G. PULLAIAH COLLEGE OF ENGINEERING AND TECHNOLOGY
Accredited by NAAC with 'A' Grade of UGC, Approved by AICTE, New Delhi
Permanently Affiliated to JNTUA, Ananthapuramu
(Recognized by UGC under 2(f) and 12(B) & ISO 9001:2008 Certified Institution)
Nandikotkur Road, Venkayapalli, Kurnool – 518452

Department of Electronics and Communication Engineering

Bridge Course
On
Digital System Design

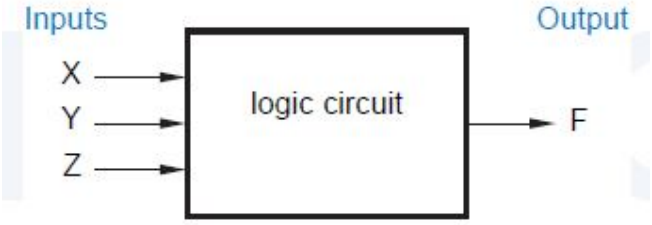
Logic Signals and Gates

A logic value, 0 or 1, is often called a *binary digit*, or *bit*. Digital designers often use the words "LOW" and "HIGH" in place of "0" and "1".

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*. 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 1values, 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.

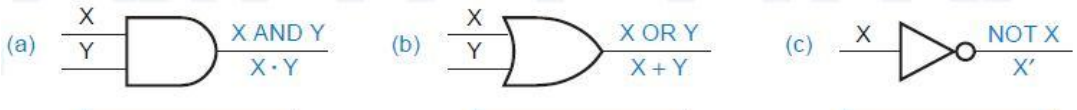
Table is the truth table for a logic circuit with three inputs X, Y, and Z and a single output F.

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.



X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

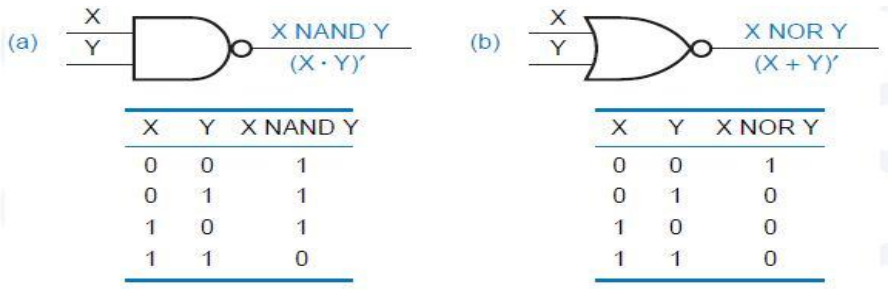
X	NOT X
0	1
1	0

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.

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.



X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0

X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0

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.

BIPOLAR LOGIC AND INTERFACING Diode Logic:

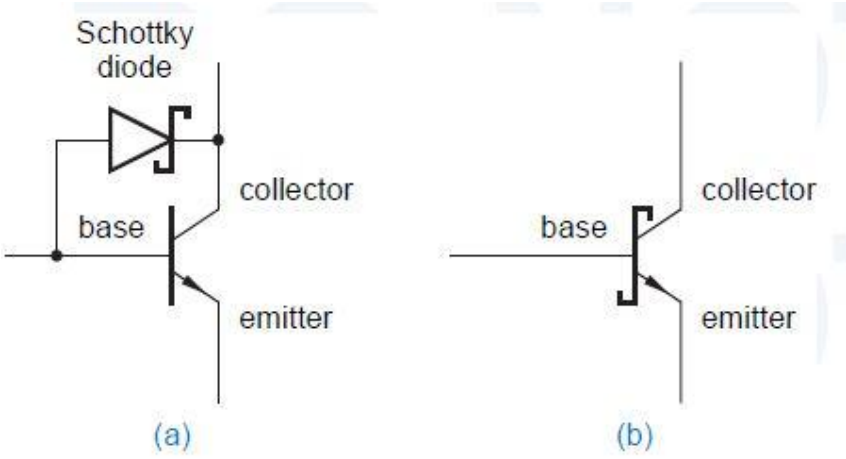
Diode action can be exploited to perform logical operations. Within the 5-volt range, signal voltages are partitioned into two ranges, LOW and HIGH, with a 1-volt noise margin between. A voltage in the LOW range is considered to be a logic 0, and a voltage in the HIGH range is a logic 1.

Bipolar Junction Transistors

A *bipolar junction transistor* is a three-terminal device that, in most logic circuits, acts like a current-controlled switch. If we put a small current into one of the terminals, called the *base*, then the switch is "on"—current may flow between the other two terminals, called the *emitter* and the *collector*. If no current is put into the base, then the switch is "off"—no current flows between the emitter and the collector.

Schottky Transistors

When the input of a saturated transistor is changed, the output does not change immediately; it takes extra time, called *storage time*, to come out of saturation. In fact, storage time accounts for a significant portion of the propagation delay in the original TTL logic family. Storage time can be eliminated and propagation delay can be reduced by ensuring that transistors do not saturate in normal operation. Contemporary TTL logic families do this by placing a *Schottky diode* between the base and collector of each transistor that might saturate, as shown in Figure The resulting transistors, which do not saturate, are called *Schottky-clamped transistors* or *Schottky transistors* for short. When forward biased, a Schottky diode's voltage drop is much less than a standard diode's, 0.25 V vs. 0.6 V. In a standard saturated transistor, the base-tocollector voltage is 0.4 V, as shown in Figure .



Transistor-Transistor Logic

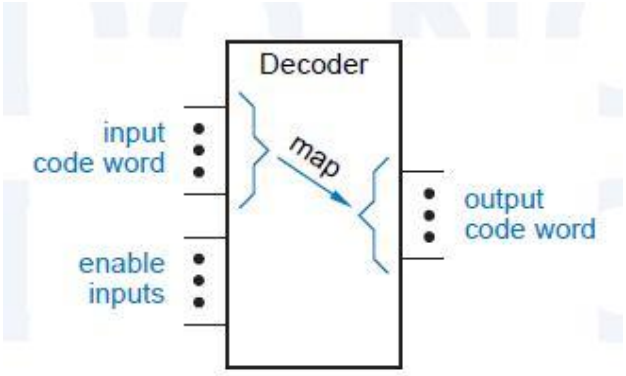
The most commonly used bipolar logic family is transistor-transistor logic. Actually, there are many different TTL families, with a range of speed, power consumption, and other characteristics. The circuit examples in this section are
based on a representative TTL family, Low-power Schottky (LS or LS-TTL). TTL families use basically the same logic levels as the TTL-compatible CMOS families in previous sections. We'll use the following definitions of LOW
and HIGH in our discussions of TTL circuit behavior: LOW 0–0.8 volts.
HIGH 2.0–5.0 volts.

Basic TTL NAND Gate

The circuit diagram for a two-input LS-TTL NAND gate, part number 74LS00, is shown in Figure 3-75. The NAND function is obtained by combining a diode AND gate with an inverting buffer amplifier. The circuit's operation is best

COMBINATIONAL LOGIC DESIGN- 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
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Table 1: 2 to 4 decoder

Table 1 is the truth table of a 2-to-4 decoder. The input code word 1,I0 represents an integer in the range 0–3. The output code word Y3,Y2,Y1,Y0 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 I1,I0.

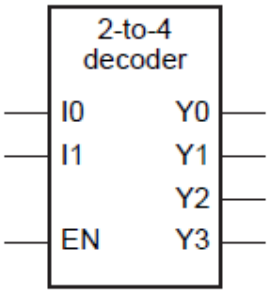


Fig 3: 2 to 4 decoder logic symbol

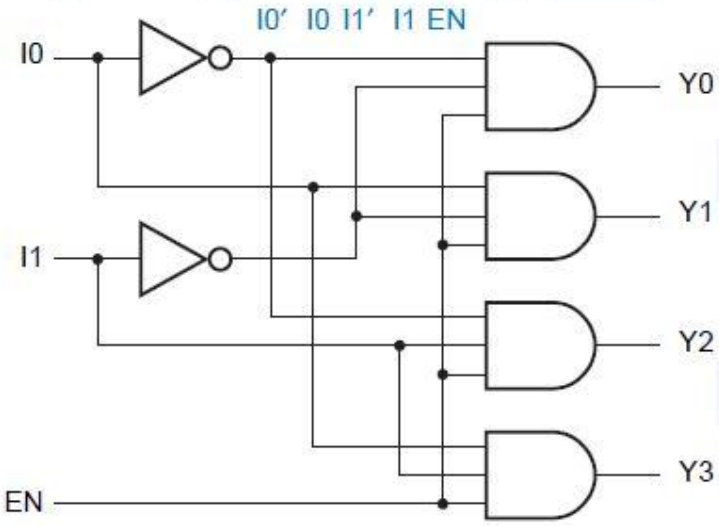
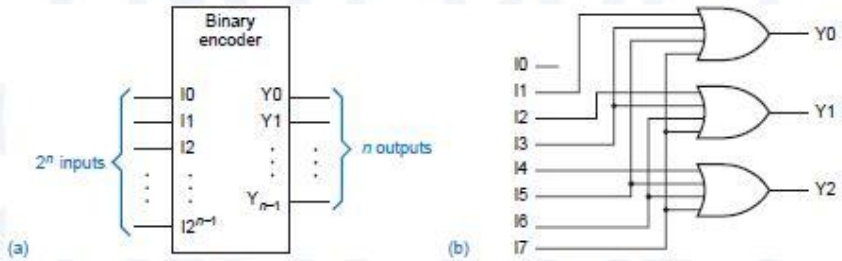


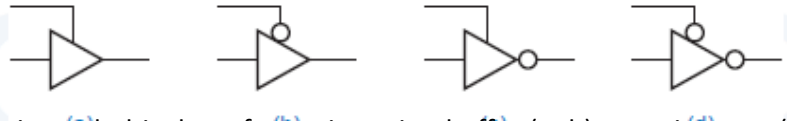
Fig 4: logic diagram of 2 to 4 decoder

The corresponding logic circuit is shown in (b). In general, a 2*n*-to-*n* encoder can be built from *n* 2*n*□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.



Three-State Devices

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.

5.7 Multiplexers

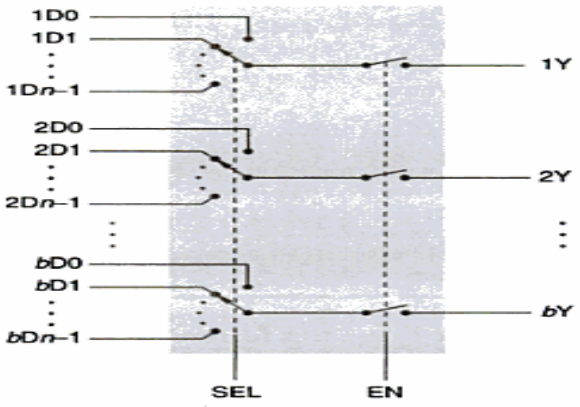
A *multiplexer* is a digital switch—it connects data from one of *n* sources to its output. Figure 5-61(a) shows the inputs and outputs of an *n*-input, *b*-bit multiplexer. There are *n* sources of data, each of which is *b* bits wide. A multiplexer is often called a *mux* for short.

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.

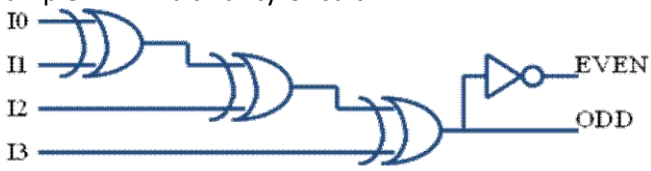


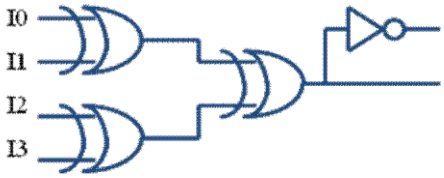
XOR GATE

Odd Parity Circuit : The output is 1 if odd number of inputs are 1

Even Parity Circuit : The output is 1 if even number of inputs are 1

Example : 4-bit Parity Circuit





Daisy-Chain Structure Tree structure

Input : 1101 Odd Parity output : 1

Even Parity output : 0

.

Four XOR gates are provided in a single 14-pin SSI IC, the *74x86* shown in Figure. New SSI logic families do not offer XNOR gates, although they are readily available in FPGA and ASIC libraries and as primitives in HDLs.

Parity Circuits

N XOR gates may be cascaded to form a circuit with *n* 1 inputs and a single output. This is called an *odd-parity circuit*, because its output is 1 if an odd number of its inputs are 1.

If the output of either circuit is inverted, we get an *even-parity circuit*, whose output is 1 if an even number of its inputs are 1.

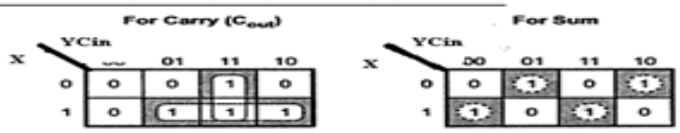
Half Adders and Full Adders

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:

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



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

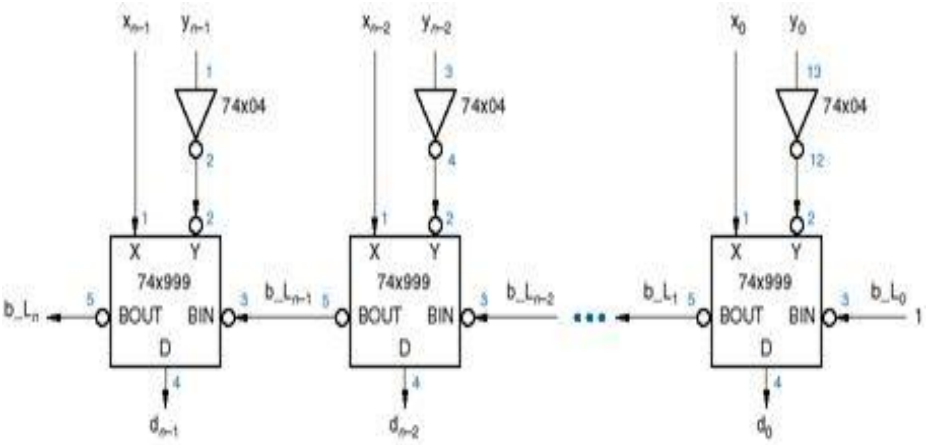
Subtraction : D = X - Y = X + (-Y) =

= X+ (Two's Complement of Y)

= X+ (One's Complement of Y) + 1

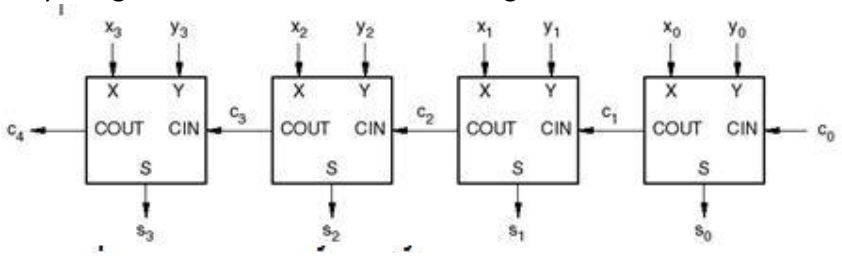
= X+ Y'+ 1

Using Adder as a Subtractor



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 (c0) 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 x0–x*i*, y0– y*i*, and c0, "multiplying out" or "adding out" to obtain a sum-ofproducts or product-of-sums expression, and building the corresponding ANDOR or OR-AND circuit. Unfortunately, beyond s2, 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 c0 = 0, a two-level AND-OR circuit for s2 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.

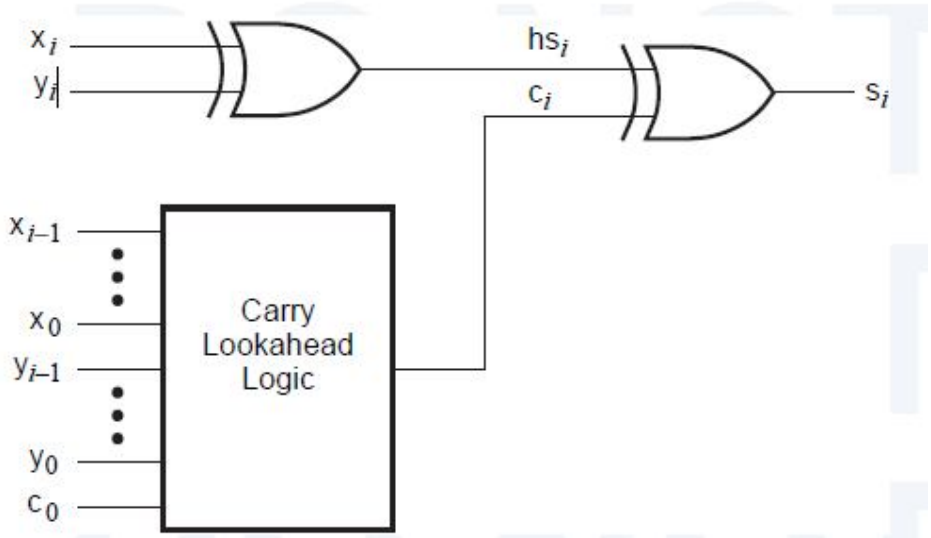


Carry Lookahead Adders

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 *xi* and *yi*, adder stage *i* is said to *generate* a carry if it produces a carry-out of 1 (c*i*1 1) independent of the inputs on x0 – x*i*1, y0 – y*i*1, and c0.

• For a particular combination of inputs *xi* and *yi*, adder stage *i* is said to *propagate* carries if it produces a carry-out of 1 (c*i*1 1) in the presence of an input combination of x0 – x*i*1, y0 – y*i*1, and c0 that causes a carry-in of



Corresponding to these definitions, we can write logic equations for a carrygenerate 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 "carr y lookahead" .

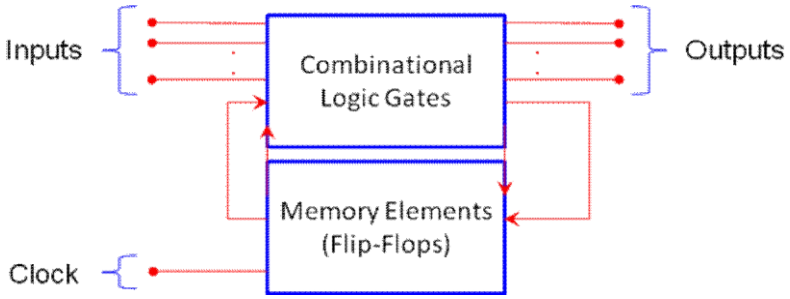
$$
\begin{aligned}
c_1 &= g_0 + p_0 \cdot c_0 \\
c_2 &= g_1 + p_1 \cdot c_1 \\
&= g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) \\
&= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \\
c_3 &= g_2 + p_2 \cdot c_2 \\
&= g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0) \\
&= g_2 + p_2 \cdot g_1 + p_2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

Latches and Flip flops

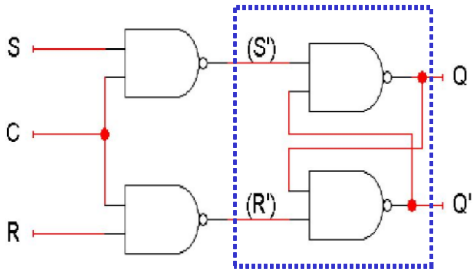
***Latches and flip-flops* (FFs) are the basic building blocks of sequential circuits.**

 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.



- The control input acts just like an enable.



C	S	R	S'	R'	Q
0	x	x	1	1	No change
1	0	0	1	1	No change
1	0	1	1	0	0 (reset)
1	1	0	0	1	1 (set)
1	1	1	0	0	Avoid!

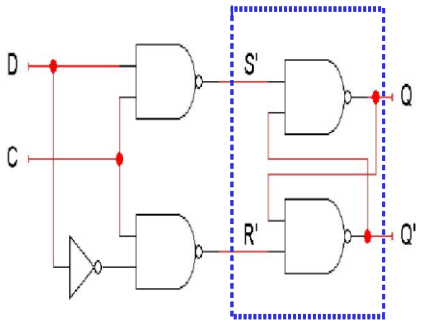
D latch

- 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!



C	D	Q
0	x	No change
1	0	0
1	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.

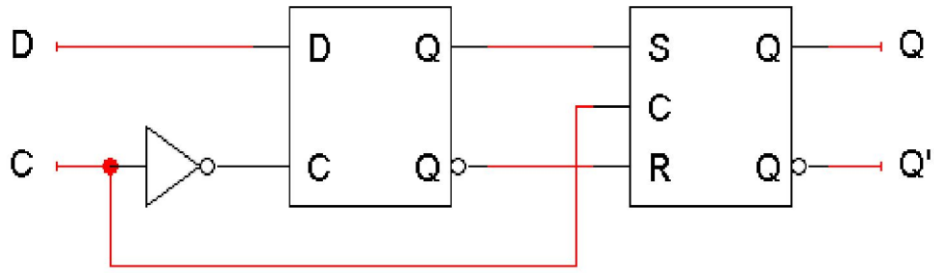
- Note the layout here.

– 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.

MASTER SLAVE



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.

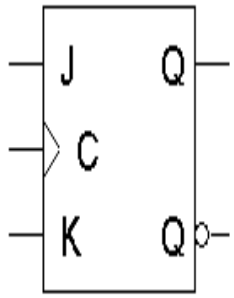
The D flip-flop's behavior is similar to that of a D latch except for the positive edge-triggered nature, which is not explicit in this table

C	D	Q
0	x	No change
1	0	0 (reset)
1	1	1 (set)

Flip-flop variations

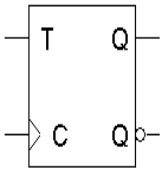
- We can make different versions of flip-flops based on the D flip-flop, just like we made different latches based on the S'R' latch.

- 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.



C	J	K	Q_{next}
0	x	x	No change
1	0	0	No change
1	0	1	0 (reset)
1	1	0	1 (set)
1	1	1	$Q'_{current}$

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



C	T	Q_{next}
0	x	No change
1	0	No change
1	1	$Q'_{current}$

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.

Q(t+1) = D

D	Q(t+1)	Operation
0	0	Reset
1	1	Set

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

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

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

= T ⊕ Q(t)

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

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.

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.

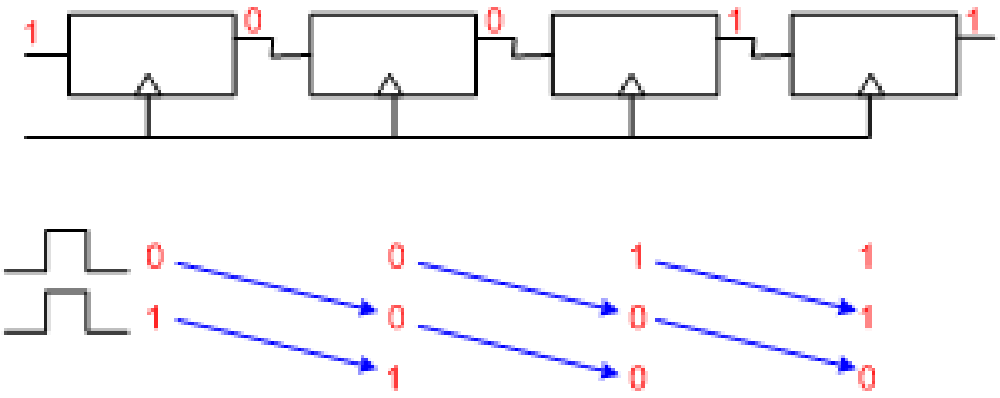
Asynchronous Counters

- This counter is called *asynchronous* because not all flip flops are hooked to the same clock.

Shift Register

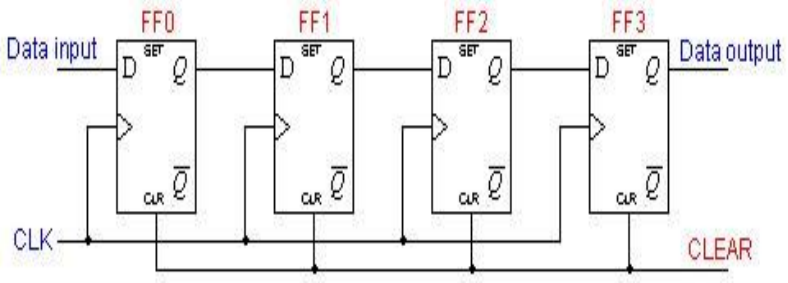
Shift registers are a type of sequential logic circuit, mainly for storage of digital data. They are a group of flip-flops connected in a chain so that the output from one flip-flop becomes the input of the next flip-flop. Most of the registers possess no characteristic internal sequence of states. All 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:



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.



CLEAR

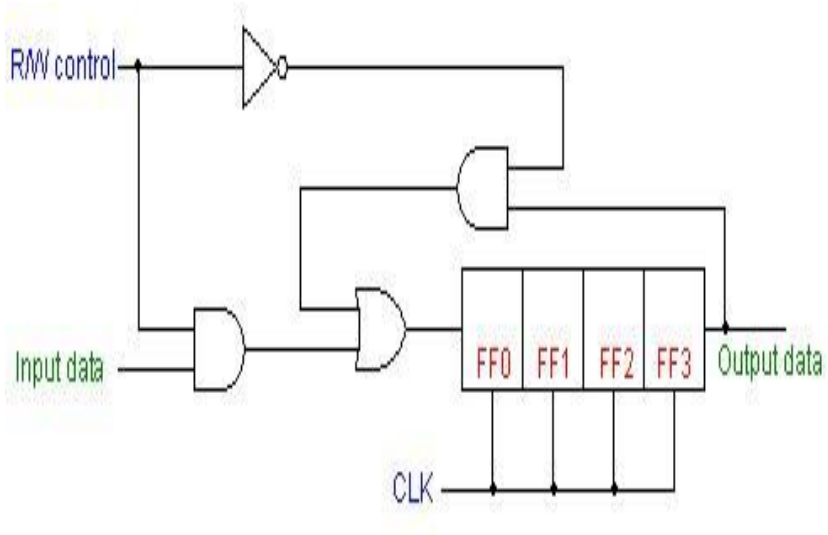
FF0	FF1	FF2	FF3
0	0	0	0

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.

CLEAR

FF0	FF1	FF2	FF3
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



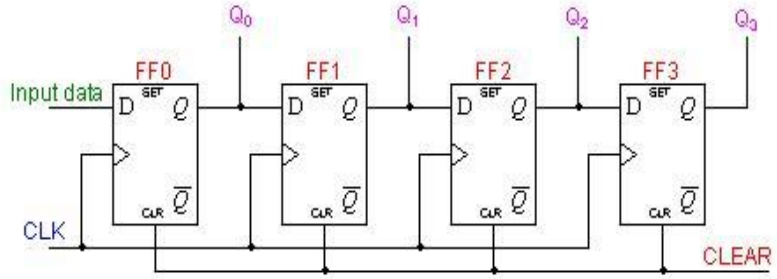
The data is loaded to the register when the control line is HIGH (ie WRITE). The data can be shifted out of the register when the control line is LOW (ie READ). This is shown in the animation below

WRITE
1001

FF0	FF1	FF2	FF3
0	0	0	0

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.



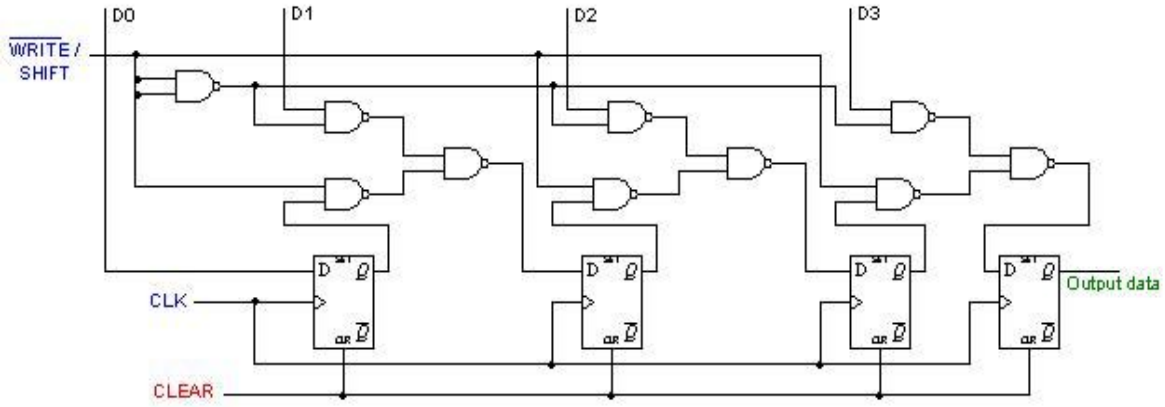
In the animation below, we can see how the four-bit binary number 1001 is shifted to the Q outputs of the register.

CLEAR
1001

Q0	Q1	Q2	Q3
0	0	0	0

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.



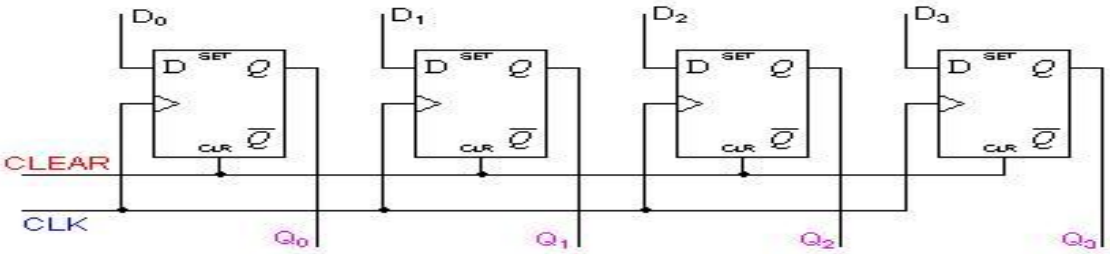
D0, D1, D2 and D3 are the parallel inputs, where D0 is the most significant bit and D3 is the least significant bit. To write data in, the mode control line is taken to LOW and the data is clocked in. The data can be shifted when the mode control line is HIGH as SHIFT is active high. The register performs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

CLEAR

Q0	Q1	Q2	Q3
0	0	0	0

Parallel In - Parallel Out Shift Register

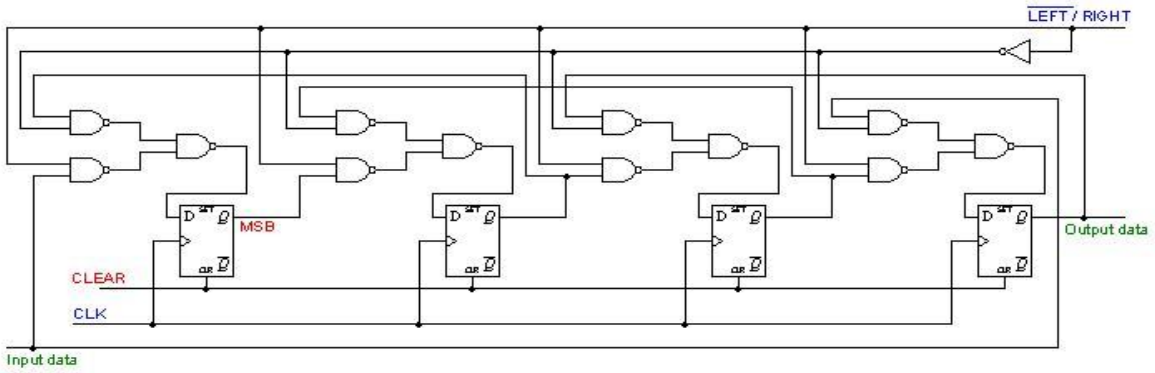
For parallel in - parallel out shift registers, all data bits appear on the parallel outputs immediately following the simultaneous entry of the data bits. The following circuit is a four-bit parallel in - parallel out shift register constructed by D flip-flops.



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**!**

RIGHT 11111001	FF0	FF1	FF2	FF3
	0	0	0	0

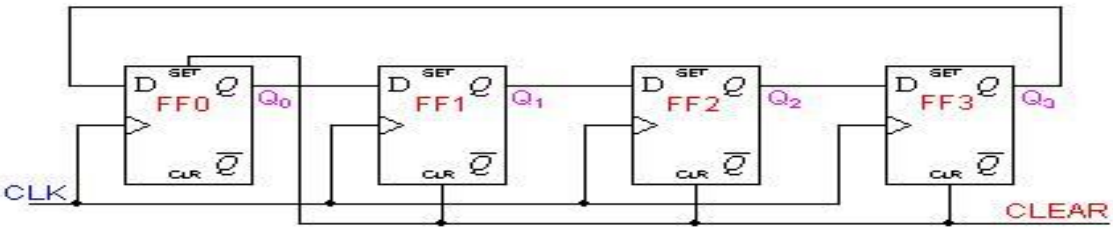
Shift Register Counters

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



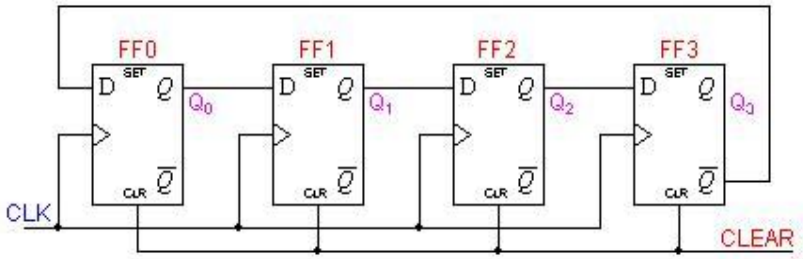
Since the count sequence has 4 distinct states, the counter can be considered as a mod-4 counter. Only 4 of the maximum 16 states are used, making ring counters very inefficient in terms of state usage. But the major advantage of a ring counter over a binary counter is that it is self-decoding. No extra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

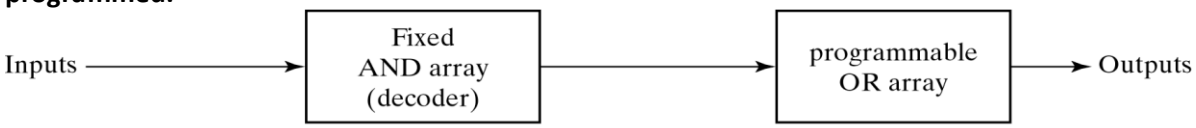
7. **Step 7:** Produce the logic equation and draw a block diagram (as in the case of combinational systems).

Combinational PLDs

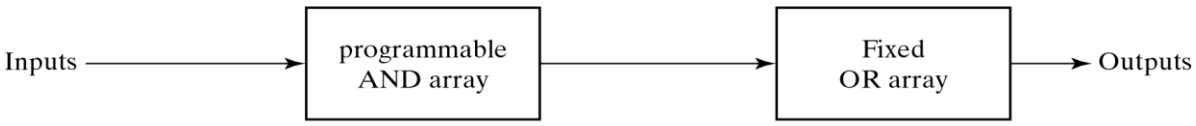
- **A combinational PLD is an integrated circuit with programmable gates divided into an AND array and an OR array to provide an AND-OR sum of product implementation.**

- **PROM: fixed AND array constructed as a decoder and programmable OR array.**

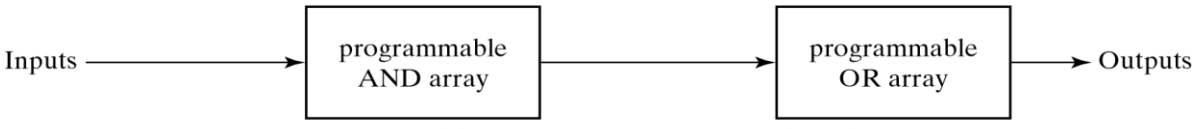
- **PAL: programmable AND array and fixed OR array. PLA: both the AND and OR arrays can be programmed.**



(a) Programmable read-only memory (PROM)



(b) Programmable array logic (PAL)



(c) Programmable logic array (PLA)

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⊕1 = x'). The output doesn't change and connect to 0 (since x⊕0 = x).

F1 = AB'+AC+A'BC' F2 = (AC+BC)'

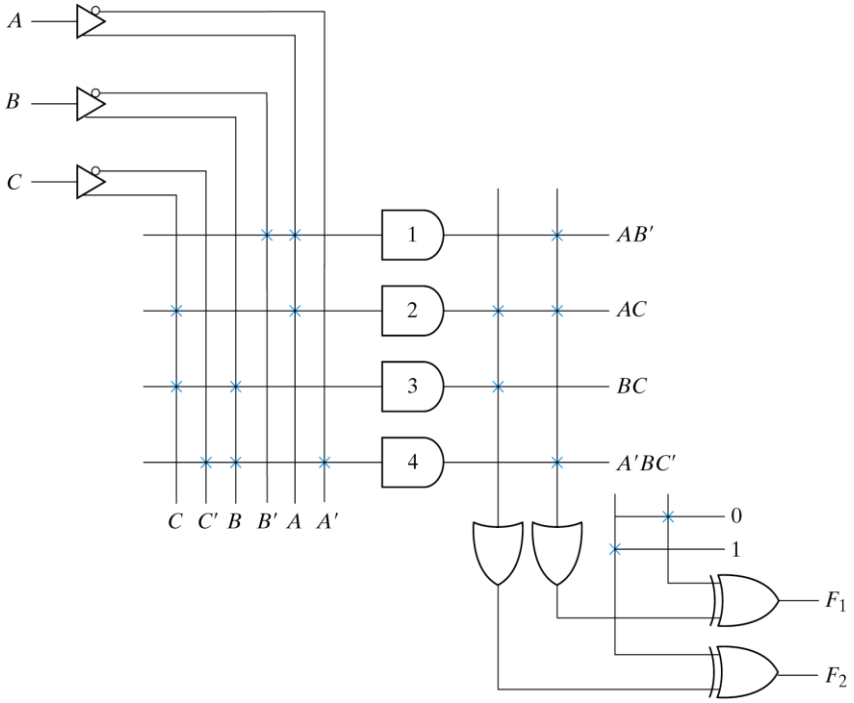


Fig. 7-14 PLA with 3 Inputs, 4 Product Terms, and 2 Outputs