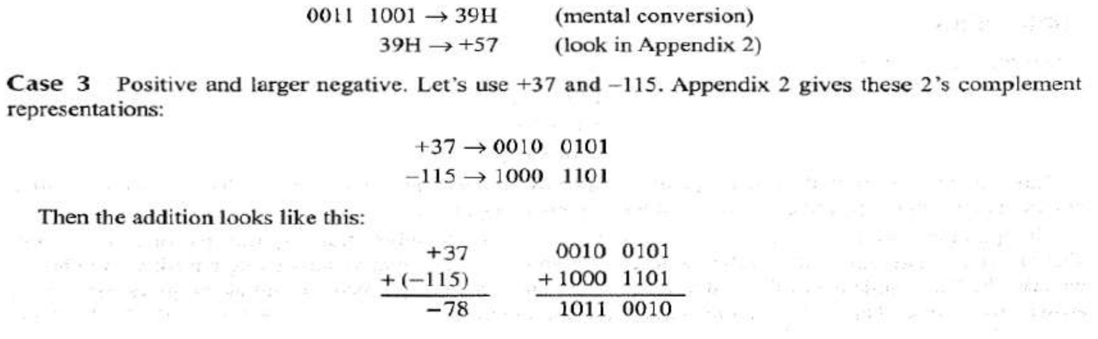
Case 2 Positive and smaller negative. Suppose that the numbers are +125 and –68. With Appendix 2, we obtain

$$+125 \rightarrow 0111\ \ 1101$$
$$-68 \rightarrow 1011\ \ 1100$$

The computer will fetch these numbers from its memory and send them to an adding circuit. The numbers are then added column by column, including the sign bits to get

```
    125          0111  1101
 + (-68)       + 1011  1100
 -------       ----------------------------
     57        1 0011  1001  →  0011  1001
```

- With 8-bit arithmetic, you *disregard the final carry* into the ninth column. The reason is related to the bi-nary odometer, which ignores final carries.
- In other words, when the eighth wheel resets, it does not generate a carry because there is no ninth wheelthe to receive carry. You can convert the binary answer to decimal as follows:



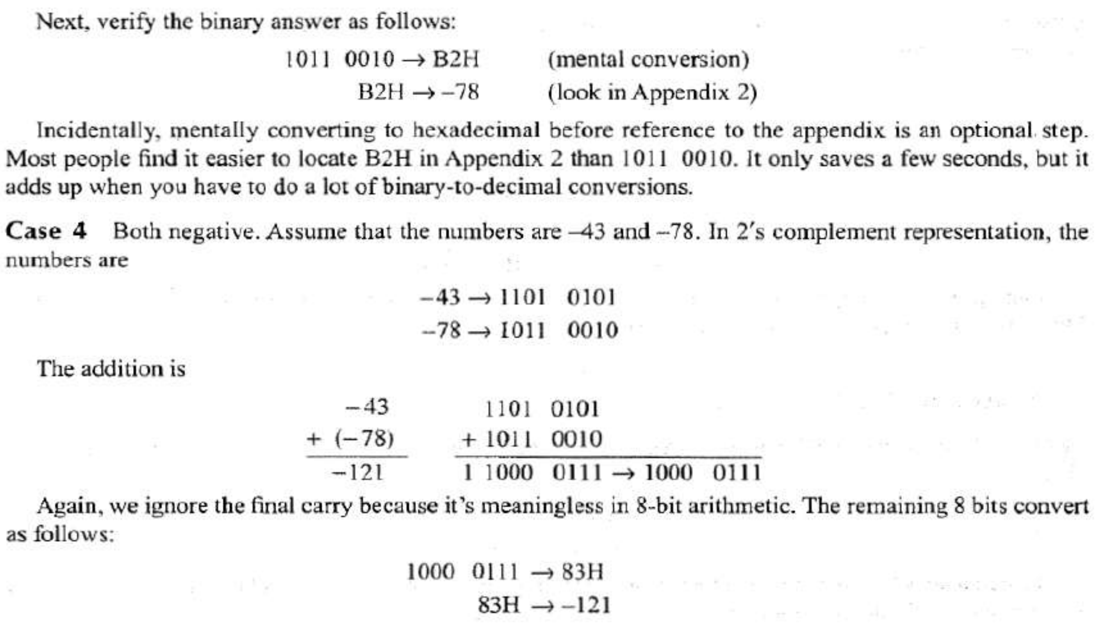
$$0011\ 1001 \rightarrow 39H \quad \text{(mental conversion)}$$
$$39H \rightarrow +57 \quad \text{(look in Appendix 2)}$$

Case 3 Positive and larger negative. Let's use +37 and −115. Appendix 2 gives these 2's complement representations:

$$+37 \rightarrow 0010\ 0101$$
$$-115 \rightarrow 1000\ 1101$$

Then the addition looks like this:

$$\begin{array}{r} +37 \\ +(-115) \\ \hline -78 \end{array} \qquad \begin{array}{r} 0010\ 0101 \\ +1000\ 1101 \\ \hline 1011\ 0010 \end{array}$$



Next, verify the binary answer as follows:

$$1011\ 0010 \rightarrow B2H \quad \text{(mental conversion)}$$
$$B2H \rightarrow -78 \quad \text{(look in Appendix 2)}$$

Incidentally, mentally converting to hexadecimal before reference to the appendix is an optional step. Most people find it easier to locate B2H in Appendix 2 than 1011 0010. It only saves a few seconds, but it adds up when you have to do a lot of binary-to-decimal conversions.

Case 4 Both negative. Assume that the numbers are −43 and −78. In 2′s complement representation, the numbers are

$$-43 \rightarrow 1101\ 0101$$
$$-78 \rightarrow 1011\ 0010$$

The addition is

$$\begin{array}{r} -43 \\ +\ (-78) \\ \hline -121 \end{array} \qquad \begin{array}{r} 1101\ 0101 \\ +\ 1011\ 0010 \\ \hline 1\ 1000\ 0111 \rightarrow 1000\ 0111 \end{array}$$

Again, we ignore the final carry because it's meaningless in 8-bit arithmetic. The remaining 8 bits convert as follows:

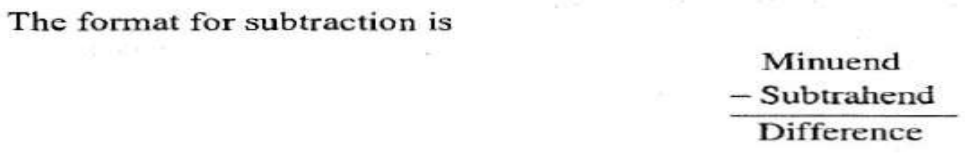
$$1000\ 0111 \rightarrow 83H$$
$$83H \rightarrow -121$$

- This agrees with the answer we obtained by direct decimal addition.

Conclusion

- We have exhausted the possibilities. In every case, 2's complement addition works. In other words, as long as positive and negative numbers are expressed in 2's complement representation, an adding circuit will automatically produce the correct answer.

Subtraction



The format for subtraction is

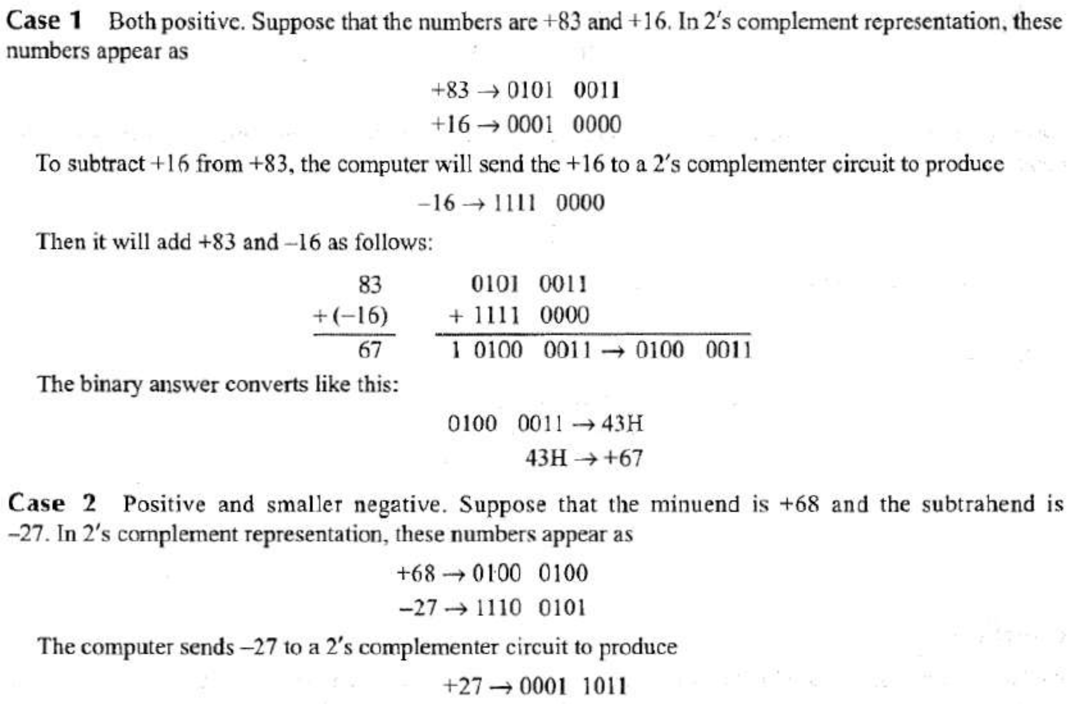
$$\begin{array}{r} \text{Minuend} \\ -\ \text{Subtrahend} \\ \hline \text{Difference} \end{array}$$

- There four cases: both numbers positive, a positive number and a smaller negative number, a negativenumber and a smaller positive number, and both numbers negative.

- The question now is *how can we use an adding circuit* to do subtraction. By trickery, of course.

- From algebra, you already know that adding a negative number is equivalent to subtracting a positive number. If we take the 2's complement of the subtrahend, addition of the complemented subtrahend gives the correct answer. Remember that the 2's complement is equivalent to negation.

- One way to remove all doubt about this critical idea is to analyze the four cases that can arise during a subtraction.



Case 1 Both positive. Suppose that the numbers are +83 and +16. In 2′s complement representation, these numbers appear as

$$+83 \rightarrow 0101\ \ 0011$$
$$+16 \rightarrow 0001\ \ 0000$$

To subtract +16 from +83, the computer will send the +16 to a 2′s complementer circuit to produce

$$-16 \rightarrow 1111\ \ 0000$$

Then it will add +83 and −16 as follows:

```
     83          0101  0011
  +(−16)       + 1111  0000
  ------      --------------------------
     67        1 0100  0011 → 0100  0011
```

The binary answer converts like this:

$$0100\ \ 0011 \rightarrow 43H$$
$$43H \rightarrow +67$$

Case 2 Positive and smaller negative. Suppose that the minuend is +68 and the subtrahend is −27. In 2′s complement representation, these numbers appear as

$$+68 \rightarrow 0100\ \ 0100$$
$$-27 \rightarrow 1110\ \ 0101$$

The computer sends −27 to a 2′s complementer circuit to produce

$$+27 \rightarrow 0001\ \ 1011$$

Then it adds +68 and +27 as follows:

$$\begin{array}{r} +68 \\ +27 \\ \hline 95 \end{array} \qquad \begin{array}{r} 0100\ \ 0100 \\ +\,0001\ \ 1011 \\ \hline 0101\ \ 1111 \end{array}$$

The binary answer converts to decimal as follows:

$$0101\ \ 1111 \rightarrow 5FH$$
$$5FH \rightarrow +95$$

Case 3 Positive and larger negative. Let's use a minuend of +14 and a subtrahend of −108. Appendix 2 gives these 2's complement representations:

$$+14 \rightarrow 0000\ \ 1110$$
$$-108 \rightarrow 1001\ \ 0100$$

The computer produces the 2's complement of −108:

$$+108 \rightarrow 0110\ \ 1100$$

Then it adds the numbers like this:

$$\begin{array}{r} 14 \\ +108 \\ \hline 122 \end{array} \qquad \begin{array}{r} 0000\ \ 1110 \\ +\,0110\ \ 1100 \\ \hline 0111\ \ 1010 \end{array}$$

The binary answer converts to decimal like this:

$$0111\ \ 1010 \rightarrow 7AH$$
$$7AH \rightarrow +122$$

Case 4 Both negative. Assume that the numbers are −43 and −78. In 2's complement representation, the numbers are

$$-43 \rightarrow 1101\ \ 0101$$
$$-78 \rightarrow 1011\ \ 0010$$

First, take the 2's complement of −78 to get

$$+78 \rightarrow 0100\ \ 1110$$

Then add to obtain

$$\begin{array}{r} -43 \\ +78 \\ \hline 35 \end{array} \qquad \begin{array}{l} \quad\ 1101\ \ 0101 \\ +\ 0100\ \ 1110 \\ \hline 1\ 0010\ \ 0011 \rightarrow 0010\ \ 0011 \end{array}$$

Then

$$0010\ \ 0011 \rightarrow 23H$$
$$23H \rightarrow +35$$

UNIT-IV

ARITHMETIC CIRCUITS

- Circuits that can perform binary addition and subtraction are constructed by combining logic gates. These circuits are used in the design of the arithmetic logic unit (ALU). The electronic circuits are capable of very fast switching action, and thus an ALU can operate at high clock rates.
- For instance, the addition of two numbers can be accomplished in a matter of nanoseconds! This chapter begins with binary addition and subtraction, then presents two different methods for representing negative numbers. You will see how an exclusive OR gate is used to construct a half-adder and a full-adder. You will see how to construct an 8-bit adder-subtracter using a popular IC.
- A technique to design a fast adder is discussed in detail followed by discussion on a multifunctional device called Arithmetic Logic Unit or ALU. Finally, an outline to perform binary multiplication and division is also presented.

ARITHMETIC BUILDING BLOCKS

- We are on the verge of seeing a logic circuit that performs 8-bit arithmetic on positive and negative numbers. But first we need to cover three basic circuits that will be used as building blocks.
- These building blocks are the half-adder, the full-adder, and the controller inverter. Once you understand how these work, it is only a short step to see how it all comes together, that is, how a computer is able to add and subtract binary numbers of any length.

Half-Adder

- When we add two binary numbers, we start with the least-significant column. This means that we have to add two bits with the possibility of a carry. The circuit used for this is called a *half-adder*.
- Figure 6.3 shows how to build a half-adder. The output of the exclusive-OR gate is called the *SUM,* while the output of the AND gate is the *CARRY.* The AND gate produces a high output only when both inputs are highexclusive.The-OR gate produces a high output if either input, but not both, is high. Table 6.2 shows the truth table of a half-adder.
- When you examine each entry in Table 6.2, you are struck by the fact that a half-adder performs binary addition.

- As you see, the half-adder mimics our brain pro-cesses in adding bits. The only difference is the half-adder is about a million times faster than we are.

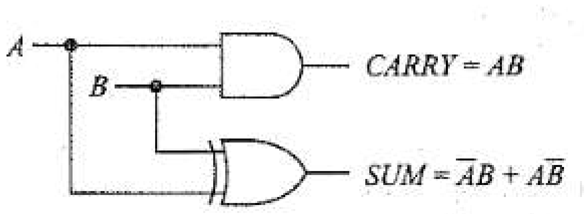


Fig. 6.3 Half-adder

Table 6.2 Half-adder Truth Table

A	B	CARRY	SUM
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full-Adder

- For the higher-order columns, we have to use *afi1ll-adder,* a logic circuit that can add 3 bits at a time. The third bit is the carry from a lower column. This implies that we need a logic circuit with three inputs and two outputs, similar to the full-adder shown in Fig. 6.4a. (Other designs are possible. This one is the simplest.)
- Table 6.3 shows the truth table of a full-adder. You can easily check this truth table for its validity. For instance, CARRY is high in Fig. 6.4a when two or more of the *ABC* inputs are high; this agrees with the CARRY column in Table 6.3. Also, when an odd number of high *ABC* inputs drives the exclusive-OR gate, it produces a high output; this verifies the SUM column of the truth table.

Table 6.3 **Full-Adder Truth Table**

A	B	C	$CARRY$	SUM
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

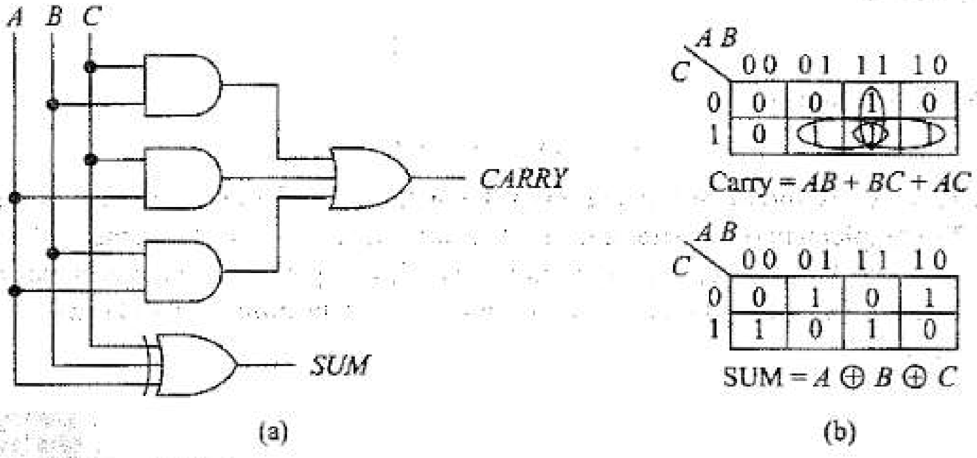


Fig. 6.4 **(a) Full-adder, (b) Karnaugh map of Table 6.3**

From this truth table we get Karnaugh map as shown in Fig. 6.4b that gives following logic equations,

$$\text{CARRY} = AB + BC + AC \quad \text{and} \quad \text{SUM} = A \oplus B \oplus C.$$

A general representation of full-adder which adds i-th bit *A;* and *B;* of two numbers *A* and Band takes carry from (i-l)th bit could be

$$C_i = A_iB_i + B_iC_{i-1} + A_iC_{i-1} \text{ or } C_i = A_iB_i + (A_i + B_i)C_{i-1} \quad \text{and} \quad S_i = A_i \oplus B_i \oplus C_{i-1}$$

Controlled Inverter

Figurea 6.5 shows *controlled inverter.* When INVERT is low, it transmits the 8-bit input to the output; when INVERT is high, it transmits the l's complement. For instance, if the input number is

$$A_7 \cdots A_0 = 0110 \quad 1110$$

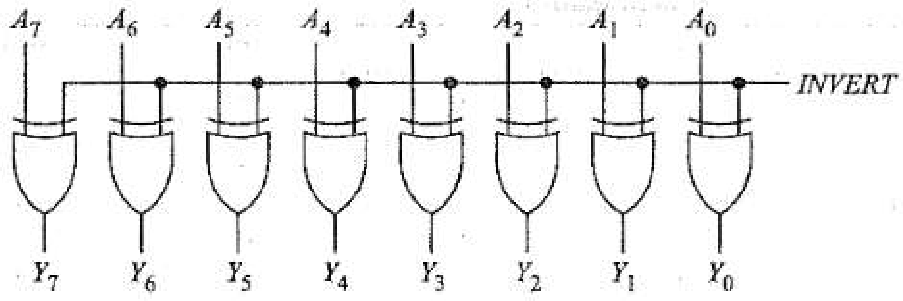


Fig. 6.5 Controlled inverter

a low INVERT produces

$$Y_7 \cdots Y_0 = 0110 \quad 1110$$

But a high INVERT results in

$$Y_7 \cdots Y_0 = 1001 \quad 0001$$

- The controlled inverter is important because it is a step in the right direction. During a subtraction, we first need to take the 2's complement of the subtrahend. Then we can add the complemented subtrahend to obtain the answer.
- With a controlled inverter, we can produce the l's complement. There is an easyto way get the 2 's complement, discussed in the next section. So, we now have all the building blocks: half-adder, full-adder, and controlled inverter.

THE ADDER..SUBTRACTER

- We can connect full-adders as shown in Fig. 6.6 to add or subtract binary numbers. The circuit is laid out from right to left, similar to the way we add binary numbers. Therefore, the least-significant column is on the right, and the most-significant column is on the left.
- The boxes labeled FA are full-adders. (Some adding circuits use a half-adder instead of a full-adder in the least-significant column.)
- The CARRY OUT from each full-adder is the CARRY IN to the next-higher full-adder. The numbers be~ ing processed are A_7 ... *Ao* and B_7 ... *Bo,* and the answer is S7 ... S_0.
- With 8-bit arithmetic, the final carry is ignored for reasons given earlier. With 16-bit arithmetic, the final carry is the carry into the addition of the upper bytes.

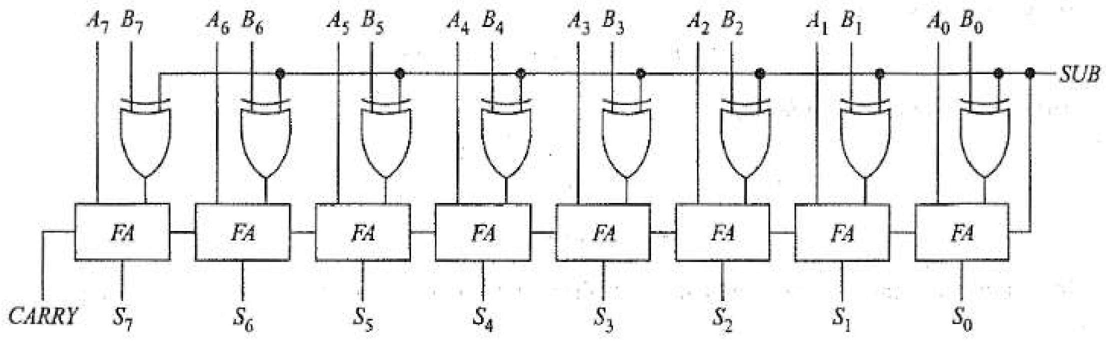


Fig. 6.6 Binary adder-subtracter

Addition

Here is how an addition appears:

$$\begin{array}{r} A_7A_6A_5A_4 \quad A_3A_2A_1A_0 \\ +\,B_7B_6B_5B_4 \quad B_3B_2B_1B_0 \\ \hline S_7S_6S_5S_4 \quad S_3S_2S_1S_0 \end{array}$$

- During an addition, the *SUB* signal is deliberately kept in the low state. Therefore, the binary number B_7... *Bo* passes through the controlled inverter with no change. The full-adders then produce the correct output sum.
- They do this by adding the bits in each column, passing carries to the next higher column, and so on. For instance, starting at the LSB, the full-adder adds *Ao, Bo,* and SUB. This produces a SUM of So and a CARRY OUT to the next-higher full-adder; The next-higher full-adder then adds A_1, B 1, and the CARRY IN to produce S_1 and a CARRY OUT.
- A similar addition occurs for each of the remaining full-adders, and the correct sum appears at the output lines.

For instance, suppose that the numbers being added are + 125 and -67. Then, A_7 ... *Ao* = 0111 1101 and B_7 ••• B_0 = 1011 1101. This is the problem:

$$\begin{array}{r} 0111 \quad 1101 \\ +1011 \quad 1101 \\ \hline ? \end{array}$$

Since SUB = 0 during an addition, the CARRY IN to the least-significant column is 0:

$$\begin{array}{r} 0 \leftarrow \text{SUB} \\ 0111 \quad 1101 \\ +1011 \quad 1101 \\ \hline ? \end{array}$$

The first full-adder performs this addition:

$0 + 1 + 1 = 0$ with a carry of 1

The CARRY OUT of the first full-adder is the CARRY IN to the second full-adder:

```
            1 ← Carry
  0111  1101
+1011  1101
-----------
           0
```

In the second column

$1 + 0 + 0 = 1$ with a carry of 0The carry goes to the third full-adder:

```
           0 ← Carry
  0111  1101
+1011  1101
-----------
          10
```

In a similar way, the remaining full-adders add their 3 input bits until we arrive at the last full-adder:

```
  1           ← Carry
  0111  1101
+1011  1101
-----------
  0011  1010
```

When the CARRY IN to the MSB appears, the full-adder produces

$1 + 0 + 1 = 0$with a carry of 1

The addition process ends with a final carry:

```
    0111  1101
  +1011  1101
  -----------
  1 0011  1010
```

During 8-bit arithmetic, this last carry is ignored as previously discussed; therefore, the answer is

$S_7 \dots S_0 = 0011\ 1010$

This answer is equivalent to decimal +58, which is the algebraic sum of the numbers we started with: +125 and-67.

Subtraction

Here is how a subtraction appears:

$$\begin{array}{rr} A_7A_6A_5A_4 & A_3A_2A_1A_0 \\ +\ B_7B_6B_5B_4 & B_3B_2B_1B_0 \\ \hline S_7S_6S_5S_4 & S_3S_2S_1S_0 \end{array}$$

During a subtraction, the SUB signal is deliberately put into the high state. Therefore, the controlled inverter produces the l's complement of B_7 ••• B_0. Furthermore, because SUB is the CARRY IN to the first full-adder, the circuit processes the data like this:

$$\begin{array}{rr} & 1 \leftarrow \text{SUB} \\ A_7A_6A_5A_4 & A_3A_2A_1A_0 \\ +\ \overline{B}_7\overline{B}_6\overline{B}_5\overline{B}_4 & \overline{B}_3\overline{B}_2\overline{B}_1\overline{B}_0 \\ \hline S_7S_6S_5S_4 & S_3S_2S_1S_0 \end{array}$$

When A_7 ... A_0 = 0, the circuit produces the 2's complement of B_7 •.. B_0 because 1 is being added to the l's complement B_7 ... B_0. When A_7 ... A_0 does not equal zero, the effect is equivalent to adding A_7 ... A_0 and the 2's complement of B_7 . .. Bo.

Here is an example. Suppose that the numbers are +82 and +17. Then *A7* ... *Ao*= 0101*Bo*= 0001inverter000I.The controlled produces the l's complement ofB, which is 11100010 and *B7* ...1110. Since SUB

- 1 during a subtraction, the circuit performs the following addition:

$$\begin{array}{rr} & 1 \leftarrow \text{SUB} \\ 0101 & 0010 \\ +\ 1110 & 1110 \\ \hline 1\ 0100 & 0001 \end{array}$$

- For 8-bit arithmetic, the final carry is ignored as previously discussed; therefore, the answer is S7 · ..

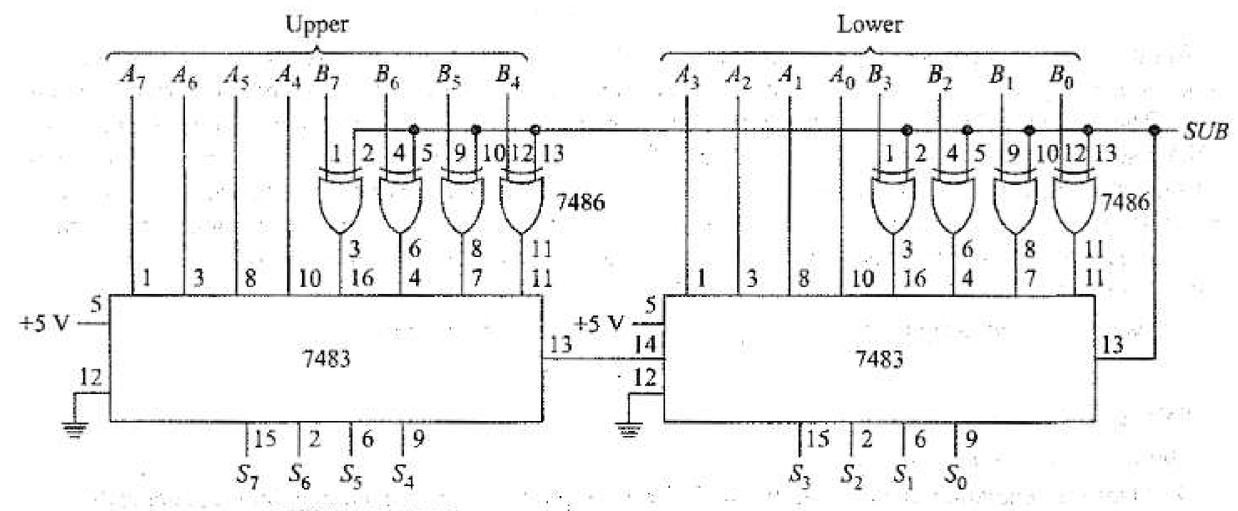


Fig. 6.7 Two 7483s can add or subtract bytes

- This answer is equivalent to decimal +65, which is the algebraic difference between the numbers we started with: +82 and+ 17.

FAST ADDER

- Fast adder is also called *parallel adder* or *carry look ahead adder* because that is how it attains high speed in addition operation. Before we go into that circuit,see let's what limits the speed of an adder.
- Consider, the worst case scenario when two four bit numbers A: 1111 and B: 0001 are added. This generates a carry in the first stage that propagates to the last stage as shown next.

```
Carry:    111
   A:    1111
   B:    0001
        -----
        10000
```

- Addition such as these (Fig. 6.6) is called *serial addition* or *ripple carry addition.* It also reveals from the adder equation (given in Section 6.8) result of every stage depends on the availability of carry from previous stage.
- The minimum delay required for carry generation in each stage is two gate delays, one coming from AND gates (1st level) and second from OR gate (2nd level). For 32-bit serial addition there will be 32 stagesn working in serial.
- In worst case,2 itx will require 32 = 64 gate delays to generate the final carry. Though each gate delay is of nanosecond order, serial addition definitely limits the speed of high speed computing. Parallel adder increases the speed by generating the carry in advance

(look ahead) and there is no need to wait for the result from previous stage. This is achieved by following method.

Let us use the second equation for carry generation from previous section, i.e.

$$C_i = A_iB_i + (A_i + B_i)C_{i-1}$$

This can be written as, $C_i = G_i + P_iC_{i-1}$

where, $G_i = A_iB_i$ and $P_i = A_i + B_i$

- *G;* stands for generation of carry and *P;* stands for propagation of carry in a particular stage depending on input to that stage. As explained in previous section, *ifA;B;* = 1, then ith stage will generate a carry, no matter previous stage generates it or not. And if *A;*+ *B;* = 1 then this stage will propagate a carry if available from previous stage to next stage.
- Note that, all *G;* and *P;* are available after one gate delay once the numbers *A* and *B* are placed.

Starting from LSB, designated by suffix O ifwe proceed iteratively
we get,

$C_0 = G_0 + P_0.C_{-1}$ [C_{-1} will normally be 0 if we are not using it as subtractor or cascading it.]

$C_1 = G_1 + P_1.C_0 = G_1 + P_1.(G_0 + P_0.C_{-1}) = G_1 + P_1.G_0 + P_1P_0.C_{-1}$ [Substituting C_0]

Similarly,

$C_2 = G_2 + P_2.C_1 = G_2 + P_2.(G_1 + P_1.G_0 + P_1P_0.C_{-1})$ [Substituting C_1]

$= G_2 + P_2.G_1 + P_2P_1.G_0 + P_2P_1P_0.C_{-1}$

$C_3 = G_3 + P_3.C_2 = G_3 + P_3(G_2 + P_2.G_1 + P_2P_1.G_0 + P_2P_1P_0.C_{-1})$ [Substituting C_2]

$= G_3 + P_3G_2 + P_3P_2.G_1 + P_3P_2P_1.G_0 + P_3P_2P_1P_0.C_{-1}$

- The equationsButlook pretty complicated. do we gain in any way? Note that, these equations can be real-ized in hardware using multi-input AND and OR gates and in two levels. Now, for each carry whether Co or C3 we require only two gate delays once the *G;* and *P;* are available.
- We have already seen they are available after 1 gate delay. Thus parallel adder (circuit diagram for 2-bit is shown in Fig. 6.8a) generates carry within

1 + 2 = 3 gate delays. Note that, after the carry is available at any stage there are two more gate delays from Ex-OR gate to generate the sum bit as we can write S; = G; EB *P;* EB C;$_{\_1}$.

- Thus serial adder in worst case requires at least *(2n* + 2) gate delays for n-bit addition and parallel adder requires only 3 + 2 = 5 gate delays for that. One can imagine the gain for higher values of *n.*

- However, there is a caution. We cannot increase *n* indiscriminately for parallel adder as every logic gate has a capacity to accept at most a certain number of inputs, *termed/an-in.* This is a characteristic of the logic family to which the gate belongs. More about this is discussed in Chapter 14.
- The other disadvantage of parallel adder is in-creased hardware complexity for large *n.* In Fig. 6.8b we present functional diagram and pin connections of a popular fast adder, IC 74283.

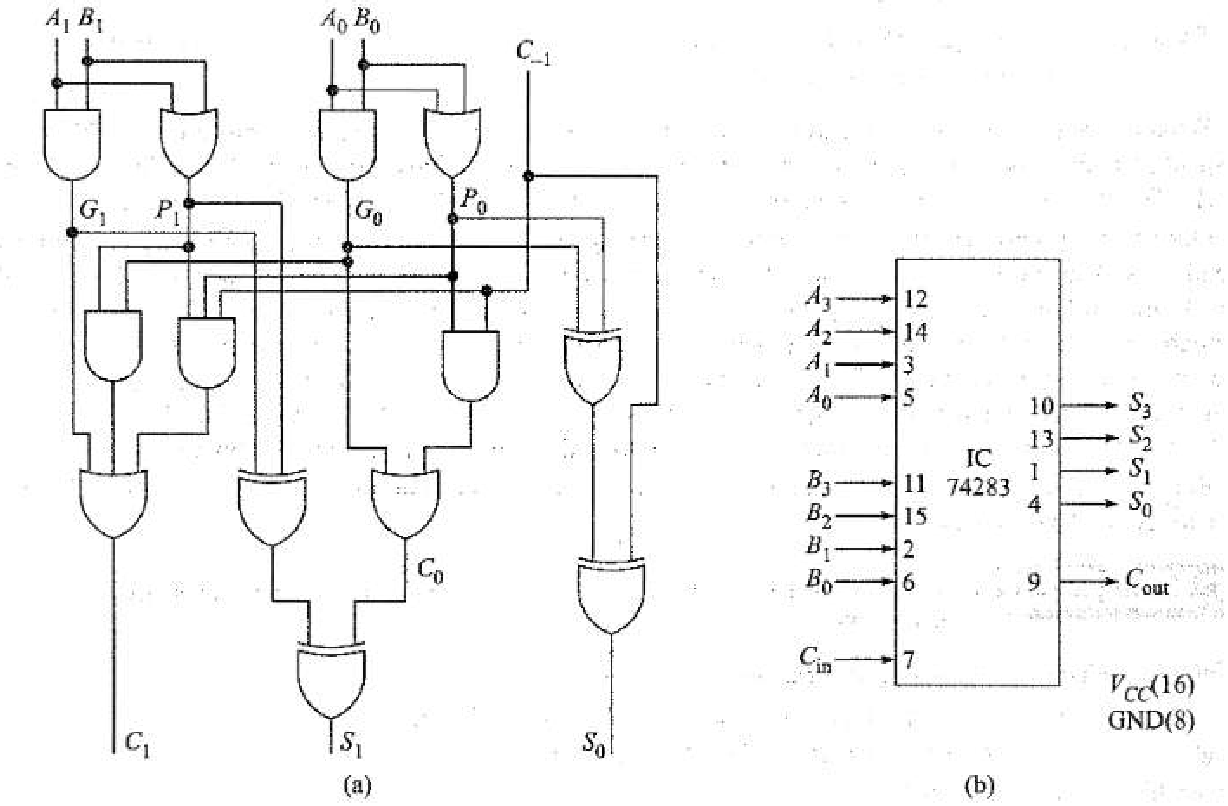


Fig. 6.8 (a) Logic circuit for 2-bit fast adder, (b) Functional diagram of IC 74283

- Now, how do we add two 8-bit numbers using IC 74283? Obviously, we need two such devices and Cout of LSB adder will be fed as C;n of MSB unit. This way each individual 4-bit addition is done parallely but between two ICs carry propagates by rippling.

To avoid carry ripple between two ICs and get truly parallel addition the following approach can be useful. Let each individual 4-bit adder unit generate two additional outputs Group Carry Generate (G_3-0) and Group Carry Propagate (P_3-0). They are defined as follows

$$G_{3-0} = G_3 + P_3G_2 + P_3P_2.G_1 + P_3P_2P_1.G_0$$
$$P_{3-0} = P_3P_2P_1P_0$$

so that

$$C_3 = G_{3-0} + P_{3-0}\, C_{-1} \qquad \text{[From equation of } C_3 \text{ in previous discussion]}$$

Now, let us see how this is useful in 8-bit parallel addition. For the 4-bit adder adding MSB taking C_3 as carry input, we can similarly write

$$C_7 = G_{7-4} + P_{7-4}\, C_3 \qquad [C_3 \text{ is equivalent to } C_{-1} \text{ input for this adder}]$$

where

$$G_{7-4} = G_7 + P_7G_6 + P_7P_6.G_5 + P_7\,P_6P_5.G_4$$
$$P_{7-4} = P_7P_6P_5P_4$$

Thus $$C_7 = G_{7-4} + P_{7-4}(G_{3-0} + P_{3-0}\, C_{-1}) \qquad \text{[substituting } C_3]$$

or $$C_7 = G_{7-4} + P_{7-4}G_{3-0} + P_{7-4}P_{3-0}\, C_{-1}$$

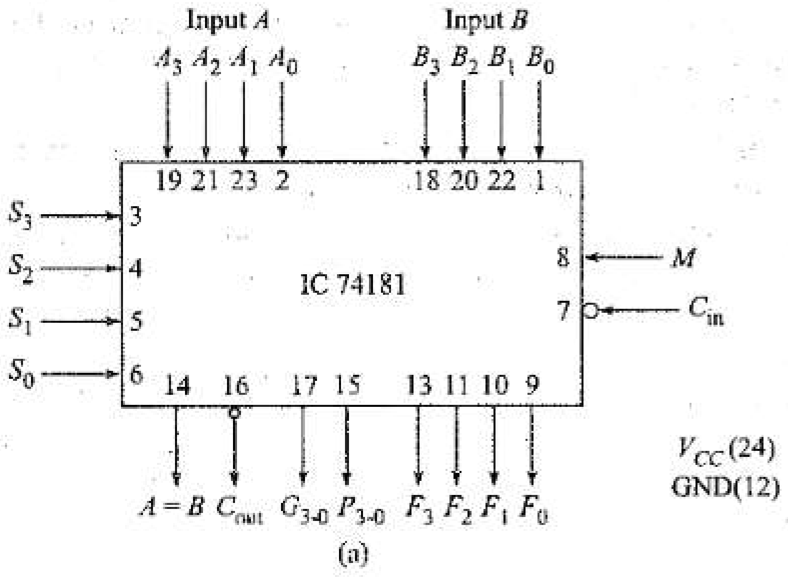
- ➢ What do we get from above equation? Group carry generation and propagation terms are available from respective adder blocks (G_{3_0}, *P3_0* from LSB and G $_7$-4, P $_7$-4 from MSB) after 3 and 2 gate delays respec-tively.
- ➢ This comes from the logic equations that define them with *Gi*, *Pi* available after 1 gate delay.
- ➢ Once these group-carry terms are available, we can generate C_7 from previousby equation designing a small Look Ahead Carry (LAC) Generator circuit.
- ➢ This requires a bank of AND gates (here one 2 input and one 3 input) followed by a multi-input OR gate (here, three input) totaling 2 gate delays.
- ➢ Thus final carry is available in 3 + 2 = 5 gate delays and this indeed is what we were looking for in parallel addition. In next section we discuss a versatile IC 74181 that while performing 4-bit addition generates this group carry generation and propagation terms.
- ➢ LAC generator circuits are also commercially available; IC 74182 can take up to four pairs of group carry terms from four adder units and generate final carry for 16 bit addition.

ARITHMETIC LOGIC UNIT

➢ Arithmetic Logic Unit, popularly called ALU is multifunctional device that can perfonn both arithmetic and logic function. ALU is an integral part of central processing unit or CPU of a computer.

➢ It comes invarious forms with wide range of functionalitynormalthan. Other addition, subtraction it can also perform increment, decrement operations. As logic unit it performs usual AND, OR, NOT, EX-OR and many other complex logic functions.

➢ It also comes with PRESET and CLEAR options, invoking which all the function outputs are made 1 and O respectively. Normally, a mode selector input (M) decides whether ALU perfonns a logic operation or an arithmetic operation. In each mode different functions are chosen by appropriately activating a set of selection inputs.



$S_3\,S_2\,S_1\,S_0$	$M = 1$ (Logic Function)	$M = 0$ (Arithmetic Function) $C_{in} = 1$ (For $C_{in} = 0$, add 1 to F)
0 0 0 0	$F = A'$	$F = A$
0 0 0 1	$F = (A + B)'$	$F = A + B$
0 0 1 0	$F = A'B$	$F = A + B'$
0 0 1 1	$F = 0$	$F =$ minus 1
0 1 0 0	$F = (AB)'$	$F = A$ plus (AB')
0 1 0 1	$F = B'$	$F = (A + B)$ plus (AB')
0 1 1 0	$F = A \oplus B$	$F = A$ minus B minus 1
0 1 1 1	$F = AB'$	$F = AB'$ minus 1
1 0 0 0	$F = A' + B$	$F = A$ plus (AB)
1 0 0 1	$F = (A \oplus B)'$	$F = A$ plus B
1 0 1 0	$F = B$	$F = (A + B')$ plus (AB)
1 0 1 1	$F = AB$	$F = AB$ minus 1
1 1 0 0	$F = 1$	$F = A$ plus A
1 1 0 1	$F = A + B'$	$F = (A + B)$ plus A
1 1 1 0	$F = A + B$	$F = (A + B')$ plus A
1 1 1 1	$F = A$	$F = A$ minus 1

(b)

Fig. 6.9 **(a) Functional representation of ALU IC 74181, (b) Its truth table**

- In this section, we take up one very popular discrete ALU device from TTL family for discussion. IC 74181 is a 4-bit ALU that can generate 16 different kinds of outputs in each mode selected by four selection inputs S_3, S_2, S_1 and So.
- The functional diagram of this IC with pin numbers and corresponding truth table is shown in Fig. 6.9(a) and Fig. 6.9(b) respectively. Note that this truth table considers data inputs *A* and Bare activeA high. similar but different trnth table is obtained if data is considered as active low.
- Well, the truth table is pretty exhaustive though one might wonder what could be the utility of functions like *(A+ B)* plus *AB'*. But a careful observation shows one important function missing, that ofa comparator. Is it truly so? No, it can be obtained in an indirect way.
- The Cout is activated (active low) by addition as well as subtractio.n because subtraction is carried out by 2 's complement addition. Note that, if the result of an arithmetic operation is negative it will be available in 2 's complement form.
- The *A*= *B* output is activated when an the function outputs are 1, i.e. $F_3 \ldots F_0$ = 1111. Output *A*= *B*, together with Cout can give functions like *A*> Band *A* <*B*. Note that

- *B* is an open collector output; thus when more than 4-bits are to be compared this output of different ALU devices are wire-ANDed, simply by knotting outputs together to get the final result. To know more about open collector gates refer to Section 14.5 of Chapter 14.
- The outputs Cout, G_{3_0} and P $_{3_}$o are useful when addition and subtraction of more than 4-bits are performed using more than one IC 74181 as discussed in previous sections.
- Logic operations are done bit-wise by making *M* = l and choosing appropriate select inputs. Note that, carry is inhibited for *M* = 1.
- Let us see how AND operation between two 4-bit numbers 1101 and O111 is to be performed. Enter input A_3 .. *Ao* = 1101 and B_3 .. *Bo* = 0111. Make S3 .. S_0 = 1011 and of course *M* = I to choose logic function. The output is shown as F_3 . .F_0 = 00 ll.

For arithmetic operations *M* = 0 to be chosen and we have to appropriately place Cin (active low), if any. For example, ifwe want to add decimal numbers 6 with 4 we have to place 0110 for 6 at *A* and 0100 for 4 at

➢ Then with S_3 .. S_0 = 1001 (from truth table) and Cin = 1 (active low) the output generated is F_3 ..F_0 = 1010 which is decimal equivalent of 10.

BINARY MULTIPLICATION AND DIVISION

➢ Typical 8-bit microprocessors like the 6502 and the 8085 use software multiplication and division. In other words, multiplication is done with addition instructions and division with subtraction instructions.

➢ Therefore, an adder-subtracter is all that is needed for addition, subtraction, multiplication, and division.

For example, multiplication is equivalent to repeated addition. Given a problem such as

$$8x4=?$$

the first number is called the *multiplicand* and the second number, the *multiplier*. Multiplying 8 by 4 is the same as adding 8 four times:

$$8+8+8+8=?$$

- ➢ One way to multiply 8 by 4 is to program a computer to add 8 until a total of four 8s have this been added. approach known as programmed multiplication by repeated addition.
- ➢ There are other software solutions to multiplication and division that you will learn about if you study assembly-language programming.

- ➢ There availableareICs that will multiply two binary numbers. For instance, the 74284 and the 74285 will produce an 8-bitthatbinary number is the product of two 4-bit binary numbersI.

- ➢ These veryCsare fast, and the total multiplication time is only about 40 nanoseconds (ns)!

CLOCK AND TIMING CIRCUITS

- ➢ The heart of every digital system is the system clock. The system clock provides the heartbeat without which the system would cease to function. In this chapter we consider the characteristics of a digital clock signal as well as some typical clock circuits.

- *Schmitt triggers* are used to produce nearly ideal digital signals from otherwise noisy or degraded signals. *Propagation delay* is the time required for a signal to pass from the input of a circuit to its output.
- You will see how to utilize logic gate propagation delay time to construct a pulse-forming circuit. A *monostable* is a basic digital timing circuit that is used in a wide variety of timing applications. We consider a number of different commercially available monostable circuits and examine some common applications.

CLOCK WAVEFORMS

- Up to this point, we have been considering *static* digital logic levels, that is, voltage levels that do not change with time. However, all digital computer systems operate by "stepping through" a series oflogical operations.
- The system signals are therefore changing with time: they are *dynamic.* The concept of a system *clock* was in-troduced in Chapter. 1. It is the clock signal that advances the system logic through its sequence of steps. The square wave shown in fig 7.1 a is a typical clock.
- Waveform used in a digital system. It should be noted that the clock need not be the perfectly symmetrical wavefonn shown. It could simply be a se1ies ofpositive (ornegative) pulses as shown in Fig. 7.lb.
- This waveform could of course be considered an asymmetrical square wave with a duty cycle other than 50 percent. The main requirement is that the clock be perfectly periodic, and stable.
- Notice that each signal in Fig. 7.1 defines a basic timing interval during which logic operations must be performed.
- This basic timing interval is defined as the *clock cycle time,* and it is equal to one period of the clock wave-fonn. Thus all logic elements must complete their transitions in less than one clock cycle time.

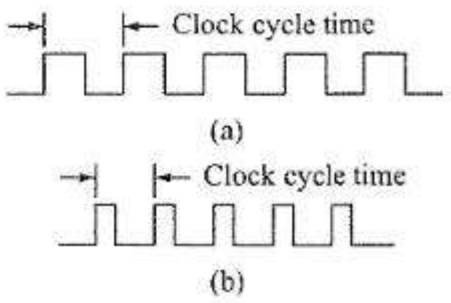


Fig. 7.1 Ideal clock waveforms

Synchronous Operation

- Nearly all of the circuits in a digital system (computer) change states in *.STnchronism* with the system clock. A change of state will either occur as the clock transitions from low to high or as it transitions from high to low.
- The low-to-high transition is frequently called the *positive transition (PT),* as shown in Fig. 7.2.
- The PT is given emphasis by drawing a small arrowthe on *rising edge* of the clock waveform. A circuit that changes state at this time is said to be *positive-edge-triggered.* The high-to-low transition is called the *negative transition* (NI), as shown in Fig. 7.2.
- The NT is emphasized by drawing a small arrow on *thefalling edge* of the clock wavefom1. A circuit that changes state at iliis time is said to be *negative- edge-triggered.*
- Virtually all circuits in a digital system are either positive-edge- triggered or negative-edge-triggered, and thus are synchronized with the system clock. There are a few exceptions.
- For instance, the operation of a push button (RESET) by a human operator might result in an instant change of state that is not in synchronism with the clock. This is called an *asynchronous operation.*

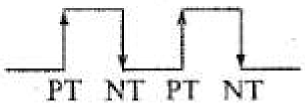


Fig. 7.2

Characteristics

- The clock waveform drawn above the time line in Fig. 7.3a is a perfect, ideal clock. What exactly are the characteristics that make up an ideal clock? First, the clock levels must be absolutely stable.
- When the clock is high, the level must hold a steady value of +5 V, as shown between points

a and *b* on the time line.

- When the clock is low, the level must be an unchanging O V, as it is between points *b* and *c*. In actual practice, the stability of the clock is much more important than the absolute value of the voltage level.
- For instance, it might be perfectly acceptable to have a high level of +4.8 V instead of+ 5.0 V, provided it is a *steady, unchanging,* +4.8V.

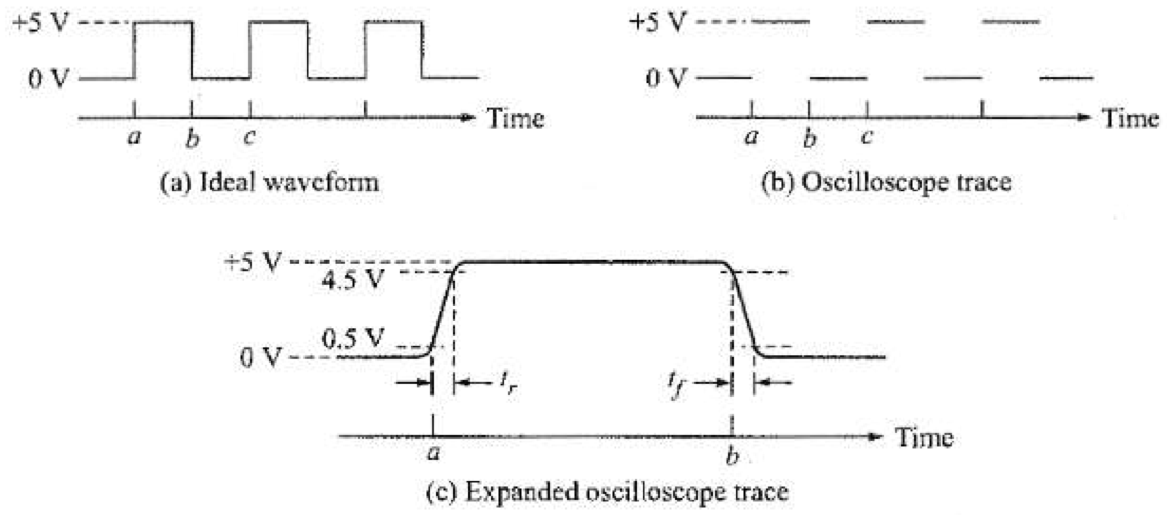


Fig. 7.3 Clock waveforms

- The second characteristic deals with the time required for the clock levels to change from high to low or vice versa. The transition of the clock from low to high at point *a* in Fig. 7.3a is shown by a vertical line segment.
- This implies a time of zero; that is, the transition occurs instantaneously-it requires zero time. The same is true of the transition time from high to low at point bin Fig. 7.3a. Thus an ideal clock has zero transition time.
- A nearly perfect clock waveform might appear on an oscilloscope trace as shown in Fig. 7.3b. At first glance this would seem to be two horizontal traces composed of line segments.
- On closer examination, however, it can be seen that the waveform is exactly like the ideal waveform in Fig. 7.3a if the vertical segments are removed. The vertical segments might not appear on the oscilloscope trace because the transition times are so small (nearly zero) and the oscilloscope is not capable of responding quickly enough.
- The vertical segments can usually be made visible by either increasing the oscilloscope "intensity," or by reducing the "sweep time."

- Figure 7.3c shows a portion of the wavefonn in Fig. 7.3b expanded by reducing the "sweep time" such that the transition times are visible. Clearly it requires some time for the waveform to transition from low to high-this is defined as the rise time *tr*- Remember, the time required for transition from high to low is de-fined as the fall time *If* It is customary to measure the rise and fall times from points on the wavefonn referred to as the *JO* and *90 percent* points.

- In this case, a 100 percent level change is 5.0 V, so 10 percent of this is 0.5 V and 90 percent is 4.5 V. Thus the rise time is that time required for the waveform to travel from 0.5 up to 4.5 V. Similarly, the fall time is that time required for the waveform to transition from 4.5 down to 0.5 V.
- The second characteristic deals with the time required for the clock levels to change from high to low or vice versa. The transition of the clock from low to high at point *a* in Fig. 7.3a is shown by a vertical line segment.
- This implies a time of zero; that is, the transition occurs instantaneously-it requires zero time. The same is true of the transition time from high to low at point bin Fig. 7.3a. Thus an ideal clock has zero transition time.
- A nearly perfect clock waveform might appear on an oscilloscope trace as shown in Fig. 7.3b. At first glance this would seem to be two horizontal traces composed of line segments.
- On closer examination, however, it can be seen that the waveform is exactly like the ideal waveform in Fig. 7.3a if the vertical segments are removed.
- The vertical segments might not appear on the oscilloscope trace because the transition times are so small (nearly zero) and the oscilloscope is not capable of responding quickly enough.
- The vertical segments can usually be made visible by either increasing the oscilloscope "intensity," or by reducing the "sweep time."
- Figure 7.3c shows a portion of the wavefonn in Fig. 7.3b expanded by reducing the "sweep time" such that the transition times are visible.
- Clearly it requires some time for the waveform to transition from low to high-this is defined as the rise time *tr*- Remember, the time required for transition from high to low is de-fined as the fall time *If* It is customary to measure the rise and fall times from points on

the wavefonn referred to as the *JO* and *90 percent* points. In this case, a 100 percent level change is 5.0 V, so 10 percent of this is 0.5 V and 90 percent is 4.5 V.

- Thus the rise time is that time required for the waveform to travel from 0.5 up to 4.5 V. Similarly, the fall time is that time required for the waveform to transition from 4.5 down to 0.5 V.
- Finally, the third requirement that defines an ideal clock is its frequency stability. The frequency of the clock should beoversteady and unchanging a specified period of time.
- Short-term stability can be specified by requiring that the clock frequency (or its period) not be allowed to vary by more than a given percentage over a short period of time-say, a few hours.
- Clock signals with short-tenn stability can be derived from straightforward electronic circuits as shown in the following sections.
- Long-term stability deals with longer periods of time~perhaps days, months, or years.
- Clock signals that have long-term stability are generally derived from rather special circuits placed in a heated enclosure (usually called an "oven") in order to guarantee close control of temperature and hence frequency.
- Such circuits can provide clock frequencies having stabilities better than a few parts in 10^9 per day.

FLIP-FLOPS

- The outputs of the digital circuits considered previously are dependent entirely on their inputs. That is, if an input changes state, output may also change state.
- However, there are requirements for a digital device or circuit whose output will remain unchanged, once set, even if there is a change in input level(s). Such a device could be used to store a binary number.
- A flip-flop is one such circuit, and the characteristics of the most common types of flip-flops used in digital systems are considered in this chapter.
- Flip-flops are used in the construction of registers and counters, and in numerous other applications.
- The elimination of switch contact bounce is a clever application utilizing the unique operating characteristics of flip-flops. In a sequential logic circuit flip-flops serve as key memory elements.

- Analysis of such circuits are done through tablestruth or characteristic equations of flip-flops. The analysis result is normally presented through state

***RS* FLIP-FlOPS**

- Any device or circuit that has two stable states is said to be *bistable.* For instance, a toggle switch has two states. stable It is either up or down, depending on the position of the switch as shown in Fig. 8.la.
- The switch is also said to have *memo,y* since it will remain as set until someone changes its position.
- *Aflip-fiop* is a bistable electronic circuit that has two stable states-that is, its output is either O or +5 V dc as shown in Fig. 8.1 b. The flip-flop also has memory since its output will remain as set l something is done to change it.
- As such, the flip-flop (or the switch) can be regarded as a memory device.
- In fact, any bistable device can be used to store one binary digit (bit). For instance, when the flip-flop has its output set O V dc.It can be regarded as storing a logic O and when its output is set at + 5 V de, as storing a logic l.
- flip-flop is often called a *latch,* since it will hold, or latch, in either stable state.

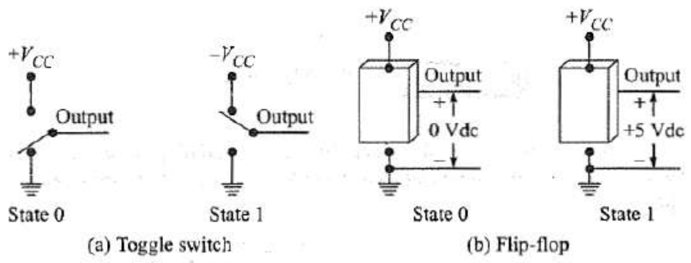


Fig. 8.1 Bistable devices

Basic Idea

- One of the easiest ways to construct a flip-flop is to connect two inverters in series as shown in Fig. 8.2a.
- The line connecting the output of inverter *B (INV B)* back to the input of inverter *A (INV* A) is referred to as the *feedback line.*

- For the moment, remove the feedback line and consider V_1 as the input and V_3 as the output as shown in Fig. 8.2b.
- There are only two possible signals in a digital system, and in this case we will define $L = 0 = 0$ V dc and $H = 1 = +5$ V dc.
- If V_1 is set to OV dc, then V_3 will also be OV dc. Now, if the feedback line shown in Fig. 8.2b is reconnected, the ground can be removed from *Vi*, and V_3, will remain at OVdc.
- This is true since once the input of INVA is grounded, the output of INV B will go low and can then be used to hold the input of INV lowA by using the feedback line. This is one stable state-$V_3 = 0$ Vdc.

Conversely, if *Vi* is +5 Vdc, V_3 will also be +5 Vdc as seen in Fig. 8.2c. The feedback line can again be used to hold *Vi* at + 5 V de since V_3 is also at + 5 V de. This is then the second stable state- *V3* = + 5 V de.

NOR-Gate latch

- The basic flip-flop shown in Fig. 8.2a can be improved by replacing the inverters with either NAND or NOR gates. The additional inputs on these gates provide a convenient means for application of input signals to

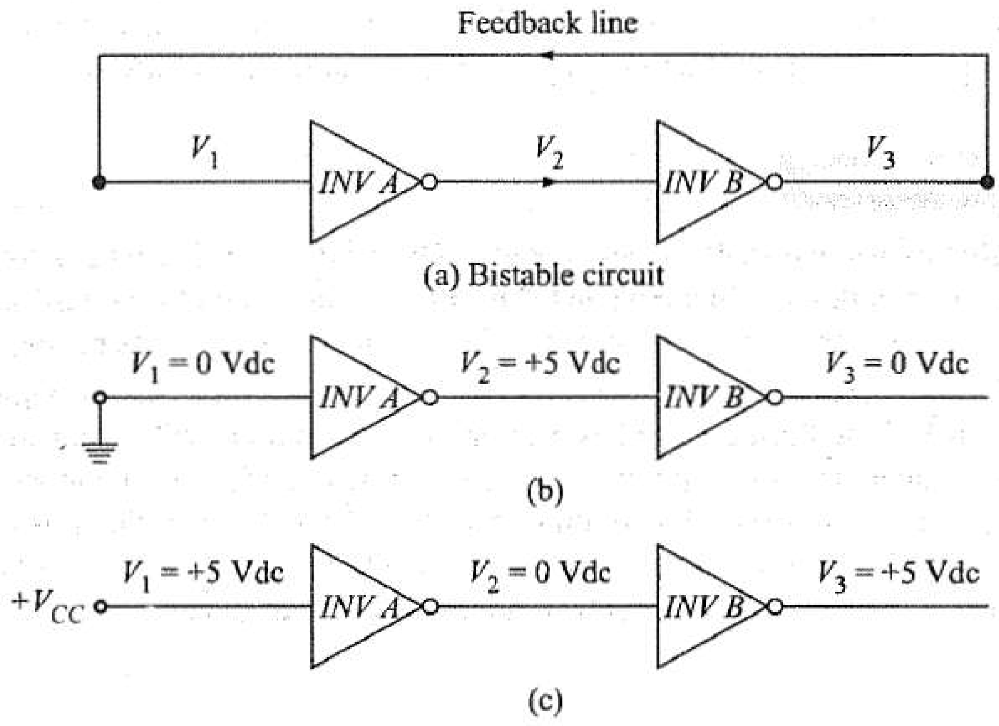


Fig. 8.2 **Bistable circuit**

switch the flip-flop from one stable state to the other. Two 2-input NOR gates are connected in Fig. 8.3a to fom1 a flip-flop.

Notice that if the two inputs labeled *R* and Sare ignored, this circuit will function exactly as the one shown in Fig. 8.. 2a.

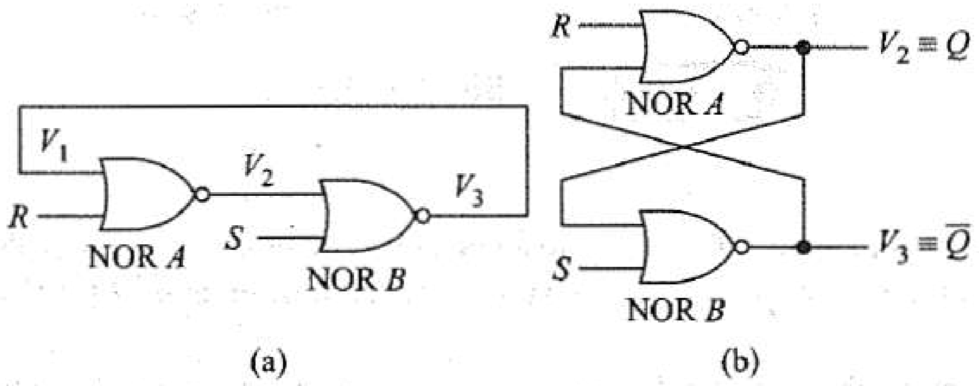


Fig. 8.3 **NOR-gate flip-flop**

- This circuit is redrawn in a more conventional form in Fig. 8.3b. The flip-flop actually has two outputs, defined in more general terms as *Q* and *Q*.

- It should be clear that regardless of the value of *Q*, its complement is *Q*. There are two inputs to the flip-flop defined as *R* and *S*.
- The input/output possibilities for this *RS* flip-flop are summarized in the truth table in Fig. 8.4.
- To aid in understanding the operation of this circuit, recall that an *H* = I at any input of a NOR gate forces its output to an *L* = 0.

1. The first input condition in the truth table is *R* = 0 and *S* = 0. Since a O at the input of a NOR gate has no effect on its output, the flip-flop simply remains in its present state; that is, *Q* remains unchanged.
2. The second input condition *R* = 0 and *S* = I forces the output of NOR gate *B* low. Both inputs to NORgate *A* are now low, and the NOR-gate output must be high. Thus a I at the *S* input is said to *SET* the flip-flop, and it switches to the stable state where *Q* = 1.
3. The third input condition is *R* = I and *S* = 0. This condition forces the output of NOR gate A low, and since both inputs to NOR gate B are now low,the output must be high. Thus a 1 at the input is said to RESET the flip-flop. and it switches to the stable state where $Q = 0$ (or $\overline{Q} = 1$).
4. The inlast input condition table, R=1 and S=1, is forbidden,as it forces the outputs of both NOR gates to the low state.
   - In other words, both Q=0 and $\overline{Q} = 0$ at the same time ! But this violates the basic definition of a flip-flop that requires *Q* to be the complement of *Q*, and so it is generally agreed never to impose this input condition. Incidentally, if this condition is for some reason, imposed and the next input is *R* = 0, *S* = 0 then the resulting state *Q* depends on propagation delays of two NOR gates.
   - If delay of gate *A* is less, i.e. it acts faster, then *Q* = 1 else it is 0. Such dependence makes the job of a design engineer difficult, as any replacement of a NOR gate will make *Q* unpredictable.

R	*S*	*Q*	Action
0	0	Last state	No change
0	1	1	SET
1	0	0	RESET
1	1	?	Forbidden

Fig. 8.4 **Truth table for a NOR-gate *RS* flip-flop**

NAND-Gate latch

- A slightly different latch can be constructed by using NAND gates as shown in Fig. 8.7. The truth table for this NAND-gate latch is different from that for the NOR-gate latch.
- We will call this latch an *RS* flip-flop. To understand how this circuit functions, recall that *a low on any input to a NAND gate will force its output high.* Thus a low on the S input will set the latch (Q = 1 and *Q* = 0).
- A low on the *R* input will reset it (Q = 0). If both *R* and S are high, the flip-flop will rem~n in its previous state. Setting both Rand S low simultaneously is forbidden since this forces both *Q* and *Q* high.

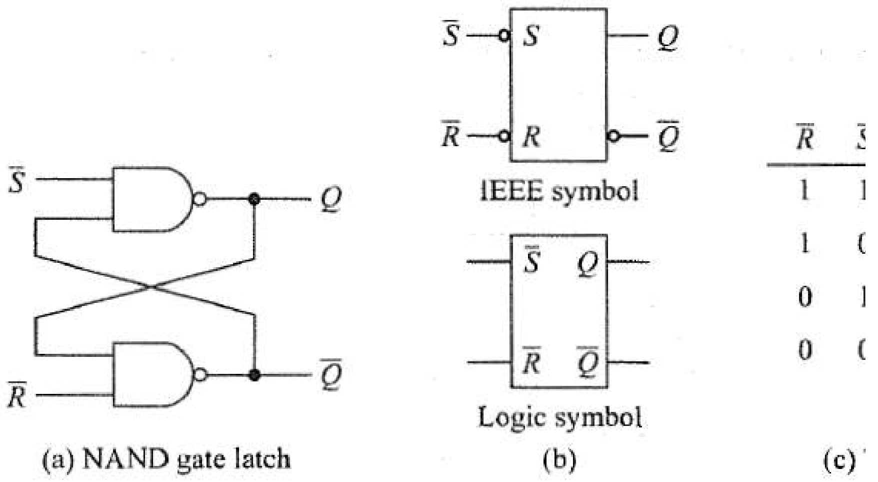


Fig. 8.7 **$\overline{R}\,\overline{S}$ flip-flop**

EDGE-TRIGGERED *D* FLIP FLOP

Although the *D* latch is used for temporary storage in electronic instruments, an even more popular kind of *D* flip-flop is used in digital computers and systems. This kind of flip-flop samples the data bit at a unique point in time.

- Figure 8.21 shows a positive pulse-forming circuit at the input of a *D* latch. The narrow positive pulse (PT) enables the AND gates for an instant.
- The effect is to activate the AND gates during the PT of C, which is equivalent to sampling the value of *D* for an instant. At this unique point in time, *D* and its complement hit the flip-flop inputs, forcing *Q* to set or reset (unless *Q* already equals *D).*
- Again, this operation is called *edge triggering* because the flip-flop responds only when the clock is in transition between its two voltage states. The triggering in Fig. 8.21 occurs on the positive-going edge of the clock; this is why it's referred to as *positive-edge triggering.*

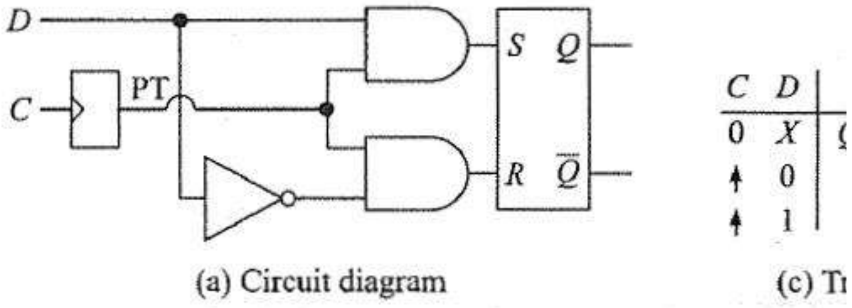


Fig. 8.21 **Positive-edge-triggered *D* flip-**

- The truth table in Fig. 8.21 b summarizes the action of a positive-edge-triggeredD flip-fl.op. When the clock is low, *D* is a don't care and *Q* is latched in its last state.
- On the leading edge of the clock (PT), designated by the up arrow, the data bit is loaded into the flip-flop and *Q* takes on the value of *D.*

- When power is first applied, flip-flops come up in random states. To get some computers started, an operator h<1s to push a RESET button. This sends a CLEAR or RESET signal to all flip-flops.

  Also, it's necessary in some digital systems to preset (synonymous with set) certain flip-flops.

- Figure 8.22 shows how to include both functions in a *D* flip-flop. The edge triggering is the same as previously described. Depressing the RESET button will set *Q* to I with the first PT of the clock. *Q* will remain high as long as the button is held closed.
- The first PT of the clock after releasing the button will set *Q* according to the *D* input. Furthermore, the OR gates allow us to slip in a high PRESET or a high CLEAR when desired. A high PRESET forces *Q* to equal 1; a high CLEAR resets *Q* to 0.

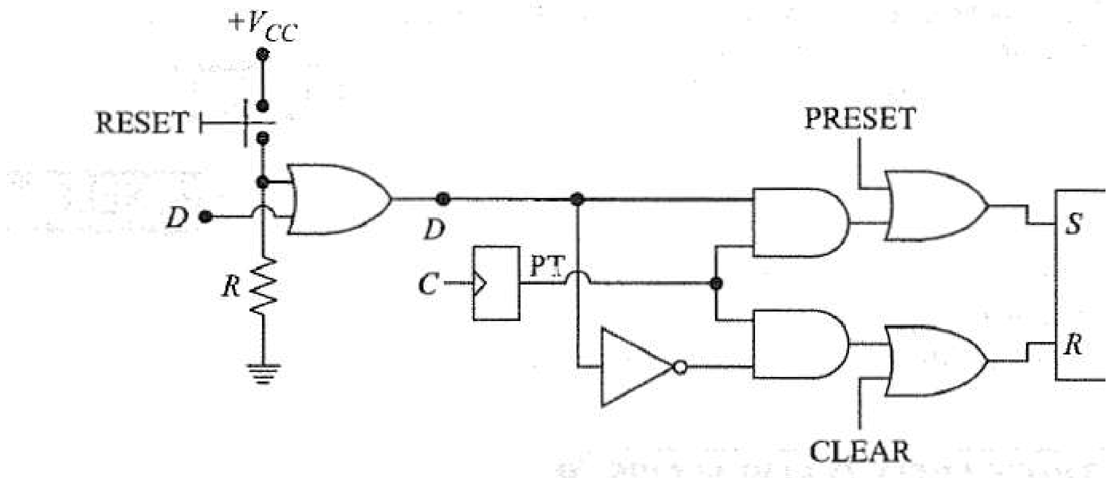


Fig. 8.22 **PRESET and CLEAR functions**

- The PRESET and CLEAR are called *asynchronous inputs* because they activate the flip-flop independentlyof the clock.
- On other hand, the *D* input is a synchronous input because it has an effect only with PTs of the clock.
- Figure 8.23a is the IEEE symbol for a positive-edge-triggered *D* flip-flop. The clock input has a small triangle to serve as a reminder of edge triggering.
- When you see this symbol, remember what it means; the *D* input is sampled and stored on PTs of the clock.
- Sometimes, triggering on NTs of the clock is better suited to the application. In this case, an internal inverter can complement the clock pulse before it reaches the AND gates.
- Figure 8.23b is the symbol for a negative-edge-triggered *D* flip-flop. The bubble and triangle symboiize the negative-edge triggering.

- Figureis 8.23c another commercially available *D* flip-flop (the 54/74175 or 54/74LS 175). Besides having positive-edge triggering, it has an inverted CLEAR input This means that a low CLR resets it.
  - The 54/74175 has four of these *D* flip-flops in a single 16-pin dual in-line package (DIP), and it's referred to as a *quad D-type flip-flop with clear*.

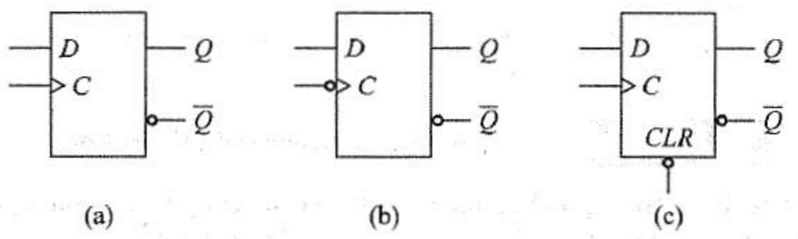


Fig. 8.23 *D*flip-flopsymbols: (a) Positive-edge-triggered, (b) Negative-edge-triggered, (c) Positive-edge-triggered with active low clear

EDGE-TRIGGERED *JK* FLIP FLOP

- Setting *R* = *S* = I with an edge-triggered *RS* flip-flop forces both *Q* and *Q* to the same logic level. This is an *illegal* condition, and it is not possible to predict the final state of *Q*.
- The *JK* flip-flop accounts for this illegal input, and is therefore a more versatile circuit.
- Among other things, flip-flops can be used to build counters. Counters can be used to count the number of PTs orNTs of a clock. For purposes of counting, theJK flip-flop is the ideal element to use.
- There are many commercially available edge-triggered *JK* flip-flops. Let's see how they function.

Positive-Edge-Triggered *JK* Flip-Flops

In Fig. 8.24, the pulse-forming box changes the clock into a series of positive pulses, and thus this circuit will be sensitive to PTs of the clock. The basic circuit is identical to the previous positive-edge-triggered *RS* flip-flop, with two important additions:

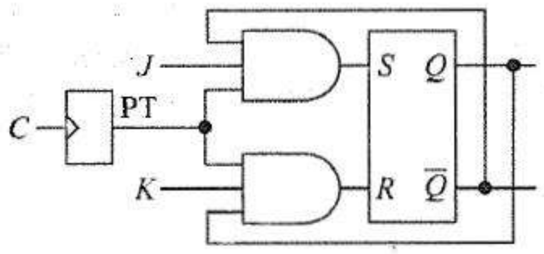
1. The *Q* output is connected back to the input of the lower AND gate.

2. The *Q* output is connected back to the input of the upper AND gate.

This cross-coupling from outputs to inputs changes the *RS* flip-flop into a *JK* flip-flop. The previous *S* input is now labeled *J,* and the previous *R* input is labeled *K*. Here's how it works:

1. When *J* and Kare both low, both AND gates are disabled. Therefore, clock pulses have no effect. This first possibility is the initial entry in the truth table. As shown, when *J* and *K* are both Os, *Q* retains its last value.

2. When *J* is low and *K* is high, the upper gate is disabled, so there's no way set the flip-flop.



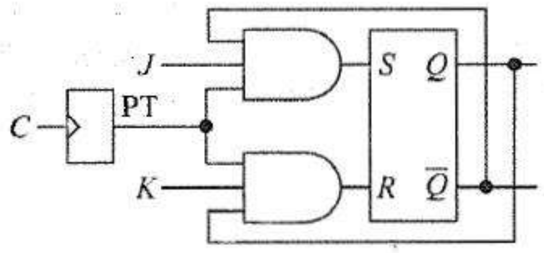
(a) One way to implement a *JK* flip-flop

C	J	K	Q_{n+1}	Action
↑	0	0	Q_n (last state)	No change
↑	0	1	0	RESET
↑	1	0	1	SET
↑	1	1	$\overline{Q}_n$ (toggle)	Toggle

(b) Truth table

Fig. 8.24 **A positive-edge-triggered *JK* flip-flop**

The only possibility is reset. When *Q* is high, the lower gate passes passes a RESET pulse as soon as the next positive.



(a) One way to implement a *JK* flip-flop

C	J	K	Q_{n+1}	Action
↑	0	0	Q_n (last state)	No change
↑	0	1	0	RESET
↑	1	0	1	SET
↑	1	1	$\overline{Q}_n$ (toggle)	Toggle

(b) Truth table

Fig. 8.24 **A positive-edge-triggered *JK* flip-flop**

clock edge arrives. This forces *Q* to become low (the second entry in the truth table). Therefore, *J* = 0 and *K* = I means that the next PT of the clock resets the flip-flop (unless *Q* is already reset).

3. When *J* is high and *K* is low, the lower gate is disabled, so it's impossible to reset the flip-flop. But you can set the flip flop as follows. When *Q* is low, *Q* is high; therefore, the upper gate passes a SET pulse on the next positive clock edge.
4. This drives *Q* into the high state (the third entry in the truth table). As you can see, *J* = 1 and *K* = 0 means that the next PT of the clock sets the flip-flop (unless *Q* is already high).

5. When *J* and Kare both high (notice that this is the forbidden state with an *RS* flip-flop), it's possible to set or reset the flip-flop. If *Q* is high, the lower gate passes a RESET pulse on the next PT. On the

- other hand, when *Q* is low, the upper gate passes a SET pulse on the next PT. Either way, *Q* changes to the complement of the last state (see the truth table). Therefore, *J* = I and *K* = I mean the flip-flop will *toggle* (switch to the opposite state) on the next positive clock edge.

Propagation delay prevents the *JK* flip-flop from racing (toggling more than once during a positive dock edge). Here's why. In Fig. 8.24, the outputs change after the PT of the clock. By then, the new *Q* and *Q* values are too late to coincide with the PTs driving the AND gates. For instance, if *tP* = 20 ns, the outputs change approximately 20 ns after the leading edge of the clock.

- If the PTs are narrower than 20. ns, the returning *Q* and *Q* arrive too late to cause false triggering.
- Figure 8.25a shows a symbol for a *JK* flip-flop of any design. When you see this on a schematic diagram, remember that on the next PT of the clock:

I. *J* and *K* low: no change of *Q*.

2. *J* low and *K* high: *Q* is reset low.

3. *J* high and *K* low: *Q* is set high.

4. *J* and *K* both high: *Q* toggles to opposite state.

You can include OR gates in the design to accommodate PRESET and CLEAR as was done earlier. Figure 8.25b gives the symbol for a *JK* flip-flop with *PR* and *CLR*. Notice that it is negative-edge-triggered and requires a low *PR* to set it or a low *CLR* to reset it.

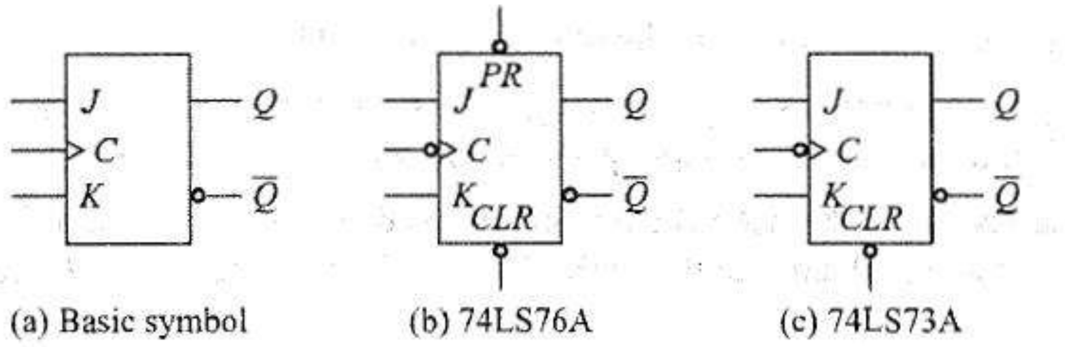


Fig. 8.25 *JK* flip-flop symbols

Figure 8.25c is another commercially available *JK* flip-flop. Itis negative-edge-triggeredrequiresand a low *CLR* to reset it. The output *Q* reacts immediately to a *PR* or *CLR* signal. Both *PR* and *CLR* are asynchronous, and they override all other input signals.

***JK* MASTER-SLAVE. FLIP-FLOPS**

Figure 8.28 shows one way to build a *JK* master-slave flip-flop. Here's how it works

1. To begin with, the master is positive-level- triggered and the slave is negative-level-triggered. Therefore, the master responds to its *J* and *K* inputs before the slave. If*J* = 1 and *K* = 0, the master sets on the positive clock transition. The high *Q* output master of the drives the J, input of the slave, so on the negative clock transition, the slave sets, copying the action of the master.

2.If *J* = 0 and *K* = I, the master resets on the PT of the clock. The high *Q* output of the master goes to the *K* input of the slave. Therefore, the NT of the clock forces the slave to reset. Again, the slave has copied the master.

3. If the master's *J* and *K* inputs are both high, it toggles on the PT of the clock and the slave then toggles on the clock NT. Regardless of what the master does, therefore, the slave copies it: if the master sets, the slave sets; if the master resets, the slave resets.

4.If *J* = *K* = 0, the flip-flop is disabled and *Q* remains unchanged.

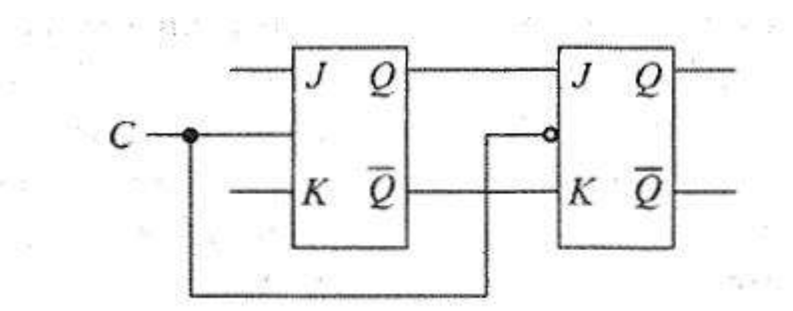


Fig. 8.28 **Master-slave flip-flop**

Figure 8.25c is another commercially available *JK* flip-flop. It is negative-edge-triggered requires and a low *CLR* to reset it.

- The output *Q* reacts immediately to a *PR* or *CLR* signal. Both *PR* and *CLR* are asynchronous, and they override all other input signals.
- The symbol for a 7476 master-slave flip-flop is by taking *PR* low, or it can be reset to *Q* = *L* by taking *CLR* low. These two inputs take precedence over all other signals!
- There is something different however. First of all, notice that the clock (C) is not edge-triggered.
- The master does in fact change state when C goes high.

However, while the clock is high, any change in *J or K* will immediately affect the master flip-flop. In other words, the master is transparent while the clock is and thus *J* and *K* must be static during this time. The truth table in Fig. 8.29b reveals this action by means of the pulse symbol (⎍).

- Second, the symbol l appearing next to the *Q* and the *Q* outputs is the IEEE designation for a *postponed output.* In this case, it means *Q* does not change state until the clock makes an NT. In other words, the contents of the master are shifting into the slave on the clock NT, and at this time *Q* changes state.
- To summarize: The master is set according to *J* and *K* while the clock is high; the contents of the master are then shifted into the slave (Q changes state) when the clock goes low. This particular flip-flop might be referred to as *pulse-triggered,* to distinguish it from the edge-triggered flip-flops previously discussed.

- There are numerous pulse-triggered master-slave flip-flops in use today. However, because edge-triggered flip-flops have overcome the restriction of holding *J* and *K* static when the clock is high, most new designs incorporate edge-triggered devices.

- Some of the more popular pulse-triggered flip-flops you might encounter include the 7473, 7476, and 7478. Their more modem, edge-triggered counterparts include the 74LS73A, the 74LS76A, and the 74LS78A.

UNIT -V

REGISTERS

A register is a very important **digital building block**. A data register is often used to momentarily store binary information appearing at the output of an encoding matrix.

A register might be used to accept input data from an alphanumeric keyboard and then present this data at the input of a microprocessor chip.

Similarly, registers are often used to momentarily store **binary data** at the output of a **decoder**.

For instance, a register could be used to accept output data from a **microprocessor chip** and then present this data to the circuitry used to drive the display on a **CRT** screen.

Thus registers /form a very important link between the main digital system and the input-output channels.

A **universal asynchronous receiver transmitter** (UART) is a chip used to exchange data in a microprocessor system. The UART is constructed using registers and some control logic.

A **binary register** also forms the basis for some very important arithmetic operations. For example, the operations of complementation, multiplication, and division are frequently implemented by means of a register.

A **shift register** can also be connected to form a number of different types of counters.

TYPES OF REGISTERS

A register is simply a **group of flip-flops** that can be used to store a binary number. There must be one flip-flop for each bit in the binary number.

A group of flip-flops connected to provide either or both of these functions is called a shift register.

The bits in a binary number (let's call them the data) can be moved from one place to another in either of two ways.

The first method involves shifting the data 1 bit at a time in a serial fashion, beginning with either the **most significant bit** (MSB) or the **least significant bit** (LSB).

This technique is referred to as serial shifting. The second method involves shifting all the data bits simultaneously and is referred to as parallel shifting.

There are two ways to shift data into a register **(serial or parallel)** and similarly two ways to shift the data out of the register.

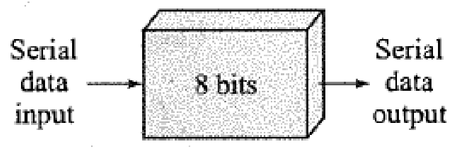
For instance :

Serial in- serial out -54/74LS91,8 bits

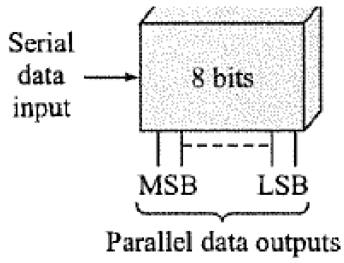
Serial in-parallel out-54/74164, 8 bits

Parallel in-serial out-54/74165, 8 bits

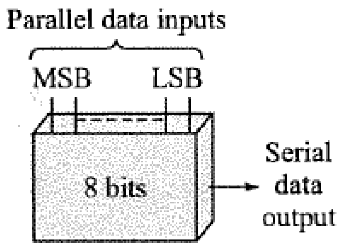
Parallel in-parallel out-54/74198, 8 bits



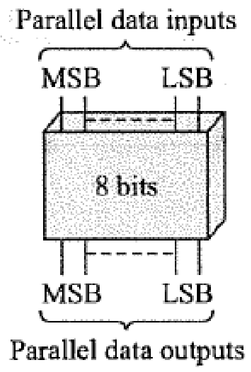
(a) Serial in–serial out



(b) Serial in–parallel out



(c) Parallel in–serial out



(d) Serial in–parallel out

The methods for shifting data in either a serial or parallel fashion .Data shifting techniques and methods for constructing the four different types of registers are discussed in the following sections.

1.SERIAL IN-SERIAL OUT

In this section we discuss how data is serially **entered or exited** from a shift register. The flip-flops used to construct registers are usually edge-triggered JK, SR or D types.

We begin our discussion with shift registers made from D type flip-flops and then extend the idea to other types.

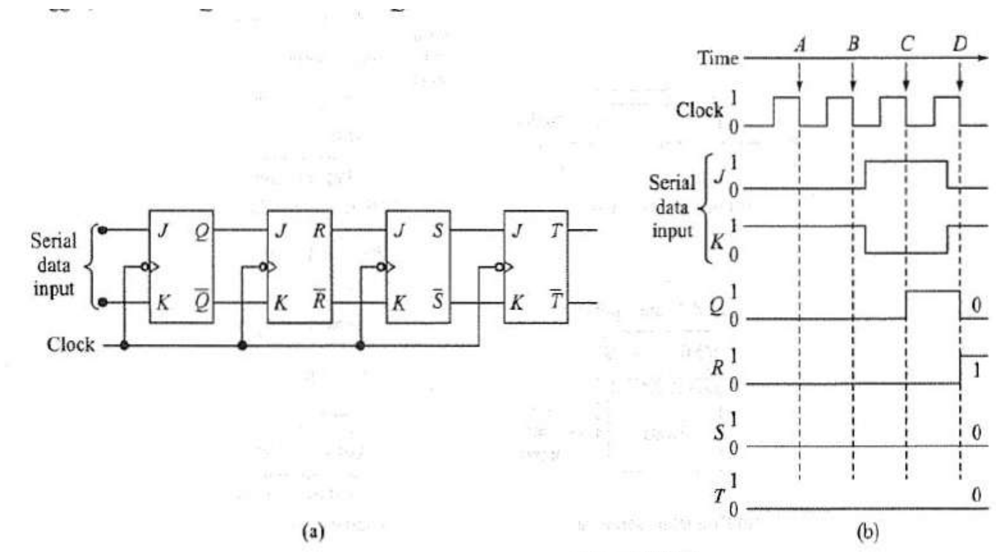
Consider four D flip-flops connected as shown in Fig. (a) Forming 4-bit shift register.

A common clock provides trigger at its negative edge to all the flip-flops. As output of one D flip-flop is com1ectedinputto of the next at every Clock trigger data stored in one flip-flop is transferred to the next.

For this circuit transfer takes place like this Q ~ R, R ~ S, S ~ T and serial data input is transferred to Q. Let us see how actual data transfer takes place by an example.

At clock edge A, flip-flop Q has input O from serial data in D, flip-flop R has input O from output of Q, flip-flop S has input O from output of Rand flip-flop .

The input from output of S. When clock triggers, these inputs get transferred to corresponding flip-flop outputs simultaneously so that QRST= 0000. Thus at clock trigger, values at DQRS is transferred to QRST.

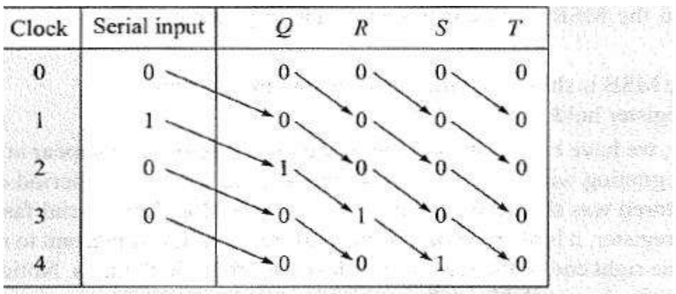


At clock edge B, serial data in= 0, i.e. DQRS = 0000. So after NT at B, QRST= 0000data. Serial becomes 1 in next clock cycle.

At clock edge C, DQRS = 1000 and after NT QRST= 1000. Serial data goes to O in next clock cycle such that at clock edge D, DQRS = 0100 and after NT QRST = 0100. Example 5 .1 will give another illustration of such data transfer.

Example: Show how a number 0100 is entered serially in a shift register shown in Fig. a using state table.

Solution Figure presents the state table. The timing diagram corresponding to this is discussed in this section. Note how the data flow across the flip-flops is highlighted by arrow direction.



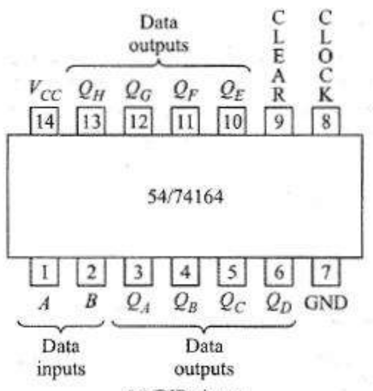
Clock	Serial input	Q	R	S	T
0	0	0	0	0	0
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4		0	0	1	0

Data transfer through serial input in a shift register

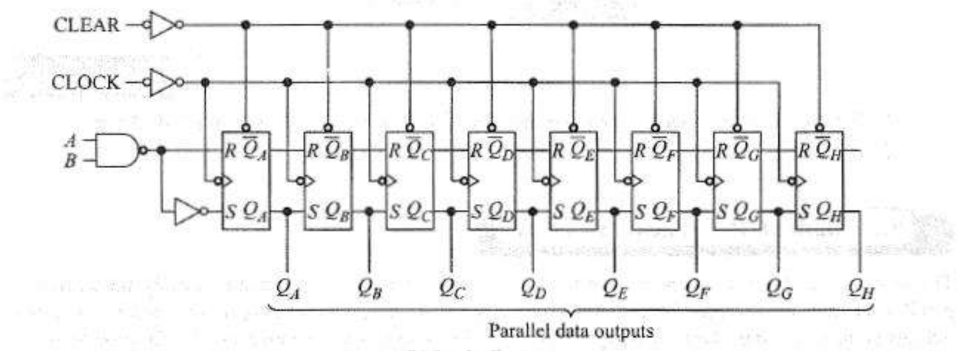
2.<u>SERIAL IN -PARALLEL OUT</u>

The second type of register mentioned is one in which data is shifted in serially, but shifted out in parallel. In order to shift the data out in parallel, it is simply necessary to have all the data bits available as outputs at the same time.

This is easily accomplished by connecting the output of each flip-flop to an output pin. For instance, an 8-bit shift register would have eight output lines-one for each flip-flop in the register.



(a) DIP pinout



(b) Logic diagram

The 54/7 4164 is an 8-bit serial input-parallel output shift register. The pin out and logic diagram for this device are given in Fig. (a)

Two exceptions:

(1) the true side of each flip-flop is available as an output-thus all 8 bits of any number stored in the register are available simultaneously as an output (this is a parallel data output);

(2) each flip-flop has an asynchronous clear input. Thus a low level at the clear input to the chip (pin 9) is applied through an amplifier and will reset (clear) every flip-flop.

Let's take a look at the gated serial inputs A and B. Suppose that the serial data is connected to A; then B can be used as a control line.

Here's how it works:

B is Held High The NAND gate is enabled and the serial input data passes through the NAND gate inverted. The input data is shifted serially into the register.

B is Held Low The NAND-gate output is forced high, the input data stream is inhibited, and the next positive
clock transition will shift a O into the first flip-flop. Each succeeding positive clock transition will shift another O into the register. After eight clock pulses, the register will be full of zeros!

Example:

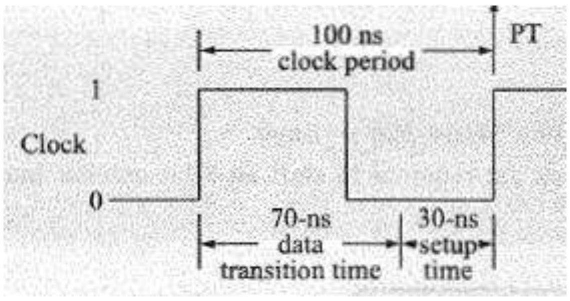
How long will it take to shift an 8-bit number into a 54164 shift register if the clock is set at IO MHz?

Solution : A minimum of eight clock periods will be required since the data is entered serially. One clock period is 100 ns, so it will require 800 ns minimum.

Example :

For the register in Example 9.4, when must the input data be stable? When can it be changed?

Solution: The data must be stable from 30 ns before a positive clock transition until the positive transition occurs. This leaves 70 ns during which the data may be changing.

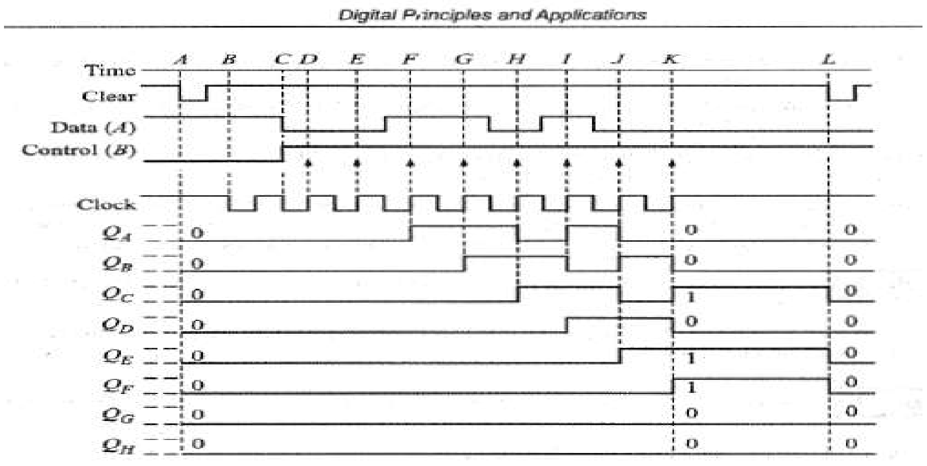


The waveforms shown in Fig. 9.9 show the typical response of a 54/74164. The serial data is input at A (pin 1), while a gating control signal is applied at B (pin 2). The first clear pulse occurs at time A and simply resets all flip-flops to 0.

The clock begins at time B, but the first PT does nothing since the control line is low. At time C the control line goes high, and the first data bit (a 0) is shifted into the register at time D.

Finally, another clear pulse occurs at time L, the flip-flops are all reset to zero, and another shift sequence may begin. Incidentally, the register can be cleared by holding the control line at B low and allowing the clock to run for eight PTs. This simply shifts eight Os into the register.

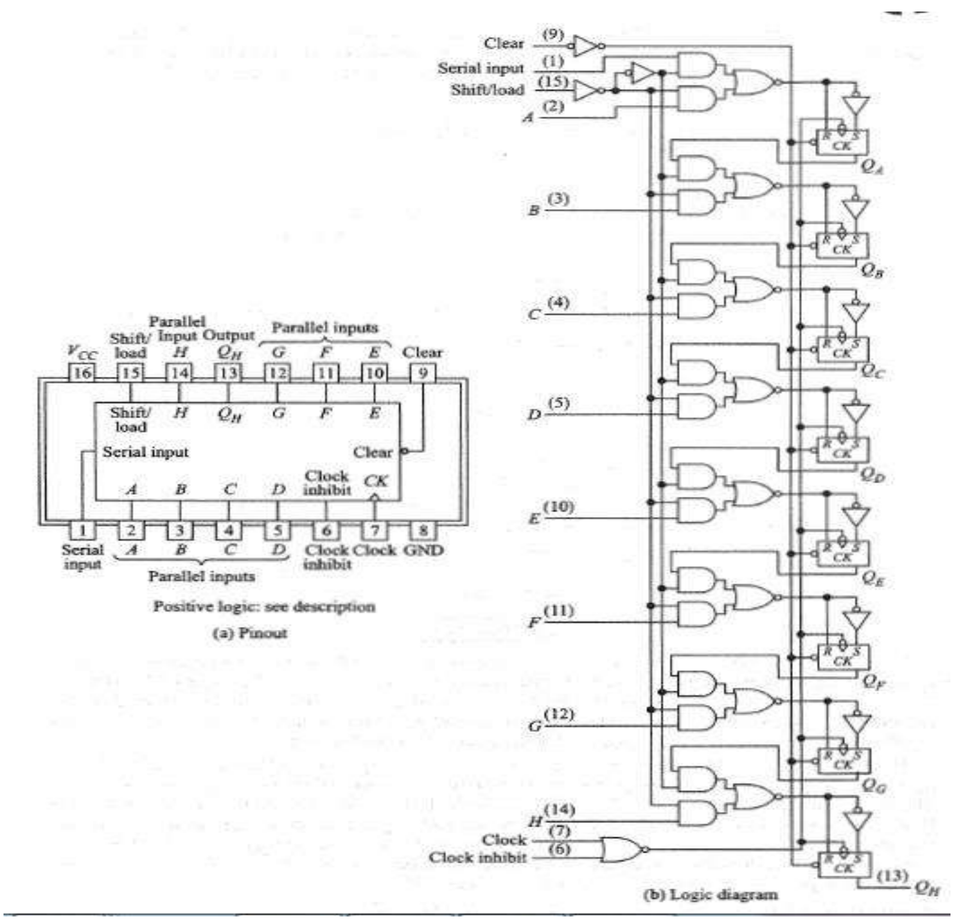
Let's analyze one of these circuits by starting with the RS flip-flops and then adding logic blocks to Accomplish.

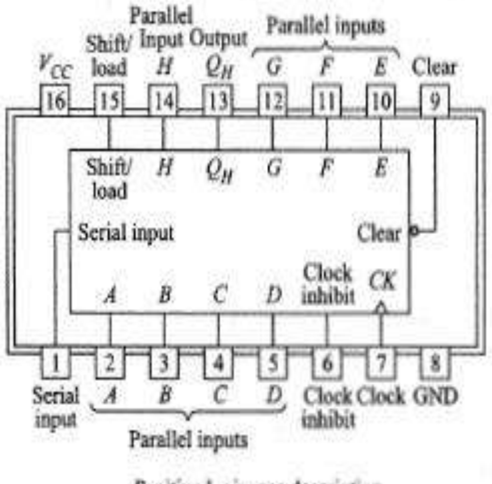


3.PARALLEL IN –SERIAL OUT

The ideas necessary for shifting data into and out of a register in serial have been developed. We can now use these same ideas to develop methods for the parallel entry of data into a register.

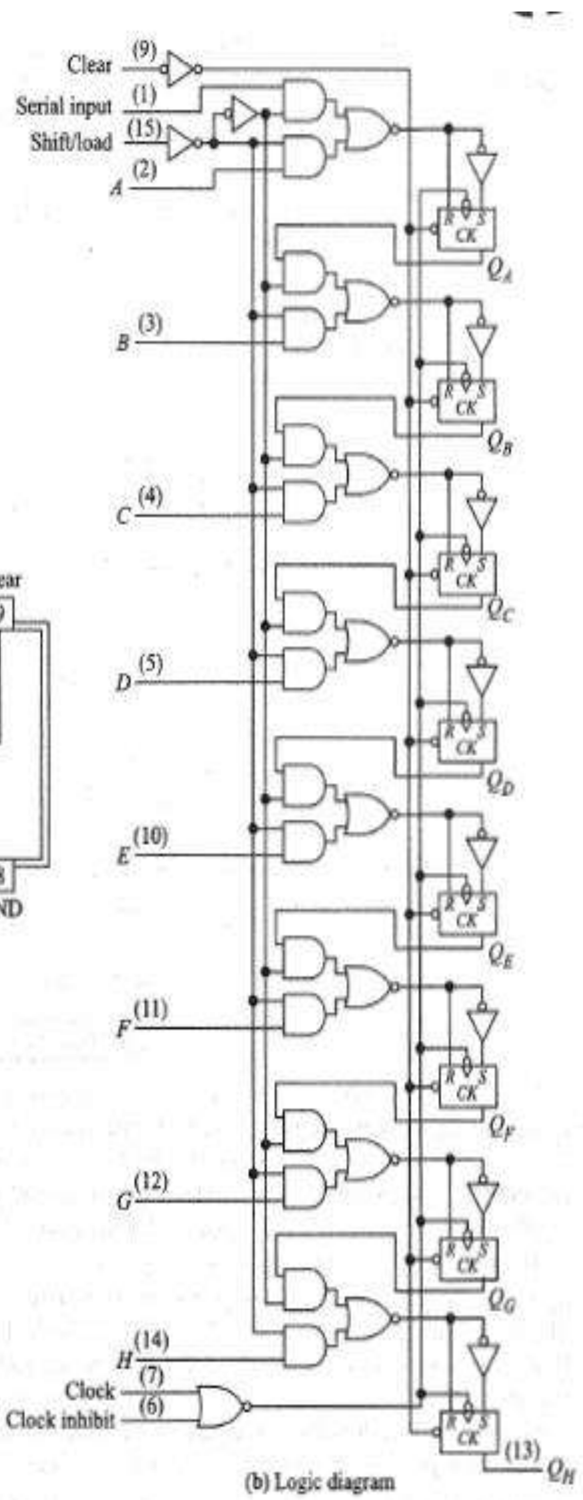
The 54/74166, for instance, is an 8-bit shift register, and the same circuit is repeated eight times. So, it's necessary to study only one of the eight circuits, and that's what we'll do here.


Clear (9)
Serial input (1)
Shift/load (15)
A (2)
B (3)
C (4)
D (5)
E (10)
F (11)
G (12)
H (14)
Clock (7)
Clock inhibit (6)
Q_A
Q_B
Q_C
Q_D
Q_E
Q_F
Q_G
(13) Q_H
R S CK
V_{CC}
Shift/load
Parallel Input H
Output Q_H
Parallel inputs G F E
Clear
Serial input
A B C D
Clock inhibit
CK
Serial input
Parallel inputs
Clock
GND
Positive logic: see description
(a) Pinout
(b) Logic diagram



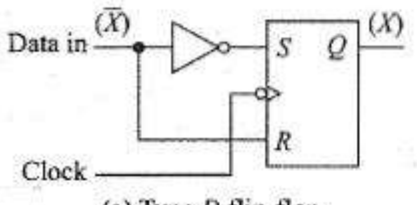
Positive logic: see description

(a) Pinout

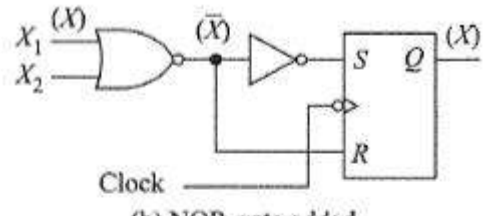


(b) Logic diagram

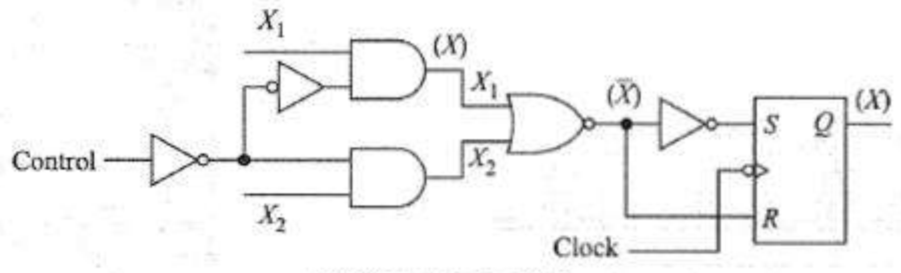
First recognize that the clocked RS flip-flop and the attached inverter given in Fig. 9.1 la fonn a type D flip-flop. If a data bit Xis to be clocked into the flip-flop, the complement of X must be present at the input. For instance, if X = 0, then R = 0 and S = 1, and a 1 will be clocked into the flip-flop when the clock transitions.



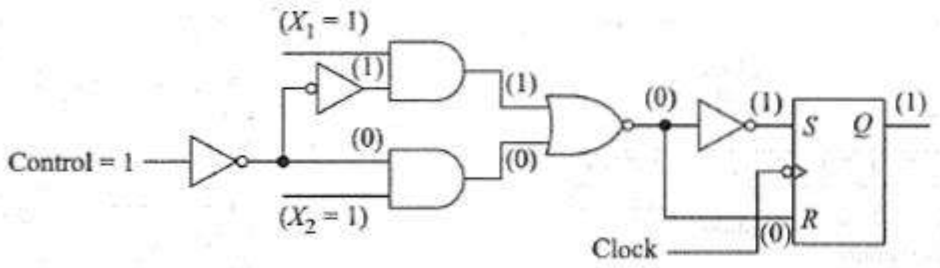
(a) Type *D* flip-flop



(b) NOR-gate added



(c) Control logic added



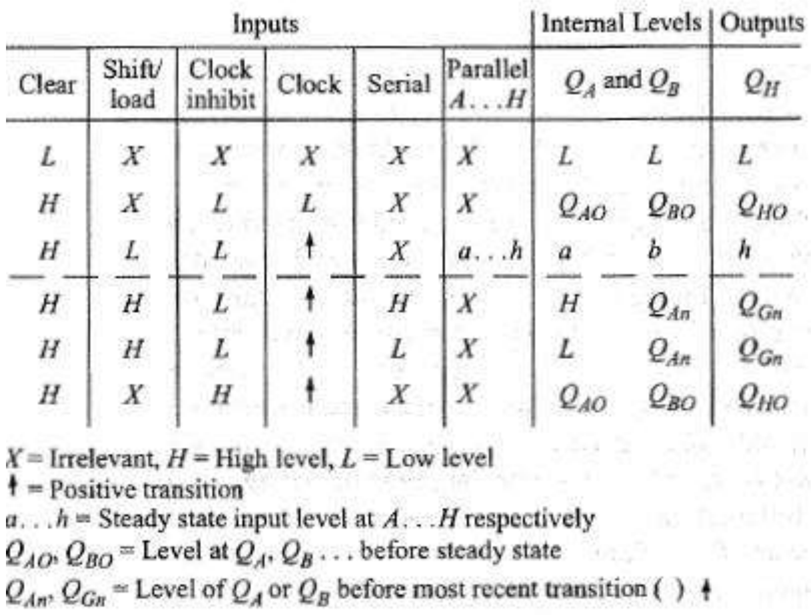
Now, add a NOR gate as shown in Fig. 9.11 b. If one leg of this NOR gate is at ground level, a data bit X at the other leg is simply inverted by the NOR gate. For instance, if X = 1, then at the output of the NOR gate X = 0, allowing a 1 to be clocked into the flip-flop.

This NOR gate offers the option of entering data from two different sources, either X 1 or X2. Holding X2 at ground will allow the data at X1 to be shifted into the flip-flop; conversely, holding X 1 at ground will allow data at X2 to be shifted in.

To summarize:

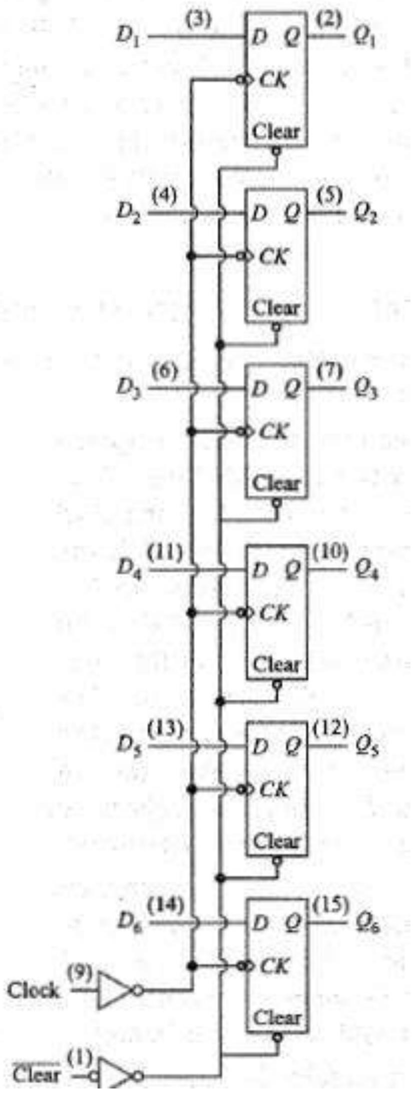
CONTROL is High Data bit atX1 will be shifted into the flip-flop at the next clock transition.
CONTROL is Low Data bit atX2 will be shifted into the flip 0 flop at the next clock transition.
For the circuit in write the logic levels present gate leg if CONTROL=1 on each X1 = 1, andX2 = I

Solution The correct 1evels are given in parentheses in Fig, The data value 1 atX1 is shift into the flip-flop when the clock transitions.



Inputs						Internal Levels		Outputs
Clear	Shift/ load	Clock inhibit	Clock	Serial	Parallel $A \ldots H$	Q_A and Q_B		Q_H
L	X	X	X	X	X	L	L	L
H	X	L	L	X	X	Q_{AO}	Q_{BO}	Q_{HO}
H	L	L	$\uparrow$	X	$a \ldots h$	a	b	h
H	H	L	$\uparrow$	H	X	H	Q_{An}	Q_{Gn}
H	H	L	$\uparrow$	L	X	L	Q_{An}	Q_{Gn}
H	X	H	$\uparrow$	X	X	Q_{AO}	Q_{BO}	Q_{HO}

X = Irrelevant, H = High level, L = Low level
$\uparrow$ = Positive transition
$a \ldots h$ = Steady state input level at $A \ldots H$ respectively
Q_{AO}, Q_{BO} = Level at Q_A, Q_B . . . before steady state
Q_{An}, Q_{Gn} = Level of Q_A or Q_B before most recent transition ( ) $\uparrow$



4. PARALLEL IN-PARALLEL out

That data can be shifted either in to out or of the register in parallel. In fact, simply adding an output line from each flip-flop in the 54/74166

The 54/74174

The 74174 in Fig. 9.13 is an example of a parallel in-parallel out register. The Texas Instruments data sheet refers to it as a hex D-type flip-flop with clear. It is simply a parallel arrangement of six D-type flip-flops.

Each flip-flop is negative-edge-triggered, and thus a PT will shift data into the register. The six data bits, D 1 through D6 are all shifted into the register in parallel. The stored data is immediately available, in parallel, at the outputs, Q1 through Q6.

This type of register is simply used to store data, and is sometimes called a data register, or data latch. Notice that it is not possible to shift stored data either to the right or to the left.

A low level at the clear input will immediately reset all flip-flops low. The clear input is asynchronous-that is, it can be done at any time and it takes precedence over all other inputs.

The 54/74198

The 54/74198 is an 8-bit TTL MSI having both parallel input and parallel output capability. The DIP pin out for this device is given in Fig. 9.14 on the next page.

It uses positive edge-triggered flip-flops, as indicated by the small triangle at pin 11. Notice that a 24-pin package is required since 16 pins are needed just for the input and output data lines.

Not only does this chip satisfy the parallel input-output requirements; it can also be used to shift data through the register in either direction-referred to as shift right and shift left.

All the registers previously discussed have the ability to shift right, that is, to shift data serially from the data input flip-flop toward the right, or from a flip-flop QA toward flip-flop Qs. We now need to consider how to shift left.

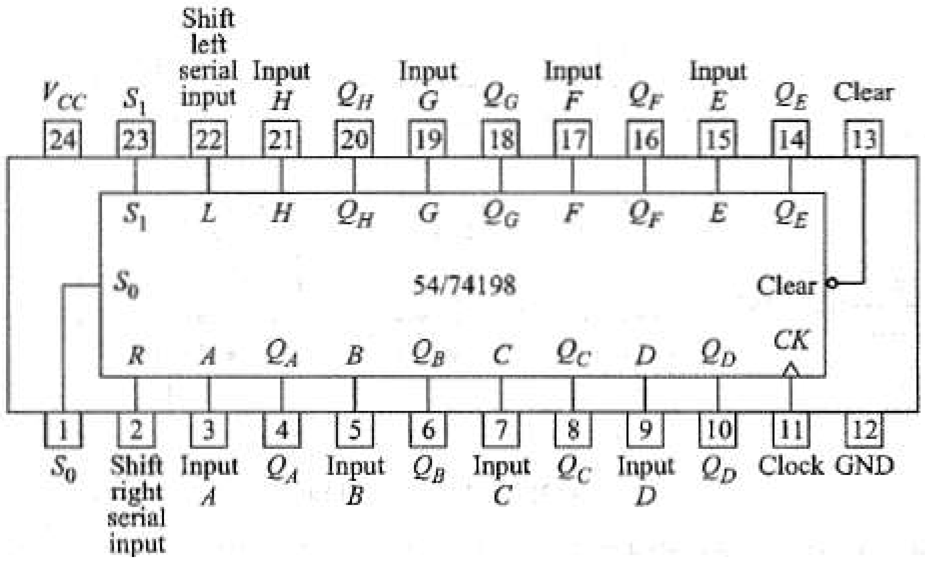


Fig -54/74198, 8-bit shift register. Parallel input-parallel output

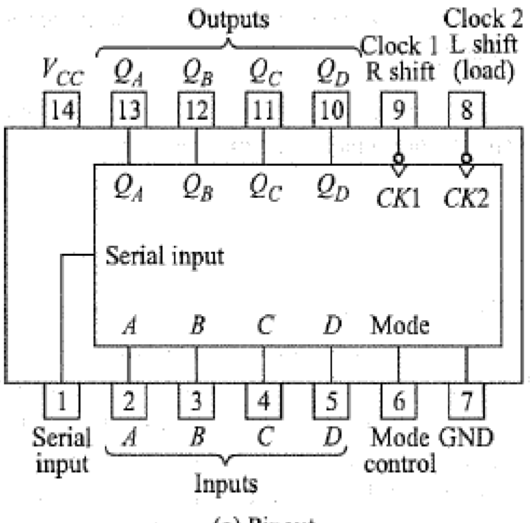
There are a number of 4-bit parallel in-,-parallel out shift registers available since they can be conveniently packaged in a 16-pin DIP.

An 8-bit register can be created by either connecting two 4-bit registers in series or by manufacturing the two 4-bit registers on a single chip and placing the chip in a 24-pin package (such as the 54/74198). Let's analyze a typical 4-bit register, say, a 5417495A.

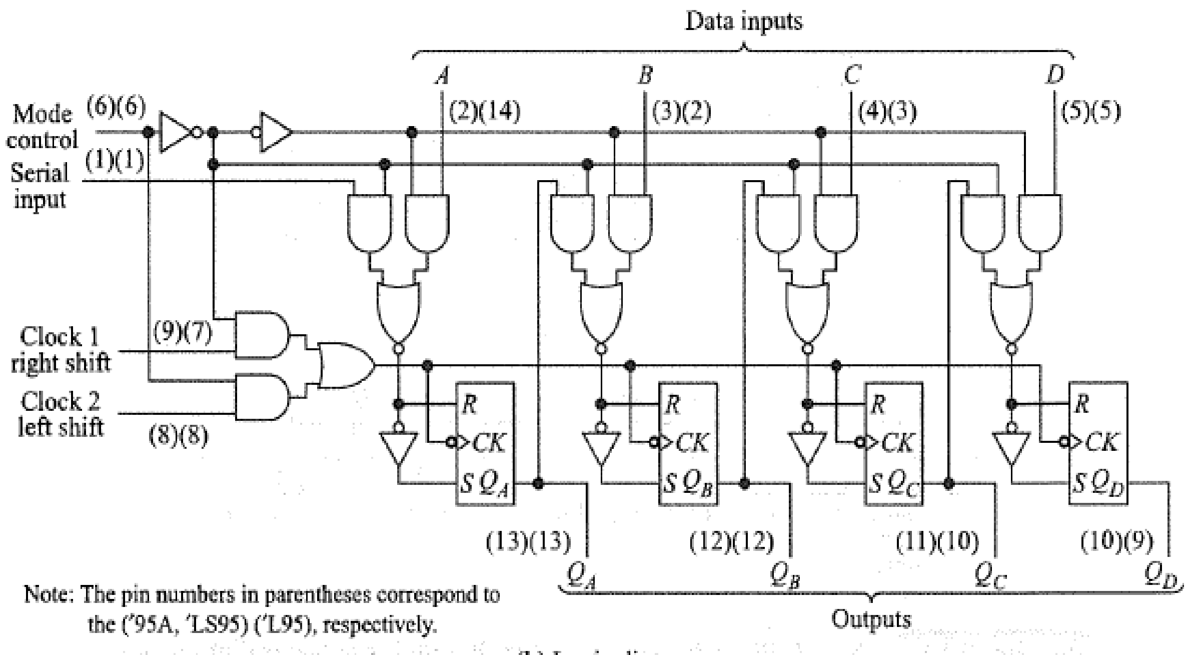
The data sheet for the 5417495A describes it as a 4-bit parallel-access shift register. It also has serial data input and can be used to shift data to the right (from QA toward Qs) and in the opposite direction-,to the left. The DIP pin out and logic diagram are given in Fig.

The basic flip-flop and control logic used here are exactly the same as used in the 54/74164 as shown in Fig.

The parallel data outputs are simply the Q sides of each of the four flip-flops in the register. In fact, note that the output QD could be used as a serial output when data is shifted from left to right through the register (right shift).



(a) Pinout



(b) Logic diagram

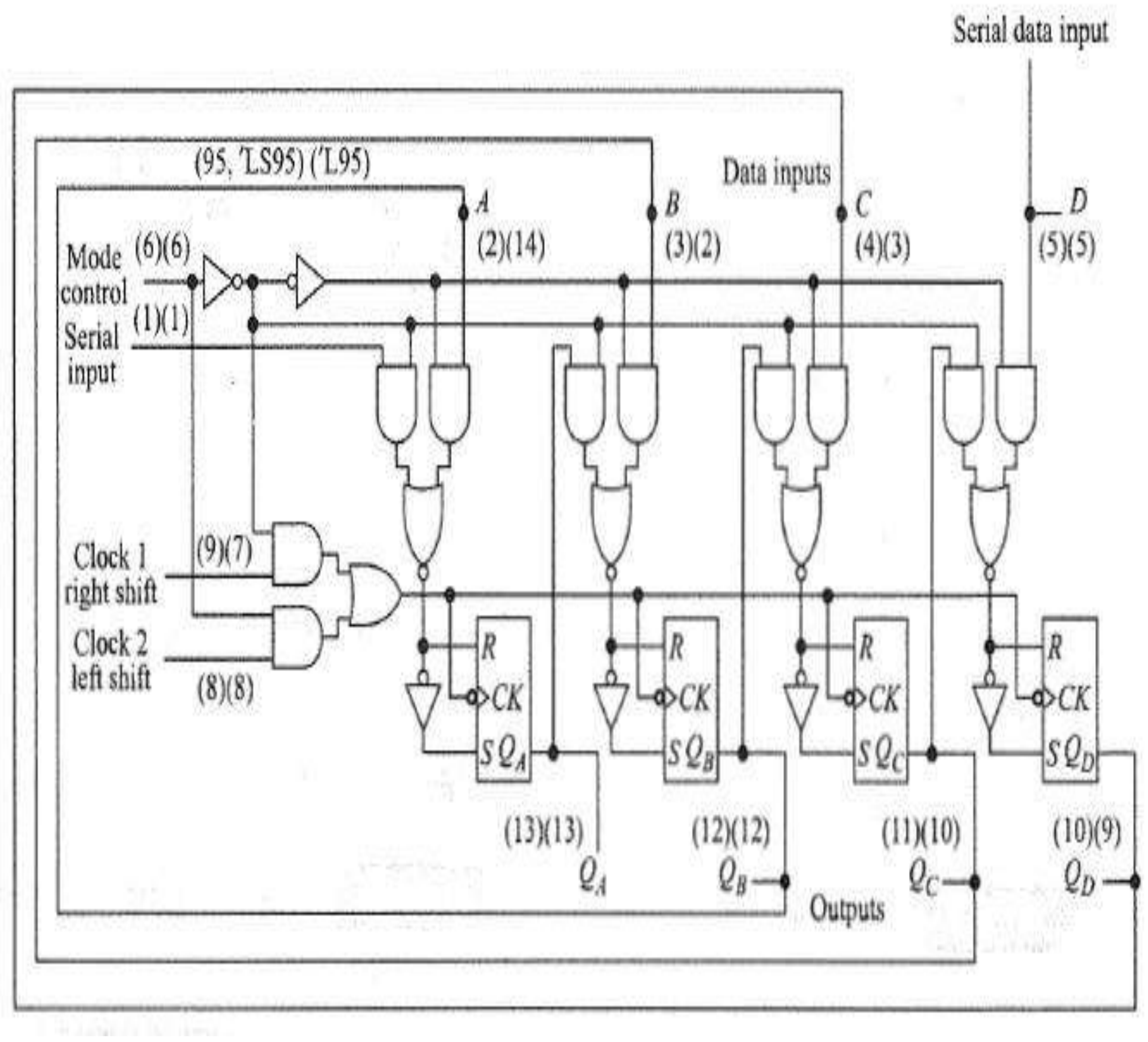
When the mode control line is held high, the AND gate on the right input to each NOR gate is enabled while the left AND gate is disabled.

The data at inputs, A, B, C and D be will then loaded into the register on a negative transition of the clock-this is parallel data input.

When control the mode line is low, the AND gate on the right input to each NOR gate is disabled while the left AND gate is enabled.

The data input to flip-flop QA is now at serial input; the data input to Qs is QA and so on down the line. On each clock NT, a data bit is entered serially into the register at the first flip-flop QA, and each stored data bit is shifted one flip-flop to the right (toward the last flip-flop Qv).

Now, when the mode control line is held high, data bit will be entered into flip-flop QD, and each stored data bit will be shifted one flip-flop to the left on each clock NT.



There are two clock inputs--clock 1 and clock 2. This is to accommodate requirements

where the clock used to shift data to the right is separate from the clock used to shift data to the left.

If such a requirement is unnecessary, simply connect clock 1 and clock 2 together. The clock signal will then pass through the AND-OR gate combination non inverted, and the flip-flops will respond to clock NTs.

Memory

Used to store information (to remember) is an important requirement in a digital system. Circuits and/ or systems designed specifically for data storage are referred to as memory.

In the simplest application, the memory may be **a flip-flop, or perhaps** a number of flip-flops connected to a register. In a larger system, such as a microcomputer, the memory may be composed of semiconductor memory chips.

Semiconductor memories are composed of bipolar transistors or MOS transistors on **an integrated circuit (IC),** and are available

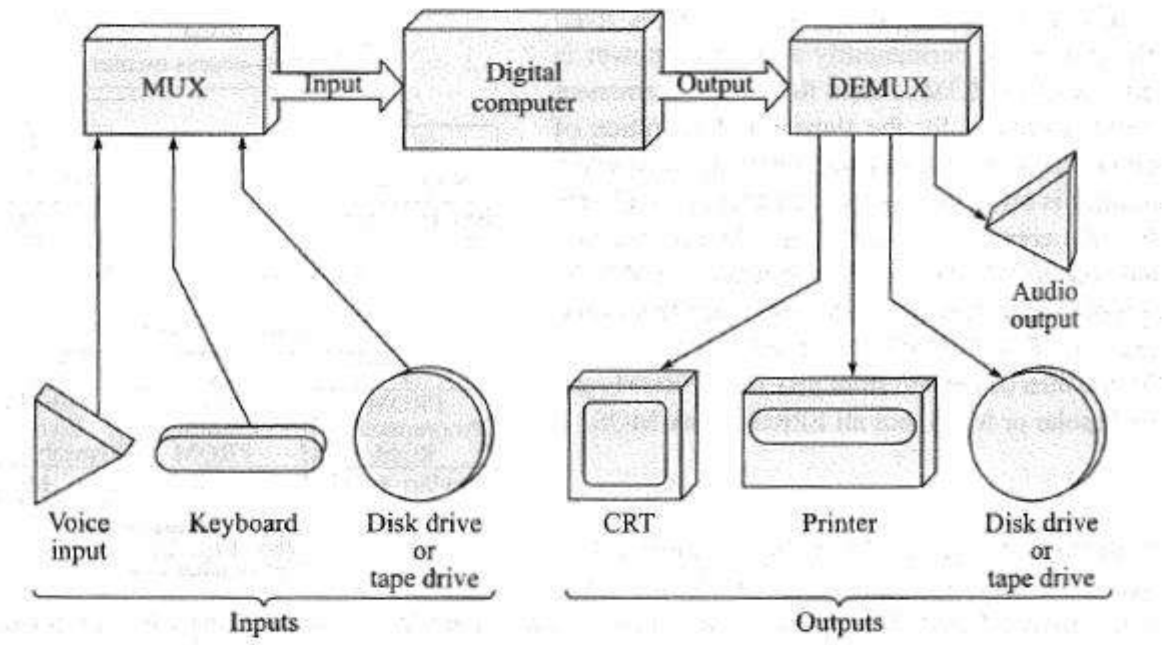
Two general categories

1. **read-only memory (ROM**)
2. **random-access memory (RAM).**

ROM and RAM memories can be constructed to store impressive amounts of data entirely within a computer system.

Large amounts of data (such as banking or insurance records) are generally stored using **magnetic memory** techniques.

Magnetic memory includes the recording of digital information on magnetic tape, hard
Disks , and floppy storage disks.



BASIC TERMS AND IDEAS

Semiconductor Memory

Recent advances in semiconductor technology have provided a number of reliable and economical **MSI and LSI** memory circuits.

The typical semiconductor memory consists of a rectangular array of memory cells, fabricated on a silicon wafer, and housed in a convenient package, such as a DIP.

The basic memory cell is typically a transistor flip-flop or a circuit capable of storing charge and is used to store 1 bit of information.

Memories are usually classified as either bipolar, **metal oxide semiconductor (MOS)**, or **complementary metal oxide semiconductor (CMOS)** according to the type of transistor used to construct the individual memory cells.

Characteristics

The two general categories of memory

1. **RAM**
2. **ROM**

It can be further divided as illustrated in Fig. A de power supply is required to energize any semiconductor memory chip. Once de power is applied to a ***static RAM*** *(SRAM),* the SRAM retains stored information indefinitely, without any further action.

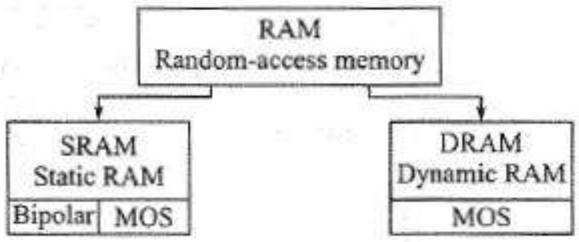
A ***dynamic RAM*** *(DRAM),* on the other hand, does not retain stored data indefinitely; any stored data must be stored again (refreshed) periodically. Both SRAMs and DRAMs are used to construct the memory inside a microcomputer or minicomputer

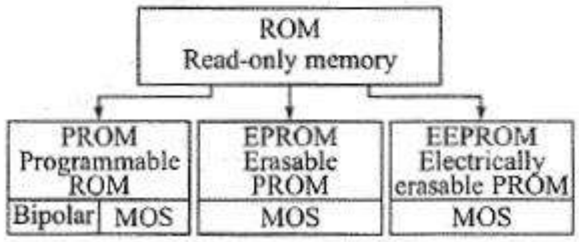
DRAMs are used as the bulk of the memory, and high-speed SRAMs are used for a smaller, rapid-access type of memory known as ***cache memory.***

The cache is used to momentarily store selected data in order to improve computer speed of operation. SRAMs can be either bipolar or MOS, but all DRAMs are MOS.

Data stored in a programmable ROM (PROM) is permanent-a PROM can be programmed only once .However, the data stored in an erasable PROM (EPROM) can be "erased".

.





RAM

An application in which data changes frequently calls for the use of a RAM.

The logic circuitry associated with a RAM will allow a single bit of information to be stored in any of the memory cells-this is the write operation.

There is also logic circuitry that will detect whether a O or a 1 is stored in any particular cell-this is the read operation.

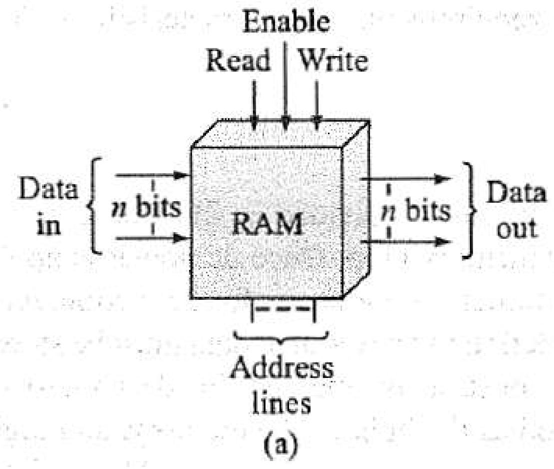
The fact that a bit can be written(stored) in any cell or read (detected) from any cell suggests the description ***random access.***

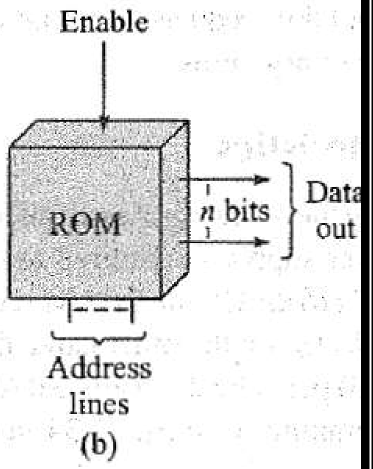
A control signal, usually called *chip-select* or *chip-enable,* is used to enable or disable the chip. In the read mode, data from the selected memory cells is made available at the output.

The address lines determine the cells written into or read from. Since each cell is a transistor circuit, a loss of de power means a loss of data-a RAM that has this type of memory cell is said to provide ***volatile storage.***

ROM

. An application in which the data does not change dictates the use of a ROM. For instance, **a "lookup table**" that stores the values of mathematical constants such as trigonometric functions or a fixed program such as that used to find the square root of a number could be stored in a ROM.





The content of a ROM is fixed during manufacturing, perhaps by metallization or by the presence or absence of a working transistor in a memory cell, by opening or shorting the gate structure, or by the oxide-layer thickness.

Since data is permanently stored in each cell, a loss of power does not cause a loss of data, and thus a ROM provides ***non volatile*** *data storage.*

An application in which the data may change from time to time might call for the use of an EPROM.

Example : State the most likely type of semiconductor memory for each application:

(a) main memory

(b)in a hand calculator;   storing values oflogarithms

(c) storing prices of vegetable produce;

(d) emergency stop procedures for an industrial mill now the design stage.

Solution (a) RAM; (b) ROM; (c) EPROM; (d) PROM.

<u>MAGNETIC MEMORY</u>

Magnetic tape, floppy disks, and hard disks are all capable of **storing large quantities of digital data.**

A hard disk drive and a floppy disk drive are important components in nearly all microcomputer and minicomputer systems.

Large reels of magnetic tape are economical and widely used mass storage components in large computer systems.

The basic principle involved in each case is the magnetization of small spots in a thin film of magnetic material.

1) Magnetic Recording

Magnetic tape is produced by the deposition of a thin film of magnetic material on a long strip of plastic, which is then wound on a reel.

Magnetic material deposited on a rigid disk forms the basis of a ***hard disk***.

A current *i* in the coil shown in Fig. 1.1 a or, the next page will produce a magnetic field across the gap. A portion of this field will extend into the magnetic material below the gap, and the material will be magnetized with a fixed orientation.

When the current is removed, a magnetized spot remains, as shown in Fig.1.2 b. Thus, information has been stored. If the current is reversed in direction, a spot will again be magnetized, but with the opposite fixed orientation, as shown in Fig1.3. c.

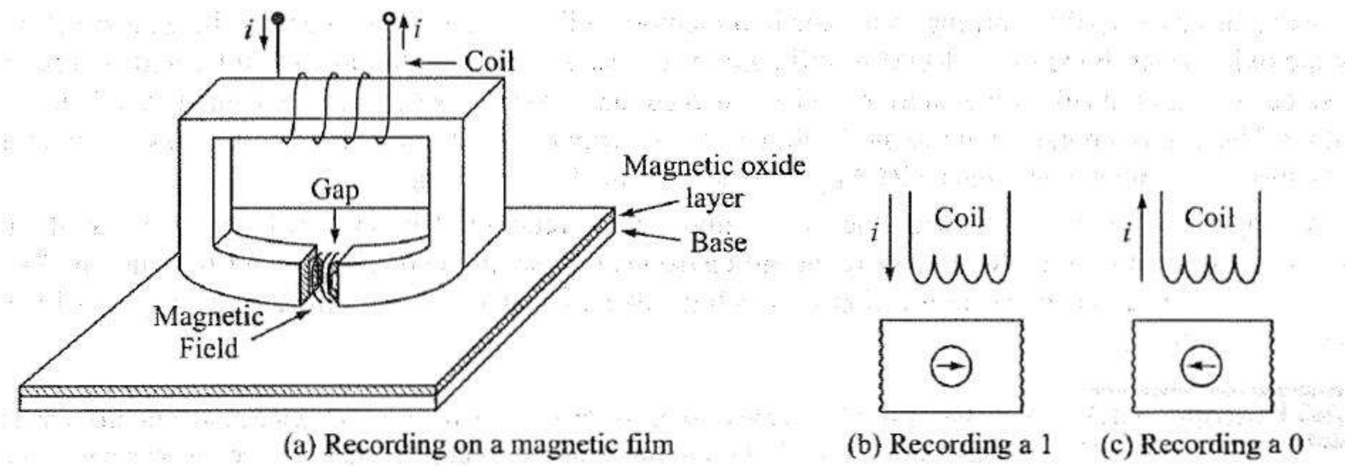
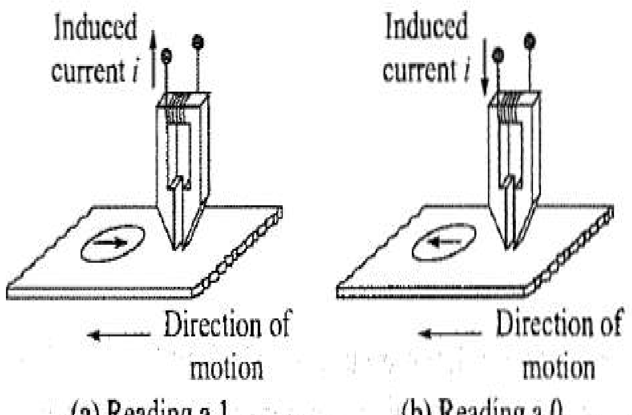


Fig 1.1 a ,b,c



(a) Reading a 1 (b) Reading a 0

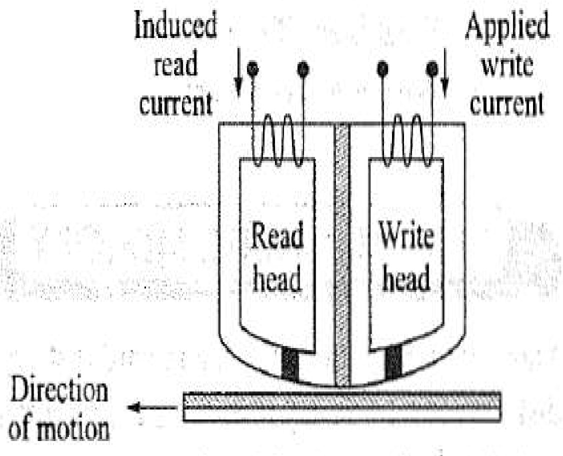


Fig 2.1 a,b, Fig 3.1

Now if a fixed, magnetized spot with a given orientation Fig. 2 .1a on the next page, a current with the direction shown be will is moved past a gap as shown in induced in the coil

Detecting the orientation of the magnetized spot by measuring the induced current is *reading* information (1 or 0).

At time t_1, a spot is recorded under the write head. A short time later, at time t_2, this spot passes under the read head. It can then be read out and a check can be made to ensure that the correct data was in fact recorded.

2. Magnetic Tape

Either seven or nine dual read-write heads are connected in parallel for use with magnetic tape as illustrated in Fig. 4.1 a. As the tape moves past the heads, data is read or written, 7 (or 9) bits at a time. In the 7-bit system, alphanumeric info1mation is recorded in coded form, and there is 1 parity(bit even or odd).

Data can be stored in coded form or in straight binary form.

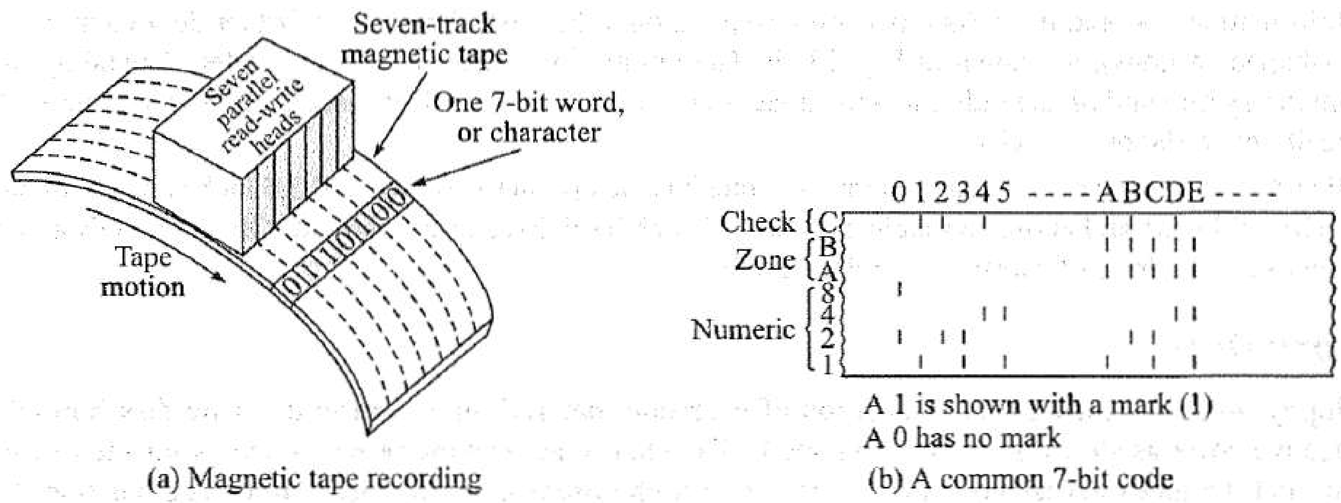

Seven-track magnetic tape
One 7-bit word, or character
Seven parallel read-write heads
0110110
Tape motion
(a) Magnetic tape recording
0 1 2 3 4 5 - - - - A B C D E - - - -
Check C
Zone B A
Numeric 8 4 2 1
A 1 is shown with a mark (|)
A 0 has no mark
(b) A common 7-bit code

Fig4.1-a,b

Data storage on a magnetic tape is ***sequential.*** That is, data is stored one word after another, in sequence. To recover (read) data from the tape requires sequential searching.

Clearly, the storage (or recovery) of data in a sequential system such as this requires considerably more time than storage (or recovery) using RAM. Tape is said to have a longer ***access time*** than RAM.

3. Hard Disks

Magnetic material deposited on a rigid disk (usually aluminum) is the basis for a hard disk system.

One or more of these disks are mounted in an enclosure similar to that shown in Fig. 5.1a. The hard disks used in small computer systems are typically 3.5 in. or 5.25 in. in diameter.

Hard disk drives with 40 to 400 gigabyte capacities are common in microcomputer systems. The disk is rotated at speeds between 3600 to 7200 rpm and in high end servers up to 15000 rpm resulting in typical access time of 16 ms to 3.6 ms.

Because of the relatively short access times and the high storage density, hard disks are widely used in all computer systems.

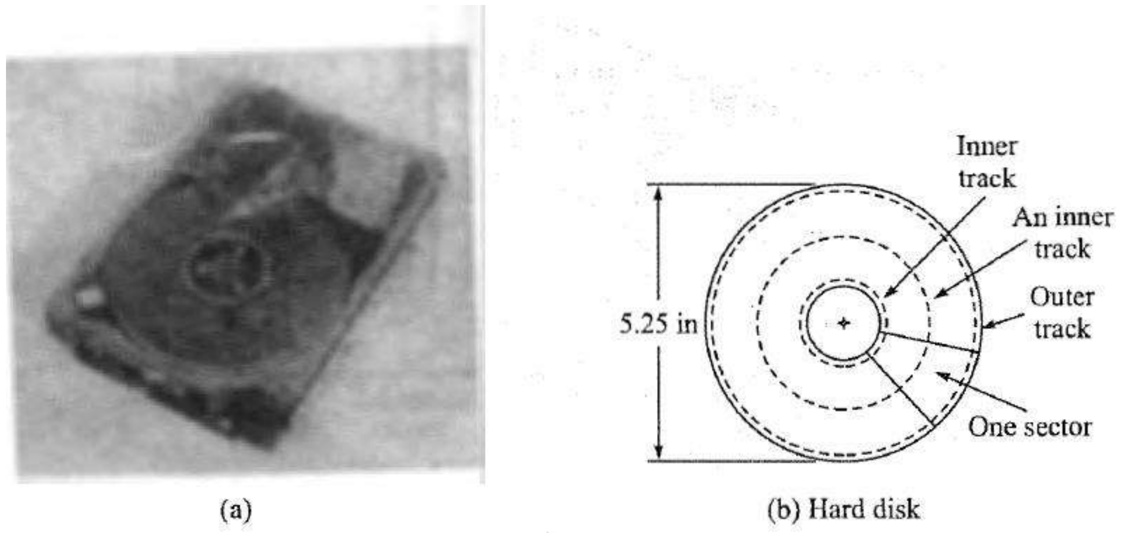


Fig 5.1

Hard Disk System

Information is stored in tracks (concentric rings) around the disk. The disk is further divided into sectors (pie-shaped sections

4.Floppy Disks

A floppy disk is formed by the deposition of magnetic material on a
semirigid plastic disk housed in a protective cover as shown in Figs. 5.2a and b.

The read-write opening provides access for the read-write head, and the index access hole allows the use of a photosensor to establish a reference position.

When the write-protect notch is covered,be data cannot recorded on the disk, preventing accidental loss of data.

Double-sided high-density 5.25-in disks have a capacity of 1.2 MB. Double-sided high-density 3.5-in disks have a capacity of2.88 MB.

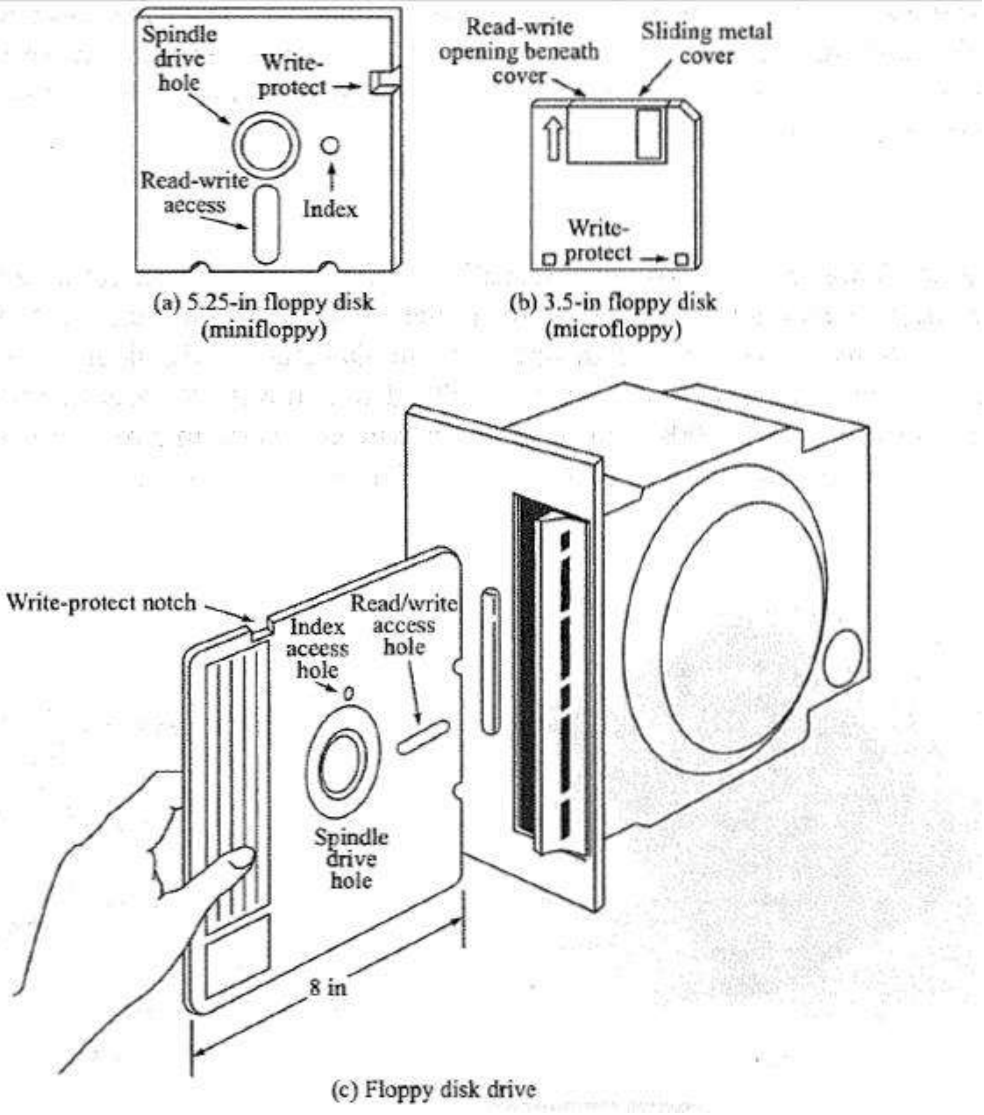


Fig 5.2 a , b & c

The floppy disk is portable, and it must be inserted into a disk drive as shown in Fig. 5.2.c. The drive unit consists of a single read-write head, read-write and control electronics, a drive mechanism, and a track-positioning mechanism.

The spindle drive rotates the magnetic disk at a speed of 360 rpm. Access time is thus somewhat higher than the hard disk, being about 80 ms on average.

OPTICAL MEMORY

Introduced in 1982 jointly by **Philips** and **Sony** for storing digital audio data, **Compact Disk (CD)** found its way into computer storage in 1985.

There was no looking back since then and today we find different types of CDs flooding the market binary where data is optically .coded.

The memory capacity of a CD is in the range of 650-700 MB, i.e. nearly 500 times more than 1.44 MB magnetic floppy disk. Both come in **movable** datastorage category with almost same price tag optical in but data integrity disk is maintained over much longer period of time.

Its newer variety called **Digital Versatile Disk** (DVD) can store data from 4.7 GB to 17.1 GB depending on configuration and make.

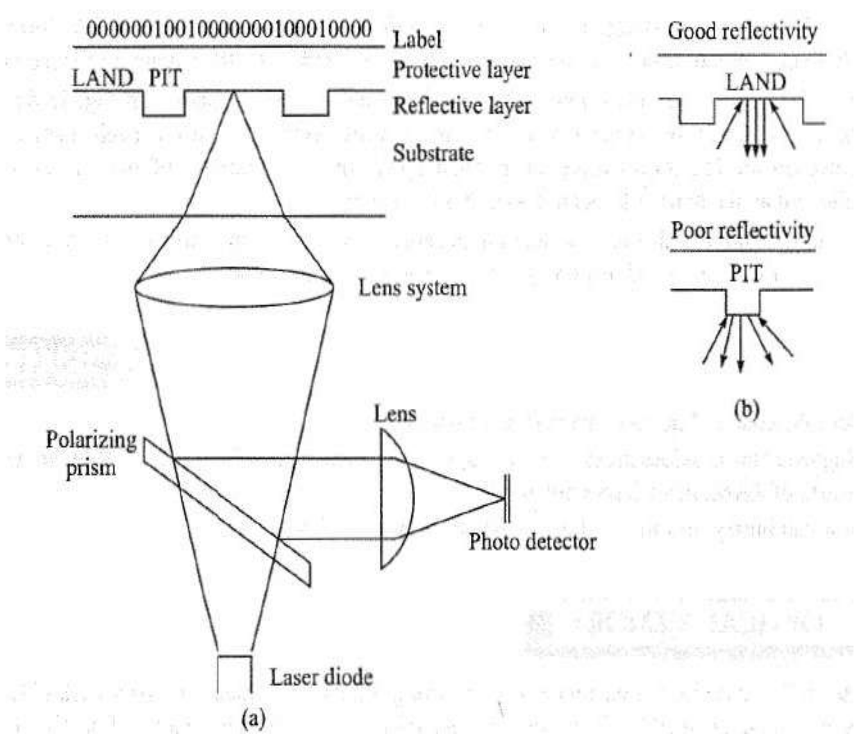
1.CD. ROM

CD ROM or CD Read Only Memory devices are mass produced in factory using a stamp press technology.

CD ROM drives uses LASER **(Light Amplification by Stimulated Emission of Radiation)** technology to read. data from A it. Semi conductor LASER generates a high intensity light wave of stable wavelength"' 780 run.

A lens system is used to direct the LASER towards the disk over approximately 1 micron diameter spot.

The pit width is such that there is at least and at most 10 zeroes between every This is achieved by converting every 8-bit byte into a 14-bit value, a process called **Eight to Fourteen Modulation**



A compact disk reading system

(a) label layer

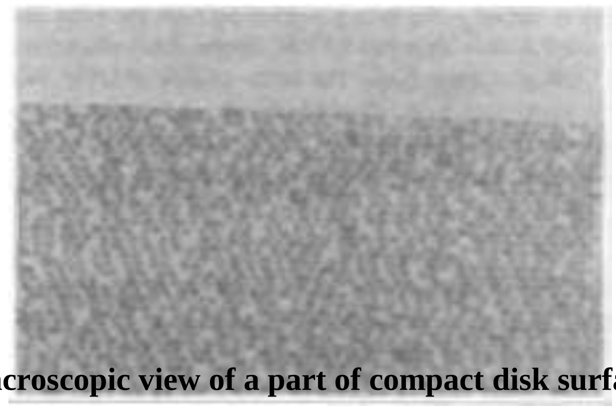
(b) protective layer

(c) reflective layer

(d) a transparent substrate layer on which land

2. CD-R

CD-R or **CD-Recordable** allows user to write data but once. The CD-R drive has laser unit which uses higher intensity light wave for write operations than read.



A macroscopic view of a part of compact disk surface

This melts or chemically decomposes the dye to form a depression mark in the recording layer in appropriate places. The places burnt have lower reflectivity of light.

Thus read laser gets two different intensities on reflected while light reading the disk, similar to read operation of a CD ROM.

Disadvantage :The writing speed of CD-R is much slower than the read speed.

3.CD-RW

CD-RW or **CD Read Write,** previously known as CD-Erasable gives user facility to write and erase data many times, unlike CD-R.

Writing data on disk uses highest power of laser that heats up selected spots to 500°-700°C . The read process is like CD-ROM and CD-R that notes the difference in reflectivity of the reflecting surface.

The main drawback of CD-RW is very low reflectivity of the material and the difference between two levels is also not much.

.

4.DVD

Digital Versatile Disk or Digital Video Disk, popular as DVD resemble compact disk in dimension and look but contains much higher storage space.

DVD driver uses smaller wavelength (635 nm or 650 nm) and lower numerical aperture of lens system to read smaller dimension land and pits.

Each side can have two layers from which data is read and in certain disks data is written on both the sides.

Single sided, single layer has capacity of 4. 7 GB, single sided double layer has 8.5 GB, double sided single layer has 9 .4 GB while double sided double layer has 17 .1 · GB of storage space.

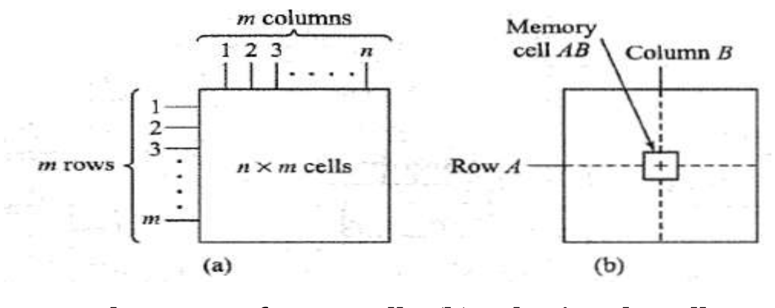
The better quality of DVD output compared to CD comes from better channel coding correction error scheme and of course a higher data transfer rate.

MEMORY ADDRESSING

- **Cell Selection**

Addressing is the process of selecting one of the cells in a memory to be written into or to be read from, In order to facilitate selection, memories are generally arranged by placing cells in a rectangular arrangement of rows and columns as shown in Fig. a.

In this particular case, there are m rows and n ·columns, for a total of n X m cells in the memory.



A rectangular array of *m* x *n* cells, (b) Selecting the cell at memory address *AB*

The control circuitry that accompanies the basic memory array is designed such that if one and only one row line is activated and one and only one column line is activated, the memory cell at the intersection of these two lines is selected.

For instance, in Fig. 7.1 b, if row *A* is activated and column B is activated, the cell at the intersection of this row and column is selected--that is, it can be read from or written into.

For convenience, this cell is then called *AB,* corresponding to the row and the column selected. This designation is defined as the *address* of the cell. The activation of a line (row or column) is achieved by placing a logic 1 (or perhaps a logic 0) on it.

2.Matrix Addressing

Let's take a little time to consider the various possible configurations for a rectangular array of memory cells.

The different rectangular arrays of 16 cells are shown in Fig. 8.1 In each of the five cases given, there are exactly 16 cells.

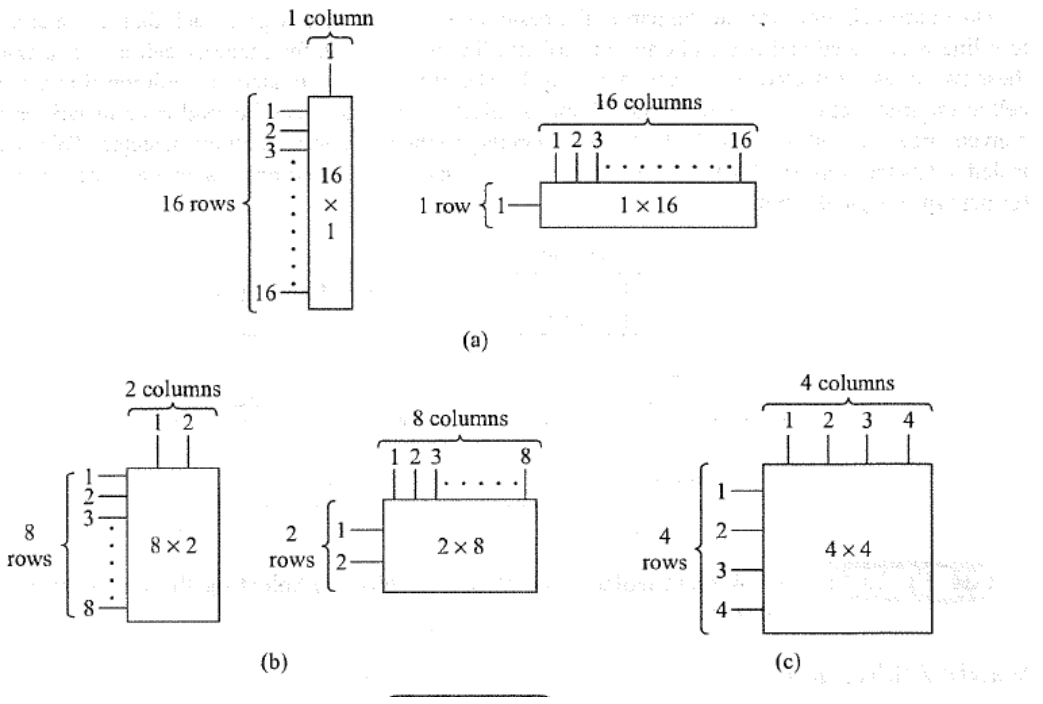


Fig 8.1 a,b & c

For any of the three configurations, the selection of a single cell still requires a single row and a single column to define a unique address.

The minimum requirement in either case is really only 16 lines. However, either arrangement in Fig. 8.2b requires only 10 address lines-8 rows and 2 columns, or 2 rows and 8 columns.

In general, the arrangement that requires the fewest address lines is a square array of n rows and n columns for a total memory capacity of n x n = n2 cells. It is exactly for this reason that t he square configuration is so widely used in industry.

This arrangement of n rows and n columns is frequently referred to as matrix ad-dressing. In contrast, a single column that has n rows (such as the 16 x 1 array of cells) is frequently called linear addressing, since selection of a cell simply means selection of the corresponding row, and the column is always used.

The small triangle (V) at the *Q* output means that the output is three-state (tri-state). We'll
use this chip in Sec. 13.5.

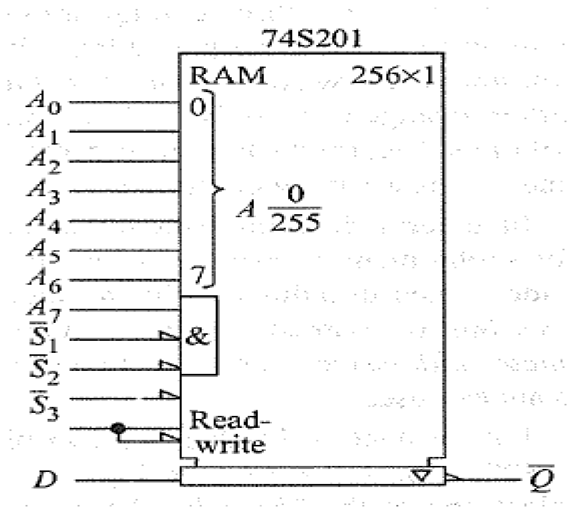
3.Address Decoding

Take another look at the 4 x 4 memory in Fig. 8.1c. To select a single cell, we one must activate and only one row, and one and only one column.

This suggests the use of two l of 4 binary to decimal decoders as shown in Fig. 8.1. Consider the selection of the cell at address 43 (row 4 and column 3). If A_4 = 1 and A_3 = 1, the decoder will hold the row 4 line high while all other row lines will be low. Similarly, if A_2 = 1 and A_1 = 0, the decoder will hold column 3 high and all other column lines low.

Thus an input $A_4A_3A_2A_1$ = 1110 will sekct cell 43. We can consider A_4A_3 as a row address of 2 bits and A_2A_1 as a column address of 2 bits. Taken together, any cell in the array can be uniquely specified by the 4-bit address $A_4A_3A_2A_1$• As another example, the address $A_4A_3A_2A_1$ =the0110 selectcell at row 2 and column 3 (address 23).

The address decoders shown in Fig. 13.14 further reduce the number of address lines needed to uniquely locate a memory cell, and they are almost always included on the memory chip.



4.Expandable Memory

So far, we have only discussed memories that provide access to a single cell or bit at a time. It is often advantageous to access groups of bits-particularly groups of 4 bits (a nibble) and groups of 8 bits (a byte).

It is not difficult to extend our discussion here to accommodate such requirements. There are at least two popular methods. The first simply accesses groups of cells on the same memory chip, and we discuss this idea next.

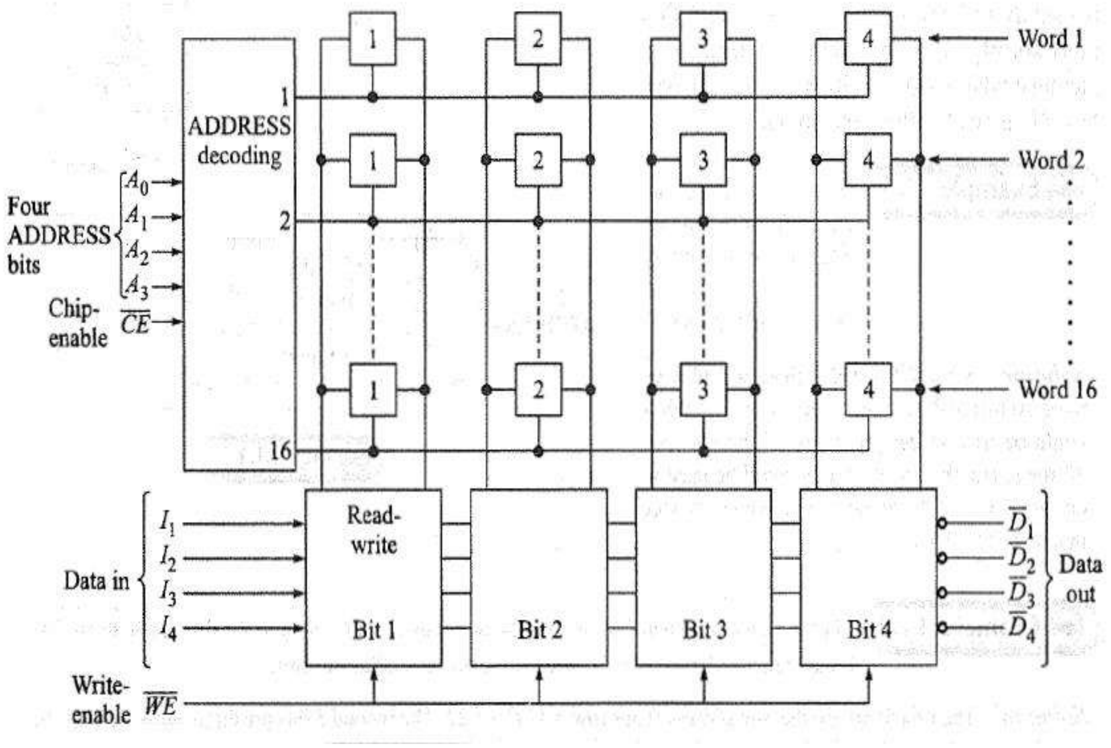
The second connects memory chips in parallel, and we consider this technique in a following section.

The logic diagram for a 64-bit (16 x 4) bipolar memory is given in Fig. 13.16. There are 16 rows of cells with four cells in each row; thus the description (16 x 4). Each cell is a bipolar junction transistor flip-flop.

The address decoder has 4 address bits and thus 16 select lines--0ne for each row. In this case, each select line is connected to all four of the cells in a row.

So, each select line will now select four cells at a tin1e. Therefore, each select line will select a 4-bit word (a nibble), rather than a single cell.

You might think of this arrangement as a "stack" ofsixteen4-bit registers. This is really a form of l in ear addressing, since the 4 address bits, when decoded, select one of the sixteen 4-bit registers



64-bit (16 x4) **memory**

ROMs, PROMs, AND EPROMs

Let's turn our attention to the operation of a ROM. The tem1 ROM is generally reserved for memory chips that are programmed by the manufacturer.

Such a chip is said to be ***mask-programmable***, in contrast to a PROM, which is said to *be. field-programmable- that* is, it can be programmed by the user.

EPROMs can be programmed, erased, and programmed again;

Programming

What exactly does programming a ROM, PROM or EPROM involve? It simply involves writing, or storing, a desired pattern of Os and ls (data).

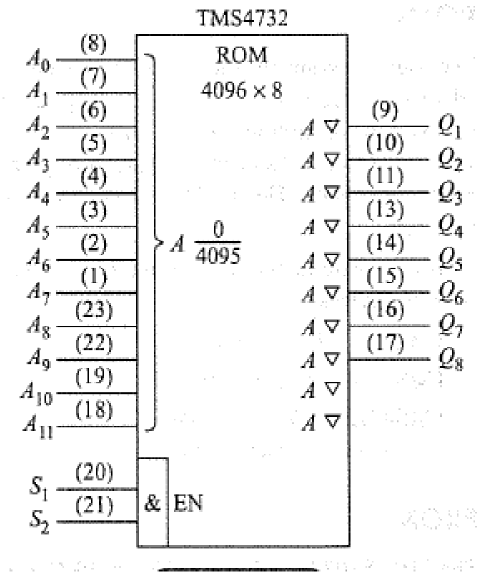
Each cell in the memory chip can store either a 1 or a 0. As supplied from the manufacturer, most chips have a O stored in each cell. The chip is then programmed by entering ls in the appropriate cells. For instance, the content of every 4-bit word in a 64 x 4 chip is initially 0000.

If the desired content of a word is to be 0110, then the two inner bit positions will be altered to In the case of a ROM, you must supply the manufacturer with the exact memory contents to be stored in the ROM.

The Texas Instruments TMS4732 is a ROM having 4096 eight-bit words (a 4096 x 8 ROM). The logic diagram is given in Fig. 13.17. The 8-bit word length makes this NMOS (n-channel MOS) chip ideal for microprocessor applications.

Texas Instruments will store user-specified data during manufacturing. The user must supply data storage requirements in accordance with detailed instructions given on the TMS4732 data sheet

The Texas Instruments TBP l 8S030 is a bipolar memory chip arranged as thirty-two 8-bit words (256 bits). The logic diagram for this user-programmable PROM chip is given in Fig. 13 .18.



Basically, the programming is done by applying a current pulse to each output terminal where a logic 1 must appear (be stored).

The current pulse will destroy an *existing fuse link.* When the fuse link is present, the transistor circuit in that cell stores a 0.

After the fuse link is destroyed, the circuit storesprogramming1.

1. Apply the address of the word to be programme (A_0, *Al'* A_2, A_3 , A_4). For instance, to programme the word at address 14H (hex 14). apply

AOAIA2A3A4 = 10100

2. To store the word Q_0 $Q_1 Q_2$ Q_3 Q_4 Q_5 Q_6 Q_1 00101000:

(A)Ground output Q_2 and connect output sall other to +5 Vdc through a 3.9-k.Q resistor Raise the +5-Vdc supply to +9.25 Vdc and momentarily enable the chip. This will program a 1 in bit position Q_2•

(B) Repeat for bit position Q_4 • This will program a 1 in bit position Q_4 •

3. Repeat steps 2 and 3 for each word to be programmed.

ROMs

The logic diagram for the Texas Instruments TMS4732, a 4096 x 8 ROM, is given i Fig. a. Twelve address bits are required, A_0 , *A1, ... , All* (2^{12} + 4096).

There are two chip-enable inputs, S_1 and S_2• Both S_1 and S_2 must be high in order t enable the chip.

Each of the eight data output lines is a three-state line (the small V symbol). As mentioned previously, this chip is ideal for microprocessor applications because of the 8-bit word length. This ROM is mask-programmable, and data must be specified for the manufacturer before purchase.

Texas Instruments offers a number of other ROMs with larger memory capacity, all of which are LSI NMOS devices.

TMS4664: 8192 x 8-bit

TMS4764: 8192 x 8 bit

TMS47128: 16,384 x 8-bit

TMS47256: 32,768 x 8-bit

PROMs

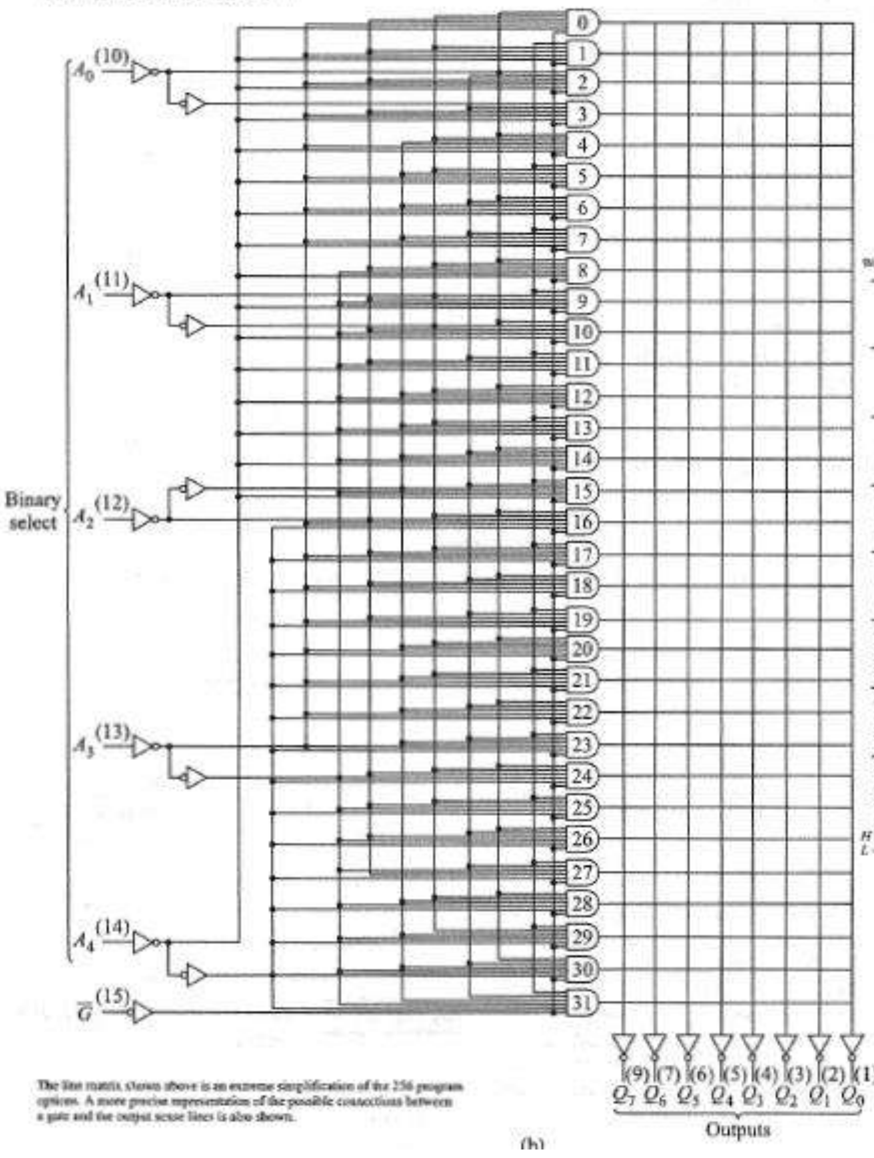
The TBPl 8S030 is a 256-bit (32 x 8) PROM arranged as a stack of thirty-two 8-bit words. The 74S288 is an equivalent designation.

As shown in Fig., the 5 row address bits are labeled A_0 , *AI'* A_2, A_3, A_4 and the 8 output bits in a word are labeled Q_0 , QI' Q_2, Q_3, Q_4, Q_5, Q_6, *Qr*

Input *G* is used to enable or disable the entire set of 32 input decoding gates.

When *G* is high, all the address decoding gates are inhibited and the memory chip is disabled, causing the eight output data bit lines to be high.

Functional block diagram and word selection



Word select table

Word	Inputs				
	A_4	A_3	A_2	A_1	A_0
0	*L*	*L*	*L*	*L*	*L*
1	*L*	*L*	*L*	*L*	*H*
2	*L*	*L*	*L*	*H*	*L*
3	*L*	*L*	*L*	*H*	*H*
4	*L*	*L*	*H*	*L*	*L*
5	*L*	*L*	*H*	*L*	*H*
6	*L*	*L*	*H*	*H*	*L*
7	*L*	*L*	*H*	*H*	*H*
8	*L*	*H*	*L*	*L*	*L*
9	*L*	*H*	*L*	*L*	*H*
10	*L*	*H*	*L*	*H*	*L*
11	*L*	*H*	*L*	*H*	*H*
12	*L*	*H*	*H*	*L*	*L*
13	*L*	*H*	*H*	*L*	*H*
14	*L*	*H*	*H*	*H*	*L*
15	*L*	*H*	*H*	*H*	*H*
16	*H*	*L*	*L*	*L*	*L*
17	*H*	*L*	*L*	*L*	*H*
18	*H*	*L*	*L*	*H*	*L*
19	*H*	*L*	*L*	*H*	*H*
20	*H*	*L*	*H*	*L*	*L*
21	*H*	*L*	*H*	*L*	*H*
22	*H*	*L*	*H*	*H*	*L*
23	*H*	*L*	*H*	*H*	*H*
24	*H*	*H*	*L*	*L*	*L*
25	*H*	*H*	*L*	*L*	*H*
26	*H*	*H*	*L*	*H*	*L*
27	*H*	*H*	*L*	*H*	*H*
28	*H*	*H*	*H*	*L*	*L*
29	*H*	*H*	*H*	*L*	*H*
30	*H*	*H*	*H*	*H*	*L*
31	*H*	*H*	*H*	*H*	*H*

H = high level
L = low level

The line matrix shown above is an extreme simplification of the 256 program options. A more precise representation of the possible connections between a gate and the output sense lines is also shown.

Functional block diagram and word selection

EPROMs

One disadvantage of a PROM is that once it is programmed, the contents are stored in that memory chip permanently-it can't be changed; a mistake in programming the chip can't be corrected. The EPROM overcomes this difficulty.

The EPROM has a structure and addressing scheme similar to those of the previously discussed PROM, but it is constructed using MOS devices rather than bipolar devices.

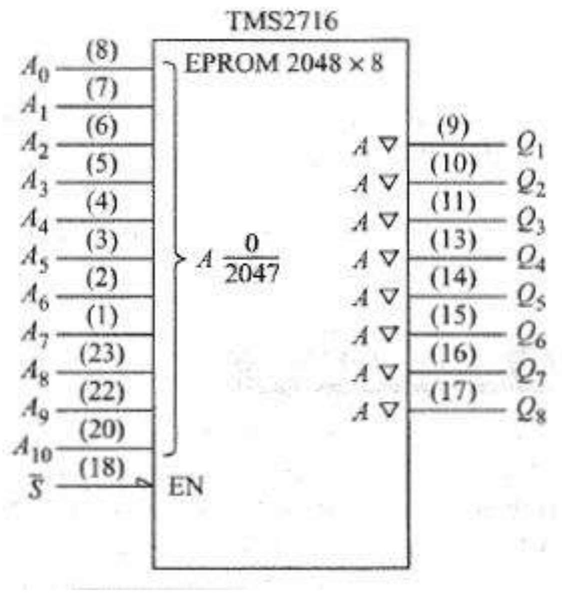
Many MOS EPROMs are TTL-compatible, and even the technique used to program the chip is similar to that used with a bipolar memory.

The current pulse used to store a 1 when programming a bipolar PROM is used to destroy ("bum out") a connection on the chip. The same technique is used to program an MOS-type EPROM, but the current pulse is now applied for a period of time (usually a few milliseconds) in order to store a fixed charge in that particular memory cell.

This stored charge will cause the cell to store a logic 1.The interesting thing about this phenomenon is that the charge can be removed (or erased), and the cell will now contain a logic O! Furthermore, the process can be repeated.

The memory cells are "erased" by shining an ultraviolet light through a quartz window onto the top of the chip. The light bleeds off the charge and all cells will now contain Os.

The requirements for program runing and erasing an EPROM vary widely from chip to chip, and data sheet information must be consulted in each individual case.



EEPROM, Flash Memory

Electrically Erasable Programmable Read Only Memory (EEPROM) is similar to EPROM as far as writing into memory is considered, i.e. effecting a current pulse to store charge.

The erasing, however, is different and is done by removing the charge and sending a pulse of opposite polarity.

There are two types of EEPROM-parallel and serial. Parallel EEPROM is faster, costlier and comes in 28xx family.

Their pin out and functioning is similar to 27xx EPROM family. Serial EEPROM is slower, cheaper, uses lower number of pins and comes in 24xx family.

*Flash memo, y*on is a further advancement EEPROM. This, too, writes and erases data electrically-can be both parallel and serial type. The number of write/erase cycle is finite and often, there is a separate management scheme to take note of this.

There is an internal voltage generation block that takes single voltage supply and generates different voltages required for writing and erasing. Different manufacturers have created different standards for flash memory chip which differ in pinout, memory organization, etc. Intel family chips are 28Fxxx while AMD chips are numbered as 29Fxxx.

RAMs

The basic difference between a **RAM and a ROM** is that data can be written into (stored in) a RAM at any address as often as desired.

Naturally data can be read from any address in either a RAM or a ROM, and the addressing and for read cycles both devices are similar. The characteristics of both bipolar and MOS "static" RAMs are discussed in this section.

A static RAM (SRAM) uses a flip-flop as the basic memory cell (either bipolar or MOS) and once a bit is stored in a flip-flop, it will remain there as long as power is available to the chip-essentially forever-thus the term "static."

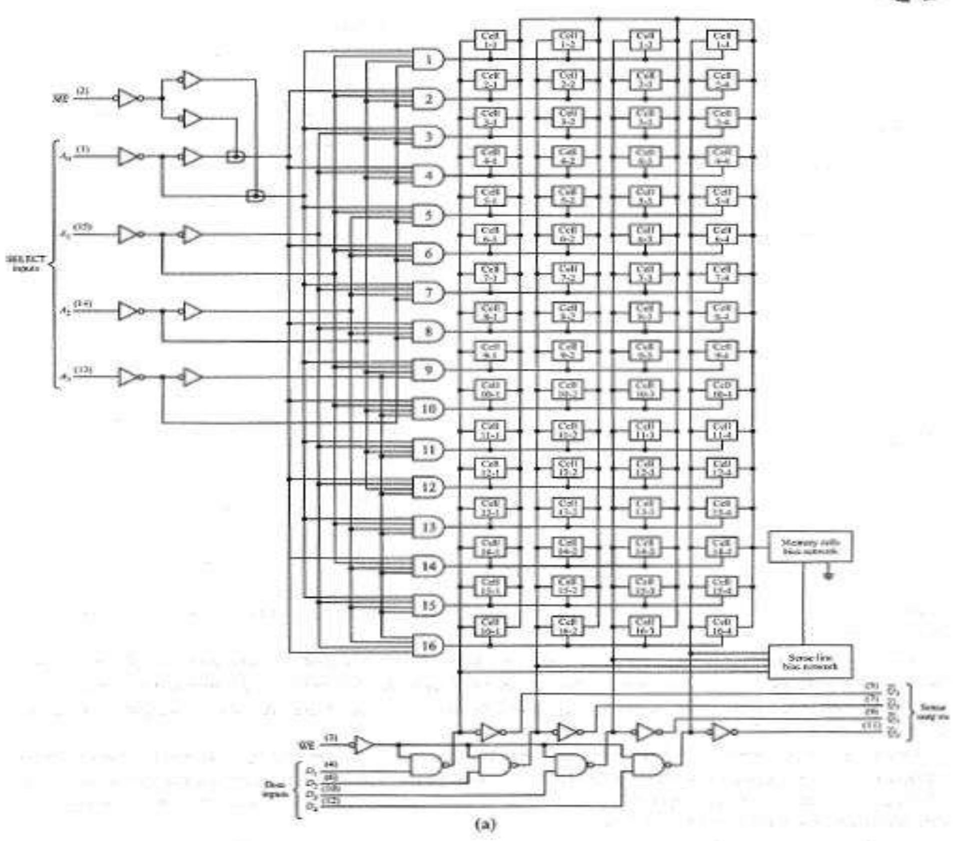
On the other hand, the basic memory cell in a "dynamic" RAM (DRAM) utilizes stored charge in conjunction with an MOS device to store a bit of information. Since this stored charge will not Remain periods for long of time

The 7489

The 7489 shown in Fig. 13.22 is a TTLRAM,LSI 64-bit arranged as 16 words of 4 bits each.

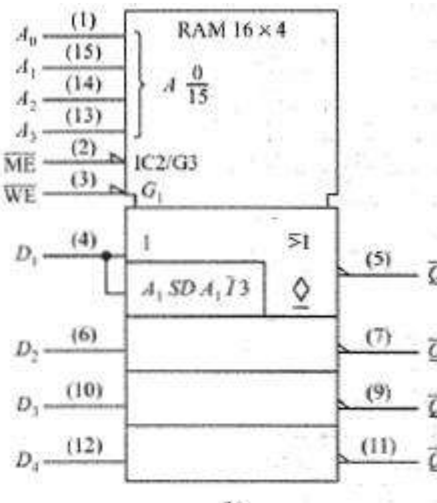
Holding the memory-enable (ME) input low will enable the chip for either a read or a write operation, and the four data address lines will select which one of the sixteen 4-bit word positions to read from or write into.

Then, if the write-enable (WE) is held low, the 4 bits present at the data inputs (D_1, D_2, D_3, D_4) will be stored in the selected address.


(a)

The read operation is no different from that for a ROM. For this chip, simply hold ME low and WE high, and select the desired address. The 4-bit data word then appears at the "sense" outputs.

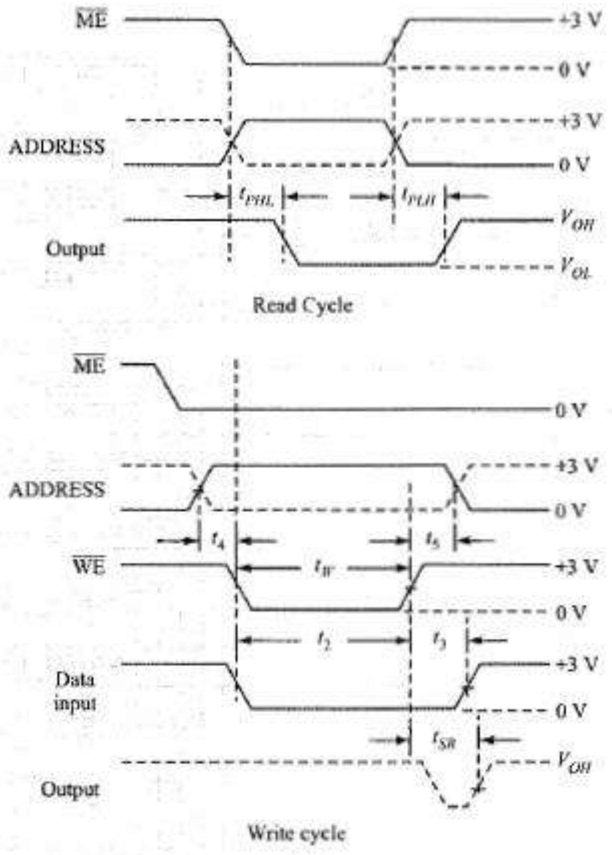
The timing for a read operation is shown by the wave forms in Fig. 13.22d. The propagation delay time t PHL is that period of time from the fall of ME until stable data appears at the outputs-the data sheet gives a maximum value of 50 ns, with 33 ns typical. Naturally the address input lines must be stable during the entire read operation, beginning·



(b)

$\overline{ME}$	$\overline{WE}$	Operation	Condition of outputs
L	*L*	Write	Complement of data inputs
L	*H*	Read	Complement of selected word
H	*L*	Inhibit storage	Complement of data inputs
H	*H*	Do nothing	High

(c)



(d)

with the fall of ME. Notice carefully that the data appearing at the four outputs will be the *complement* of the stored data word!

You will notice from the truth table that when the chip is *deselected*, that is, when ME is high, the outputs all go to a high level, provided we are in a read mode (WE is high). So, in the read the operation waveforms, time PLH is the delay time from the rise of ME until the outputs assume the high state.

The data sheet gives and 50 ns maximum 26 ns typical for this delay time.

During a write operation the 4 bits present at the data inputs will be stored in the selected memory address by holding the ME input low (selecting the chip) and holding the WE low.

Let's look carefully at the timing requirements for the write cycle. First, the WE must be held low for a minimum period of time in order to store information in the memory cells-this is given as time *tw* on the waveforms, and the data sheet calls for 40 ns minimum.

Memory-enable selects .the chip when low, and is allowed to go low coincident with or before a write operation is called for by WE going low.

Next, the data to be written into memory must be stable at the data inputs for a minimum period of time before WE and, also for a minimum period of time after WE.

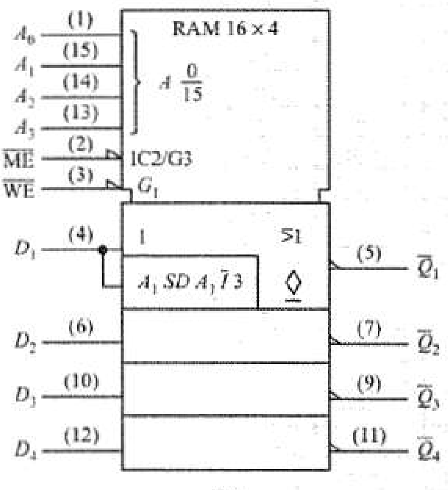
The time period prior to WE is called the *data-setup time* t_2• This time is measured from the end of the write-enable signal back to the point where the

In other words, the address lines are allowed to become stable coincident with or before

WE goes low. The address lines must also be stable for a period of time after the rise of WE; this is called the ***address-hold*** **or** ***select-hold*** time t_5, and the data sheet calls for 5 ns minimum.

Finally, after a write operation, if the chip is deselected (ME goes high), the outputs will return to a high state. The maximum time for this to occur is the sense-recovery time *tsR'* given as 70 ns maximum on the data sheet.

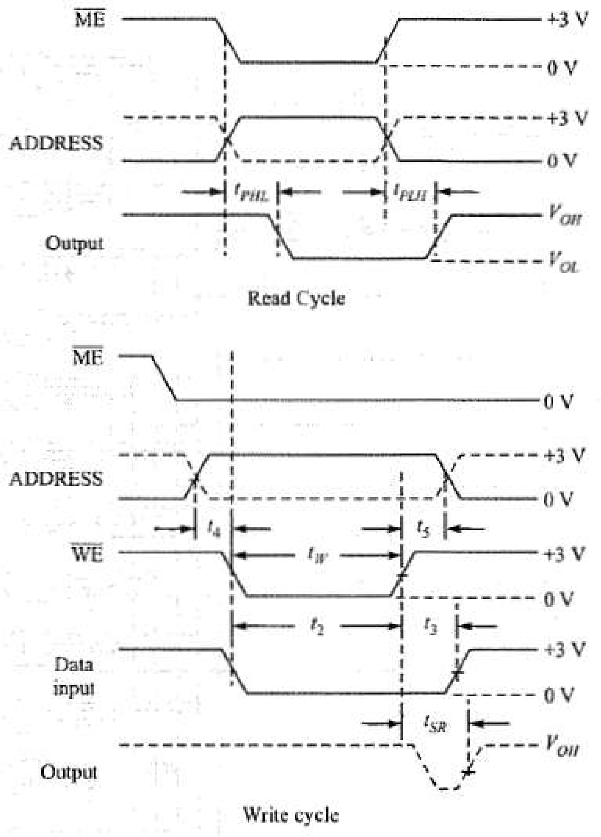
The operation of a 7489 ns straightforward and easy to understand; therefore it is a good chip to study in elementary discussions of RAMs. It can be used to construct memories having larger capacities by connecting chips in parallel, but it's not too practical when we wish to consider memories of 16K, 32K, ... , 256K, 512K, and so on.



(b)

$\overline{ME}$	$\overline{WE}$	Operation	Condition of outputs
L	*L*	Write	Complement of data inputs
L	*H*	Read	Complement of selected word
H	*L*	Inhibit storage	Complement of data inputs
H	*H*	Do nothing	High

(c)



(d)

A SIMPLE COMPUTER DESIGN

A digital computer is capable of computation and taking decision based on binary coded instructions stored inside it.

The **central processing unit** (CPU), also known as the brain of the computer sequentially fetches these instructions, decodes it and then executes it by performing some action through available hardware.

The technique you learn in developing this simple machine will be useful when you go for a full-fledged computer design in some higher-level courses.

We'll also discuss a simple hardware operation description language, called **Register Transfer Language (RTL)** useful for state machine design.

BUILDING BLOCKS

Depending how we program it, we will be able to solve different arithmetic and logic problems and that is shown towa.rds the end of this chapter through examples.

The purpose of defining a problem is to choose specific hardware components that will serve as building block of our simple computer.

The Problem

Add 10 numbers stored in consecutive locations of memory. Subtract from this total a number, stored in 11 th location of memory. Multiply this result with 2and store it in 12th location of memory. All the numbers brought from memory lie between O and 9.

Memory

The problem says the numbers or data to be fetched from memory and we also know that programs, i.e. binary coded instructions are also stored in memory, let us divide the memory used in our computer in two parts.

One part stores the program or series of instructions the computer executes sequentially and this is known as *program memory*. The other part houses data that program uses and is also used for storing result. This is called *data memo!J'·*

Now we try to decide how many bits of information we store in each address location. Usually, bits in memory locations are stored in multiple of 8 called *byte*.

Let's now see the requirement of program memory. There, in each location, certain number of bits are allocated that defines the instruction to be executed. This is called operation code or in short, *opcode.* The rest of the bits can be used for referring the memory .location from which data is to be brought or stored, if required by the instruction.

7 5	4 0
opcode	Address

Register Array

The computer needs a set of registers to perform its operation. **Memory Address Register** *(MAR)* is a 5-bit register that stores the address of the memory location referred in a particular instruction.

The output of this is fed to a 5-to-32 address decoder. Each output of the decoder points to a location in the memory. All memory referenced instruction loads memory address in *MAR.*

Memory Data Register ***(MDR)*** is an 8-bit register that stores the memory output when a memory read operation is performed. During memory write operation it stores the value that gets written to the memory. Thus it can also be called a memory buffer.

In arithmetic or logic operation when more than one operand is required by ALU, one operand in our simple machine comes from *MDR.*

Program Counter ***(PC)*** is a 5-bit counter that stores the address of the memory location from which next instruction is fetched. At power on, our machine *PC* is reset so that its content is all zero.

Thus location 00000 has to be a part of program memory and this is also the starting address from which program execution begins. Since, in our simple machine all the instructions are single byte instruction, every time an op code is fetched we'll increment *PC* by one, and thus *PC* will point to location of the next op code.

Instruction Register ***(IR)*** is a 3-bit register, which retains the opcode till it is properly executed in one or more clock cycles. Since all memory read and write operations are done through *MDR,* after an instruction is read from memory, 3 MSB that contains the opcode are transferred to *JR.*

Accumulator (ACC) is a multi-purpose register that always stores one operand of an arithmetic or logic operation. The result of this operation, i.e. ALU output is also stored in *ACC.* Functions like shifting of bits to left or right are also carried on *ACC.* Thus, in our simple computer *ACC* is a shift register with parallel load facility.

Timing Counter ***(TC)*** is a synchronous parallel load counter that stores and updates the timing information. The timing counter output is decoded to generate different timing signal, which in tum triggers different events in execution of an instruction.

The counter is reset synchronously with clock once an instruction is fully executed. If an instruction is conceived as a ***macro operation*** then series of sequential steps necessary to carry out the instructionis in the computer called ***micro operations***

Other Important Hardware

Arithmetic Logic Unit (ALU) is a versatile combinatorial circuit that can perform a large range of arithmetic and logic operations. Since the data is 8-bit long, we use an 8-bit ALU.

The control input value decides the function ALU executes at a particular time. ALU can accept to two operands a time, one from *A* CC and the other from *MDR.* The ALU output is stored in *ACC.*

Decoder Instruction (ID) is a 3-to-8 decoder, which takes input from *JR* and thereby decodes the opcode. In our simple computer there are 8 different op codes, each one making one of the decoder output (D_0, *Dp*···, D_7*)* high. This in turn initiates specific micro operations necessary to execute that op code in subsequent clock cycles.

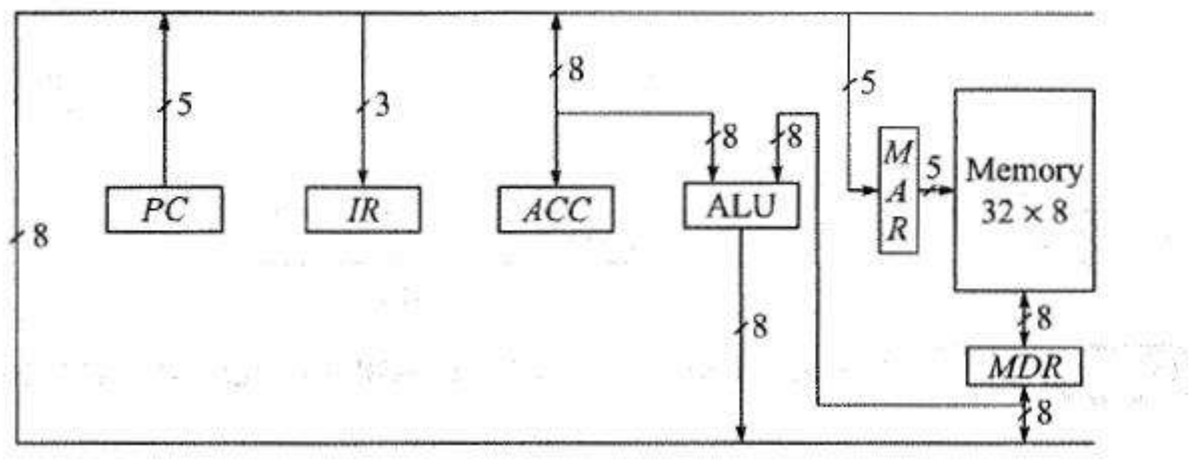
Timing Sequence Decoder (TSD) is again a 3-to-8 decoder that takes input from *TC* and provides necessary timing information in the form of decoded output (~J' T_1, • •• , T_7*)* for a micro operation to be executed.

BUS is a group of wires that serve as a shared common path for data transfer of all the devices connected to it. With this, we do not need a separate device to device connection which increases the number of wiring specially when large number of devices are used in a system. Since, the largest group of binary data that is transferred in our computer is 8-bit, the bus used is an 8-bit bus.

BUS Selector (BS) is a multiplexer, which decides which one of all the connected devices is in transmission mode, i.e. has placed data in the BUS. Note that, if more than one device try to send data simultaneously, there will be a conflict producing erroneous result.

This is called *control path* and we'll design it when we define the instruction set for our computer.

Generally speaking, *address bus* is the group of wires that transfer address information, *data bus* is another group that transfers data and *control bus* transfers control information. Often, address information and data are transferred through a common bus and a control logic decides which is to be transferred and when.



REGISTER TRANSFER LANGUAGE

Before we go for design of control path and the control unit as a whole we have to define macro operations and then we need to break up each macro operation in series of micro operations at register level.

Register Transfer Language (RTL) gives a simple tool through which these micro operations can be expressed and then control unit can be designed from that. The basic structure of this language is

$$X: A \leftarrow B$$

This means, if condition Xis TRUE, i.e. *X* = l then content of register Bis transferred to register *A. X* can be a single logic variable or a logic expression like *xy =x&y, x + y =x* I*y*, etc.

In RTL we distinguish logic operation 'OR' from arithmetic operation 'addition' by assigning symbols ' I 'and '+'respectively. The logic AND is expressed by symbol'&'. However, if the'+' sign appears left to':' in an RTL statement it means logical OR and '.' refers to logical AND. This is so because to left of ':' only logical operators can reside.

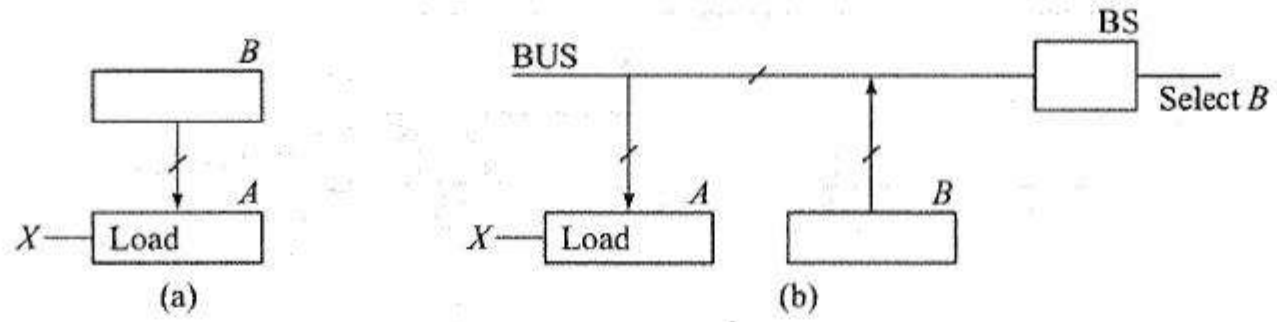
Often AND, OR, NOT are expressed by 'A', 'v', '-' respectively. Also note, this register transfer destroys the previous content of *A* but not that of *B.* Both the register *A* and *B* now have the same value. If register transfer takes via BUS

$$A \leftarrow B = BUS \leftarrow B, \quad A <\text{--} BUS$$

Since, *BUS* is not a register but a group of wire this means *B* getting access to *BUS* through BUS selector (BS) and the whole event takes place in one clock cycle. Figure 16.3 pictorially depicts register transfer withoutand BUS.

To write anything to memory, in our simple computer we have to place the address information in *MAR* and the data to be written in *MDR*. Thus, memory write operation in RTL is expressed as

X: M[MAR] <--MDR



Similarly, memory read operation is also done through *MAR* and *MDR* and RTL expression is

X:MD<--M[MAR]

If certain bits of a register are to be addressed we use RTL as follows:

X: IR ←*MDR[7:5]*

The statement above refers to transfer of three most significant bits of *MDR* to *IR*, a 3-bit register when *X*= l.

The arithmetic and logic operations of ALU that bring operands from *ACC* and *MDR* and store the result in *ACC* can be expressed in RTL in the following way

X:AC←*ACC&MDR* [logic AND]

X:ACC←*ACC* I *MDR* [logic OR]

X:ACC←*ACCEBMDR* [logic EX-OR]

X:ACC←*ACC'* [logic NOT]

X:ACC🡐*ACC]+MDR* [arithmetic addition]

X:ACC🡐*ACC-MDR* [arithmetic subtraction]

X:ACC🡐*ACC+* [increment by l]

And finally if data is to be shifted in a register say by I bit to left we can write

X: ACC[7:l] 🡐ACC[6:0],ACC[O) 🡐0

If such left shift occurs through carry the statement will be

X: ACC[7:l] 🡐ACC[6:0),ACC[0] 🡐*CY*

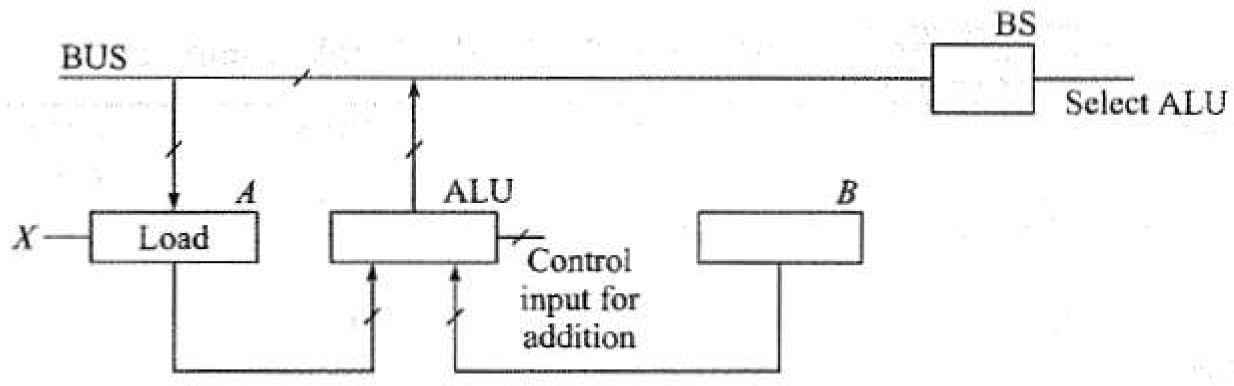
Normally, we come across these four kinds of micro operations namely

(i) Inter-Register transfer

ii) Arithmetic operation

(iii) Logic operation

(iv) Shift operation



EXECUTION Of INSTRUCTIONS, MACRO AND MICRO OPERATIONS

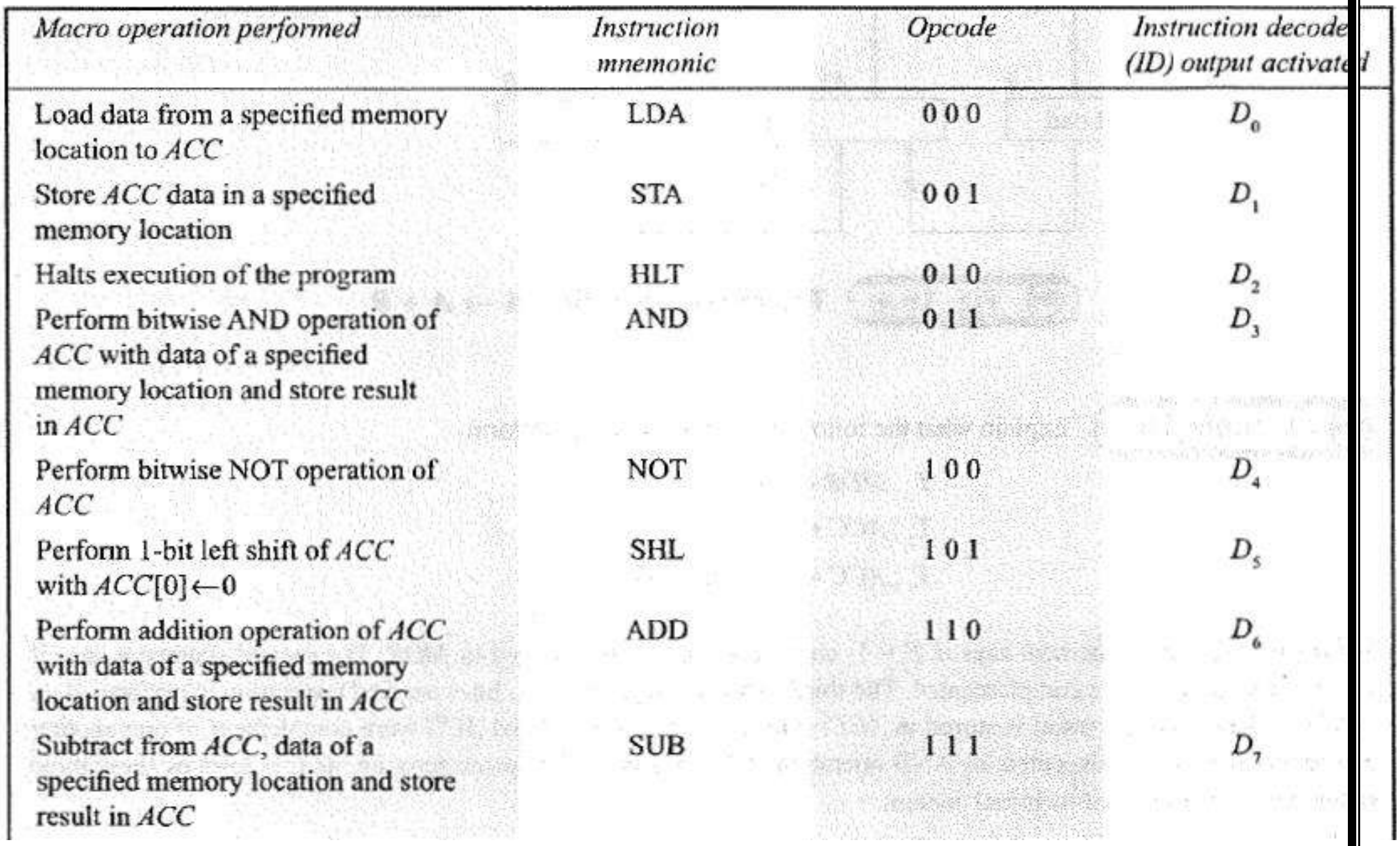
In a computer, ***execution of instructions*** is carried through macro operations which again can be subdivided into· micro operations.

In this section, we first define the macro operations that we want to be executed in the computer we are designing.

Next, we'll discuss micro operations necessary to execute each macro operation and it will be expressed through RTL.

Remember that we have assigned only 3-bits as op code and hence we can define 2^3 = 8 instructions or macro operations with them.

Table lists all the instructions, corresponding mnemonics (easy to remember short forms), op codes and 3-to-8 decoder (ID) output when *JR* is loaded with this op code.



Macro operation performed	*Instruction mnemonic*	*Opcode*	*Instruction decode (ID) output activat*
Load data from a specified memory location to *ACC*	LDA	0 0 0	D_0
Store *ACC* data in a specified memory location	STA	0 0 1	D_1
Halts execution of the program	HLT	0 1 0	D_2
Perform bitwise AND operation of *ACC* with data of a specified memory location and store result in *ACC*	AND	0 1 1	D_3
Perform bitwise NOT operation of *ACC*	NOT	1 0 0	D_4
Perform 1-bit left shift of *ACC* with $ACC[0] \leftarrow 0$	SHL	1 0 1	D_5
Perform addition operation of *ACC* with data of a specified memory location and store result in *ACC*	ADD	1 1 0	D_6
Subtract from *ACC*, data of a specified memory location and store result in *ACC*	SUB	1 1 1	D_7

Instruction Cycles:

To carry out each instruction or macro operation the computer has to go through three distinct phases or cycles.

In fetch cycle it brings the instruction or op code from the program memory. In decoding phase it decodes the op code and finally the execution is done in execute cycle.

These cycles together known as *instruction cycle* are again repeated for next instruction. It is understandable that fetch and decode phase will be same for all instructions in our simple computer as we have only single byte directly addressed instructions.

However, the execution cycle will be different for different instructions depending on the tasks the instruction wants to perform.

Fetch Cycle

An instruction cycle begins with *fetch cycle* when *TC* is reset to 0. Then, only T_0 output of TSD will be high and rest low.

PC now can be incremented to point to address of next location in program memory, which stores next instruction.

Content of *MAR* will be useful in execute phase if the op code makes some memory reference, the address of which remain available at *MAR.* In RTL the above operations can be represented as

T_0 $: MAR \leftarrow PC$

$T_1 : MDR \leftarrow M[MAR], \; PC \leftarrow PC + 1$

$T_2 : IR \leftarrow MDR[7:5], MAR \leftarrow .MDR[4:0]$

Decode Cycle

In *decode cycle* we decode the op code fetched from program memory. Since at T_2, register *IR* is loaded with op code and 3-to-8 decoder (ID) that decodes the op code is a combinatorial circuit, we finish decoding in T_2 itself. In RTL we express it as

$$T_2 : D_0 \ldots D_7 \leftarrow \text{DECODE}\ (JR)$$

Often, the 3rd statement of previously mentioned fetch cycle that loads *IR* with new opcode is considered a part of decode cycle or fetch-decode together is called fetch cycle.

Execute Cycle

Micro operations for each instruction are different and we list them first and then give the explanation.

LDA	$D_0T_3 : MDR \leftarrow \mathrm{M}[MAR]$
	$D_0T_4 : ACC \leftarrow MDR,\ TC \leftarrow 0$
STA	$D_1T_3 : MDR \leftarrow ACC$
	$D_1T_4 : \mathrm{M}[MAR] \leftarrow MDR,\ TC \leftarrow 0$
HLT	$D_2T_3 : S \leftarrow 1,\ TC \leftarrow 0$
AND	$D_3T_3 : MDR \leftarrow \mathrm{M}[MAR]$
	$D_3T_4 : ACC \leftarrow ACC\ \&\ MDR\ \ TC \leftarrow 0$
NOT	$D_4T_3 : ACC \leftarrow ACC',\ TC \leftarrow 0$
SHL	$D_5T_3 : ACC[7{:}1] \leftarrow ACC[6{:}0], ACC[0] \leftarrow 0, TC \leftarrow 0$
ADD	$D_6T_3 : MDR \leftarrow \mathrm{M}[MAR]$
	$D_6T_4 : ACC \leftarrow ACC + MDR,\ TC \leftarrow 0$
SUB	$D_7T_3 : MDR \leftarrow \mathrm{M}[MAR]$
	$D_7T_4 : ACC \leftarrow ACC - MDR, TC \leftarrow 0$

In operations like LDA, AND, ADD, SUB data is brought from memory, address of which is available in *MAR.* In executing STA the *MAR* content denotes the location where data is to be stored in memory.

Macro operations AND, NOT, ADD, SUB use ALU. When HLT is executed S flag is set which stops execution This of the program.

We reset *TC* and let the computer begin a new instruction cycle. This analysis can be extended to explain execution of other instructions.

At this point we make an important observation that all the instruction executions are completed within 5 clock cycles (T_0 to T_4) and hence a 3-bit counter, which can count up to 8 is sufficient as *TC* in our simple computer.

DESIGN OF CONTROL UNIT

The control unit is primarily a combinatorial circuit that supplies necessary controls inputs to all the important hardware elements of the computer.

This takes timing information from computer master clock and is thus responsible for providing necessary ***timing and control*** information.

The path through which these signals travel to reach different parts of a computer is called *control path.* Often we assign a group of wires, called *control bus* as shared path for this.

The control logic is arrived at from

(i) basic computer architecture we have adopted in the beginning,

(ii) conditions appearing at left hand side of symbol ':' in RTL statements for our simple computer, given in previous section.

(iii) certain other issues, e.g. power-on-reset, control variables need to be activated for intended operation of a particular hardware, etc.

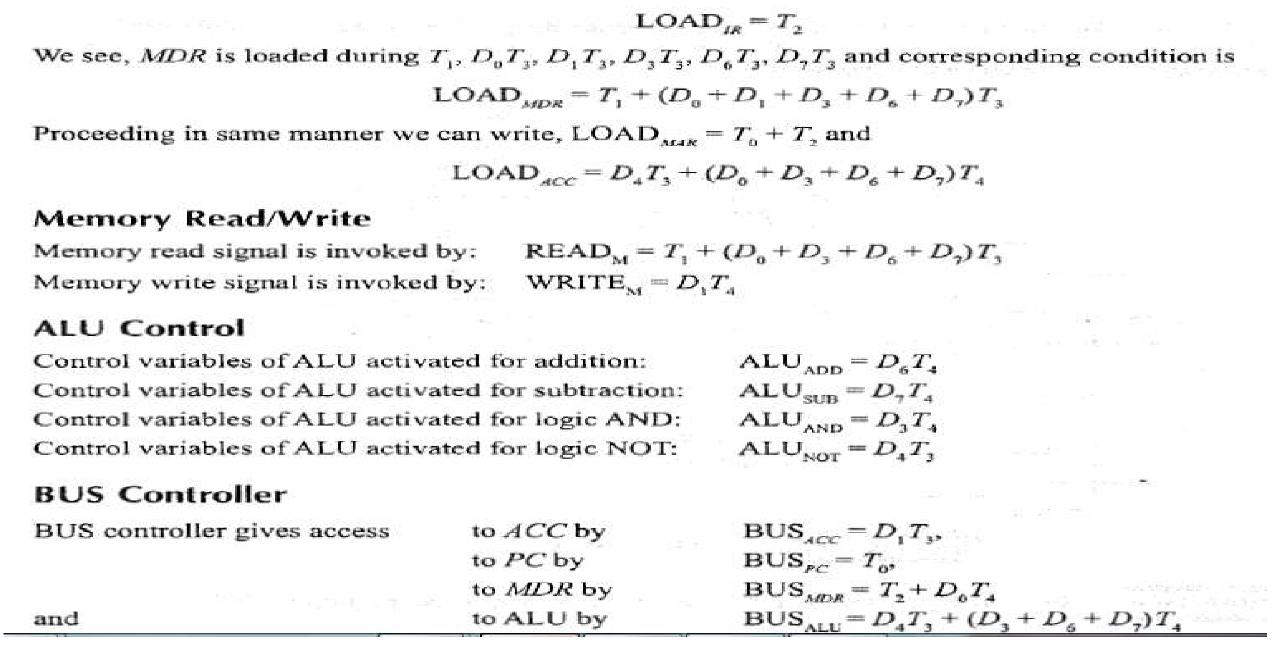
Loading Registers

Let us first see when parallel load control of*JR* is to be activated.

We find from discussion of previous section, only during T_2 it is loaded. So TSD (Timing counter decoder)T output $_2$ can be directly connected as parallel load control input of *JR.*

Obviously, at that time **BUS selector (BS**) should place content of *lvfDR* into BUS. This we'll discuss while designing control for BS.

What happens if we allow loading of *IR* say, in every clock cycle instead of above? Whenever there is some data made available in BUS by any hardware 3 MSB of that will be loaded into *JR;* ID (decoder) will immediately change and execution corresponding to a different opcode, not the intended one, may begin.



$$\text{LOAD}_{IR} = T_2$$

We see, *MDR* is loaded during $T_1, D_0T_3, D_1T_3, D_3T_3, D_6T_3, D_7T_3$ and corresponding condition is

$$\text{LOAD}_{MDR} = T_1 + (D_0 + D_1 + D_3 + D_6 + D_7)T_3$$

Proceeding in same manner we can write, $\text{LOAD}_{MAR} = T_0 + T_2$ and

$$\text{LOAD}_{ACC} = D_4T_3 + (D_0 + D_3 + D_6 + D_7)T_4$$

Memory Read/Write

Memory read signal is invoked by: $\text{READ}_M = T_1 + (D_0 + D_3 + D_6 + D_7)T_3$

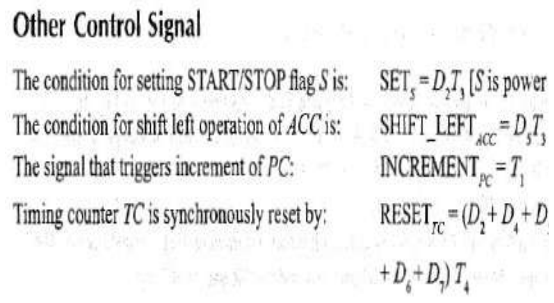
Memory write signal is invoked by: $\text{WRITE}_M = D_1T_4$

ALU Control

Control variables of ALU activated for addition:	$\text{ALU}_{\text{ADD}} = D_6T_4$
Control variables of ALU activated for subtraction:	$\text{ALU}_{\text{SUB}} = D_7T_4$
Control variables of ALU activated for logic AND:	$\text{ALU}_{\text{AND}} = D_3T_4$
Control variables of ALU activated for logic NOT:	$\text{ALU}_{\text{NOT}} = D_4T_3$

BUS Controller

BUS controller gives access	to *ACC* by	$\text{BUS}_{ACC} = D_1T_3$,
	to *PC* by	$\text{BUS}_{PC} = T_0$,
	to *MDR* by	$\text{BUS}_{MDR} = T_2 + D_6T_4$
and	to ALU by	$\text{BUS}_{\text{ALU}} = D_4T_3 + (D_3 + D_6 + D_7)T_4$



Other Control Signal

The condition for setting START/STOP flag *S* is:	$\text{SET}_S = D_2 T_3$ [*S* is power
The condition for shift left operation of *ACC* is:	$\text{SHIFT\_LEFT}_{ACC} = D_5 T_3$
The signal that triggers increment of *PC*:	$\text{INCREMENT}_{PC} = T_1$
Timing counter *TC* is synchronously reset by:	$\text{RESET}_{TC} = (D_2 + D_4 + D$
	$+ D_6 + D_7) T_4$

Finally, the master clock remains enabled if flag Sis not set. Thus ENABLE clock K = S'

Based on these equations the control unit of our simple computer can be made.

We show the controlcircuit of *ACC*, *TC* and *TSD*, BS in following three examples. Refer to problems.

PROGRAMMING COMPUTER

Now that our simple computer is ready with hardware and instruction sets let us see what computer program can solve the problem with which we started designing our simple machine.

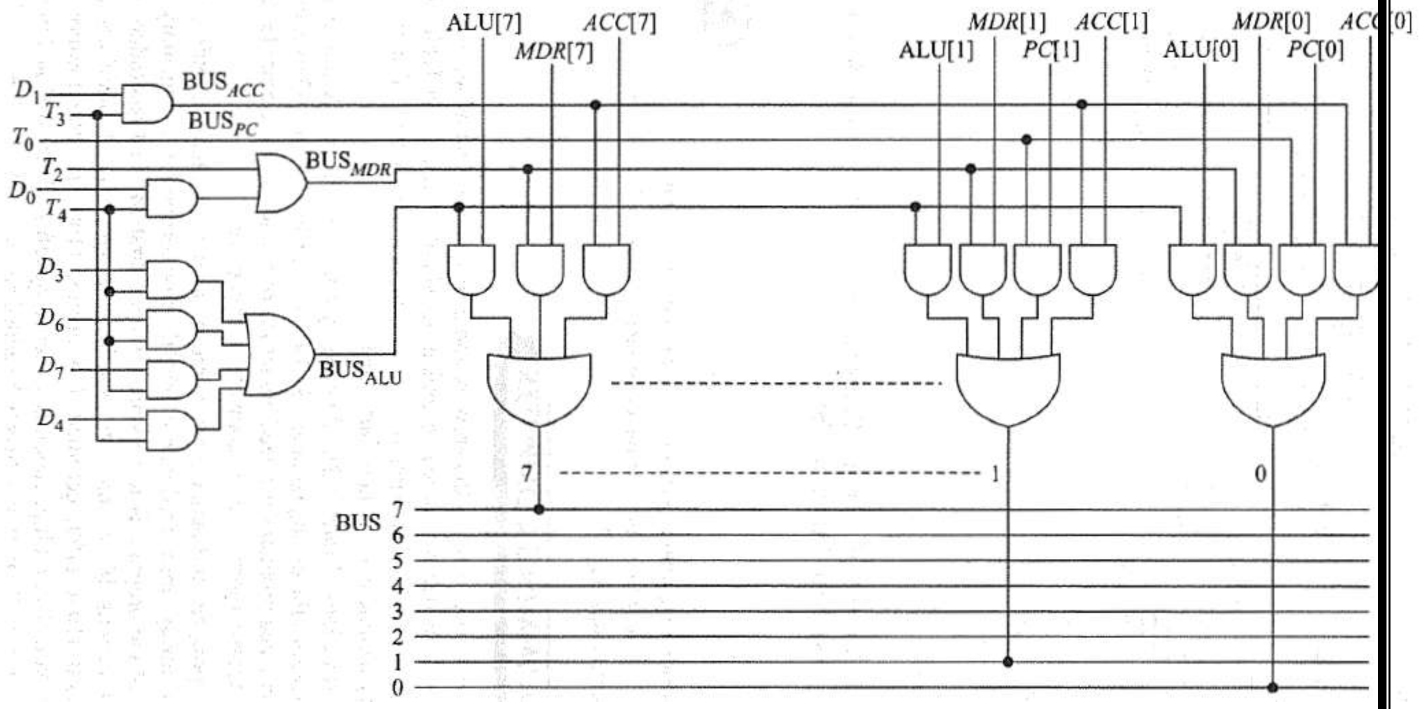
we present the program in mnemonics along with comments on job done by each instruction. Program in binary code as exists in 32 x 8 memory module will be shown after that.

Thus we need 14 instructions all single byte to solve the problem in our simple computer. We need 12 memory locations for storing numbers. So 14 + 12 = 26 bytes of our 32 byte memory are used for this problem.

For bigger sized problems we need bigger memory and for more complex problem additional instruction sets and, of course, more complex computer architecture.

Now let us see how program and data remain stored in memory in binary numbers. We know that due to power-on-reset *PC* is always initialized with 00000, the first location of the memory where first instruction of the program is to be stored.

We use first 14 locations (address 00000 to 01101) of memory to store instructions. If we store data used in the program, i.e. 11 numbers in next consecutive locations then addresses O1110 to 11000 get filled. The location 11001, i.e. 26th location of memory can be used to store the result.



Program Execution

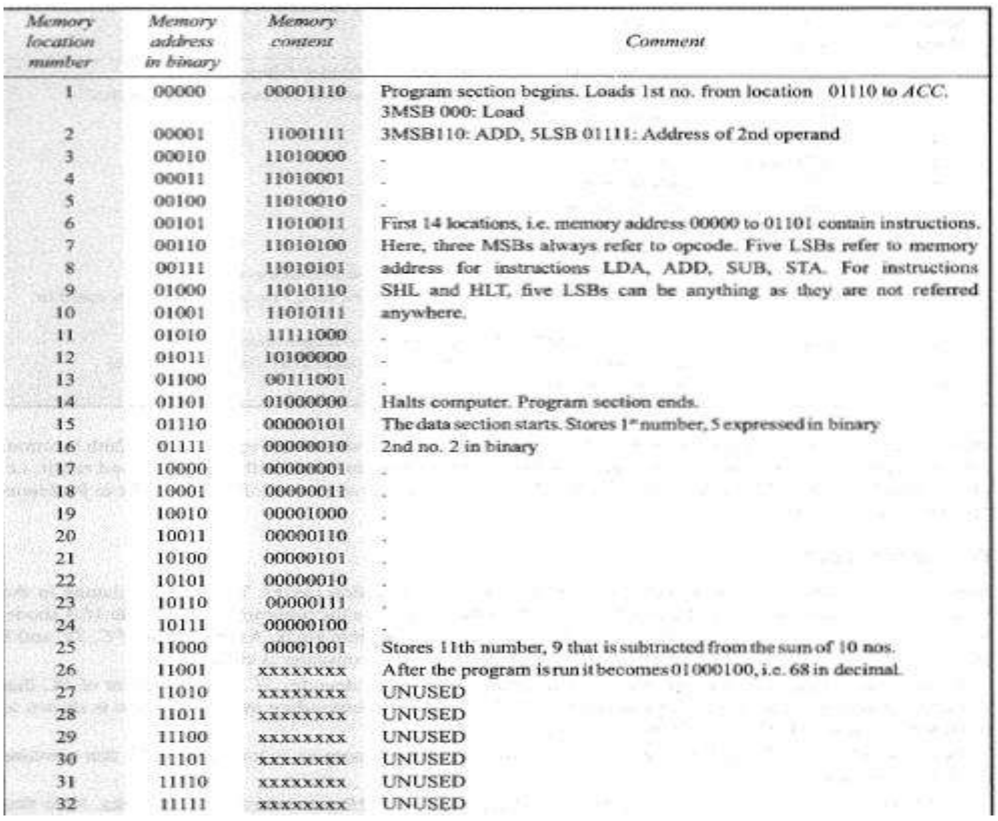
Now let us see how the program gets executed in first few instruction cycles. We note the change in the value of the registers along with ID and TSD in each clock cycle since the program begins. Table shows sequential progress of our simple computer with every trigger of system clock.

As told before *PC*, *TC* and S are power on reset. They all contain zero in the beginning when the computer is switched on.

In first clock cycle, the machine is in T_0 state given by TSD that decodes *TC.* At T_0, content of *PC* that contains the starting address of the program is copied to *MAR.* Corresponding micro operation is shown in rightmost column of Table 16.4. *TC* is incremented by 1.

In next clock cycle, *TC* and *PC* are incremented by 1, data from memory is loaded to *MDR* that contains the first instruction.

This completes the first instruction cycle. Note that timing counter (TC) is to be reset after execution of data transfer from *MDR* to *ACC* and that begins the next



Memory location number	*Memory address in binary*	*Memory content*	*Comment*
1	00000	00001110	Program section begins. Loads 1st no. from location 01110 to *ACC*. 3MSB 000: Load
2	00001	11001111	3MSB110: ADD, 5LSB 01111: Address of 2nd operand
3	00010	11010000	-
4	00011	11010001	-
5	00100	11010010	.
6	00101	11010011	First 14 locations, i.e. memory address 00000 to 01101 contain instructions.
7	00110	11010100	Here, three MSBs always refer to opcode. Five LSBs refer to memory
8	00111	11010101	address for instructions LDA, ADD, SUB, STA. For instructions
9	01000	11010110	SHL and HLT, five LSBs can be anything as they are not referred
10	01001	11010111	anywhere.
11	01010	11111000	-
12	01011	10100000	-
13	01100	00111001	.
14	01101	01000000	Halts computer. Program section ends.
15	01110	00000101	The data section starts. Stores 1st number, 5 expressed in binary
16	01111	00000010	2nd no. 2 in binary
17	10000	00000001	-
18	10001	00000011	.
19	10010	00001000	.
20	10011	00000110	.
21	10100	00000101	.
22	10101	00000010	.
23	10110	00000111	.
24	10111	00000100	-
25	11000	00001001	Stores 11th number, 9 that is subtracted from the sum of 10 nos.
26	11001	xxxxxxxx	After the program is run it becomes 01000100, i.e. 68 in decimal.
27	11010	xxxxxxxx	UNUSED
28	11011	xxxxxxxx	UNUSED
29	11100	xxxxxxxx	UNUSED
30	11101	xxxxxxxx	UNUSED
31	11110	xxxxxxxx	UNUSED
32	11111	xxxxxxxx	UNUSED

The fetch cycle is repeated in clock cycle 6 to 8. Since the instruction fetched is ADD (op code 110) corresponding micro operations are performed in clock cycles 9 and 10 followed by next instruction fetch, starting again at 11th clock cycle.

This continues till we reach 14th instruction HLT which when executed, sets S flag. This inhibits the system clock output in our design; thus content of all registers and memory will remain unchanged after that till the computer is switched off.

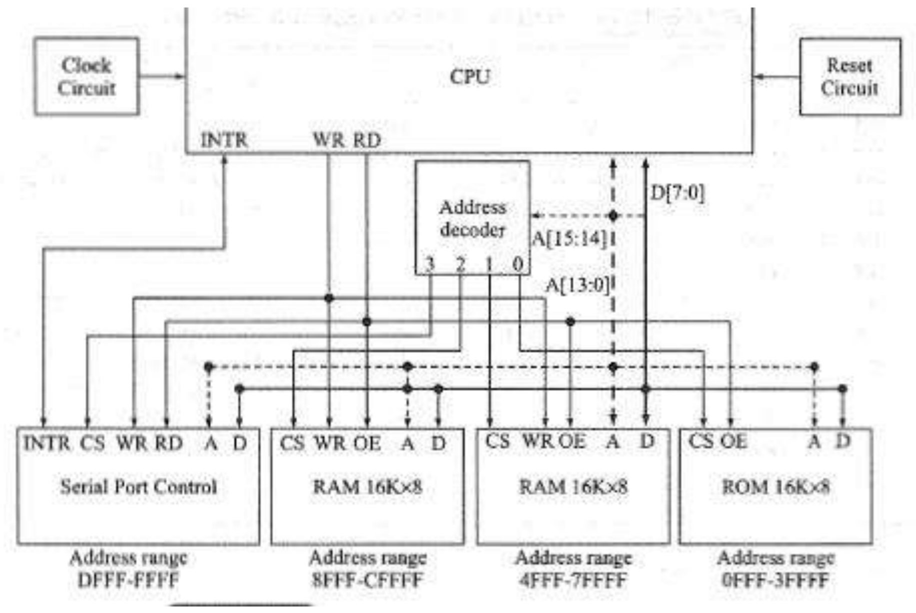
Concluding Remark

Before we conclude our computer design exercise let us see what we have achieved and what more is needed to make this computer fully functional.

We have designed a simple processor comprising register arrays, flag, small memory, BUS and control unit. In short, we have designed a ***central processing unit*** **(CPU)** that connects to a small memory module and is able to execute programs built on a small instruction set.

What we have not discussed is how data is entered into computer from an external device say, keypad and also how it displays data in some output device say, a monitor.

These are covered in detail in titles related to modem computer design courses and an interested reader can refer to the same.



Basic architecture of an 8-bit computer

signals, connected to each of the memory and output module generating unique address ranges as specified in the bottom of the figure. The calculation is as follows.

CPU read is enabled by activating the control signal, **RD (Read).** This, in turn, requests outputs of the devices from which data is to be read to be enabled through **OE (Output Enable)** or through RD, if it is a serial input-output port, following which CPU takes the value from data bus.

The control **signal WR (Write)** is activated to enable CPU writing to devices. Note that WR and RD should not be activated simultaneously.

This is usually used during *power on* of the computer and also in between, if the computer is asked to stop all operations and start afresh. The other time a computer may be asked to stop its usual fetch-decode-execute operations, but only temporarily, is when an *interrupt* is invoked. Then the computer's present state is stored in a designated memory space called *stack.*

The computer comes back to its usual operating state once the interrupt is served usually through a ***interrupt service routine*** **(ISR).** There could be both software and hardware interrupts. The serial port control block shows how a hardware interrupt can ask service from CPU by activating INTR.

Note that the *mask able* ***interrupts*** can be masked (disabled) by writing into a control register while *non-mask able interrupts* cannot be disabled.