

LECTURE NOTES
ON
Digital IC Applications using VHDL

III B.Tech II semester (IARE - R16)
(2018 - 19)

Dr. Vijay Vallabhuni, Professor
Dr. K Nehru, Professor
Mr. D Khalandar Basha, Assistant professor



ELECTRONICS AND COMMUNICATION ENGINEERING
INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
DUNDIGAL, HYDERABAD - 500043

UNIT - I

CMOS LOGIC

Logic Signals and Gates

Digital logic hides the pitfalls of the analog world by mapping the infinite set of real values for a physical quantity into two subsets corresponding to just two possible numbers or *logic values*—0 and 1. As a result, digital logic circuits can be analyzed and designed functionally, using switching algebra, tables, and other abstract means to describe the operation of well-behaved 0s and 1s in a circuit.

A logic value, 0 or 1, is often called a *binary digit*, or *bit*. If an application requires more than two discrete values, additional bits may be used, with a set of n bits representing 2^n different values. With most phenomena, there is an undefined region between the 0 and 1 states (e.g., voltage = 1.8 V, dim light, capacitor slightly charged, etc.). This undefined region is needed so that the 0 and 1 states can be unambiguously defined and reliably detected. Noise can more easily corrupt results if the boundaries separating the 0 and 1 states are too close. When discussing electronic logic circuits such as CMOS and TTL, digital designers often use the words **LOW** and **HIGH** in place of 0 and 1 to remind them that they are dealing with real circuits, not abstract quantities:

LOW A signal in the range of algebraically lower voltages, which is interpreted as a logic 0.

HIGH A signal in the range of algebraically higher voltages, which is interpreted as a logic 1.

Note that the assignments of 0 and 1 to **LOW** and **HIGH** are somewhat arbitrary. Assigning 0 to **LOW** and 1 to **HIGH** seems most natural, and is called *positive logic*. The opposite assignment, 1 to **LOW** and 0 to **HIGH**, is not often used, and is called *negative logic*.

Because a wide range of physical values represent the same binary value, digital logic is highly immune to component and power supply variations and noise. Furthermore, *buffer amplifier* circuits can be used to regenerate **weak** values into **strong** ones, so that digital signals can be transmitted over arbitrary distances without loss of information. For example, a buffer amplifier for CMOS logic converts any **HIGH** input voltage into an output very close to 5.0 V, and any **LOW** input voltage into an output very close to 0.0 V. A logic circuit can be represented with a minimum amount of detail simply as a **black box** with a certain number of inputs and outputs. For example, Figure shows a logic circuit with three inputs and one output. However, this representation does not describe how the circuit responds to input signals.

From the point of view of electronic circuit design, it takes a lot of information to describe the precise electrical behavior of a circuit. However, since the inputs of a digital logic circuit can be viewed as taking on only discrete 0 and 1 values, the circuit's **logical** operation can be described with a table that ignores electrical behavior and lists only discrete 0 and 1 values.

A logic circuit whose outputs depend only on its current inputs is called a *combinational circuit*. Its operation is fully described by a *truth table* that lists all combinations of input values and the output value(s) produced by each one.

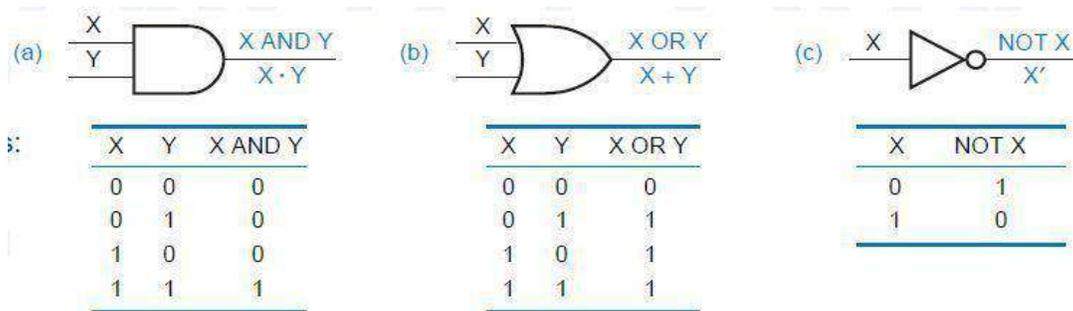
Table is the truth table for a logic circuit with three inputs X, Y, and Z and a single output F.

| X | Y | Z | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

A circuit with memory, whose outputs depend on the current input *and* the sequence of past inputs, is called a *sequential circuit*. The behavior of such a circuit may be described by a *state table* that specifies its output and next state as functions of its current state and input.

The gates' functions are easily defined in words:

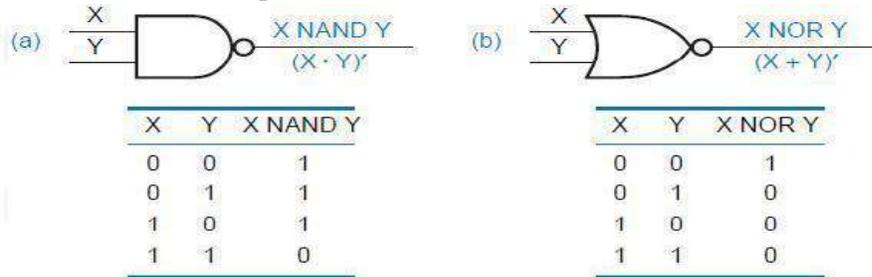
- An AND gate produces a 1 output if and only if all of its inputs are 1.
- An OR gate produces a 1 if and only if one or more of its inputs are 1.
- A NOT gate, usually called an *inverter*, produces an output value that is the opposite of its input value.



The circle on the inverter symbol's output is called an *inversion bubble*, and is used in this and other gate symbols to denote *-inverting* behavior. Notice that in the definitions of AND and OR functions, we only had to state the input conditions for which the output is 1, because there is only one possibility when the output is not 1—it must be 0. Two more logic functions are

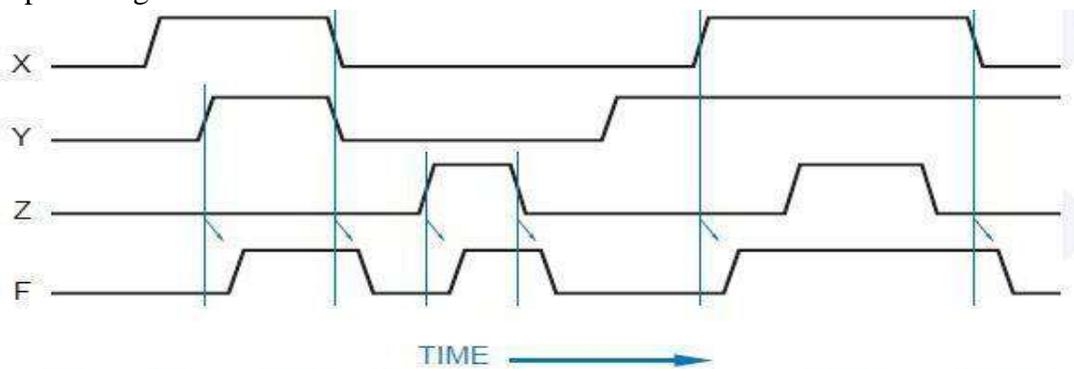
obtained by combining NOT with an AND or OR function in a single gate. Figure 3-3 shows the truth tables and symbols for these gates; Their functions are also easily described in words:

- A NAND *gate* produces the opposite of an AND gate's output, a 0 if and only if all of its inputs are 1.
- A NOR *gate* produces the opposite of an OR gate's output, a 0 if and only if one or more of its inputs are 1.



As with AND and OR gates, the symbols and truth tables for NAND and NOR may be extended to gates with any number of inputs.

Real logic circuits also function in another analog dimension—time. For example, Figure is a *timing diagram* that shows how the circuit might respond to a time-varying pattern of input signals. The timing diagram shows that the logic signals do not change between 0 and 1 instantaneously, and also that there is a lag between an input change and the corresponding output change.



Logic Families

There are many, many ways to design an electronic logic circuit.

1. The first electrically controlled logic circuits, developed at Bell Laboratories in 1930s, were based on relays.
2. In the mid-1940s, the first electronic digital computer, the Eniac, used logic circuits based on vacuum tubes. The Eniac had about 18,000 tubes and a similar number of logic gates, not a lot by today's standards of microprocessor chips with tens of millions of transistors. However, the Eniac could hurt you a lot more than a chip could if it fell on you—it was 100 feet long, 10 feet high, 3 feet deep, and consumed 140,000 watts of power!

3. The inventions of the *semiconductor diode* and the *bipolar junction transistor* allowed the development of smaller, faster, and more capable computers in the late 1950s.
4. In the 1960s, the invention of the *integrated circuit (IC)* allowed multiple diodes, transistors, and other components to be fabricated on a single chip, and computers got still better.

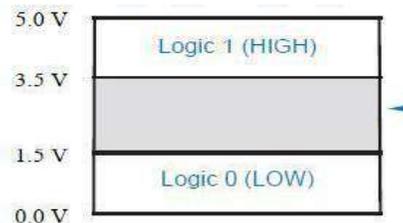
A logic family: is a collection of different integrated-circuit chips that have similar input, output, and internal circuit characteristics, but that perform different logic functions. Chips from the same family can be interconnected to perform any desired logic function.

CMOS Logic

The functional behavior of a CMOS logic circuit is fairly easy to understand, even if your knowledge of analog electronics is not particularly deep. The basic (and typically only) building blocks in CMOS logic circuits are MOS transistors, described shortly. Before introducing MOS transistors and CMOS logic circuits, we must talk about logic levels.

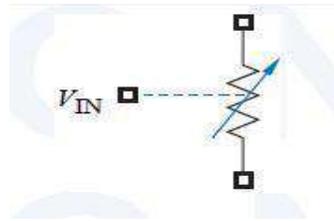
CMOS Logic Levels

Abstract logic elements process binary digits, 0 and 1. However, real logic circuits process electrical signals such as voltage levels. In any logic circuit, there is a range of voltages (or other circuit conditions) that is interpreted as a logic 0, and another, nonoverlapping range that is interpreted as a logic 1. A typical CMOS logic circuit operates from a 5-volt power supply. Such a circuit may interpret any voltage in the range 0–1.5 V as a logic 0, and in the range 3.5–5.0 V as a logic 1. Thus, the definitions of LOW and HIGH for 5-volt CMOS logic are as shown in Figure. Voltages in the intermediate range are not expected to occur except during signal transitions, and yield undefined logic values (i.e., a circuit may interpret them as either 0 or 1). CMOS circuits using other power supply voltages, such as 3.3 or 2.7 volts, partition the voltage range similarly.



MOS Transistors

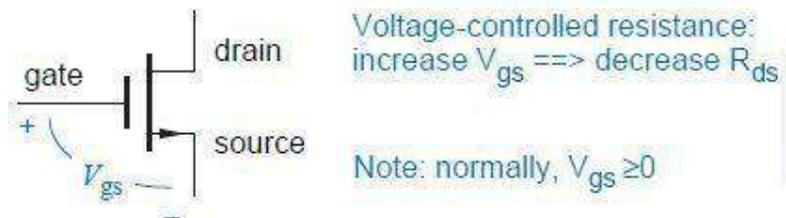
A MOS transistor can be modeled as a 3-terminal device that acts like a voltage-controlled resistance. As suggested by Figure, an input voltage applied to one terminal controls the resistance between the remaining two terminals. In digital logic applications, a MOS transistor is operated so its resistance is always either very high (and the transistor is —off) or very low (and the transistor is —on).



There are two types of MOS transistors, *n*-channel and *p*-channel; the names refer to the type of semiconductor material used for the resistance-controlled terminals.

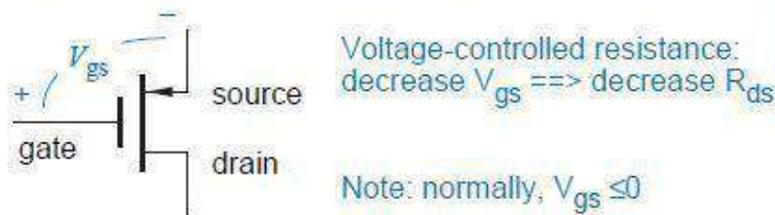
n-channel MOS (NMOS) transistor:

The circuit symbol for an *n*-channel MOS (NMOS) transistor is shown in Figure . The terminals are called *gate*, *source*, and *drain*. (Note that the -gate of a MOS transistor has nothing to do with a -logic gate.) As you might guess from the orientation of the circuit symbol, the drain is normally at a higher voltage than the source. The voltage from gate to source (V_{gs}) in an NMOS transistor is normally zero or positive. If $V_{gs} = 0$, then the resistance from drain to source (R_{ds}) is very high, on the order of a megohm (10^6 ohms) or more. As we increase V_{gs} (i.e., increase the voltage on the gate), R_{ds} decreases to a very low value, 10 ohms or less in some devices.



p-channel MOS

The circuit symbol for a *p*-channel MOS (PMOS) transistor is shown in Figure. Operation is analogous to that of an NMOS transistor, except that the source is normally at a higher voltage than the drain, and V_{gs} is normally zero or negative. If V_{gs} is zero, then the resistance from source to drain (R_{ds}) is very high. As we algebraically decrease V_{gs} (i.e., *decrease* the voltage on the gate), R_{ds} decreases to a very low value.

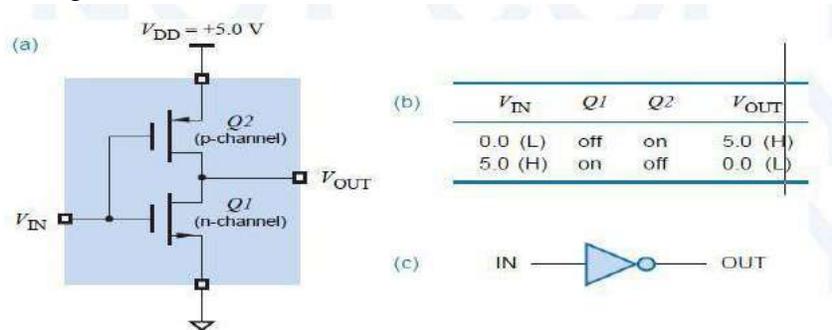


The gate of a MOS transistor has a very high impedance. That is, the gate is separated from the source and the drain by an insulating material with a very high resistance. However, the gate voltage creates an electric field that enhances or retards the flow of current between source and drain. This is the -field effect in the -MOSFET name.

Basic CMOS Inverter Circuit

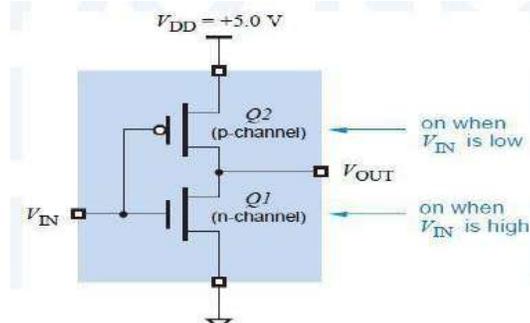
NMOS and PMOS transistors are used together in a complementary way to form *CMOS logic*. The simplest CMOS circuit, a logic inverter, requires only one of each type of transistor, connected as shown in Figure. The power supply voltage, V_{DD} , typically may be in the range 2–6 V, and is most often set at 5.0 V for compatibility with TTL circuits. Ideally, the functional behavior of the CMOS inverter circuit can be characterized by just two cases tabulated in Figure:

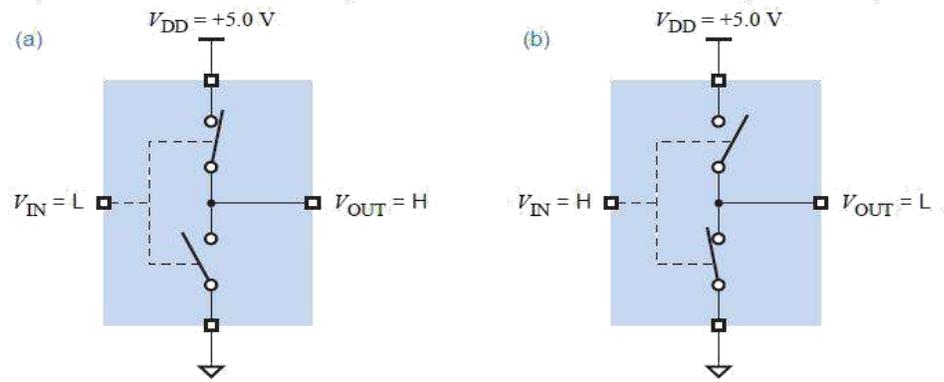
1. V_{IN} is 0.0 V. In this case, the bottom, n -channel transistor $Q1$ is off, since its V_{gs} is 0, but the top, p -channel transistor $Q2$ is on, since its V_{gs} is a large negative value (5.0 V). Therefore, $Q2$ presents only a small resistance between the power supply terminal (V_{DD} , 5.0 V) and the output terminal (V_{OUT}), and the output voltage is 5.0 V.
2. V_{IN} is 5.0 V. Here, $Q1$ is on, since its V_{gs} is a large positive value (+5.0 V), but $Q2$ is off, since its V_{gs} is 0. Thus, $Q1$ presents a small resistance between the output terminal and ground, and the output voltage is 0 V.



With the foregoing functional behavior, the circuit clearly behaves as a logical inverter, since a 0-volt input produces a 5-volt output, and vice versa. Another way to visualize CMOS operation uses switches. As shown in Figure, the n -channel (bottom) transistor is modeled by a normally-open switch, and the p -channel (top) transistor by a normally-closed switch. Applying a HIGH voltage changes each switch to the opposite of its normal state, as shown in (b).

The switch model gives rise to a way of drawing CMOS circuits that makes their logical behavior more readily apparent. As shown in Figure, different symbols are used for the p - and n -channel transistors to reflect their logical behavior. The n -channel transistor ($Q1$) is switched —on, and current flows between source and drain, when a HIGH voltage is applied to its gate; this seems natural enough. The p -channel transistor ($Q2$) has the opposite behavior. It is —on when a LOW voltage is applied; the inversion bubble on its gate indicates this inverting behavior.





CMOS NAND and NOR Gates

Both NAND and NOR gates can be constructed using CMOS. A k -input gate uses k p -channel and k n -channel transistors. Figure shows a 2-input CMOS NAND gate. If either input is LOW, the output Z has a low-impedance connection to V_{DD} through the corresponding on p -channel transistor, and the path to ground is blocked by the corresponding off n -channel transistor. If both inputs are HIGH, the path to V_{DD} is blocked, and Z has a low-impedance connection to ground. Figure shows the switch model for the NAND gate's operation.

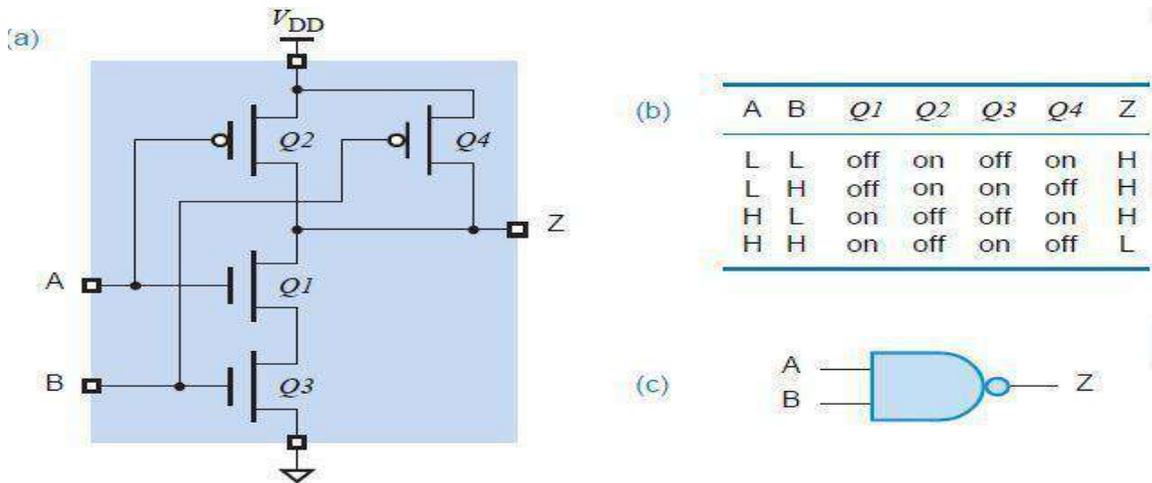
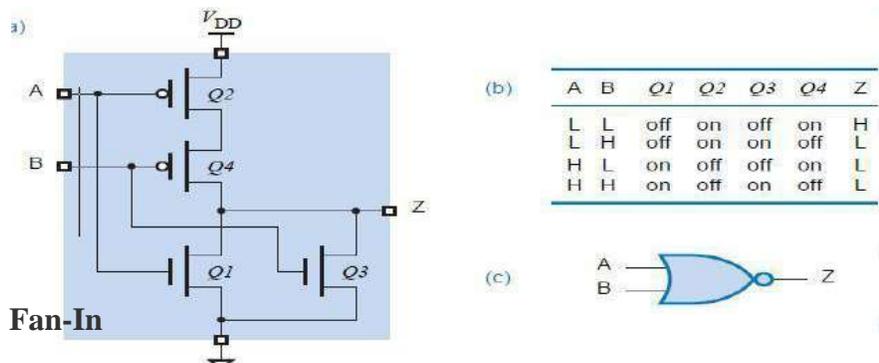


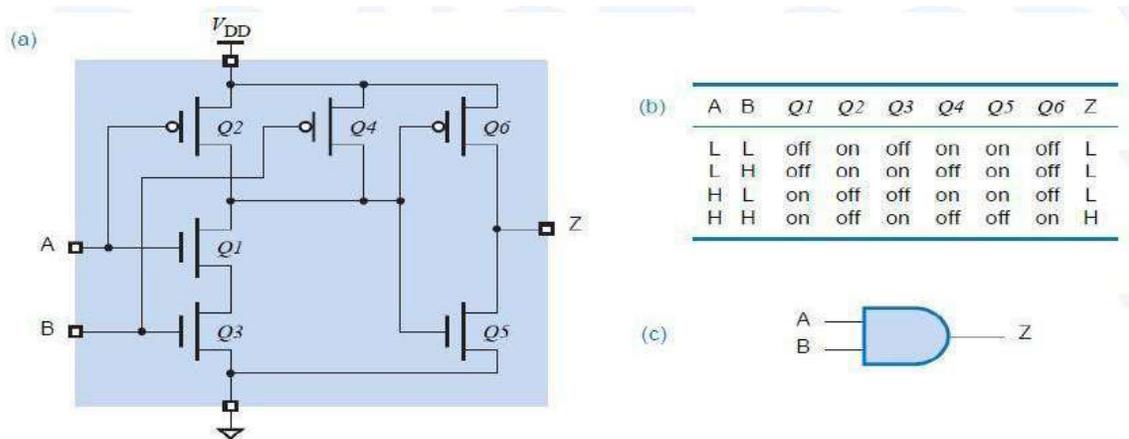
Figure shows a CMOS NOR gate. If both inputs are LOW, the output Z has a low-impedance connection to V_{DD} through the on p -channel transistors, and the path to ground is blocked by the off n -channel transistors. If either input is HIGH, the path to V_{DD} is blocked, and Z has a low-impedance connection to ground.



The number of inputs that a gate can have in a particular logic family is called the logic family's *fan-in*. As the number of inputs is increased, CMOS gate designers may compensate by increasing the size of the series transistors to reduce their resistance and the corresponding switching delay. However, at some point this becomes inefficient or impractical. Gates with a large number of inputs can be made faster and smaller by cascading gates with fewer inputs.

Noninverting Gates

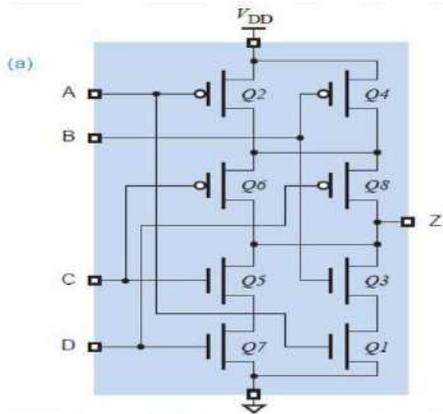
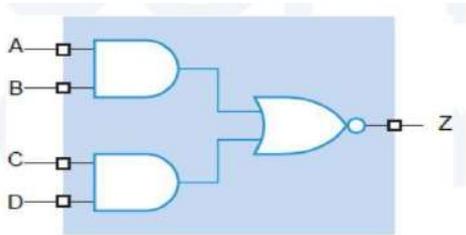
In CMOS, and in most other logic families, the simplest gates are inverters, and the next simplest are NAND gates and NOR gates. A logical inversion comes –for free, and it typically is not possible to design a noninverting gate with a smaller number of transistors than an inverting one. CMOS noninverting buffers and AND and OR gates are obtained by connecting an inverter to the output of the corresponding inverting gate.



CMOS AND-OR-INVERT and OR-AND-INVERT Gates

CMOS circuits can perform two levels of logic with just a single –level of transistors. For example, the circuit in Figure is a two-wide, two-input CMOS AND-OR-INVERT (AOI) gate. The function table for this circuit is shown in (b) and a logic diagram for this function using AND and NOR gates is shown in Figure 3-21. Transistors can be added to or removed from this circuit to obtain an AOI function with a different number of ANDs or a different number of inputs per AND.

The contents of each of the Q1–Q8 columns in Figure 0(b) depends only on the input signal connected to the corresponding transistor's gate. The last column is constructed by examining each input combination and determining whether Z is connected to VDD or ground by —onll transistors for that input combination. Note that Z is never connected to both VDD and ground for any input combination; in such a case the output would be a non-logic value some where between LOW and HIGH, and the output structure would consume excessive power due to the low-impedance connection between VDD and ground.

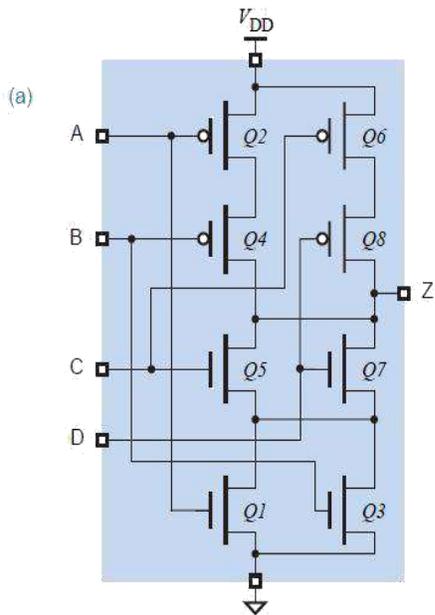


Share to create, manage and send PDF files.

(b)

| A | B | C | D | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q8 | Z |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|---|
| L | L | L | L | off | on | off | on | off | on | off | on | H |
| L | L | L | H | off | on | off | on | off | on | off | on | H |
| L | L | H | L | off | on | off | on | off | on | off | on | H |
| L | L | H | H | off | on | off | on | off | on | off | on | L |
| L | H | L | L | off | on | on | off | off | on | off | on | H |
| L | H | L | H | off | on | on | off | off | on | off | on | H |
| L | H | H | L | off | on | on | off | on | off | on | off | H |
| L | H | H | H | off | on | on | off | on | off | on | off | L |
| H | L | L | L | on | off | off | on | off | on | off | on | H |
| H | L | L | H | on | off | off | on | off | on | on | off | H |
| H | L | H | L | on | off | off | on | on | off | off | on | H |
| H | L | H | H | on | off | off | on | on | off | on | off | L |
| H | H | L | L | on | off | on | off | off | on | off | on | L |
| H | H | L | H | on | off | on | off | off | on | on | off | L |
| H | H | H | L | on | off | on | off | on | off | off | on | L |
| H | H | H | H | on | off | on | off | on | off | on | off | L |

A circuit can also be designed to perform an OR-AND-INVERT function. For example, Figure is a two-wide, two-input CMOS OR-AND-INVERT (OAI) gate. The function table for this circuit is shown in (b); the values in each column are determined just as we did for the CMOS AOI gate.



(b)

| A | B | C | D | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q8 | Z |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|---|
| L | L | L | L | off | on | off | on | off | on | off | on | H |
| L | L | L | H | off | on | off | on | off | on | on | off | H |
| L | L | H | L | off | on | off | on | on | off | off | on | H |
| L | L | H | H | off | on | off | on | on | off | on | off | H |
| L | H | L | L | off | on | on | off | off | on | off | on | H |
| L | H | L | H | off | on | on | off | off | on | on | off | L |
| L | H | H | L | off | on | on | off | on | off | off | on | L |
| L | H | H | H | off | on | on | off | on | off | on | off | L |
| H | L | L | L | on | off | off | on | off | on | off | on | H |
| H | L | L | H | on | off | off | on | off | on | on | off | L |
| H | L | H | L | on | off | off | on | on | off | off | on | L |
| H | L | H | H | on | off | off | on | on | off | on | off | L |
| H | H | L | L | on | off | on | off | off | on | off | on | H |
| H | H | L | H | on | off | on | off | off | on | on | off | L |
| H | H | H | L | on | off | on | off | on | off | off | on | L |
| H | H | H | H | on | off | on | off | on | off | on | off | L |

3.4 Electrical Behavior of CMOS Circuits

- *Logic voltage levels.* CMOS devices operating under normal conditions are guaranteed to produce output voltage levels within well-defined LOW and HIGH ranges. And they recognize LOW and HIGH input voltage levels over somewhat wider ranges. CMOS manufacturers specify these ranges and operating conditions very carefully to ensure compatibility among different devices in the same family, and to provide a degree of interoperability (if you're careful) among devices in different families.
- *DC noise margins.* Nonnegative DC noise margins ensure that the highest LOW voltage produced by an output is always lower than the highest voltage that an input can reliably interpret as LOW, and that the lowest HIGH voltage produced by an output is always higher than the lowest voltage that an input can reliably interpret as HIGH. A good understanding of noise margins is especially important in circuits that use devices from a number of different families.
- *Fanout.* This refers to the number and type of inputs that are connected to a given output. If too many inputs are connected to an output, the DC noise margins of the circuit may be inadequate. Fanout may also affect the speed at which the output changes from one state to another.
- *Speed.* The time that it takes a CMOS output to change from the LOW state to the HIGH state, or vice versa, depends on both the internal structure of the device and the characteristics of the other devices that it drives, even to the extent of being affected by the wire or printed-circuit-board traces connected to the output. We'll look at two separate components of –speed— transition time and propagation delay.
- *Power consumption.* The power consumed by a CMOS device depends on a number of factors, including not only its internal structure, but also the input signals that it receives, the other devices that it drives, and how often its output changes between LOW and HIGH.
- *Noise.* The main reason for providing engineering design margins is to ensure proper circuit operation in the presence of noise. Noise can be generated by a number of sources; several of them are listed below, from the least likely to the (perhaps surprisingly) most likely:
 - Cosmic rays.
 - Magnetic fields from nearby machinery.
 - Power-supply disturbances.
 - The switching action of the logic circuits themselves.
- *Electrostatic discharge.* Would you believe that you can destroy a CMOS device just by touching it?
- *Open-drain outputs.* Some CMOS outputs omit the usual *p*-channel pullup transistors. In the HIGH state, such outputs are effectively a –no-connection, which is useful in some applications.
- *Three-state outputs.* Some CMOS devices have an extra –output enable control input that can be used to disable both the *p*-channel pull-up transistors and the *n*-channel pull-down transistors. Many such device outputs can be tied together to create a multisource bus, as long as the control logic is arranged so that at most one output is enabled at a time.

CMOS Steady-State Electrical Behavior

Logic Levels and Noise Margins

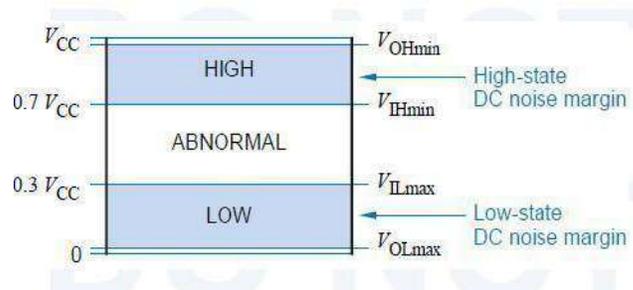
V_{OHmin} The minimum output voltage in the HIGH state.

V_{IHmin} The minimum input voltage guaranteed to be recognized as a HIGH.

V_{ILmax} The maximum input voltage guaranteed to be recognized as a LOW.

V_{OLmax} The maximum output voltage in the LOW state.

The input voltages are determined mainly by switching thresholds of the two transistors, while the output voltages are determined mainly by the —onll resistance of the transistors.



The power-supply voltage V_{CC} and ground are often called the *powersupplyrails*. CMOS levels are typically a function of the power-supply rails:

V_{OHmin} $V_{CC} - 0.1 \text{ V}$

V_{IHmin} 70% of V_{CC}

V_{ILmax} 30% of V_{CC}

V_{OLmax} ground + 0.1 V

DC noise margin is a measure of how much noise it takes to corrupt a worst-case output voltage into a value that may not be recognized properly by an input. For HC-series CMOS in the LOW state, V_{ILmax} (1.35 V) exceeds V_{OLmax} (0.1 V) by 1.25 V so the LOW-state DC noise margin is 1.25 V. Likewise, there is DC noise margin of 1.25 V in the HIGH state. In general, CMOS outputs have excellent DC noise margins when driving other CMOS inputs. Regardless of the voltage applied to the input of a CMOS inverter, the input consumes very little current, only the leakage current of the two transistors' gates. The maximum amount of current that can flow is also specified by the device manufacturer:

I_{IH} The maximum current that flows into the input in the LOW state.

I_{IL} The maximum current that flows into the input in the HIGH state.

Circuit Behavior with Resistive Loads

As mentioned previously, CMOS gate inputs have very high impedance and consume very little current from the circuits that drive them. There are other devices, however, which require nontrivial amounts of current to operate. When such a device is connected to a CMOS output, we call it a *resistive load* or a *DC load*. Here are some examples of resistive loads:

- Discrete resistors may be included to provide transmission-line termination, discussed in Section 12.4.

- Discrete resistors may not really be present in the circuit, but the load presented by one or more TTL or other non-CMOS inputs may be modeled by a simple resistor network.
- The resistors may be part of or may model a current-consuming device such as a light-emitting diode (LED) or a relay coil.

When the output of a CMOS circuit is connected to a resistive load, the output behavior is not nearly as ideal as we described previously. In either logic state, the CMOS output transistor that is —on|| has a nonzero resistance, and a load connected to the output terminal will cause a voltage drop across this resistance. Thus, in the LOW state, the output voltage may be somewhat higher than 0.1 V, and in the HIGH state it may be lower than 4.4 V. The easiest way to see how this happens is look at a resistive model of the CMOS circuit and load.

Circuit Behavior with Nonideal Inputs

So far, we have assumed that the HIGH and LOW inputs to a CMOS circuit are ideal voltages, very close to the power-supply rails. However, the behavior of a real CMOS inverter circuit depends on the input voltage as well as on the characteristics of the load. If the input voltage is not close to the power-supply rail, then the —on|| transistor may not be fully —on|| and its resistance may increase. Likewise, the —off|| transistor may not be fully —off|| and its resistance may be quite a bit less than one megohm. These two effects combine to move the output voltage away from the power-supply rail.

Fanout

The *fanout* of a logic gate is the number of inputs that the gate can drive without exceeding its worst-case loading specifications. The fanout depends not only on the characteristics of the output, but also on the inputs that it is driving. Fanout must be examined for both possible output states, HIGH and LOW.

Unused Inputs

Sometimes not all of the inputs of a logic gate are used. In a real design problem, you may need an n -input gate but have only an $n+1$ -input gate available. Tying together two inputs of the $n+1$ -input gate gives it the functionality of an n -input gate.

Current Spikes and Decoupling Capacitors

When a CMOS output switches between LOW and HIGH, current flows from VCC to ground through the partially-on p - and n -channel transistors. These currents, often called *current spikes* because of their brief duration, may show up as noise on the power-supply and ground connections in a CMOS circuit, especially when multiple outputs are switched simultaneously. For this reason, systems that use CMOS circuits require *decoupling capacitors* between VCC and ground. These capacitors must be distributed throughout the circuit, at least one within an inch or so of each chip, to supply current during transitions. The large *filtering capacitors* typically found in the power supply itself don't satisfy this requirement, because stray wiring inductance prevents them from supplying the current fast enough, hence the need for a *physically distributed* system of decoupling capacitors.

How to Destroy a CMOS Device

Hit it with a sledge hammer. Or simply walk across a carpet and then touch an input pin with your finger. Because CMOS device inputs have such high impedance, they are subject to damage from *electrostatic discharge (ESD)*. ESD occurs when a buildup of charge on one surface arcs through a dielectric to another surface with the opposite charge. In the case of a CMOS input, the dielectric is the insulation between an input transistor's gate and its source and drain. ESD may damage this insulation, causing a short-circuit between the device's input and output.

The input structures of modern CMOS devices use various measures to reduce their susceptibility to ESD damage, but no device is completely immune. Therefore, to protect CMOS devices from ESD damage during shipment and handling, manufacturers normally package their devices in conductive bags, tubes, or foam. To prevent ESD damage when handling loose CMOS devices, circuit assemblers and technicians usually wear conductive wrist straps that are connected by a coil cord to earth ground; this prevents a static charge from building up on their bodies as they move around the factory or lab.

Once a CMOS device is installed in a system, another possible source of damage is *latch-up*. The physical input structure of just about any CMOS device contains parasitic bipolar transistors between VCC and ground configured as a silicon-controlled rectifier (SCR). In normal operation, this parasitic SCR has no effect on device operation. However, an input voltage that is less than ground or more than VCC can trigger the SCR, creating a virtual short-circuit between VCC and ground. Once the SCR is triggered, the only way to turn it off is to turn off the power supply. Before you have a chance to do this, enough power may be dissipated to destroy the device (i.e., you may see smoke). One possible trigger for latch-up is undershoot on high-speed HIGH-to-LOW signal transitions.

In this situation, the input signal may go several volts below ground for several nanoseconds before settling into the normal LOW range. However, modern CMOS logic circuits are fabricated with special structures that prevent latch-up in this transient case. Latch-up can also occur when CMOS inputs are driven by the outputs of another system or subsystem with a separate power supply. If a HIGH input is applied to a CMOS gate before power is present, the gate may come up in the latched-up state when power is applied. Again, modern CMOS logic circuits are fabricated with special structures that prevent this in most cases. However, if the driving output is capable of sourcing lots of current (e.g., tens of mA), latchup is still possible. One solution to this problem is to apply power before hooking up input cables.

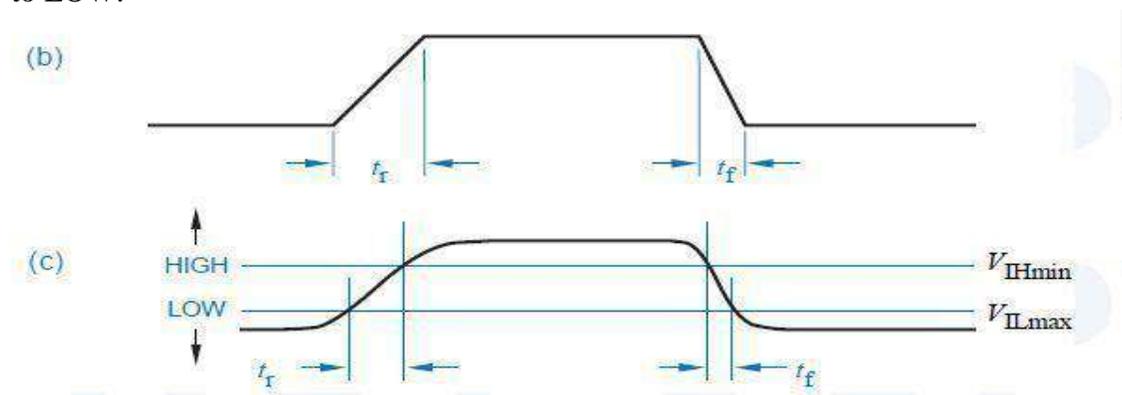
CMOS Dynamic Electrical Behavior

Both the speed and the power consumption of a CMOS device depend to a large extent on AC or dynamic characteristics of the device and its load, that is, what happens when the output changes between states. As part of the internal design of CMOS ASICs, logic designers must carefully examine the effects of output loading and redesign where the load is too high. Even in board-level design, the effects of loading must be considered for clocks, buses, and other signals that

have high fanout or long interconnections. Speed depends on two characteristics, transition time and propagation delay.

Transition Time

The amount of time that the output of a logic circuit takes to change from one state to another is called the *transition time*. Figure shows how we might like outputs to change state—in zero time. However, real outputs cannot change instantaneously, because they need time to charge the stray capacitance of the wires and other components that they drive. A more realistic view of a circuit's output is shown in (b). An output takes a certain time, called the *rise time* (t_r), to change from LOW to HIGH, and a possibly different time, called the *fall time* (t_f), to change from HIGH to LOW.



Even Figure (b) is not quite accurate, because the rate of change of the output voltage does not change instantaneously, either. Instead, the beginning and the end of a transition are smooth, as shown in (c). To avoid difficulties in defining the endpoints, rise and fall times are normally measured at the boundaries of the valid logic levels as indicated in the figure. With the convention in (c), the rise and fall times indicate how long an output voltage takes to pass through the -undefined- region between LOW and HIGH. The initial part of a transition is not included in the rise- or fall-time number. Instead, the initial part of a transition contributes to the -propagation delay- number discussed in the next subsection. The rise and fall times of a CMOS output depend mainly on two factors, the —on- transistor resistance and the load capacitance. A large capacitance increases transition times; since this is undesirable, it is very rare for a logic designer to purposely connect a capacitor to a logic circuit's output. However, *stray capacitance* is present in every circuit; it comes from at least three sources:

1. Output circuits, including a gate's output transistors, internal wiring, and packaging, have some capacitance associated with them, on the order of picofarads (pF) in typical logic families, including CMOS.
2. The wiring that connects an output to other inputs has capacitance, about 1 pF per inch or more, depending on the wiring technology.
3. Input circuits, including transistors, internal wiring, and packaging, have capacitance, from 2 to 15 pF per input in typical logic families.

Power Consumption

The power consumption of a CMOS circuit whose output is not changing is called *static power dissipation* or *quiescent power dissipation*. (The words *consumption* and *dissipation* are used pretty much interchangeably when discussing how much power a device uses.) Most CMOS circuits have very low static power dissipation. This is what makes them so attractive for laptop computers and other low-power applications—when computation pauses, very little power is consumed. A CMOS circuit consumes significant power only during transitions; this is called *dynamic power dissipation*.

One source of dynamic power dissipation is the partial short-circuiting of the CMOS output structure. When the input voltage is not close to one of the power supply rails (0 V or VCC), both the *p*-channel and *n*-channel output transistors may be partially -on, creating a series resistance of 600 or less. In this case, current flows through the transistors from VCC to ground. The amount of power consumed in this way depends on both the value of VCC and the rate at which output transitions occur, according to the formula The following variables are used in the formula:

PT The circuit's internal power dissipation due to output transitions.

VCC The power supply voltage. As all electrical engineers know, power dissipation across a resistive load (the partially-on transistors) is proportional to the *square* of the voltage.

f The *transition frequency* of the output signal. This specifies the number of power-consuming output transitions per second. (But note that frequency is defined as the number of transitions divided by 2.)

CPD The *power dissipation capacitance*. This constant, normally specified by the device manufacturer, completes the formula. CPD turns out to have units of capacitance, but does not represent an actual output capacitance.

Three-State Outputs

Logic outputs have two normal states, LOW and HIGH, corresponding to logic values 0 and 1. However, some outputs have a third electrical state that is not a logic state at all, called the *high impedance*, *Hi-Z*, or *floating state*. In this state, the output behaves as if it isn't even connected to the circuit, except for a small leakage current that may flow into or out of the output pin. Thus, an output can have one of three states—logic 0, logic 1, and Hi-Z. An output with three possible states is called (surprise!) a *three-state output* or, sometimes, a *tri-state output*. Three-state devices have an extra input, usually called -output enable or -output disable, for placing the device's output(s) in the high-impedance state.

A *three-state bus* is created by wiring several three-state outputs together. Control circuitry for the -output enables must ensure that at most one output is enabled (not in its Hi-Z state) at any time. The single enabled device can transmit logic levels (HIGH and LOW) on the bus.

Open-Drain Outputs

The p -channel transistors in CMOS output structures are said to provide *active pull-up*, since they actively pull up the output voltage on a LOW-to-HIGH transition. These transistors are omitted in gates with *open-drain outputs*.

CMOS Logic Families

The first commercially successful CMOS family was *4000-series CMOS*. Although 4000-series circuits offered the benefit of low power dissipation, they were fairly slow and were not easy to interface with the most popular logic family of the time, bipolar TTL. Thus, the 4000 series was supplanted in most applications by the more capable CMOS families discussed in this section. All of the CMOS devices that we discuss have part numbers of the form $-74FAMnn$, where $-FAM$ is an alphabetic family mnemonic and nn is a numeric function designator. Devices in different families with the same value of nn perform the same function. For example, the 74HC30, 74HCT30, 74AC30, 74ACT30, and 74AHC30 are all 8-input NAND gates.

The prefix -74 is simply a number that was used by an early, popular supplier of TTL devices, Texas Instruments. The prefix -54 is used for identical parts that are specified for operation over a wider range of temperature and power-supply voltage, for use in military applications. Such parts are usually

$$R_{min} = 5.0 - 0.0 / I_{R(max)} = 1562.5$$
$$I_{R(leak)} = 4 \quad 5 + 2 \quad 20 = 60 \quad A$$

$$R_{max} = 5.0 - 2.4 / I_{R(leak)} = 43.3$$

fabricated in the same way as their 74-series counterparts, except that they are tested, screened, and marked differently, a lot of extra paperwork is generated, and a higher price is charged, of course.

HC and HCT

The first two 74-series CMOS families are *HC (High-speed CMOS)* and *HCT (High-speed CMOS, TTL compatible)*. Compared with the original 4000 family, HC and HCT both have higher speed and better current sinking and sourcing capability. The HCT family uses a power supply voltage V_{CC} of 5 V and can be intermixed with TTL devices, which also use a 5-V supply. The HC family is optimized for use in systems that use CMOS logic exclusively, and can use any power supply voltage between 2 and 6 V. A higher voltage is used for higher speed, and a lower voltage for lower power dissipation. Lowering the supply voltage is especially effective, since most CMOS power dissipation is proportional to the square of the voltage (CV^2f power). Even when used with a 5-V supply, HC devices are not quite compatible with TTL.

VHC and VHCT

Several new CMOS families were introduced in the 1980s and the 1990s. Two of the most recent and probably the most versatile are *VHC (Very High-Speed CMOS)* and *VHCT (Very High-Speed CMOS, TTL compatible)*. These families are about twice as fast as HC/HCT while maintaining backwards compatibility with their predecessors. Like HC and HCT, the VHC and VHCT

families differ from each other only in the input levels that they recognize; their output characteristics are the same. Also like HC/HCT, VHC/VHCT outputs have *symmetric output drive*. That is, an output can sink or source equal amounts of current; the output is just as $-strong$ in both states. Other logic families, including the FCT and TTL families introduced

later, have *asymmetric output drive*; they can sink much more current in the LOW state than they can source in the HIGH state.

HC, HCT, VHC, and VHCT Electrical Characteristics

Electrical characteristics of the HC, HCT, VHC, and VHCT families are summarized in this subsection. The specifications assume that the devices are used with a nominal 5-V power supply, although (derated) operation is possible with any supply voltage in the range 2–5.5 V (up to 6 V for HC/HCT). Commercial (74-series) parts are intended to be operated at temperatures between 0 C and 70 C, while military (54-series) parts are characterized for operation between 55 C and 125 C.

FCT and FCT-T

In the early 1990s, yet another CMOS family was launched. The key benefit of the *FCT (Fast CMOS, TTL compatible)* family was its ability to meet or exceed the speed and the output drive capability of the best TTL families while reducing power consumption and maintaining full compatibility with TTL.

UNIT II
Combinational Logic Design

The VHDL Hardware Description Language

VHDL stands for **VHSIC** (Very High Speed Integrated Circuits) **H**ardware **D**escription **L**anguage. The VHDL Hardware Design Language In the mid-1980s, the U.S. Department of Defense (DoD) and the IEEE sponsored the development of a highly capable hardware-description language called *VHDL*.

It has become now one of industry's standard languages used to describe digital systems. The other widely used hardware description language is Verilog . Both are powerful languages that allow you to describe and simulate complex digital systems. A third HDL language is ABEL (Advanced Boolean Equation Language) which was specifically designed for Programmable Logic Devices (PLD). ABEL is less powerful than the other two languages and is less popular in industry. This tutorial deals with VHDL, as described by the IEEE standard.

VHDL has the following features:

- Designs may be decomposed hierarchically.
- Each design element has both a well-defined interface (for connecting it to other elements) and a precise behavioral specification (for simulating it).
- Behavioral specifications can use either an algorithm or an actual hardware structure to define an element's operation
- Concurrency, timing, and clocking can all be modeled. VHDL handles asynchronous as well as synchronous sequential-circuit structures.
- The logical operation and timing behavior of a design can be simulated.

Design Flow:

There are several steps in a VHDL-based design process, often called the design flow.

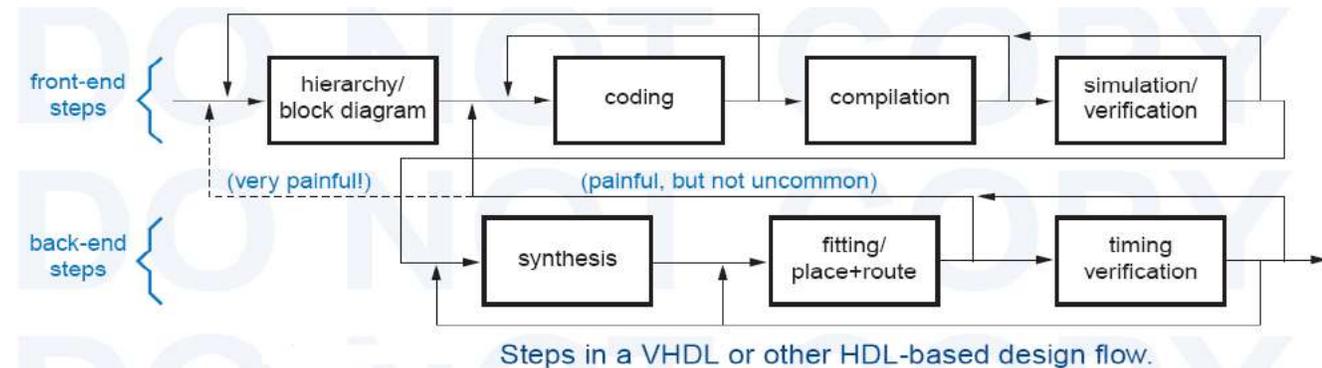


Fig.2.1 The “front end” begins with figuring out the basic approach and building blocks at the block diagram level. Large logic designs, like software programs, are usually hierarchical, and VHDL gives us a good framework for defining modules and their interfaces, and filling in the details.

Coding:

The actual writing of VHDL code for modules, their interfaces, and their internal details. VHDL is a text-based language, in principle you can use any text editor for this part.

A specialized VHDL text editor include features like automatic highlighting of VHDL keywords, automatic indenting, built-in templates for frequently used program structures, and built-in syntax checking and one-click access to the compiler.

Compiler: A VHDL compiler analyzes your code for syntax errors and also checks your code for compatibility with other modules on which it relies.

It also creates the internal information that is needed for a simulator to process your design later. As in other programming endeavors, you probably shouldn't wait until the very end of coding to compile all of your code.

Doing a piece at a time can prevent you from proliferating syntax errors, inconsistent names, and so on,

Simulator:

A VHDL simulator allows you to define and apply inputs to your design, and to observe its outputs, without ever having to build the physical circuit.

In small projects we would probably generate inputs and observe outputs manually. But for larger projects, VHDL gives you the ability to create "test benches" that automatically apply inputs and compare them with expected outputs.

There are two types of verification

- 1) Functional Verification
- 2) Timing Verification

In functional verification, we study the circuit's logical operation independent of timing considerations; gate delays and other timing parameters are considered to be zero.

In timing verification, we study the circuit's operation including estimated delays, and we verify that the setup, hold, and other timing requirements for sequential devices like flip-flops are met. Functional verification before starting the back-end steps.

In back-end there are three basic steps

- 1) Synthesis
- 2) Fitting/Place + Route
- 3) Timing Verification

Synthesis:

The synthesis, converting the VHDL description into a set of primitives or components that can be assembled in the target technology.

For example, with PLDs or CPLDs, the synthesis tool may generate two-level sum-of-products equations.

In the fitting step, a fitting tool or fitter maps the synthesized primitives or components onto available device resources.

Place & Route:

For a PLD or CPLD, this may mean assigning equations to available AND-OR elements. For an ASIC, it may mean laying down individual gates in a pattern and finding ways to connect them within the physical constraints of the ASIC die.

The "final" step is timing verification of the fitted circuit. actual circuit delays due to wire lengths, electrical loading, and other factors can be calculated with reasonable precision.

VHDL Program Structure

A VHDL *entity* is simply a declaration of a module's inputs and outputs, while a VHDL *architecture* is a detailed description of the module's internal structure or behavior.

In the text file of a VHDL program, the entity declaration and architecture definition are separated shown in fig.2.2

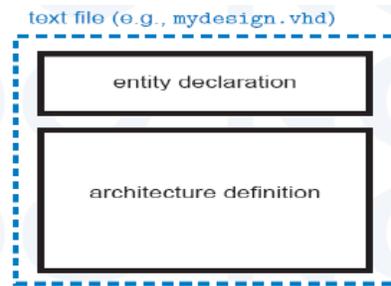


Figure 2.2 VHDL program file structure.

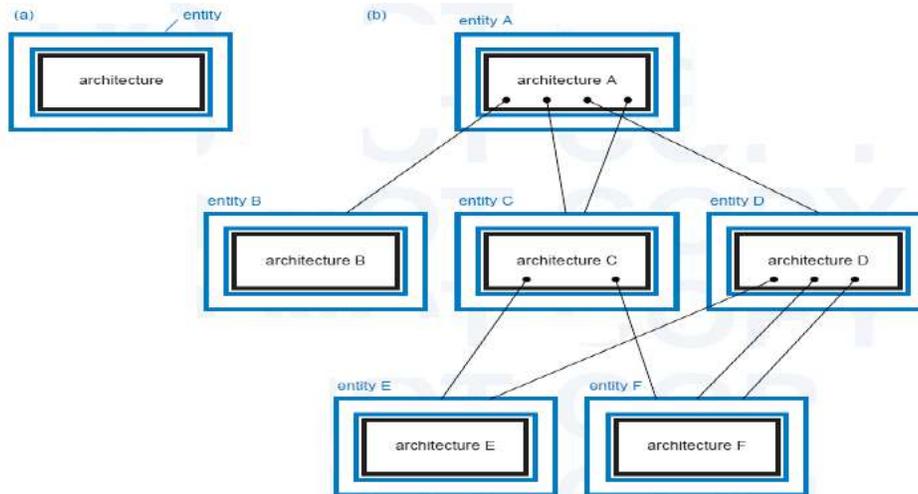


Figure 2.3 (a) illustrates the concept. Many designers like to think of a VHDL entity declaration as a “wrapper” for the architecture, hiding the details of what’s inside while providing the “hooks” for other modules to use it.

This forms the basis for hierarchical system design—the architecture of a top-level entity may use (or “instantiate”) other entities, while hiding the architectural details of lower-level entities from the higher-level ones. As shown in (b), a higher-level architecture may use a lower-level entity multiple times, and multiple top-level architectures may use the same lower-level one. In the figure, architectures B, E and F stand alone; they do not use any other entities.

Like other high-level programming languages, VHDL generally ignores spaces and line breaks, and these may be provided as desired for readability.

Comments begin with two hyphens (--) and end at the end of a line.

Syntax of a VHDL entity declaration.

```
entity entity-name is
    port (signal-names : mode signal-type;
          signal-names : mode signal-type;
          ...
          signal-names : mode signal-type);
end entity-name;
```

An entity declaration has shown in the general syntax. The entity declaration is to define its external interface signals or *ports* in its *port declaration* part. In addition to the keywords *entity*, *is*, *port*, and *end*,

An entity declaration has the following elements:

entity-name A user-selected identifier to name the entity.

signal-names A comma-separated list of one or more user-selected identifiers to name external-interface signals.

mode One of four reserved words, specifying the signal direction:

in The signal is an input to the entity.

out The signal is an output of the entity. Note that the value of such a signal cannot be “read” inside the entity’s architecture, only by other entities that use it.

buffer The signal is an output of the entity, and its value can also be read inside the entity’s architecture.

inout The signal can be used as an input or an output of the entity. This mode is typically used for three-state input/output pins on PLDs.

signal-type A built-in or user-defined signal type.

Note that there is no semicolon after the final **signal-type**; swapping the closing parenthesis with the semicolon after it is a common syntax error for beginning VHDL programmers.

- *type*: a built-in or user-defined signal type. Examples of types are bit, bit_vector, Boolean, character, std_logic, and std_ulogic.
 - *bit* – can have the value 0 and 1
 - *bit_vector* – is a vector of bit values (e.g. bit_vector (0 to 7))
 - *std_logic*, *std_ulogic*, *std_logic_vector*, *std_ulogic_vector*: can have 9 values to indicate the value and strength of a signal. Std_ulogic and std_logic are preferred over the bit or bit_vector types.
 - *boolean* – can have the value TRUE and FALSE.
 - *integer* – can have a range of integer values.
 - *real* – can have a range of real values.
 - *character* – any printing character.
 - *time* – to indicate time.

Entity Examples:

An example of the entity declaration of a Four-to-one multiplexer of which each input is an 8-bit word.

entity mux4_to_1 **is**

```
port (I0,I1,I2,I3: in std_logic_vector(7 downto 0);
      SEL: in std_logic_vector (1 downto 0);
      OUT1: out std_logic_vector(7 downto 0));
end mux4_to_1;
```

An example of the entity declaration of a D flip-flop with set and reset inputs is

entity dff_sr **is**

```
port (D,CLK,S,R: in std_logic;
      Q,Qnot: out std_logic);
end dff_sr;
```

Architecture body

The architecture body specifies how the circuit operates and how it is implemented. An entity or circuit can be specified

in a variety of ways, such as behavioral, structural (interconnected components), or a combination of the above.

The *entity-name* in this definition must be the same as the one given previously in the entity declaration.

The *architecture-name* is a user-selected identifier, usually related to the entity name; it can be the same as the entity name if desired.

The architecture body looks as follows,

```
architecture architecture_name of NAME_OF_ENTITY is  
  -- Declarations  
    -- components declarations  
    -- signal declarations  
    -- constant declarations  
    -- function declarations  
    -- procedure declarations  
    -- type declarations  
begin  
  -- Statements  
end architecture_name;
```

Behavioral model

The architecture body for the example of Figure 2, described at the behavioral level, is given below,

```
architecture behavioral of BUZZER is  
begin  
  WARNING <= (not DOOR and IGNITION) or (not SBELT and IGNITION);  
end behavioral;
```

The header line of the architecture body defines the architecture name, e.g. behavioral, and associates it with the entity, BUZZER. The architecture name can be any legal identifier. The main body of the architecture starts with the keyword **begin** and gives the Boolean expression of the function. We will see later that a behavioral model can be described in several other ways. The “<= ” symbol represents an assignment [operator](#) and assigns the value of the expression on the right to the signal on the left. The architecture body ends with an **end** keyword followed by the architecture name.

A few other examples follow. The behavioral description of a two-input AND gate is shown below.

Architecture *architecture-name* of *entity-name* is

```
type declarations  
signal declarations  
constant declarations  
function definitions  
procedure definitions  
component declarations  
begin
```

concurrent-statement

...

concurrent-statement

End *architecture-name*;

The *signal declaration* gives the same information about a signal as in a port declaration, except that no mode is specified:

signal signal-names : signal-type;

VHDL *variables* are similar to signals, except that they usually don't have physical significance in a circuit.

The syntax of a *variable declaration* is just like that of a signal declaration, except that the variable keyword is used.

Variable variable-names : variable-type;

Types and Constants

All signals, variables, and constants in a VHDL program must have an associated "type". The *type* specifies the set or range of values that the object can take on,

VHDL predefined types

| | | |
|------------|-----------|----------------|
| bit | character | severity_level |
| bit_vector | integer | string |
| Boolean | real | time |

Predefined operators for VHDL's integer and boolean types.

| integer Operators | boolean Operators |
|----------------------|---------------------|
| + addition | and AND |
| - subtraction | or OR |
| * multiplication | nand NAND |
| / division | nor NOR |
| Mod modulo division | xor Exclusive OR |
| rem modulo remainder | xnor Exclusive NOR |
| abs absolute value | not complementation |
| ** exponentiation | |

Type integer is defined as the range of integers including at least the range $-2,147,483,647$ through $+2,147,483,647$ ($-2^{31}+1$ through $+2^{31}-1$).

Type boolean has two values, true and false. The character type contains all of the characters in the ISO 8-bit character set; the first 128 characters are the ASCII characters.

The most commonly used types in typical VHDL programs are *userdefined types*, and the most common of these are *enumerated types*, which are defined by listing their values.

value-list is a comma-separated list (enumeration) of all possible values of the type. The values may be user-defined identifiers or characters.

type type-name is (value-list);

subtype subtype-name is type-name start to end;

subtype subtype-name is type-name start downto end;

constant constant-name: type-name := value;

```
type traffic_light_state is (reset, stop, wait, go);
```

A standard user-defined logic type `std_logic`,

Definition of VHDL `std_logic` type

```
type STD_ULOGIC is ( 'U', -- Uninitialized
                    'X', -- Forcing Unknown
                    '0', -- Forcing 0
                    '1', -- Forcing 1
                    'Z', -- High Impedance
                    'W', -- Weak Unknown
                    'L', -- Weak 0
                    'H', -- Weak 1
                    '-' -- Don't care
                    );
subtype STD_LOGIC is resolved STD_ULOGIC;
type traffic_light_state is (reset, stop, wait, go);
```

VHDL also allows users to create *subtypes* of a type. The values in the subtype must be a contiguous range of values of the base type, from *start* to *end*.

```
subtype twoval_logic is std_logic range '0' to '1';
subtype fourval_logic is std_logic range 'X' to 'Z';
subtype negint is integer range -2147483647 to 1;
subtype bitnum is integer range 31 downto 0;
```

Syntax of VHDL array declarations

```
type type-name is array (start to end) of element-type;
type type-name is array (start downto end) of element-type;
type type-name is array (range-type) of element-type;
type type-name is array (range-type range start to end) of element-type;
type type-name is array (range-type range start downto end) of element-type;
```

```
constant BUS_SIZE: integer := 32; -- width of component
constant MSB: integer := BUS_SIZE-1; -- bit number of MSB
constant Z: character := 'Z'; -- synonym for Hi-Z value
```

VHDL defines an *array* as an ordered set of elements of the same type, where each element is selected by an *array index*.

Examples of VHDL array declarations.

```
type monthly_count is array (1 to 12) of integer;
type byte is array (7 downto 0) of STD_LOGIC;

constant WORD_LEN: integer := 32;
type word is array (WORD_LEN-1 downto 0) of STD_LOGIC;

constant NUM_REGS: integer := 8;
type reg_file is array (1 to NUM_REGS) of word;

type statecount is array (traffic_light_state) of integer;
```

A VHDL *string* is a sequence of ISO characters enclosed in double quotes, such as "Hi there". A string is just an array of

characters.

```
B := "11111111";
```

```
W := "11111110111111011111101111110";
```

Libraries and Packages:

A VHDL *library* is a place where the VHDL compiler stores information about a particular design project, including intermediate files that are used in the analysis, simulation, and synthesis of the design.

The location of the library within a host computer's file system is implementation dependent. For a given VHDL design, the compiler automatically creates and uses a library named "work".

A complete VHDL design usually has multiple files, each containing different design units including entities and architectures.

When the VHDL compiler analyzes each file in the design, it places the results in the "work" library, and it also searches this library for needed definitions, such as other entities.

Not all of the information needed in a design may be in the "work" library.

Each project has its own "work" library (typically a subdirectory within that project's overall directory), but must also refer to a common library containing the shared definitions.

Even small projects may use a standard library such as the one containing IEEE standard definitions.

The designer can specify the name of such a library using a library *clause* at the beginning of the design file. For example, we can specify the IEEE library:

```
library ieee;
```

The clause "library work;" is included implicitly at the beginning of every VHDL design file.

Specifying a library name in a design gives it access to any previously analyzed entities and architectures stored in the library, but it does not give access to type definitions and the like. This is the function of "packages" and "use clauses," described next.

Packages:

A VHDL *package* is a file containing definitions of objects that can be used in other programs. The kind of objects that can be put into a package include signal, type, constant, function, procedure, and component declarations.

Signals that are defined in a package are "global" signals, available to any VHDL entity that uses the package.

Types and constants defined in a package are known in any file that uses the package. Likewise, functions and procedures defined in a package can be called in files that use the package, and components (described in the next subsection) can be "instantiated" in architectures that use the package.

A design can "use" a package by including a use *clause* at the beginning of the design file. For example, to use all of the definitions in the IEEE standard

1164 package, we would write

```
use ieee.std_logic_1164.all;
```

Here, "ieee" is the name of a library which has been previously given in a library clause. Within this library, the

file named “std_logic_1164” contains the desired definitions. The suffix “all” tells the compiler to use all of the definitions in this file. Instead of “all”, you can write the name of a particular object to use just its definition, for example,

use ieee.std_logic_1164.std_ulogic

This clause would make available just the definition of the std_ulogic type, without all of the related types and functions. However, multiple “use” clauses can be written to use additional definitions.

Syntax of a VHDL package definition:

```
package package-name is
    type declarations
    signal declarations
    constant declarations
    component declarations
    function declarations
    procedure declarations
end package-name;
package body package-name is
    type declarations
    constant declarations
    function definitions
    procedure definitions
end package-name;
```

Structural Design Elements

In VHDL, each *concurrent statement* executes simultaneously with the other concurrent statements in the same architecture body.

The most basic of VHDL’s concurrent statements is the component *statement*, whose basic syntax is shown in Table 3.1. Here, *component-name* is the name of a previously defined entity that is to be used, or *instantiated*, within the current architecture body.

Syntax of a VHDL component statement

```
label: component-name port map(signal1, signal2, ..., signaln);
label: component-name port map(port1=>signal1, port2=>signal2, ..., portn=>signaln);
```

The port map keywords introduce a list that associates ports of the named entity with signals in the current architecture. The list may be written in either of two different styles. The first is a positional style; as in conventional programming languages, the signals in the list are associated with the entity’s ports in the same order that they appear in the entity’s definition.

The second is an explicit style; each of the entity’s ports is connected to a signal using the “=>” operator, and these associations may be listed in any order.

Syntax of a VHDL component declaration.

```
component component-name
port (signal-names : mode signal-type;
```

```

        signal-names : mode signal-type;
        ...
        signal-names : mode signal-type);
end component;

```

A component must be declared in a *component declaration* in the architecture's definition. In above syntax, a component declaration is essentially the same as the port declaration part of the corresponding entity declaration—it lists the name, mode, and type of each of its ports.

The components used in an architecture may be ones that were previously defined as part of a design, or they may be part of a library.

A VHDL architecture that uses components is often called a *structural description* or *structural design*, because it defines the precise interconnection structure of signals and entities that realize the entity.

Example Structural VHDL program for a prime-number detector.

```

library IEEE;
use IEEE.std_logic_1164.all;
entity prime is
    port ( N: in STD_LOGIC_VECTOR (3 downto 0);
          F: out STD_LOGIC );
end prime;
architecture prime1_arch of prime is
    signal N3_L, N2_L, N1_L: STD_LOGIC;
    signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
    component INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;
    component AND2 port (I0,I1: in STD_LOGIC; O: out STD_LOGIC); end component;
    component AND3 port (I0,I1,I2: in STD_LOGIC; O: out STD_LOGIC); end component;
    component OR4 port (I0,I1,I2,I3: in STD_LOGIC; O: out STD_LOGIC); end component;
begin
    U1: INV port map (N(3), N3_L);
    U2: INV port map (N(2), N2_L);
    U3: INV port map (N(1), N1_L);
    U4: AND2 port map (N3_L, N(0), N3L_N0);
    U5: AND3 port map (N3_L, N2_L, N(1), N3L_N2L_N1);
    U6: AND3 port map (N2_L, N(1), N(0), N2L_N1_N0);
    U7: AND3 port map (N(2), N1_L, N(0), N2_N1L_N0);
    U8: OR4 port map (N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0, F);
end prime1_arch;

    label: for identifier in range generate
        concurrent-statement
    end generate;

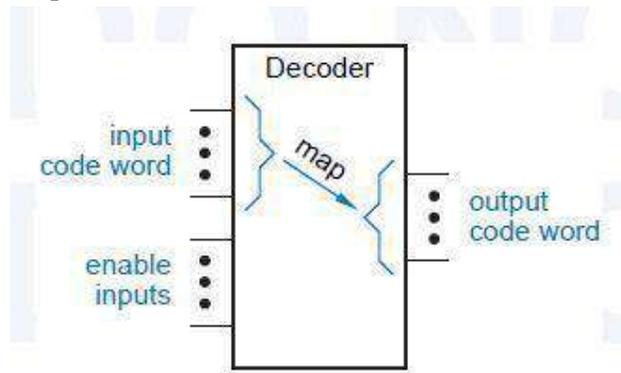
```

UNIT III
Combinational Logic Design

UNIT 3 (COMBINATIONAL LOGIC DESIGN-I)

Decoders

A *decoder* is a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs, where the input and output codes are different. The input code generally has fewer bits than the output code, and there is a one-to-one mapping from input code words into output code words. In a *one-to-one mapping*, each input code word produces a different output code word.



The general structure of a decoder circuit is shown in Figure 1. The enable inputs, if present, must be asserted for the decoder to perform its normal mapping function. Otherwise, the decoder maps all input code words into a single, -disabled, output code word.

The most commonly used output code is a 1-out-of- m code, which contains m bits, where one bit is asserted at any time. Thus, in a 1-out-of-4 code with active-high outputs, the code words are 0001, 0010, 0100, and 1000. With active-low outputs, the code words are 1110, 1101, 1011, and 0111.

Binary Decoders

The most common decoder circuit is an n -to- 2^n decoder or *binary decoder*. Such a decoder has an n -bit binary input code and a 1-out-of- 2^n output code. A binary decoder is used when you need to activate exactly one of 2^n outputs based on an n -bit input value.

| Inputs | | | Outputs | | | |
|--------|----|----|---------|----|----|----|
| EN | I1 | I0 | Y3 | Y2 | Y1 | Y0 |
| 0 | x | x | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |

Table 1: 2 to 4 decoder

Table 1 is the truth table of a 2-to-4 decoder. The input code word I_1, I_0 represents an integer in the range 0–3. The output code word Y_3, Y_2, Y_1, Y_0 has Y_i equal to 1 if and only if the input code word is the binary representation of i and the *enable input* EN is 1. If EN is 0, then all of the outputs are 0. A gate-level circuit for the 2-to-4 decoder is shown in Figure 2. Each AND gate *decodes* one combination of the input code word I_1, I_0 .

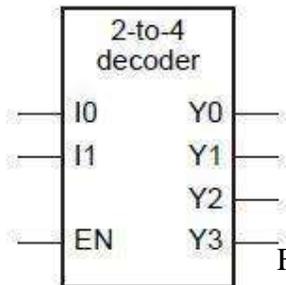


Fig 3: 2 to 4 decoder logic symbol

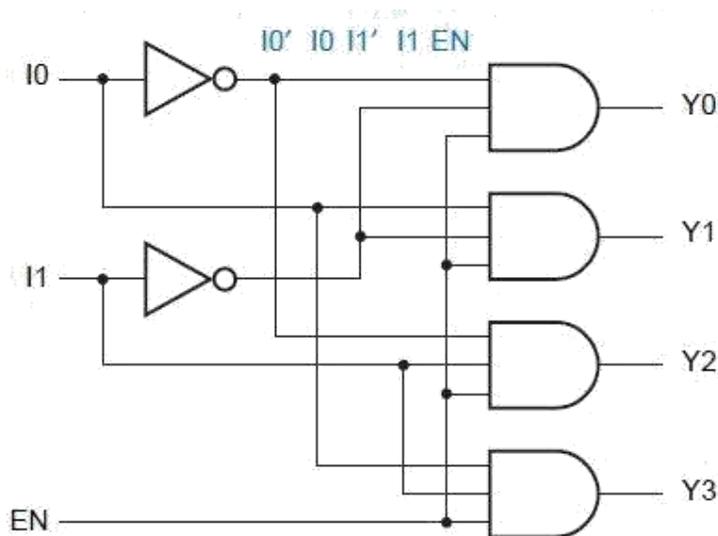


Fig 4: logic diagram of 2 to 4 decoder

The 74x139 Dual 2-to-4 Decoder

Two independent and identical 2-to-4 decoders are contained in a single MSI part, the 74x139. The gate-level circuit diagram for this IC is shown in Figure 5.

1. The outputs and the enable input of the '139 are active-low.
2. Most MSI decoders were originally designed with active-low outputs, since TTL inverting gates are generally faster than non inverting ones.
3. '139 has extra inverters on its select inputs. Without these inverters, each select input would present three AC or DC loads instead of one, consuming much more of the fanout budget of the device that drives it.

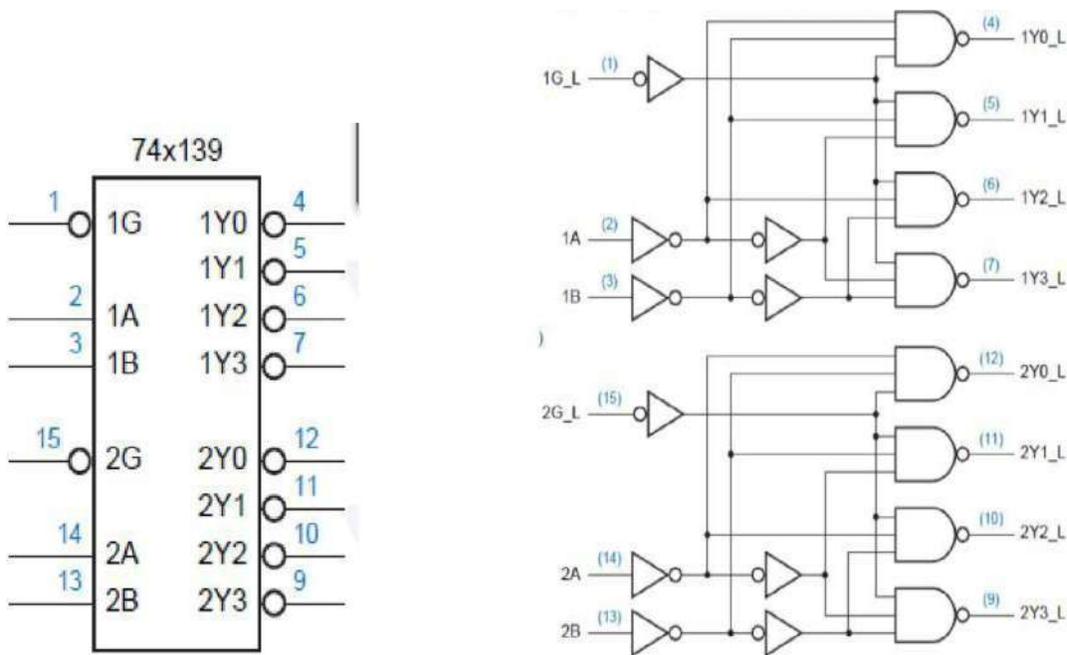


Figure 6 The 74x139 dual 2-to-4 decoder: (a) traditional logic symbol (b) logic diagram, including pin numbers for a standard 16-pin dual in-line package;

In this case, the assignment of the generic function to one half or the other of a particular '139 package can be deferred until the schematic is completed Table 5-6 is the truth table for a 74x139-type decoder.

The 74x138 3-to-8 Decoder

The 74x138 is a commercially available MSI 3-to-8 decoder whose gate-level circuit diagram and symbol are shown in Figure 7; its truth table is given in [Table 5-7](#). Like the 74x139, the 74x138 has active-low outputs, and it has three enable inputs (G1, /G2A, /G2B), all of which must be asserted for the selected output to be asserted.

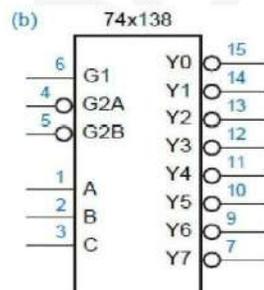
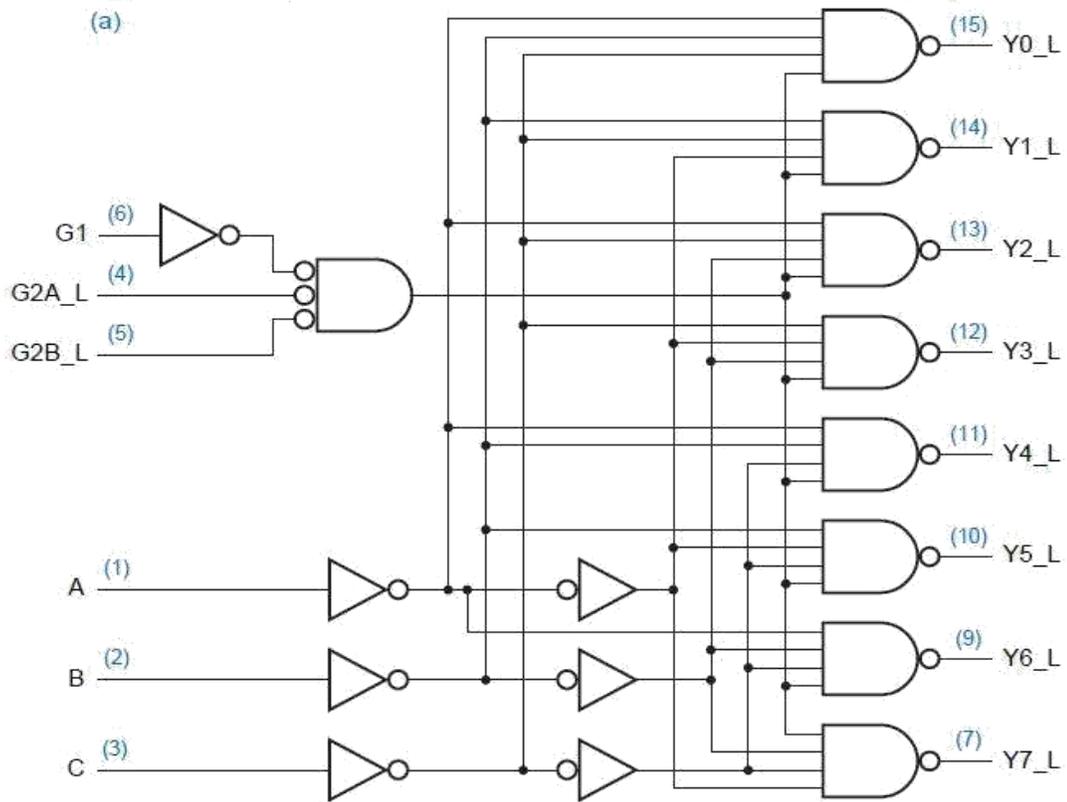


Figure 7: logic symbol of 74X138

| Inputs | | | | | | Outputs | | | | | | | |
|--------|-------|-------|---|---|---|---------|------|------|------|------|------|------|------|
| G1 | G2A_L | G2B_L | C | B | A | Y7_L | Y6_L | Y5_L | Y4_L | Y3_L | Y2_L | Y1_L | Y0_L |
| 0 | x | x | x | x | x | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| x | 1 | x | x | x | x | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| x | x | 1 | x | x | x | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Fig: Truth Table of 74X138 3 to 8 decoder



The logic function of the '138 is straightforward—an output is asserted if and only if the decoder is enabled and the output is selected. Thus, we can easily write logic equations for an internal output signal such as Y5 in terms of the internal input signals:

$$Y5 = \underbrace{G1 \cdot G2A \cdot G2B}_{\text{enable}} \cdot \underbrace{C \cdot B' \cdot A}_{\text{select}}$$

However, because of the inversion bubbles, we have the following relations between internal and external signals:

$$\begin{aligned} G2A &= G2A_L' \\ G2B &= G2B_L' \\ Y5 &= Y5_L' \end{aligned}$$

Therefore, if we're interested, we can write the following equation for the external output signal $Y5_L$ in terms of external input signals:

$$\begin{aligned} Y5_L = Y5' &= (G1 \cdot G2A_L' \cdot G2B_L' \cdot C \cdot B' \cdot A)' \\ &= G1' + G2A_L + G2B_L + C' + B + A' \end{aligned}$$

On the surface, this equation doesn't resemble what you might expect for a decoder, since it is a logical sum rather than a product. However, if you practice bubble-to-bubble logic design, you don't have to worry about this; you just give the output signal an active-low name and remember that it's active low when you connect it to other inputs.

Cascading Binary Decoders

Multiple binary decoders can be used to decode larger code words. Figure 5-38 shows how two 3-to-8 decoders can be combined to make a 4-to-16 decoder. The availability of both active-high and active-low enable inputs on the 74x138 makes it possible to enable one or the other directly based on the state of the most significant input bit. The top decoder (U1) is enabled when $N3$ is 0, and the bottom one (U2) is enabled when $N3$ is 1.

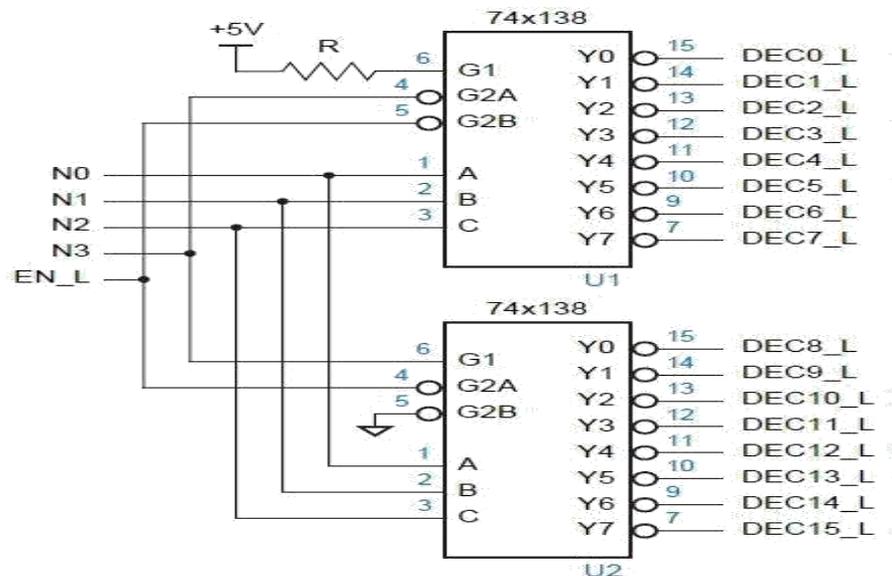
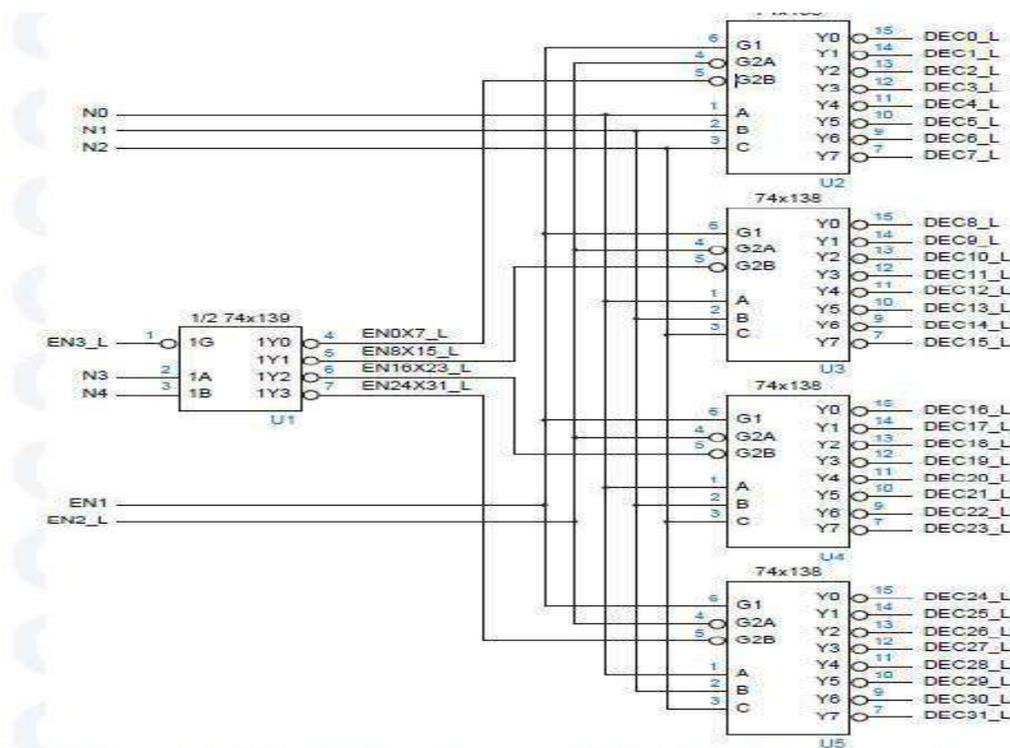


Figure 5-38 Design of a 4-to-16 decoder using 74x138s.



Seven-Segment Decoders

Look at your wrist and you'll probably see a *seven-segment display*. This type of display, which normally uses light-emitting diodes (LEDs) or liquid-crystal display (LCD) elements, is used in watches, calculators, and instruments to display decimal data. A digit is displayed by illuminating a subset of the seven line segments shown in Figure 5-43(a).

A *seven-segment decoder* has 4-bit BCD as its input code and the seven-segment code.

| Inputs | | | | | Outputs | | | | | | |
|--------|---|---|---|---|---------|---|---|---|---|---|---|
| BI_L | D | C | B | A | a | b | c | d | e | f | g |
| 0 | x | x | x | x | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

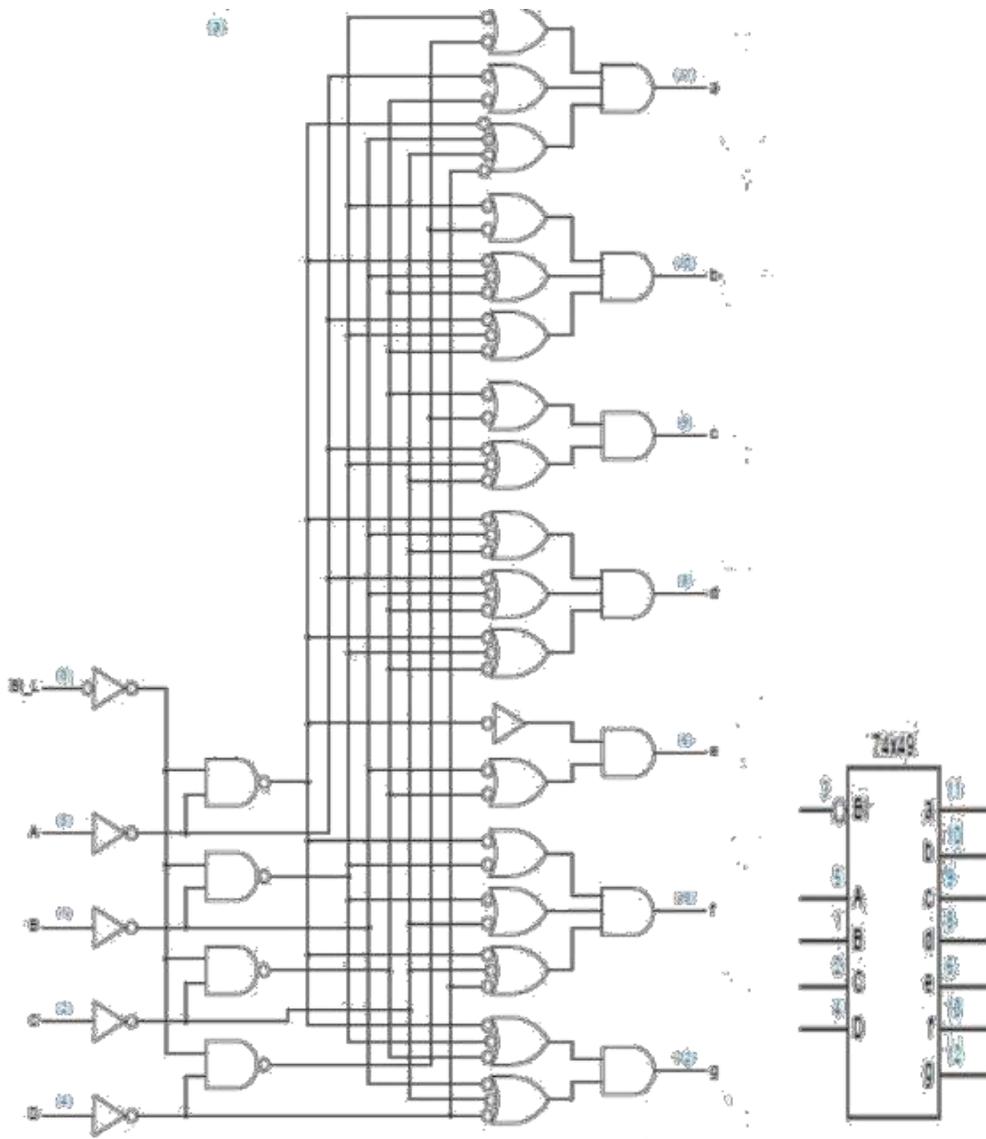


Figure The 74x49 seven-segment decoder: (a) logic diagram, including pin numbers; (b) traditional logic symbol. NOT COPY

5.5 Encoders

A decoder's output code normally has more bits than its input code. If the device's output code has *fewer* bits than the input code, the device is usually called an *encoder*.

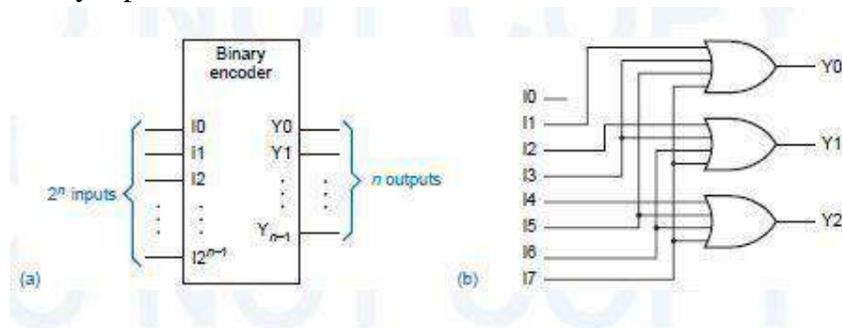
Probably the simplest encoder to build is a $2n$ -to- n or *binary encoder*. As shown in Figure 5-45(a), it has just the opposite function as a binary *decoder*— its input code is the 1-out-of- $2n$ code and its output code is n -bit binary. The equations for an 8-to-3 encoder with inputs I0–I7 and outputs Y0–Y2 are given below:

$$Y0 = I1 + I3 + I5 + I7$$

$$Y1 = I2 + I3 + I6 + I7$$

$$Y2 = I4 + I5 + I6 + I7$$

The corresponding logic circuit is shown in (b). In general, a 2^n -to- n encoder can be built from $n \cdot 2^{n-1}$ 1-input OR gates. Bit i of the input code is connected to OR gate j if bit j in the binary representation of i is 1.



5.5.1 Priority Encoders

The 1-out-of- 2^n coded outputs of an n -bit binary decoder are generally used to control a set of 2^n devices, where at most one device is supposed to be active at any time. Conversely, consider a system with 2^n inputs, each of which indicates a request for service. This structure is often found in microprocessor input/output subsystems, where the inputs might be interrupt requests.

In this situation, it may seem natural to use a binary encoder. to observe the inputs and indicate which one is requesting service at any time. However, this encoder works properly only if the inputs are guaranteed to be asserted at most one at a time. If multiple requests can be made simultaneously, the encoder gives undesirable results. For example, suppose that inputs I_2 and I_4 of the 8-to-3 encoder are both 1; then the output is 110, the binary encoding of 6.

$$Y_0 = I_1 + I_3 + I_5 + I_7$$

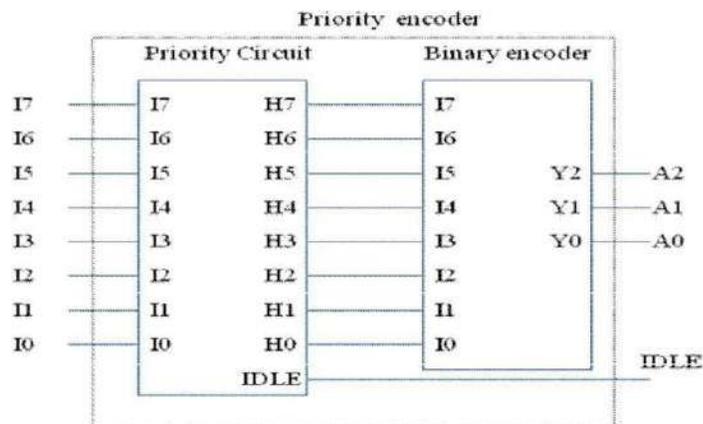
$$Y_1 = I_2 + I_3 + I_6 + I_7$$

$$Y_2 = I_4 + I_5 + I_6 + I_7$$

Either 2 or 4, not 6, would be a useful output in the preceding example, but how can the encoding device decide which? The solution is to assign *priority* to the input lines, so that when multiple requests are asserted, the encoding device produces the number of the highest-priority requestor. Such a device is called a *priority encoder*. Input I_7 has the highest priority. Outputs A_2 – A_0 contain the number of the highest-priority asserted input, if any. The IDLE output is asserted if no inputs are asserted.

In order to write logic equations for the priority encoder's outputs, we first define eight intermediate variables H_0 – H_7 , such that H_i is 1 if and only if I_i is the highest priority 1 input: Using these signals, the equations for the A_2 – A_0 outputs are similar to the ones for a simple binary encoder:

| | | | | | | | | |
|--|---|-----------|---|-----------|---|-----------|---|-----------|
| $H7=I7$ (Highest Priority) | | | | | | | | |
| $H6=I6.I7'$ | | | | | | | | |
| $H5=I5.I6'.I7'$ | | | | | | | | |
| $H4=I4.I5'.I6'.I7'$ | | | | | | | | |
| $H3=I3.I4'.I5'.I6'.I7'$ | | | | | | | | |
| $H2=I2.I3'.I4'.I5'.I6'.I7'$ | | | | | | | | |
| $H1=I1.I2'.I3'.I4'.I5'.I6'.I7'$ | | | | | | | | |
| $H0=I0.I1'.I2'.I3'.I4'.I5'.I6'.I7'$ | | | | | | | | |
| $IDLE=I0'.I1'.I2'.I3'.I4'.I5'.I6'.I7'$ | | | | | | | | |
| - | | | | | | | | Encoder |
| $A0=Y0$ | = | H1 | + | H3 | + | H5 | + | H7 |
| $A1=Y1$ | = | H2 | + | H3 | + | H6 | + | H7 |
| $A2=Y2 = H4 + H5 + H6 + H7$ | | | | | | | | |



8-input priority encoder

- I7 has the highest priority, I0 least
- A2-A0 contain the number of the highest-priority asserted input if any.
- IDLE is asserted if no inputs are asserted.

Figure 5-50 shows how four 74x148s can be connected in this way to accept 32 request inputs and produce a 5-bit output, RA4–RA0, indicating the highest-priority requestor. Since the A2–A0 outputs of at most one '148 will be enabled at any time, the outputs of the individual '148s can be ORed to produce RA2–RA0. Likewise, the individual GS_L outputs can be combined in a 4-to-2 encoder to produce RA4 and RA3. The RGS output is asserted if any GS output is asserted.

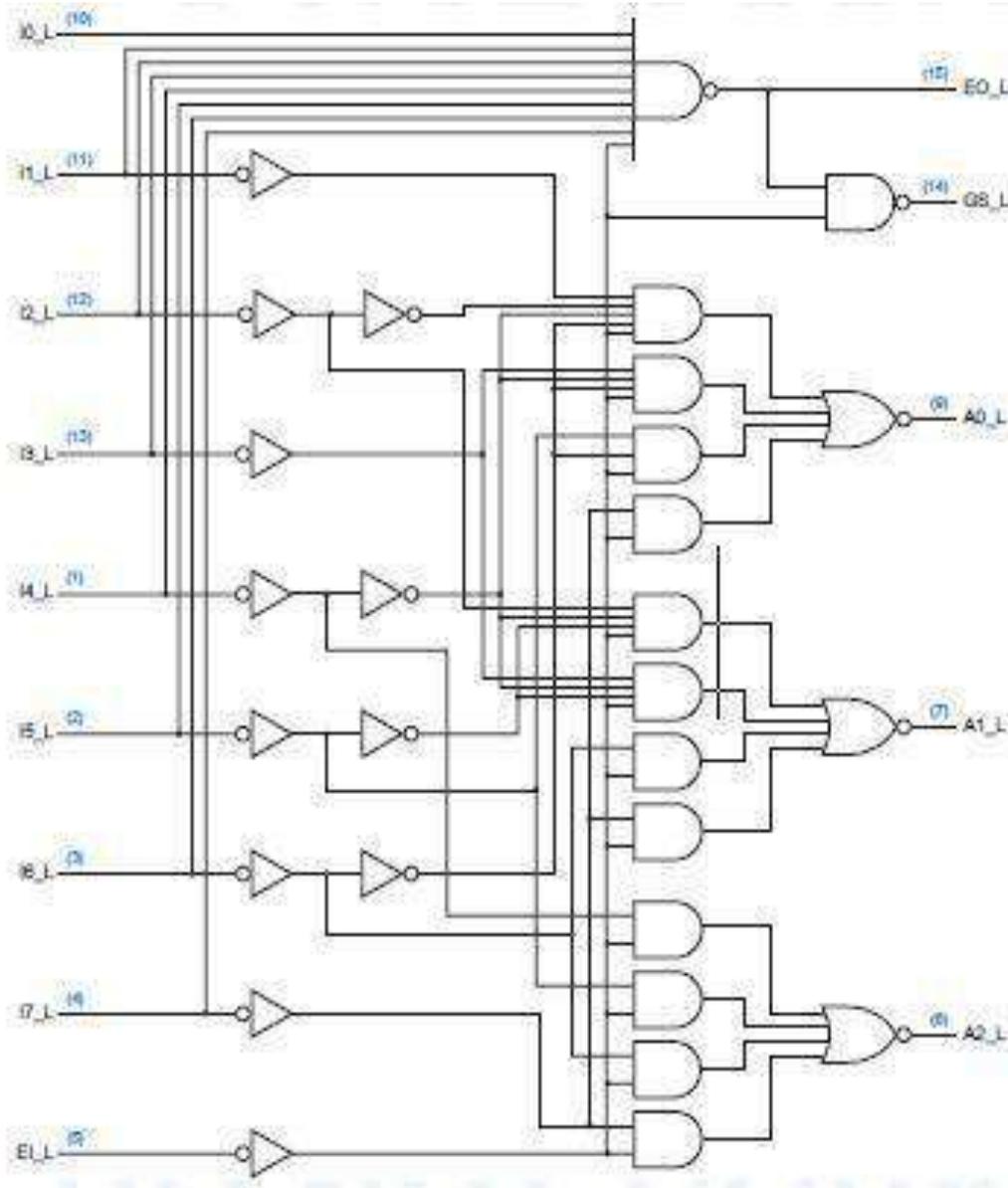


Figure 5-49 Logic diagram for the 74x148 8-input priority encoder, including pin numbers for a standard 16-pin dual in-line package.

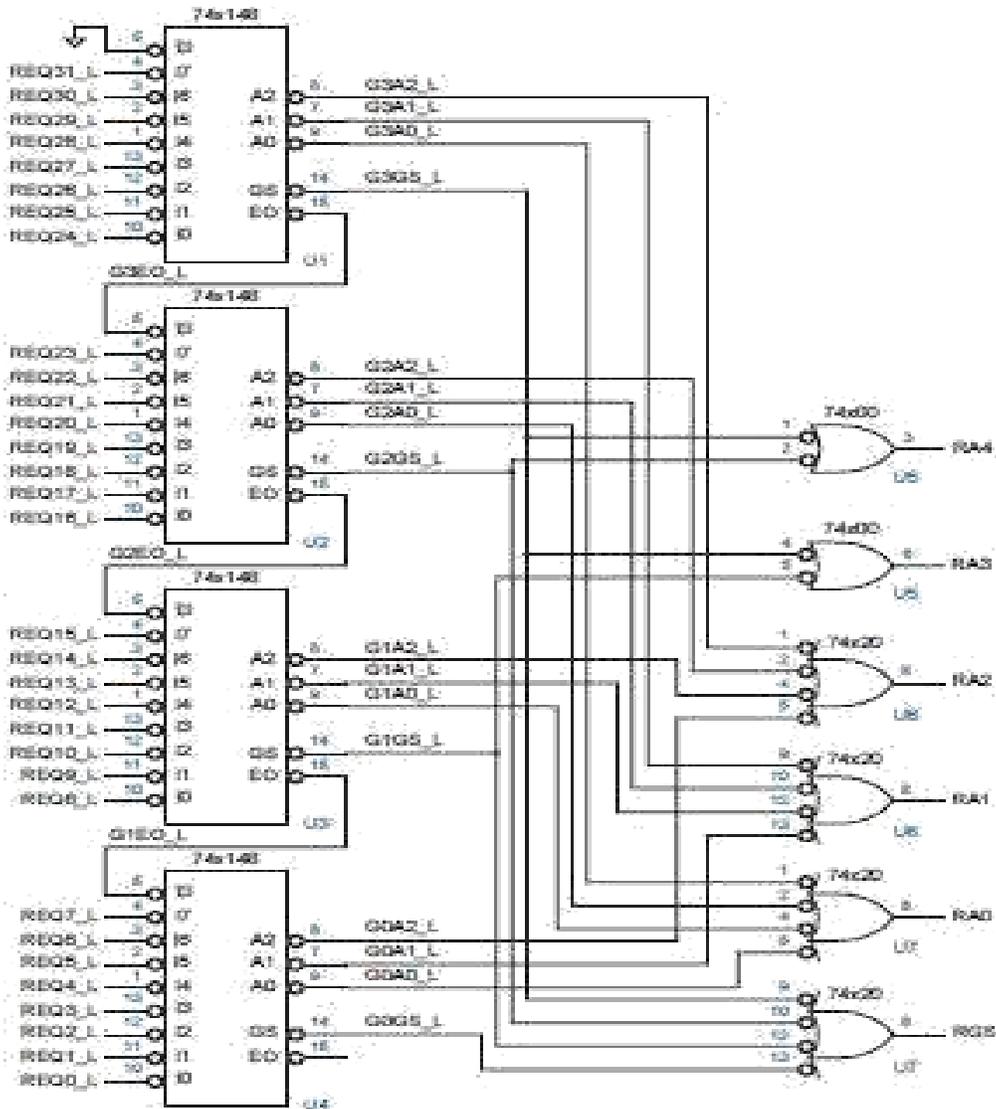


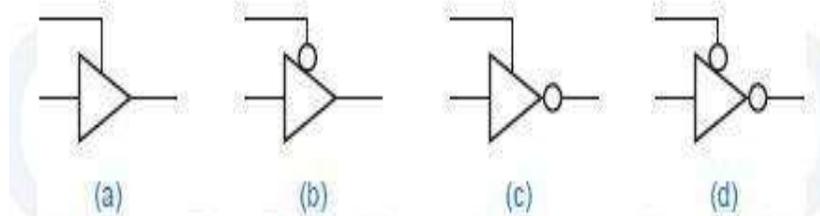
Figure 5-50 Four 74x148s cascaded to handle 32 requests.

Three-State Devices

5.6.1 Three-State Buffers

The most basic three-state device is a *three-state buffer*, often called a *three-state driver*.

The logic symbols for four physically different three-state buffers are shown in Figure 5-52.



The basic symbol is that of a noninverting buffer (a, b) or an inverter (c, d). The extra signal at the top of the symbol is a *three-state enable* input, which may be active high (a, c) or active low (b, d). When the enable input is asserted, the device behaves like an ordinary buffer or inverter. When the enable input is negated, the device output –floats!; that is, it goes to a high impedance (Hi-Z), disconnected state and functionally behaves as if it weren't even there.

Both enable inputs, G1_L and G2_L, must be asserted to enable the device's three-state outputs. The little rectangular symbols inside the buffer symbols indicate *hysteresis*, an electrical characteristic of the inputs that improves noise immunity. The 74x541 inputs typically have 0.4 volts of hysteresis.

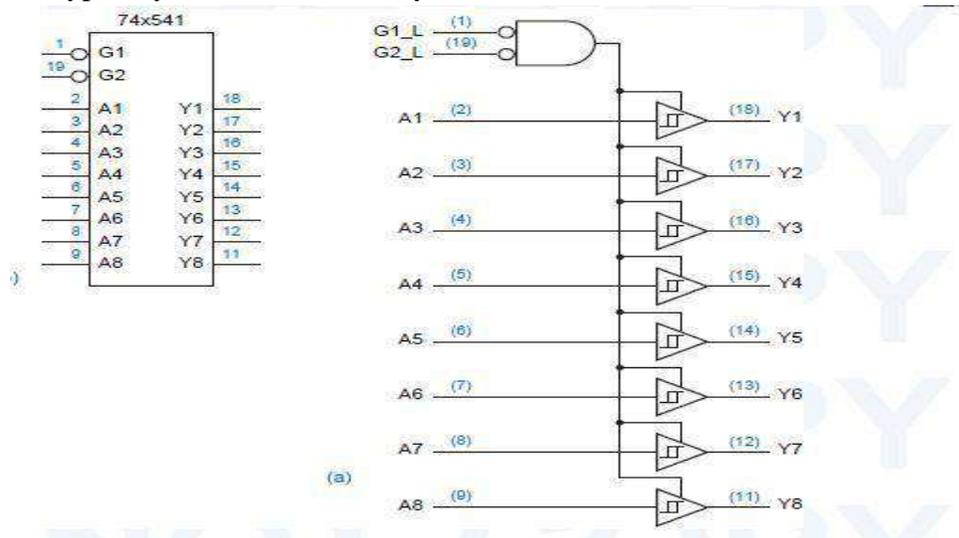
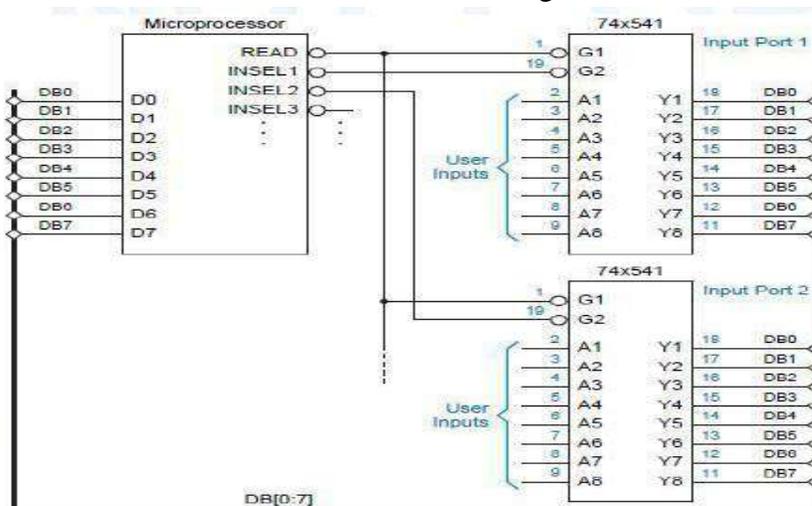
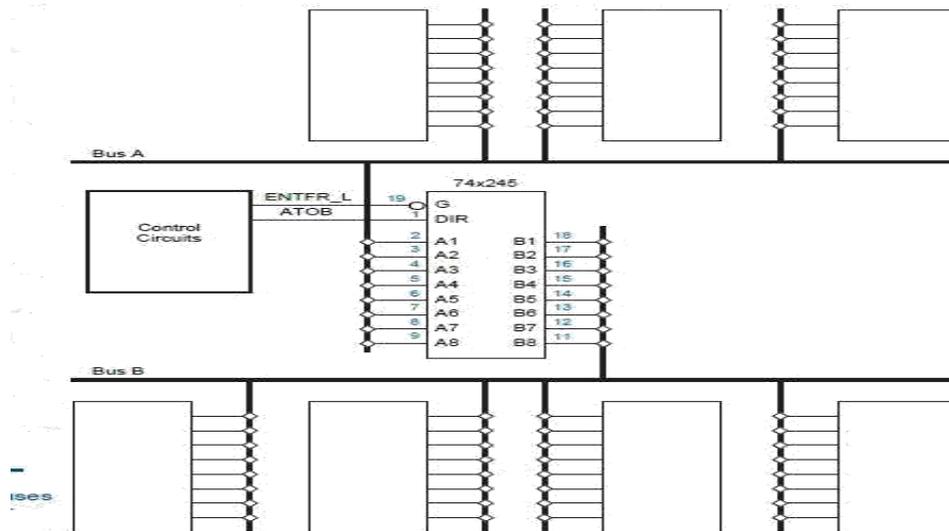


Figure 5-57 shows part of a microprocessor system with an 8-bit data bus, DB[0–7], and a 74x541 used as an input port. The microprocessor selects Input Port 1 by asserting INSEL1 and requests a read operation by asserting READ. The selected 74x541 responds by driving the microprocessor data bus with user supplied input data. Other input ports may be selected when a different INSEL line is asserted along with READ.



A bus transceiver is typically used between two *bidirectional buses*, as shown in Figure 5-59. Three different modes of operation are possible, depending on the state of G_L and DIR, as shown in Table 5-26. As usual, it is the designer's responsibility to ensure that neither bus is ever driven simultaneously by two devices. However, independent transfers where both buses are driven at the same time may occur when the transceiver is disabled, as indicated in the last row of the table.

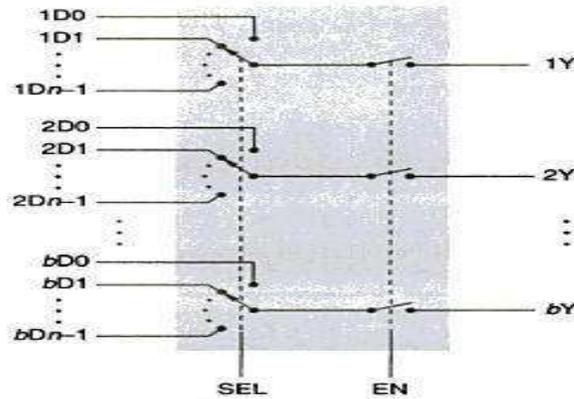


5.7 Multiplexers

A *multiplexer* is a digital switch—it connects data from one of n sources to its output. Figure 5-61(a) shows the inputs and outputs of an n -input, b -bit multiplexer. There are n sources of data, each of which is b bits wide. A multiplexer is often called a *mux* for short.

A multiplexer can use addressing bits to select one of several input bits to be the output. A selector chooses a single data input and passes it to the MUX output. It has one output selected at a time.

Figure shows a switch circuit that is roughly equivalent to the multiplexer. However, unlike a mechanical switch, a multiplexer is a unidirectional device: information flows only from inputs (on the left) to outputs (on the right). Multiplexers are obviously useful devices in any application in which data must be switched from multiple sources to a destination. A common application in computers is the multiplexer between the processor's registers and its arithmetic logic unit (ALU). For example, consider a 16-bit processor in which each instruction has a 3-bit field that specifies one of eight registers to use. This 3-bit field is connected to the select inputs of an 8-input, 16-bit multiplexer. The multiplexer's data inputs are connected to the eight registers, and its data outputs are connected to the ALU to execute the instruction using the selected register.



Standard MSI Multiplexers

The sizes of commercially available MSI multiplexers are limited by the number of pins available in an inexpensive IC package. Commonly used muxes come in 16-pin packages. Shown in fig which selects among eight 1-bit inputs. The select inputs are named C, B, and A, where C is most significant numerically. The enable input EN_L is active low; both active-high (Y) and active-low (Y_L) versions of the output are provided.

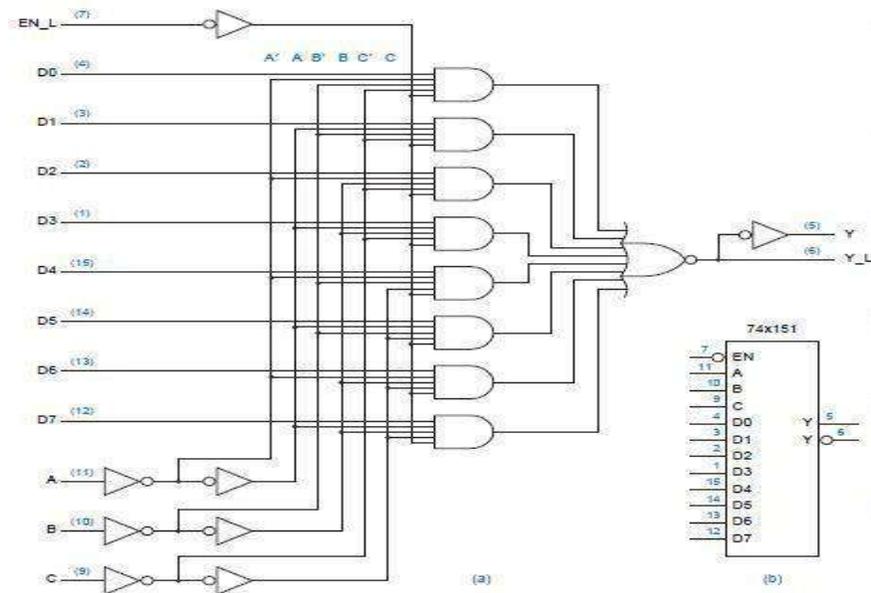
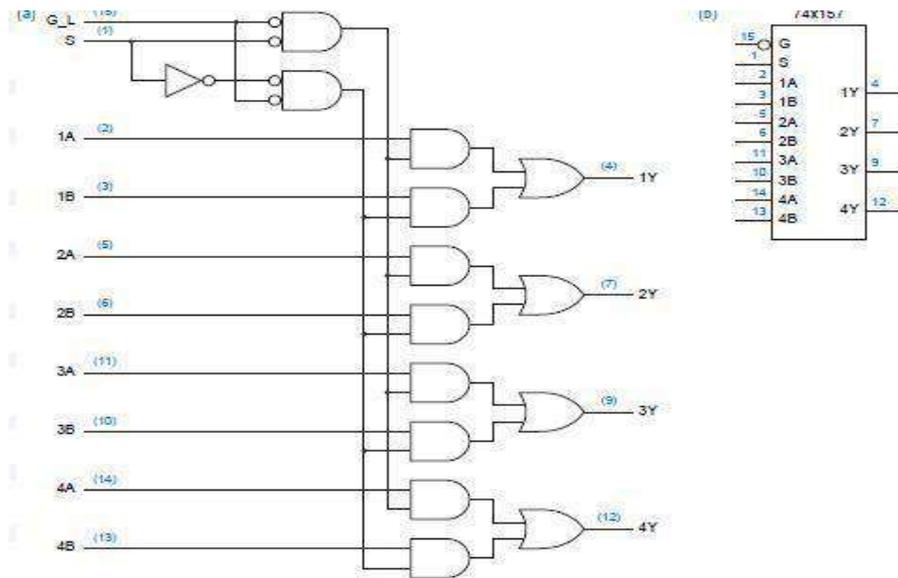


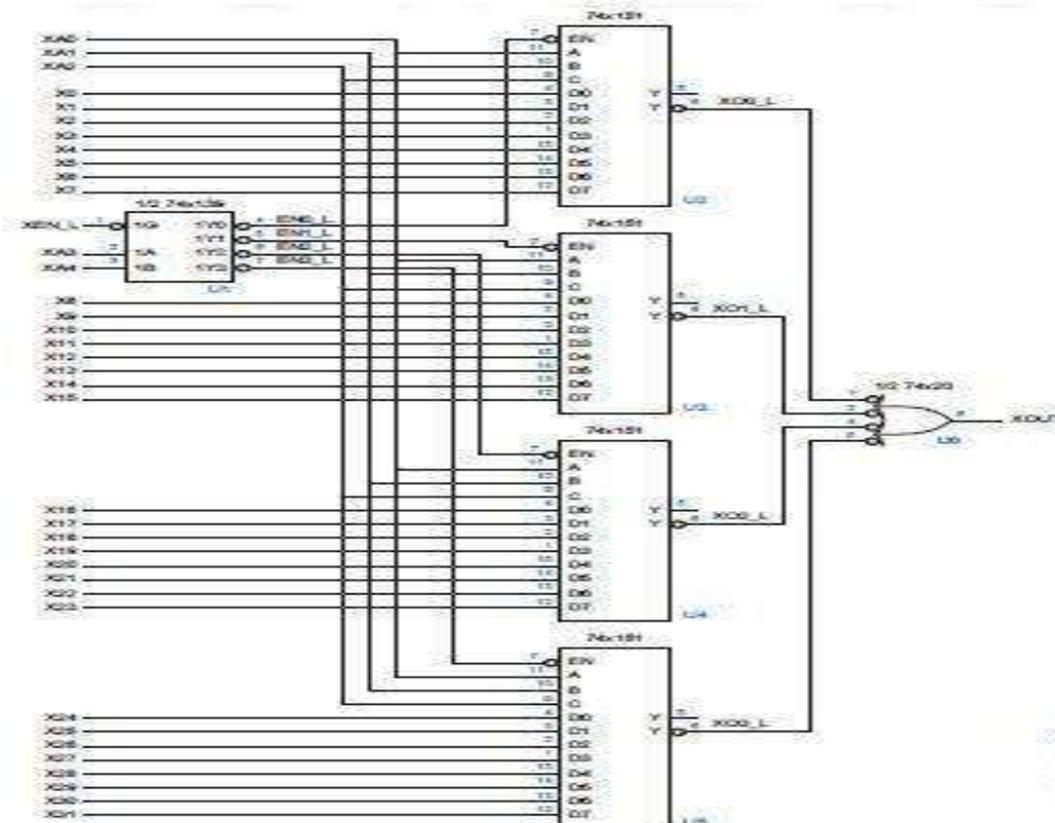
Figure The 74x151 8-input, 1-bit multiplexer: (a) logic diagram, including pin numbers for a standard 16-pin dual in-line package; (b) traditional logic symbol.

At the other extreme of muxes in 16-pin packages, we have the 74x157, shown in Figure, which selects between two 4-bit inputs. Just to confuse things, the manufacturer has named the select input S and the active-low enable input G_L. Also note that the data sources are named A and B .



Expanding Multiplexers

Seldom does the size of an MSI multiplexer match the characteristics of the problem at hand. For example, we suggested earlier that an 8-input, 16-bit multiplexer might be used in the design of a computer processor. This function could be performed by 16 74x151 8-input, 1-bit multiplexers or equivalent ASIC cells, each handling one bit of all the inputs and the output. The processor's 3-bit register-select field would be connected to the A, B, and C inputs of all 16 muxes, so they would all select the same register source at any given time.



Another dimension in which multiplexers can be expanded is the number of data sources. For example, suppose we needed a 32-input, 1-bit multiplexer. Figure shows one way to build it. Five select bits are required. A 2-to-4 decoder (one-half of a 74x139) decodes the two high-order select bits to enable one of four 74x151 8-input multiplexers. Since only one '151 is enabled at a time, the '151 outputs can simply be ORed to obtain the final output. The 32-to-1 multiplexer can also be built using 74x251s. The circuit is identical to Figure 5-65, except that the output NAND gate is eliminated. Instead, the Y (and, if desired, Y_L) outputs of the four '251s are simply tied together. The '139 decoder ensures that at most one of the '251s has its threestate outputs enabled at any time. If the '139 is disabled (XEN_L is negated), then all of the '251s are disabled, and the XOUT and XOUT_L outputs are undefined. However, if desired, resistors may be connected from each of these signals to 5 volts to pull the output HIGH in this case.

Multiplexers, Demultiplexers, and Buses

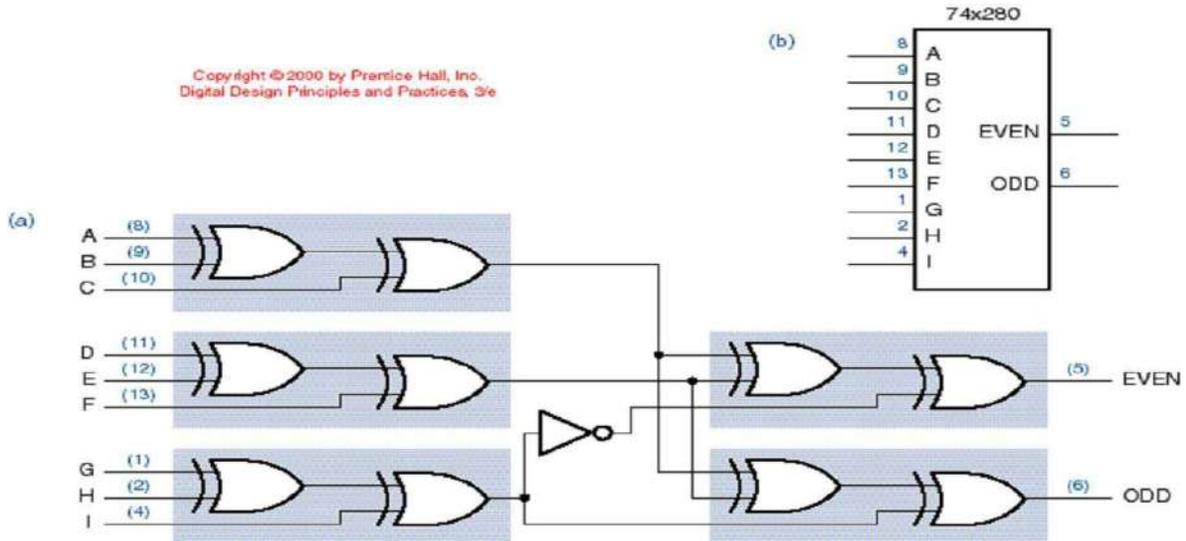
A multiplexer can be used to select one of n sources of data to transmit on a bus. At the far end of the bus, a *demultiplexer* can be used to route the bus data to one of m destinations. Such an application, using a 1-bit bus. In fact, block diagrams for logic circuits often depict multiplexers and demultiplexers, to suggest visually how a selected one of multiple data sources gets directed onto a bus and routed to a selected one of multiple destinations.

The function of a demultiplexer is just the inverse of a multiplexer's. For example, a 1-bit, n -output demultiplexer has one data input and s inputs to select one of n $2s$ data outputs. In normal operation, all outputs except the selected one are 0; the selected output equals the data input. This definition may be generalized for a b -bit, n -output demultiplexer; such a device has b data inputs, and its s select inputs choose one of n $2s$ sets of b data outputs.

A binary decoder with an enable input can be used as a demultiplexer, as shown in Figure 5-67. The decoder's enable input is connected to the data line, and its select inputs determine which of its output lines is driven with the data bit. The remaining output lines are negated.

Thus, the 74x139 can be used as a 2-bit, 4-output demultiplexer with active-low data inputs and outputs, and the 74x138 can be used as a 1-bit, 8-output demultiplexer. In fact, the manufacturer's catalog typically lists these ICs as -decoders/demultiplexers.

A Mux is used to select one of n sources of data to transmit on a bus. A demultiplexer can be used to route the bus data to one of m destinations. Just the inverse of a mux. A binary decoder with an enable input can be used as a Demux. E.g. 74x139 can be used as a 2-bit, 4-output Demux.



Parity-Checking Applications

described error-detecting codes that use an extra bit, called a parity bit, to detect errors in the transmission and storage of data. In an evenparity code, the parity bit is chosen so that the total number of 1 bits in a code word is even. Parity circuits like the 74x280 are used both to generate the correct value of the parity bit when a code word is stored or transmitted, and to check the parity bit when a code word is retrieved or received.

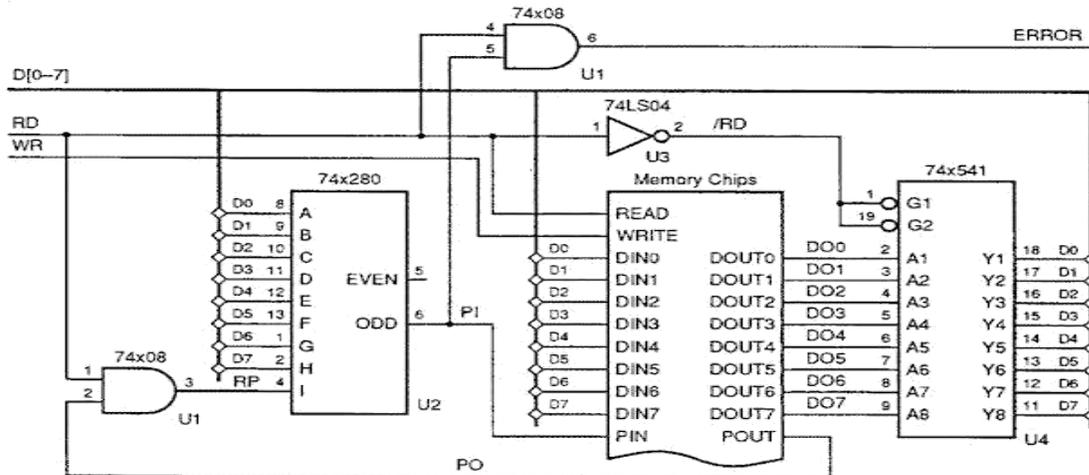


Figure 5-67 Parity generation and checking for an 8-bit-wide memory system.

Figure Parity generation and checking for an 8-bit-wide memory system.

Figure shows how a parity circuit might be used to detect errors in the memory of a microprocessor system. The memory stores 8-bit bytes, plus a parity bit for each byte. The microprocessor uses a bidirectional bus D[0:7] to transfer data to and from the memory. Two control lines, RD and WR, are used to indicate whether a read or write operation is desired, and an ERROR signal is asserted to indicate parity errors during read operations. Complete details of the memory chips, such as addressing inputs, are not shown; memory chips are described in detail chapter {MEMORY}.

To store a byte into the memory chips, we specify an address (not shown), place the byte on D[0–7], generate its parity bit on PIN, and assert WR. The AND gate on the I input of the 74x280 ensures that I is 0 except during read operations, so that during writes the '280's output depends only on the parity of the D-bus data. The '280's ODD output is connected to PIN, so that the total number of 1s stored is even. To retrieve a byte, we specify an address (not shown) and assert RD; the byte value appears on DOUT[0–7] and its parity appears on POUT. A 74x541 drives the byte onto the D bus, and the '280 checks its parity. If the parity of the 9-bit word DOUT[0–7],POUT is odd during a read, the ERROR signal is asserted.

Parity circuits are also used with error-correcting codes such as the Hamming codes described in Section 2.15.3. We showed the parity-check matrix for a 7-bit Hamming code in Figure 2-13 on page 59. We can correct errors in this code as shown in Figure 5-76. A 7-bit word, possibly containing an error, is presented on DU[1–7]. Three 74x280s compute the parity of the three bit-groups defined by the parity-check matrix. The outputs of the '280s form the syndrome, which is the number of the erroneous input bit, if any. A 74x138 is used to decode the syndrome. If the syndrome is zero, the NOERROR_L signal is asserted (this signal also could be named ERROR). Otherwise, the erroneous 74x280 ordingly.

Comparators

Comparing two binary words for equality is a commonly used operation in computer systems and device interfaces. we showed a system structure in which devices are enabled by comparing a –device select word with a predetermined –device ID. A circuit that compares two binary words and indicates whether they are equal is called a *comparator*. Some comparators interpret their input words as signed or unsigned numbers and also indicate an arithmetic relationship (greater or less than) between the words. These devices are often called *magnitude comparators*.

| Inputs | | | | Outputs | | |
|----------------|----------------|----------------|----------------|---------|-------|-------|
| A ₁ | A ₀ | B ₁ | B ₀ | A > B | A = B | A < B |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 |

A > B

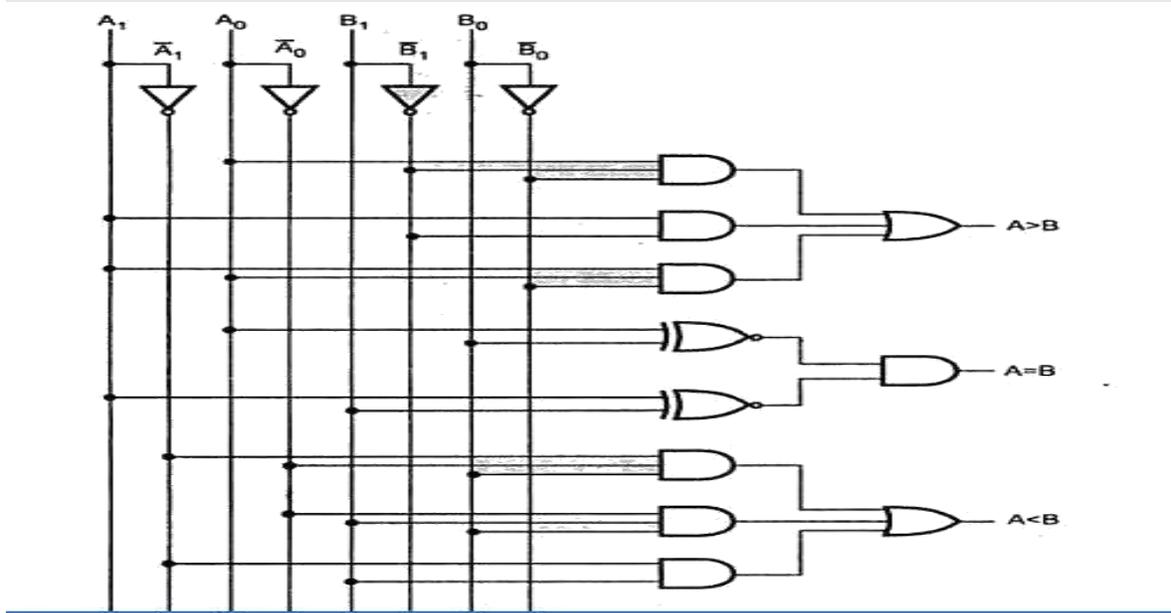
| | | | | | |
|----|-------------------------------|-------------------------------|----|----|----|
| | | B ₁ B ₀ | | | |
| | A ₁ A ₀ | 00 | 01 | 11 | 10 |
| 00 | | 0 | 0 | 0 | 0 |
| 01 | | 1 | 0 | 0 | 0 |
| 11 | | 1 | 1 | 0 | 1 |
| 10 | | 1 | 1 | 0 | 0 |

$$A > B = A_0 \bar{B}_1 \bar{B}_0 + A_1 \bar{B}_1 + A_1 A_0 \bar{B}_0$$

| | | | | | | | | | | |
|----|-------------------------------|-------|----|----|----|--|-------|----|----|----|
| | | A = B | | | | | A < B | | | |
| | A ₁ A ₀ | 00 | 01 | 11 | 10 | | 00 | 01 | 11 | 10 |
| 00 | | 1 | 0 | 0 | 0 | | 0 | 1 | 1 | 1 |
| 01 | | 0 | 1 | 0 | 0 | | 0 | 0 | 1 | 1 |
| 11 | | 0 | 0 | 1 | 0 | | 0 | 0 | 0 | 0 |
| 10 | | 0 | 0 | 0 | 1 | | 0 | 0 | 1 | 0 |

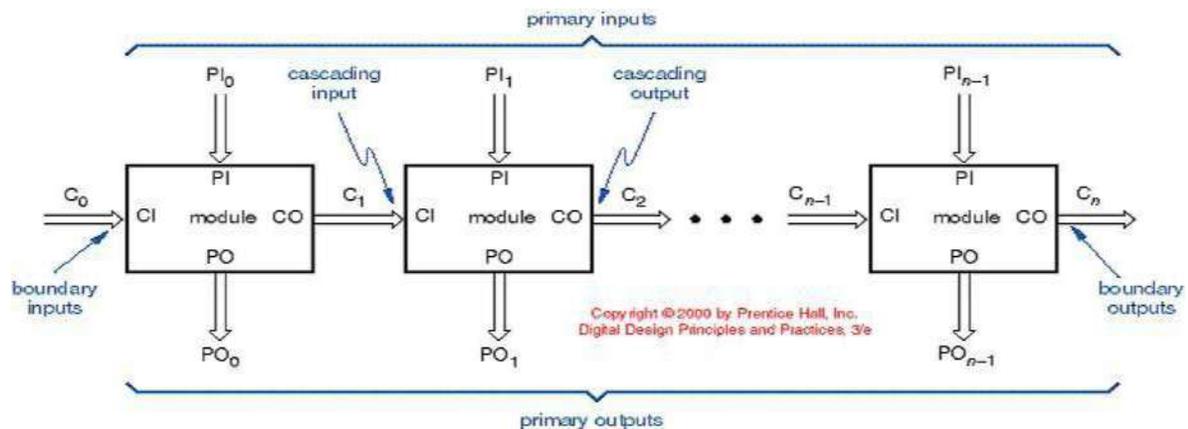
Fig. 4.93

$$\begin{aligned}
 (A = B) &= \bar{A}_1 \bar{A}_0 \bar{B}_1 \bar{B}_0 + \bar{A}_1 A_0 \bar{B}_1 B_0 \\
 &\quad + A_1 A_0 B_1 B_0 + A_1 \bar{A}_0 B_1 \bar{B}_0 \\
 &= \bar{A}_1 \bar{B}_1 (\bar{A}_0 \bar{B}_0 + A_0 B_0) \\
 &\quad + A_1 B_1 (A_0 B_0 + \bar{A}_0 \bar{B}_0) \\
 &= (A_0 \odot B_0) (A_1 \odot B_1) \\
 (A < B) &= \bar{A}_1 \bar{A}_0 B_0 + \bar{A}_0 B_1 B_0 + \bar{A}_1 B_1
 \end{aligned}$$



Iterative Circuits

An *iterative circuit* is a special type of combinational circuit, with the structure shown in Figure. The circuit contains n identical modules, each of which has both *primary inputs* and *outputs* and *cascading inputs* and *outputs*. The leftmost cascading inputs are called *boundary inputs* and are connected to fixed logic values in most iterative circuits. The rightmost cascading outputs are called *boundary outputs* and usually provide important information.



Iterative circuits are well suited to problems that can be solved by a simple iterative algorithm:

1. Set C_0 to its initial value and set i to 0.
2. Use C_i and PI_i to determine the values of PO_i and C_{i+1} .
3. Increment i .
4. If $i < n$, go to step 2.

AN ITERATIVE COMPARATOR

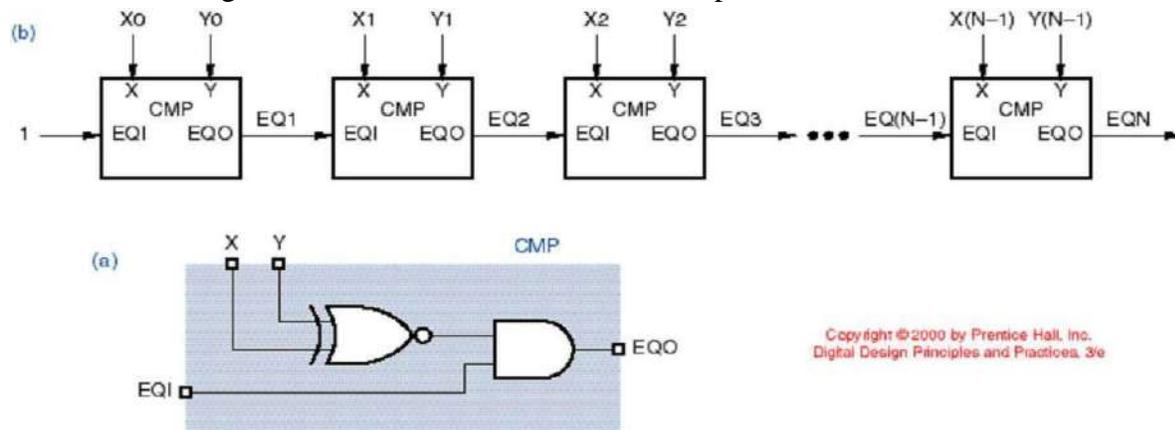
The n -bit comparators in the preceding subsection might be called *parallel comparators* because they look at each pair of input bits simultaneously and deliver the 1-bit comparison results in parallel to an n -input OR or AND function. It is also possible to design an n -bit iterative comparator that looks at its bits one at a time using a small, fixed amount of logic per bit.

An Iterative Comparator Circuit

Two n -bit values X and Y can be compared one bit at a time using a single bit EQ_i at each step to keep track of whether all of the bit-pairs have been equal so far:

1. Set EQ_0 to 1 and set i to 0.
2. If EQ_i is 1 and X_i and Y_i are equal, set EQ_{i+1} to 1. Else set EQ_{i+1} to 0.
3. Increment i .
4. If $i < n$, go to step 2.

Figure shows a corresponding iterative circuit. Note that this circuit has no primary outputs; the boundary output is all that interests us. Other iterative circuits, such as the ripple adder of Section 5.10.2, have primary outputs of interest. Given a choice between the iterative comparator circuit in this subsection and one of the parallel comparators shown previously, you would probably prefer the parallel comparator. The iterative comparator saves little if any cost, and it's very slow because the cascading signals need time to ripple from the leftmost to the rightmost module. Iterative circuits that process more than one bit



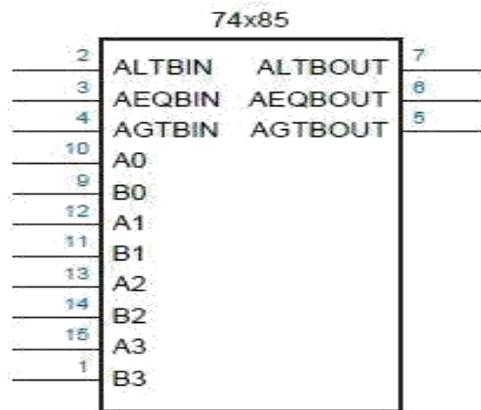
Standard MSI Comparators

Comparator applications are common enough that several MSI comparators have been developed commercially. The $74x85$ is a 4-bit comparator with the logic symbol shown in Figure 5-80. It provides a greater-than output (AGTBOUT) and a less-than output (ALTBOUT) as well as an equal output (AEQBOUT). The '85 also has *cascading inputs* (AGTBIN, ALTBIN, AEQBIN) for combining multiple '85s to create comparators for more than four bits. Both the cascading inputs and the outputs are arranged in a 1-out-of-3 code, since in normal operation exactly one input and one output should be asserted. The

cascading inputs are defined so the outputs of an '85 that compares less-significant bits are connected to the inputs of an '85 that compares more.

- 4 bit comparator
- 3 outputs : A=B, A<B, A>B
- 3 Cascading inputs

| | | | |
|--------------|--------|------------------|--------|
| □ Functional | Output | Equations | : |
| (A>B) | OUT)= | (A>B)+(A=B).(A>B | IN) |
| (A<B) | OUT)= | (A<B)+(A=B).(A<B | IN) |
| (A=B | OUT)= | (A=B).(A=B | IN) |
| □ Cascading | inputs | initial | values |
| (A=B | | N) | =1 |
| (A>B | | N) | =0 |
| (A<B | IN) | | =0 |



significant bits, as shown in Figure for a 12-bit comparator. This is an iterative circuit according to the definition Each '85 develops its cascading outputs roughly according to the following pseudo-logic equations: The parenthesized subexpressions above are not normal logic expressions, but indicate an arithmetic comparison that occurs between the A3–A0 and B3–B0 inputs. In other words, AGTBOUT is asserted if A > B or if A = B and AGTBIN is asserted (if the higher-order bits are equal, we have to look at the lower-order bits for the answer).

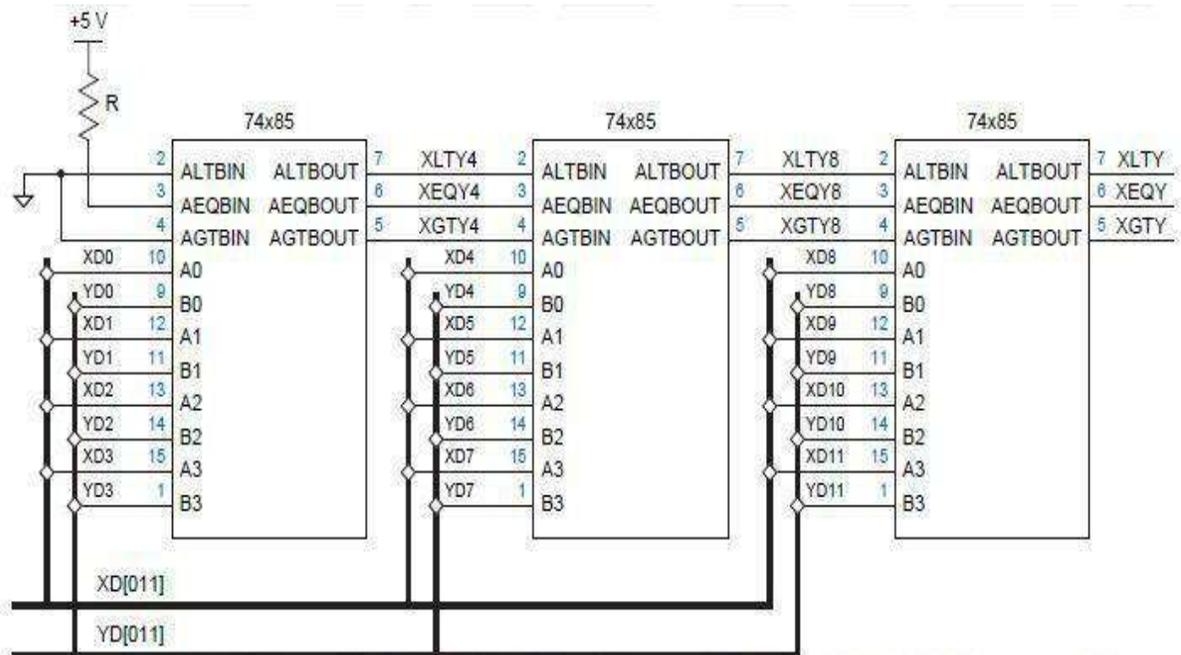
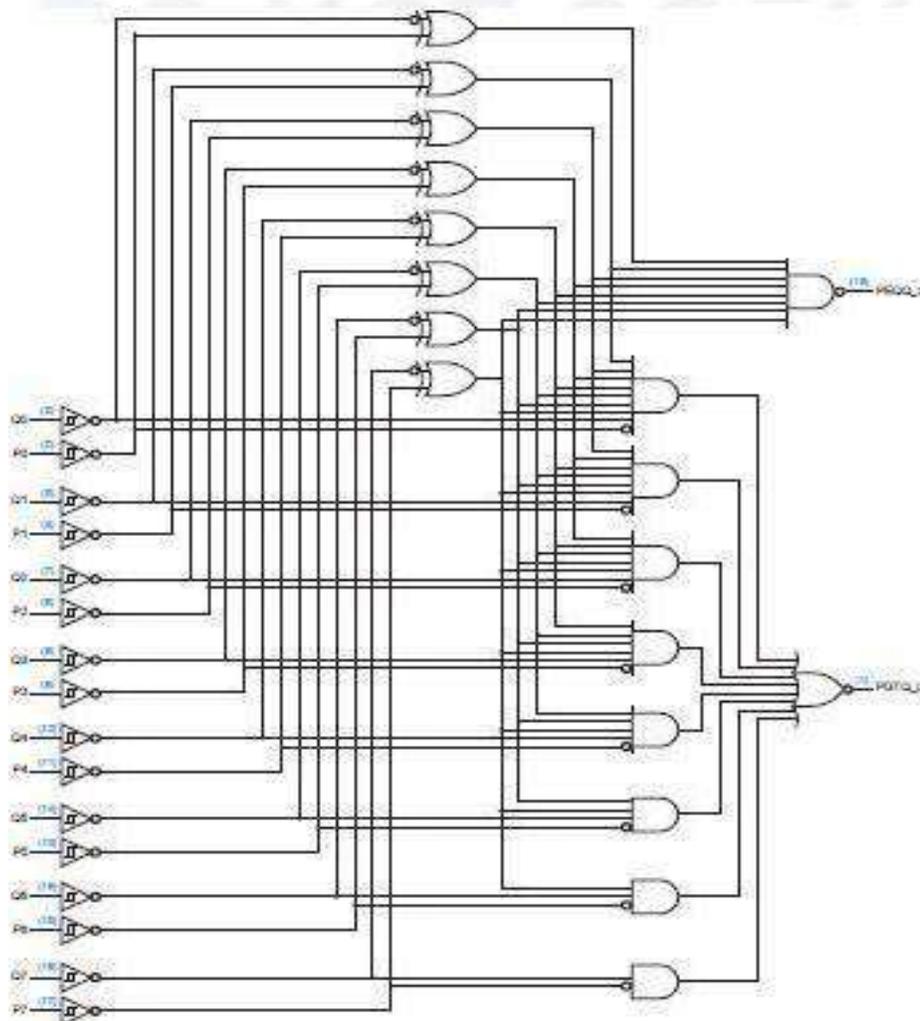


Figure A 12-bit comparator using 74x85s.



Adders, Subtractors, and ALUs

Addition is the most commonly performed arithmetic operation in digital systems. An *adder* combines two arithmetic operands using the addition rules. The same addition rules and therefore the same adders are used for both unsigned and two's-complement numbers. An adder can perform subtraction as the addition of the minuend and the complemented (negated) subtrahend, but you can also build *subtractor* *Half Adder: adds two 1-bit operands*

Half Adders and Full Adders

The simplest adder, called a *half adder*, adds two 1-bit operands X and Y, producing a 2-bit sum. The sum can range from 0 to 2, which requires two bits to express. The low-order bit of the sum may be named HS (half sum), and the high-order bit may be named CO (carry out). We can write the following equations for HS and CO:

| Inputs | | Outputs | |
|--------|---|---------|-----|
| A | B | Carry | Sum |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Table 4.32 Truth table for half-adder

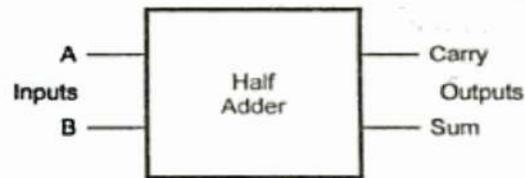
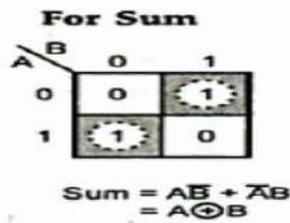
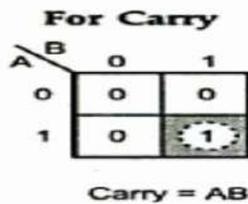


Fig. 4.99 Block schematic of half-adder



Logic Diagram

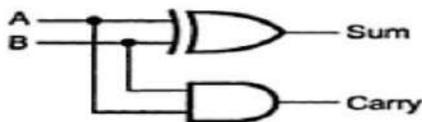
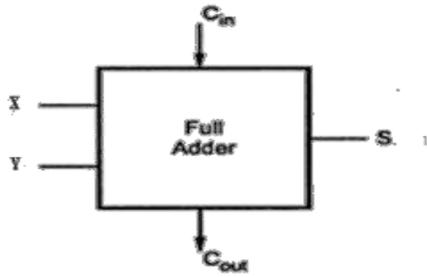


Fig. 4.101 Logic diagram for half-adder

To add operands with more than one bit, we must provide for carries between bit positions. The building block for this operation is called a *full adder*. Besides the addend-bit inputs X and Y, a full adder has a carry-bit input, CIN. The sum of the three inputs can range from 0 to 3, which can still be expressed with just two output bits, S and COUT, having the following equations: Here, S is 1 if an odd number of the inputs are 1, and COUT is 1 if two or more of the inputs are 1.



Block schematic of full-adder

| X | Y | Cin | S | Cout |
|---|---|-----|---|------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

For Carry (C_{out})

| X | YCin | | | |
|---|------|----|----|----|
| | 00 | 01 | 11 | 10 |
| 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 |

For Sum

| X | YCin | | | |
|---|------|----|----|----|
| | 00 | 01 | 11 | 10 |
| 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |

Subtractors

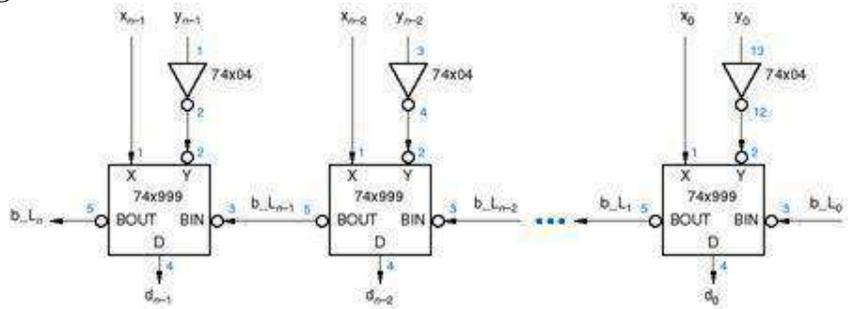
A binary subtraction operation analogous to binary addition. A *full subtractor* handles one bit of the binary subtraction algorithm, having input bits X (minuend), Y (subtrahend), and BIN (borrow in), and output bits D (difference) and BOUT (borrow out).

X, Y are n-bit unsigned binary numbers

Addition : $S = X + Y$

Subtraction : $D = X - Y$
 $= X + \text{Two's Complement of } Y = X + \text{Complement of } Y + 1$

Using Adder as a Subtractor



MSI ADDERS

The 74x283 is a 4-bit binary adder that forms its sum and carry outputs with just a few levels of logic, using the carry lookahead technique. The older 74x83 is identical except for its pinout, which has nonstandard locations for power and ground. The logic diagram for the '283, has just a few differences from the general carry-lookahead design that we described in the preceding subsection. First of all, its addends are named A and B instead of X and Y; no big deal. Second, it produces active-low versions of the carry-generate (g_i) and carry-propagate (p_i) signals, since inverting gates are generally faster than noninverting ones.

Third, it takes advantage of the fact that we can algebraically manipulate the half-sum equation as follows: Thus, an AND gate with an inverted input can be used instead of an XOR gate to create each half-sum bit.

Finally, the '283 creates the carry signals using an INVERT-OR-AND structure (the DeMorgan equivalent of an AND-OR-INVERT), which has about the same delay as a single CMOS or TTL inverting gate. This requires some explaining, since the carry equations that we derived in the preceding subsection are used in a slightly modified form. In particular, the c_{i+1} equation uses the term

$p_i g_i$ instead of g_i . This has no effect on the output, since p_i is always 1 when g_i is 1.

However, it allows the equation to be factored as follows:

This leads to the following carry equations, which are used by the circuit :

$$\begin{aligned}
 hs_i &= x_i \oplus y_i \\
 &= x_i \cdot y_i' + x_i' \cdot y_i \\
 &= x_i \cdot y_i' + x_i \cdot x_i' + x_i' \cdot y_i + y_i \cdot y_i' \\
 &= (x_i + y_i) \cdot (x_i' + y_i') \\
 &= (x_i + y_i) \cdot (x_i \cdot y_i)' \\
 &= p_i \cdot g_i'
 \end{aligned}$$

The propagation delay from the C0 input to the C4 output of the '283 is very short, about the same as two inverting gates. As a result, fairly fast *groupripple adders* with more than

four bits can be made simply by cascading the carry outputs and inputs of '283s, as shown in Figure for a 16-bit adder. The total propagation delay from C0 to C16 in this circuit is about the same as

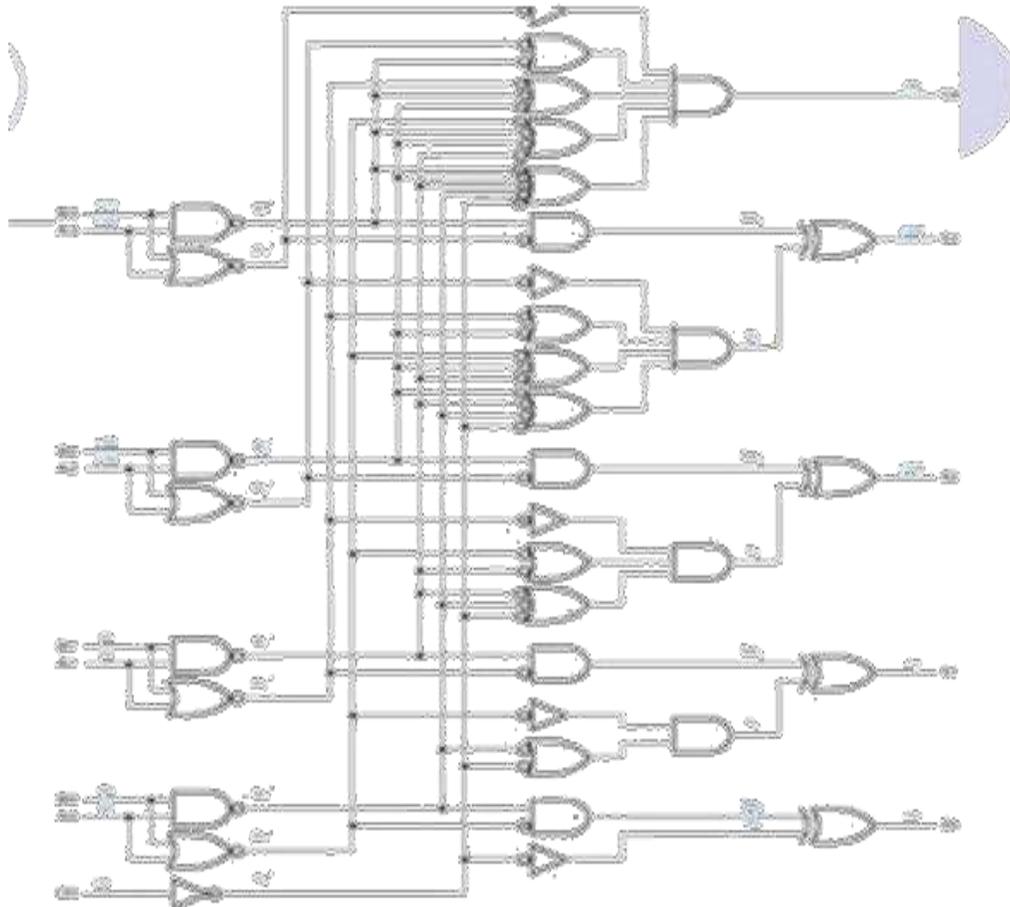
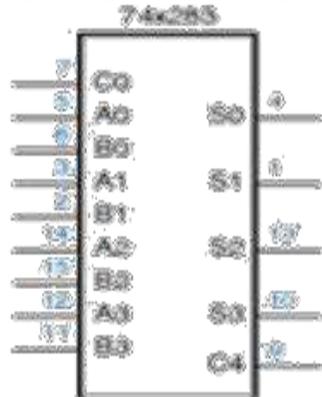
that of eight inverting gates.

$$c_1 = p_0 \cdot (g_0 + c_0)$$

$$c_2 = p_1 \cdot (g_1 + c_1)$$

$$= p_1 \cdot (g_1 + p_0 \cdot (g_0 + c_0))$$

$$= p_1 \cdot (g_1 + p_0) \cdot (g_0 + c_0)$$



MSI Arithmetic and Logic Units

An *arithmetic and logic unit (ALU)* is a combinational circuit that can perform any of a number of different arithmetic and logical operations on a pair of b -bit operands. The operation to be performed is specified by a set of function-select inputs. Typical MSI ALUs have 4-bit operands and three to five function select inputs, allowing up to 32 different functions to be performed. Figure is a logic symbol for the *74x181* 4-bit ALU. The operation performed by the '181 is selected by the M and S3–S0 inputs, as detailed in Table. Note that the identifiers A, B, and F in the table refer to the 4-bit words A3–A0, B3–B0, and F3–F0;

The 181's M input selects between arithmetic and logical operations. When M = 1, logical operations are selected, and each output F_i is a function only of the corresponding data inputs, A_i and B_i . No carries propagate between stages, and the CIN input is ignored. The S3–S0 inputs select a particular logical operation; any of the 16 different combinational logic functions on two variables may be selected.

When M = 0, arithmetic operations are selected, carries propagate between the stages, and CIN is used as a carry input to the least significant stage. For operations larger than four bits, multiple '181 ALUs may be cascaded like the group-ripple adder in the Figure 5-91, with the carry-out (COUT) of each ALU connected to the carry-in (CIN) of the next most significant stage. The same function-select signals (M, S3–S0) are applied to all the '181s in the cascade.



To perform two's-complement addition, we use S3–S0 to select the operation $-A$ plus B plus CIN. The CIN input of the least-significant ALU is normally set to 0 during addition operations. To perform two's-complement subtraction, we use S3–S0 to select the operation A minus B minus plus CIN. In this case, the CIN input of the least significant ALU is normally set to 1, since CIN acts as the complement of the borrow during subtraction.

The '181 provides other arithmetic operations, such as $-A$ minus 1 plus CIN, that are useful in some applications (e.g., decrement by 1). It also provides a bunch of weird arithmetic

operations, such as $-A \cdot B$ plus $(A \cdot B)$ plus CIN, that are almost never used in practice, but that fall out of the circuit for free. Notice that the operand inputs A_3_L – A_0_L and B_3_L – B_0_L and the function outputs F_3_L – F_0_L of the '181 are active low. The '181 can also be used with active-high operand inputs and function outputs. In this case, a different version of the function table must be constructed. When $M = 1$, logical operations are still performed, but for a given input combination on S_3 – S_0 , the function obtained is precisely the dual of the one listed in Table . When $M = 0$, arithmetic operations are performed, but the function table is once again different.

| Inputs | | | | Function | |
|--------|-------|-------|-------|--|-------------------|
| S_3 | S_2 | S_1 | S_0 | $M = 0$ (arithmetic) | $M = 1$ (logic) |
| 0 | 0 | 0 | 0 | $F = A$ minus 1 plus CIN | $F = A'$ |
| 0 | 0 | 0 | 1 | $F = A \cdot B$ minus 1 plus CIN | $F = A' + B'$ |
| 0 | 0 | 1 | 0 | $F = A \cdot B'$ minus 1 plus CIN | $F = A' + B$ |
| 0 | 0 | 1 | 1 | $F = 1111$ plus CIN | $F = 1111$ |
| 0 | 1 | 0 | 0 | $F = A$ plus $(A + B')$ plus CIN | $F = A' \cdot B'$ |
| 0 | 1 | 0 | 1 | $F = A \cdot B$ plus $(A + B')$ plus CIN | $F = B'$ |
| 0 | 1 | 1 | 0 | $F = A$ minus B minus 1 plus CIN | $F = A \oplus B'$ |
| 0 | 1 | 1 | 1 | $F = A + B'$ plus CIN | $F = A + B'$ |
| 1 | 0 | 0 | 0 | $F = A$ plus $(A + B)$ plus CIN | $F = A' \cdot B$ |
| 1 | 0 | 0 | 1 | $F = A$ plus B plus CIN | $F = A \oplus B$ |
| 1 | 0 | 1 | 0 | $F = A \cdot B'$ plus $(A + B)$ plus CIN | $F = B$ |
| 1 | 0 | 1 | 1 | $F = A + B$ plus CIN | $F = A + B$ |
| 1 | 1 | 0 | 0 | $F = A$ plus A plus CIN | $F = 0000$ |
| 1 | 1 | 0 | 1 | $F = A \cdot B$ plus A plus CIN | $F = A \cdot B'$ |
| 1 | 1 | 1 | 0 | $F = A \cdot B'$ plus A plus CIN | $F = A \cdot B$ |
| 1 | 1 | 1 | 1 | $F = A$ plus CIN | $F = A$ |

Combinational Multipliers

Combinational Multiplier Structures

Multiplier has an algorithm that uses n shifts and adds to multiply n -bit binary numbers. Although the shift-and-add algorithm emulates the way that we do paper-and-pencil multiplication of decimal numbers, there is nothing inherently sequential or time dependent about multiplication. That is, given two n -bit input words X and Y , it is possible to write a truth table that expresses the $2n$ -bit product $P = X \cdot Y$ as a *combinational* function of X and Y . A *combinational multiplier* is a logic circuit with such a truth table. Most approaches to combinational multiplication are based on the paper-and-pencil shift-and-add algorithm. Figure 5-96 illustrates the basic idea for an 8×8 multiplier for two unsigned integers, multiplicand $X = x_7x_6x_5x_4x_3x_2x_1x_0$ and multiplier $Y = y_7y_6y_5y_4y_3y_2y_1y_0$. We call each row a *product component*, a shifted multiplicand that is multiplied by 0 or 1 depending on the corresponding multiplier bit. Each small box

represents one product-component bit $y_i x_j$, the logical AND of multiplier bit y_i and multiplicand bit x_j . The product $P = p_{15}p_{14} \dots p_2p_1p_0$ has 16 bits and is obtained by adding together all the product components.

Figure shows one way to add up the product components. Here, the product-component bits have been spread out to make space, and each $\text{--}+$ box is a full adder equivalent to Figure 5-85(c) on page 391. The carries in each row of full adders are connected to make an 8-bit ripple adder. Thus, the first ripple adder combines the first two product components to produce the first partial product. Subsequent adders combine each partial product with the next product component.

Sequential multipliers use a single adder and a register to accumulate the partial products. The partial-product register is initialized to the first product component, and for an n n -bit multiplication, $n - 1$ steps are taken and the adder is used $n - 1$ times, once for each of the remaining $n - 1$ product components to be added to the partial-product register. Some sequential multipliers use a trick called *carry-save addition* to speed up multiplication. The idea is to break the carry chain of the ripple adder to shorten the delay of each addition. This is done by applying the carry output from bit i during step j to the carry input for bit $i+1$ during the *next* step, $j+1$. After the last product component is added, one more step is needed in which the carries are hooked up in the usual way and allowed to ripple from the least to the most significant bit.

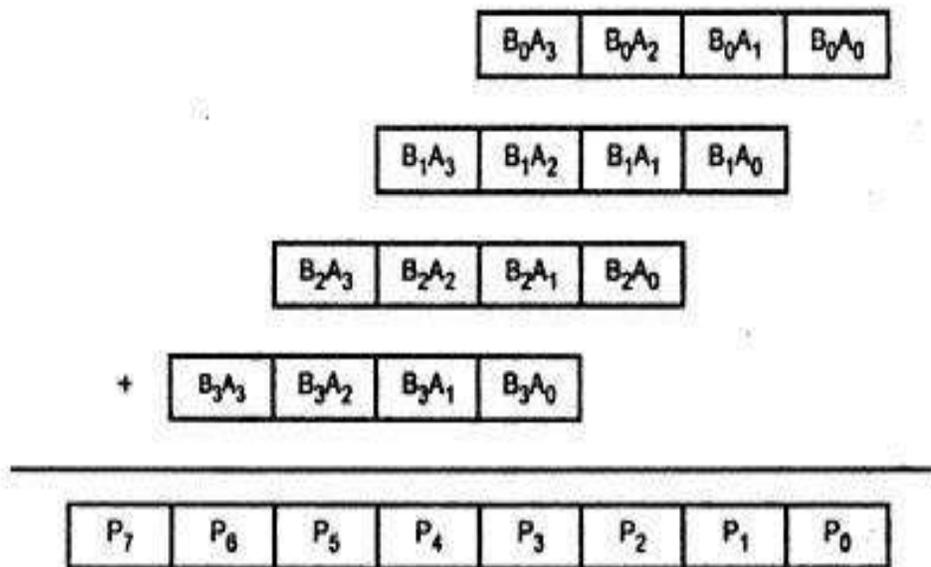
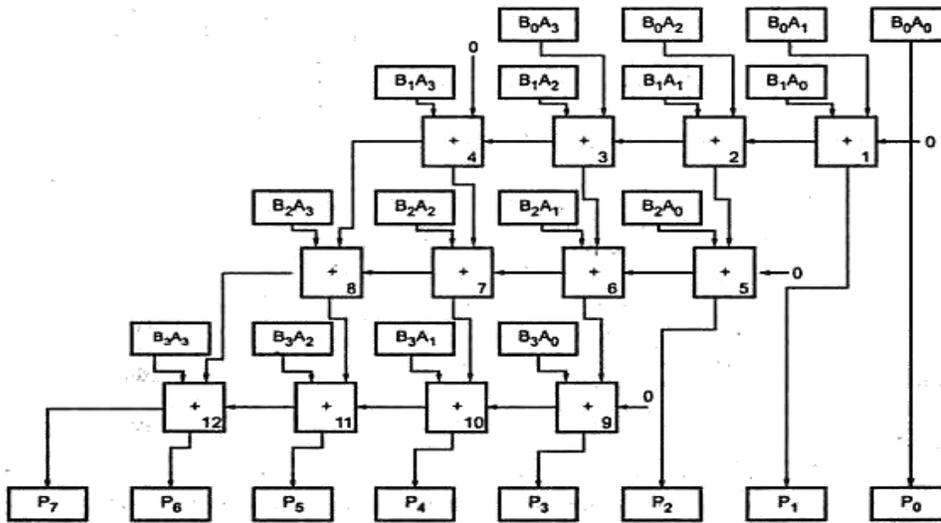


Fig. Multiplication process of 4×4 multiplier

4x4 combinational multipliers



4X4 combinational multiplier with carry save

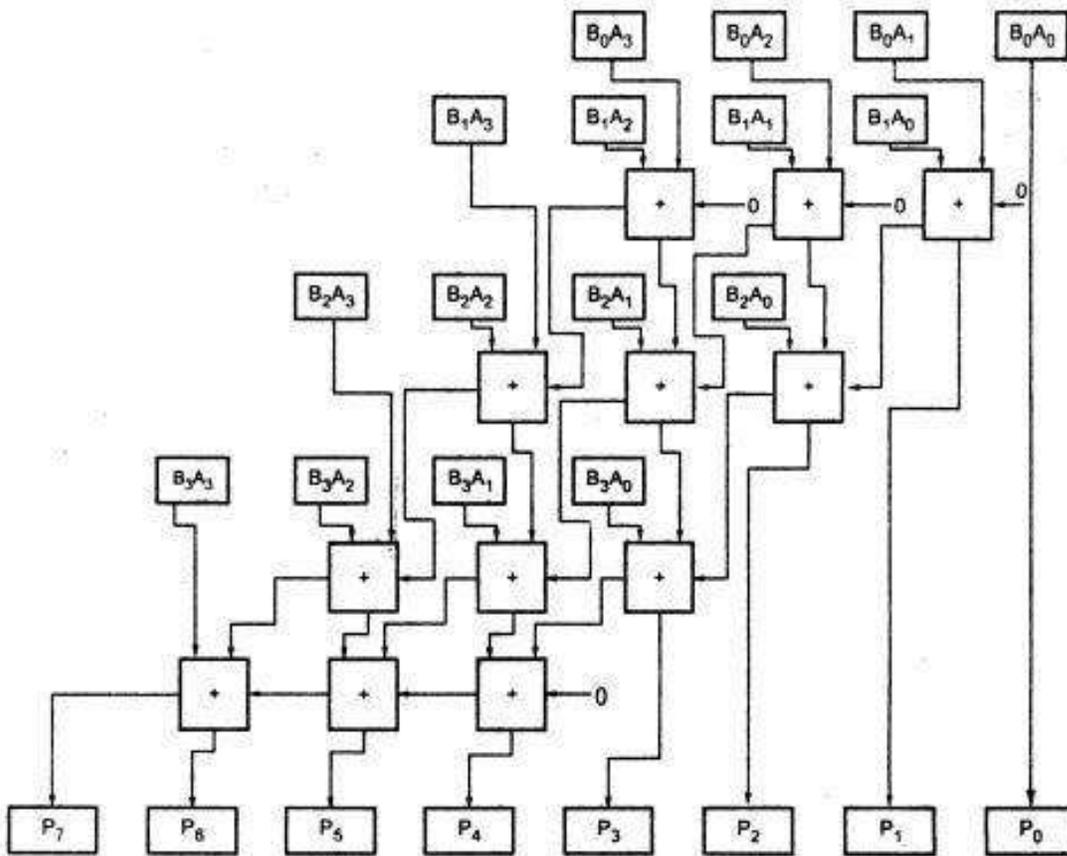
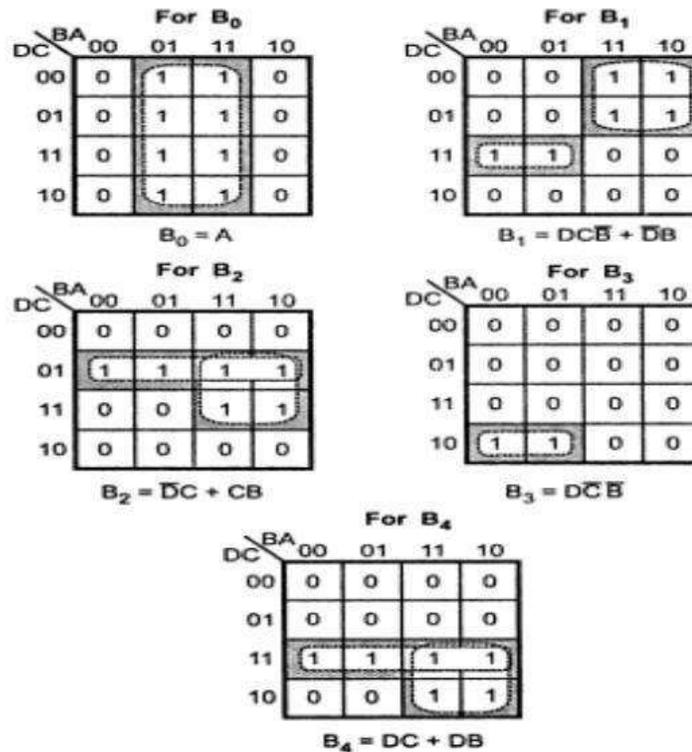


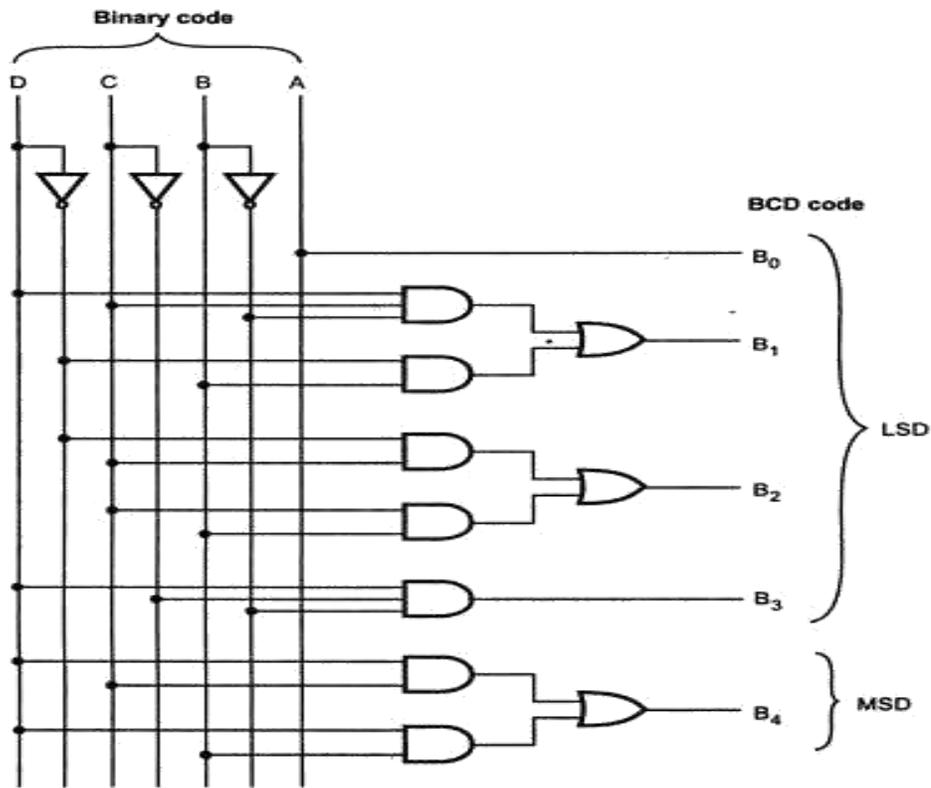
Fig. 4 x 4 combination multiplier with carry save addition

Code Converters
Binary to BCD converter

| Binary code | | | | BCD code | | | | |
|-------------|---|---|---|----------------|----------------|----------------|----------------|----------------|
| D | C | B | A | B ₄ | B ₃ | B ₂ | B ₁ | B ₀ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |

K-map simplification



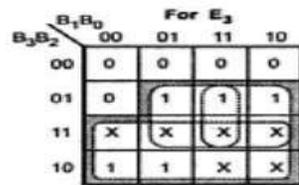


Binary to BCD converter

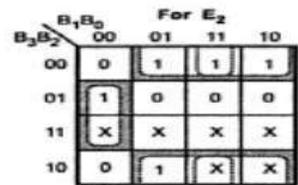
BCD to Excess-3 code converter

| Decimal | B ₃ | B ₂ | B ₁ | B ₀ | E ₃ | E ₂ | E ₁ | E ₀ |
|---------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 5 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 6 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 7 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 8 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |

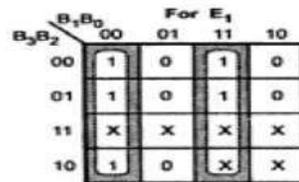
K-map simplification



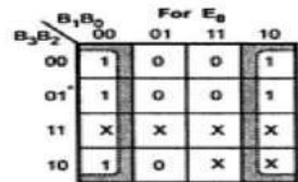
$$\therefore E_3 = B_3 + B_2(B_0 + B_1)$$



$$\therefore E_2 = B_2B_1B_0 + B_2(B_0 + B_1)$$



$$E_1 = B_1B_0 + B_1B_0 = B_1 \oplus B_0$$



$$E_0 = B_0$$

Logic diagram

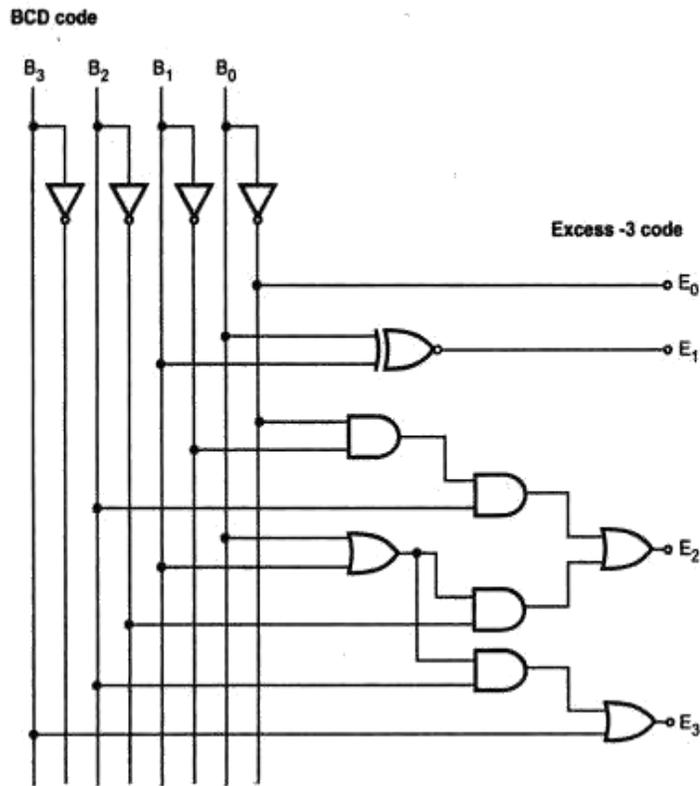
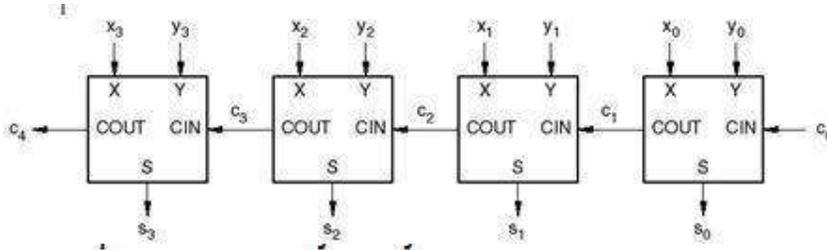


Fig. BCD to Excess-3 code converter

Ripple Adders

Two binary words, each with n bits, can be added using a *ripple adder*—a cascade of n full-adder stages, each of which handles one bit. Figure 5-86 shows the circuit for a 4-bit ripple adder. The carry input to the least significant bit (c_0) is normally set to 0, and the carry output of each full adder is connected to the carry input of the next most significant full adder. The ripple adder is a classic example of an iterative circuit as defined in Section 5.9.2. A ripple adder is slow, since in the worst case a carry must propagate from the least significant full adder to the most significant one. This occurs if, for example, one addend is 11 11 and the other is 00 01. Assuming that all of the addend bits are presented simultaneously, the total worst-case delay is where t_{XYCout} is the delay from X or Y to COUT in the least significant stage, $t_{CinCout}$ is the delay from CIN to COUT in the middle stages, and t_{CinS} is the delay from CIN to S in the most significant stage. A faster adder can be built by obtaining each sum output s_i with just two levels of logic. This can be accomplished by writing an equation for s_i in terms of x_0-x_i , y_0-y_i , and c_0 , multiplying out or adding out to obtain a sum-of-products or product-of-sums expression, and implementing the corresponding AND-OR or OR-AND circuit. Unfortunately, beyond s_2 , the resulting

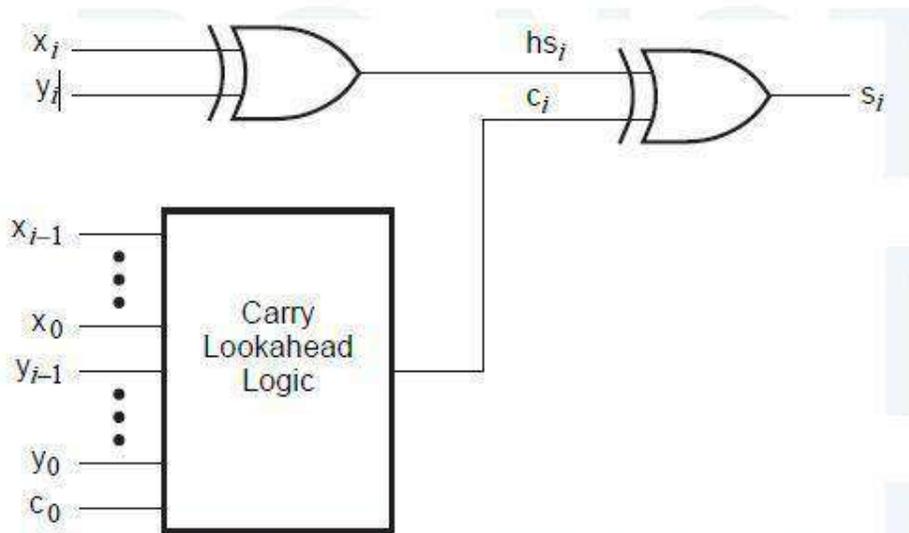
expressions have too many terms, requiring too many first-level gates and more inputs than typically possible on the second-level gate. For example, even assuming that $c_0 = 0$, a two-level AND-OR circuit for s_2 requires fourteen 4-input ANDs, four 5-input ANDs, and an 18-input OR gate; higher-order sum bits are even worse. Nevertheless, it is possible to build adders with just a few levels of delay using a more reasonable number of gates.



Carry Lookahead Adders

However, if we're willing to forego the XOR expansion, we can at least streamline the design of c_i logic using ideas of *carry lookahead* discussed in this subsection. The block labeled *Carry Lookahead Logic* calculates c_i in a fixed, small number of logic levels for any reasonable value of i . Two definitions are the key to carry lookahead logic:

- For a particular combination of inputs x_i and y_i , adder stage i is said to *generate* a carry if it produces a carry-out of 1 ($c_i = 1$) independent of the inputs on x_0 – x_{i-1} , y_0 – y_{i-1} , and c_0 .
- For a particular combination of inputs x_i and y_i , adder stage i is said to *propagate* carries if it produces a carry-out of 1 ($c_i = 1$) in the presence of an input combination of x_0 – x_{i-1} , y_0 – y_{i-1} , and c_0 that causes a carry-in of



Corresponding to these definitions, we can write logic equations for a carry-generate signal, g_i , and a carry-propagate signal, p_i , for each stage of a carry lookahead adder:

That is, a stage unconditionally generates a carry if both of its addend bits are 1, and it propagates carries if at least one of its addend bits is 1. The carry output of a stage can now be written in terms of the generate and propagate signals: To eliminate carry ripple, we recursively expand the c_i term for each stage, and multiply out to obtain a 2-level AND-OR expression. Using this technique, we can obtain the following carry equations for the first four adder stages:

Each equation corresponds to a circuit with just three levels of delay—one for the generate and propagate signals, and two for the sum-of-products shown. A *carry lookahead adder* uses three-level equations such as these in each adder stage for the block labeled -carry lookahead.

$$\begin{aligned}
 c_1 &= g_0 + p_0 \cdot c_0 \\
 c_2 &= g_1 + p_1 \cdot c_1 \\
 &= g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) \\
 &= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \\
 c_3 &= g_2 + p_2 \cdot c_2 \\
 &= g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0) \\
 &= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \\
 c_4 &= g_3 + p_3 \cdot c_3 \\
 &= g_3 + p_3 \cdot (g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0) \\
 &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0
 \end{aligned}$$

UNIT IV
Design Examples
(using VHDL)

UNIT IV DESIGN EXAMPLES (USING VHDL)

Barrel Shifter

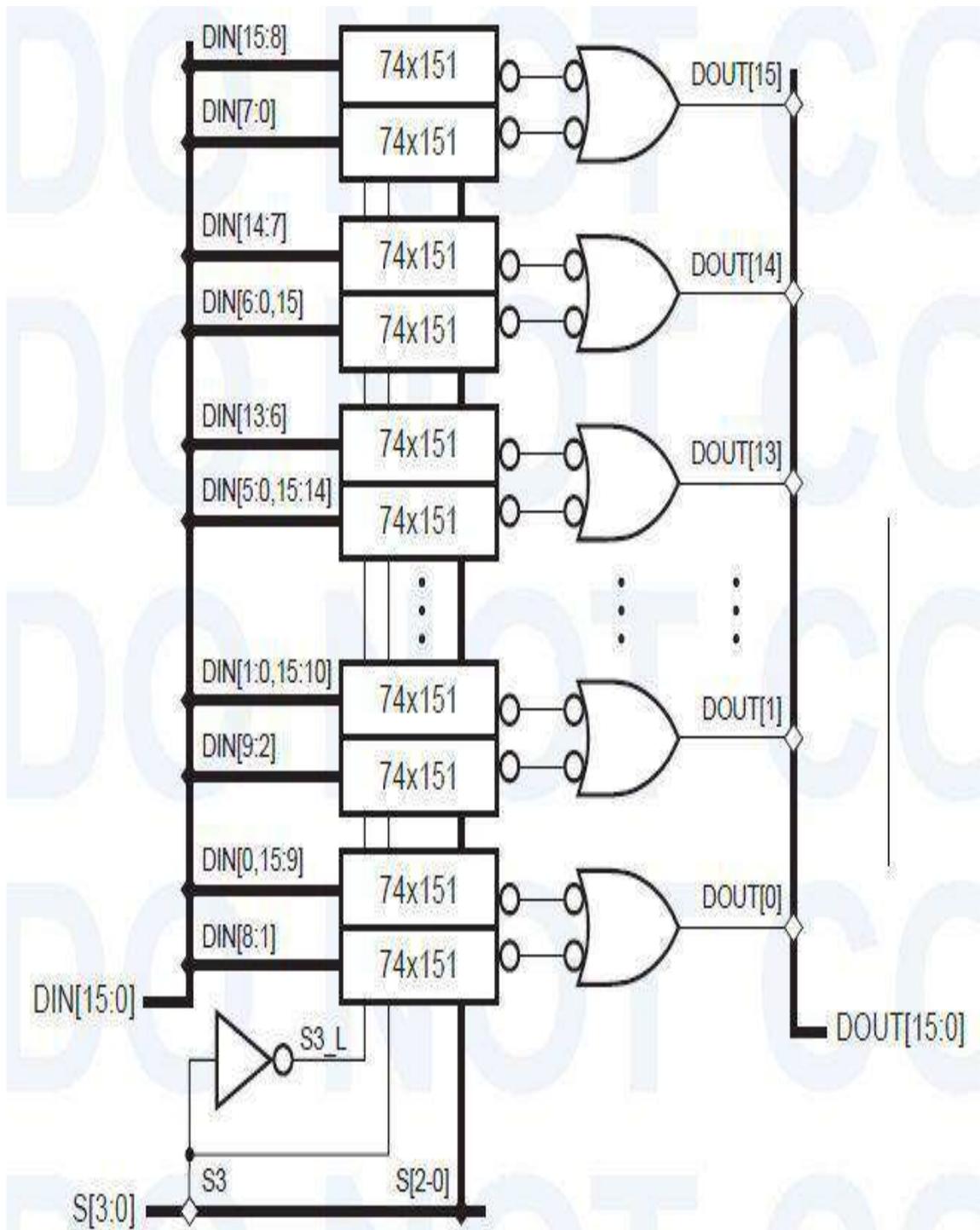
A *barrel shifter* is a combinational logic circuit with n data inputs, n data outputs, and a set of control inputs that specify how to shift the data between input and output. A barrel shifter that is part of a microprocessor CPU can typically specify the direction of shift (left or right), the type of shift (circular, arithmetic, or logical), and the amount of shift (typically 0 to $n-1$ bits, but sometimes 1 to n bits).

A simple 16-bit barrel shifter that does left circular shifts only, using a 4-bit control input $S[3:0]$ to specify the amount of shift. For example, if the input word is ABCDEFGHGIHKLMNOP (where each letter represents one bit), and the control input is 0101 (5), then the output word is FGHGIHKLMNOPABCDE. From one point of view, this problem is deceptively simple.

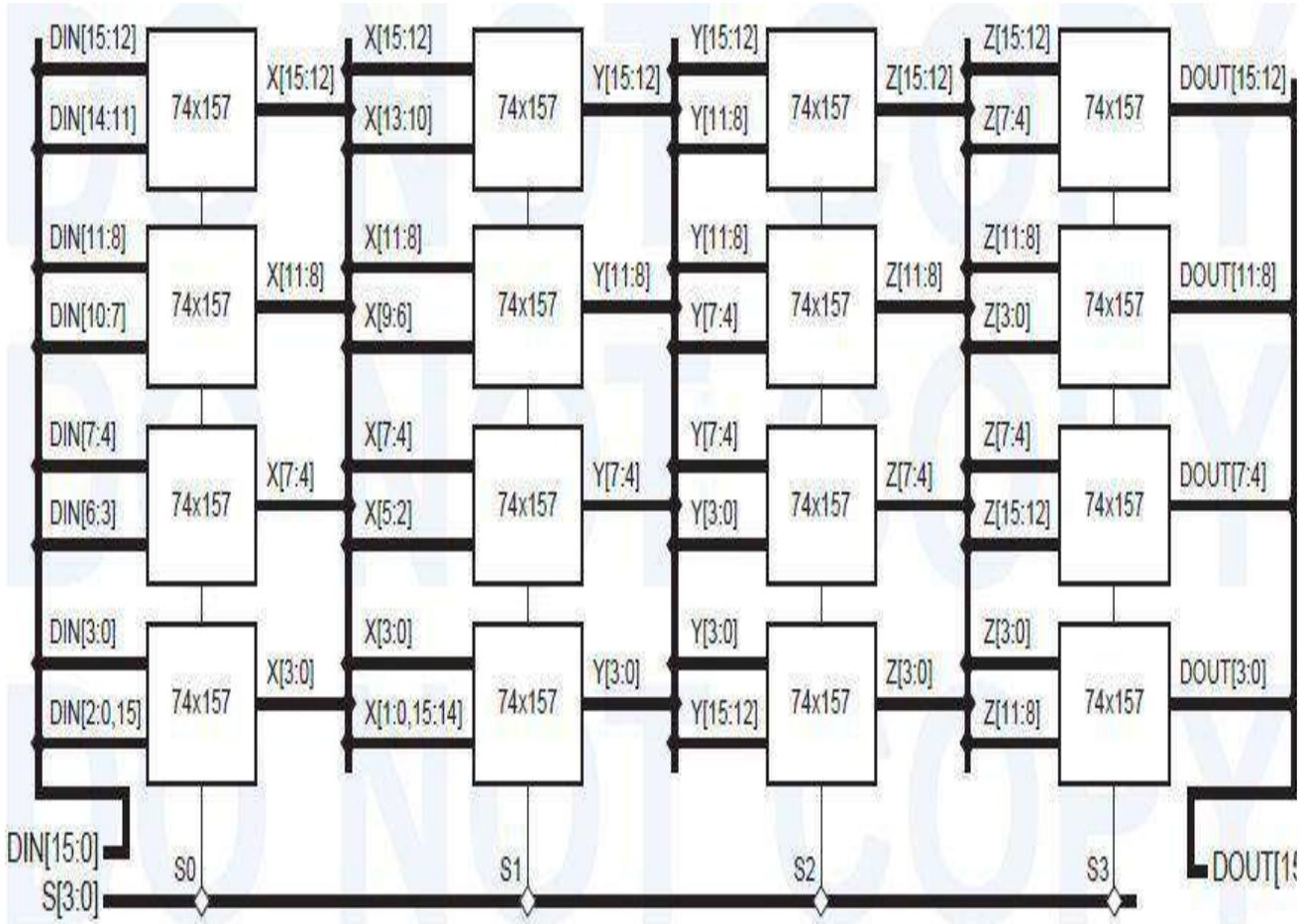
Each output bit can be obtained from a 16-input multiplexer controlled by the shift-control inputs, which each multiplexer data input connected to the appropriate On the other hand, when you look at the details of the design, you'll see that there are trade-offs in the speed and size of the circuit. Let us first consider a design that uses off-the-shelf MSI multiplexers.

A 16-input, one-bit multiplexer can be built using two 74x151s, by applying S_3 and its complement to the EN_L inputs and combining the Y_L data outputs with a NAND gate, as we showed in Figure 5-66 for a 32-input multiplexer. The lower-order shift-control inputs, S_2-S_0 , connect to the like-named select inputs of the '151s.

We complete the design by replicating this 16-input multiplexer 16 times and hooking up the data inputs appropriately, as shown in Figure 6-1. The top '151 of each pair is enabled by S_3_L , and the bottom one by S_3 ; the remaining select bits are connected to all 32 '151s. Data inputs D_0-D_7 of each '151 are connected to the DIN inputs in the listed order from left to right.



The '157-based approach requires only half as many MSI packages and has far less loading on the control and data inputs. On the other hand, it has the longest data-path delay, since each data bit must pass through four 74x157s. Halfway between the two approaches, we can use eight 74x153 4-input, 2-bit multiplexers two build a 4-input, 16-bit multiplexer. Cascading two sets of these, we can use $S[3:2]$ to shift selectively by 0, 4, 8, or 12 bits, and $S[1:0]$ to shift by 0–3 bits.



Simple Floating-Point Encoder

The previous example used multiple copies of a single building block, a multiplexer, and it was pretty obvious from the problem statement that a multiplexer was the appropriate building block. The next example shows that you sometimes have to look a little harder to see the solution in terms of known building blocks. Now let's look at a design problem whose MSI solution is not quite so obvious, a fixed-point to floating-point encoder. An unsigned binary integer B in the range $0 \leq B < 211$ can be represented by 11 bits in fixed-point format, $B = b_{10}b_9 \dots b_1b_0$. We can represent numbers in the same range with less precision using only 7 bits in a floating-point notation, $F = M \cdot 2^E$, where M is a 4-bit mantissa $m_3m_2m_1m_0$ and E is a 3-bit exponent $e_2e_1e_0$. The smallest integer in this format is 0 and the largest is $(2^4 - 1) \cdot 2^3 = 27$.

Given an 11-bit fixed-point integer B , we can convert it to our 7-bit floating-point notation by picking off four high-order bits beginning with the most significant 1, for example, The last term in each equation is a truncation error that results from the loss of precision in the conversion. Corresponding to this conversion operation, we can write the specification for a fixed-point to floating-point encoder circuit:

- A combinational circuit is to convert an 11-bit unsigned binary integer B into a 7-bit floating-point number M, E , where M and E have 4 and 3 bits, respectively. The numbers have the relationship $B = M \cdot 2^E \cdot T$, where T is the truncation error, $0 < T < 2E$.

Starting with a problem statement like the one above, it takes some creativity to come up with an efficient circuit design—the specification gives no clue. However, we can get some ideas by looking at how we converted numbers by hand earlier. We basically scanned each input number from left to right to find the first position containing a 1, stopping at the b_3 position if no 1 was found. We picked off four bits starting at that position to use as the mantissa, and the starting position number determined the exponent. These operations are beginning to sound like MSI building blocks.

–Scanning for the first 1 is what a generic priority encoder does. The output of the priority encoder is a number that tells us the position of the first 1. The position number determines the exponent; first-1 positions of $b_{10} b_3$ imply exponents of 7–0, and positions of $b_2 b_0$ or no-1-found imply an exponent of 0. Therefore, we can scan for the first 1 with an 8-input priority encoder with inputs

| | |
|----------------|-----------|
| 11010110100101 | 7 0110100 |
| 00100101111001 | 5 01111 |
| 00000111110111 | 2 10 |
| 00000001011011 | 0 0 |
| 00000000010010 | 0 0 |

I7 (highest priority) through I0 connected to $b_{10} b_3$. We can use the priority encoder's A2 – A0 outputs directly as the exponent, as long as the no-1-found case produces A2 – A0 000.

–Picking off four bits sounds like a –selecting or multiplexing operation.

The 3-bit exponent determines which four bits of B we pick off, so we can use the exponent bits to control an 8-input, 4-bit multiplexer that selects the appropriate four bits of B to form M .

- Since the available MSI priority encoder, the 74x148, has active-low inputs, the input number B is assumed to be available on an active-low bus $B_L[10:0]$. If only an active-high version of B is available, then eight inverters can be used to obtain the active-low version.
- If you think about the conversion operation a while, you'll realize that the most significant bit of the mantissa, m_3 , is always 1, except in the no-1-found case. The '148 has a GS_L output that indicates this case, allowing us to eliminate the multiplexer for m_3 .
- The '148 has active-low outputs, so the exponent bits ($E0_L$ – $E2_L$) are produced in active-low form. Naturally, three inverters could be used to produce an active-high version.
- Since everything else is active-low, active-low mantissa bits are used too. Active-high bits are also readily available on the '148 EO_L and the '151 Y_L outputs.

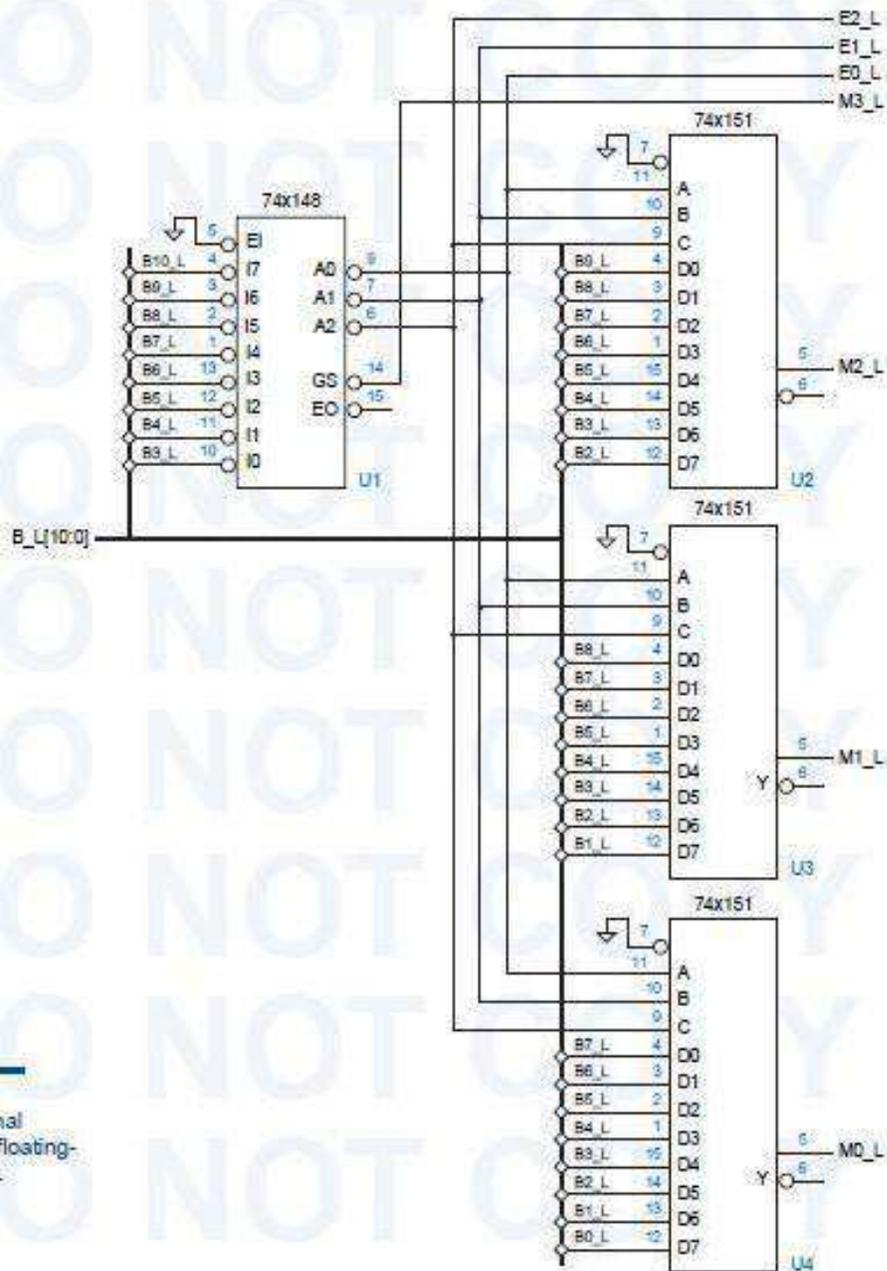


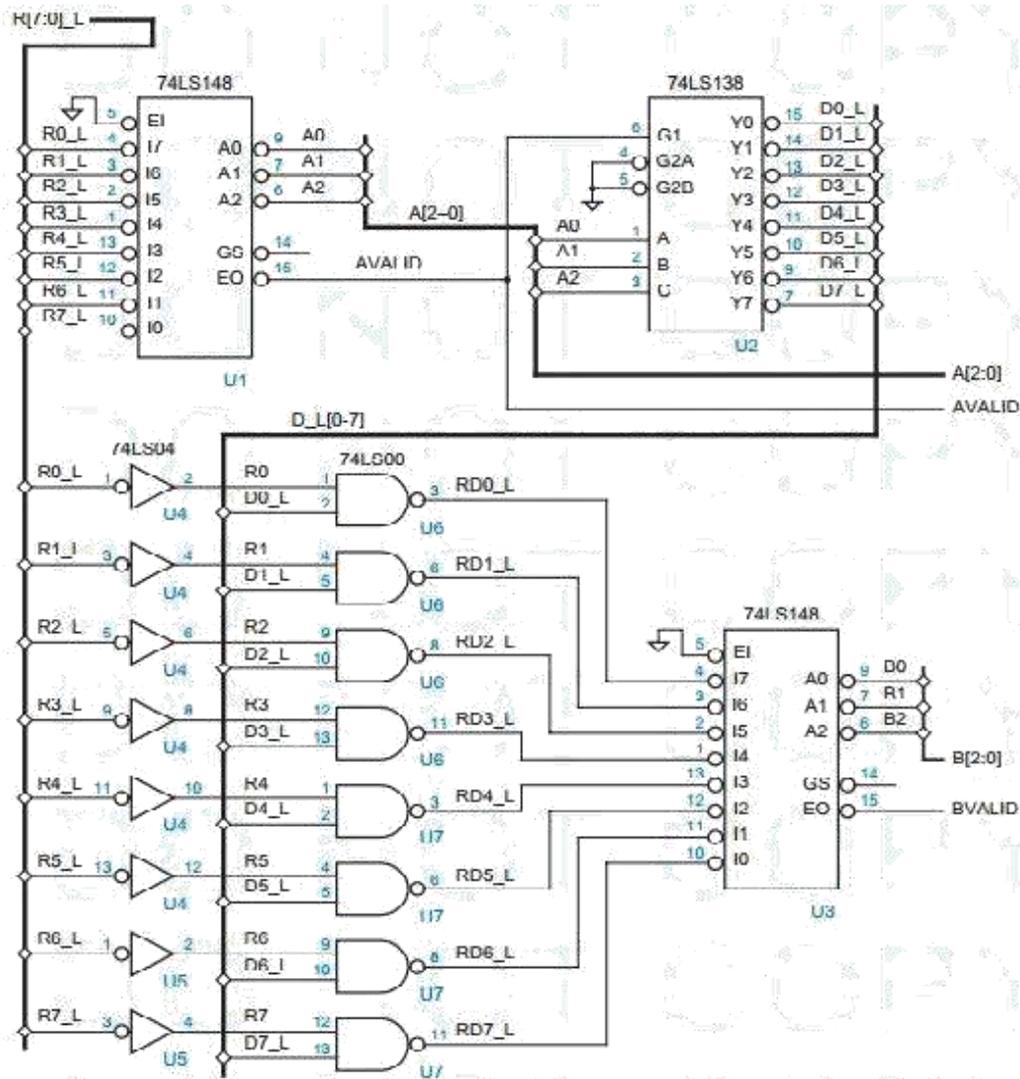
Figure 6-3
A combinational
fixed-point to floating-
point encoder.

Dual-Priority Encoder

Quite often MSI building blocks need a little help from their friends—ordinary gates—to get the job done. In this example, we'd like to build a priority encoder that identifies not only the highest but also the second-highest priority asserted signal among a set of eight request inputs. We'll assume for this example that the request inputs are active low and are named R0_L–R7_L, where R0_L has the highest priority. We'll use A2–A0 and AVALID to identify the highest-priority request, where AVALID is asserted only if at least one request

input is asserted. We'll use B2–B0 and BVALID to identify the second-highest-priority request, where BVALID is asserted only if at least two request inputs are asserted.

Finding the highest-priority request is easy enough, we can just use a 74x148. To find the second highest-priority request, we can use another '148, but only if we first knock out the highest-priority request before applying the request inputs. This can be done using a decoder to select a signal to knock out, based on A2–A0 and AVALID from the first '148. These ideas are combined in the solution shown in Figure . A 74x138 decoder asserts at most one of its eight outputs, corresponding to the highest-priority request input. The outputs are fed to a rank of NAND gates to turn off the highest-priority request. A trick is used in this solution is to get active-high outputs from the '148s. We can rename the address outputs A2_L–A0_L to be active high if we also change the name of the request input that is associated with each output combination. In particular, we complement the bits of the request number. In the redrawn symbol, request input I0 has the highest priority.



Cascading Comparators

Since the 74x85 uses a serial cascading scheme, it can be used to build arbitrarily large comparators. The 74x682 8-bit comparator, on the other hand, doesn't have any cascading inputs and outputs at all. Thus, at first glance, you might think that it can't be used to build larger comparators. But that's not true. If you think about the nature of a large comparison, it is clear that two wide inputs, say 32 bits (four bytes) each, are equal only if their corresponding bytes are equal. If we're trying to do a greater-than or less-than comparison, then the corresponding most-significant that are not equal determine the result of the comparison.

Using these ideas, Figure uses three 74x682 8-bit comparators to do equality and greater-than comparison on two 24-bit operands. The 24-bit results are derived from the individual 8-bit results using combinational logic for the following equations:

$$\begin{aligned} \text{PEQ} &= \text{EQ2} \text{ EQ1} \text{ EQ0} \\ \text{PGT} &= \text{GT2} \text{ EQ2} \text{ EQ1} \text{ GT1} \text{ EQ1} \text{ EQ0} \end{aligned}$$

This parallel expansion approach is actually faster than the 74x85's serial cascading scheme, because it does not suffer the delay of propagating the cascading signals through a cascade of comparators. The parallel approach can be used to build very wide comparators using two-level AND-OR logic to combine the 8-bit results, limited only by the fan-in constraints of the AND-OR logic. Arbitrary large comparators can be made if you use additional levels of logic to do the combining.

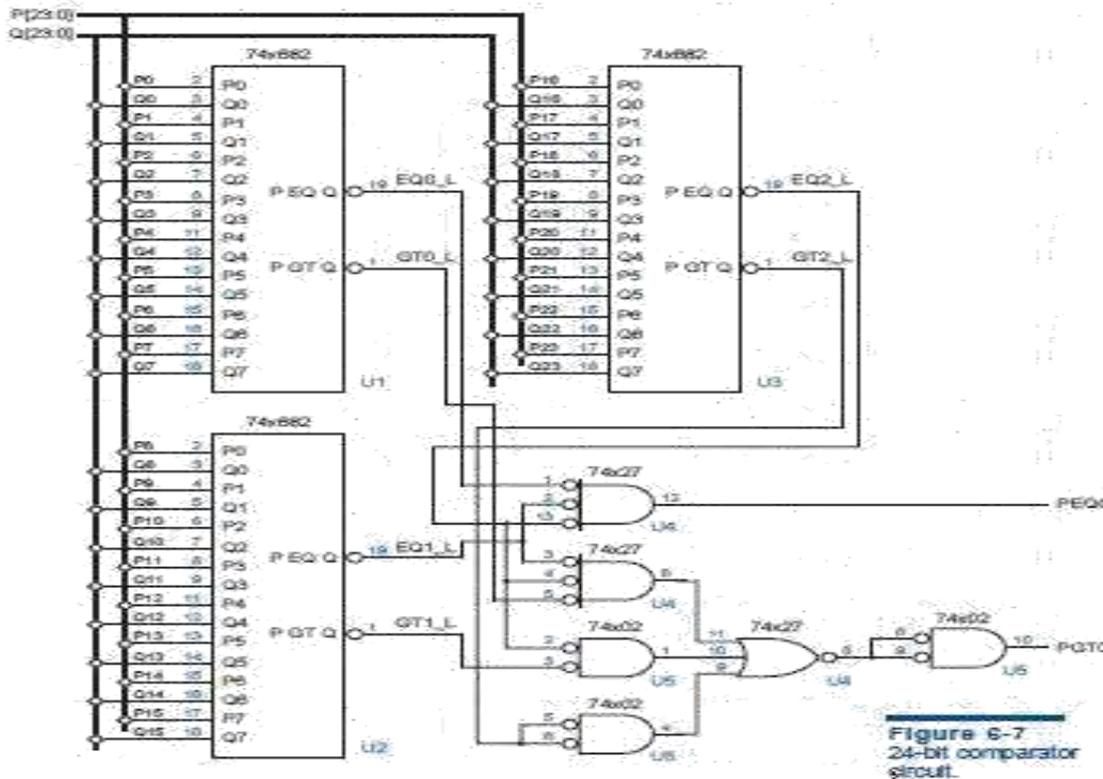


Figure 6-7
24-bit comparator circuit.

(SEQUENTIAL LOGIC DESIGN)

Basic Bistable Element

A combinational system is a system whose outputs depends only upon its current inputs.

A sequential system is a system whose output depends on current input *and* past history of inputs.

All systems we have looked at to date have been combinational systems.

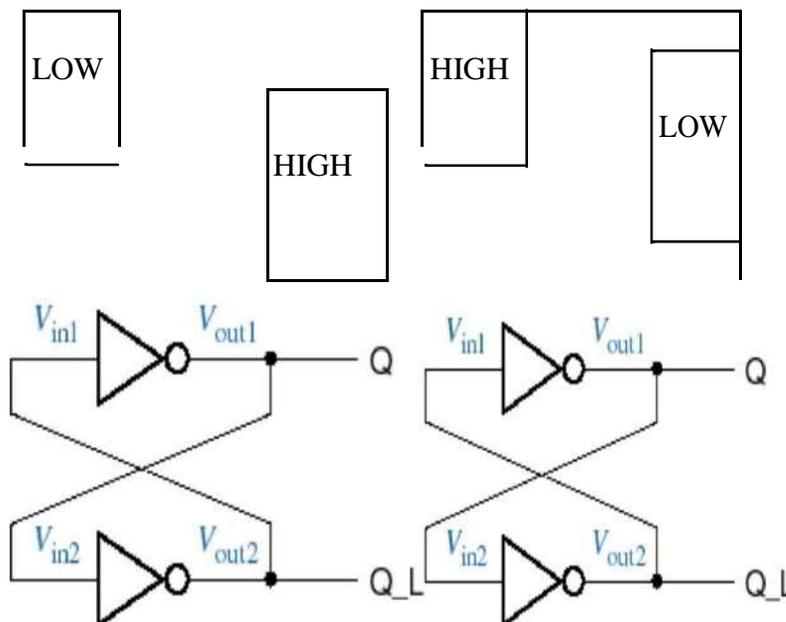
Outputs depends on the current inputs and the system's current state.

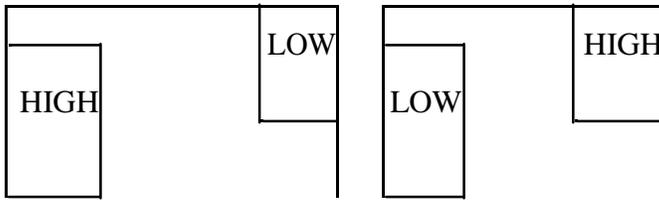
State embodies all the information about the past needed to predict current output based on current input.

- *State variables*, one or more bits of information.

The state is a collection of state variables whose values at any one time contain all the information about the past necessary to account for the circuit's future behavior.

- The simplest sequential circuit, no way to control its state.
- Two states
 - One state variable, say, Q, two possible states





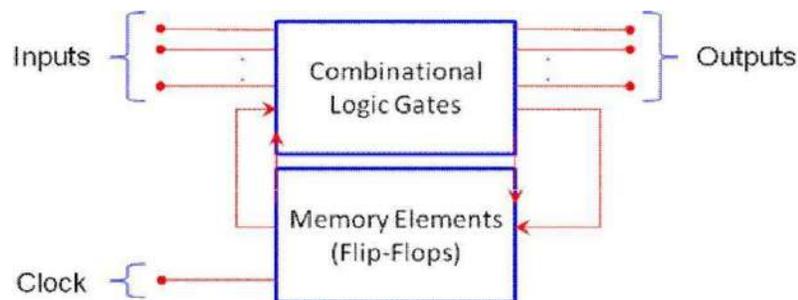
- How to control it?
 - Control inputs

A *bistable memory device* is the generic term for the elements we are studying.

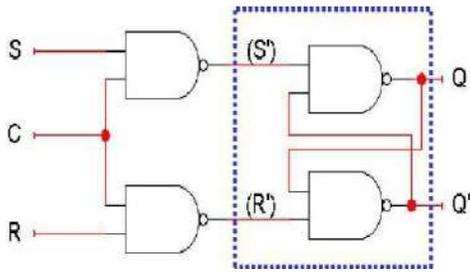
Latches and Flip flops

Latches and flip-flops (FFs) are the basic building blocks of sequential circuits.

- latch: bistable memory device with level sensitive triggering (no clock), watches all of its inputs continuously and changes its outputs at any time, independent of a clocking signal.
- flip-flop: bistable memory device with edge-triggering (with clock), samples its inputs, and changes its output only at times determined by a clocking signal.



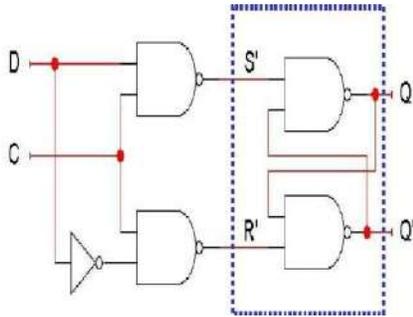
- Here is an SR latch with a control input C.
- Notice the hierarchical design!
 - The dotted blue box is the S'R' latch.
 - The additional NAND gates are simply used to generate the correct inputs for the S'R' latch.
- The control input acts just like an enable.



| | | | | Q |
|---|---|---|---|-----------|
| 0 | | 1 | 1 | No change |
| 1 | 0 | 0 | 1 | No change |
| 1 | 0 | 1 | 0 | 0 (reset) |
| 1 | 1 | 0 | 1 | 1 (set) |
| 1 | 1 | 1 | 0 | Avoid! |

D latch

- Finally, a D latch is based on an S'R' latch. The additional gates generate the S' and R' signals, based on inputs D (-data) and C (-control).
 - When C = 0, S' and R' are both 1, so the state Q does not change.
 - When C = 1, the latch output Q will equal the input D.
- No more messing with one input for set and another input for reset!

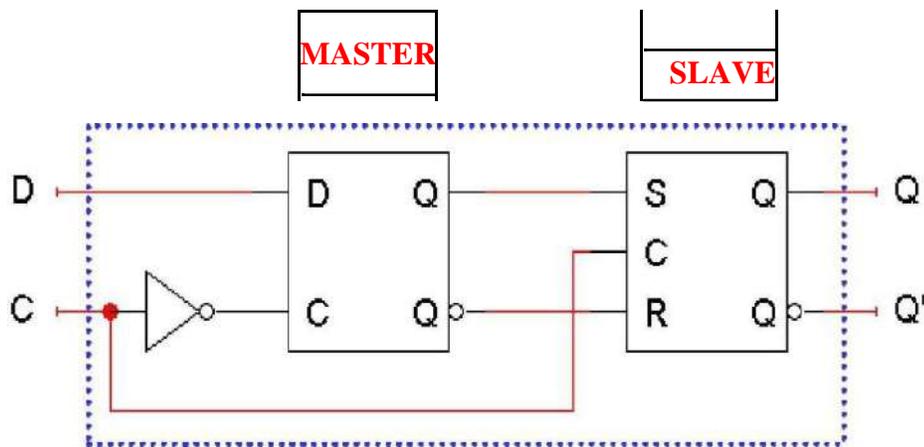


| | | Q |
|---|--|-----------|
| 0 | | No change |
| 1 | | 0 |
| 1 | | 1 |

- Also, this latch has no -bad! input combinations to avoid. Any of the four possible assignments to C and D are valid.

Flip-flops

- Here is the internal structure of a D flip-flop.
 - The flip-flop inputs are C and D, and the outputs are Q and Q'.
 - The D latch on the left is the master, while the SR latch on the right is called the slave.
- Note the layout here.
 - The flip-flop input D is connected directly to the master latch.
 - The master latch output goes to the slave.
 - The flip-flop outputs come directly from the slave latch.



D flip-flops when $C=0$

- The D flip-flop's control input C enables *either* the D latch or the SR latch, but not both.
- When $C = 0$:
 - The master latch is enabled, and it monitors the flip-flop input D . Whenever D changes, the master's output changes too.
 - The slave is disabled, so the D latch output has no effect on it. Thus, the slave just maintains the flip-flop's current state.

D flip-flops when $C=1$

- As soon as C becomes 1,
 - The master is disabled. Its output will be the *last* D input value seen just before C became 1.
 - Any subsequent changes to the D input while $C = 1$ have no effect on the master latch, which is now disabled.
 - The slave latch is enabled. Its state changes to reflect the master's output, which again is the D input value from right when C became 1.

Positive edge triggering

- This is called a positive edge-triggered flip-flop.
 - The flip-flop output Q changes *only* after the positive edge of C .

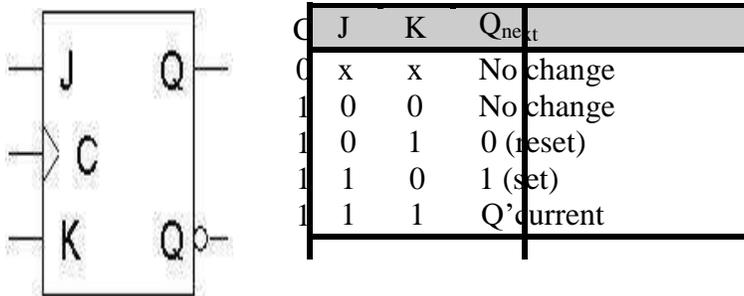
- The change is based on the flip-flop input values that were present right at the positive edge of the clock signal.

The D flip-flop's behavior is similar to that of a D latch except for the positive edge-triggered nature, which is not explicit in this table

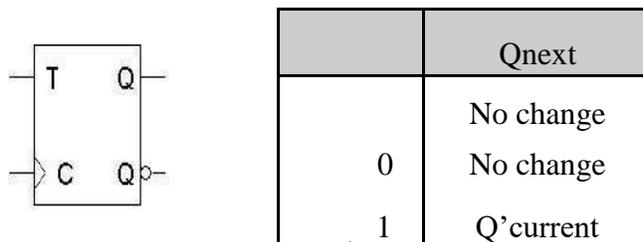
| | Q |
|--|-----------|
| | No change |
| | 0 (reset) |
| | 1 (set) |

Flip-flop variations

- We can make different versions of flip-flops based on the D flip-flop, just like we made different latches based on the S'R' latch.
- A JK flip-flop has inputs that act like S and R, but the inputs JK=11 are used to *complement* the flip-flop's current state.



A T flip-flop can only maintain or complement its current state



Characteristic equations

- We can also write characteristic equations, where the next state $Q(t+1)$ is defined in terms of the current state $Q(t)$ and inputs.

| D | $Q(t+1)$ | Operation |
|---|----------|-----------|
| 0 | 0 | Reset |
| 1 | 1 | Set |

$$Q(t+1) = D$$

| J | K | $Q(t+1)$ | Operation |
|---|---|----------|------------|
| 0 | 0 | $Q(t)$ | No change |
| 0 | 1 | 0 | Reset |
| 1 | 0 | 1 | Set |
| 1 | 1 | $Q'(t)$ | Complement |

$$Q(t+1) = K'Q(t) + JQ'(t)$$

| T | $Q(t+1)$ | Operation |
|---|----------|------------|
| 0 | $Q(t)$ | No change |
| 1 | $Q'(t)$ | Complement |

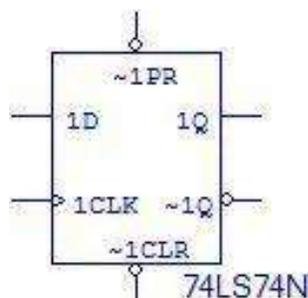
$$Q(t+1) = T'Q(t) + TQ'(t)$$

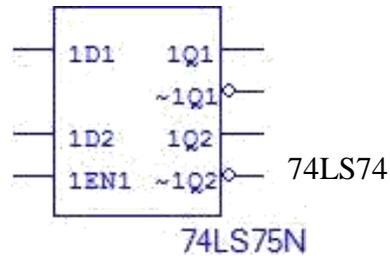
$$= T \oplus Q(t)$$

Flip-Flop Vs. Latch

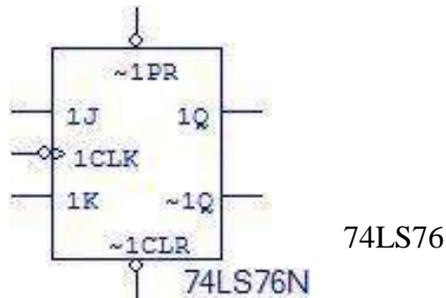
- The primary difference between a D flip-flop and D latch is the EN/CLOCK input.
- The flip-flop's CLOCK input is edge sensitive, meaning the flip-flop's output changes on the edge (rising or falling) of the CLOCK input.
- The latch's EN input is level sensitive, meaning the latch's output changes on the level (high or low) of the EN input.

Flip-Flops & Latches





Dual Positive-Edge-Triggered D Flip-Flops with Preset, Clear, and Complementary Outputs



Dual Negative-Edge-Triggered J-K Flip-Flops with Preset, Clear, and Complementary Outputs 74LS75 Quad Latch

74LS74: D Flip-Flop

Function Table

| Inputs | | | | Outputs | |
|--------|-----|-----|---|------------|-------------|
| PR | CLR | CLK | D | Q | \bar{Q} |
| L | H | X | X | H | L |
| H | L | X | X | L | H |
| L | L | X | X | H (Note 1) | H (Note 1) |
| H | H | ↑ | H | H | L |
| H | H | ↑ | L | L | H |
| H | H | L | X | Q_0 | \bar{Q}_0 |

H = HIGH Logic Level

X = Either LOW or HIGH Logic Level

L = LOW Logic Level

↑ = Positive-going Transition

Q_0 = The output logic level of Q before the indicated input conditions were established.

Note 1: This configuration is nonstable; that is, it will not persist when either the preset and/or clear inputs return to their inactive (HIGH) level.

74LS76: J/K Flip-Flop

Function Table

| Inputs | | | | | Outputs | |
|--------|-----|---|---|---|-------------------|-------------------------|
| PR | CLR | CLK | J | K | Q | \bar{Q} |
| L | H | X | X | X | H | L |
| H | L | X | X | X | L | H |
| L | L | X | X | X | H | H |
| H | H |  | L | L | (Note 1) Q_0 | (Note 1) \bar{Q}_0 |
| H | H |  | H | L | H | L |
| H | H |  | L | H | L | H |
| H | H |  | H | H | Toggle | |

H = High Logic Level

L = Low Logic Level

X = Either Low or High Logic Level

 = Positive pulse data. The J and K inputs must be held constant while the clock is high. Data is transferred to the outputs on the falling edge of the clock pulse.

Q_0 = The output logic level before the indicated input conditions were established.

Toggle = Each output changes to the complement of its previous level on each complete active high level clock pulse.

Note 1: This configuration is nonstable; that is, it will not persist when the preset and/or clear inputs return to their inactive (high) level.

74LS75: D Latch

Function Table (Each Latch)

| Inputs | | Outputs | |
|--------|--------|---------|-------------|
| D | Enable | Q | \bar{Q} |
| L | H | L | H |
| H | H | H | L |
| X | L | Q_0 | \bar{Q}_0 |

H = HIGH Level

L = LOW Level

X = Don't Care

Q_0 = The Level of Q Before the HIGH-to-LOW Transition of ENABLE

Counters

- Counters are a specific type of sequential circuit.
- Like registers, the state, or the flip-flop values themselves, serves as the output.
- The output value increases by one on each clock cycle.
- After the largest value, the output wraps around back to 0.
- Using two bits, we'd get something like this:

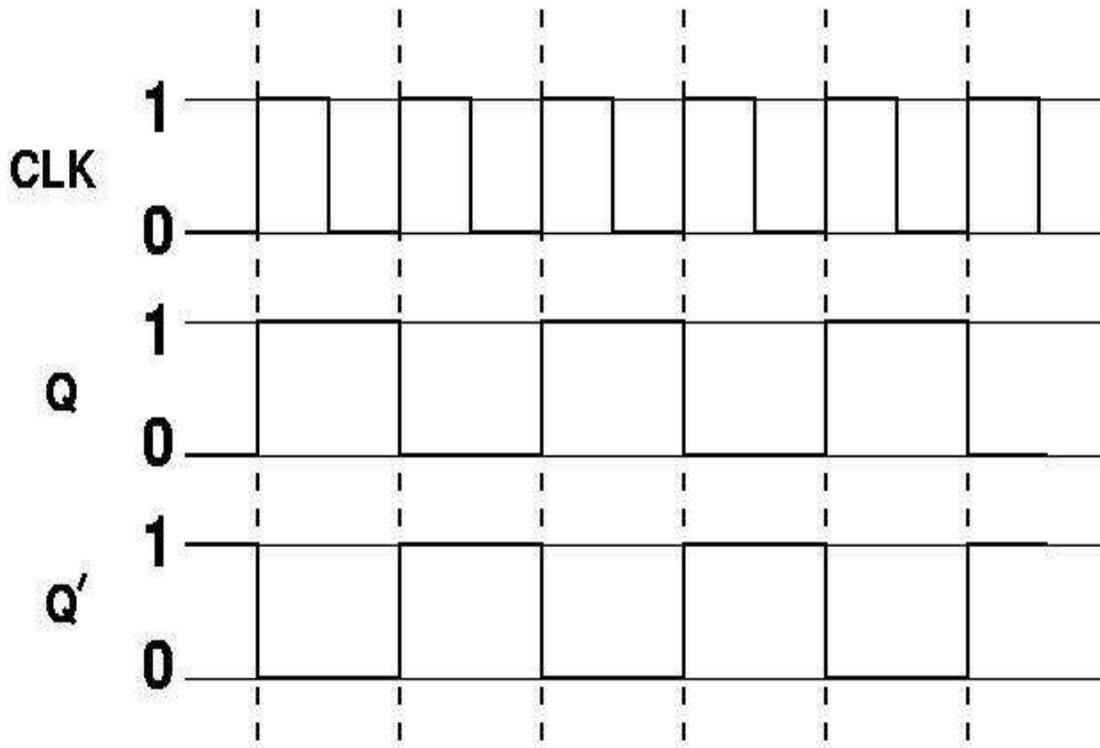
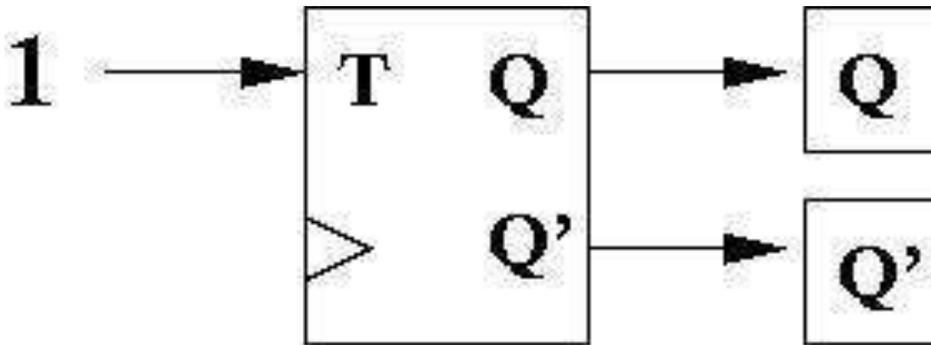
| Present State | | Next State | |
|---------------|---|------------|---|
| A | B | A | B |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |

- Counters can act as simple clocks to keep track of time.
- You may need to record how many times something has happened.
 - How many bits have been sent or received?
 - How many steps have been performed in some computation?
- All processors contain a program counter, or PC.
 - Programs consist of a list of instructions that are to be executed one after another (for the most part).
 - The PC keeps track of the instruction currently being executed.
 - The PC increments once on each clock cycle, and the next program instruction is then executed.

Asynchronous Counters

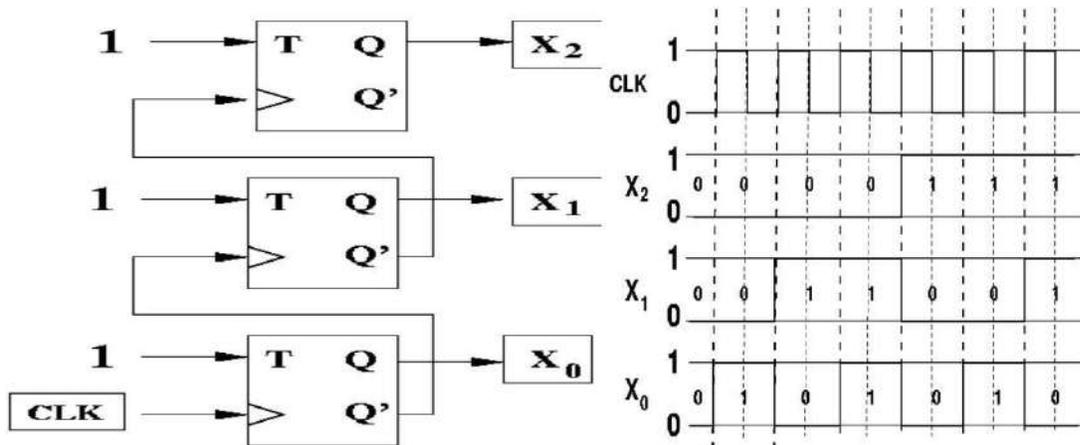
- This counter is called *asynchronous* because not all flip flops are hooked to the same clock.
- Look at the waveform of the output, **Q**, in the timing diagram. It resembles a clock as well. If the period of the clock is T , then what is the period of **Q**, the output of the flip flop? It's $2T$!

- We have a way to create a clock that runs twice as slow. We feed the clock into a T flip flop, where T is hardwired to 1. The output will be a clock who's period is twice as long.
- If the clock has period T. **Q0** has period 2T. **Q1** period is 4T
- With n flip flops the period is 2^n



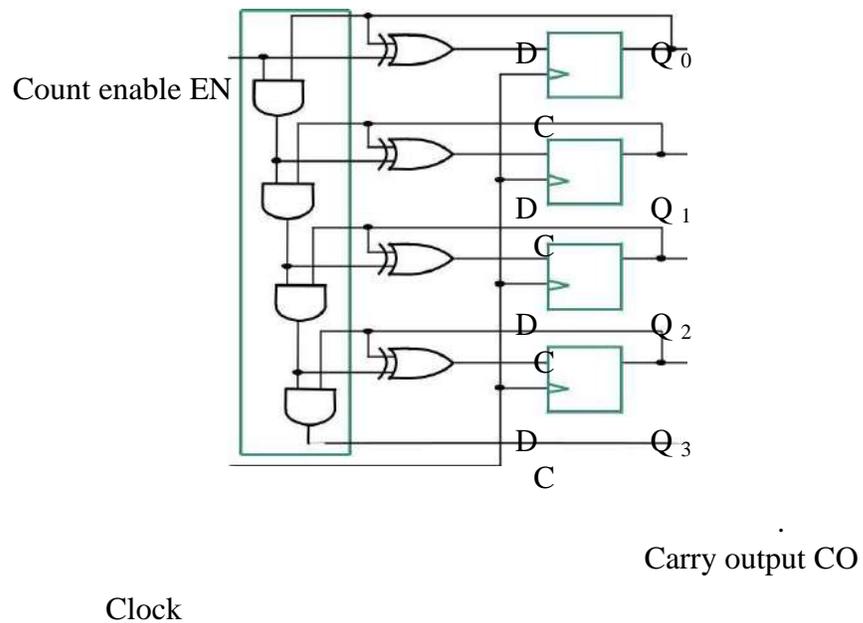
3 bit asynchronous "ripple" counter using T flip flops

This is called as a *ripple counter* due to the way the FFs respond one after another in a kind of rippling effect.



Synchronous Counters

- To eliminate the "ripple" effects, use a common clock for each flip-flop and a combinational circuit to generate the next state.
- For an up-counter, use an incrementer =>
- Internal details =>



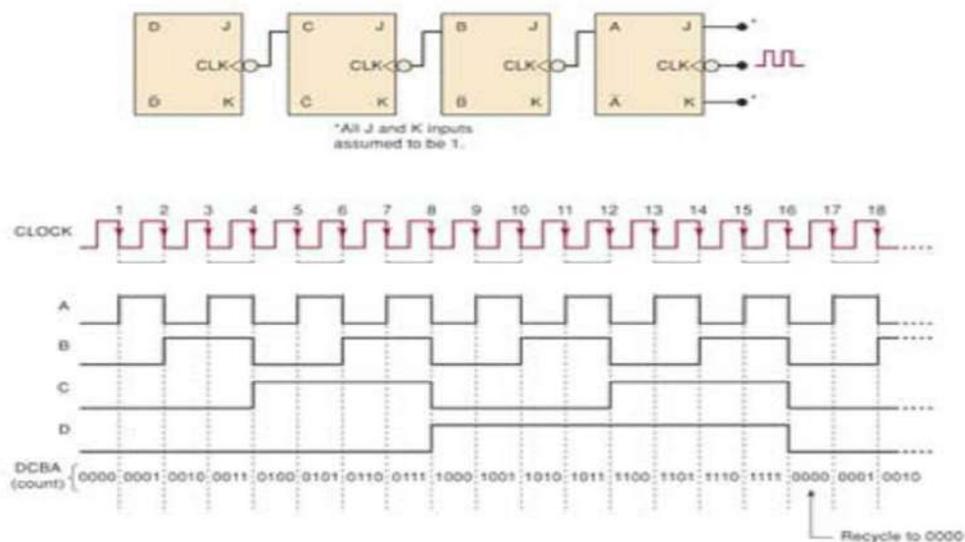
- Internal Logic
 - (a) Logic Diagram-Serial Gating
- XOR complements each bit
- AND chain causes complement of a bit if all bits toward LSB from it equal 1
- Count Enable
- Forces all outputs of AND chain to 0 to —hold the state
- Carry Out
- Added as part of incrementer
- Connect to Count Enable of additional 4-bit counters to form larger counters

Design of Counters using digital ICs

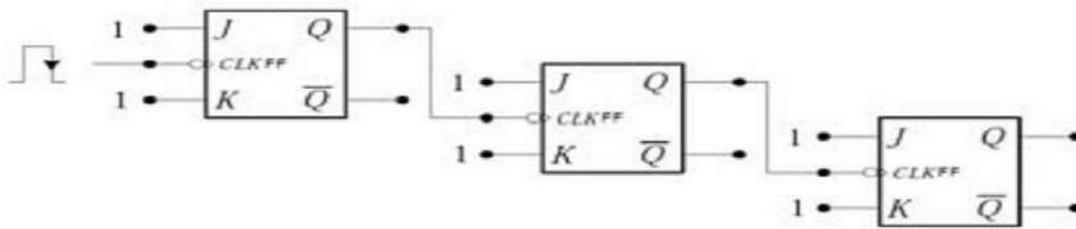
Asynchronous (Ripple) Counters

Clock is applied only to FF A. J and K are high in all FFs to toggle on every clock pulse. Output of FF A is CLK of FF B and so forth. FF outputs D, C, B, and A are a 4 bit binary number with D as the MSB. After the negative transition of the 15th clock pulse the counter recycles to 0000. This is an asynchronous counter because state is not changed in exact synchronism with the clock.

Four-bit asynchronous (ripple) counter



Frequency division
 The output frequency of each FF = the clock frequency of input / 2.
 The output frequency of the last FF = the clock frequency / MOD



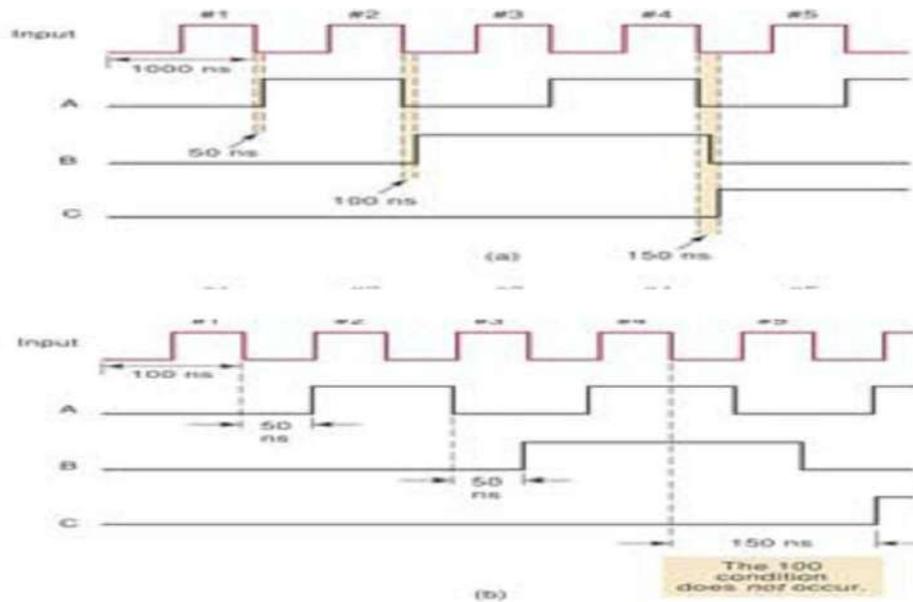
Propagation Delay in Ripple Counters

Ripple counters are simple, but the cumulative propagation delay can cause problems at high

Frequencies. For proper operation the following apply:

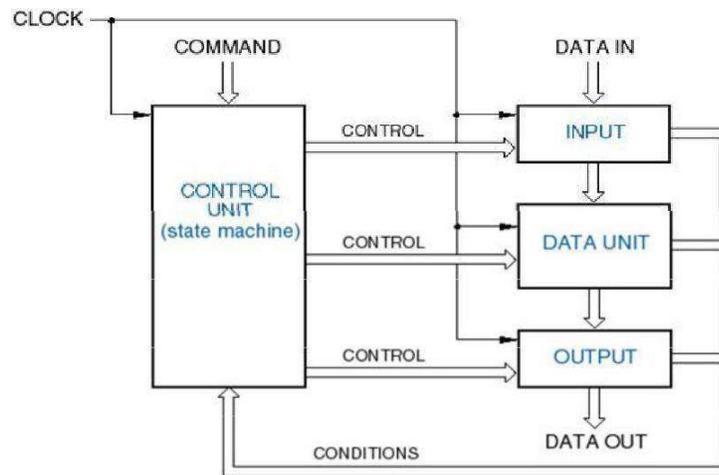
$$T_{\text{clock}} \geq N \times t_{pd}$$

$$F_{\text{max}} = 1 / (N \times t_{pd})$$



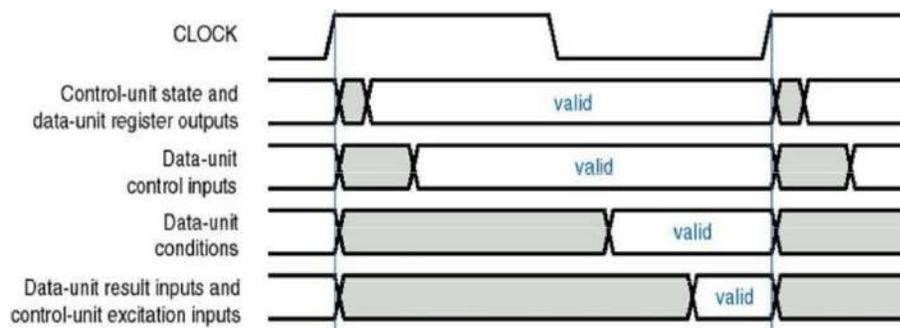
Synchronous design methodology

Synchronous System Structure



Everything is clocked by the same, common clock

Typical synchronous-system timing



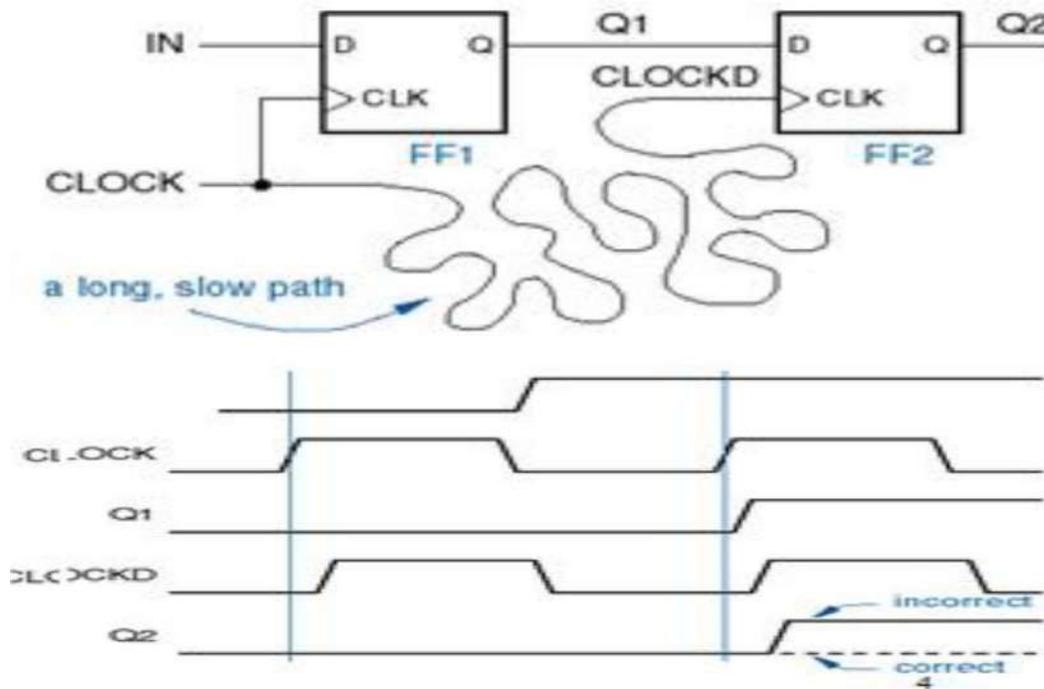
- Outputs have one complete clock period to propagate to inputs.
- Must take into account flip-flop setup times at next clock period.

Clock Skew

Clock signal may not reach all flip-flops simultaneously. Output changes of flip-flops receiving -early| clock may reach D inputs of flip-flops with -late| clock too soon.

Reasons for slowness:

- (a) wiring delays
- (b) capacitance
- (c) incorrect design



Clock-skew calculation

$$t_{ffpd}(\min) + t_{comb}(\min) - t_{hold} - t_{skew}(\max) > 0$$

First two terms are minimum time after clock edge that a D input changes

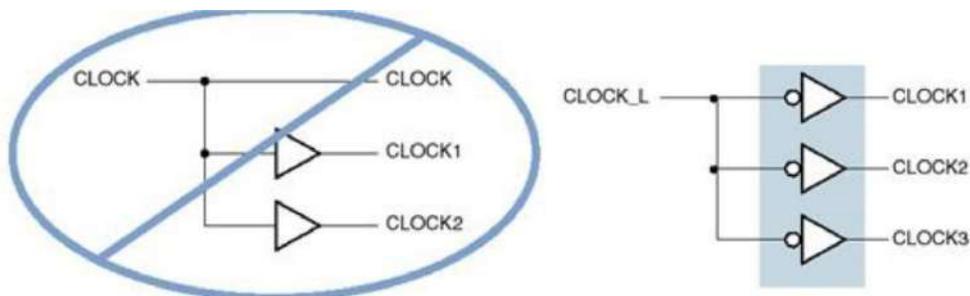
Hold time is earliest time that the input may change

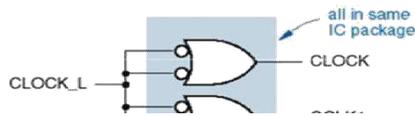
Clock skew subtracts from the available hold-time margin

Compensating for clock skew:

- Longer flip-flop propagation delay
- Explicit combinational delays
- Shorter (even negative) flip-flop hold times

Example of bad clock distribution

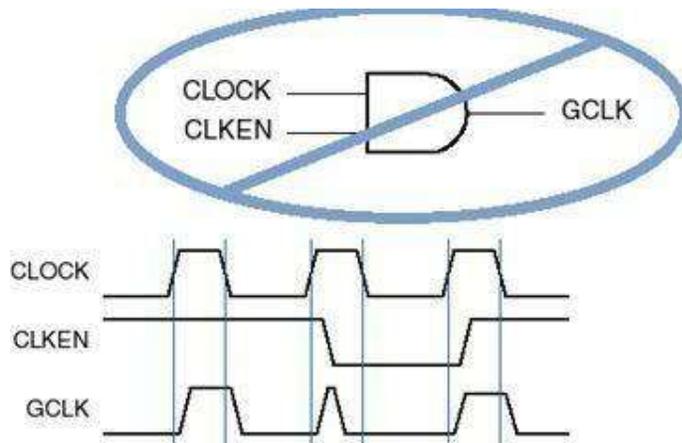




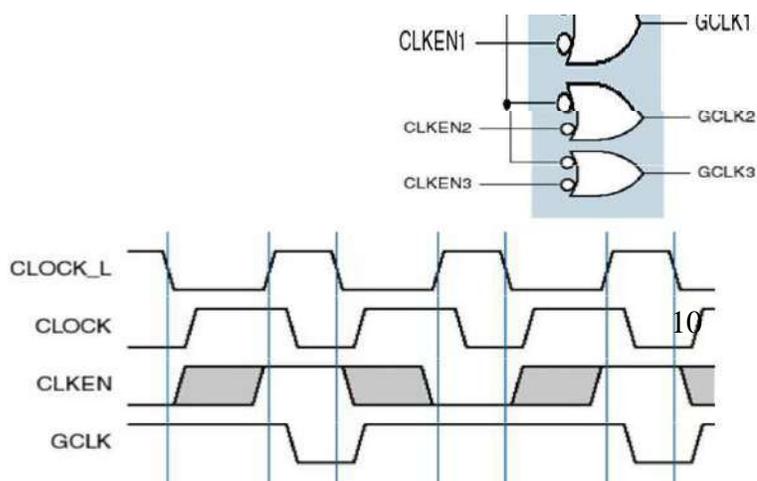
Gating the clock

Definitely a no-no

- Glitches possible if control signal (CLKEN) is generated by the same clock
- Excessive clock skew in any case.



If you really must gate the clock...



Asynchronous inputs

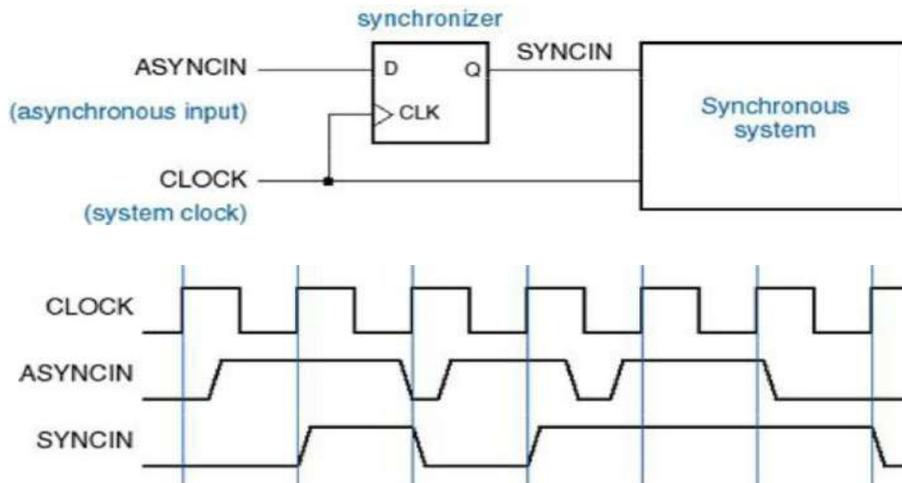
Not all inputs are synchronized with the clock

Examples:

- Keystrokes
- Sensor inputs
- Data received from a network (transmitter has its own clock)

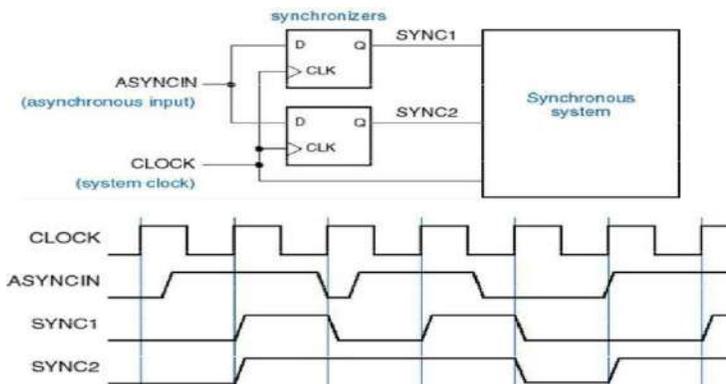
Inputs must be synchronized with the system clock before being applied to a synchronous system.

A simple synchronizer



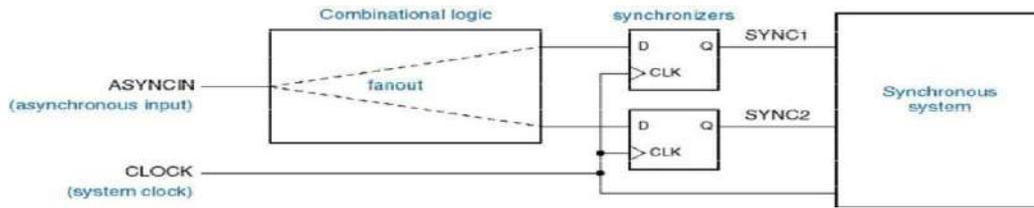
12

Only **one** synchronizer per input



13

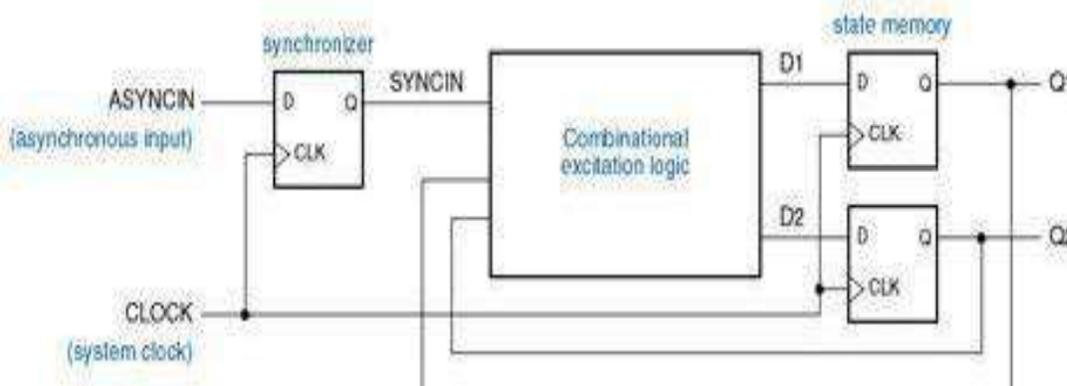
Even worse



- Combinational delays to the two synchronizers are likely to be different.

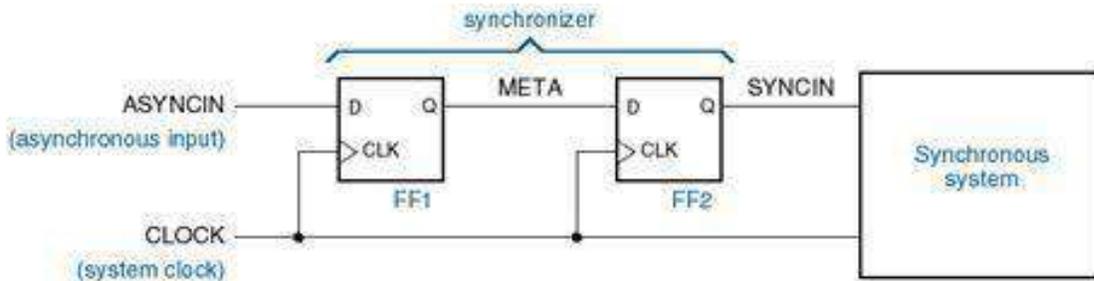
way to do it

One synchronizer per input. Carefully locate the synchronization points in a sys. But still a problem -- the synchronizer output may become metastable when setup and hold time are not met.



Recommended synchronizer design

Hope that FF1 settles down before -META| is sampled. In this case, -SYNCIN| is valid for almost a full clock period. Can calculate the probability of -synchronizer failure| (FF1 still metastable when META sampled).



Impediments to synchronous design

- Clock skew
 - Definition

The difference between arrival times of the clock at different memory devices

Example of clock skew

Influence of clock skew

Reduce the setup and hold time margins. For proper operation

$$t_{ffpd}(\min) + t_{comb}(\min) - t_{hold} - t_{skew}(\max) > 0$$

$$t_{setup} - t_{clk} - t_{ffpd}(\max) - t_{comb}(\max) - t_{skew}(\max) > 0$$

Reducing clock skew

proper buffering the clock Better clock

distribution .

Gating clock

Why not to gate the clock .

An acceptable way.

Asynchronous inputs

Why use the asynchronous inputs?

Problem with asynchronous inputs

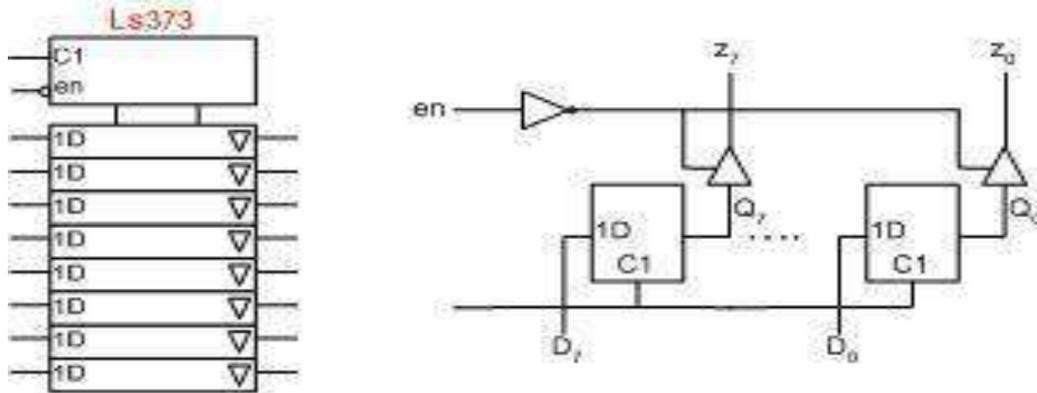
Meta-stable

Need synchronizers

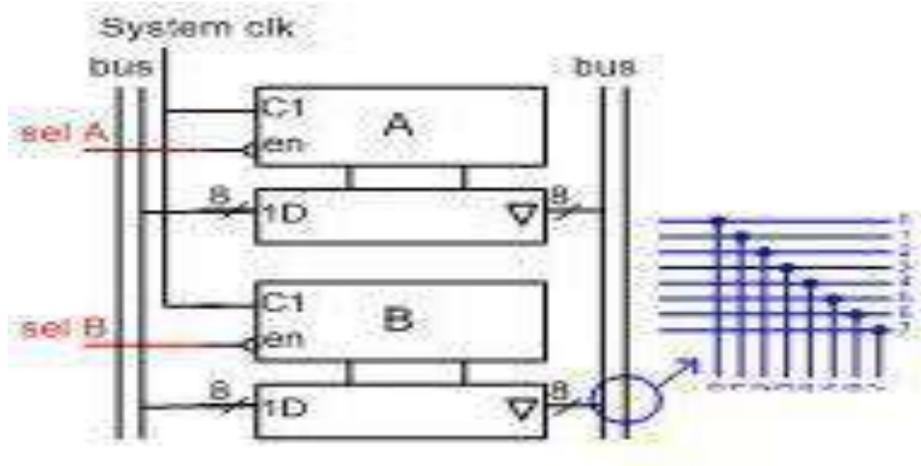
A simple one.

MSI Registers

We will be discussing the 7400 register series which is a rather popular series of registers. 74ls373: This register is made up of 8 latches and to have a clock enable the following structure is used:

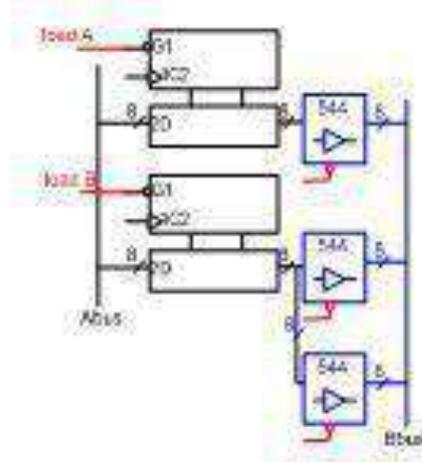
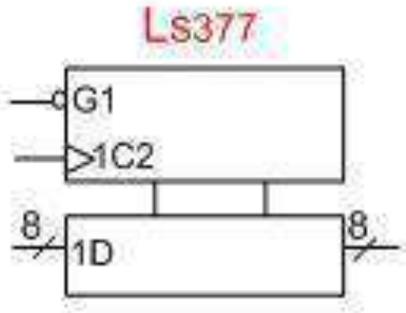


When the enable signal is high although clocking is done, the values of the latches won't appear on the output lines. The following figure shows us how this enable line can be used to select which one of the numerous registers' value is to be set on the output bus:



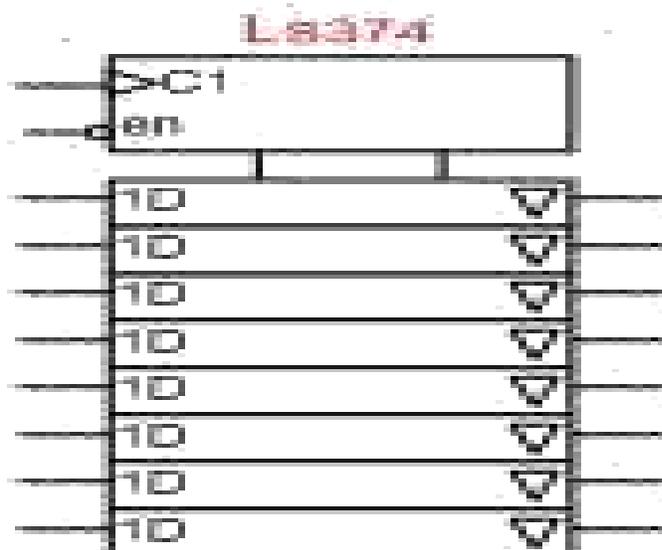
Quote: Sometimes although a register may be able to store for instance 32 bits, only 16 communication lines are used that will load or send out data in two clock pulses.

74ls377: This register can be used where a bus is linked to the input of more than one register and we want to be able to choose into which one, data will be loaded. Here we have lost the ability to bus the outputs. In order to be able to bus the outputs we need to use three state packages.

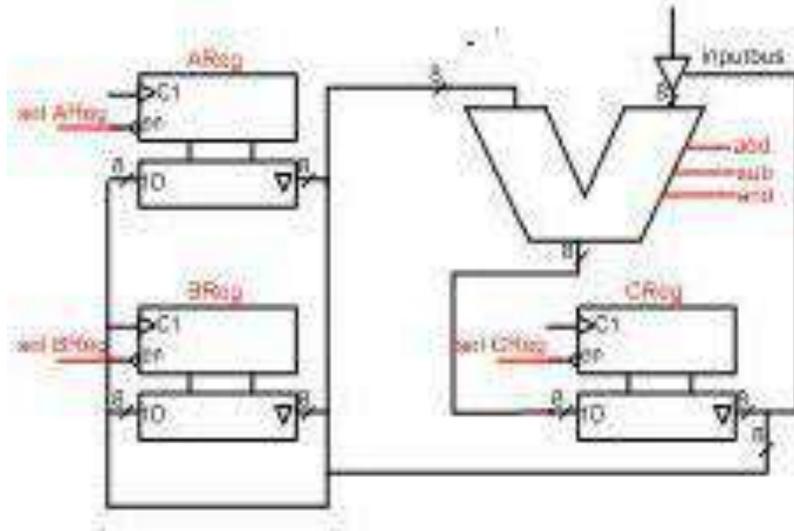


The reason we have either load control on a register package or three state control or not both at the same time is that packages have a standard number of pins and thus both can't be on the package at the same time. Using outer three state packages or AND-OR structures for bussing wastes a lot of space, thus using the components with three state outputs are preferable, except when we want to use a component on two busses we will have to use some form of extra hardware.

74ls374: This package is very similar to that of the 74ls373. The only particular difference is that here we are allowed to feeding of outputs through combinational logic back into the register and this is because we have flip flops instead of latches in the 74ls373:

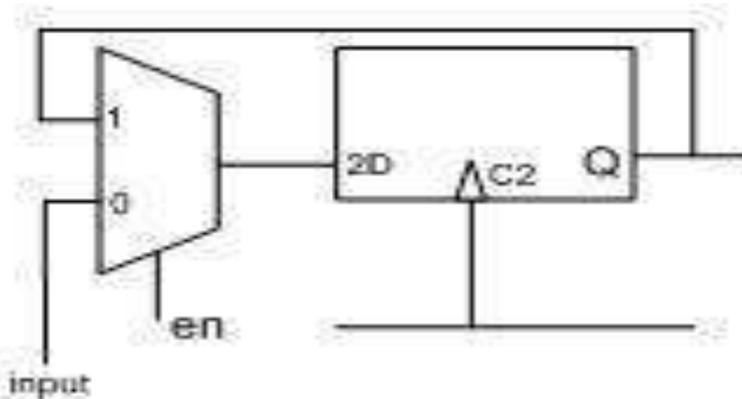


Consider the following figure. We can add the contents of CReg and AReg, and put the total in CReg using this circuit and activating the control signals selAReg, add and selCReg. This will be done in two clock pulses and this is why the clock frequency must be carefully decided (according to the longest path delay of the circuit).



The following structure shows how the clock enable of the registers actually works:

As we mentioned before, gating must not be done on the clock, because it can easily give this input hazards that we don't want.

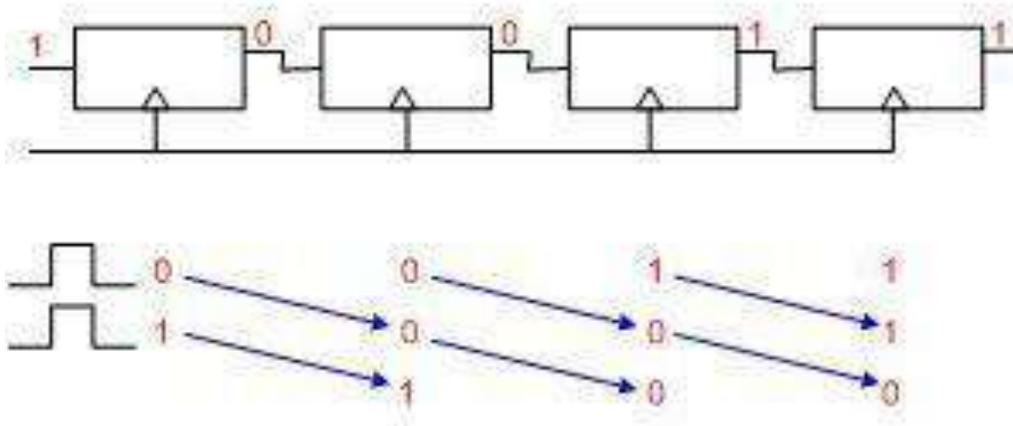


In system designs, such as the latter example, all control signals last from one rising edge to the next one. Remember that the clock frequency is determined by the longest common path.

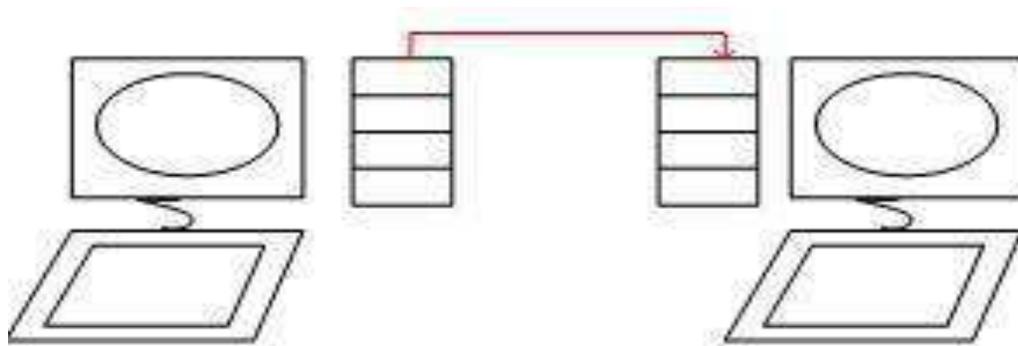
Shift Register

Shift registers are a type of sequential logic circuit, mainly for storage of digital data. They are a group of flip-flops connected in a chain so that the output from one flip-flop becomes the input of the next flip-flop. Most of the registers possess no characteristic internal sequence of states. All the flip-flops are driven by a common clock, and all are set or reset simultaneously. In this chapter, the basic types of shift registers are studied, such as Serial In - Serial Out, Serial In - Parallel Out, Parallel In - Serial Out, Parallel In - Parallel Out, and bidirectional shift registers. A special form of counter - the shift register counter, is also introduced.

Let's observe the values of the flip flops in this shift register for the next couple of clock pulse:



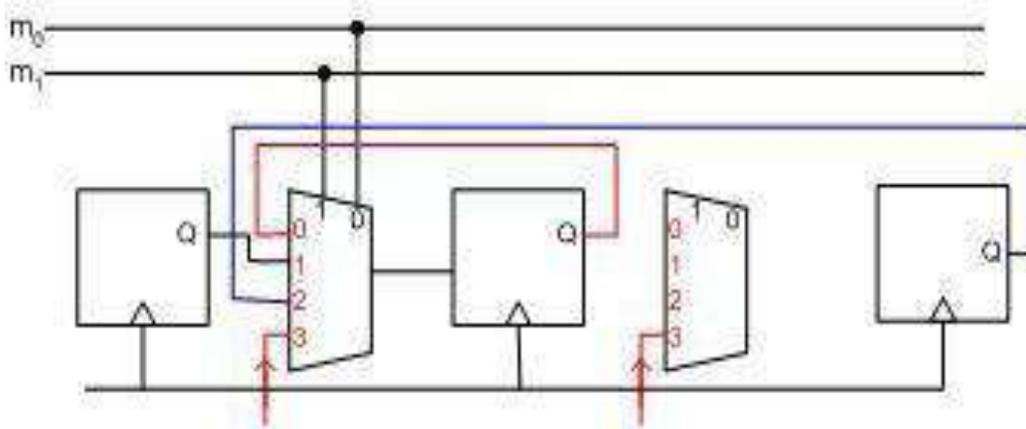
We are actually shifting our data to the right on every clock pulse. Shift registers are widely used in parallel to serial converters which find applications in computer communications.



74ls164: This package has two inputs and eight outputs. It can be useful in serial to parallel conversion of data but not parallel to serial because there is no parallel loading.

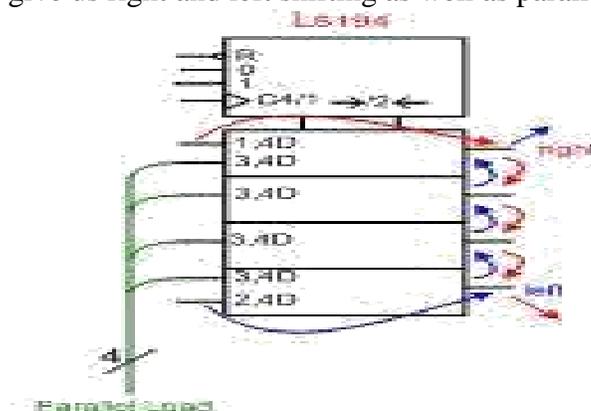


We now want to see how universal shift registers are made. Consider the following circuit:



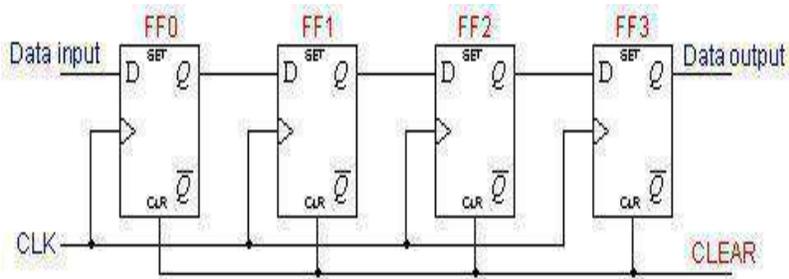
In the last diagram, you can see that 4 modes of operation exist. When m_1m_0 is 00, nothing happens, that is the contents of the flip flops don't change due to feed backing. $m_1m_0=01$ puts us in right shift mode and 10 in left shift, whereas $m_1m_0=11$ gives us parallel load. This structure can be used in a shift register to give us parallel to serial conversion abilities.

741s194: This package give us right and left shifting as well as parallel load in mode 11.



Serial In - Serial Out Shift Registers

A basic four-bit shift register can be constructed using four D flip-flops, as shown below. The operation of the circuit is as follows. The register is first cleared, forcing all four outputs to zero. The input data is then applied sequentially to the D input of the first flip-flop on the left (FF0). During each clock pulse, one bit is transmitted from left to right. Assume a data word to be 1001. The least significant bit of the data has to be shifted through the register from FF0 to FF3.

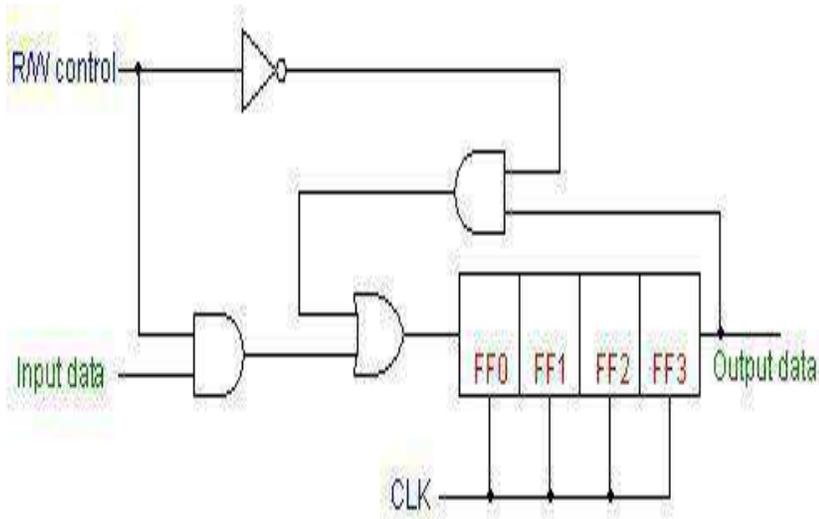


| | FF0 | FF1 | FF2 | FF3 |
|-------|-----|-----|-----|-----|
| CLEAR | 0 | 0 | 0 | 0 |

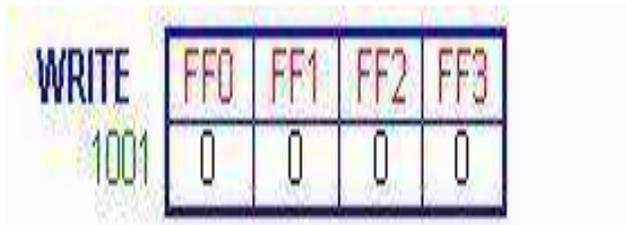
In order to get the data out of the register, they must be shifted out serially. This can be done destructively or non-destructively. For destructive readout, the original data is lost and at the end of the read cycle, all flip-flops are reset to zero.

| | FF0 | FF1 | FF2 | FF3 |
|-------|-----|-----|-----|-----|
| CLEAR | 0 | 0 | 0 | 0 |

To avoid the loss of data, an arrangement for a non-destructive reading can be done by adding two AND gates, an OR gate and an inverter to the system. The construction of this circuit is shown below

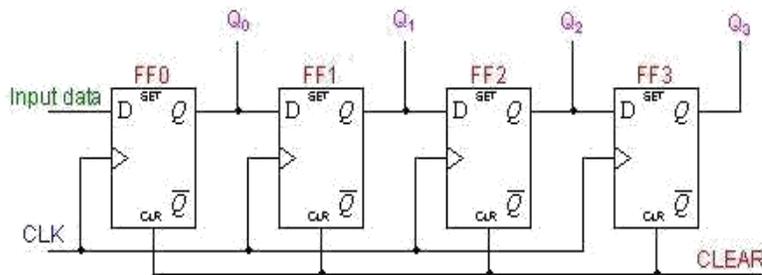


The data is loaded to the register when the control line is HIGH (ie WRITE). The data can be shifted out of the register when the control line is LOW (ie READ). This is shown in the animation below

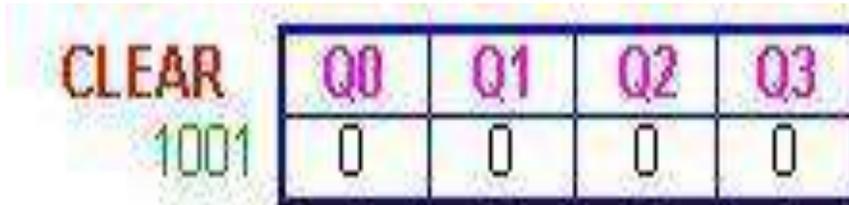


Serial In - Parallel Out Shift Registers

For this kind of register, data bits are entered serially in the same manner as discussed in the last section. The difference is the way in which the data bits are taken out of the register. Once the data are stored, each bit appears on its respective output line, and all bits are available simultaneously. A construction of a four-bit serial in - parallel out register is shown below.

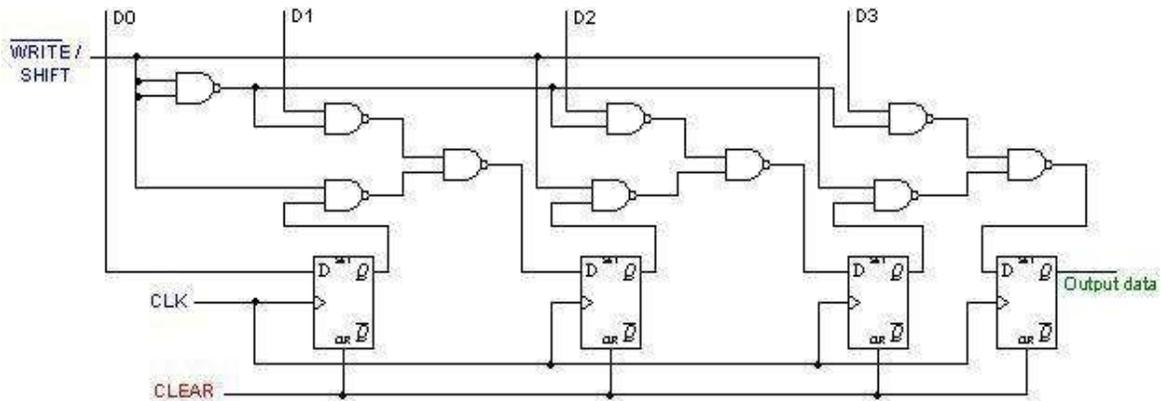


In the animation below, we can see how the four-bit binary number 1001 is shifted to the Q outputs of the register.

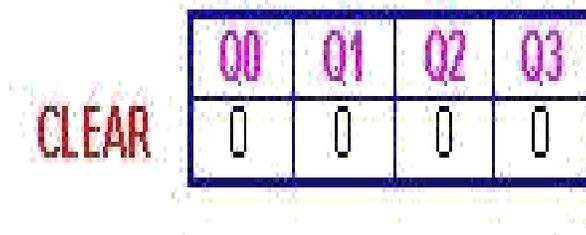


Parallel In - Serial Out Shift Registers

A four-bit parallel in - serial out shift register is shown below. The circuit uses D flip-flops and NAND gates for entering data (ie writing) to the register.

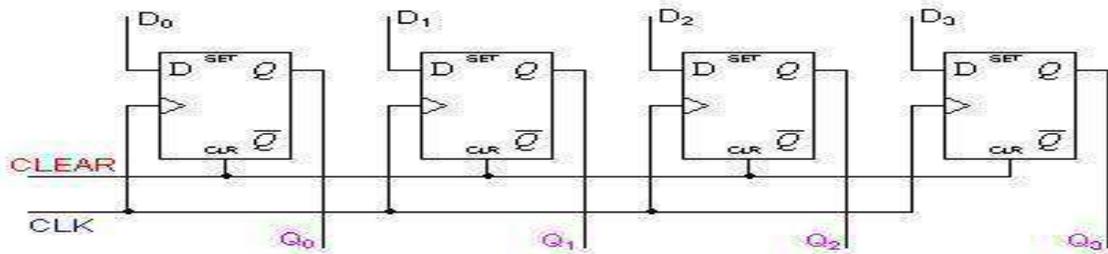


D0, D1, D2 and D3 are the parallel inputs, where D0 is the most significant bit and D3 is the least significant bit. To write data in, the mode control line is taken to LOW and the data is clocked in. The data can be shifted when the mode control line is HIGH as SHIFT is active high. The register performs right shift operation on the application of a clock pulse, as shown in the animation below.



Parallel In - Parallel Out Shift Register

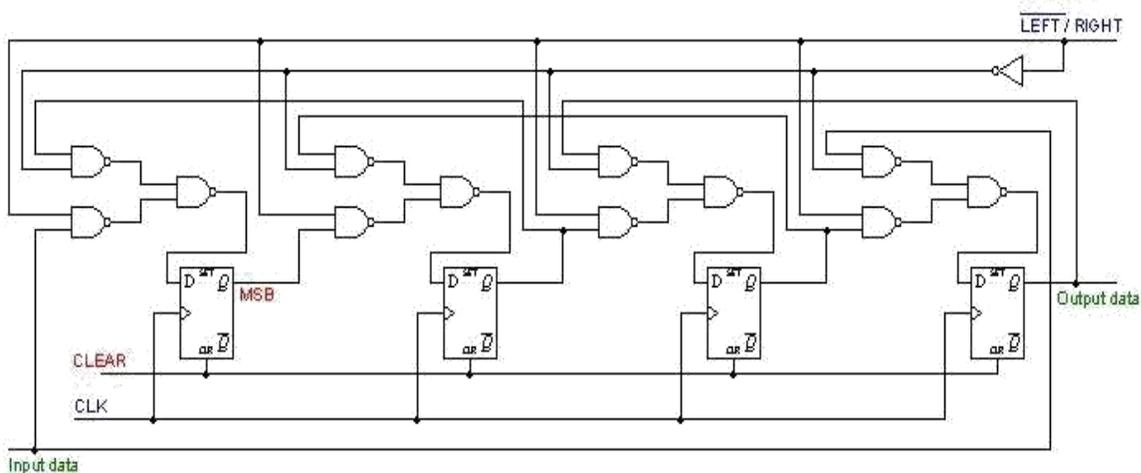
For parallel in - parallel out shift registers, all data bits appear on the parallel outputs immediately following the simultaneous entry of the data bits. The following circuit is a four-bit parallel in - parallel out shift register constructed by D flip-flops.



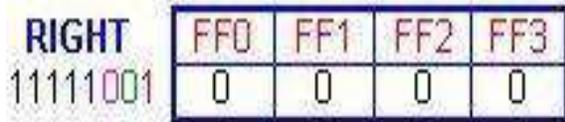
The D's are the parallel inputs and the Q's are the parallel outputs. Once the register is clocked, all the data at the D inputs appear at the corresponding Q outputs simultaneously.

Bidirectional Shift Registers

The registers discussed so far involved only right shift operations. Each right shift operation has the effect of successively dividing the binary number by two. If the operation is reversed (left shift), this has the effect of multiplying the number by two. With suitable gating arrangement a serial shift register can perform both operations. A *bidirectional*, or *reversible*, shift register is one in which the data can be shift either left or right. A four-bit bidirectional shift register using D flip-flops is shown below.



Here a set of NAND gates are configured as OR gates to select data inputs from the right or left adjacent bistables, as selected by the LEFT/RIGHT control line. The animation below performs right shift four times, then left shift four times. Notice the order of the four output bits are not the same as the order of the original four input bits. They are actually reversed!

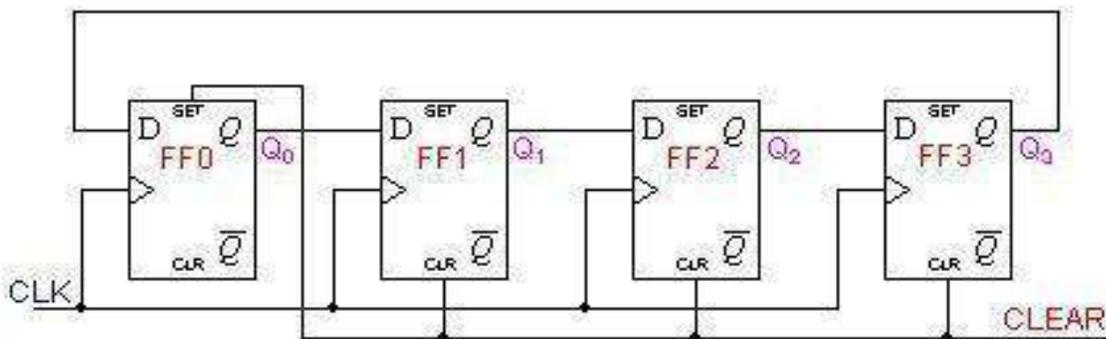


Shift Register Counters

Two of the most common types of shift register counters are introduced here: the Ring counter and the Johnson counter. They are basically shift registers with the serial outputs connected back to the serial inputs in order to produce particular sequences. These registers are classified as counters because they exhibit a specified sequence of states.

Ring Counters

A ring counter is basically a circulating shift register in which the output of the most significant stage is fed back to the input of the least significant stage. The following is a 4-bit ring counter constructed from D flip-flops. The output of each stage is shifted into the next stage on the positive edge of a clock pulse. If the CLEAR signal is high, all the flip-flops except the first one FF0 are reset to 0. FF0 is preset to 1 instead



Since the count sequence has 4 distinct states, the counter can be considered as a mod-4 counter. Only 4 of the maximum 16 states are used, making ring counters very inefficient in terms of state usage. But the major advantage of a ring counter over a binary counter is that it is self-decoding. No extra decoding circuit is needed to determine what state the counter is in.

| Clock Pulse | Q3 | Q2 | Q1 | Q0 |
|-------------|----|----|----|----|
| 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 |

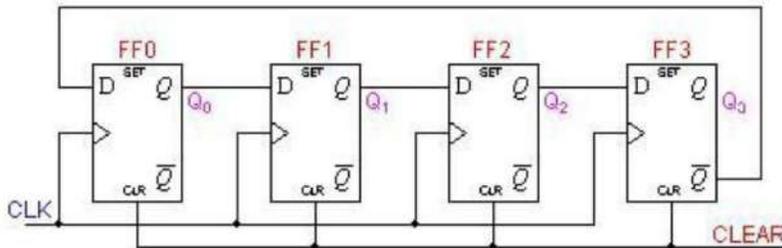


CLEAR

| FF0 | FF1 | FF2 | FF3 |
|-----|-----|-----|-----|
| 1 | 0 | 0 | 0 |

Johnson Counters

Johnson counters are a variation of standard ring counters, with the inverted output of the last stage fed back to the input of the first stage. They are also known as twisted ring counters. An n -stage Johnson counter yields a count sequence of length $2n$, so it may be considered to be a mod- $2n$ counter. The circuit above shows a 4-bit Johnson counter. The state sequence for the counter is given in the table as well as the animation on the left.



| Clock Pulse | Q3 | Q2 | Q1 | Q0 |
|-------------|----|----|----|----|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 1 |
| 3 | 0 | 1 | 1 | 1 |
| 4 | 1 | 1 | 1 | 1 |
| 5 | 1 | 1 | 1 | 0 |
| 6 | 1 | 1 | 0 | 0 |
| 7 | 1 | 0 | 0 | 0 |



CLEAR

| FF0 | FF1 | FF2 | FF3 |
|-----|-----|-----|-----|
| 0 | 0 | 0 | 0 |

Again, the apparent disadvantage of this counter is that the maximum available states are not fully utilized. Only eight of the sixteen states are being used. Beware that for both the Ring and the Johnson counter must initially be forced into a valid state in the count sequence because they operate on a subset of the available number of states. Otherwise, the ideal sequence will not be followed.

Applications

Shift registers can be found in many applications. Here is a list of a few

To produce time delay

The serial in -serial out shift register can be used as a time delay device. The amount of delay can be controlled by:

1. the number of stages in the register
2. the clock frequency

To simplify combinational logic

The ring counter technique can be effectively utilized to implement synchronous sequential circuits. A major problem in the realization of sequential circuits is the assignment of binary codes to the internal states of the circuit in order to reduce the complexity of circuits required. By assigning one flip-flop to one internal state, it is possible to simplify the combinational logic required to realize the complete sequential circuit. When the circuit is in a particular state, the flip-flop corresponding to that state is set to HIGH and all other flip-flops remain LOW.

To convert serial data to parallel data

A computer or microprocessor-based system commonly requires incoming data to be in parallel format. But frequently, these systems must communicate with external devices that send or receive serial data. So, serial-to-parallel conversion is required. As shown in the previous sections, a serial in - parallel out register can achieve this.

Basic sequential Design steps

1. **Step 1:** From a word description, determine what needs to be stored in memory, that is, what are the possible states.
2. **Step 2:** If necessary, code the inputs and outputs in binary.
3. **Step 3:** Derive a state table or state diagram to describe the behavior of the system.
4. **Step 4:** Use state reduction techniques to find a state table that produces the same input/output behavior, but has fewer states.

5. **Step 5:** Choose a state assignment, that is, code the states in binary.
6. **Step 6:** Choose a flip flop type and derive the flip flop input maps or tables.
7. **Step 7:** Produce the logic equation and draw a block diagram (as in the case of combinational systems).

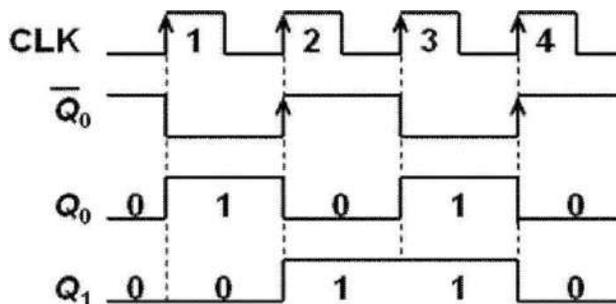
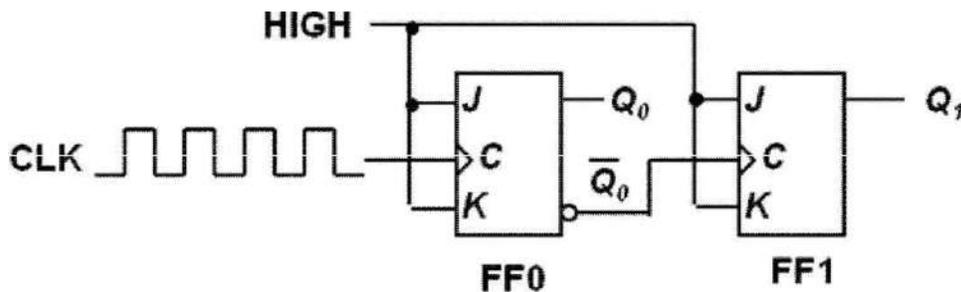
Design of Modulus N Synchronous Counters

Asynchronous (Ripple) Counters

Asynchronous counters: the flip-flops do not change states at exactly the same time as they do not have a common clock pulse. Also known as ripple counters, as the input clock pulse –ripples through the counter – cumulative delay is a drawback. n flip-flops a MOD (modulus) 2^n counter. (Note: A MOD- x counter cycles through x states.) Output of the last flip-flop (MSB) divides the input clock frequency by the MOD number of the counter, hence a counter is also a *frequency divider*.

Example: 2-bit ripple binary counter.

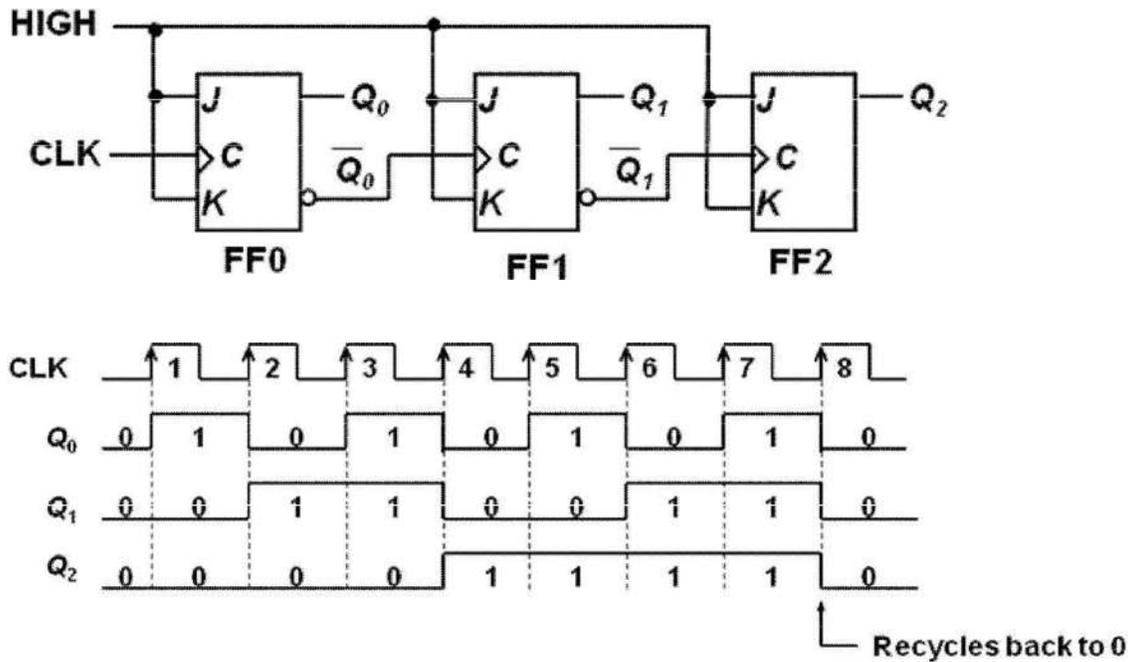
Output of one flip-flop is connected to the clock input of the next more-significant flip-flop.



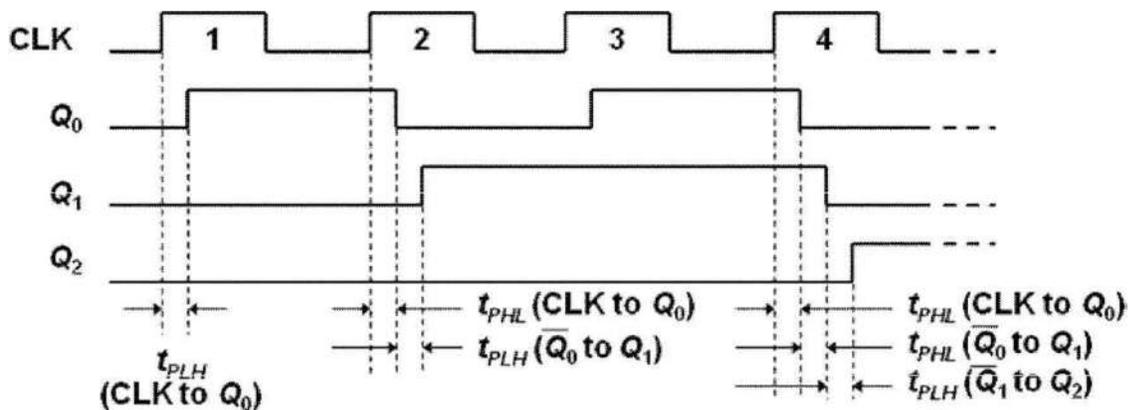
Timing diagram

00 → 01 → 10 → 11 → 00 ...

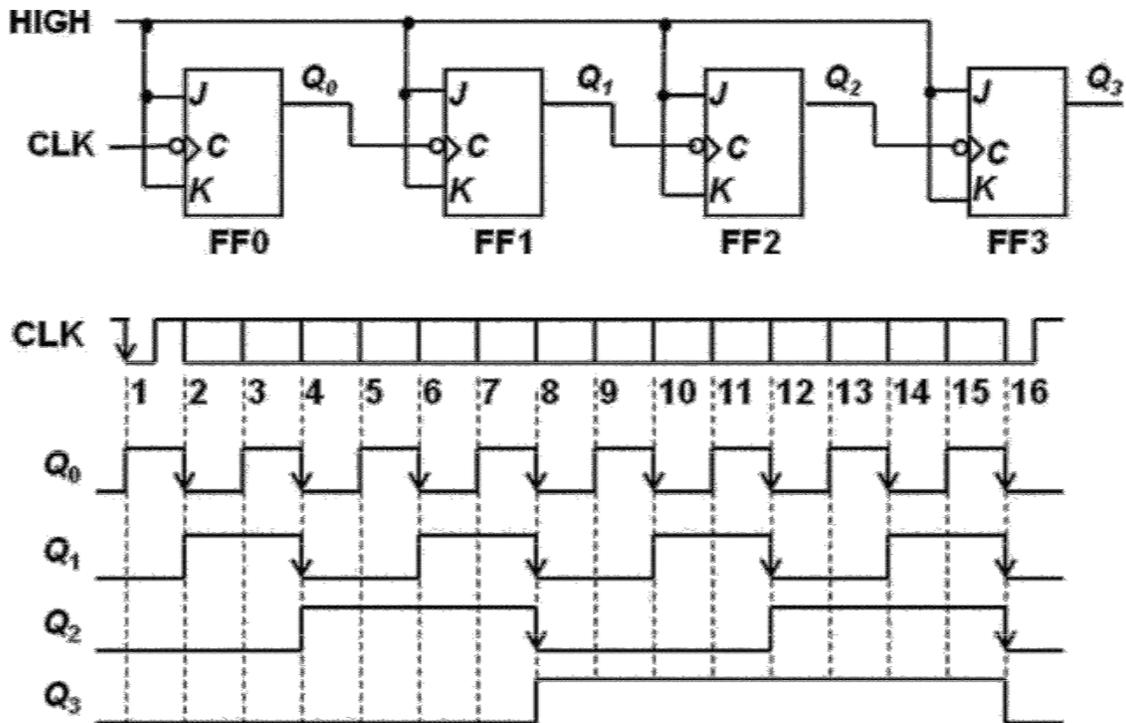
Example: 3-bit ripple binary counter



Propagation delays in an asynchronous (ripple-clocked) binary counter. If the accumulated delay is greater than the clock pulse, some counter states may be misrepresented



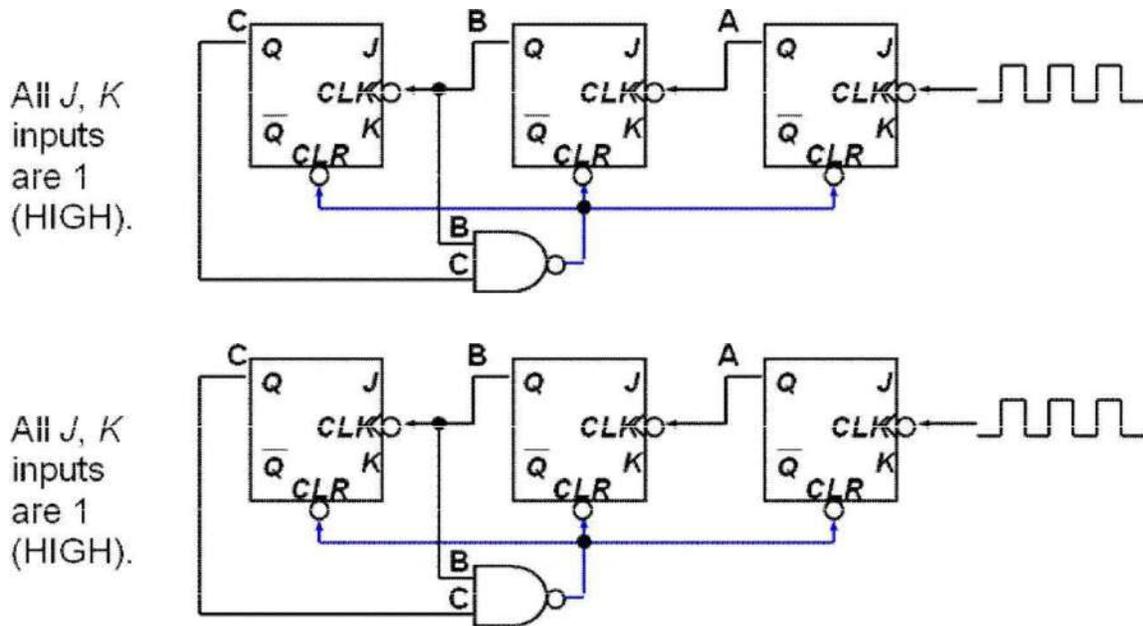
Example: 4-bit ripple binary counter (negative-edge triggered).

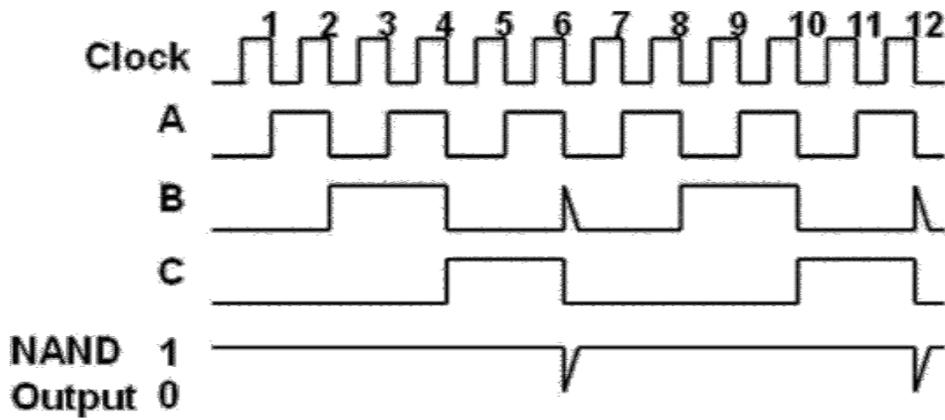


Asyn. Counters with MOD no. $< 2^n$

States may be skipped resulting in a truncated sequence. Technique: force counter to *recycle* before going through all of the states in the binary sequence.

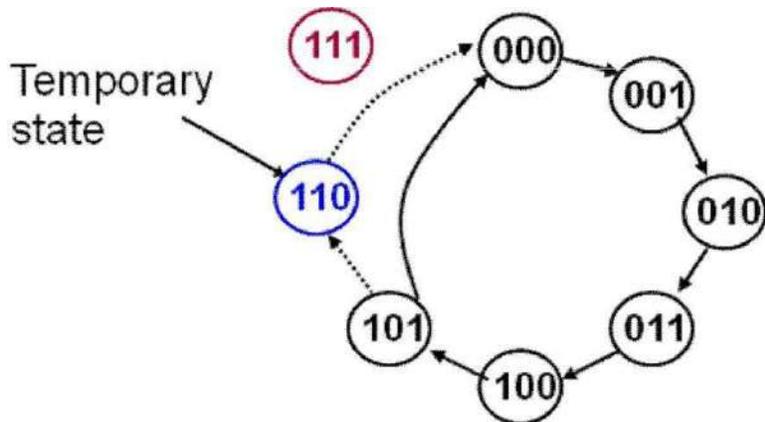
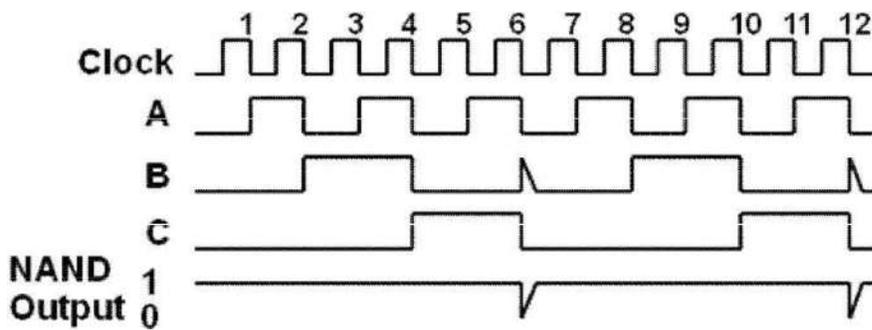
Example: Given the following circuit, determine the counting sequence (and hence the modulus no.)





MOD-6 counter produced by clearing (a MOD-8 binary counter) when count of six (110) occurs.

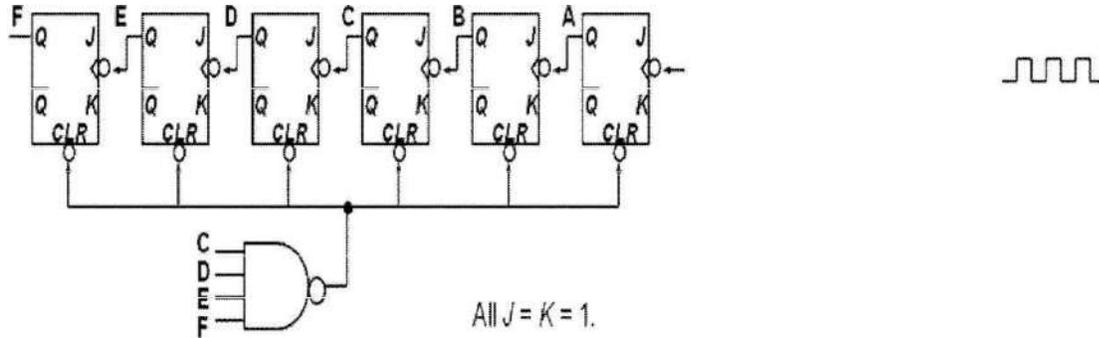
Counting sequence of circuit (in CBA order).



Counter is a MOD-6 counter.

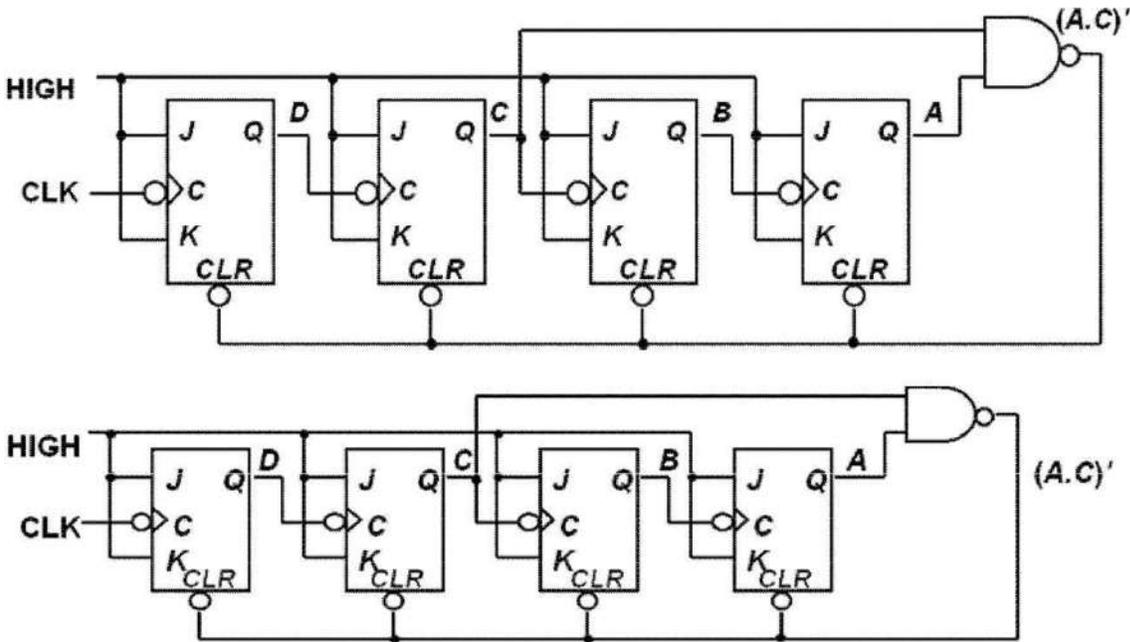
Exercise: How to construct an asynchronous MOD-5 counter? MOD-7 counter? MOD-12 counter?

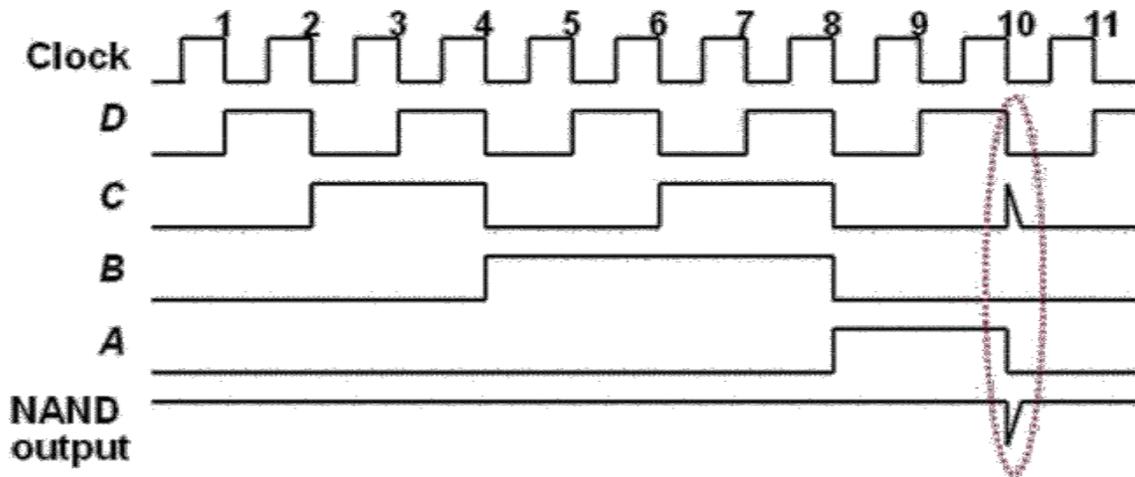
The following is a MOD-? counter?



Decade counters (or BCD counters) are counters with 10 states (modulus-10) in their sequence. They are commonly used in daily life (e.g.: utility meters, odometers, etc.).

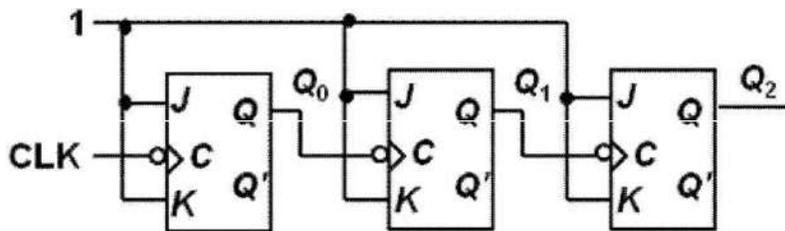
Design an asynchronous decade counter.



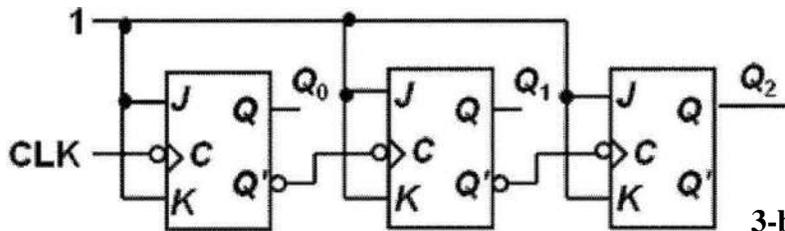


So far we are dealing with *up counters*. *Down counters*, on the other hand, count downward from a maximum value to zero, and repeat.

Example: A 3-bit binary (MOD- 2^3) down counter.



3-bit binary up counter

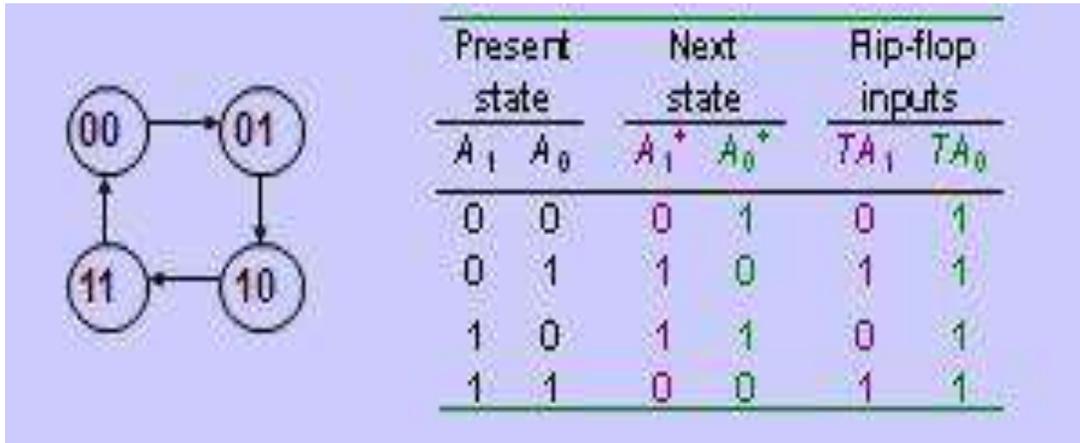


3-bit binary down counter

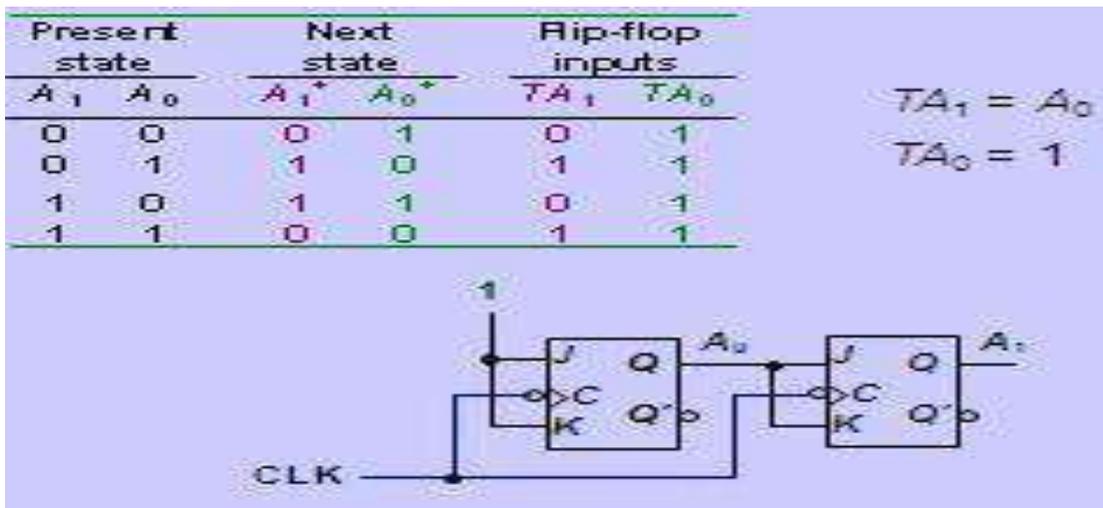
Synchronous (Parallel) Counters

Synchronous (parallel) counters: the flip-flops are clocked at the same time by a common clock pulse. We can design these counters using the sequential logic design process.

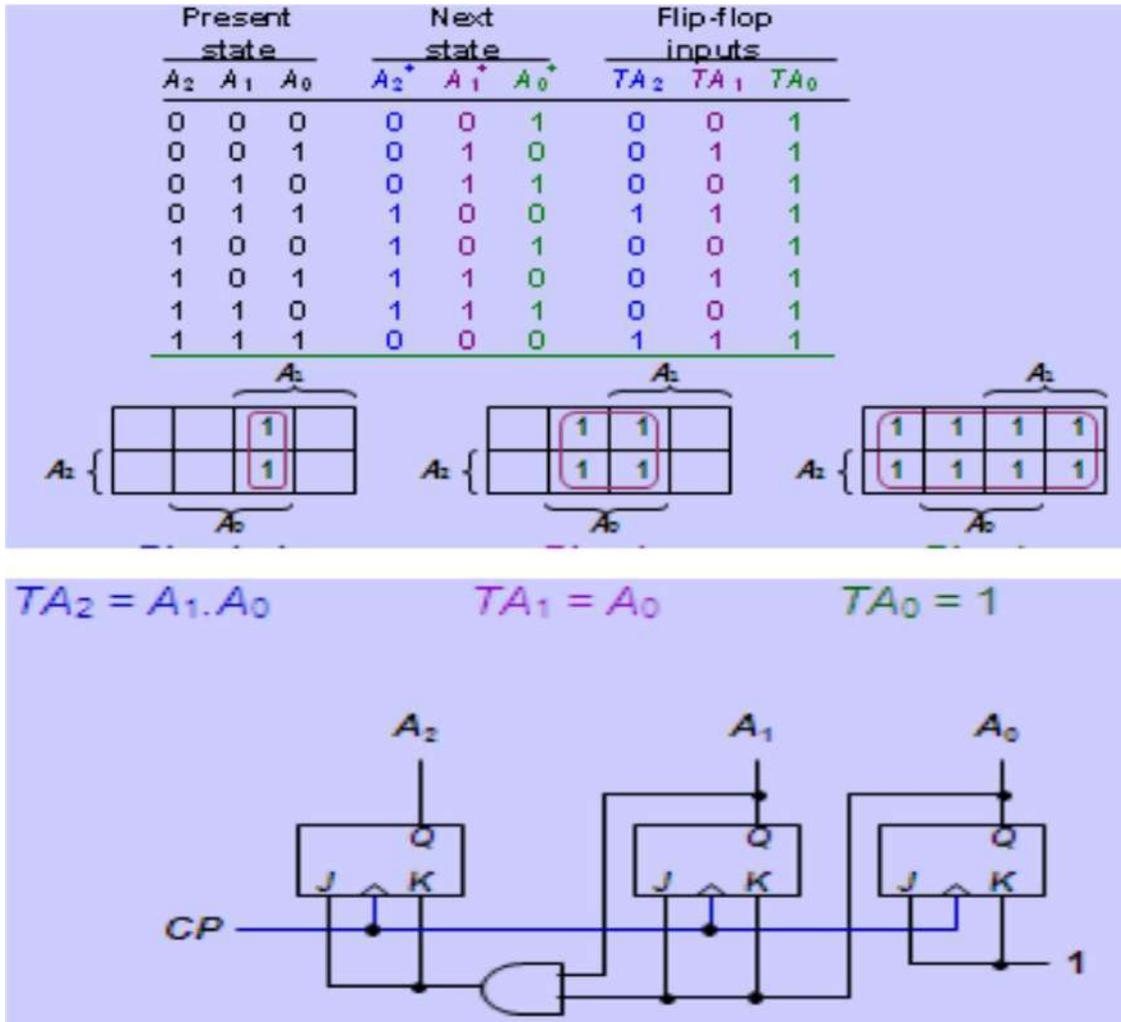
Example: 2-bit synchronous binary counter (using T flip-flops, or JK flip-flops with identical J,K inputs).



Example: 2-bit synchronous binary counter (using T flip-flops, or JK flip-flops with identical J,K inputs).



Example: 3-bit synchronous binary counter (using T flip-flops, or JK flip-flops with identical J, K inputs).



Note that in a binary counter, the n^{th} bit (shown underlined) is always complemented whenever

011...11 100...00 or 111...11

000...00

Hence, X_n is complemented whenever $X_{n-1}X_{n-2} \dots$

$X_1X_0 = 11 \dots 11$.

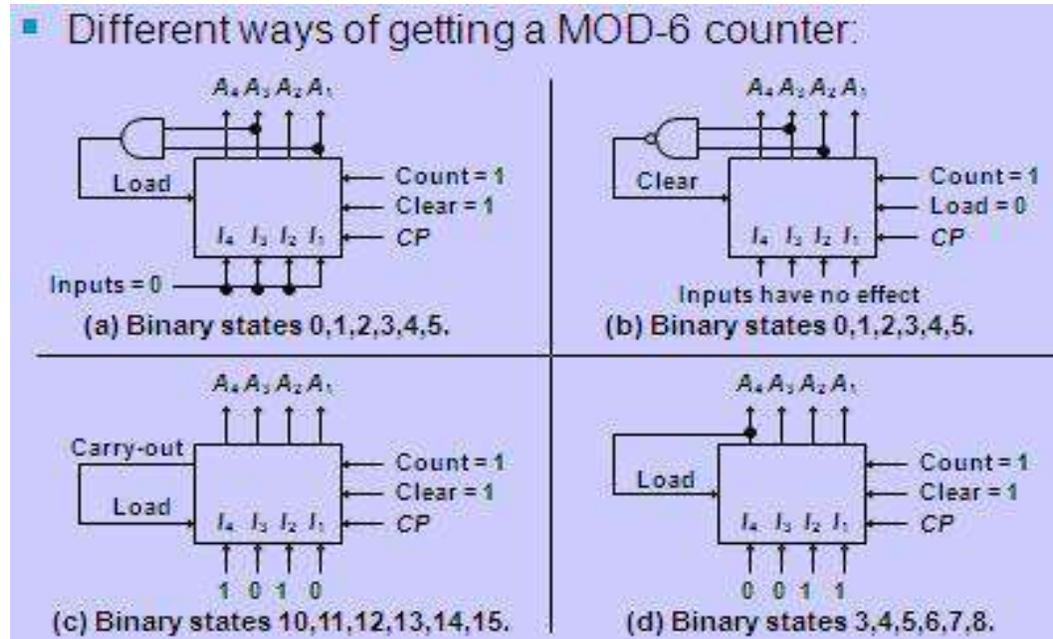
As a result, if T flip-flops are used, then $TX_n = X_n$.

$1 \cdot X_{n-2} \cdot \dots \cdot X_1 \cdot X_0 \setminus$

Counters with Parallel Load

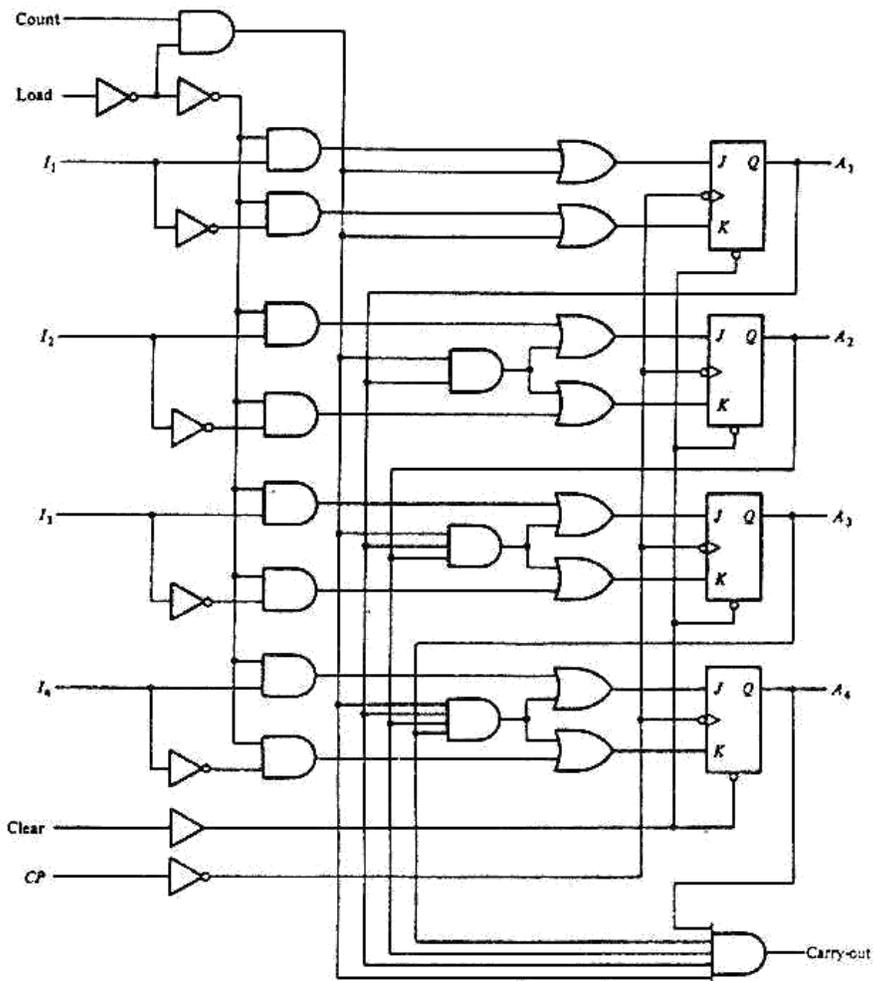
Counters could be augmented with parallel load capability for the following purposes:

- To start at a different state
- To count a different sequence
- As more sophisticated register with increment/decrement functionality.



4-bit counter with parallel load.

| Clear | CP | Load | Count | Function |
|-------|----|------|-------|-------------|
| 0 | X | X | X | Clear to 0 |
| 1 | X | 0 | 0 | No change |
| 1 | | 1 | X | Load inputs |
| 1 | | 0 | 1 | Next state |



There are two types of memories that are used in digital systems:

Random-access memory(RAM): perform both the write and read operations.

Read-only memory(ROM): perform only the read operation.

The read-only memory is a programmable logic device. Other such units are the programmable logic array(PLA), the programmable array logic(PAL), and the field-programmable gate array(FPGA).

Array logic

A typical programmable logic device may have hundreds to millions of gates interconnected through hundreds to thousands of internal paths. In order to show the internal logic diagram in a concise form, it is necessary to employ a special gate symbology applicable to array logic.



Fig. 7-1 Conventional and Array Logic Diagrams for OR Gate

Programmable Read Only Memory (PROM)

A block diagram of a ROM is shown below. It consists of k address inputs and n data outputs. The number of words in a ROM is determined from the fact that k address input lines are needed to specify 2^k words.

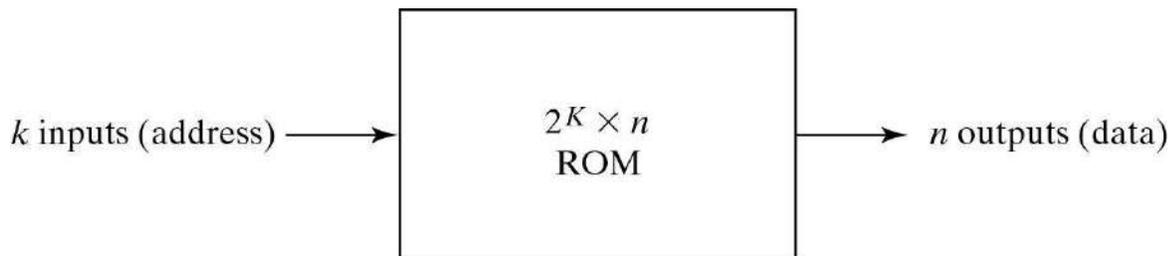


Fig. 7-9 ROM Block Diagram

Construction of ROM

Each output of the decoder represents a memory address. Each OR gate must be considered as having 32 inputs. A $2^k \times n$ ROM will have an internal $k \times 2^k$ decoder and n OR gates.

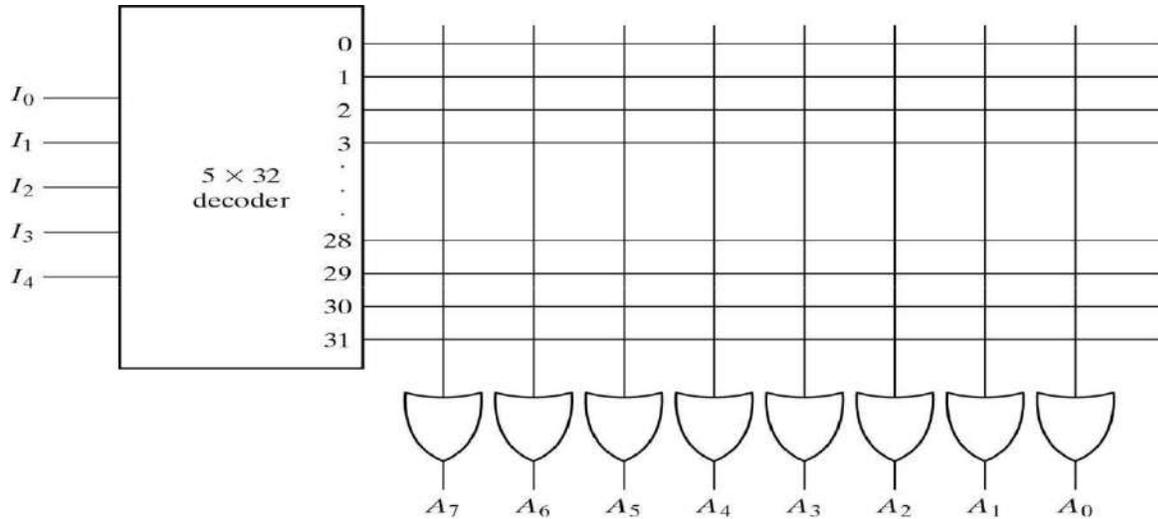


Fig. 7-10 Internal Logic of a 32×8 ROM

Truth table of ROM

A programmable connection between two lines is logically equivalent to a switch that can be altered to either be close or open. Intersection between two lines is sometimes called a cross-point.

Table 7-3
ROM Truth Table (Partial)

| Inputs | | | | | Outputs | | | | | | | |
|--------|----|----|----|----|---------|----|----|----|----|----|----|----|
| I4 | I3 | I2 | I1 | I0 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| | | | ⋮ | | | | ⋮ | | | | | |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |

Programming the ROM

In Table 7-3, 0 □ no connection

1 □ connection

Address 3 = 10110010 is permanent storage using fuse link

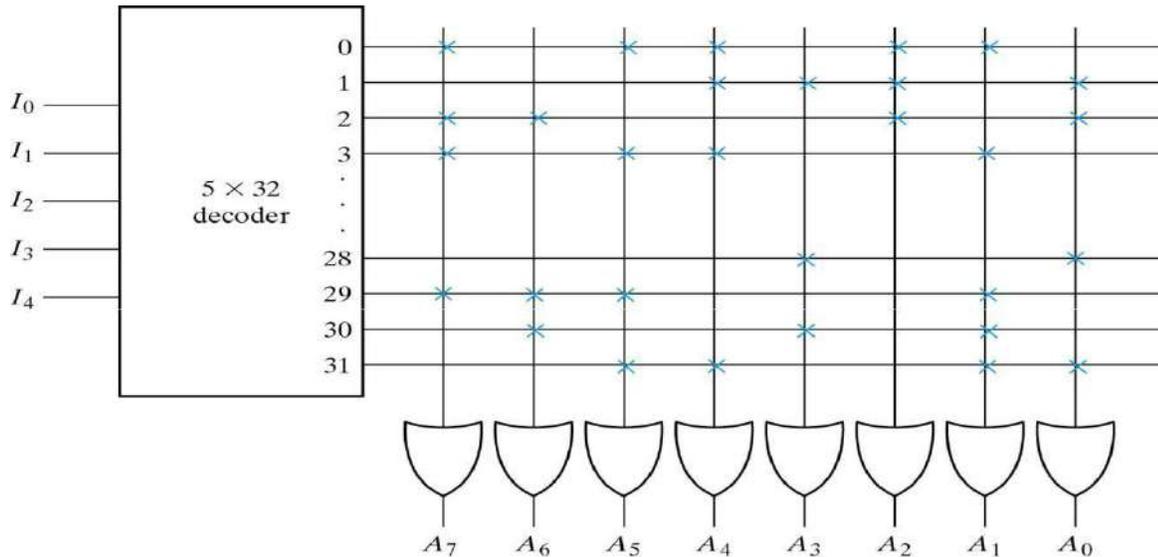


Fig. 7-11 Programming the ROM According to Table 7-3

Combinational circuit implementation

The internal operation of a ROM can be interpreted in two way: First, a memory unit that contains a fixed pattern of stored words. Second, implements a combinational circuit Fig. 7-11 may be considered as a combinational circuit with eight outputs, each being a function of the five input variables.

$$A_7(I_4, I_3, I_2, I_1, I_0) = \Sigma(0, 2, 3, \dots, 29)$$

Sum of minterms

In Table 7-3, output A_7

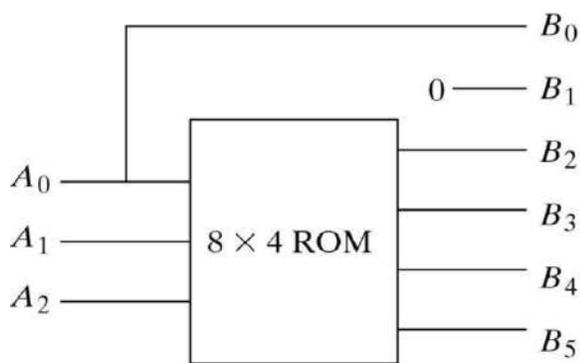
Example

Design a combinational circuit using a ROM. The circuit accepts a 3-bit number and generates an output binary number equal to the square of the input number.

Derive truth table first

Table 7-4
Truth Table for Circuit of Example 7-1

| Inputs | | | Outputs | | | | | | Decimal |
|--------|-------|-------|---------|-------|-------|-------|-------|-------|---------|
| A_2 | A_1 | A_0 | B_5 | B_4 | B_3 | B_2 | B_1 | B_0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 9 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 16 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 25 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 36 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 49 |



(a) Block diagram

| A_2 | A_1 | A_0 | B_5 | B_4 | B_3 | B_2 |
|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |

(b) ROM truth table

Fig. 7-12 ROM Implementation of Example 7-1

Types of ROMs

The required paths in a ROM may be programmed in four different ways.

1. Mask programming: fabrication process
2. Read-only memory or PROM: blown fuse /fuse intact
3. Erasable PROM or EPROM: placed under a special ultraviolet light for a given period of time will erase the pattern in ROM.
4. Electrically-erasable PROM(EEPROM): erased with an electrical signal instead of ultraviolet light.

Combinational PLDs

- A combinational PLD is an integrated circuit with programmable gates divided into an AND array and an OR array to provide an AND-OR sum of product implementation.
- PROM: fixed AND array constructed as a decoder and programmable OR array.
- PAL: programmable AND array and fixed OR array.

PLA: both the AND and OR arrays can be programmed.

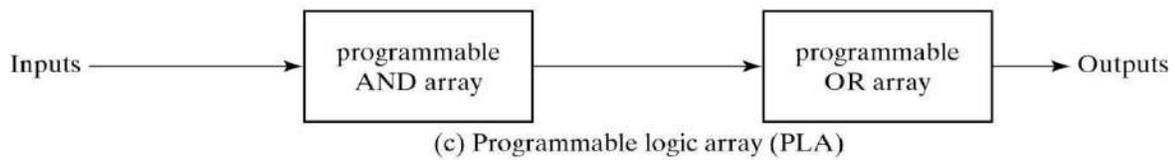
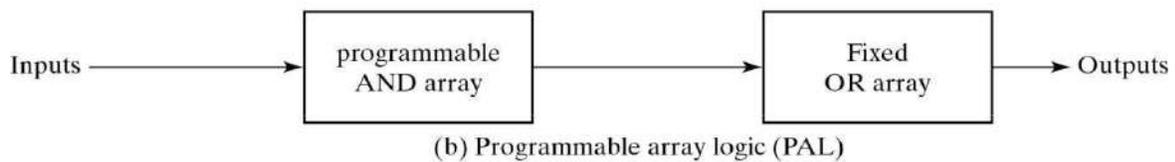
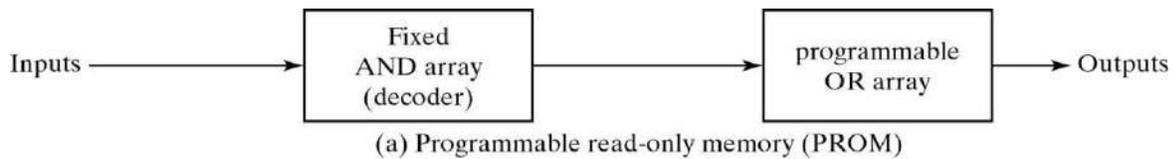


Fig. 7-13 Basic Configuration of Three PLDs

Programmable Logic Array

Fig.7-14, the decoder in PROM is replaced by an array of AND gates that can be programmed to generate any product term of the input variables. The product terms are then connected to OR gates to provide the sum of products for the required Boolean functions. The output is inverted when the XOR input is connected to 1 (since $x \oplus 1 = x'$). The output doesn't change and connect to 0 (since $x \oplus 0 = x$).

$$F_1 = AB' + AC + A'BC'$$

$$F_2 = (AC + BC)'$$

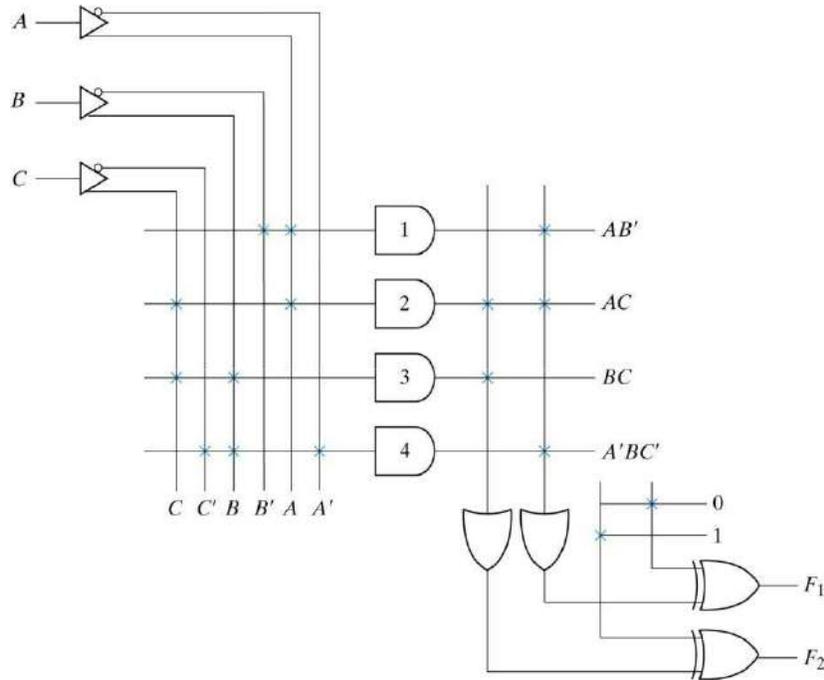


Fig. 7-14 PLA with 3 Inputs, 4 Product Terms, and 2 Outputs

Table 7-5
PLA Programming Table

| Product Term | | Inputs | | | Outputs | |
|--------------|---|--------|-------|---|---------|-----|
| | | A | B | C | (T) | (C) |
| | | F_1 | F_2 | | | |
| AB' | 1 | 1 | 0 | - | 1 | - |
| AC | 2 | 1 | - | 1 | 1 | 1 |
| BC | 3 | - | 1 | 1 | - | 1 |
| $A'BC'$ | 4 | 0 | 1 | 0 | 1 | - |

Programming Table

1. First: lists the product terms numerically
2. Second: specifies the required paths between inputs and AND gates
3. Third: specifies the paths between the AND and OR gates
4. For each output variable, we may have a T(ure) or C(omplement) for programming the XOR gate

Simplification of PLA

Careful investigation must be undertaken in order to reduce the number of distinct product terms, PLA has a finite number of AND gates. Both the true and complement of each function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions.

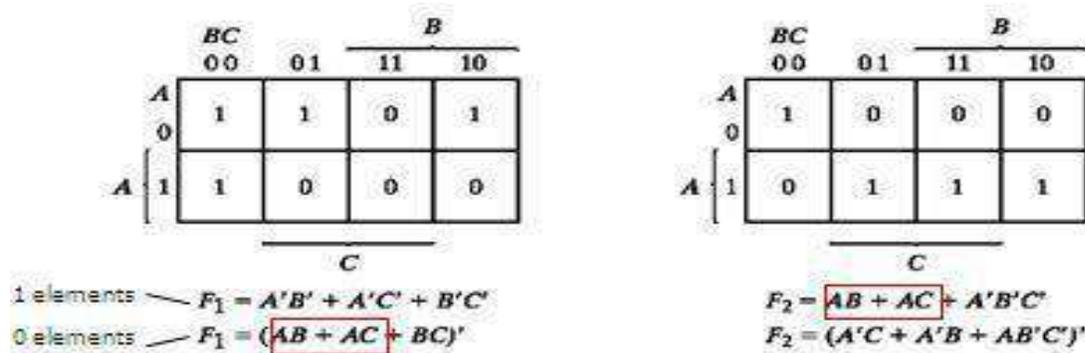
Example

Implement the following two Boolean functions with a PLA:

$$F_1(A, B, C) = \sum(0, 1, 2, 4)$$

$$F_2(A, B, C) = \sum(0, 5, 6, 7)$$

The two functions are simplified in the maps of Fig.7-15



PLA table by simplifying the function

Both the true and complement of the functions are simplified in sum of products. We can find the same terms from the group terms of the functions of F_1 , F_1' , F_2 and F_2' which will make the minimum terms.

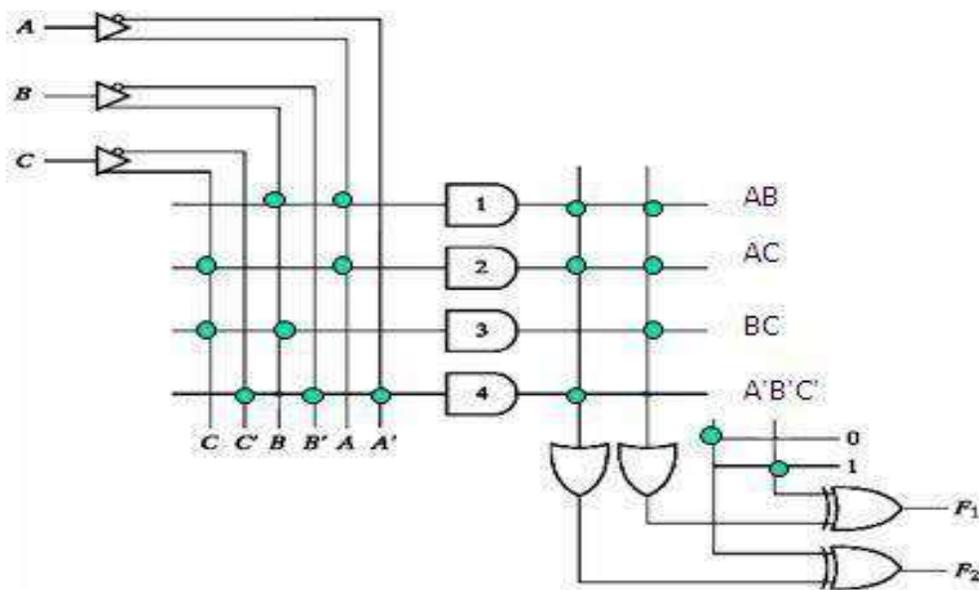
$$F_1 = (AB + AC + BC)'$$

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

| PLA programming table | | | | | | |
|-----------------------|--------------|--------|---|---|--------------|--------------|
| | Product term | Inputs | | | Outputs | |
| | | A | B | C | (C) F_1 | (T) F_2 |
| AB | 1 | 1 | 1 | - | 1 | 1 |
| AC | 2 | 1 | - | 1 | 1 | 1 |
| BC | 3 | - | 1 | 1 | 1 | - |
| $A'B'C'$ | 4 | 0 | 0 | 0 | - | 1 |

Fig. 7-15 Solution to Example 7-2

PLA implementation



Programmable Array Logic

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

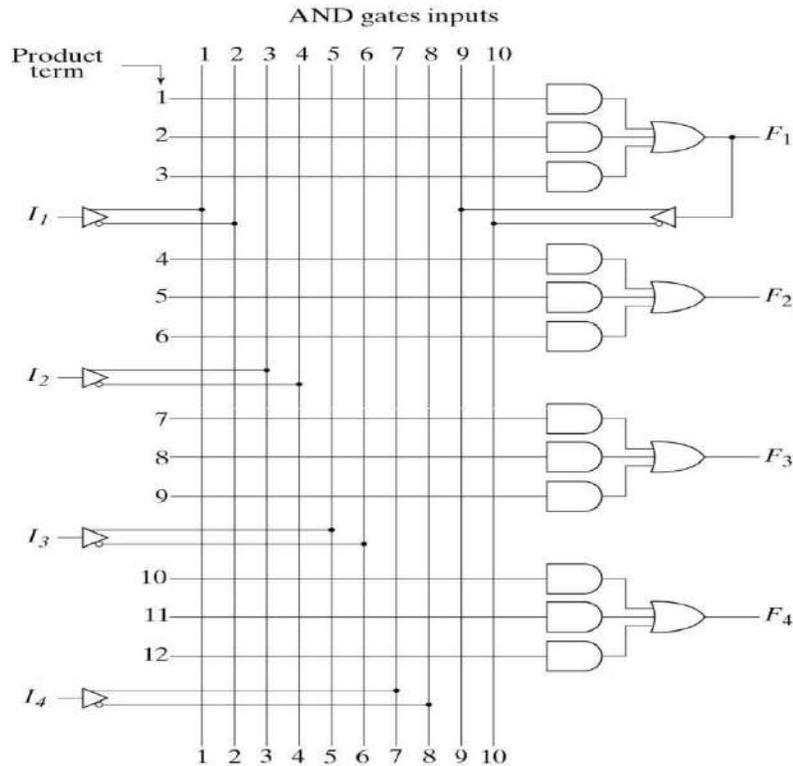


Fig. 7-16 PAL with Four Inputs, Four Outputs, and Three-Wide AND-OR Structure

When designing with a PAL, the Boolean functions must be simplified to fit into each section. Unlike the PLA, a product term cannot be shared among two or more OR gates. Therefore, each function can be simplified by itself without regard to common product terms. The output terminals are sometimes driven by three-state buffers or inverters.

Example

$$w(A, B, C, D) = \sum(2, 12, 13)$$

$$x(A, B, C, D) = \sum(7, 8, 9, 10, 11, 12, 13, 14, 15)$$

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

$$z(A, B, C, D) = \sum(1, 2, 8, 12, 13)$$

Simplifying the four functions as following Boolean functions:

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

$$x = A + BCD$$

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

$$w = ABC' + A'B'CD' + AC'D' + A'B'C'D = w + AC'D' + A'B'C'D$$

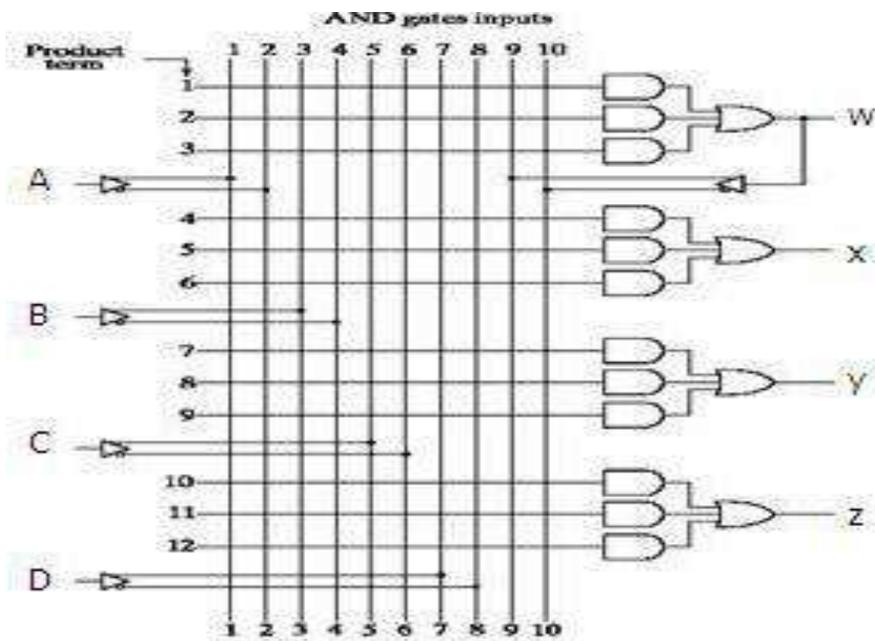
PAL Table

z has four product terms, and we can replace by w with two product terms, this will reduce the number of terms for z from four to three.

Table 7-6
PAL Programming Table

| Product Term | AND Inputs | | | | | Outputs |
|--------------|------------|---|---|---|---|-------------------------------------|
| | A | B | C | D | W | |
| 1 | 1 | 1 | 0 | - | - | $w = ABC'$ $+ A'B'CD'$ |
| 2 | 0 | 0 | 1 | 0 | - | |
| 3 | - | - | - | - | - | |
| 4 | 1 | - | - | - | - | $x = A$ $+ BCD$ |
| 5 | - | 1 | 1 | 1 | - | |
| 6 | - | - | - | - | - | |
| 7 | 0 | 1 | - | - | - | $y = A'B$ $+ CD$ $+ B'D'$ |
| 8 | - | - | 1 | 1 | - | |
| 9 | - | 0 | - | 0 | - | |
| 10 | - | - | - | - | 1 | $z = w$ $+ AC'D'$ $+ A'B'C'D$ |
| 11 | 1 | - | 0 | 0 | - | |
| 12 | 0 | 0 | 0 | 1 | - | |

PAL implementation



Fuse map for example

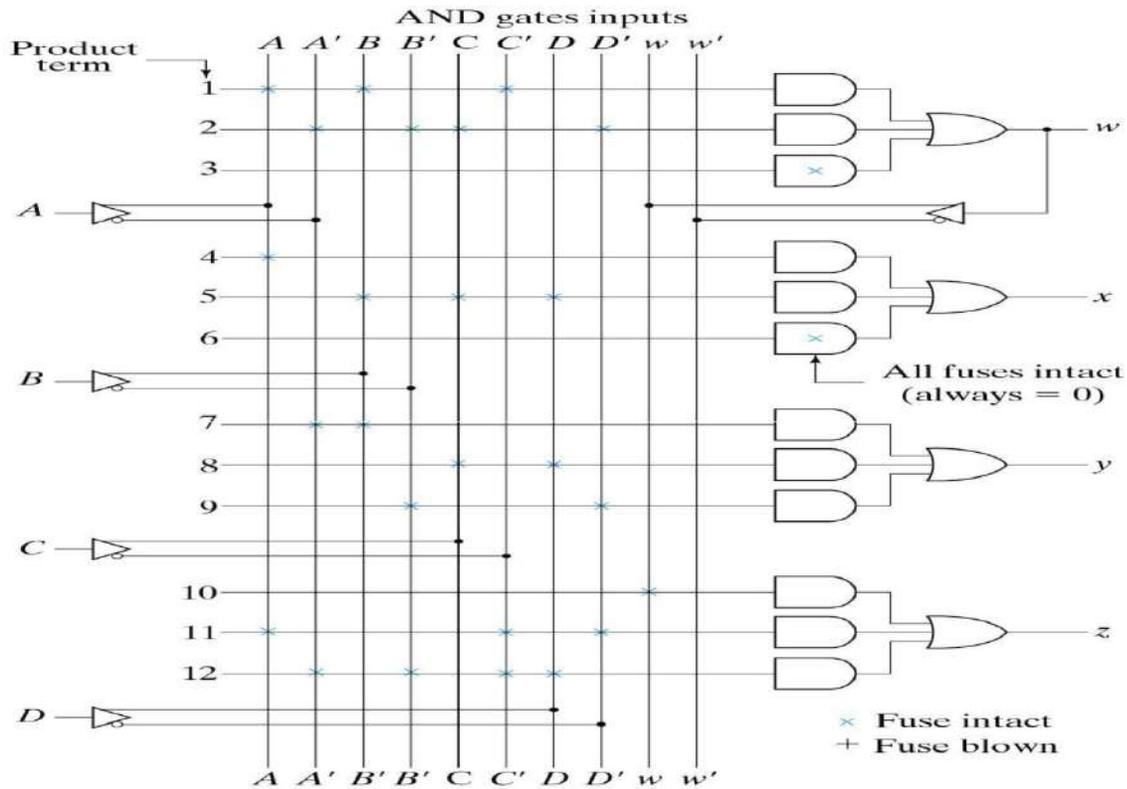


Fig. 7-17 Fuse Map for PAL as Specified in Table 7-6

Sequential Programmable Devices

Sequential programmable devices include both gates and flip-flops. There are several types of sequential programmable devices, but the internal logic of these devices is too complex to be shown here. We will describe three major types without going into their detailed construction.

1. Sequential (or simple) Programmable Logic Device (SPLD)
2. Complex Programmable Logic Device (CPLD)
3. Field Programmable Gate Array (FPGA)

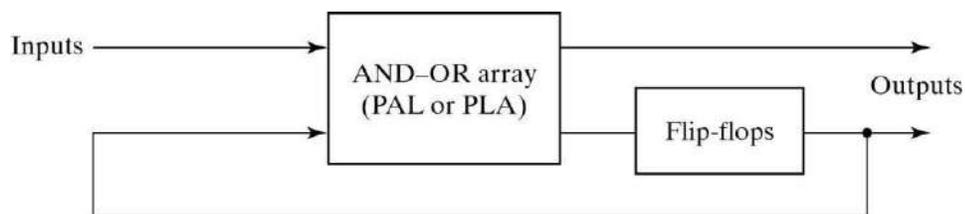


Fig. 7-18 Sequential Programmable Logic Device

FPLS

The first programmable device developed to support sequential circuit implementation is the field-programmable logic sequencer (FPLS). A typical FPLS is organized around a PLA with several outputs driving flip-flops. The flip-flops are flexible in that they can be programmed to operate as either JK or D type. The FPLS did not succeed commercially because it has too many programmable connections.

SPLD

Each section of an SPLD is called a macrocell. A macrocell is a circuit that contains a sum-of-products combinational logic function and an optional flip-flop. We will assume an AND-OR sum of products but in practice, it can be any one of the two-level implementation.

Macrocell

Fig.7-19 shows the logic of a basic macrocell. The AND-OR array is the same as in the combinational PAL shown in Fig.7-16.

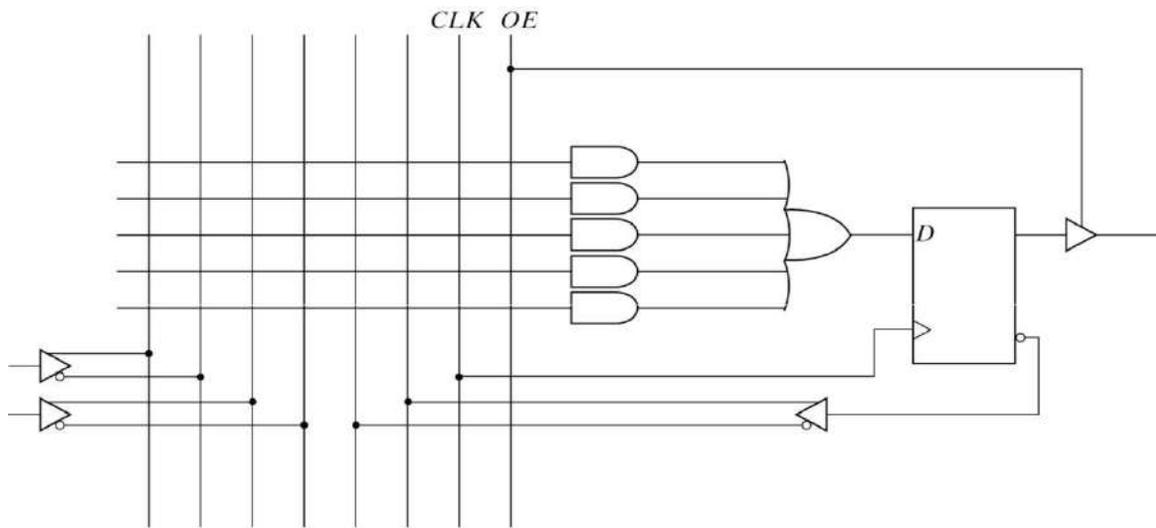


Fig. 7-19 Basic Macrocell Logic

CPLD

A typical SPLD has from 8 to 10 macrocells within one IC package. All the flip-flops are connected to the common CLK input and all three-state buffers are controlled by the EO input. The design of a digital system using PLD often requires the connection of several devices to produce the complete specification. For this type of application, it is more economical to use a complex programmable logic device (CPLD). A CPLD is a collection of individual PLDs on a single integrated circuit.

Fig.7-20 shows a general configuration of a CPLD. It consists of multiple PLDs interconnected through a programmable switch matrix. 8 to 16 macrocell per PLD.

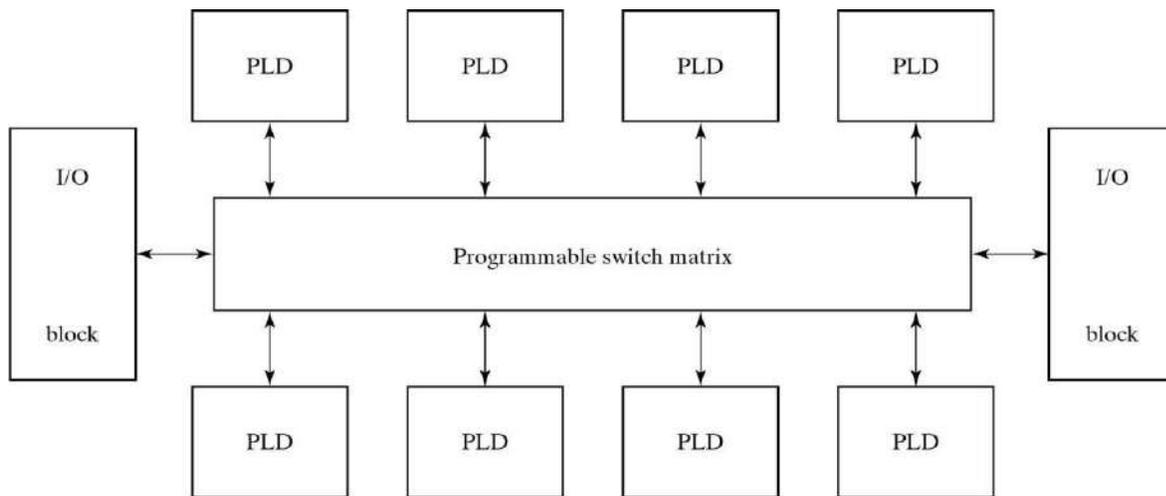


Fig. 7-20 General CPLD Configuration

Gate Array

The basic component used in VLSI design is the gate array. A gate array consists of a pattern of gates fabricated in an area of silicon that is repeated thousands of times until the entire chip is covered with the gates. Arrays of one thousand to hundred thousand gates are fabricated within a single IC chip depending on the technology used.

FPGA

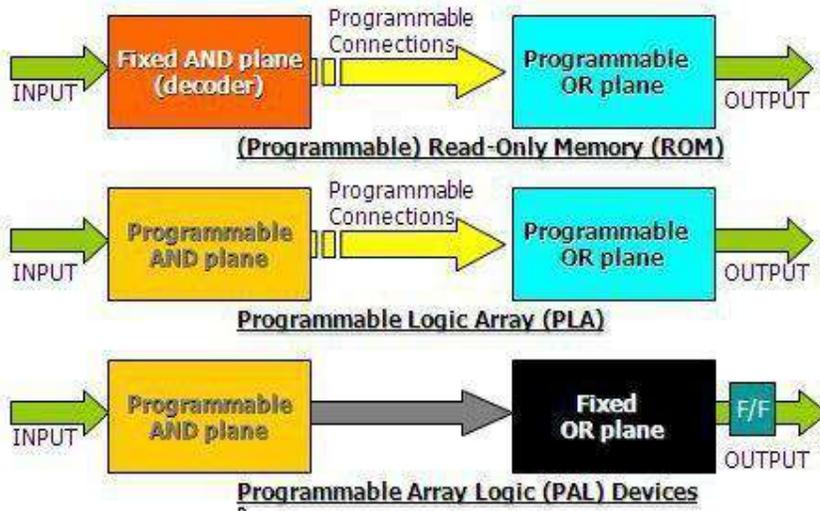
FPGA is a VLSI circuit that can be programmed in the user's location. A typical FPGA logic block consists of look-up tables, multiplexers, gates, and flip-flops. Look-up table is a truth table stored in a SRAM and provides the combinational circuit functions for the logic block.

Differential of RAM and ROM in FPGA

The advantage of using RAM instead of ROM to store the truth table is that the table can be programmed by writing into memory.

The disadvantage is that the memory is volatile and presents the need for the look-up table content to be reloaded in the event that power is disrupted.

Comparison between PROM, PLA and PAL



UNIT V

ROMs

UNIT V (MEMORIES)

Introduction

There are two types of memories that are used in digital systems:

Random-access memory (RAM): perform both the write and read operations.

Read-only memory (ROM): perform only the read operation.

The read-only memory is a programmable logic device. Other such units are the programmable logic array (PLA), the programmable array logic (PAL), and the field-programmable gate array (FPGA).

Random-Access Memory

A memory unit stores binary information in groups of bits called words.

$$1 \text{ byte} = 8 \text{ bits}$$

$$1 \text{ word} = 2 \text{ bytes}$$

The communication between a memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer.

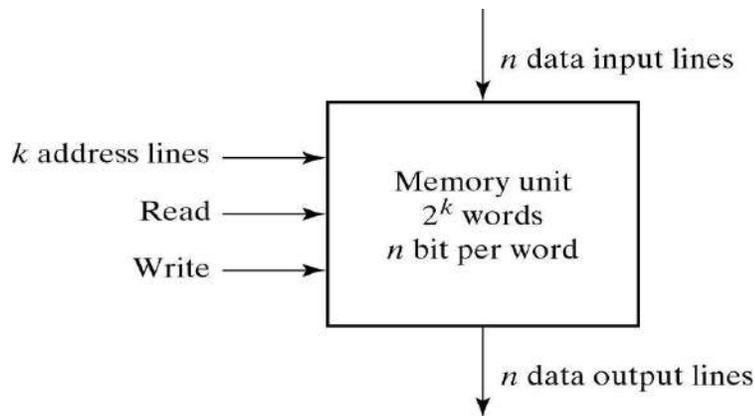


Fig. 7-2 Block Diagram of a Memory Unit

Content of a memory

Each word in memory is assigned an identification number, called an address, starting from 0 up to $2^k - 1$, where k is the number of address lines. The number of words in a memory with one of the letters $K=2^{10}$, $M=2^{20}$, or $G=2^{30}$.

$$64K = 2^{16} \quad 2M = 2^{21} \quad 4G = 2^{32}$$

| Memory address | | |
|----------------|---------|------------------|
| Binary | decimal | Memory content |
| 0000000000 | 0 | 1011010101011101 |
| 0000000001 | 1 | 1010101110001001 |
| 0000000010 | 2 | 0000110101000110 |
| | • | • |
| | • | • |
| | • | • |
| 1111111101 | 1021 | 1001110100010100 |
| 1111111110 | 1022 | 0000110100011110 |
| 1111111111 | 1023 | 1101111000100101 |

Fig. 7-3 Content of a 1024×16 Memory

Write and Read operations

Transferring a new word to be stored into memory:

1. Apply the binary address of the desired word to the address lines.
2. Apply the data bits that must be stored in memory to the data input lines.
3. Activate the write input.

Transferring a stored word out of memory:

1. Apply the binary address of the desired word to the address lines.
2. Activate the read input.

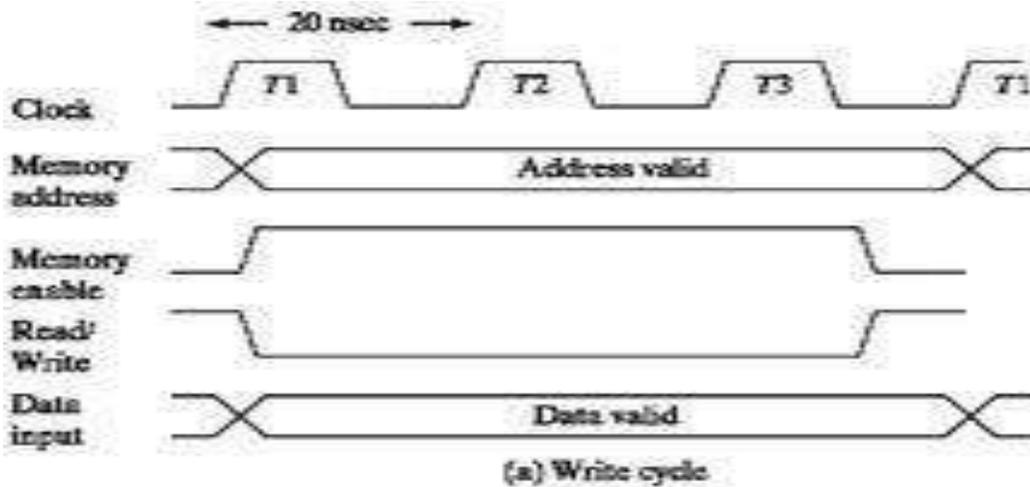
Commercial memory sometimes provide the two control inputs for reading and writing in a somewhat different configuration in table 7-1.

Table 7-1
Control Inputs to Memory Chip

| Memory Enable | Read/Write | Memory Operation |
|---------------|------------|-------------------------|
| 0 | X | None |
| 1 | 0 | Write to selected word |
| 1 | 1 | Read from selected word |

Timing Waveforms (write)

The access time and cycle time of the memory must be within a time equal to a fixed number of CPU clock cycles. The memory enable and the read/write signals must be activated after the signals in the address lines are stable to avoid destroying data in other memory words. Enable and read/write signals must stay active for at least 50ns.



Timing Waveforms (read)

The CPU can transfer the data into one of its internal registers during the negative transition of T3.

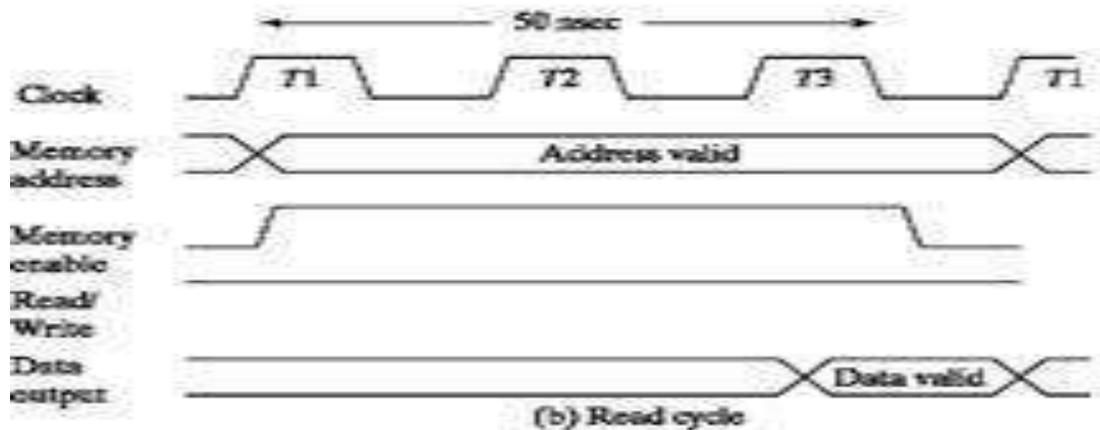


Fig. 7-4 Memory Cycle Timing Waveforms

Types of memories

In random-access memory, the word locations may be thought of as being separated in space, with each word occupying one particular location. In sequential-access memory, the information stored in some medium is not immediately accessible, but is available only certain intervals of time. A magnetic disk or tape unit is of this type. In a random-access memory, the access time is always the same regardless of the particular location of the word. In a sequential-access memory, the time it takes to access a word depends on the position of the word with respect to the reading head position; therefore, the access time is variable.

Static RAM

SRAM consists essentially of internal latches that store the binary information. The stored information remains valid as long as power is applied to the unit. SRAM is easier to use and has shorter read and write cycles. Low density, low capacity, high cost, high speed, high power consumption.

Dynamic RAM

DRAM stores the binary information in the form of electric charges on capacitors. The capacitors are provided inside the chip by MOS transistors. The capacitors tend to discharge with time and must be periodically recharged by refreshing the dynamic memory. DRAM

offers reduced power consumption and larger storage capacity in a single memory chip. High density, high capacity, low cost, low speed, low power consumption.

Types of memories

Memory units that lose stored information when power is turned off are said to be volatile. Both static and dynamic, are of this category since the binary cells need external power to maintain the stored information. Nonvolatile memory, such as magnetic disk, ROM, retains its stored information after removal of power.

Memory decoding

The equivalent logic of a binary cell that stores one bit of information is shown below.

Read/Write = 0, select = 1, input data to S-R latch

Read/Write = 1, select = 1, output data from S-R latch

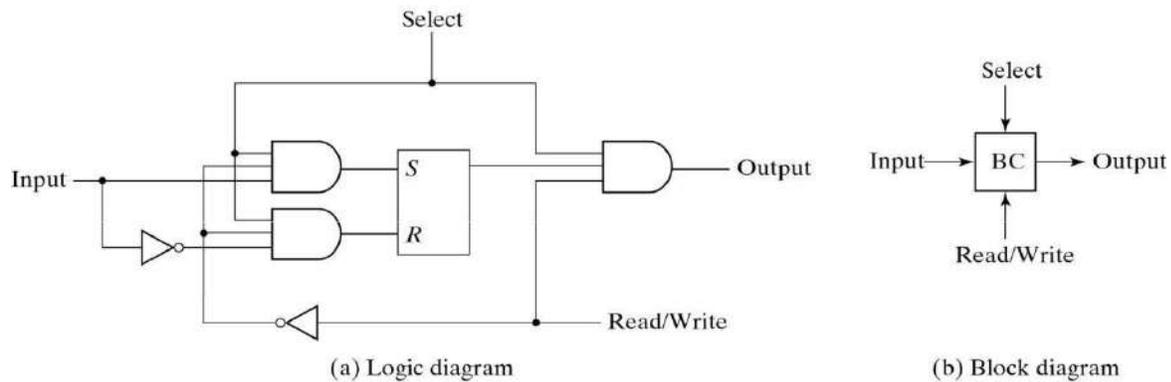


Fig. 7-5 Memory Cell

***Refer SR latch with NOR gates**

4X4 RAM

There is a need for decoding circuits to select the memory word specified by the input address. During the read operation, the four bits of the selected word go through OR gates to the output terminals. During the write operation, the data available in the input lines are transferred into the four binary cells of the selected word. A memory with 2^k words of n bits per word requires k address lines that go into $k \times 2^k$ decoder

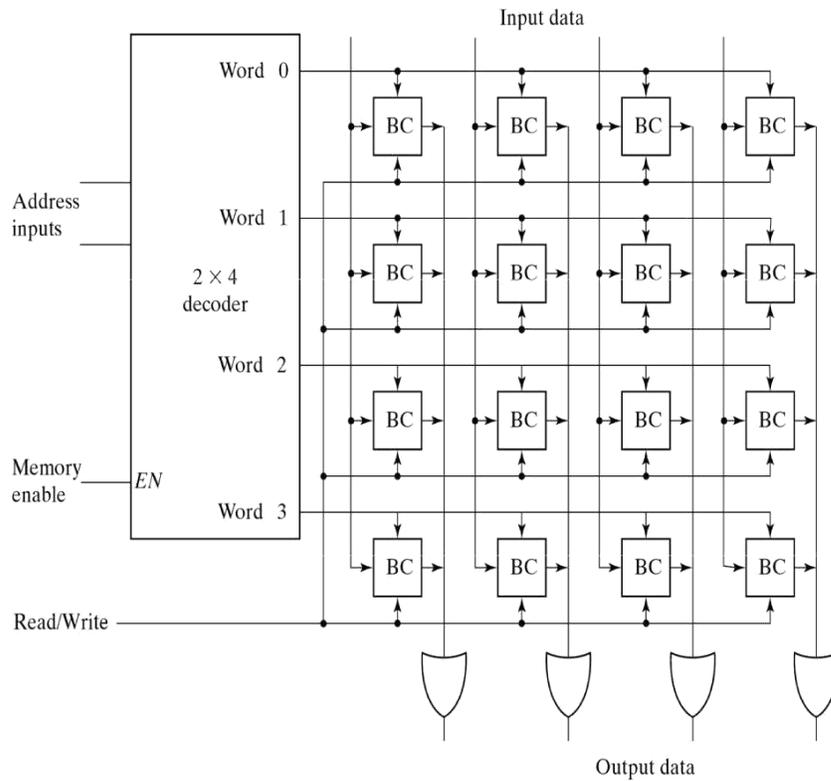


Fig. 7-6 Diagram of a 4 × 4 RAM

Coincident decoding

A decoder with k inputs and 2^k outputs requires 2^k AND gates with k inputs per gate. Two decoding in a two-dimensional selection scheme can reduce the number of inputs per gate. 1K-word memory, instead of using a single 10X1024 decoder, we use two 5X32 decoders.

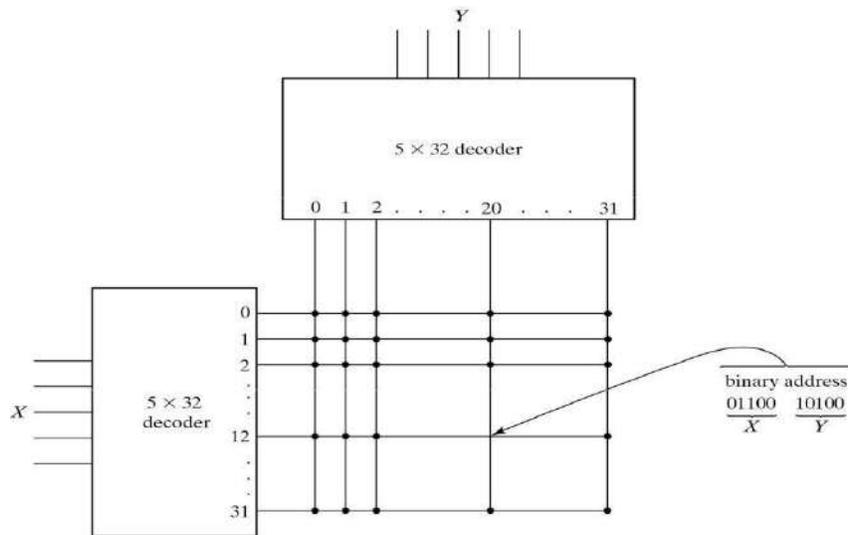


Fig. 7-7 Two-Dimensional Decoding Structure for a 1K-Word Memory

Address multiplexing

DRAMs typically have four times the density of SRAM. The cost per bit of DRAM storage is three to four times less than SRAM. Another factor is lower power requirement. Address multiplexing will reduce the number of pins in the IC package. In a two-dimensional array, the address is applied in two parts at different times, with the row address first and the column address second. Since the same set of pins is used for both parts of the address, so can decrease the size of package significantly.

Address multiplexing for 64K DRAM

After a time equivalent to the settling time of the row selection, RAS goes back to the 1 level. Registers are used to store the addresses of the row and column. CAS must go back to the 1 level before initialing another memory operation.

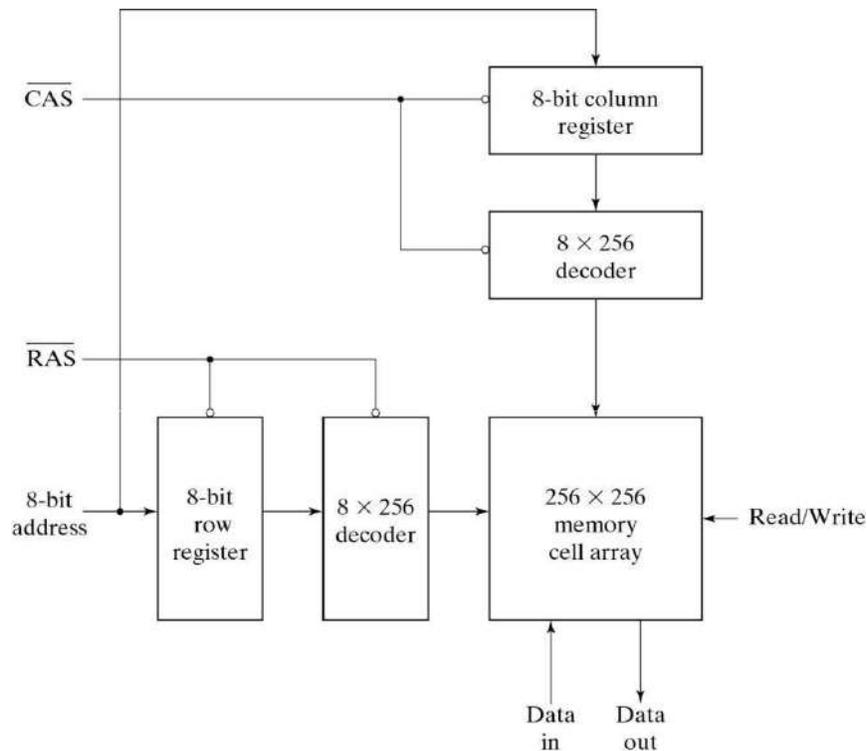


Fig. 7-8 Address Multiplexing for a 64K DRAM

Internal Structure of ROM

An array of semiconductor devices

- diodes
- transistors

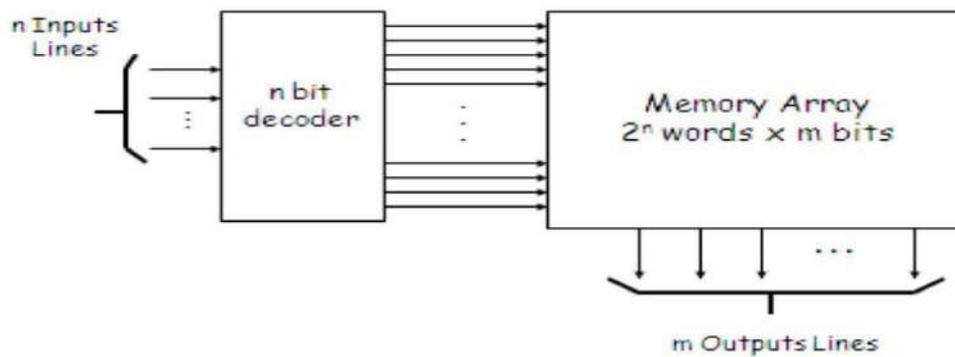
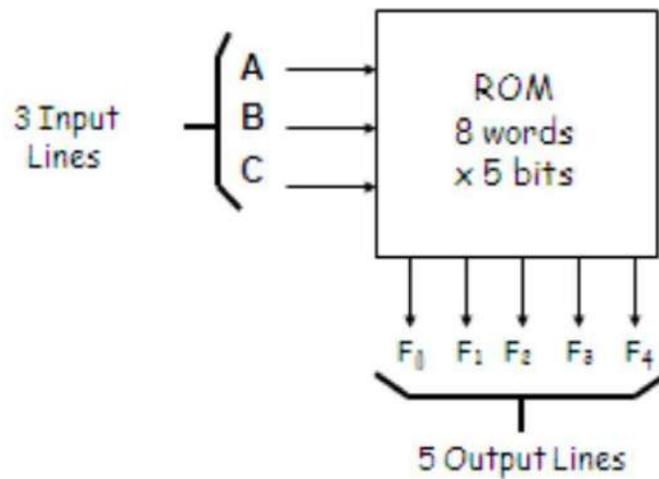
- field effect transistors 2^N

words by M bits

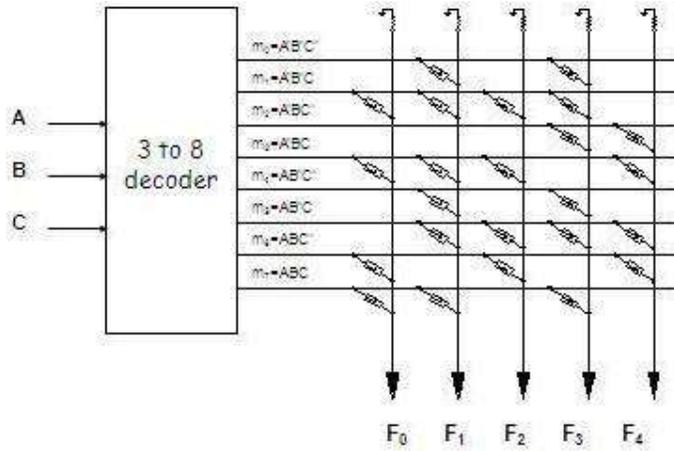
Data can be read but not changed

- (normal operating conditions)
- N input bits
- 2^N words by M bits
- Implement M arbitrary functions of N variables

Example 8 words by 5 bits:



ROM Memory Array



Alternate view

Each possible horizontal/vertical intersection indicates a possible connection. Or gates at bottom output the word selected by the decoder (32×8)

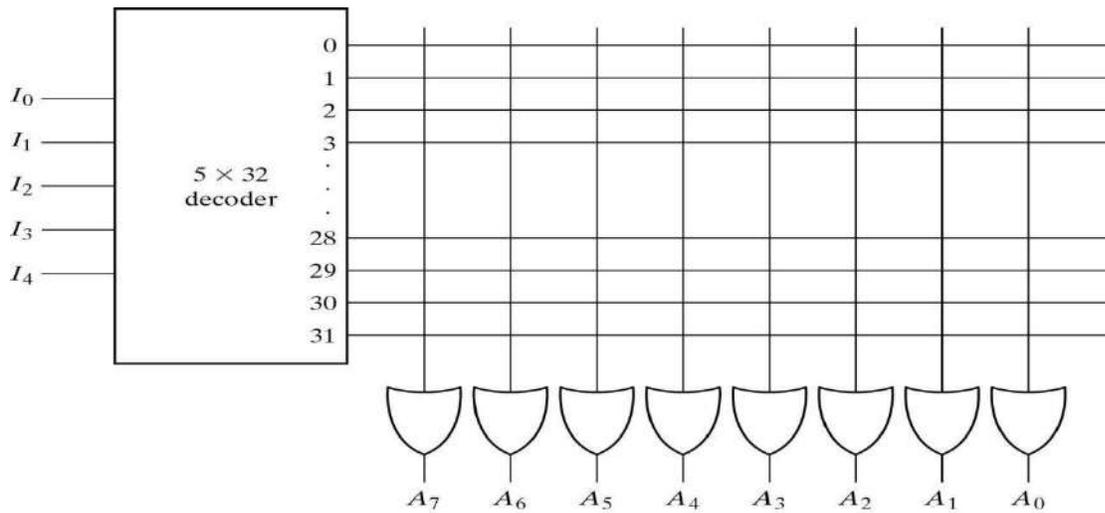


Fig. 7-10 Internal Logic of a 32×8 ROM

Commercial ROM types

| Type | Tech | ReadCyc | WrCyc | Comments |
|---------|---------|----------|--------------|---------------------|
| MASKROM | NMOS | 10-100ns | 4 weeks | Write once, low pwr |
| | CMOS | | | |
| MASKROM | Bipolar | <100ns | 4 weeks | Write once, h pwr |
| | | | | low density |
| PROM | Bipolar | <100ns | 10-50us/byte | Write once, h pwr |
| EPROM | NMOS | 25-200ns | 10-50us/byte | Reusable, low pwr |
| | CMOS | | | |
| EEPROM | NMOS | 50-200ns | 10-50us/byte | 10,000 to 100,000 |
| | | | | writes per location |

EPROM

Uses a floating gate for the FET at each bit location. User uses a programming voltage that causes a temporary breakdown in the dielectric between the gate and the floating gate to charge it. When programming voltage is removed the charge stays. How long? EPROM manufacturers -guaranteee properly programmed bit has 70% of charge after 10 years. Use UV light to erase.

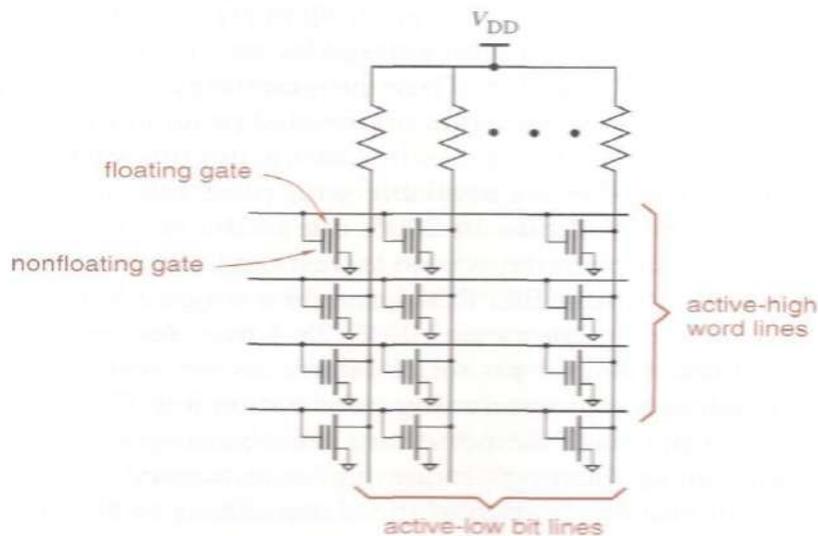


Figure 9-10
Storage matrix in an EPROM using floating-gate MOS transistors.

EEPROM

Electrically Erasable PROM. Like the EPROM only electrically erasable in circuit. Many times referred to a -flashl programmable memory. Very slow on writes so not a substitute for RAM.

General Block Diagram of ROM

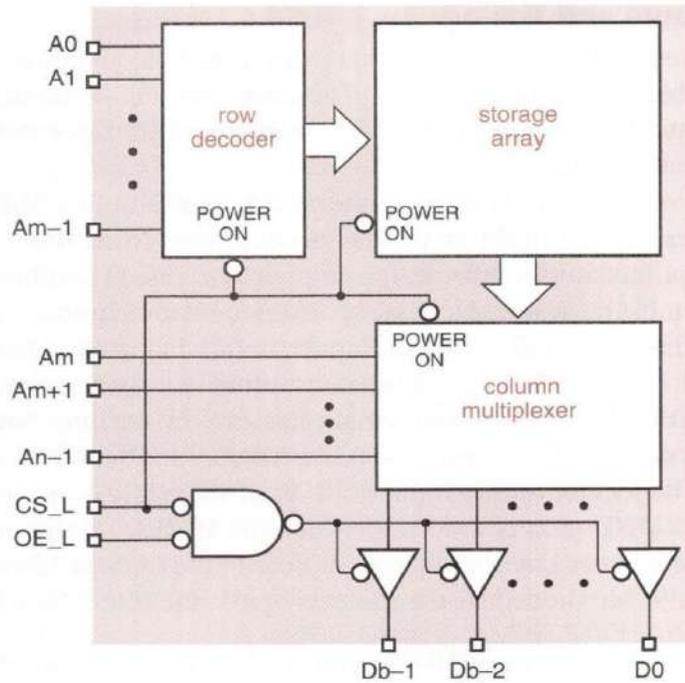
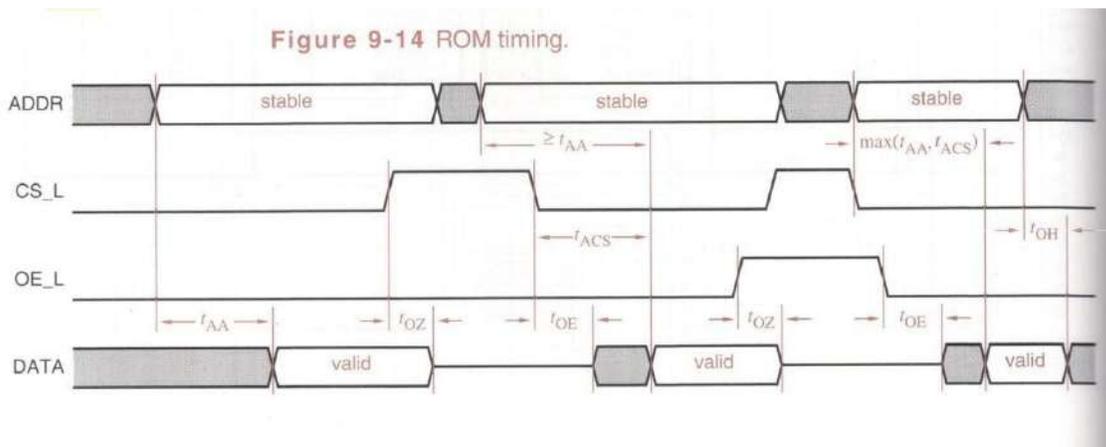


Figure 9-13
Internal ROM structure, showing use of control inputs.

Timing Diagram of ROM



- Access time from address – t_{AA}
- Access time from chip select - t_{ACS}
- Output-enable time - t_{OE}
- Output-disable time - t_{OZ}
- Output-hold time - t_{OH}

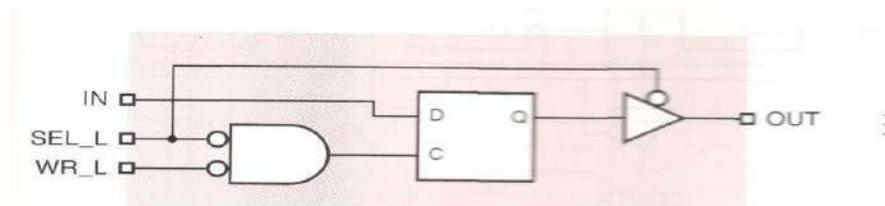
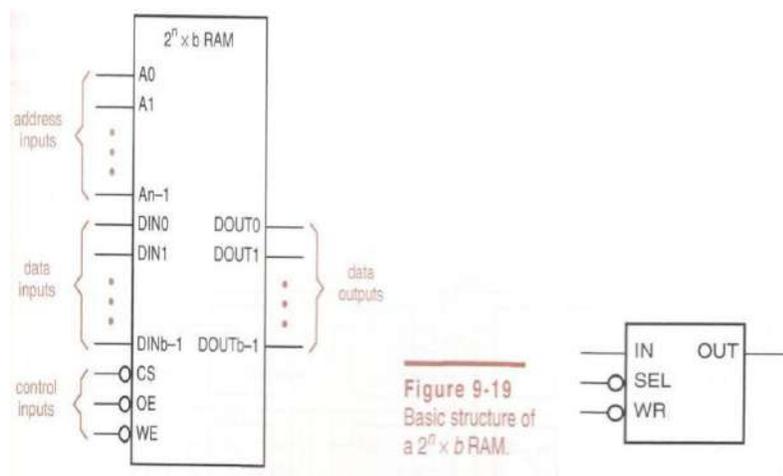
R/W Memory

Memory to store and retrieve data when more than F/Fs. A few types Static RAM – SRAM

As long as power is maintained data is held

SRAM

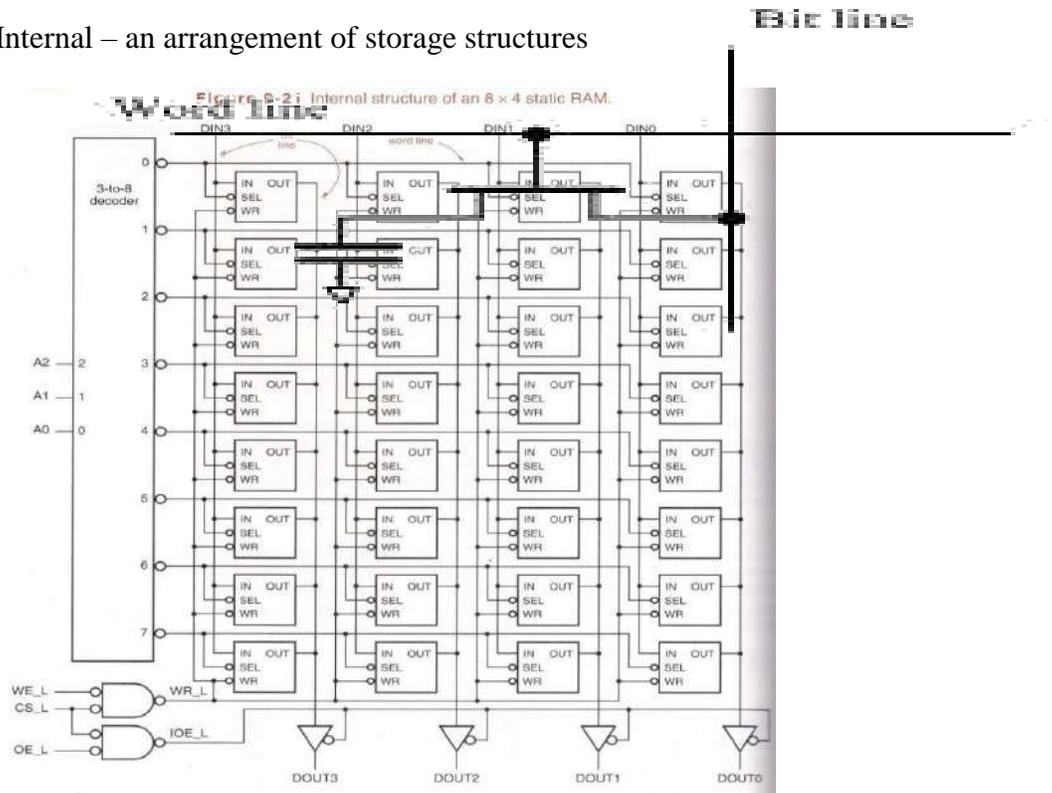
data storage



static RAM chip

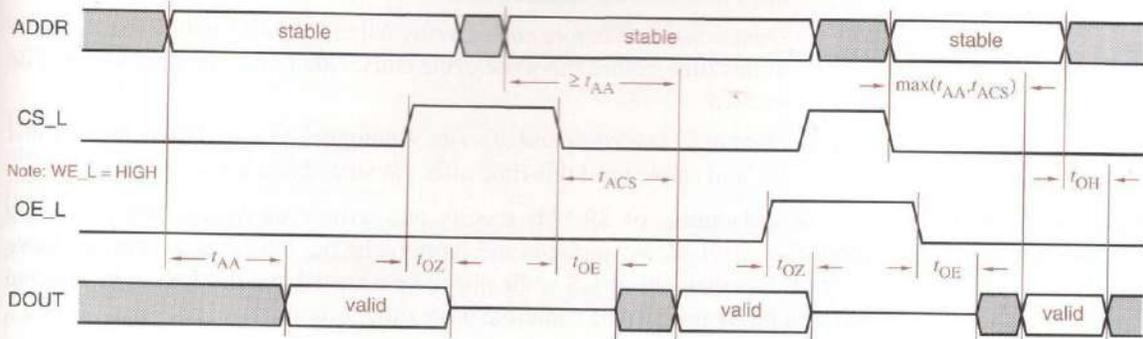
SRAM Timing

Internal – an arrangement of storage structures



Timing for read

Figure 9-22 Timing parameters for read operations in a static RAM.

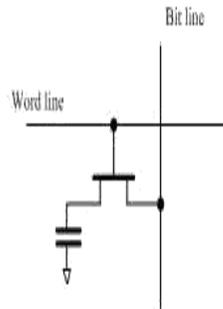


DRAM

Next step in memory is Synchronous SRAM which has a clocked interface for control, address and data. Then comes DRAM – dynamic ram. In DRAM data is stored in a semiconductor capacitor.

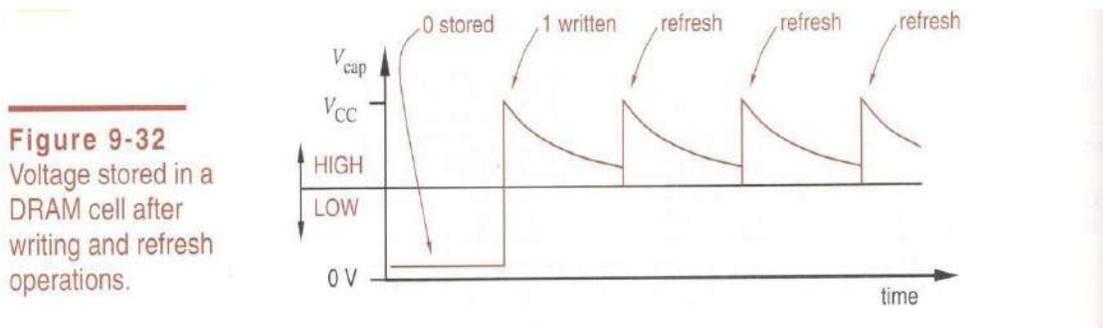
DRAM Read

A read sees the bit line precharged to high. The word line is then activated. If cell stores a 0 then there is a small drop on the voltage on the bit line. This is monitored by a sense amp which provides the value stored. Value must be written back after the read.



DRAM Refresh

Charge stored leaks off over time. Must restore the values stored a 4096 row DRAM it refresh every 64ms and thus each row every 15.6 usec. Larger DRAMs are banks of smaller.



DDR SDRAM

Double data rate SDRAM. Double the data transfer rate of an SDRAM by transferring on both edges of the clock. Access and setup times are the same as SRAM. Increased data throughput as data is transferred in blocks.

DRAM structure and operation

Write operation

Setting the word line to 1. To store a 1, a HIGH voltage is placed on the bit line, which charges the capacitor through the -onl transistor. To store a 0, a LOW voltage is placed on the bit line, which discharges the capacitor through the —onl transistor.

Read operation

The bit line is first precharged to a voltage halfway between HIGH and LOW. The word line is set HIGH so that the precharged bit line is pulled slightly higher or slightly lower. A sense amplifier detects this small change and recovers a 1 or 0 accordingly. Reading a DRAM cell destroys the original voltage stored on the capacitor, the DRAM cell must be written back the original data after reading.

Internal structure of a 64Kx1 DRAM

Multiplexed address inputs.

RAS_L: Row address strobe to store the higher order bits of the address into the *row-address register*.

CAS_L: Column address strobe to store the lower order bits of the address into the *column-address register*.

Row latches: the latches used to store data input/output from the memory array.

RAM Cells

Static RAM (SRAM):

- The basic element of a static RAM cell is the D-Latch.
- Data remains stored in the cell until it is intentionally modified.
- SRAM is fast (Access time: 1ns).
- SRAM needs more space on the semiconductor chip than DRAM.
 - SRAM more expensive than DRAM
 - SRAM needs more space than DRAM
- SRAM consumes power only when accessed.
- SRAM is used as a *Cache*

Dynamic RAM (DRAM):

- DRAM stores data in the form of electric charges in capacitors.
- Charges leak out, thus need to refresh data every few ms.
- DRAM is slow (Access time: 60ns).
- DRAM needs less space on the semiconductor chip than SRAM.
 - DRAM less expensive than SRAM
 - DRAM needs less space than SRAM
- DRAM needs to be refreshed
- DRAM is used as the main memory

Types of semiconductor memory devices: **Static RAM**

Static RAM (also called SRAM) devices retain their data for as long as the DC power is applied. The most common family of SRAM are the 61XXX, 62XXX or the CMOS

62CXXX series, where XXX indicates the memory capacity in Kbits. Some members of this family are the following:

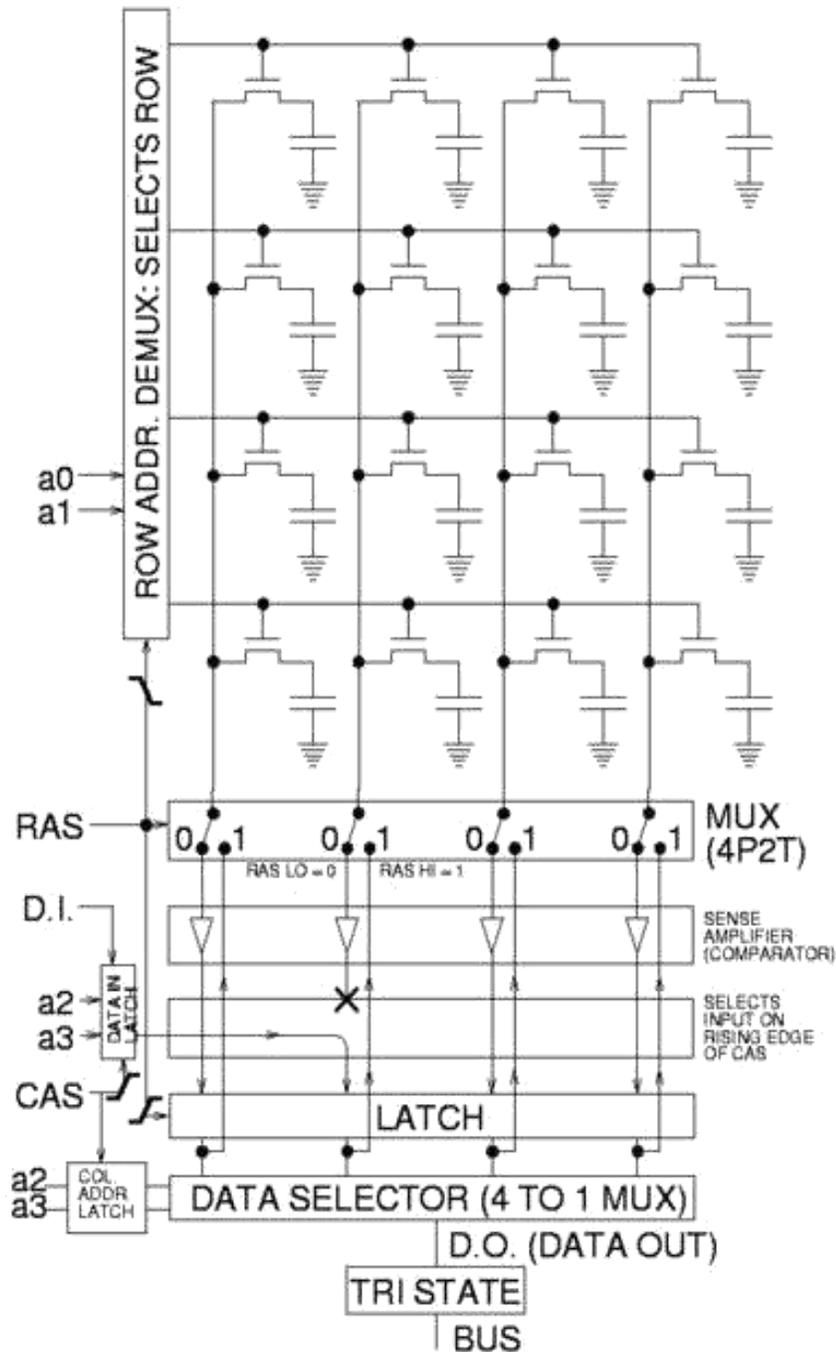
| | | |
|------------------|--------------|----------|
| 6116/6216 Kx8) | 164/6264 | Kx8) |
| 61256/62256 Kx8) | 11024/621024 | (128Kx8) |

These series of SRAM devices are pin compatible with the 27XXX series of EPROMs, with the difference that the WR signal is replaced by the programming voltage pin (Vpp) on the EPROM. This allows a single socket on the PCB hold either a SRAM, during system development, or an EPROM, after the operation of the program is verified to be the expected one. Static RAM is fast with access times much less than 100ns. SRAM chips with access times less than 10ns are often used as cache memory in computers.

DYNAMIC RAM CELL ARRAY

Asynchronous DRAM

This is the basic form, from which all others are derived. An asynchronous DRAM chip has power connections, some number of address inputs (typically 12), and a few (typically 1 or 4) bidirectional data lines. There are four [active low](#) control signals: /RAS, the Row Address Strobe. The address inputs are captured on the falling edge of /RAS, and select a row to open. The row is held open as long as /RAS is low. /CAS, the Column Address Strobe. The address inputs are captured on the falling edge of /CAS, and select a column from the currently open row to read or write. /WE, Write Enable. This signal determines whether a given falling edge of /CAS is a read (if high) or write (if low). If low, the data inputs are also captured on the falling edge of /CAS. /OE, Output Enable. This is an additional signal that controls output to the data I/O pins. The data pins are driven by the DRAM chip if /RAS and /CAS are low, and /WE is high, and /OE is low. In many applications, /OE can be permanently connected low (output always enabled), but it can be useful when connecting multiple memory chips in parallel.



DYNAMIC RAM

DRAM requires refreshing every 2 to 4 ms . Refreshing

occurs automatically during a read or write.

Internal circuitry takes care of refreshing cells that are not accessed over this interval.

– For a 256K X 1 DRAM with 256 rows, a refresh must occur every 15.6us (4ms/256).

- For the 8086, a read or write occurs every 800ns .
- This allows 19 memory reads/writes per refresh or 5% of the time.

DRAM technologies

- EDO DRAM
- SDRAM
- DRDRAM
- DDR DRAM

Soft errors occur on DRAMs which often require ERROR DETECTION and/or ERROR CORRECTION

A DRAM CONTROLLER is required for using DRAM

SYNCHRONOUS DYNAMIC RAM

In a synchronous DRAM, the control signals are synchronized with the system bus clock and therefore with the microprocessor. It allows *pipelined* read/write operations

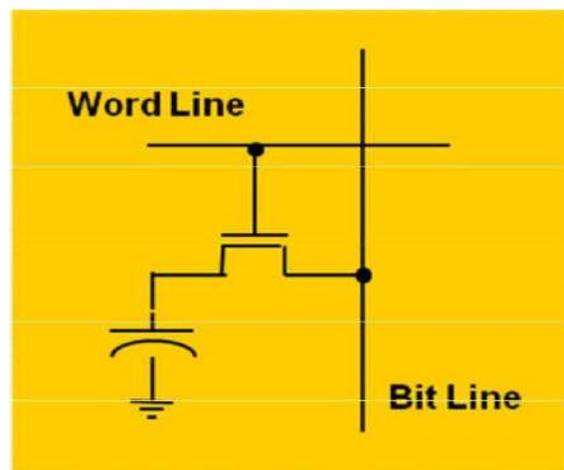
Double Data Rate (DDR) DRAM

An SDRAM type of memory where data are transferred on both the rising and the falling clock edge, effectively doubling the transfer rate without increasing the clock frequency. DDR-200 means a transfer rate of 200 million transfers per second, at a clock rate of 100 MHz. DDR1 upto 400 MHz. DDR2 standard allows higher clock frequencies.

DRAM

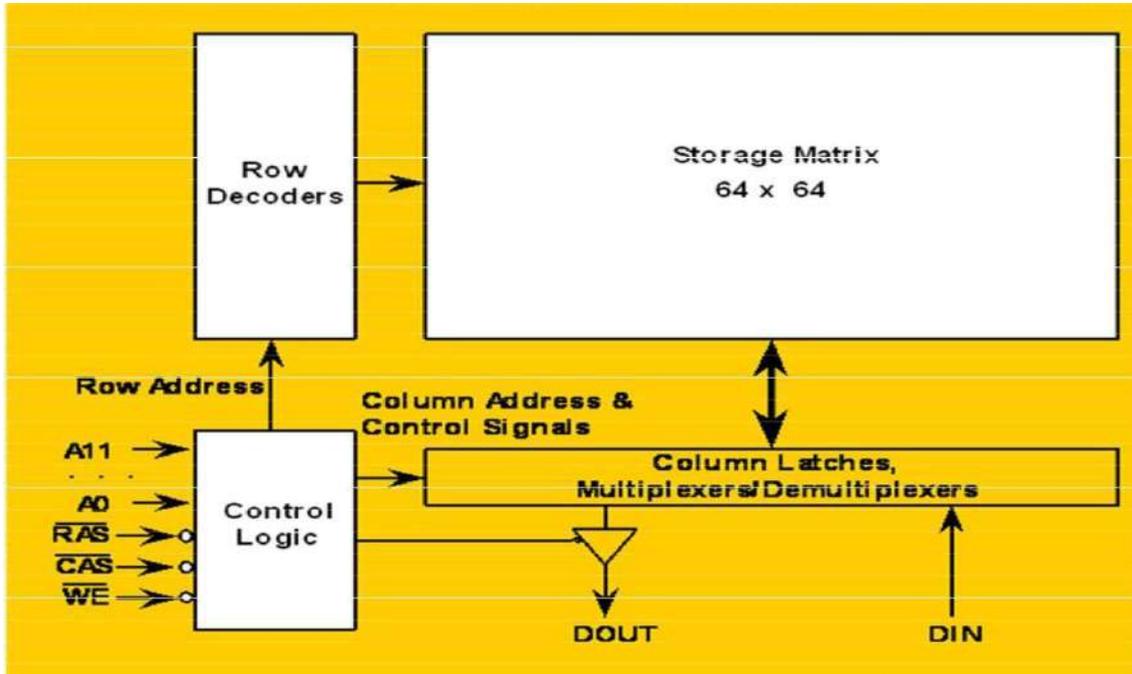
Refresh circuit : **storage decay in ms**

DRAMs take up much less space, typically ¼ the silicon area of SRAMs or less (one transistor and a capacitor)



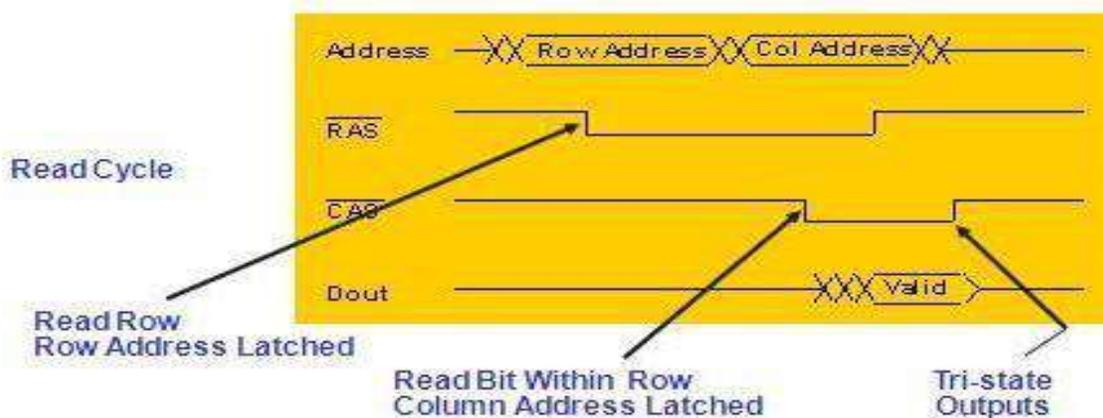
DRAM Organization

Long rows to simplify refresh. Two new signals: RAS, CAS. Row Address Strobe, Column Address Strobe replace Chip Select.

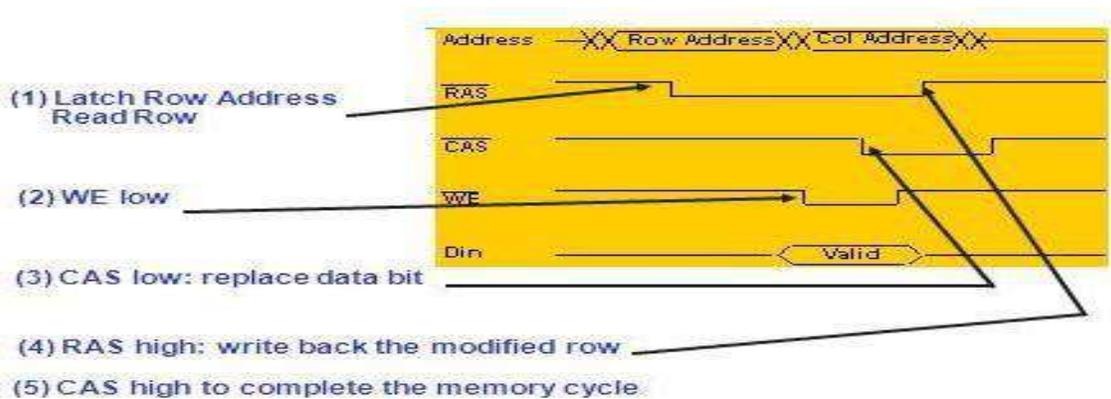


RAS, CAS Addressing

Even to read 1 bit, an entire 64-bit row is read! Separate addressing into two cycles: Row Address, Column Address. Saves on package pins, speeds RAM access for sequential bits!



Write cycle timing



RAM Refresh

Refresh Frequency :(4ms – 64ms)

- 4096 word RAM -- refresh each word once every 4 ms
- Assume 120ns memory access cycle
- This is one refresh cycle every 976 ns (1 in 8 DRAM accesses)!
- But RAM is really organized into 64 rows
- This is one refresh cycle every 62.5 ms (1 in 500 DRAM accesses)
- Large capacity DRAMs have 256 rows, refresh once every 16 ms RAS-only

Refresh (RAS cycling, no CAS cycling)

- External controller remembers last refreshed row

Some memory chips maintain refresh row pointer

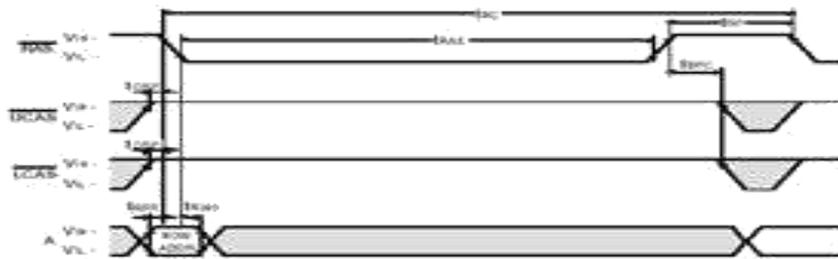
CAS before RAS refresh: if CAS goes low before RAS, then refresh

DRAM Technologies

- Conventional DRAM
- Fast Page Mode (FPM) DRAM
- Extended Data Out (EDO) DRAM
- Synchronous DRAM (SDRAM)

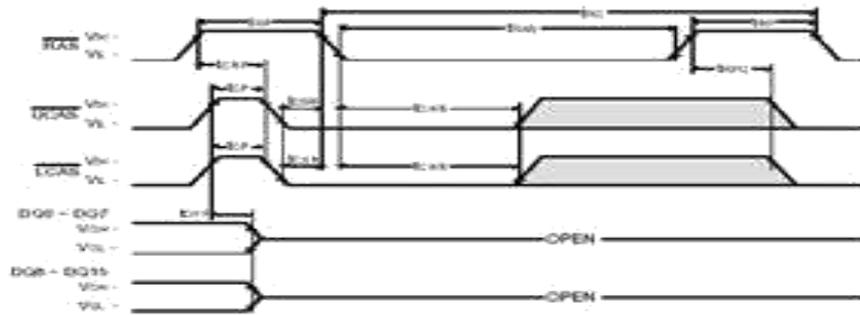
RAS - ONLY REFRESH CYCLE

NOTE: \bar{W} , \bar{CE} , \bar{Din} = Don't care
Dout = OPEN



CAS - BEFORE - RAS REFRESH CYCLE

NOTE: \bar{CE} , A = Don't care



Synchronous DRAM

Tied to the system clock

Burst mode

- System timing : 5-1-1-1
- Internal interleaving New memory

standard for modern PCs Speed

- Access time: 10ns, 12ns,...
- MHz rating: 100 MHz, 133MHz

Latency

- SDRAMs are still DRAMs
- 5-1-1-1 (10ns means the second, third and fourth access times) 2-clock and 4-

clock Circuitry

- 2-clock: 2 different DRAM chips on the module

□ □4-clock: 4

different DRAM chips

Packaging

- Usually comes in DIMM packaging
- Buffered and unbuffered, 3.3 V and 5.0V

Comparison of semiconductor memories

| | Bit.org. | Cell size (mm ²) | Chip size (mm ²) | Cell 密度 (%) | Real Access/cycle | Write cycle | Erase time | Write 集中 | Power consumption (Act./Stdby) |
|-----------|-----------------|------------------------------|------------------------------|-------------|-------------------|-------------|------------|----------------------------------|--------------------------------|
| 64M DRAM | 8Mx8b 8k ref | 1.7 | 211 | 0.52 | 38/100 (ns) | 100 (ns) | - | | 85/ (mA) |
| 16M SRAM | 2Mx8b | 8.4 | 226 | 0.59 | 14/33 (ns) | 14 (ns) | - | | 70/ (mA) |
| 64M Flash | 4Mx16b | 1.7 | 257 | 0.42 | 50/100 (ns) | 6.4 (us) | 0.8s | 10 ⁵ ~10 ⁶ | 30/0.1 (mA) |

* 0.4 mm design rule