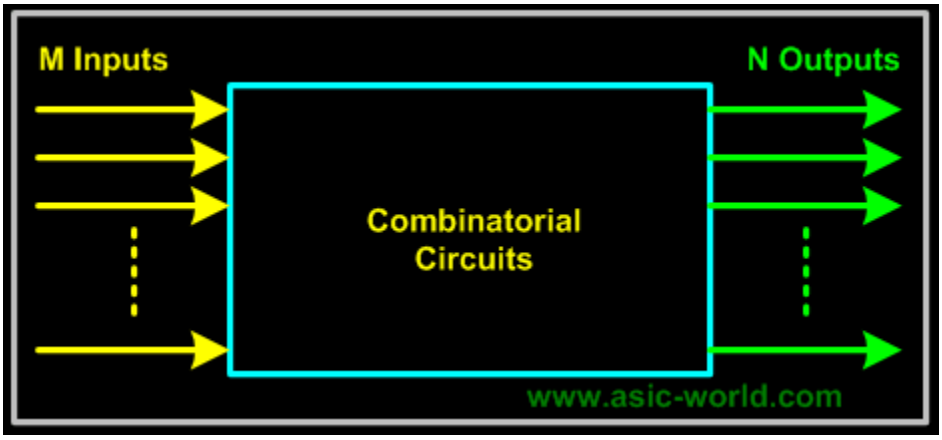
UNIT-II

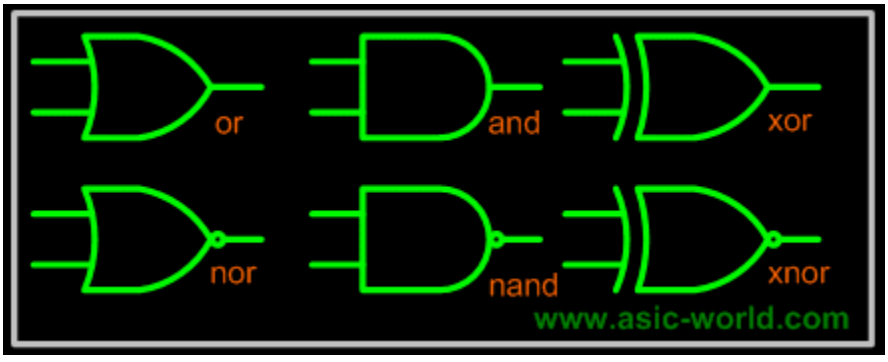
Combinational Circuit

Introduction

Combinatorial Circuits are circuits which can be considered to have the following genericstructure.



Whenever the same set of inputs is fed in to a combinatorial circuit, the same outputs will be generated. Such circuits are said to be stateless. Some simple combinational logic elements that we have seen in previous sections are"Gates".



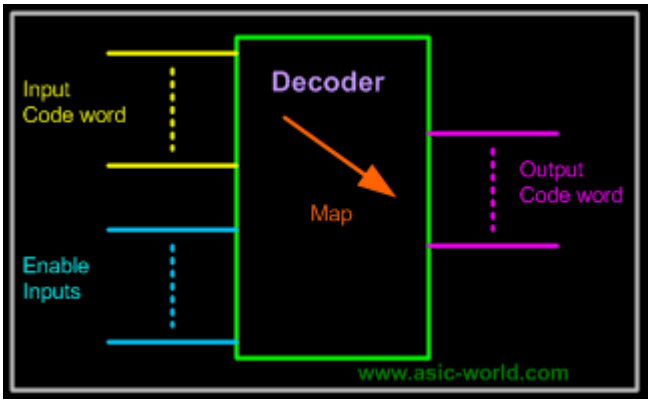
All the gates in the above figure have 2 inputs and one output; combinational elements simplest form are "not" gate and "buffer" as shown in the figure below. They have only one input and oneoutput.

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; e.g. n-to-2n, BCD decoders.

Enable inputs must be on for the decoder to function, otherwise its outputs assume a single "disabled" output codeword.

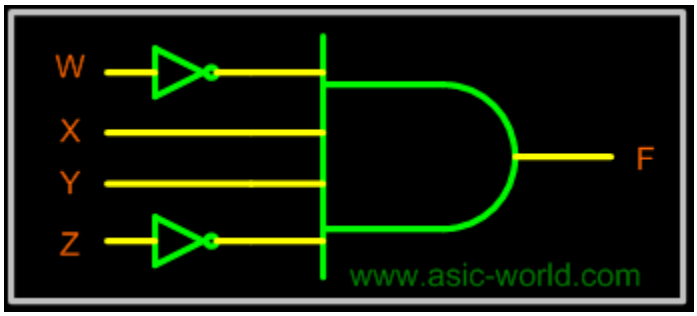
Decoding is necessary in applications such as data multiplexing, 7 segment display and memory address decoding. Figure below shows the pseudo block of adecoder.



Basic BinaryDecoder

And AND gate can be used as the basic decoding element, because its output is HIGH only when all its inputs are HIGH. For example, if the input binary number is 0110, then, to make all the inputs to the AND gate HIGH, the two outer bits must be inverted using

two inverters as shown in figurebelow.



A binary decoder has n inputs and 2^n outputs. Only one output is active at any one time, corresponding to the input value. Figure below shows a representation of Binary n-to-2^n decoder

Example - 2-to-4 BinaryDecoder

A 2 to 4 decoder consists of two inputs and four outputs, truth table and symbols of which is shownbelow.

TruthTable

X	Y	F0	F1	F2	F3

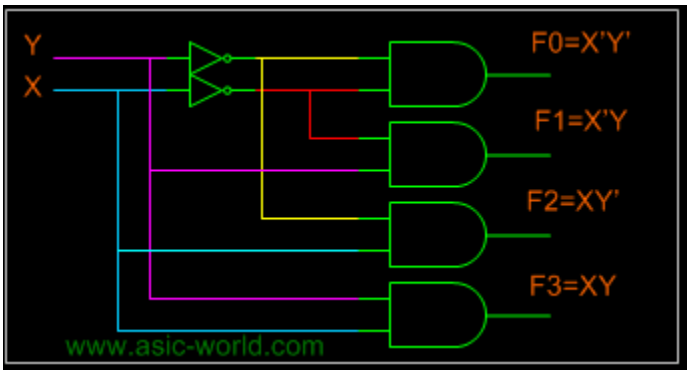
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Symbol

To minimize the above truth table we may use kmap, but doing that you will realize that it is a waste of time. One can directly write down the function for each of the outputs. Thus we can draw the circuit as shown in figurebelow.

Note: Each output is a 2-variable minterm (X'Y', X'Y, XY',XY)

Circuit



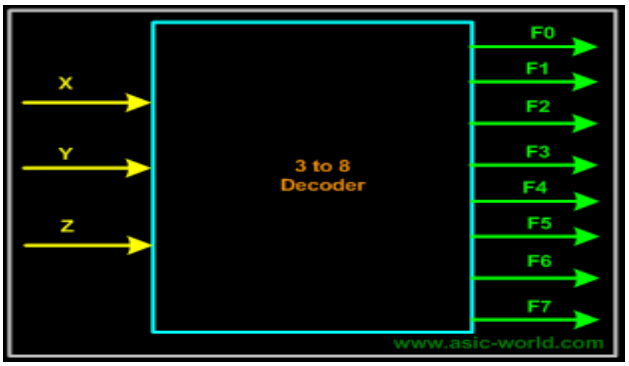
Example - 3-to-8 BinaryDecoder

A 3 to 8 decoder consists of three inputs and eight outputs, truth table and symbols of which is shownbelow.

TruthTable

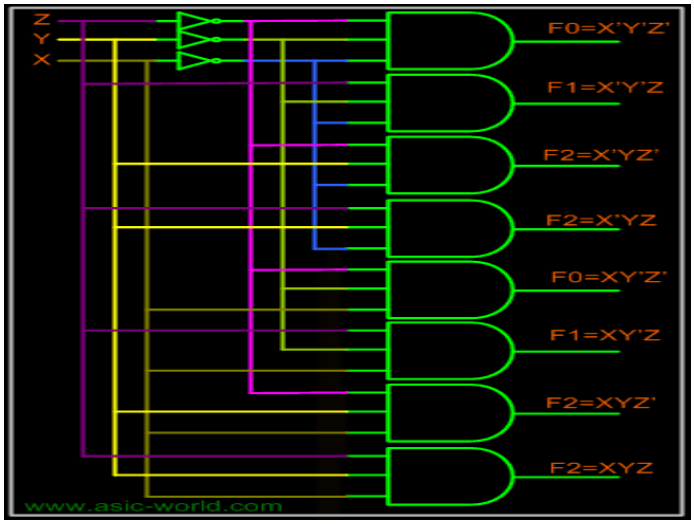
X	Y	Z	F0	F1	F2	F3	F4	F5	F6	F7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Symbol



From the truth table we can draw the circuit diagram as shown in figurebelow

Circuit



Implementing Functions UsingDecoders

- Any n-variable logic function, in canonical sum-of-minterms form can be implemented using a single n-to-2^n decoder to generate the minterms, and an OR gate to form thesum.
  - The output lines of the decoder corresponding to the mintermsofthefunction are used as inputs to the orgate.
- Any combinational circuit with n inputs and m outputs can be implemented with an n-to-2^n decoder with m ORgates.
- Suitable when a circuit has many outputs, and each output function is expressed with fewminterms.

Example -Fulladder

Equation

$S(x, y, z) = \Sigma(1,2,4,7)$

$C(x, y, z) = \Sigma(3,5,6,7)$

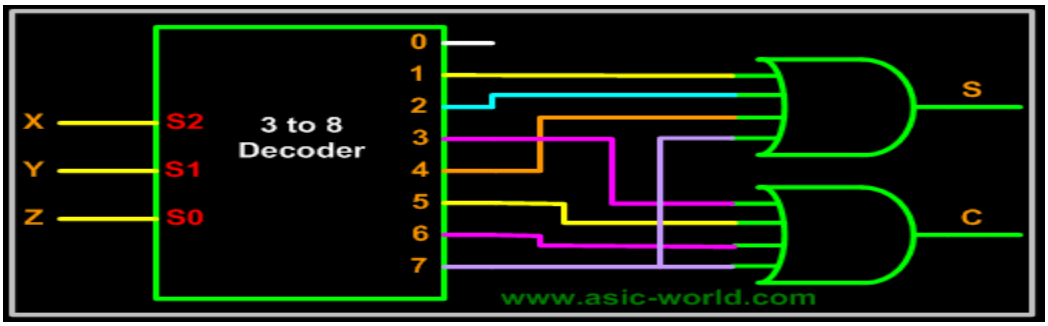
TruthTable

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0

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

From the truth table we know the values for which the sum (s) is active and also the carry (c) is active. Thus we have the equation as shown above and a circuit can be drawn as shown below from the equationderived.

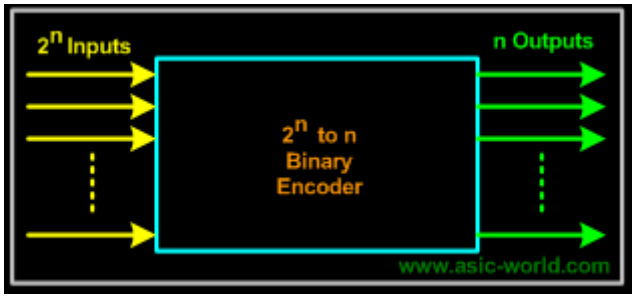
Circuit



Encoders

An encoder is a combinational circuit that performs the inverse operation of a decoder. If a device output code has fewer bits than the input code has, the device is usually called an encoder. e.g. 2^n-to-n, priorityencoders.

The simplest encoder is a 2^n-to-n binary encoder, where it has only one of 2^n inputs = 1 and the output is the n-bit binary number corresponding to the activeinput.



Example - Octal-to-BinaryEncoder

Octal-to-Binary take 8 inputs and provides 3 outputs, thus doing the opposite of what the 3-to-8 decoder does. At any one time, only one input line has a value of 1. The figure below shows the truth table of an Octal-to-binaryencoder.

TruthTable

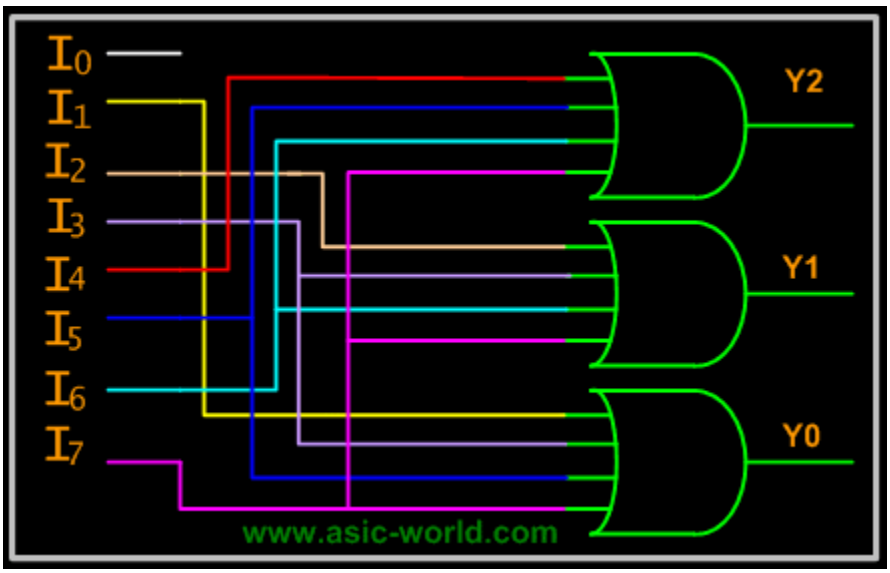
I0	I1	I2	I3	I4	I5	I6	I7	Y2	Y1	Y0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

For an 8-to-3 binary encoder with inputs I0-I7 the logic expressions of the outputs Y0-Y2 are:

Y0 = I1 + I3 + I5 + I7
Y1= I2 + I3 + I6 + I7
Y2 = I4 + I5 + I6+I7

Based on the above equations, we can draw the circuit as shownbelow

Circuit



Example - Decimal-to-BinaryEncoder

Decimal-to-Binary take 10 inputs and provides 4 outputs, thus doing the opposite of what the 4-to-10 decoder does. At any one time, only one input line has a value of 1. The figure below shows the truth table of a Decimal-to-binaryencoder.

TruthTable

I0	I1	I2	I3	I4	I5	I6	I7	I8	I9	Y3	Y2	Y1	Y0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0	0	0	0	1	1	0
0	0	0	0	0	0	0	1	0	0	0	1	1	1
0	0	0	0	0	0	0	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	1	0	0	1

From the above truth table , we can derive the functions Y3, Y2, Y1 and Y0 as given below.

Y3 = I8 +I9
Y2 = I4 + I5 + I6 + I7
Y1 = I2 + I3 + I6 +I7
Y0 = I1 + I3 + I5 + I7 +I9

PriorityEncoder

If we look carefully at the Encoder circuits that we got, we see the following limitations. If more then two inputs are active simultaneously, the output is unpredictable or rather it is not what we expect it tobe.

This ambiguity is resolved if priority is established so that only one input is encoded, no matter how many inputs are active at a given point oftime.

The priority encoder includes a priority function. The operation of the priority encoder is

such that if two or more inputs are active at the same time, the input having the highest priority will takeprecedence.

Example - 4to3 PriorityEncoder

The truth table of a 4-input priority encoder is as shown below. The input D3 has the highest priority, D2 has next highest priority, D0 has the lowest priority. This means output Y2 and Y1 are 0 only when none of the inputs D1, D2, D3 are high and only D0 ishigh.

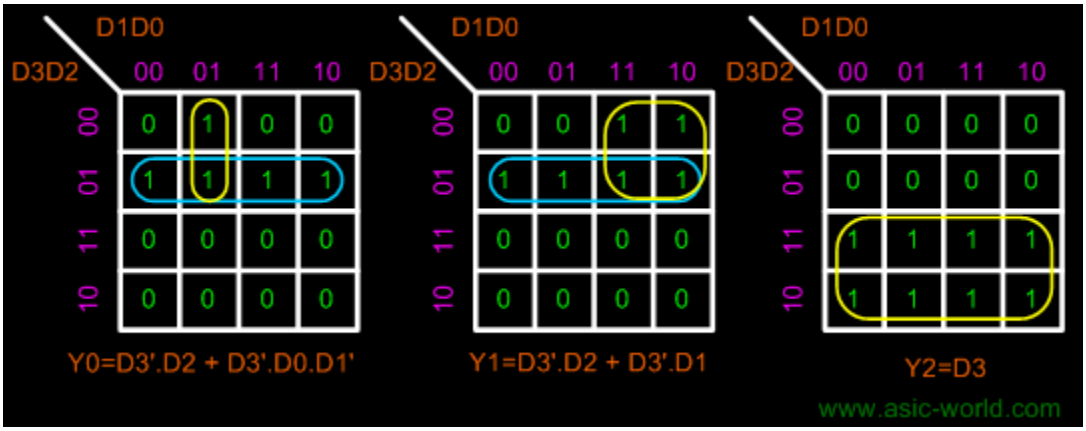
A 4 to 3 encoder consists of four inputs and three outputs, truth table and symbols of which is shownbelow.

TruthTable

D3	D2	D1	D0	Y2	Y1	Y0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	x	0	1	0
0	1	x	x	0	1	1
1	x	x	x	1	0	0

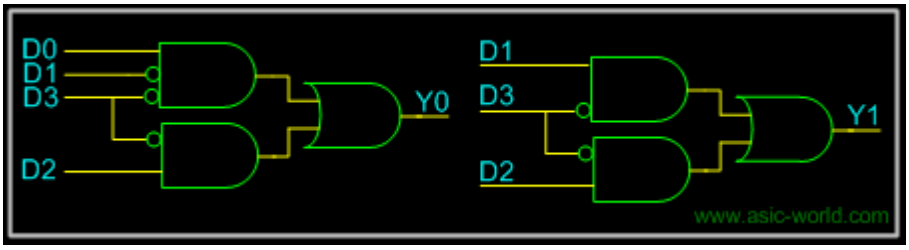
Now that we have the truth table, we can draw the Kmaps as shownbelow.

Kmaps



From the Kmap we can draw the circuit as shown below. For Y2, we connect directly to

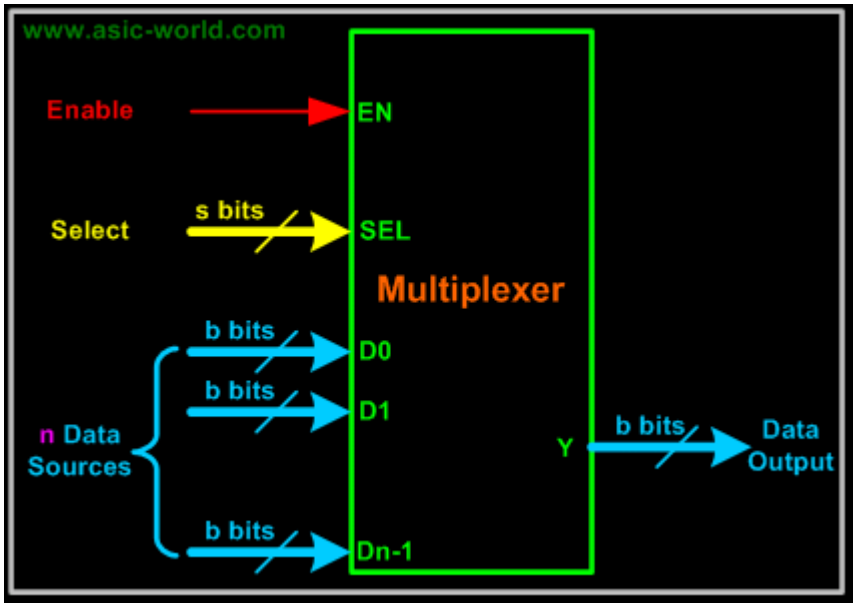
D3.



We can apply the same logic to get higher order priorityencoders.

Multiplexer

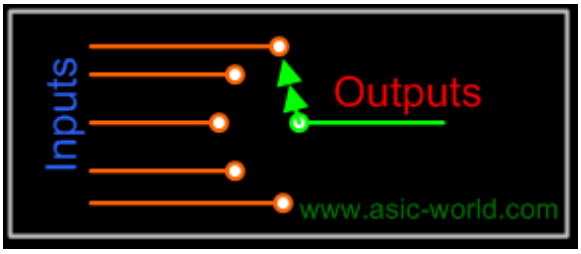
A multiplexer (MUX) is a digital switch which connects data from one of n sources to the output. A number of select inputs determine which data source is connected to the output. The block diagram of MUX with n data sources of b bits wide and s bits wide select line is shown in belowfigure.



MUX acts like a digitally controlled multi-position switch where the binary code applied to the select inputs controls the input source that will be switched on to the output as shown in the figure below. At any given point of time only one input gets selected and is connected to output, based on the select inputsignal.

Mechanical Equivalent of aMultiplexer

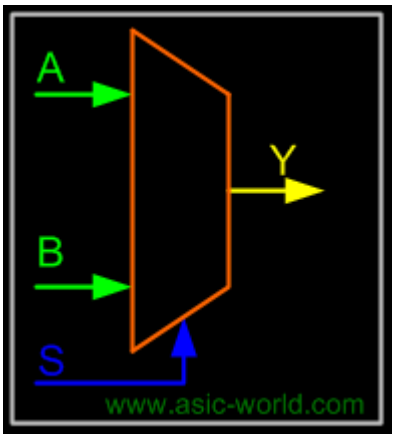
The operation of a multiplexer can be better explained using a mechanical switch as shown in the figure below. This rotary switch can touch any of the inputs, which is connected to the output. As you can see at any given point of time only one input gets transferred tooutput.



Example - 2x1MUX

A 2 to 1 line multiplexer is shown in figure below, each 2 input lines A to B is applied to one input of an AND gate. Selection lines S are decoded to select a particular AND gate. The truth table for the 2:1 mux is given in the tablebelow.

Symbol



TruthTable

S	Y
0	A
1	B

Design of a 2:1Mux

To derive the gate level implementation of 2:1 mux we need to have truth table as shown in figure. And once we have the truth table, we can draw the K-map as shown in figure for all the cases when Y is equal to'1'.

Combining the two 1' as shown in figure, we can drive the output y as shownbelow

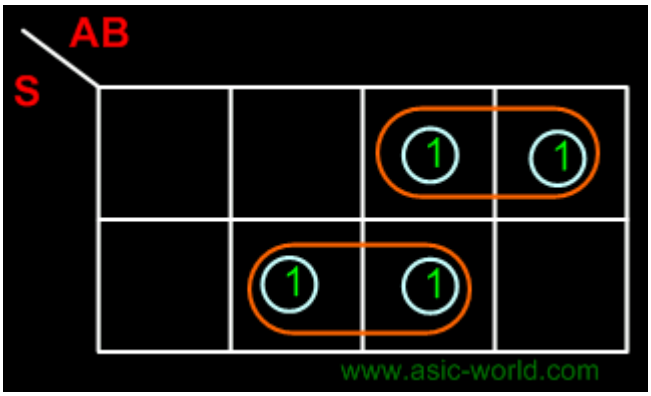
Y = A.S' +B.S

TruthTable

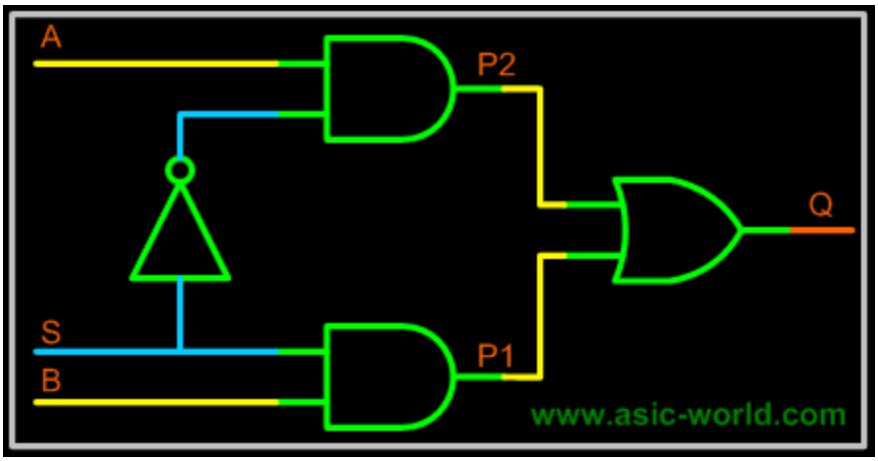
B	A	S	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0

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

Kmap



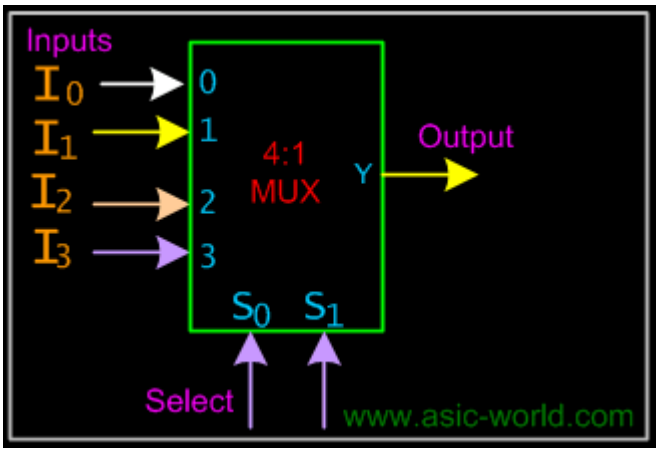
Circuit



Example : 4:1MUX

A 4 to 1 line multiplexer is shown in figure below, each of 4 input lines I0 to I3 is applied to one input of an AND gate. Selection lines S0 and S1 are decoded to select a particular AND gate. The truth table for the 4:1 mux is given in the tablebelow.

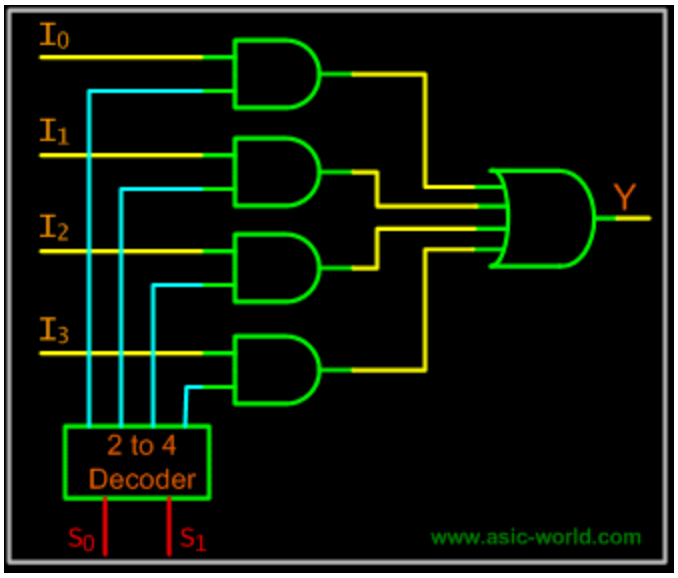
Symbol



TruthTable

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

Circuit



LargerMultiplexers

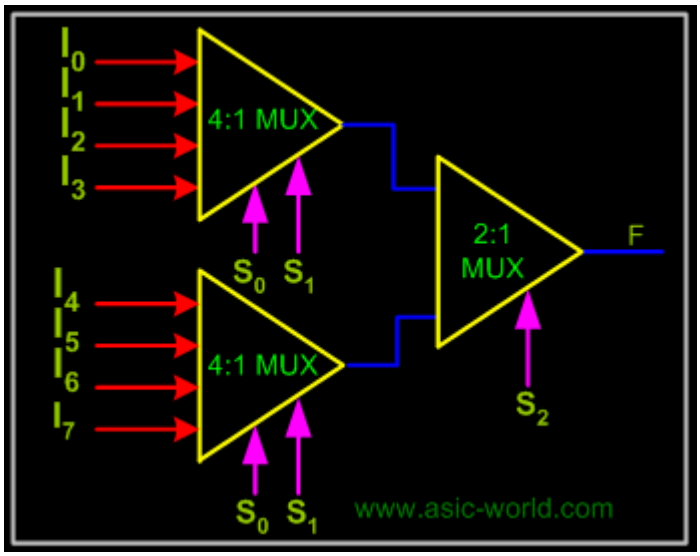
Larger multiplexers can be constructed from smaller ones. An 8-to-1 multiplexer can be constructed from smaller multiplexers as shownbelow.

Example - 8-to-1 multiplexer fromSmallerMUX

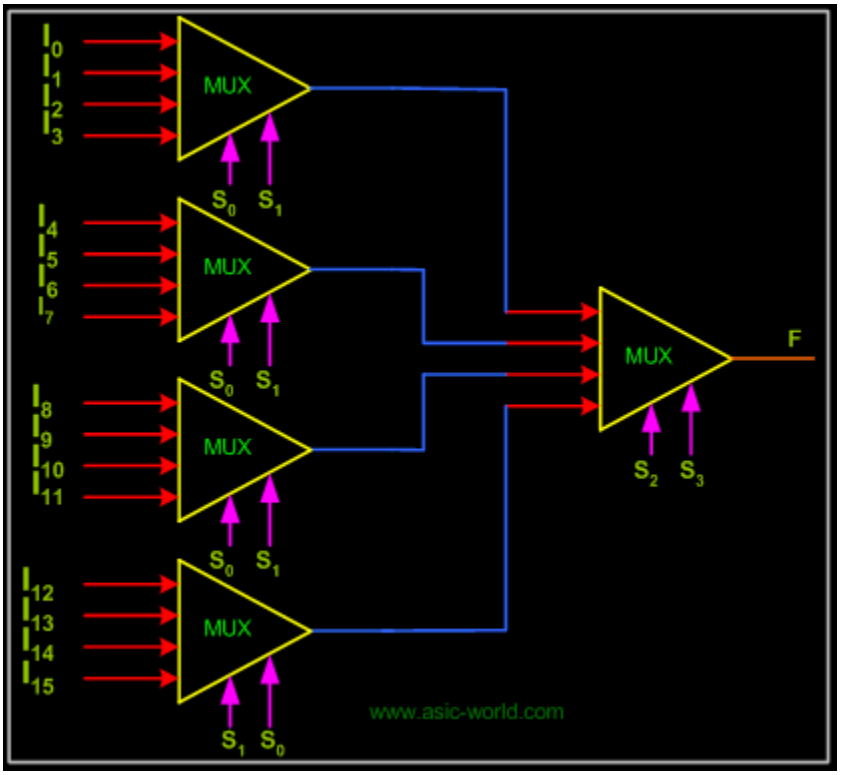
TruthTable

S2	S1	S0	F
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

Circuit



Example - 16-to-1 multiplexer from 4:1mux

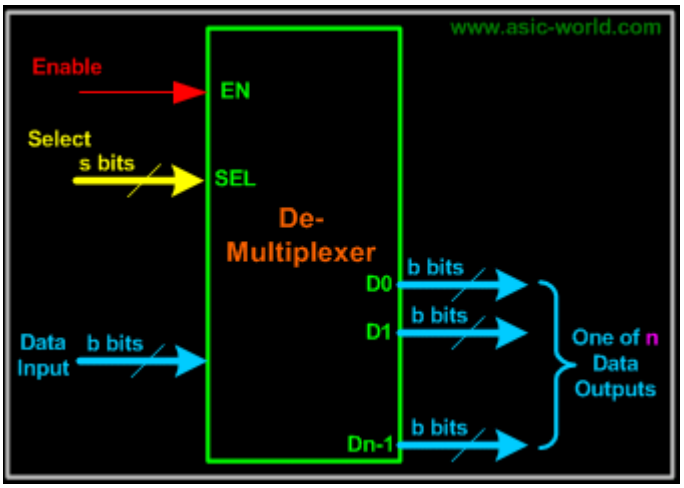


De-multiplexers

They are digital switches which connect data from one input source to one of noutputs.

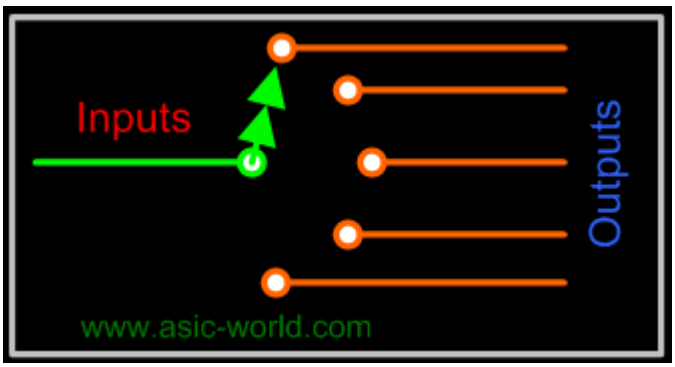
Usually implemented by using n-to-2^n binary decoders where the decoder enable line is used for data input of thede-multiplexer.

The figure below shows a de-multiplexer block diagram which has got s-bits-wide select input, one b-bits-wide data input and n b-bits-wideoutputs.

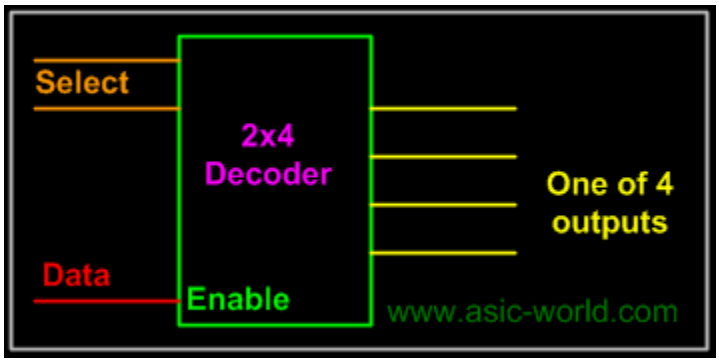


Mechanical Equivalent of aDe-Multiplexer

The operation of a de-multiplexer can be better explained using a mechanical switch as shown in the figure below. This rotary switch can touch any of the outputs, which is connected to the input. As you can see at any given point of time only one output gets connected toinput.

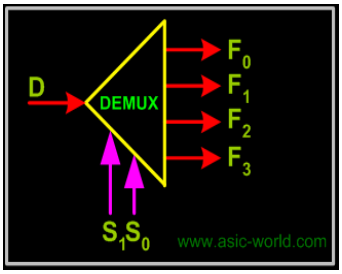


1-bit 4-output de-multiplexer using a 2x4 binarydecoder.



Example:1-to-4De-multiplexer

Symbol



TruthTable

S1	S0	F0	F1	F2	F3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

Boolean FunctionImplementation

Earlier we had seen that it is possible to implement Boolean functions using decoders. In the same way it is also possible to implement Boolean functions using muxers and de-muxers.

Implementing FunctionsMultiplexers

Any n-variable logic function can be implemented using a smaller 2^{n-1}-to-1 multiplexer

and a single inverter (e.g 4-to-1 mux to implement 3 variable functions) asfollows.

Express function in canonical sum-of-minterms form. Choose n-1 variables as inputs to mux select lines. Construct the truth table for the function, but grouping inputs by selection line values (i.e select lines as most significantinputs).

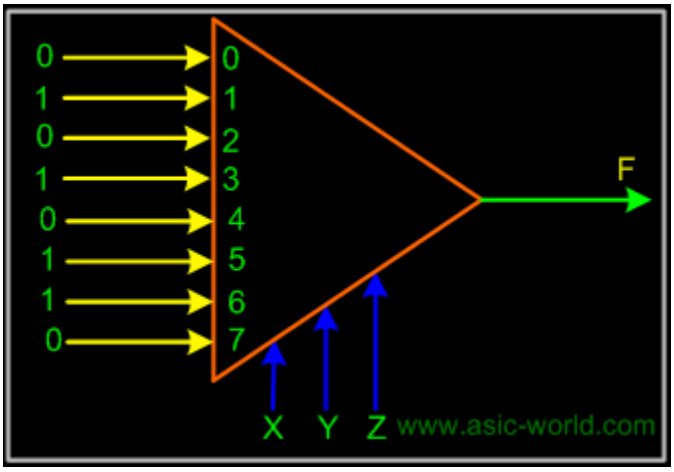
Determine multiplexer input line i values by comparing the remaining input variable and the function F for the corresponding selection lines valuei.

We have four possible mux input line ivalues:

- Connect to 0 if the function is 0 for both values of remainingvariable.
- Connect to 1 if the function is 1 for both values of remainingvariable.
- Connect to remaining variable if function is equal to the remainingvariable.
- Connect to the inverted remaining variable if the function is equal to the remaining variableinverted.

Example: 3-variable Function Using 8-to-1mux

Implement the function F(X,Y,Z) = S(1,3,5,6) using an 8-to-1 mux. Connect the input variables X, Y, Z to mux select lines. Mux data input lines 1, 3, 5, 6 that correspond to the function minterms are connected to 1. The remaining mux data input lines 0, 2, 4, 7 are connected to0.



Example: 3-variable Function Using 4-to-1mux

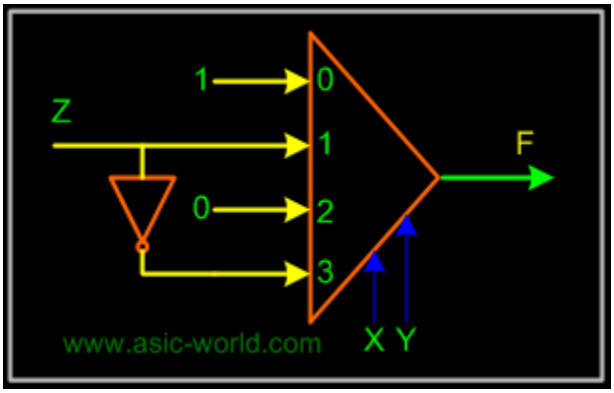
Implement the function F(X,Y,Z) = S(0,1,3,6) using a single 4-to-1 mux and an inverter. We choose the two most significant inputs X, Y as mux selectlines.

Construct truthtable.

TruthTable

Selecti	X	Y	Z	F	Mux Inputi
0	0	0	0	1	1
0	0	0	1	1	1
1	0	1	0	0	Z
1	0	1	1	1	Z
2	1	0	0	0	0
2	1	0	1	0	0
3	1	1	0	1	Z'
3	1	1	1	0	Z'

Circuit

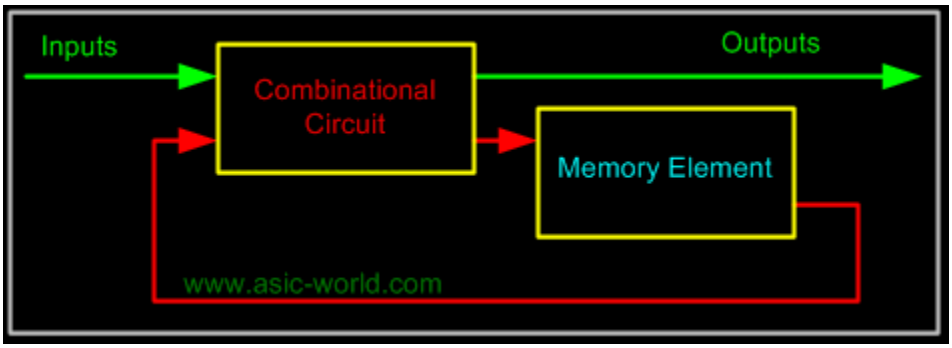


We determine multiplexer input line i values by comparing the remaining input variable Z and the function F for the corresponding selection lines valuei

- when XY=00 the function F is 1 (for both Z=0, Z=1) thus mux input0 =1
- when XY=01 the function F is Z thus mux input1 =Z
- when XY=10 the function F is 0 (for both Z=0, Z=1) thus mux input2 =0
- when XY=11 the function F is Z' thus mux input3 =Z

Sequential Circuits :Digital electronics is classified into combinational logic and sequential logic.

Combinational logic output depends on the inputs levels, whereas sequential logic output depends on stored levels and also the inputlevels.



The memory elements are devices capable of storing binary info. The binary info stored in the memory elements at any given time defines the state of the sequential circuit. The input and the present state of the memory element determines the output. Memory elements next state is also a function of external inputs and present state. A sequential circuit is specified by a time sequence of inputs, outputs, and internalstates.

There are two types of sequential circuits. Their classification depends on the timing of theirsignals:

- Synchronous sequentialcircuits
- Asynchronous sequentialcircuits

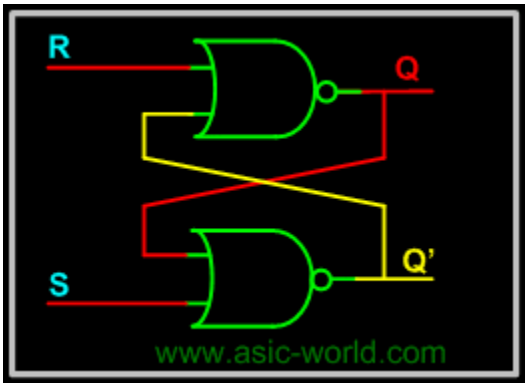
Latches andFlip-Flops

There are two types types of sequentialcircuits.

- AsynchronousCircuits.
- SynchronousCircuits.
- As seen in last section, Latches and Flip-flops are one and the same with a slight variation: Latches have level sensitive control signal input and Flip-flops have edge sensitive control signal input. Flip-flops and latches which use this control signals are called synchronous circuits. So if they don't use clock inputs, then they are called asynchronouscircuits.

RSLatch

RS latch have two inputs, S and R. S is called set and R is called reset. The S input is used to produce HIGH on Q (i.e. store binary 1 in flip-flop). The R input is used to produce LOW on Q (i.e. store binary 0 in flip-flop). Q' is Q complementary output, so it always holds the opposite value of Q. The output of the S-R latch depends on current as well as previous inputs or state, and its state (value stored) can change as soon as its inputs change. The circuit and the truth table of RS latch is shown below. (This circuit is as we saw in the last page, but arranged to look beautiful :-)).



S	R	Q	Q+
0	0	0	0
0	0	1	1
0	1	X	0
1	0	X	1

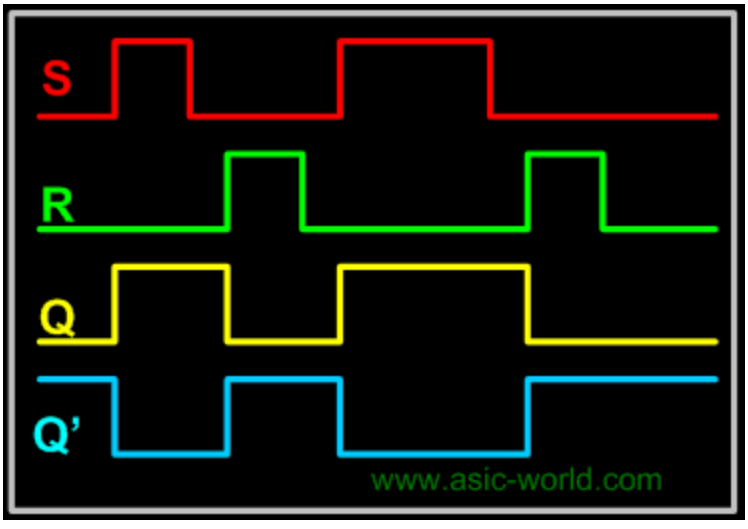
1	1	X	0

The operation has to be analyzed with the 4 inputs combinations together with the 2 possible previousstates.

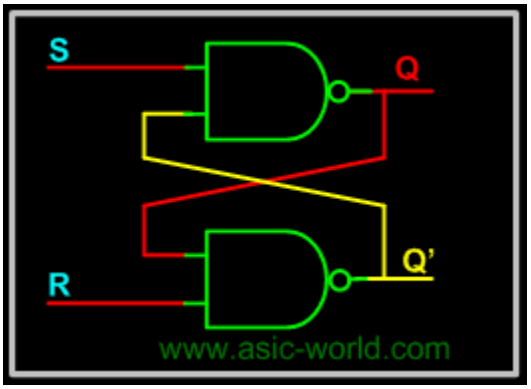
- **When S = 0 and R = 0:** If we assume Q = 1 and Q' = 0 as initial condition, then output Q after input is applied would be Q = (R + Q')' = 1 and Q' = (S + Q)' = 0. Assuming Q = 0 and Q' = 1 as initial condition, then output Q after the input applied would be Q = (R + Q')' = 0 and Q' = (S + Q)' = 1. So it is clear that when both S and R inputs are LOW, the output is retained as before the application of inputs. (i.e. there is no statechange).
- **When S = 1 and R = 0:** If we assume Q = 1 and Q' = 0 as initial condition, then output Q after input is applied would be Q = (R + Q')' = 1 and Q' = (S + Q)' = 0. Assuming Q = 0 and Q' = 1 as initial condition, then output Q after theinput

  applied would be Q = (R + Q')' = 1 and Q' = (S + Q)' = 0. So in simple words when S is HIGH and R is LOW, output Q isHIGH.
- **When S = 0 and R = 1:** If we assume Q = 1 and Q' = 0 as initial condition, then output Q after input is applied would be Q = (R + Q')' = 0 and Q' = (S + Q)' = 1. Assuming Q = 0 and Q' = 1 as initial condition, then output Q after the input applied would be Q = (R + Q')' = 0 and Q' = (S + Q)' = 1. So in simple words when S is LOW and R is HIGH, output Q isLOW.
- **When S = 1 and R =1 :** No matter what state Q and Q' are in, application of 1 at input of NOR gate always results in 0 at output of NOR gate, which results in both Q and Q' set to LOW (i.e. Q = Q'). LOW in both the outputs basically is wrong, so this case isinvalid.

The waveform below shows the operation of NOR gates based RSLatch.



It is possible to construct the RS latch using NAND gates (of course as seen in Logic gates section). The only difference is that NAND is NOR gate dual form (Did I say that in Logic gates section?). So in this case the R = 0 and S = 0 case becomes the invalid case. The circuit and Truth table of RS latch using NAND is shownbelow.

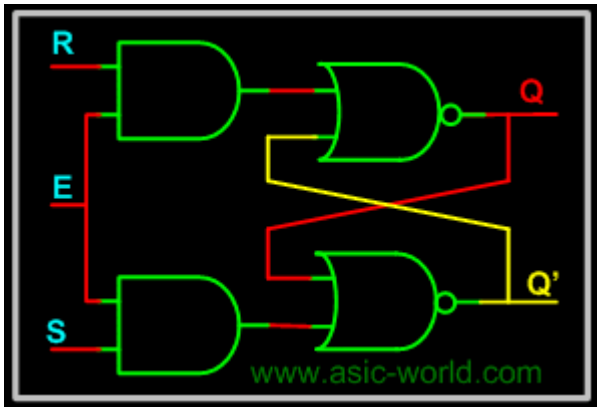


S	R	Q	Q+
1	1	0	0
1	1	1	1
0	1	X	0
1	0	X	1
0	0	X	1

If you look closely, there is no control signal (i.e. no clock and no enable), so this kind of latches or flip-flops are called asynchronous logic elements. Since all the sequential circuits are built around the RS latch, we will concentrate on synchronous circuits and not on asynchronouscircuits.

RS Latch withClock

We have seen this circuit earlier with two possible input configurations: one with level sensitive input and one with edge sensitive input. The circuit below shows the level sensitive RS latch. Control signal "Enable" E is used to gate the input S and R to the RS Latch. When Enable E is HIGH, both the AND gates act as buffers and thus R and S appears at the RS latch input and it functions like a normal RS latch. When Enable E is LOW, it drives LOW to both inputs of RS latch. As we saw in previous page, when both inputs of a NOR latch are low, values are retained (i.e. the output does notchange).



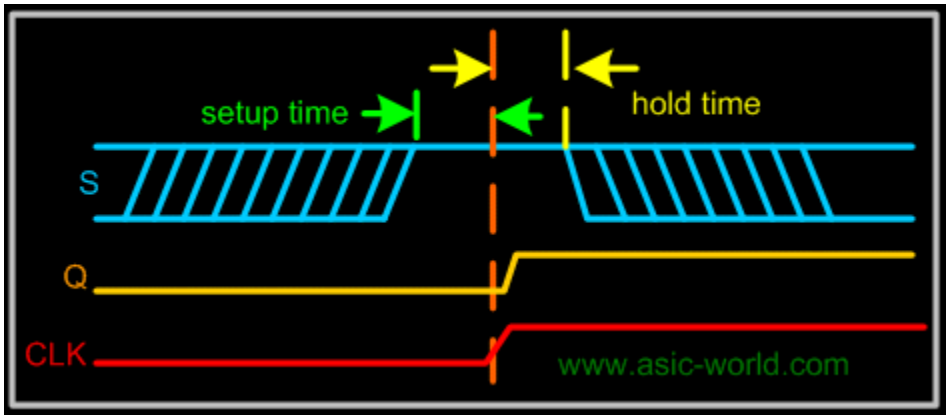
Setup and HoldTime

For synchronous flip-flops, we have special requirements for the inputs with respect to clock signal input. They are**Setup Time:** Minimum time period during which data must be stable before the clock makes a valid transition. For example, for a posedge triggered flip-flop, with a setup time of 2 ns, Input Data (i.e. R and S in the case of RS flip-flop) should be stable for at least 2 ns before clock makes transition from 0 to1.

- **Hold Time:** Minimum time period during which data must be stable after the clock has made a valid transition. For example, for a posedge triggered flip-flop, with a hold time of 1 ns. Input Data (i.e. R and S in the case of RS flip-flop) should be stable for at least 1 ns after clock has made transition from 0 to1.
- If data makes transition within this setup window and before the hold window, then the flip-flop output is not predictable, and flip-flop enters what is known as **meta stable state**. In this state flip-flop output oscillates between 0 and 1. It takes some time for the flip-flop to settle down. The

whole process is called **metastability**. You could refer to tidbits section to know more information on this topic.

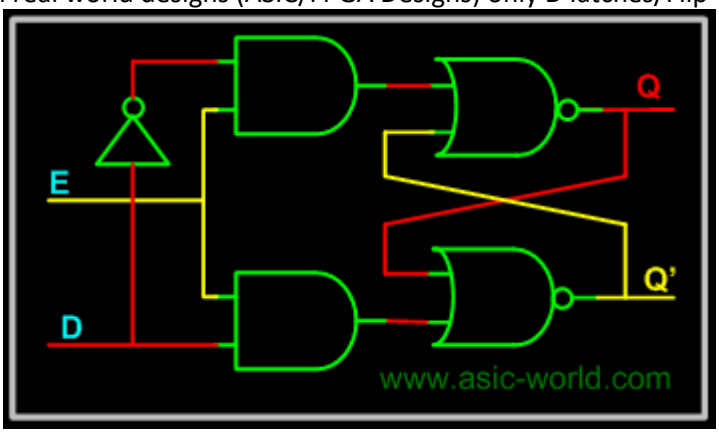
The waveform below shows input S (R is not shown), and CLK and output Q (Q' is not shown) for a SR posedgeflip-flop.



DLatch

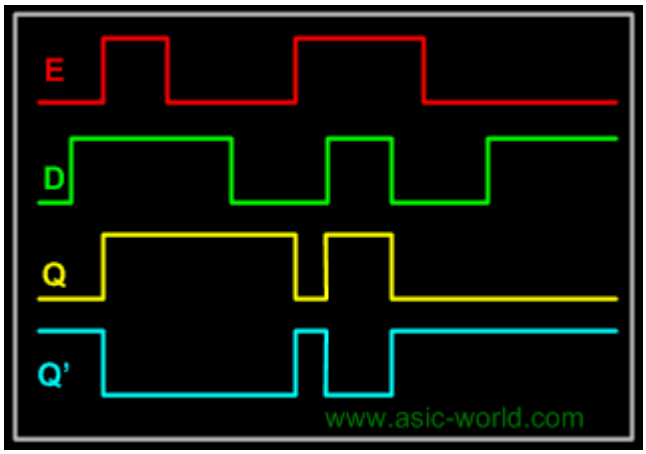
The RS latch seen earlier contains ambiguous state; to eliminate this condition we can ensure that S and R are never equal. This is done by connecting S and R together with an inverter. Thus we have D Latch: the same as the RS latch, with the only difference that there is only one input, instead of two (R and S). This input is called D or Data input. D latch is called D transparent latch for the reasons explained earlier. Delay flip- flop or delay latch is another name used. Below is the truth table and circuit of Dlatch.

In real world designs (ASIC/FPGA Designs) only D latches/Flip-Flops areused.



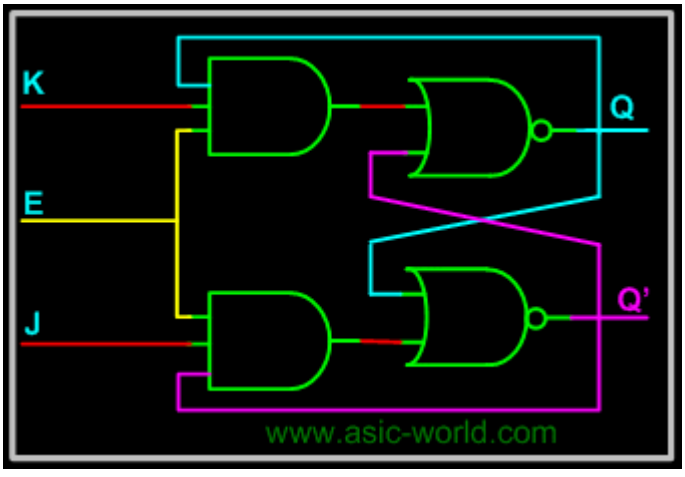
D	Q	Q+
1	X	1
0	X	0

Below is the D latch waveform, which is similar to the RS latch one, but with Rremoved.



JKLatch

The ambiguous state output in the RS latch was eliminated in the D latch by joining the inputs with an inverter. But the D latch has a single input. JK latch is similar to RS latch in that it has 2 inputs J and K as shown figure below. The ambiguous state has been eliminated here: when both inputs are high, output toggles. The only difference we see here is output feedback to inputs, which is not there in the RSlatch.

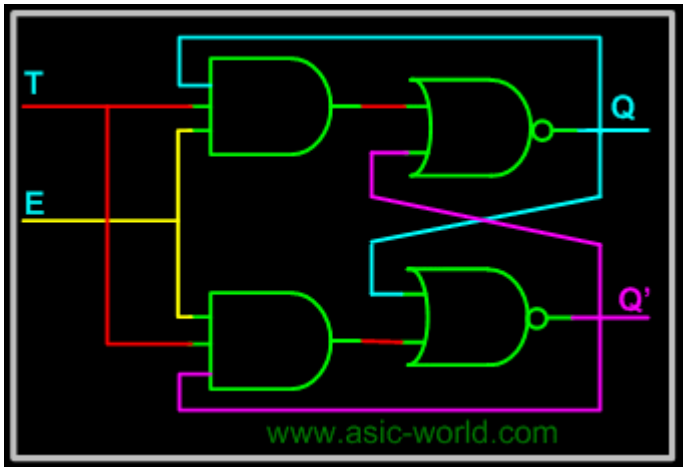


J	K	Q
1	1	0

1	1	1
1	0	1
0	1	0

TLatch

When the two inputs of JK latch are shorted, a T Latch is formed. It is called T latch as, when input is held HIGH, outputtoggles.

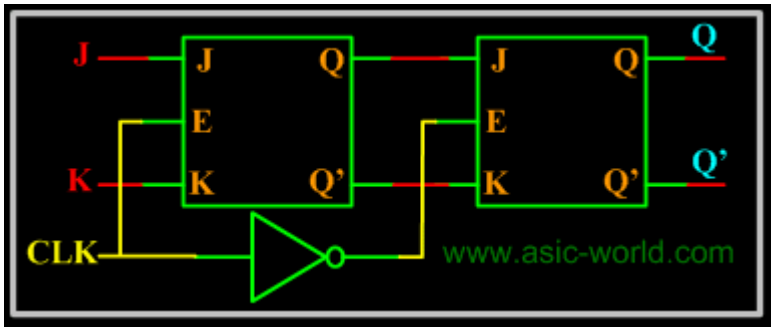


T	Q	Q+
1	0	1
1	1	0
0	1	1
0	0	0

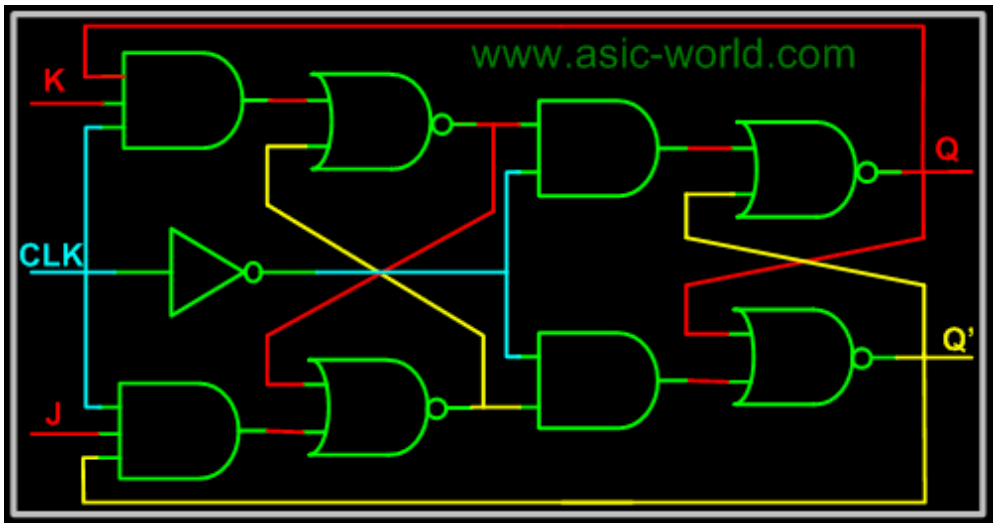
JK Master SlaveFlip-Flop

All sequential circuits that we have seen in the last few pages have a problem (All level sensitive sequential circuits have this problem). Before the enable input changes state from HIGH to LOW (assuming HIGH is ON and LOW is OFF state), if inputs changes, then another state transition occurs for the same enable pulse. This sort of multiple transition problem is calledracing.

If we make the sequential element sensitive to edges, instead of levels, we can overcome this problem, as input is evaluated only during enable/clockedges.



In the figure above there are two latches, the first latch on the left is called master latch and the one on the right is called slave latch. Master latch is positively clocked and slave latch is negativelyclocked.



Sequential CircuitsDesign

We saw in the combinational circuits section how to design a combinational circuit from the given problem. We convert the problem into a truth table, then draw K-map for the truth table, and then finally draw the gate level circuit for the problem. Similarly we have a flow for the sequential circuit design. The steps are givenbelow.

- Draw statediagram.
- Draw the state table (excitation table) for eachoutput.
- Draw the K-map for eachoutput.
- Draw thecircuit.

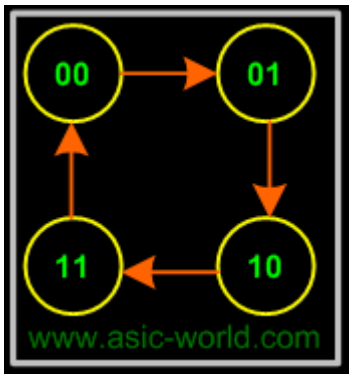
Looks like sequential circuit design flow is very much the same as for combinationalcircuit.

StateDiagram

The state diagram is constructed using all the states of the sequential circuit in question. It builds up the relationship between various states and also shows how inputs affect thestates.

To ease the following of the tutorial, let's consider designing the 2 bit up counter (Binary counter is one which counts a binary sequence) using the Tflip-flop.

Below is the state diagram of the 2-bit binarycounter.



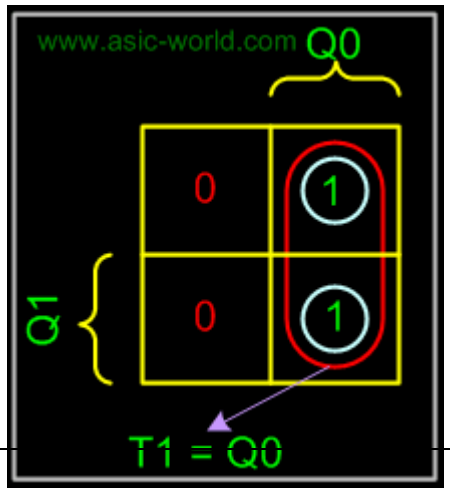
StateTable

Thestatetableist hesameastheexcitationtableofaflip-flop,i.e.whatinputsneedtobe applied to get the required output. In other words this table gives the inputs required to produce the specificoutputs.

Q1	Q0	Q1+	Q0+	T1	T0
0	0	0	1	0	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1

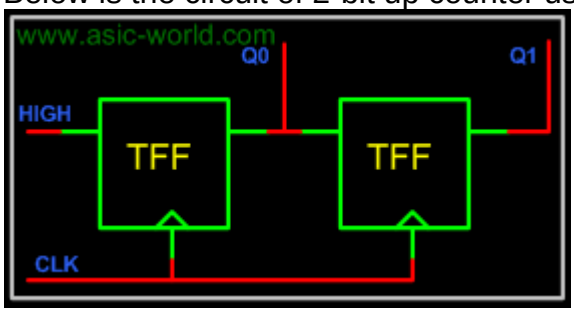
K-map

The K-map is the same as the combinational circuits K-map. Only difference: we draw K-map for the inputs i.e. T1 and T0 in the above table. From the table we deduct that we don't need to draw K-map for T0, as it is high for all the state combinations. But for T1 we need to draw the K- map as shown below, usingSOP.



Circuit

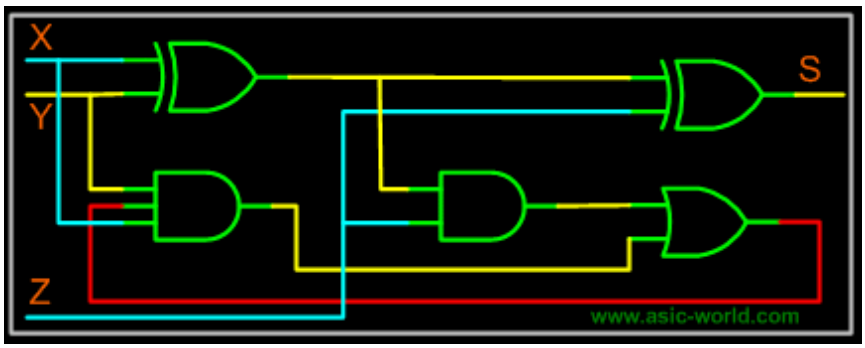
There is nothing special in drawing the circuit, it is the same as any circuit drawing from K-map output. Below is the circuit of 2-bit up counter using the Tflip-flop.



Asynchronous sequentialcircuit

This is a system whose outputs depend upon the order in which its input variables change and can be affected at any instant oftime.

Gate-type asynchronous systems are basically combinational circuits with feedback paths. Because of the feedback among logic gates, the system may, at times, become unstable. Consequently they are not oftenused.



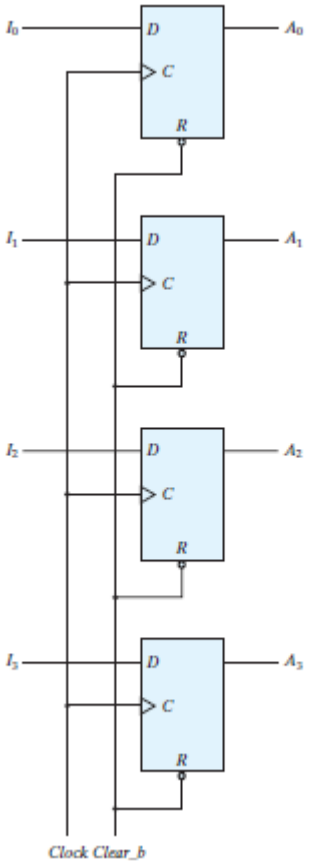
REGISTERS

A clocked sequential circuit consists of a group of flip-flops and combinational gates. The flip-flops are essential because, in their absence, the circuit reduces to a purely combinational circuit (provided that there is no feedback among the gates). A circuit with flip-flops is considered a sequential circuit even in the absence of combinational gates. Circuits that include flip-flops are usually classified by the function they perform rather than by the name of the sequential circuit. Two such circuits are registers and counters.

A *register* is a group of flip-flops, each one of which shares a common clock and is capable of storing one bit of information. An n-bit register consists of a group of n flip-flops capable of storing n bits of binary information. In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks. In its broadest definition, a register consists of a group of flip-flops together with gates that affect their operation. The flip-flops hold the binary information, and the gates determine how the information is transferred into the register.

A *counter* is essentially a register that goes through a predetermined sequence of binary states. The gates in the counter are connected in such a way as to produce the prescribed sequence of states. Although counters are a special type of register, it is common to differentiate them by giving them a different name.

Various types of registers are available commercially. The simplest register is one that consists of only flip-flops, without any gates. Figure 6.1 shows such a register constructed with four D-type flip-flops to form a four-bit data storage register. The common clock input triggers all flip-flops on the positive edge of each pulse, and the binary data available at the four inputs are transferred into the register. The value of (I_3, I_2, I_1, I_0) immediately before the clock edge determines the value of (A_3, A_2, A_1, A_0) after the clock edge. The four

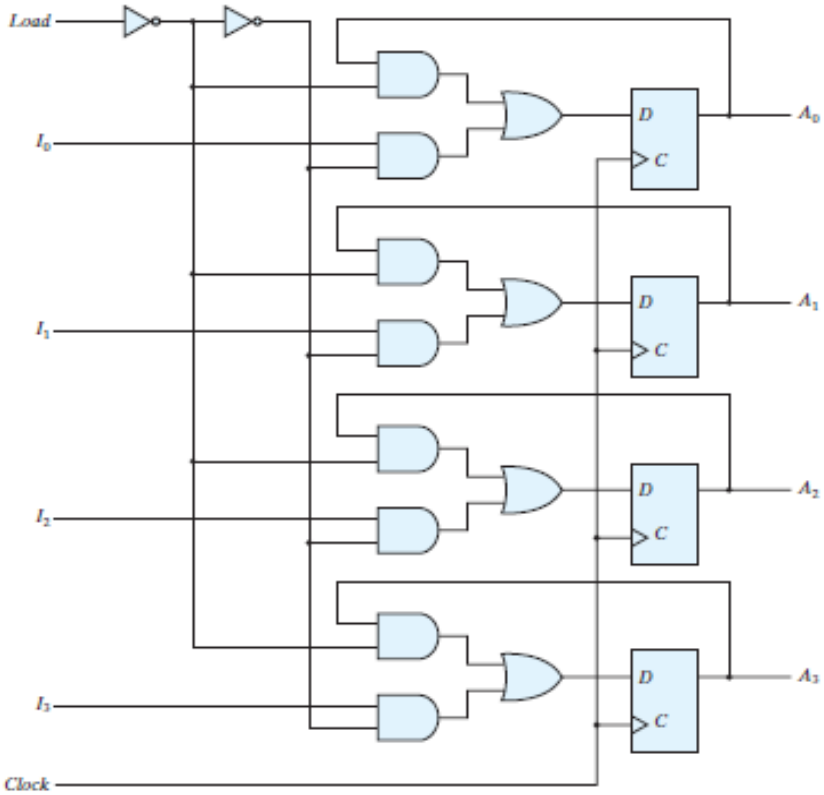


outputs can be sampled at any time to obtain the binary information stored in the register. The input *Clear_b* goes to the active-low *R* (reset) input of all four flip-flops. When this input goes to 0, all flip-flops are reset asynchronously. The *Clear_b* input is useful for clearing the register to all 0's prior to its clocked operation. The *R* inputs must be maintained

Register with Parallel Load

Registers with parallel load are a fundamental building block in digital systems. It is important that you have a thorough understanding of their behavior. Synchronous digital systems have a master clock generator that supplies a continuous train of clock pulses. The pulses are applied to all flip-flops and registers in the system. The master clock acts like a drum that supplies a constant beat to all parts of the system. A separate control signal must be used to decide which register operation will execute at each clock pulse. The transfer of new information into a register is referred to as *loading* or *updating* the register. If all the bits of the register are loaded simultaneously with a common clock pulse, we say that the loading is done *in parallel*. A clock edge applied to the *C* inputs of the register of Fig. 6.1 will load all four inputs in parallel. In this configuration, if the contents of the register must be left unchanged, the inputs must be held constant or the clock must be inhibited from the circuit. In the first case, the data bus driving the register would be unavailable for other traffic. In the second case, the clock can be inhibited from reaching the register by controlling the clock input signal with an enabling gate. However, inserting gates into the clock path is ill advised because it means that logic is performed with clock pulses. The insertion of logic gates produces uneven propagation delays between the master clock and the inputs of flip-flops. To fully synchronize the system, we must ensure that all clock pulses arrive at the same time anywhere in the system, so that all flip-flops trigger simultaneously. Performing logic with clock pulses inserts variable delays and may cause the system to go out of synchronism. For this reason, it is advisable to control the operation of the register with the *D* inputs, rather than controlling the clock in the *C* inputs of the flip-flops. This creates the effect of a gated clock, but without affecting the clock path of the circuit.

A four-bit data-storage register with a load control input that is directed through gates and into the *D* inputs of the flip-flops is shown in Fig. 6.2. The additional gates implement a two-channel mux whose output drives the input to the register with either the data bus or the output of the register. The load input to the register determines the action to be taken with each clock pulse. When the load input is 1, the data at the four external inputs are transferred into the register with the next positive edge of the clock. When the load input is 0, the outputs of the flip-flops are connected to their respective inputs. The feedback connection from output to input is necessary because a *D* flip-flop does not have a "no change" condition. With each clock edge, the *D* input determines the next state of the register. To leave the output unchanged, it is necessary to make the *D* input equal to the present value of the output (i.e., the output circulates to the input at each clock pulse). The clock pulses are applied to the *C* inputs without interruption. The load input determines whether the next pulse will accept new information or leave the information in the register intact. The transfer of information from the data inputs or the outputs of the register is done simultaneously with all four bits in response to a clock edge.



SHIFT REGISTERS

A register capable of shifting the binary information held in each cell to its neighboring cell, in a selected direction, is called a *shift register*. The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops receive common clock pulses, which activate the shift of data from one stage to the next.

The simplest possible shift register is one that uses only flip-flops, as shown in Fig. 6.3. The output of a given flip-flop is connected to the *D* input of the flip-flop at its right. This shift register is unidirectional (left-to-right). Each clock pulse shifts the contents of the

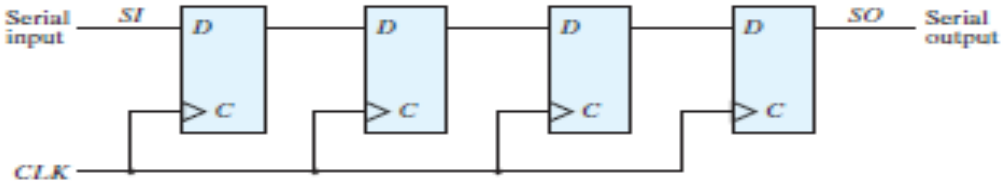


FIGURE 6.3
Four-bit shift register

register one bit position to the right. The configuration does not support a left shift. The *serial input* determines what goes into the leftmost flip-flop during the shift. The *serial output* is taken from the output of the rightmost flip-flop. Sometimes it is necessary to control the shift so that it occurs only with certain pulses, but not with others. As with the data register discussed in the previous section, the clock's signal can be suppressed by gating the clock signal to prevent the register from shifting. A preferred alternative in high-speed circuits is to suppress the clock *action*, rather than gate the clock signal, by leaving the clock path unchanged, but recirculating the output of each register cell back through a two-channel mux whose output is connected to the input of the cell. When the clock action is not suppressed, the other channel of the mux provides a datapath to the cell.

It will be shown later that the shift operation can be controlled through the D inputs of the flip-flops rather than through the clock input. If, however, the shift register of Fig. 6.3 is used, the shift can be controlled with an input by connecting the clock through an AND gate. This is not a preferred practice. Note that the simplified schematics do not show a reset signal, but such a signal is required in practical designs.

Serial Transfer

The datapath of a digital system is said to operate in serial mode when information is transferred and manipulated one bit at a time. Information is transferred one bit at a time by shifting the bits out of the source register and into the destination register. This type of transfer is in contrast to parallel transfer, whereby all the bits of the register are transferred at the same time.

The serial transfer of information from register A to register B is done with shift registers, as shown in the block diagram of Fig. 6.4(a). The serial output (SO) of register A is connected to the serial input (SI) of register B. To prevent the loss of information stored in the source register, the information in register A is made to circulate by connecting the serial output to its serial input. The initial content of register B is shifted out through its serial output and is lost unless it is transferred to a third shift register. The shift control input determines when and how many times the registers are shifted. For illustration here, this is done with an AND gate that allows clock pulses to pass into the CLK terminals only when the shift control is active. (This practice can be problematic because it may compromise the clock path of the circuit, as discussed earlier.)

Suppose the shift registers in Fig. 6.4 have four bits each. Then the control unit that supervises the transfer of data must be designed in such a way that it enables the shift

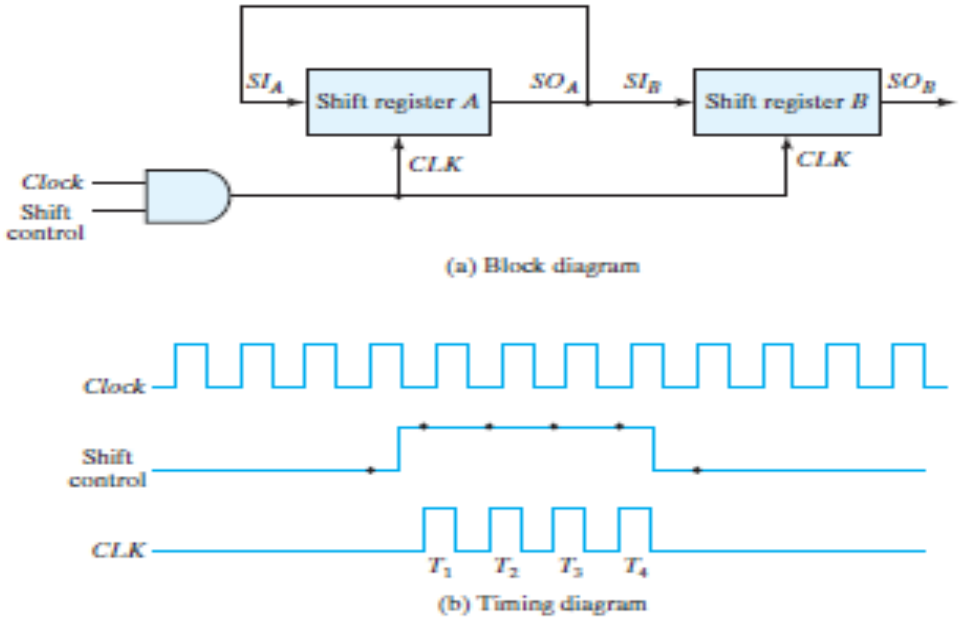


FIGURE 6.4
Serial transfer from register *A* to register *B*

registers, through the shift control signal, for a fixed time of four clock pulses in order to pass an entire word. This design is shown in the timing diagram of Fig. 6.4(b). The shift control signal is synchronized with the clock and changes value just after the negative edge of the clock. The next four clock pulses find the shift control signal in the active state, so the output of the AND gate connected to the CLK inputs produces four pulses: T_1, T_2, T_3, and T_4. Each rising edge of the pulse causes a shift in both registers. The fourth pulse changes the shift control to 0, and the shift registers are disabled.

Assume that the binary content of A before the shift is 1011 and that of B is 0010. The serial transfer from A to B occurs in four steps, as shown in Table 6.1. With the first pulse, T_1, the rightmost bit of A is shifted into the leftmost bit of B and is also circulated into the leftmost position of A. At the same time, all bits of A and B are shifted one position to the right. The previous serial output from B in the rightmost position is lost, and its value changes from 0 to 1. The next three pulses perform identical operations, shifting the bits of A into B, one at a time. After the fourth shift, the shift control goes to 0, and registers A and B both have the value 1011. Thus, the contents of A are copied into B, so that the contents of A remain unchanged i.e., the contents of A are restored to their original value.

The difference between the serial and the parallel mode of operation should be apparent from this example. In the parallel mode, information is available from all bits of a register and all bits can be transferred simultaneously during one clock pulse. In the serial

Serial-Transfer Example

Timing Pulse	Shift Register A				Shift Register B			
Initial value	1	0	1	1	0	0	1	0
After T_1	1	1	0	1	1	0	0	1
After T_2	1	1	1	0	1	1	0	0
After T_3	0	1	1	1	0	1	1	0
After T_4	1	0	1	1	1	0	1	1

mode, the registers have a single serial input and a single serial output. The information is transferred one bit at a time while the registers are shifted in the same direction.

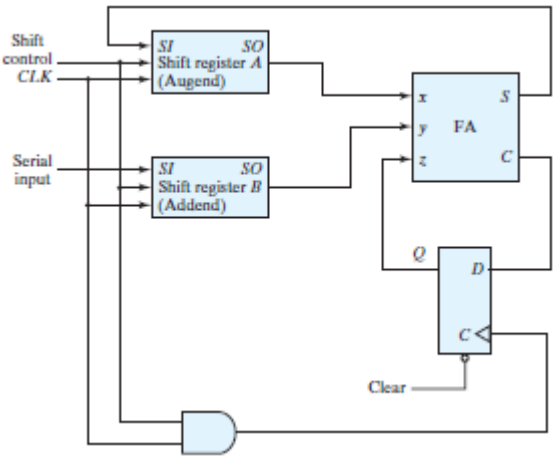
Serial Addition

Operations in digital computers are usually done in parallel because that is a faster mode of operation. Serial operations are slower because a datapath operation takes several clock cycles, but serial operations have the advantage of requiring fewer hardware components. In VLSI circuits, they require less silicon area on a chip. To demonstrate the serial mode of operation, we present the design of a serial adder. The parallel counterpart was presented in Section 4.4.

The two binary numbers to be added serially are stored in two shift registers. Beginning with the least significant pair of bits, the circuit adds one pair at a time through a single full-adder (FA) circuit, as shown in Fig. 6.5. The carry out of the full adder is transferred to a D flip-flop, the output of which is then used as the carry input for the next pair of significant bits. The sum bit from the S output of the full adder could be transferred into a third shift register. By shifting the sum into A while the bits of A are shifted out, it is possible to use one register for storing both the augend and the sum bits. The serial input of register B can be used to transfer a new binary number while the addend bits are shifted out during the addition.

The operation of the serial adder is as follows: Initially, register A holds the augend, register B holds the addend, and the carry flip-flop is cleared to 0. The outputs (SO) of A and B provide a pair of significant bits for the full adder at x and y. Output Q of the flip-flop provides the input carry at z. The shift control enables both registers and the carry flip-flop, so at the next clock pulse, both registers are shifted once to the right, the sum bit from S enters the leftmost flip-flop of A, and the output carry is transferred into flip-flop Q. The shift control enables the registers for a number of clock pulses equal to the number of bits in the registers. For each succeeding clock pulse, a new sum bit is transferred to A, a new carry is transferred to Q, and both registers are shifted once to the right. This process continues until the shift control is disabled. Thus, the addition is accomplished by passing each pair of bits together with the previous carry through a single full-adder circuit and transferring the sum, one bit at a time, into register A.

Initially, register A and the carry flip-flop are cleared to 0, and then the first number is added from B. While B is shifted through the full adder, a second number is transferred



to it through its serial input. The second number is then added to the contents of register A, while a third number is transferred serially into register B. This can be repeated to perform the addition of two, three, or more four-bit numbers and accumulate their sum in register A.

Comparing the serial adder with the parallel adder described in Section 4.4, we note several differences. The parallel adder uses registers with a parallel load, whereas the serial adder uses shift registers. The number of full-adder circuits in the parallel adder is equal to the number of bits in the binary numbers, whereas the serial adder requires only one full-adder circuit and a carry flip-flop. Excluding the registers, the parallel adder is a combinational circuit, whereas the serial adder is a sequential circuit which consists of a full adder and a flip-flop that stores the output carry. This design is typical in serial operations because the result of a bit-time operation may depend not only on the present inputs, but also on previous inputs that must be stored in flip-flops.

To show that serial operations can be designed by means of sequential circuit procedure, we will redesign the serial adder with the use of a state table. First, we assume that two shift registers are available to store the binary numbers to be added serially. The serial outputs from the registers are designated by x and y. The sequential circuit to be designed will not include the shift registers, but they will be inserted later to show the complete circuit. The sequential circuit proper has the two inputs, x and y, that provide a pair of significant bits, an output S that generates the sum bit, and flip-flop Q for storing the carry. The state table that specifies the sequential circuit is listed in Table 6.2. The present state of Q is the present value of the carry. The present carry in

State Table for Serial Adder

Present State	Inputs		Next State	Output	Flip-Flop Inputs	
Q	x	y	Q	S	J_Q	K_Q
0	0	0	0	0	0	X
0	0	1	0	1	0	X
0	1	0	0	1	0	X
0	1	1	1	0	1	X
1	0	0	0	1	X	1
1	0	1	1	0	X	0
1	1	0	1	0	X	0
1	1	1	1	1	X	0

Q is added together with inputs x and y to produce the sum bit in output S. The next state of Q is equal to the output carry. Note that the state table entries are identical to the entries in a full-adder truth table, except that the input carry is now the present state of Q and the output carry is now the next state of Q.

If a D flip-flop is used for Q, the circuit reduces to the one shown in Fig. 6.5. If a JK flip-flop is used for Q, it is necessary to determine the values of inputs J and K by referring to the excitation table (Table 5.12). This is done in the last two columns of Table 6.2. The two flip-flop input equations and the output equation can be simplified by means of maps to

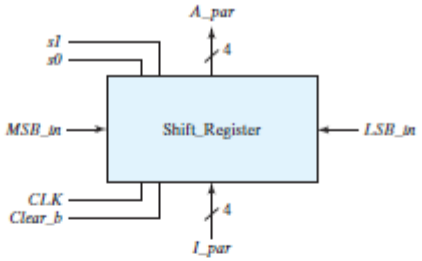
$$J_Q = xy$$
$$K_Q = x'y' = (x + y)'$$
$$S = x \oplus y \oplus Q$$

The circuit diagram is shown in Fig. 6.6. The circuit consists of three gates and a JK flip-flop. The two shift registers are included in the diagram to show the complete serial adder. Note that output S is a function not only of x and y, but also of the present state of Q. The next state of Q is a function of the present state of Q and of the values of x and y that come out of the serial outputs of the shift registers.

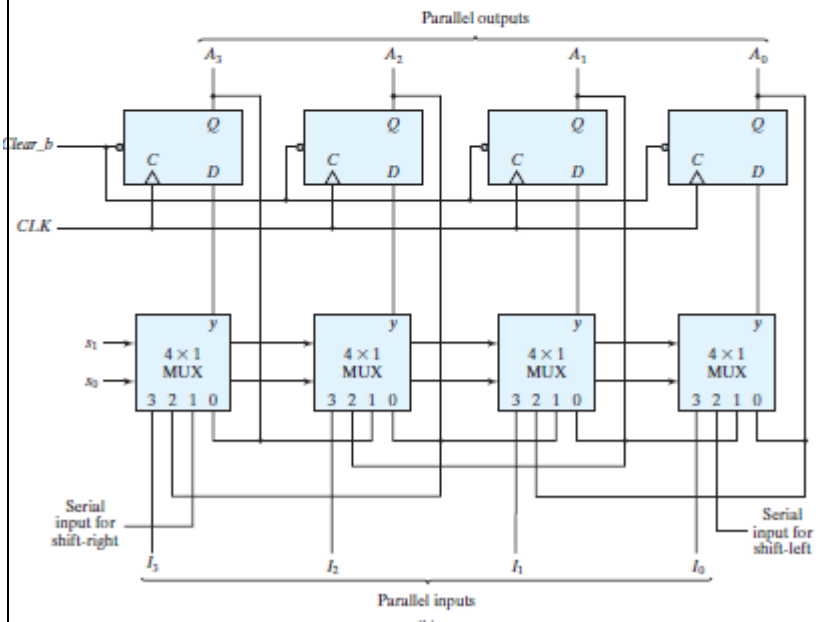
3. A *shift-right* control to enable the shift-right operation and the *serial input* and *output* lines associated with the shift right.
4. A *shift-left* control to enable the shift-left operation and the *serial input* and *output* lines associated with the shift left.
5. A *parallel-load* control to enable a parallel transfer and the n input lines associated with the parallel transfer.
6. n parallel output lines.
7. A control state that leaves the information in the register unchanged in response to the clock. Other shift registers may have only some of the preceding functions, with at least one shift operation.

A register capable of shifting in one direction only is a *unidirectional* shift register. One that can shift in both directions is a *bidirectional* shift register. If the register has both shifts and parallel-load capabilities, it is referred to as a *universal shift register.*

The block diagram symbol and the circuit diagram of a four-bit universal shift register that has all the capabilities just listed are shown in Fig. 6.7. The circuit consists of four D flip-flops and four multiplexers. The four multiplexers have two common selection inputs s_1 and s_0. Input 0 in each multiplexer is selected when $s_1s_0 = 00$, input 1 is selected when $s_1s_0 = 01$, and similarly for the other two inputs. The selection inputs control the mode of operation of the register according to the function entries in Table 6.3. When $s_1s_0 = 00$, the present value of the register is applied to the D inputs of the flip-flops. This condition forms a path from the output of each flip-flop into the input of the same flip-flop, so that the output recirculates to the input in this mode of operation. The next clock edge transfers into each flip-flop the binary value it held previously, and no change of state occurs.


A_par
s1
s0
4
MSB_in
Shift_Register
LSB_in
CLK
Clear_b
4
I_par

(a)


Parallel outputs
Clear_b
CLK
Q
C
D
y
4 × 1
MUX
3 2 1 0
Serial
input for
shift-right
Serial
input for
shift-left
Parallel inputs

RIPPLE COUNTERS

A register that goes through a prescribed sequence of states upon the application of input pulses is called a *counter*. The input pulses may be clock pulses, or they may originate from some external source and may occur at a fixed interval of time or at random. The sequence of states may follow the binary number sequence or any other sequence of states. A counter that follows the binary number sequence is called a *binary counter*. An n-bit binary counter consists of n flip-flops and can count in binary from 0 through $2^n - 1$.

Counters are available in two categories: ripple counters and synchronous counters. In a ripple counter, a flip-flop output transition serves as a source for triggering other flip-flops. In other words, the C input of some or all flip-flops are triggered, not by the common clock pulses, but rather by the transition that occurs in other flip-flop outputs. In a synchronous counter, the C inputs of all flip-flops receive the common clock. Synchronous counters are presented in the next two sections. Here, we present the binary and BCD ripple counters and explain their operation.

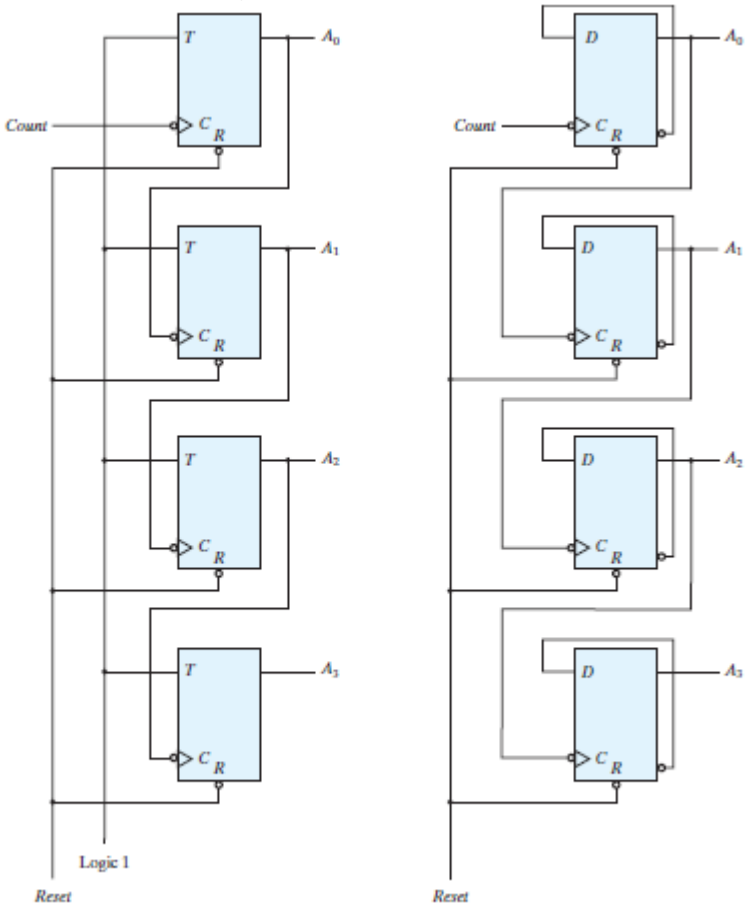
Binary Ripple Counter

A binary ripple counter consists of a series connection of complementing flip-flops, with the output of each flip-flop connected to the C input of the next higher order flip-flop. The flip-flop holding the least significant bit receives the incoming count pulses. A complementing flip-flop can be obtained from a JK flip-flop with the J and K inputs tied together or from a T flip-flop. A third possibility is to use a D flip-flop with the complement output connected to the D input. In this way, the D input is always the complement of the present state, and the next clock pulse will cause the flip-flop to complement. The logic diagram of two 4-bit binary ripple counters is shown in Fig. 6.8. The counter is constructed with complementing flip-flops of the T type in part (a) and D type in part (b). The output of each flip-flop is connected to the C input of the next flip-flop in sequence. The flip-flop holding the least significant bit receives the incoming count pulses. The T inputs of all the flip-flops in (a) are connected to a permanent logic 1, making each flip-flop complement if the signal in its C input goes through a negative transition. The bubble in front of the dynamic indicator symbol next to C indicates that the flip-flops respond to the negative-edge transition of the input. The negative transition occurs when the output of the previous flip-flop to which C is connected goes from 1 to 0.

To understand the operation of the four-bit binary ripple counter, refer to the first nine binary numbers listed in Table 6.4. The count starts with binary 0 and increments by 1 with each count pulse input. After the count of 15, the counter goes back to 0 to repeat the count. The least significant bit, A_0, is complemented with each count pulse input. Every time that A_0 goes from 1 to 0, it complements A_1. Every time that A_1 goes from 1 to 0, it complements A_2. Every time that A_2 goes from 1 to 0, it complements A_3, and so on for any other higher order bits of a ripple counter. For example, consider the transition from count 0011 to 0100. A_0 is complemented with the count pulse. Since A_0 goes from 1 to 0, it triggers A_1 and complements it. As a result, A_1 goes from 1 to 0, which in turn complements A_2, changing it from 0 to 1. A_2 does not trigger A_3, because A_2 produces a positive transition and the flip-flop responds only to negative transitions. Thus, the count from 0011 to 0100 is achieved by changing the bits one at a time, so the

Table 6.4
Binary Count Sequence

A_3	A_2	A_1	A_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0



(a) With *T* flip-flops (b) With *D* flip-flops

count goes from 0011 to 0010, then to 0000, and finally to 0100. The flip-flops change one at a time in succession, and the signal propagates through the counter in a ripple fashion from one stage to the next.

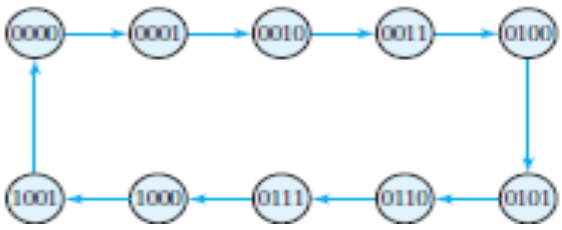
A binary counter with a reverse count is called a *binary countdown counter*. In a countdown counter, the binary count is decremented by 1 with every input count pulse. The count of a four-bit countdown counter starts from binary 15 and continues to binary counts 14, 13, 12, . . . , 0 and then back to 15. A list of the count sequence of a binary countdown counter shows that the least significant bit is complemented with every count pulse. Any other bit in the sequence is complemented if its previous least significant bit goes from 0 to 1. Therefore, the diagram of a binary countdown counter looks the same as the binary ripple counter in Fig. 6.8, provided that all flip-flops trigger on the positive edge of the clock. (The bubble in the *C* inputs must be absent.) If negative-edge-triggered flip-flops are used, then the *C* input of each flip-flop must be connected to the complemented output of the previous flip-flop. Then, when the true output goes from 0 to 1, the complement will go from 1 to 0 and complement the next flip-flop as required.

BCD Ripple Counter

A decimal counter follows a sequence of 10 states and returns to 0 after the count of 9. Such a counter must have at least four flip-flops to represent each decimal digit, since a decimal digit is represented by a binary code with at least four bits. The sequence of states in a decimal counter is dictated by the binary code used to represent a decimal digit. If BCD is used, the sequence of states is as shown in the state diagram of Fig. 6.9. A decimal counter is similar to a binary counter, except that the state after 1001 (the code for decimal digit 9) is 0000 (the code for decimal digit 0).

The logic diagram of a BCD ripple counter using *JK* flip-flops is shown in Fig. 6.10. The four outputs are designated by the letter symbol Q, with a numeric subscript equal to the binary weight of the corresponding bit in the BCD code. Note that the output of Q_1 is applied to the C inputs of both Q_2 and Q_8 and the output of Q_2 is applied to the C input of Q_4. The J and K inputs are connected either to a permanent 1 signal or to outputs of other flip-flops.

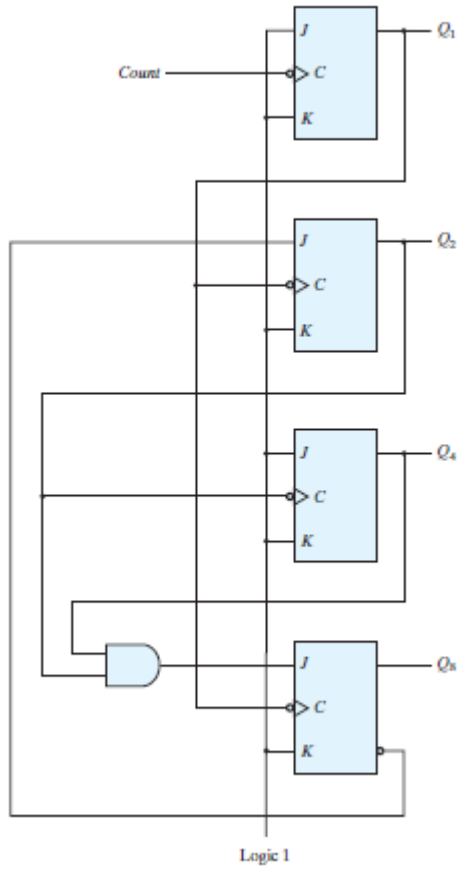
A ripple counter is an asynchronous sequential circuit. Signals that affect the flip-flop transition depend on the way they change from 1 to 0. The operation of the counter can



BCD ripple counter

be explained by a list of conditions for flip-flop transitions. These conditions are derived from the logic diagram and from knowledge of how a *JK* flip-flop operates. Remember that when the C input goes from 1 to 0, the flip-flop is set if $J = 1$, is cleared if $K = 1$, is complemented if $J = K = 1$, and is left unchanged if $J = K = 0$.

Registers and Counters



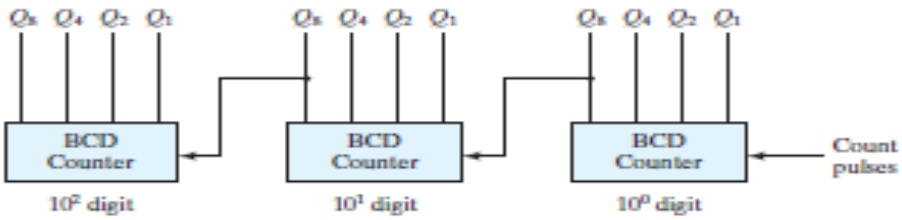


FIGURE 6.11
Block diagram of a three-decade decimal BCD counter

To verify that these conditions result in the sequence required by a BCD ripple counter, it is necessary to verify that the flip-flop transitions indeed follow a sequence of states as specified by the state diagram of Fig. 6.9. Q_1 changes state after each clock pulse. Q_2 complements every time Q_1 goes from 1 to 0, as long as $Q_8 = 0$. When Q_8 becomes 1, Q_2 remains at 0. Q_4 complements every time Q_2 goes from 1 to 0. Q_8 remains at 0 as long as Q_2 or Q_4 is 0. When both Q_2 and Q_4 become 1, Q_8 complements when Q_1 goes from 1 to 0. Q_8 is cleared on the next transition of Q_1.

The BCD counter of Fig. 6.10 is a *decade* counter, since it counts from 0 to 9. To count in decimal from 0 to 99, we need a two-decade counter. To count from 0 to 999, we need a three-decade counter. Multiple decade counters can be constructed by connecting BCD counters in cascade, one for each decade. A three-decade counter is shown in Fig. 6.11. The inputs to the second and third decades come from Q_8 of the previous decade. When Q_8 in one decade goes from 1 to 0, it triggers the count for the next higher order decade while its own decade goes from 9 to 0.

SYNCHRONOUS COUNTERS

Synchronous counters are different from ripple counters in that clock pulses are applied to the inputs of all flip-flops. A common clock triggers all flip-flops simultaneously, rather than one at a time in succession as in a ripple counter. The decision whether a flip-flop is to be complemented is determined from the values of the data inputs, such as T or J and K at the time of the clock edge. If $T = 0$ or $J = K = 0$, the flip-flop does not change state. If $T = 1$ or $J = K = 1$, the flip-flop complements.

The design procedure for synchronous counters was presented in Section 5.8, and the design of a three-bit binary counter was carried out in conjunction with Fig. 5.31. In this section, we present some typical synchronous counters and explain their operation.

Binary Counter

The design of a synchronous binary counter is so simple that there is no need to go through a sequential logic design process. In a synchronous binary counter, the flip-flop in the least significant position is complemented with every pulse. *A flip-flop in any other*

position is complemented when all the bits in the lower significant positions are equal to 1. For example, if the present state of a four-bit counter is $A_3A_2A_1A_0 = 0011$, the next count is 0100. A_0 is always complemented. A_1 is complemented because the present state of $A_0 = 1$. A_2 is complemented because the present state of $A_1A_0 = 11$. However, A_3 is not complemented, because the present state of $A_2A_1A_0 = 011$, which does not give an all-1's condition.

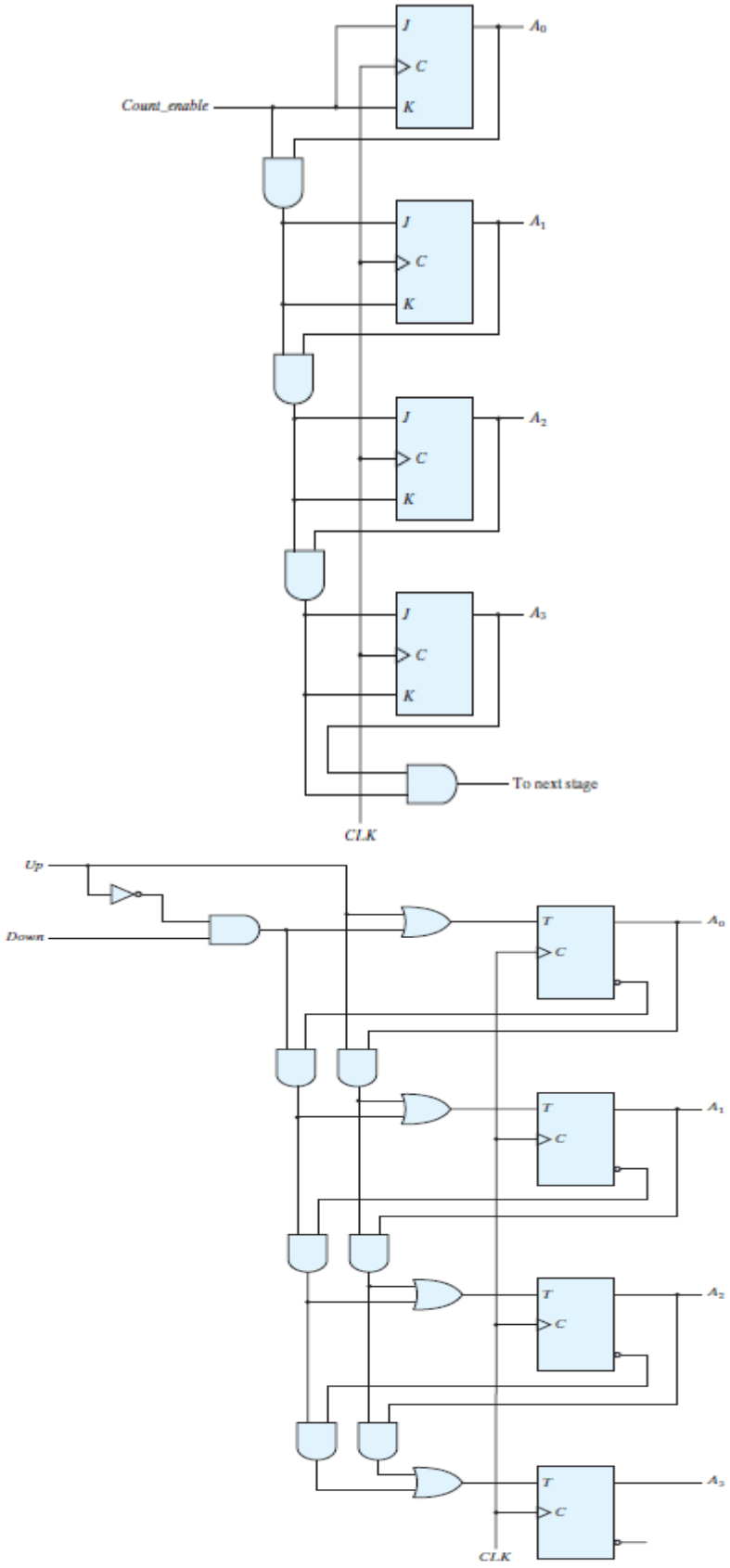
Synchronous binary counters have a regular pattern and can be constructed with complementing flip-flops and gates. The regular pattern can be seen from the four-bit counter depicted in Fig. 6.12. The C inputs of all flip-flops are connected to a common clock. The counter is enabled by *Count_enable*. If the enable input is 0, all J and K inputs are equal to 0 and the clock does not change the state of the counter. The first stage, A_0, has its J and K equal to 1 if the counter is enabled. The other J and K inputs are equal to 1 if all previous least significant stages are equal to 1 and the count is enabled. The chain of AND gates generates the required logic for the J and K inputs in each stage. The counter can be extended to any number of stages, with each stage having an additional flip-flop and an AND gate that gives an output of 1 if all previous flip-flop outputs are 1.

Note that the flip-flops trigger on the positive edge of the clock. The polarity of the clock is not essential here, but it is with the ripple counter. The synchronous counter can be triggered with either the positive or the negative clock edge. The complementing flip-flops in a binary counter can be of either the JK type, the T type, or the D type with XOR gates. The equivalency of the three types is indicated in Fig. 5.13.

Up–Down Binary Counter

A synchronous countdown binary counter goes through the binary states in reverse order, from 1111 down to 0000 and back to 1111 to repeat the count. It is possible to design a countdown counter in the usual manner, but the result is predictable by inspection of the downward binary count. The bit in the least significant position is complemented with each pulse. *A bit in any other position is complemented if all lower significant bits are equal to 0.* For example, the next state after the present state of 0100 is 0011. The least significant bit is always complemented. The second significant bit is complemented because the first bit is 0. The third significant bit is complemented because the first two bits are equal to 0. But the fourth bit does not change, because not all lower significant bits are equal to 0.

A countdown binary counter can be constructed as shown in Fig. 6.12, except that the inputs to the AND gates must come from the complemented outputs, instead of the normal outputs, of the previous flip-flops. The two operations can be combined in one circuit to form a counter capable of counting either up or down. The circuit of an up–down binary counter using T flip-flops is shown in Fig. 6.13. It has an up control input and a down control input. When the up input is 1, the circuit counts up, since the T inputs receive their signals from the values of the previous normal outputs of the flip-flops. When the down input is 1 and the up input is 0, the circuit counts down, since the complemented outputs of the previous flip-flops are applied to the T inputs. When the up and down inputs are both 0, the circuit does not change state and remains


J
C
K
A_0
Count_enable
A_1
A_2
A_3
To next stage
CLK
Up
Down
T
C
A_0
A_1
A_2
A_3
CLK

BCD Counter

A BCD counter counts in binary-coded decimal from 0000 to 1001 and back to 0000. Because of the return to 0 after a count of 9, a BCD counter does not have a regular pattern, unlike a straight binary count. To derive the circuit of a BCD synchronous counter, it is necessary to go through a sequential circuit design procedure.

The state table of a BCD counter is listed in Table 6.5. The input conditions for the T flip-flops are obtained from the present- and next-state conditions. Also shown in the table is an output y, which is equal to 1 when the present state is 1001. In this way, y can enable the count of the next-higher significant decade while the same pulse switches the present decade from 1001 to 0000.

The flip-flop input equations can be simplified by means of maps. The unused states for minterms 10 to 15 are taken as don't-care terms. The simplified functions are

$$T_{Q1} = 1$$
$$T_{Q2} = Q_8' Q_1$$
$$T_{Q4} = Q_2 Q_1$$
$$T_{Q8} = Q_8 Q_1 + Q_4 Q_2 Q_1$$
$$y = Q_8 Q_1$$

The circuit can easily be drawn with four T flip-flops, five AND gates, and one OR gate. Synchronous BCD counters can be cascaded to form a counter for decimal numbers of any length. The cascading is done as in Fig. 6.11, except that output y must be connected to the count input of the next-higher significant decade.

Table 6.5
State Table for BCD Counter

Present State				Next State				Output	Flip-Flop Inputs			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1	y	TQ_8	TQ_4	TQ_2	TQ_1
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	1	0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	1	0	0	1

Binary Counter with Parallel Load

Counters employed in digital systems quite often require a parallel-load capability for transferring an initial binary number into the counter prior to the count operation. Figure 6.14 shows the top-level block diagram symbol and the logic diagram of a four-bit register that has a parallel load capability and can operate as a counter. When equal to 1, the input load control disables the count operation and causes a transfer of data from the four data inputs into the four flip-flops. If both control inputs are 0, clock pulses do not change the state of the register.

The carry output becomes a 1 if all the flip-flops are equal to 1 while the count input is enabled. This is the condition for complementing the flip-flop that holds the next significant bit. The carry output is useful for expanding the counter to more than four bits. The speed of the counter is increased when the carry is generated directly from the outputs of all four flip-flops, because the delay to generate the carry bit is reduced. In going from state 1111 to 0000, only one gate delay occurs, whereas four gate delays occur in the AND gate chain shown in Fig. 6.12. Similarly, each flip-flop is associated with an AND gate that receives all previous flip-flop outputs directly instead of connecting the AND gates in a chain.

The operation of the counter is summarized in Table 6.6. The four control inputs—*Clear*, *CLK*, *Load*, and *Count*—determine the next state. The *Clear* input is asynchronous and, when equal to 0, causes the counter to be cleared regardless of the presence of clock pulses or other inputs. This relationship is indicated in the table by the X entries, which symbolize don't-care conditions for the other inputs. The *Clear* input must be in the 1 state for all other operations. With the *Load* and *Count* inputs both at 0, the outputs do not change, even when clock pulses are applied. A *Load* input of 1 causes a transfer from inputs $I_0 - I_3$ into the register during a positive edge of *CLK*. The input data are loaded into the register regardless of the value of the *Count* input, because the *Count* input is inhibited when the *Load* input is enabled. The *Load* input must be 0 for the *Count* input to control the operation of the counter.

A counter with a parallel load can be used to generate any desired count sequence. Figure 6.15 shows two ways in which a counter with a parallel load is used to generate the BCD count. In each case, the *Count* control is set to 1 to enable the count through the *CLK* input. Also, recall that the *Load* control inhibits the count and that the clear operation is independent of other control inputs.

The AND gate in Fig. 6.15(a) detects the occurrence of state 1001. The counter is initially cleared to 0, and then the *Clear* and *Count* inputs are set to 1, so the counter is active at all times. As long as the output of the AND gate is 0, each positive-edge clock

Table 6.6
Function Table for the Counter of Fig. 6.14

Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

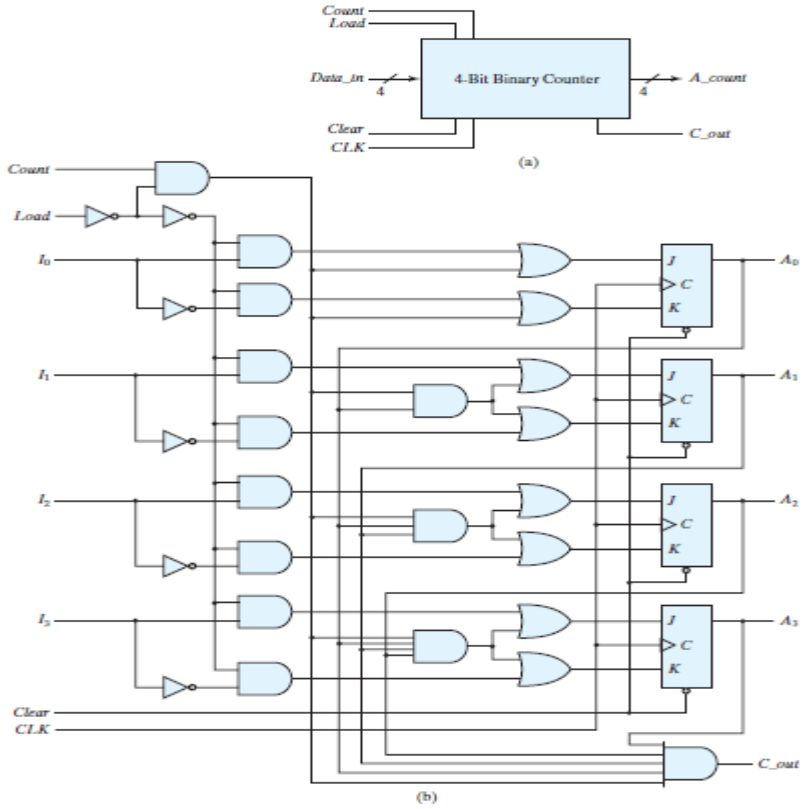


FIGURE 6.14
Four-bit binary counter with parallel load

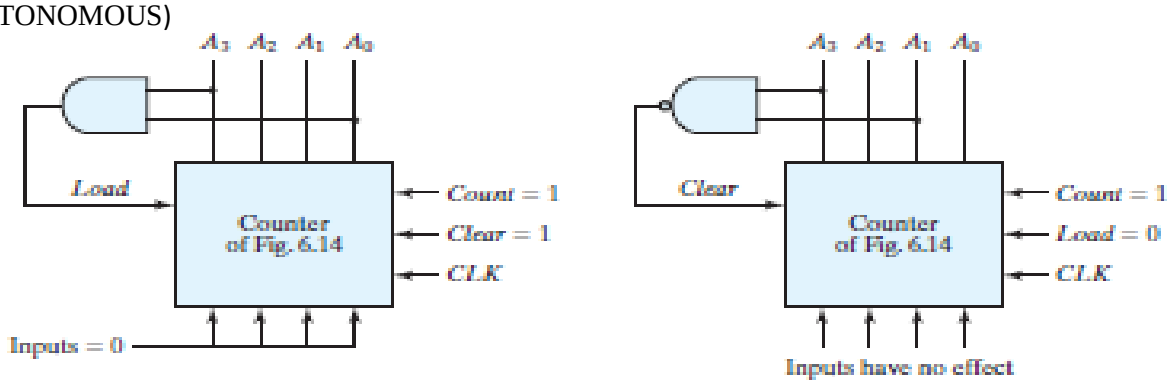


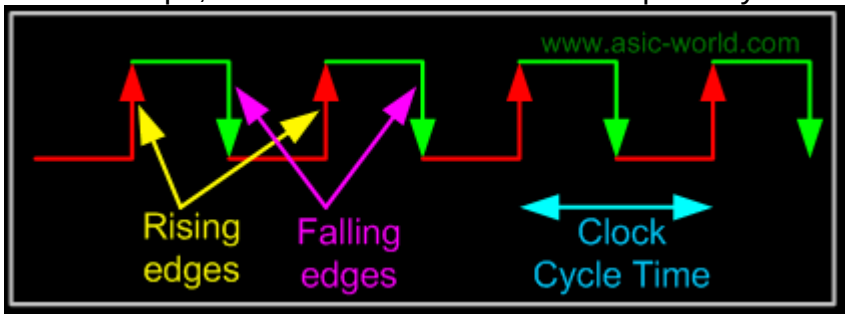
FIGURE 6.15
Two ways to achieve a BCD counter using a counter with parallel load

increments the counter by 1. When the output reaches the count of 1001, both A_0 and A_3 become 1, making the output of the AND gate equal to 1. This condition activates the *Load* input; therefore, on the next clock edge the register does not count, but is loaded from its four inputs. Since all four inputs are connected to logic 0, an all-0's value is loaded into the register following the count of 1001. Thus, the circuit goes through the count from 0000 through 1001 and back to 0000, as is required in a BCD counter.

In Fig. 6.15(b), the NAND gate detects the count of 1010, but as soon as this count occurs, the register is cleared. The count 1010 has no chance of staying on for any appreciable time, because the register goes immediately to 0. A momentary spike occurs in output A_0 as the count goes from 1010 to 1011 and immediately to 0000. The spike may be undesirable, and for that reason, this configuration is not recommended. If the counter has a synchronous clear input, it is possible to clear the counter with the clock after an occurrence of the 1001 count.

Synchronous sequentialcircuits

This type of system uses storage elements called flip-flops that are employed to change their binary value only at discrete instants of time. Synchronous sequential circuits use logic gates and flip-flop storage devices. Sequential circuits have a clock signal as one of their inputs. All state transitions in such circuits occur only when the clock value is either 0 or 1 or happen at the rising or falling edges of the clock depending on the type of memory elements used in the circuit. Synchronization is achieved by a timing device called a clock pulse generator. Clock pulses are distributed throughout the system in such a way that the flip-flops are affected only with the arrival of the synchronization pulse. Synchronous sequential circuits that use clock pulses in the inputs are called clocked-sequential circuits. They are stable and their timing can easily be broken down into independent discrete steps, each of which is consideredseparately.



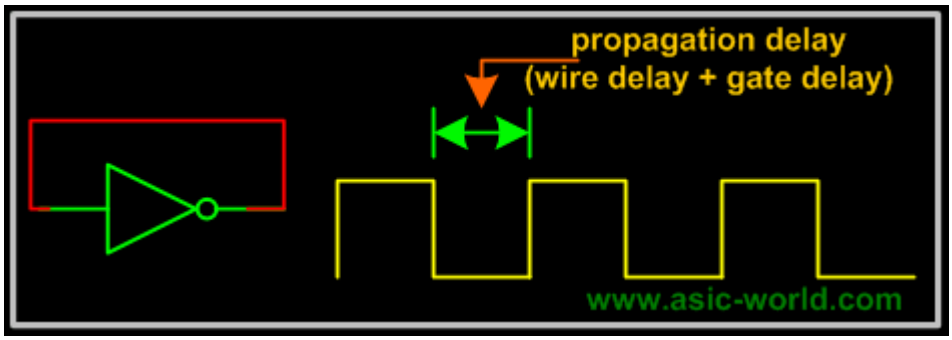
A clock signal is a periodic square wave that indefinitely switches from 0 to 1 and from 1 to 0 at fixed

intervals. Clock cycle time or clock period: the time interval between two consecutive rising or falling edges of theclock.

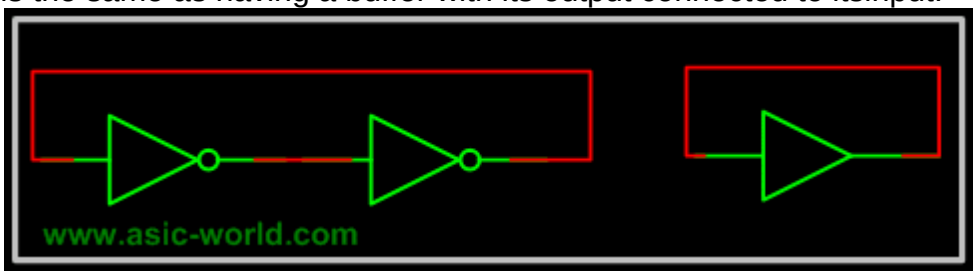
Clock Frequency = 1 / clock cycle time (measured in cycles per second orHz)

Concept of SequentialLogic

A sequential circuit as seen in the last page, is combinational logic with some feedback to maintain its current value, like a memory cell. To understand the basics let's consider the basic feedback logic circuit below, which is a simple NOT gate whose output is connected to its input. The effect is that output oscillates between HIGH and LOW (i.e. 1 and 0). Oscillation frequency depends on gate delay and wire delay. Assuming a wire delay of 0 and a gate delay of 10ns, then oscillation frequency would be (on time + off time = 20ns)50Mhz.



The basic idea of having the feedback is to store the value or hold the value, but in the above circuit, output keeps toggling. We can overcome this problem with the circuit below, which is basically cascading two inverters, so that the feedback is in-phase, thus avoids toggling. The equivalent circuit is the same as having a buffer with its output connected to itsinput.



But there is a problem here too: each gate output value is stable, but what will it be? Or in other words buffer output can not be known. There is no way to tell. If we could know or set the value we would have a simple 1-bit storage/memoryelement.

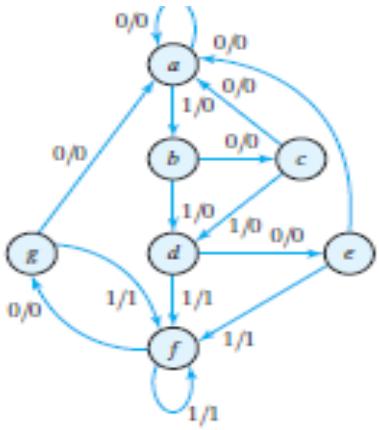


FIGURE 5.25
State diagram

by letter symbols instead of their binary values. This is in contrast to a binary counter, wherein the binary value sequence of the states themselves is taken as the outputs.

There are an infinite number of input sequences that may be applied to the circuit; each results in a unique output sequence. As an example, consider the input sequence 01010110100 starting from the initial state *a*. Each input of 0 or 1 produces an output of 0 or 1 and causes the circuit to go to the next state. From the state diagram, we obtain the output and state sequence for the given input sequence as follows: With the circuit in initial state *a*, an input of 0 produces an output of 0 and the circuit remains in state *a*. With present state *a* and an input of 1, the output is 0 and the next state is *b*. With present state *b* and an input of 0, the output is 0 and the next state is *c*. Continuing this process, we find the complete sequence to be as follows:

state	***a***	***a***	***b***	***c***	***d***	***e***	***f***	***f***	***g***	***f***	***g***	***a***
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

In each column, we have the present state, input value, and output value. The next state is written on top of the next column. It is important to realize that in this circuit the states themselves are of secondary importance, because we are interested only in output sequences caused by input sequences.

Now let us assume that we have found a sequential circuit whose state diagram has fewer than seven states, and suppose we wish to compare this circuit with the circuit whose state diagram is given by Fig. 5.25. If identical input sequences are applied to the two circuits and identical outputs occur for all input sequences, then the two circuits are said to be equivalent (as far as the input–output is concerned) and one may be replaced by the other. The problem of state reduction is to find ways of reducing the number of states in a sequential circuit without altering the input–output relationships.

We now proceed to reduce the number of states for this example. First, we need the state table; it is more convenient to apply procedures for state reduction with the use of a table rather than a diagram. The state table of the circuit is listed in Table 5.6 and is obtained directly from the state diagram.

The following algorithm for the state reduction of a completely specified state table is given here without proof: "Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an equivalent state." When two states are equivalent, one of them can be removed without altering the input–output relationships.

Now apply this algorithm to Table 5.6. Going through the state table, we look for two present states that go to the same next state and have the same output for both input combinations. States e and g are two such states: They both go to states a and f and have outputs of 0 and 1 for $x = 0$ and $x = 1$, respectively. Therefore, states g and e are equivalent, and one of these states can be removed. The procedure of removing a state and replacing it by its equivalent is demonstrated in Table 5.7. The row with present state g is removed, and state g is replaced by state e each time it occurs in the columns headed "Next State."

Present state f now has next states e and f and outputs 0 and 1 for $x = 0$ and $x = 1$, respectively. The same next states and outputs appear in the row with present state d. Therefore, states f and d are equivalent, and state f can be removed and replaced by d. The final reduced table is shown in Table 5.8. The state diagram for the reduced table consists of only five states and is shown in Fig. 5.26. This state diagram satisfies the original input–output specifications and will produce the required output sequence for any given input sequence. The following list derived from the state diagram of Fig. 5.26 is for the input sequence used previously (note that the same output sequence results, although the state sequence is different):

state	***a***	***a***	***b***	***c***	***d***	***e***	***d***	***d***	***e***	***d***	***e***	***a***
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

Table 5.6
State Table

	Next State		**Output**	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

Reducing the State Table

Present State	Next State $x=0$	Next State $x=1$	Output $x=0$	Output $x=1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

Table 5.8
Reduced State Table

Present State	Next State $x=0$	Next State $x=1$	Output $x=0$	Output $x=1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

In fact, this sequence is exactly the same as that obtained for Fig. 5.25 if we replace g by e and f by d.

Checking each pair of states for equivalency can be done systematically by means of a procedure that employs an implication table, which consists of squares, one for every suspected pair of possible equivalent states. By judicious use of the table, it is possible to determine all pairs of equivalent states in a state table.

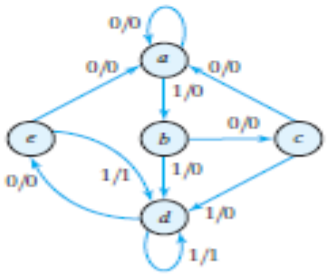


FIGURE 5.26
Reduced state diagram

State Assignment

In order to design a sequential circuit with physical components, it is necessary to assign unique coded binary values to the states. For a circuit with m states, the codes must contain n bits, where $2^n \geq m$. For example, with three bits, it is possible to assign codes to eight states, denoted by binary numbers 000 through 111. If the state table of Table 5.6 is used, we must assign binary values to seven states; the remaining state is unused. If the state table of Table 5.8 is used, only five states need binary assignment, and we are left with three unused states. Unused states are treated as don't-care conditions during the design. Since don't-care conditions usually help in obtaining a simpler circuit, it is more likely but not certain that the circuit with five states will require fewer combinational gates than the one with seven states.

The simplest way to code five states is to use the first five integers in binary counting order, as shown in the first assignment of Table 5.9. Another similar assignment is the Gray code shown in assignment 2. Here, only one bit in the code group changes when going from one number to the next. This code makes it easier for the Boolean functions to be placed in the map for simplification. Another possible assignment often used in the design of state machines to control data-path units is the one-hot assignment. This configuration uses as many bits as there are states in the circuit. At any given time, only one bit is equal to 1 while all others are kept at 0. This type of assignment uses one flip-flop per state, which is not an issue for register-rich field-programmable gate arrays. (See Chapter 7.) *One-hot encoding usually leads to simpler decoding logic for the next state and output.* One-hot machines can be faster than machines with sequential binary encoding, and the silicon area required by the extra flip-flops can be offset by the area

Table 5.9
Three Possible Binary State Assignments

State	Assignment 1, Binary	Assignment 2, Gray Code	Assignment 3, One-Hot
a	000	000	00001
b	001	001	00010
c	010	011	00100
d	011	010	01000
e	100	110	10000

Table 5.10
Reduced State Table with Binary Assignment 1

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
000	000	001	0	0
001	010	011	0	0
010	000	011	0	0
011	100	011	0	1
100	000	011	0	1

saved by using simpler decoding logic. This trade-off is not guaranteed, so it must be evaluated for a given design.

Table 5.10 is the reduced state table with binary assignment 1 substituted for the letter symbols of the states. A different assignment will result in a state table with different binary values for the states. The binary form of the state table is used to derive the next-state and output-forming combinational logic part of the sequential circuit. The complexity of the combinational circuit depends on the binary state assignment chosen.

Sometimes, the name *transition table* is used for a state table with a binary assignment. This convention distinguishes it from a state table with symbolic names for the states. In this book, we use the same name for both types of state tables.