Digital Logic Design

COURSE OBJECTIVES:

1) To design sequential logic circuits.
2) To impart to student the concepts of sequential circuits, enabling them to analyze sequential systems in terms of state machines.
3) To analyze Asynchronous sequential logic circuits.
4) To understand the concepts of verilog HDL
5) To design Digital Circuits using Gate, Behavioral and Data level Modeling Styles

UNIT –I

Sequential Machines Fundamentals: Introduction, Basic Architectural Distinctions between Combinational and Sequential circuits, classification of sequential circuits, The binary cell, The S-R-Latch Flip-Flop The D-Latch Flip- Flop, The "Clocked T" Flip-Flop, The " Clocked J-K" Flip-Flop, Design of a Clocked Flip-Flop, Conversion from one type of Flip-Flop to another.

UNIT –II

Sequential Circuit Design and Analysis: Introduction, classification of sequential circuits, State Diagram, State table, Analysis of Synchronous Sequential Circuits, State Reduction and assignment, Design procedure of synchronous sequential circuits.

Registers and Counters: Registers, Shift Registers, classification of counters, Design of Ripple counters and Synchronous counters, other counters.

UNIT –III

INTRODUCTION TO VERILOG HDL: Verilog as HDL, Levels of Design Description, Concurrency, Simulation and Synthesis, Programming Language Interface, Module.

Language Constructs and Conventions: Introduction, Keywords, Identifiers, White Space, Characters, Comments, Numbers, Strings, Logic Values, Data Types, Operators.

UNIT-IV

GATE LEVEL MODELING: Introduction, AND Gate Primitive, Module Structure, Other Gate Primitives, Illustrative Examples, Tristate Gates, Array of Instances of Primitives, Design of Flip-Flops with Gate Primitives, Delay.

MODELING AT DATAFLOW LEVEL: Introduction, Continuous Assignment Structure, Delays and Continuous Assignments, Assignment to Vector, Operators.

UNIT-V

BEHAVIORAL MODELING: Introduction, Operations and Assignments, 'Initial' Construct, Assignments with Delays, 'Wait 'Construct, Design at Behavioral Level, Blocking and Non-Blocking Assignments, The 'Case' Statement, 'If' an 'if-Else' Constructs, for loop, 'While Loop', Parallel Blocks.

TEXT BOOKS:

1) Digital Design- Morris Mano, PHI, 3rd Edition.
2) Switching Theory and Logic Design-A. Anand Kumar, PHI, 2nd Edition.
3)T.R. Padmanabhan, B Bala Tripura Sundari, Design Through Verilog HDL, Wiley 2009.
4) Verilog HDL - Samir Palnitkar, 2nd Edition, Pearson Education, 2009.

REFERENCE BOOKS:

1. Fundamentals of Logic Design –Charles H. Roth, 5th Ed., Cengage Learning.
Fundamentals of Digital Logic with Verilog Design - Stephen Brown,Zvonkoc Vranesic, TMH, 2nd Edition.
2. Zainalabdien Navabi, Verliog Digital System Design, TMH, 2nd Edition.
3. Advanced Digital Logic Design using Verilog, State Machines & Synthesis for FPGA - Sunggu Lee, Cengage Learning, 2012.
4. Advanced Digital Design with Verilog HDL - Michel D. Ciletti, PHI, 2009.

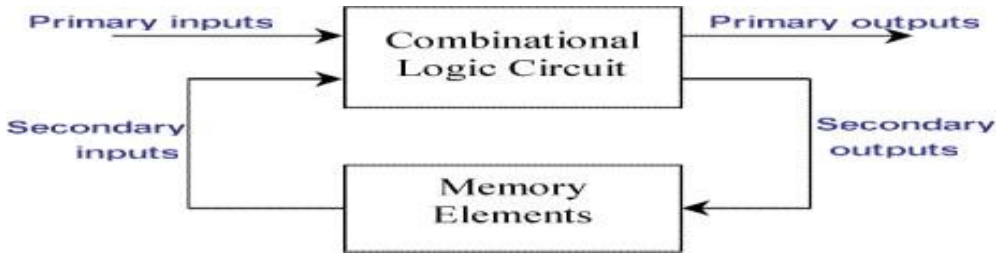
UNIT-I

Sequential machine fundamentals

Classification of sequential circuits: Sequential circuits may be classified as two types.

1. Synchronous sequential circuits
2. Asynchronous sequential circuits

Combinational logic refers to circuits whose output is strictly depended on the present value of the inputs. As soon as inputs are changed, the information about the previous inputs is lost, that is, combinational logics circuits have no memory. Although every digital system is likely to have combinational circuits, most systems encountered in practice also include memory elements, which require that the system be described in terms of sequential logic. Circuits whose output depends not only on the present input value but also the past input value are known as **sequential logic circuits**. The mathematical model of a sequential circuit is usually referred to as a **sequential machine**.



Comparison between combinational and sequential circuits

Combinational circuit	Sequential circuit
1. In combinational circuits, the output variables at any instant of time are dependent only on the present input variables	1. in sequential circuits the output variables at any instant of time are dependent not only on the present input variables, but also on the present state
2.memory unit is not requires in combinational circuit	2.memory unit is required to store the past history of the input variables
3. these circuits are faster because the delay between the i/p and o/p due to propagation delay of gates only	3. sequential circuits are slower than combinational circuits
4. easy to design	4. comparatively hard to design

Level mode and pulse mode asynchronous sequential circuits:

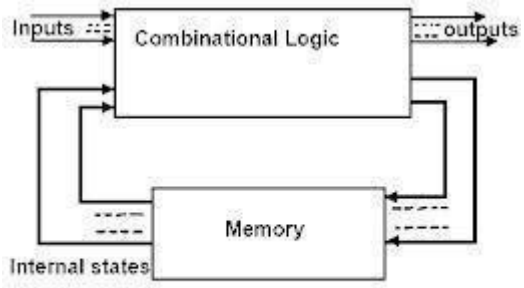


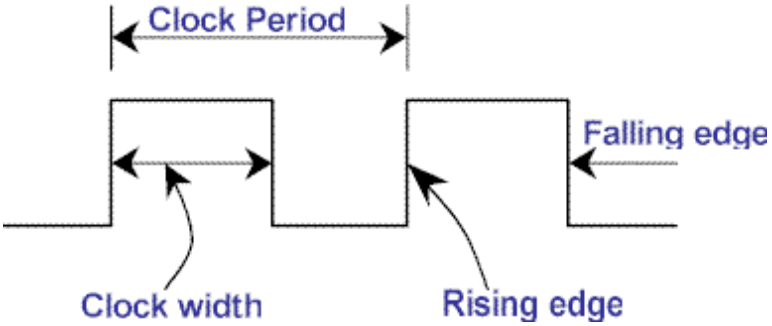
Figure 1: Asynchronous Sequential Circuit

Fig shows a block diagram of an asynchronous sequential circuit. It consists of a combinational circuit and delay elements connected to form the feedbackloops. The present state and next state variables in asynchronous sequential circuits called secondary variables and excitation variables respectively..

There are two types of asynchronous circuits: fundamental mode circuits and pulse mode circuits.

Synchronous and Asynchronous Operation:

Sequential circuits are divided into two main types: **synchronous** and **asynchronous**. Their classification depends on the timing of their signals.***Synchronous*** sequential circuits change their states and output values at discrete instants of time, which are specified by the rising and falling edge of a free-running **clock signal**. The clock signal is generally some form of square wave as shown in Figure below.



From the diagram you can see that the **clock period** is the time between successive transitions in the same direction, that is, between two rising or two falling edges. State transitions in synchronous sequential circuits are made to take place at times when the clock is making a transition from 0 to 1 (rising edge) or from 1 to 0 (falling edge). Between successive clock pulses there is no change in the information stored in memory.

The reciprocal of the clock period is referred to as the **clock frequency**. The **clock width** is defined as the time during which the value of the clock signal is equal to 1. The ratio of the clock width and clock period is referred to as the duty cycle. A clock signal is said to

be **active high** if the state changes occur at the clock's rising edge or during the clock width. Otherwise, the clock is said to be **active low**. Synchronous sequential circuits are also known as **clocked sequential circuits**.

The memory elements used in synchronous sequential circuits are usually flip-flops. These circuits are binary cells capable of storing one bit of information. A flip-flop circuit has two outputs, one for the normal value and one for the complement value of the bit stored in it. Binary information can enter a flip-flop in a variety of ways, a fact which give rise to the different types of flip-flops. For information on the different types of basic flip-flop circuits and their logical properties, see the previous tutorial on flip-flops.
In ***asynchronous*** sequential circuits, the transition from one state to another is initiated by the change in the primary inputs; there is no external synchronization. The memory commonly used in asynchronous sequential circuits are time-delayed devices, usually implemented by feedback among logic gates. Thus, asynchronous sequential circuits may be regarded as combinational circuits with feedback. Because of the feedback among logic gates, asynchronous sequential circuits may, at times, become unstable due to transient conditions. The instability problem imposes many difficulties on the designer. Hence, they are not as commonly used as synchronous systems.

Fundamental Mode Circuits assumes that:

1. The input variables change only when the circuit is stable
2. Only one input variable can change at a given time
3. Inputs are levels are not pulses

A pulse mode circuit assumes that:

1. The input variables are pulses instead of levels
2. The width of the pulses is long enough for the circuit to respond to theinput
3. The pulse width must not be so long that is still present after the new state is reached.

Latches and flip-flops

Latches and flip-flops are the basic elements for storing information. One latch or flip-flop can store one bit of information. The main difference between latches and flip-flops is that for latches, their outputs are constantly affected by their inputs as long as the enable signal is asserted. In other words, when they are enabled, their content changes immediately when their inputs change. Flip-flops, on the other hand, have their content change only either at the rising or falling edge of the enable signal. This enable signal is usually the controlling clock signal. After the rising or falling edge of the clock, the flip-flop content remains constant even if the input changes.

There are basically four main types of latches and flip-flops: SR, D, JK, and T. The major differences in these flip-flop types are the number of inputs they have and how they change state. For each type, there are also different variations that enhance their operations. In this chapter, we

will look at the operations of the various latches and flip-flops.the flip-flops has two outputs, labeled Q and Q‘. the Q output is the normal output of the flip flop and Q‘ is the inverted output.

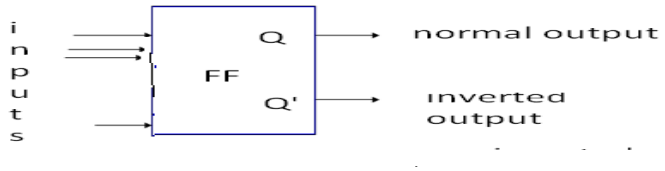


Figure: basic symbol of flipflop

A latch may be an active-high input latch or an active –LOW input latch.active –HIGH means that the SET and RESET inputs are normally resting in the low state and one of them will be pulsed high whenever we want to change latch outputs.

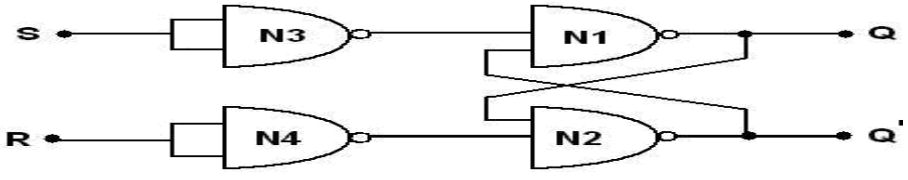
SR latch:

The latch has two outputs Q and Q‘. When the circuit is switched on the latch may enter into any state. If Q=1, then Q‘=0, which is called SET state. If Q=0, then Q‘=1, which is called RESET state. Whether the latch is in SET state or RESET state, it will continue to remain in the same state, as long as the power is not switched off. But the latch is not an useful circuit, since there is no way of entering the desired input. It is the fundamental building block in constructing flip-flops, as explained in the following sections

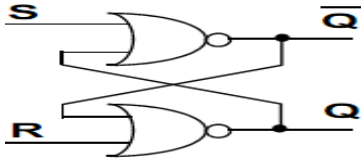
NAND latch

NAND latch is the fundamental building block in constructing a flip-flop. It has the property of holding on to any previous output, as long as it is not disturbed.

The opration of NAND latch is the reverse of the operation of NOR latch.if 0‘s are replaced by 1‘s and 1‘s are replaced by 0‘s we get the same truth table as that of the NOR latch shown



NOR latch



S	R	Q	$\overline{Q}$	Function
0	0	Q^+	$\overline{Q}^+$	Storage State
0	1	0	1	Reset
1	0	1	0	Set
1	1	0-?	0-?	Indeterminate State

The analysis of the operation of the active-HIGHNOR latch can be summarized as follows.

1. SET=0, RESET=0: this is normal resting state of the NOR latch and it has no effect on the output state. Q and Q‘ will remain in whatever stste they were prior to the occurrence of this input condition.
2. SET=1, RESET=0: this will always set Q=1, where it will remain even after SET returns to 0
3. SET=0, RESET=1: this will always reset Q=0, where it will remain even after RESET returns to 0
4. SET=1,RESET=1; this condition tries to SET and RESET the latch at the same time, and it produces Q=Q‘=0. If the inputs are returned to zero simultaneously, the resulting output stste is erratic and unpredictable. This input condition should not be used.

The SET and RESET inputs are normally in the LOW state and one of them will be pulsed HIGH. Whenever we want to change the latch outputs..

RS Flip-flop:

The basic flip-flop is a one bit memory cell that gives the fundamental idea of memory device. It constructed using two NAND gates. The two NAND gates N1 andN2 are connected such that, output of N1 is connected to input of N2 and output of N2 to input of N1. These form the feedback path the inputs are S and R, and outputs are Q and Q‘. The logic diagram and the block diagram of R-S flip-flop with clocked input

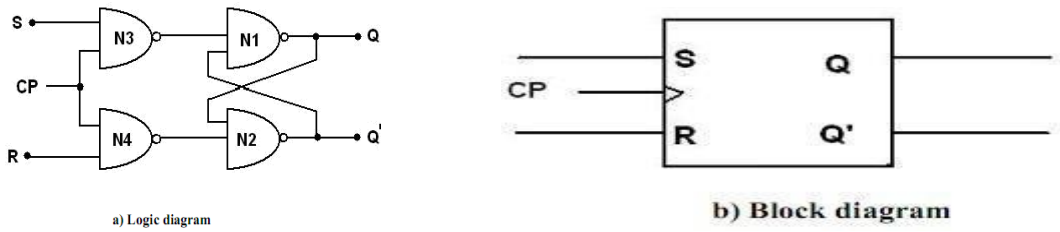


Figure: RS Flip-flop

The flip-flop can be made to respond only during the occurrence of clock pulse by adding two NAND gates to the input latch. So synchronization is achieved. i.e., flip-flops are allowed to change their states only at particular instant of time. The clock pulses are generated by a clock pulse generator. The flip-flops are affected only with the arrival of clock pulse**.**

Operation:

1. When CP=0 the output of N3 and N4 are 1 regardless of the value of S and R. This is given as input to N1 and N2. This makes the previous value of Q and Q‘unchanged.

2. When CP=1 the information at S and R inputs are allowed to reach the latch and change of state in flip-flop takes place.

3. CP=1, S=1, R=0 gives the SET state i.e., Q=1, Q‘=0.

4. CP=1, S=0, R=1 gives the RESET state i.e., Q=0, Q‘=1.

5. CP=1, S=0, R=0 does not affect the state of flip-flop.

6. CP=1, S=1, R=1 is not allowed, because it is not able to determine the next state. This condition is said to be a –race condition‖.

In the logic symbol CP input is marked with a triangle. It indicates the circuit responds to an input change from 0 to 1. The characteristic table gives the operation conditions of flip-flop. Q(t) is the present state maintained in the flip-flop at time _t‘. Q(t+1) is the state after the occurrence of clock pulse.

Truth table

S	R	$Q_{(t+1)}$	Comments
0	0	Q_t	No change
0	1	0	Reset / clear
1	0	1	Set
1	1	*	Not allowed

Edge triggered RS flip-flop:

Some flip-flops have an RC circuit at the input next to the clock pulse. By the design of the circuit the R-C time constant is much smaller than the width of the clock pulse. So the output changes will occur only at specific level of clock pulse. The capacitor gets fully charged when clock pulse goes from low to high. This change produces a narrow positive spike. Later at the trailing edge it produces narrow negative spike. This operation is called edge triggering, as the flip-flop responds only at the changing state of clock pulse. If output transition occurs at rising edge of clock pulse (0□1), it is called positively edge triggering. If it occurs at trailing edge (1□ 0) it is called negative edge triggering. Figure shows the logic and block diagram.

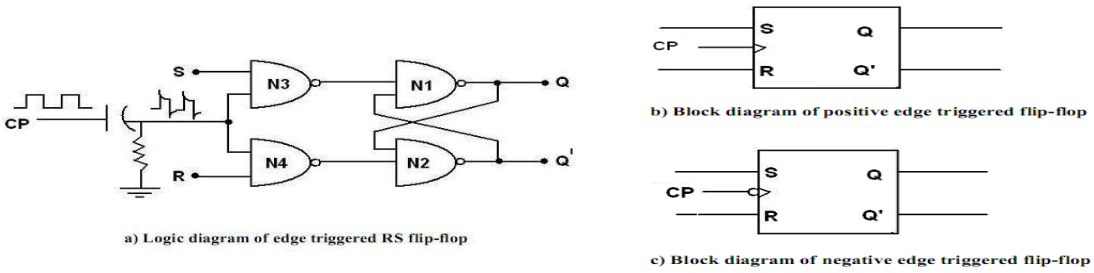
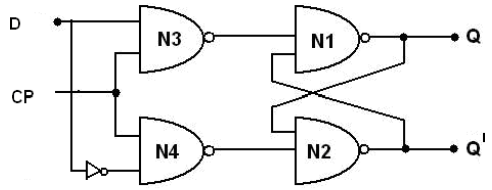


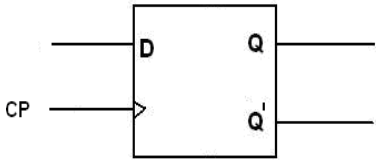
Figure: Edge triggered RS flip-flop

D flip-flop:

The D flip-flop is the modified form of R-S flip-flop. R-S flip-flop is converted to D flip-flop by adding an inverter between S and R and only one input D is taken instead of S and R. So one input is D and complement of D is given as another input. The logic diagram and the block diagram of D flip-flop with clocked input



a) Logic diagram



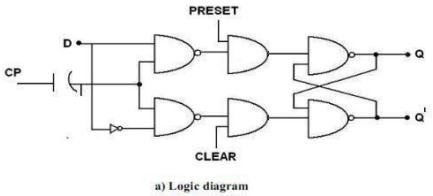
b) Block diagram

When the clock is low both the NAND gates (N1 and N2) are disabled and Q retains its last value. When clock is high both the gates are enabled and the input value at D is transferred to its output Q. D flip-flop is also called –Data flip-flop‖.

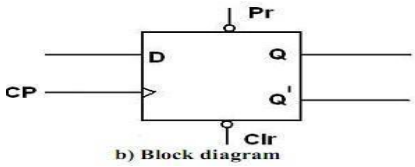
Truth table

CP	D	Q
0	x	Previous state
1	0	0
1	1	1

Edge Triggered D Flip-flop:



a) Logic diagram



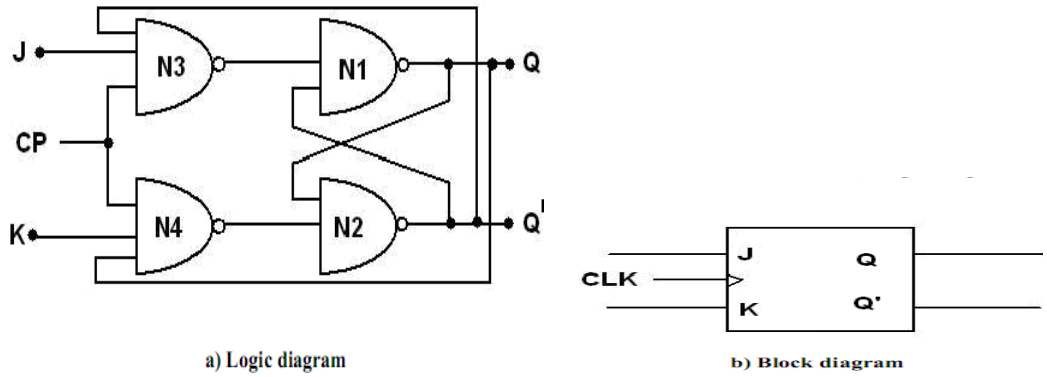
b) Block diagram

Truth table

PRESET	CLEAR	CP	D	Q
0	0	X	X	*(forbidden)
0	1	X	X	1
1	0	X	X	0
1	0	0	X	NC
1	1	1	X	NC
1	1	↓	X	NC
1	1	↑	0	0
1	1	↑	1	1

Figure: truth table, block diagram, logic diagram of edge triggered flip-flopJK flip-flop (edge triggered JK flip-flop)

The race condition in RS flip-flop, when R=S=1 is eliminated in J-K flip-flop. There is a feedback from the output to the inputs. Figure 3.4 represents one way of building a JK flip-flop.



a) Logic diagram

b) Block diagram

Truth table

J	K	$Q_{(t+1)}$	Comments
0	0	Q_t	No change
0	1	0	Reset / clear
1	0	1	Set
1	1	Q'_t	Complement/ toggle.

Figure: JK flip-flop

The J and K are called control inputs, because they determine what the flip-flop does when a positive clock edge arrives.

Operation:

1. When J=0, K=0 then both N3 and N4 will produce high output and the previous value of Q and Q‘ retained as it is.

2. When J=0, K=1, N3 will get an output as 1 and output of N4 depends on the value of Q. The final output is Q=0, Q‘=1 i.e., reset state

3. When J=1, K=0 the output of N4 is 1 and N3 depends on the value of Q‘. The final output is Q=1 and Q‘=0 i.e., set state

4. When J=1, K=1 it is possible to set (or) reset the flip-flop depending on the current state of output. If Q=1, Q‘=0 then N4 passes ‘0‘to N2 which produces Q‘=1, Q=0 which is reset state. When J=1, K=1, Q changes to the complement of the last state. The flip-flop is said to be in the toggle state.

The characteristic equation of the JK flip-flop is:

$$Q_{next} = J\overline{Q} + \overline{K}Q$$

JK flip-flop operation[28]								
Characteristic table				Excitation table				
J	**K**	**Q_{next}**	**Comment**	**Q**	**Q_{next}**	**J**	**K**	**Comment**
0	0	Q	hold state	0	0	0	X	No change
0	1	0	reset	0	1	1	X	Set
1	0	1	set	1	0	X	1	Reset
1	1	Q	toggle	1	1	X	0	No change

T flip-flop:

If the T input is high, the T flip-flop changes state ("toggles") whenever the clock input is strobed. If the T input is low, the flip-flop holds the previous value. This behavior is described by the characteristic equation

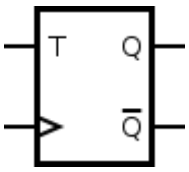


Figure : symbol for T flip flop

$$Q_{next} = T \oplus Q = T\overline{Q} + \overline{T}Q$$ (expanding the XOR operator

When T is held high, the toggle flip-flop divides the clock frequency by two; that is, if clock frequency is 4 MHz, the output frequency obtained from the flip-flop will be 2 MHz This "divide by" feature has application in various types of digital counters. A T flip-flop can also be built using a JK flip-flop (J & K pins are connected together and act as T) or D flip-flop (T input and $P_{revious}$ is connected to the D input through an XOR gate).

T flip-flop operation[28]							
Characteristic table				**Excitation table**			
T	Q	Q_{next}	**Comment**	Q	Q_{next}	T	**Comment**
0	0	0	hold state (no clk)	0	0	0	No change
0	1	1	hold state (no clk)	1	1	0	No change
1	0	1	toggle	0	1	1	Complement
1	1	0	toggle	1	0	1	Complement

Flip flop operating characteristics:

The operation characteristics specify the performance, operating requirements, and operating limitations of the circuits. The operation characteristics mentions here apply to all flip-flops regardless of the particular form of the circuit.

Propagation Delay Time: is the interval of time required after an input signal has been applied for the resulting output change to occur.

Set-up Time: is the minimum interval required for the logic levels to be maintained constantly on the inputs (J and K, or S and R, or D) prior to the triggering edge of the clock pulse in order for the levels to be reliably clocked into the flip-flop.

Hold Time: is the minimum interval required for the logic levels to remain on the inputs after the triggering edge of the clock pulse in order for the levels to be reliably clocked into the flip-flop.

Maximum Clock Frequency: is the highest rate that a flip-flop can be reliably triggered.

Power Dissipation: is the total power consumption of the device. It is equal to product of supply voltage (Vcc) and the current (Icc).

$P=V_{cc}.I_{cc}$

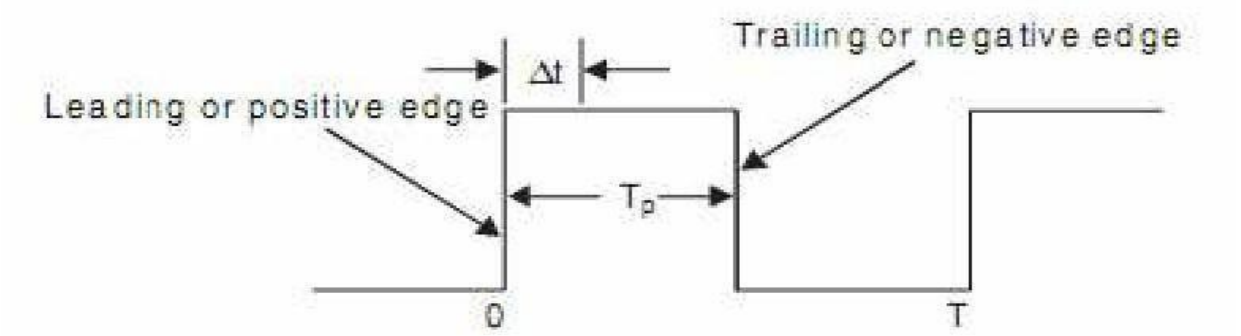
The power dissipation of a flip flop is usually in mW.

Pulse Widths: are the minimum pulse widths specified by the manufacturer for the Clock, SET and CLEAR inputs.

Clock transition times: for reliable triggering, the clock waveform transition times should be kept very short. If the clock signal takes too long to make the transitions from one level to other, the flip flop may either triggering erratically or not trigger at all.

Race around Condition

The inherent difficulty of an S-R flip-flop (i.e., S = R = 1) is eliminated by using the feedback connections from the outputs to the inputs of gate 1 and gate 2 as shown in Figure. Truth tables in figure were formed with the assumption that the inputs do not change during the clock pulse (CLK = 1). But the consideration is not true because of the feedback connections



- Consider, for example, that the inputs are J = K = 1 and Q = 1, and a pulse as shown in Figure is applied at the clock input.
- After a time interval t equal to the propagation delay through two NAND gates in series, the outputs will change to Q = 0. So now we have J = K = 1 and Q = 0.
- After another time interval of t the output will change back to Q = 1. Hence, we conclude that for the time duration of tP of the clock pulse, the output will oscillate between 0 and 1. Hence, at the end of the clock pulse, the value of the output is not certain. This situation is referred to as a race-around condition.
- Generally, the propagation delay of TTL gates is of the order of nanoseconds. So if the clock pulse is of the order of microseconds, then the output will change thousands of times within the clock pulse.
- This race-around condition can be avoided if tp< t < T. Due to the small propagation delay of the ICs it may be difficult to satisfy the above condition.
- A more practical way to avoid the problem is to use the master-slave (M-S) configuration as discussed below.

Applications of flip-flops:

Frequency Division: When a pulse waveform is applied to the clock input of a J-K flip-flop that is connected to toggle, the Q output is a square wave with half the frequency of the clock input. If more flip-flops are connected together as shown in the figure below, further division of the clock frequency can be achieved

. **Parallel data storage:** a group of flip-flops is called register. To store data of N bits, N flip-flops are required. Since the data is available in parallel form. When a clock pulse is applied to all flip-flops simultaneously, these bits will transfer will be transferred to the Q outputs of the flip flops.

Serial data storage: to store data of N bits available in serial form, N number of D-flip-flops is connected in cascade. The clock signal is connected to all the flip-flops. The serial data is applied to the D input terminal of the first flip-flop.

Transfer of data: data stored in flip-flops may be transferred out in a serial fashion, i.e., bit-by-bit from the output of one flip-flops or may be transferred out in parallel form.

Excitation Tables:

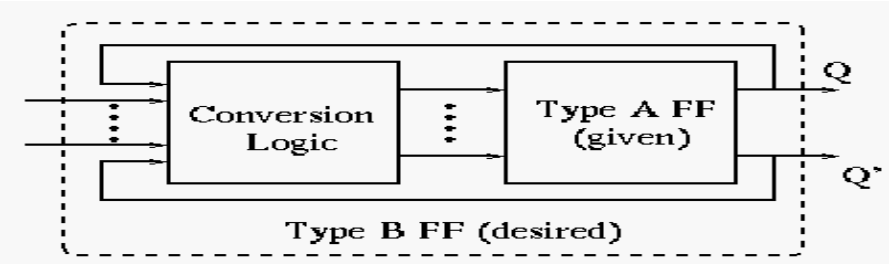
Previous State -> Present State	D
0 -> 0	0
0 -> 1	1
1 -> 0	0
1 -> 1	1

Previous State -> Present State	J	K
0 -> 0	0	X
0 -> 1	1	X
1 -> 0	X	1
1 -> 1	X	0

Previous State -> Present State	S	R
0 -> 0	0	X
0 -> 1	1	0
1 -> 0	0	1
1 -> 1	X	0

Previous State -> Present State	T
0 -> 0	0
0 -> 1	1
1 -> 0	1
1 -> 1	0

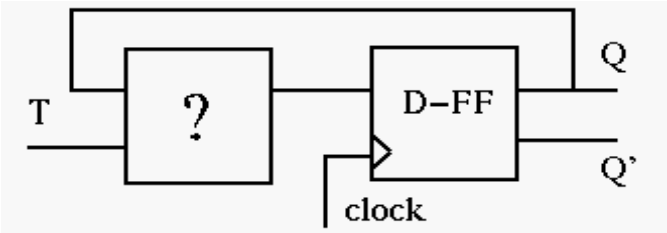
Conversions of flip-flops:



The key here is to use the excitation table, which shows the necessary triggering signal (S,R,J,K, D and T) for a desired flip-flop state transition :

Q_t	Q_{t+1}	S	R	J	K	D	T
0	0	0	x	0	x	0	0
0	1	1	0	1	x	1	1
1	0	0	1	x	1	0	1
1	1	x	0	x	0	1	0

Convert a D-FF to a T-FF:



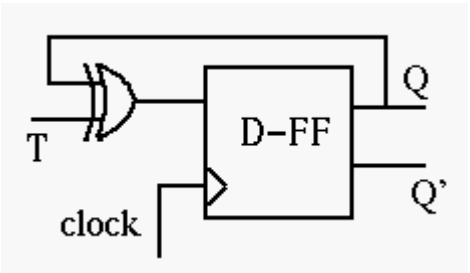
We need to design the circuit to generate the triggering signal D as a function of T and Q:
. Consider the excitation table:

$$D = f(T, Q).$$

Q_t	Q_{t+1}	T	D
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

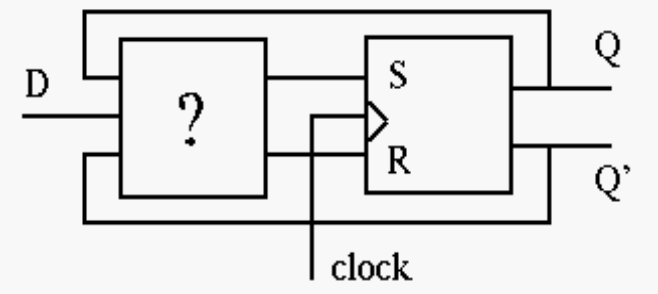
Treating as a function of and current FF state , we have

$$D = T'Q + TQ' = T \oplus Q$$



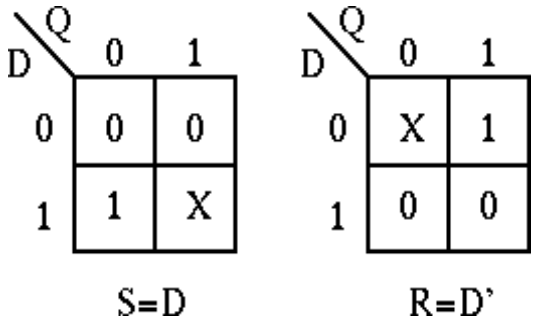
Convert a RS-FF to a D-FF:

We need to design the circuit to generate the triggering signals S and R as functions of and consider the excitation table:

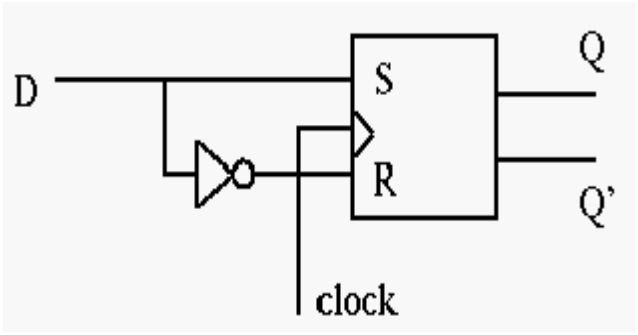


Q_t	Q_{t+1}	D	S	R
0	0	0	0	x
0	1	1	1	0
1	0	0	0	1
1	1	1	x	0

The desired signal and can be obtained as functions of and current FF state from the Karnaugh maps:



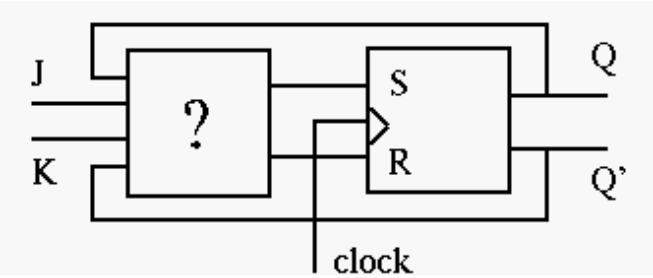
$S = D, \quad R = D'$



Convert a RS-FF to a JK-FF:

We need to design the circuit to generate the triggering signals S and R as functions of, J, K.

Consider the excitation table**:** The desired signal and as functions of, and current FF state can be obtained from the Karnaugh maps:



Q_t	Q_{t+1}	J	K	S	R
0	0	0	x	0	x
0	1	1	x	1	0
1	0	x	1	0	1
1	1	x	0	x	0

K-maps:

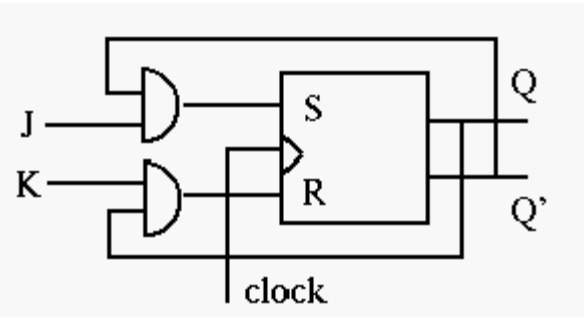
K \ QJ	00	01	11	10
0	0	1	X	X
1	0	1	0	0

S=Q'J

K \ QJ	00	01	11	10
0	X	0	0	0
1	X	0	1	1

R=QK

$S = Q'J, \quad R = QK$



The Master-Slave JK Flip-flop:

The Master-Slave Flip-Flop is basically two gated SR flip-flops connected together in a series configuration with the slave having an inverted clock pulse. The outputs from Q and Q from the "Slave" flip-flop are fed back to the inputs of the "Master" with the outputs of the "Master" flip-flop being connected to the two inputs of the "Slave" flip-flop. This feedback configuration from the slave's output to the master's input gives the characteristic toggle of the JK flip-flop as shown below.

The input signals J and K are connected to the gated "master" SR flip-flop which "locks" the input condition while the clock (Clk) input is "HIGH" at logic level "1". As the clock input of the "slave" flip-flop is the inverse (complement) of the "master" clock input, the "slave" SR flip-flop does not toggle. The outputs from the "master" flip-flop are only "seen" by the gated "slave" flip-flop when the clock input goes "LOW" to logic level "0". When the clock is "LOW", the outputs from the "master" flip-flop are latched and any additional changes to its inputs are ignored. The gated "slave" flip-flop now responds to the state of its inputs passed over by the "master" section. Then on the "Low-to-High" transition of the clock pulse the inputs of the "master" flip-flop are fed through to the gated inputs of the "slave" flip-flop and on the "High-to-Low" transition the same inputs are reflected on the output of the "slave" making this type of flip-flop edge or pulse-triggered. Then, the circuit accepts input data when the clock signal is "HIGH", and passes the data to the output on the falling-edge of the clock signal. In other words, the Master-Slave JK Flip-flop is a "Synchronous" device as it only passes data with the timing of the clock signal.

UNIT-II

Sequential circuit design and analysis

Sequential Circuit
Design

- Steps in the design process for sequential circuits
- State Diagrams and State Tables
- Examples

Steps in Design of a Sequential Circuit

1. Specification – A description of the sequential circuit. Should include a detailing of the inputs, the outputs, and the operation. Possibly assumes that you have knowledge of digital system basics.
2. Formulation: Generate a state diagram and/or a state table from the statement of the problem.
3. State Assignment: From a state table assign binary codes to the states.
4. Flip-flop Input Equation Generation: Select the type of flip-flop for the circuit and generate the needed input for the required state transitions
5. Output Equation Generation: Derive output logic equations for generation of the output from the inputs and current state.
6. Optimization: Optimize the input and output equations. Today, CAD systems are typically used for this in real systems.
7. Technology Mapping: Generate a logic diagram of the circuit using ANDs, ORs, Inverters, and F/Fs.
8. Verification: Use a HDL to verify the design.

Mealy and Moore

- Sequential machines are typically classified as either a Mealy machine or a Moore machine implementation.
- Moore machine: The outputs of the circuit depend only upon the current state ofthe circuit.
- Mealy machine: The outputs of the circuit depend upon both the current state ofthe circuit and the inputs.

An example to go through the steps

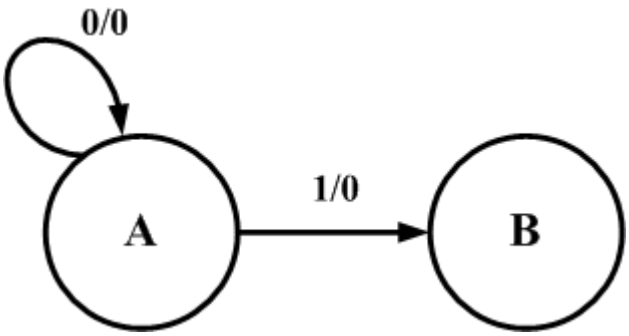
The specification: The circuit will have one input, X, and one output, Z. The output Z will be 0 except when the input sequence 1101 are the last 4 inputs received on X. In that case it will be a 1

Generation of a state diagram

- Create states and meaning for them.

State A – the last input was a 0 and previous inputs unknown. Can also be the reset state.
State B – the last input was a 1 and the previous input was a 0. The start of a new sequence possibly.

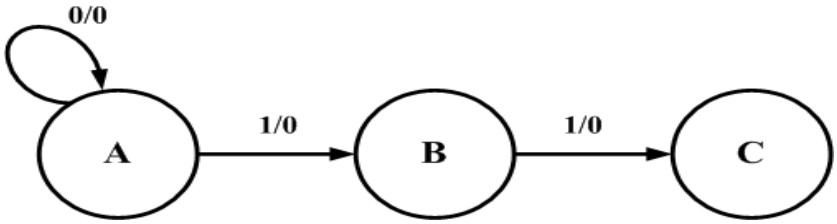
- Capture this in a state diagram



- Capture this in a state diagram

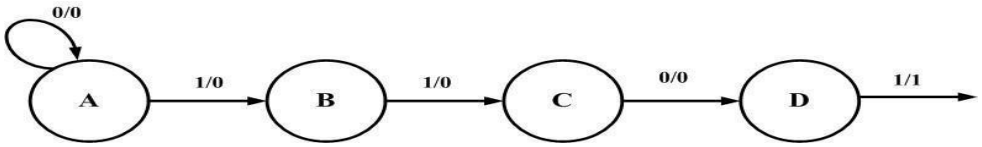
- Circles represent the states
- Lines and arcs represent the transition between states.
- The notation Input/output on the line or arc specifies the input that causes this transition and the output for this change of state.

- Add a state C – Have detected the input sequence 11 which is the start of the sequence



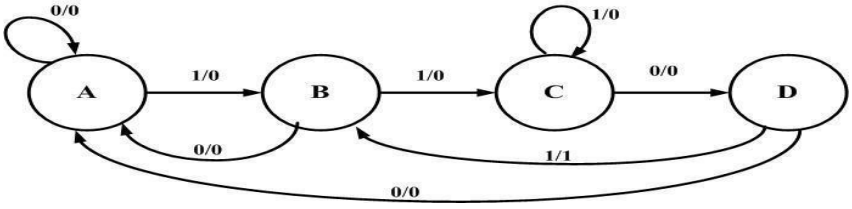
- Add a state D

  State D – have detected the 3rd input in the start of a sequence, a 0, now having 110. From State D, if the next input is a 1 the sequence has been detected and a 1 is output.



- The previous diagram was incomplete.

- In each state the next input could be a 0 or a 1. This must be included



- The state table
- This can be done directly from the state diagram

Prresent State	Next State X =0	Next State X=1	Output X=0	Output X=1
A	A	B	0	0
B	A	C	0	0
C	D	C	0	0
D	A	B	0	1

- Now need to do a state assignment

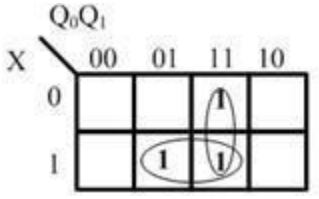
Select a state assignment

- Will select a gray encoding
- For this state A will be encoded 00, state B 01, state C 11 and state D 10

Prresent State	Next State X =0	Next State X=1	Output X=0	Output X=1
00	00	01	0	0
01	00	11	0	0
11	10	11	0	0
10	00	01	0	1

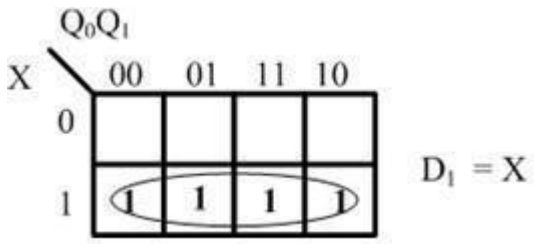
Flip-flop input equations

- Generate the equations for the flip-flop inputs
- Generate the D_0 equation



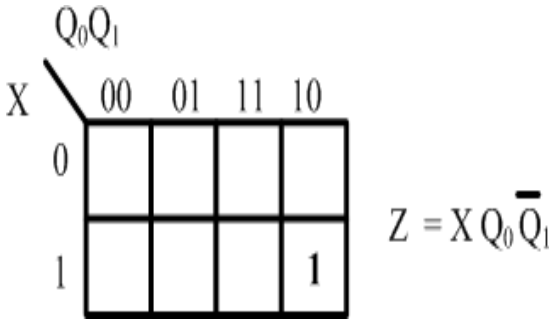
$D_0 = Q_0 Q_1 + X Q_1$

- Generate the D_1 equation



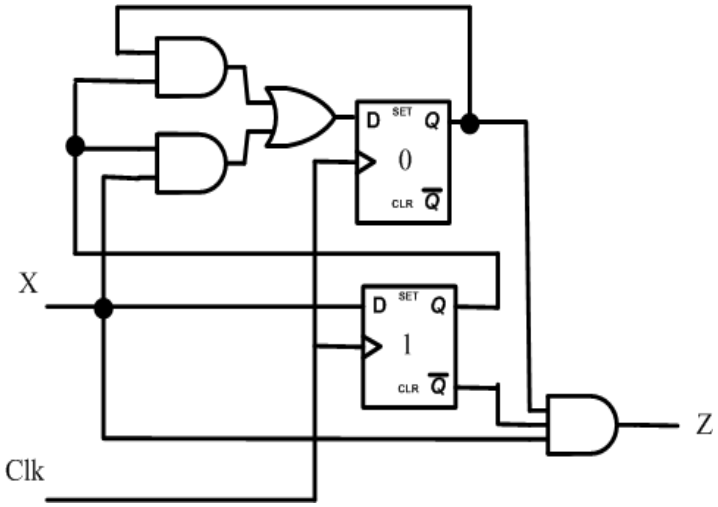
The output equation

- The next step is to generate the equation for the output Z and what is needed to generate it.
- Create a K-map from the truth table.



Now map to a circuit

- The circuit has 2 D type F/Fs



Shift registers:

In digital circuits, a **shift register** is a cascade of flip-flops sharing the same clock, in which the output of each flip-flop is connected to the "data" input of the next flip-flop in the chain, resulting in a circuit that shifts by one position the "bit array" stored in it, *shifting in* the data present at its input and *shifting out* the last bit in the array, at each transition of the clock input. More generally, a **shift register** may be multidimensional, such that its "data in" and stage outputs are themselves bit arrays: this is implemented simply by running several shift registers of the same bit-length in parallel.

Shift registers can have both parallel and serial inputs and outputs. These are often configured as **serial-in, parallel-out** (SIPO) or as **parallel-in, serial-out** (PISO). There are also types that have both serial and parallel input and types with serial and parallel output. There are also **bi-directional** shift registers which allow shifting in both directions: L→R or R→L. The serial input and last output of a shift register can also be connected to create a **circular shift register**

Shift registers are a type of logic circuits closely related to counters. They are basically for the storage and transfer of digital data.

Buffer register:

The buffer register is the simple set of registers. It is simply stores the binary word. The buffer may be controlled buffer. Most of the buffer registers used D Flip-flops.

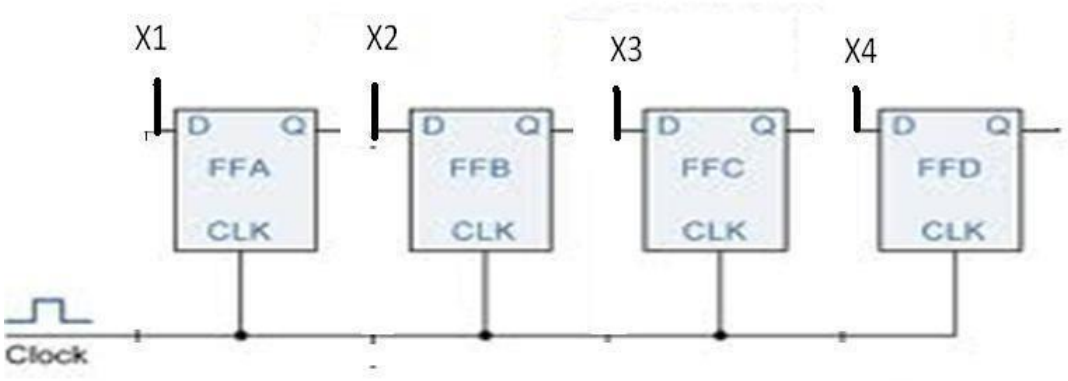


Figure: logic diagram of 4-bit buffer register

The figure shows a 4-bit buffer register. The binary word to be stored is applied to the data terminals. On the application of clock pulse, the output word becomes the same as the word applied at the terminals. i.e., the input word is loaded into the register by the application of clock pulse.

When the positive clock edge arrives, the stored word becomes:

$Q_4Q_3Q_2Q_1=X_4X_3X_2X_1$

$Q=X$

Controlled buffer register:

If goes LOW, all the FFs are RESET and the output becomes, Q=0000.

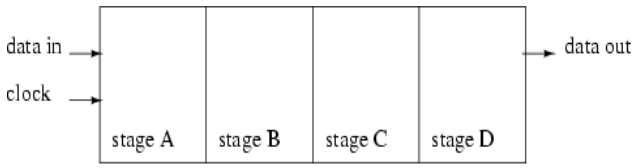
When is HIGH, the register is ready for action. LOAD is the control input. When LOAD is HIGH, the data bits X can reach the D inputs of FF‘s.

$Q_4Q_3Q_2Q_1=X_4X_3X_2X_1$

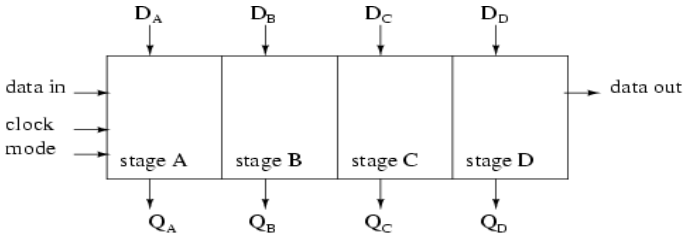
$Q=X$

When load is low, the X bits cannot reach the FF‘s.

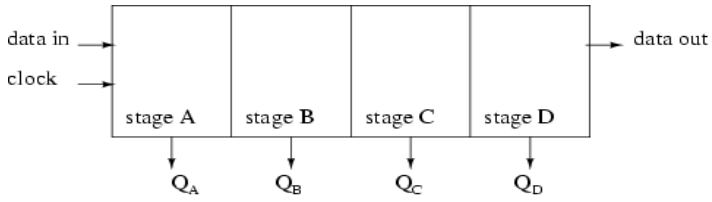
Data transmission in shift registers:



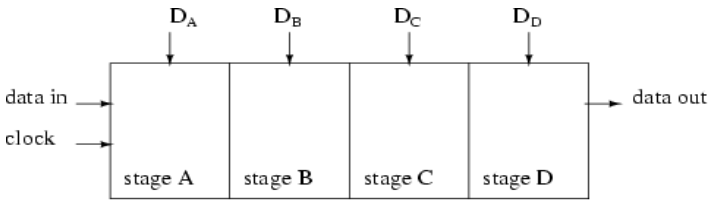
Serial-in, serial-out shift register with 4-stages



Parallel-in, parallel-out shift register with 4-stages



Serial-in, parallel-out shift register with 4-stages



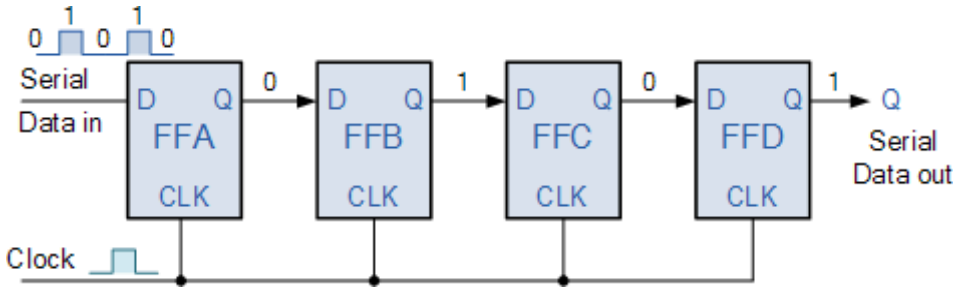
Parallel-in, serial-out shift register with 4-stages

A number of ff's connected together such that data may be shifted into and shifted out of them is called shift register. data may be shifted into or out of the register in serial form or in parallel form. There are four basic types of shift registers.

1. Serial in, serial out, shift right, shift registers
2. Serial in, serial out, shift left, shift registers
3. Parallel in, serial out shift registers
4. Parallel in, parallel out shift registers

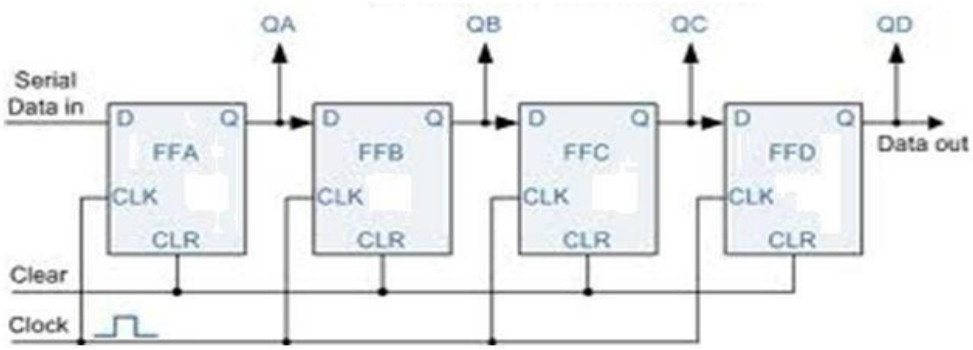
Serial IN, serial OUT, shift right, shift left register:

The logic diagram of 4-bit serial in serial out, right shift register with four stages. The register can store four bits of data. Serial data is applied at the input D of the first FF. the Q output of the first FF is connected to the D input of another FF. the data is outputted from the Q terminal of the last FF.



When serial data is transferred into a register, each new bit is clocked into the first FF at the positive going edge of each clock pulse. The bit that was previously stored by the first FF is transferred to the second FF. the bit that was stored by the Second FF is transferred to the third FF.

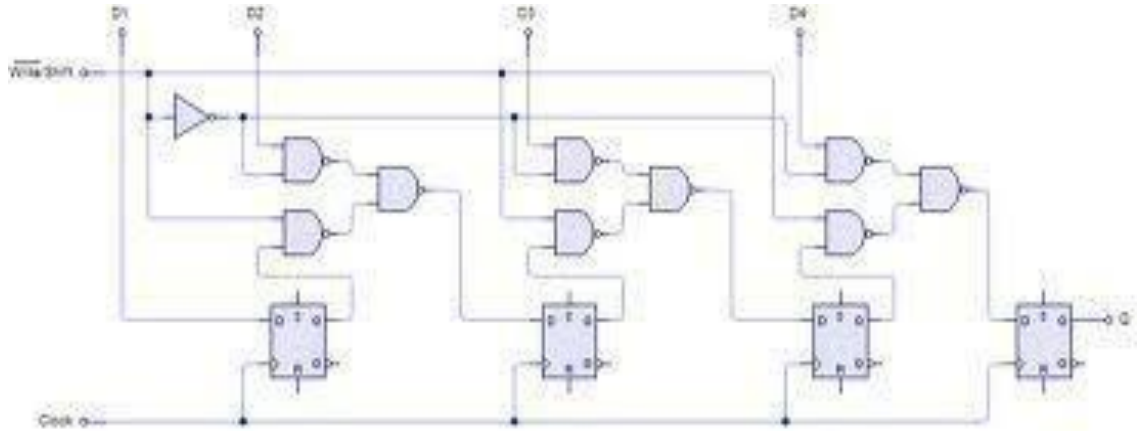
Serial-in, parallel-out, shift register:



In this type of register, the data bits are entered into the register serially, but the data stored in the register is shifted out in parallel form.

Once the data bits are stored, each bit appears on its respective output line and all bits are available simultaneously, rather than on a bit-by-bit basis with the serial output. The serial-in, parallel out, shift register can be used as serial-in, serial out, shift register if the output is taken from the Q terminal of the last FF.

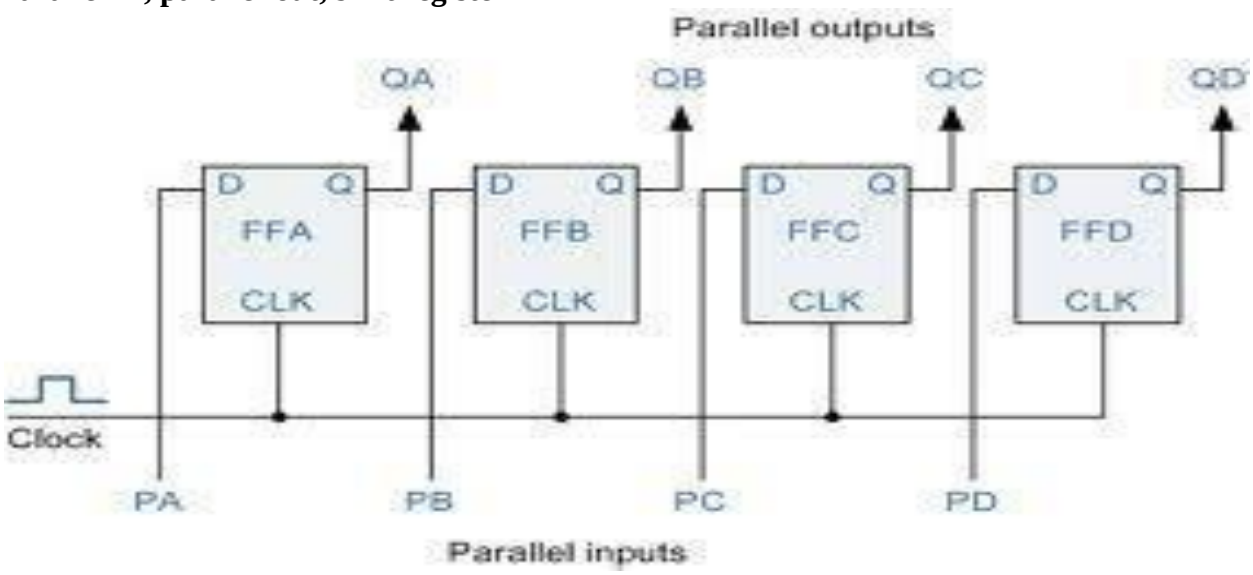
Parallel-in, serial-out, shift register:



For a parallel-in, serial out, shift register, the data bits are entered simultaneously into their respective stages on parallel lines, rather than on a bit-by-bit basis on one line as with serial data bits are transferred out of the register serially. On a bit-by-bit basis over a single line.

There are four data lines A,B,C,D through which the data is entered into the register in parallel form. The signal shift/ load allows the data to be entered in parallel form into the register and the data is shifted out serially from terminalQ4

Parallel-in, parallel-out, shift register



In a parallel-in, parallel-out shift register, the data is entered into the register in parallel form, and also the data is taken out of the register in parallel form. Data is applied to the D input terminals of the FF‘s. When a clock pulse is applied, at the positive going edge of the pulse, the D inputs are shifted into the Q outputs of the FFs. The register now stores the data. The stored data is available instantaneously for shifting out in parallel form.

Bidirectional shift register:

A bidirectional shift register is one which the data bits can be shifted from left to right or from right to left. A fig shows the logic diagram of a 4-bit serial-in, serial out, bidirectional shift register. Right/left is the mode signal, when right /left is a 1, the logic circuit works as a shift-register.the bidirectional operation is achieved by using the mode signal and two NAND gates and one OR gate for each stage.

A HIGH on the right/left control input enables the AND gates G1, G2, G3 and G4 and disables the AND gates G5,G6,G7 and G8, and the state of Q output of each FF is passed through the gate to the D input of the following FF. when a clock pulse occurs, the data bits are then effectively shifted one place to the right. A LOW on the right/left control inputs enables the AND gates G5, G6, G7 and G8 and disables the And gates G1, G2, G3 and G4 and the Q output of each FF is passed to the D input of the preceding FF. when a clock pulse occurs, the data bits are then effectively shifted one place to the left. Hence, the circuit works as a bidirectional shift register

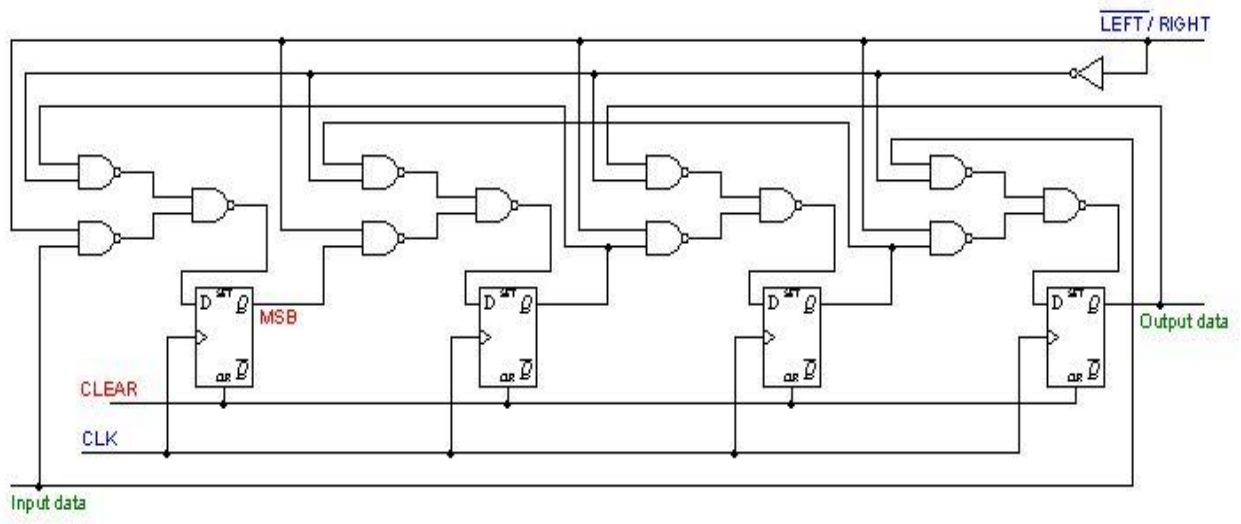


Figure: logic diagram of a 4-bit bidirectional shift register

Universal shift register:

A register is capable of shifting in one direction only is a unidirectional shift register. One that can shift 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 registers. Universal shift register is a bidirectional register, whose input can be either in serial form or in parallel form and whose output also can be in serial form or I parallel form.

The most general shift register has the following capabilities.

1. A clear control to clear the register to 0
2. A clock input to synchronize the operations
3. A shift-right control to enable the shift-right operation and serial input and output lines associated with the shift-right

4. A shift-left control to enable the shift-left operation and serial input and output lines associated with the shift-left
5. A parallel loads 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 the presence of the clock.

A universal shift register can be realized using multiplexers. The below fig shows the logic diagram of a 4-bit universal shift register that has all capabilities. It consists of 4 D flip-flops and four multiplexers. The four multiplexers have two common selection inputs s1 and s0. Input 0 in each multiplexer is selected when S1S0=00, input 1 is selected when S1S0=01 and input 2 is selected when S1S0=10 and input 4 is selected when S1S0=11. The selection inputs control the mode of operation of the register according to the functions entries. When S1S0=0, the present value of the register is applied to the D inputs of flip-flops. The condition forms a path from the output of each flip-flop into the input of the same flip-flop. The next clock edge transfers into each flip-flop the binary value it held previously, and no change of state occurs. When S1S0=01, terminal 1 of the multiplexer inputs have a path to the D inputs of the flip-flop. This causes a shift-right operation, with serial input transferred into flip-flopA4. When S1S0=10, a shift left operation results with the other serial input going into flip-flop A1. Finally when S1S0=11, the binary information on the parallel input lines is transferred into the register simultaneously during the next clock cycle

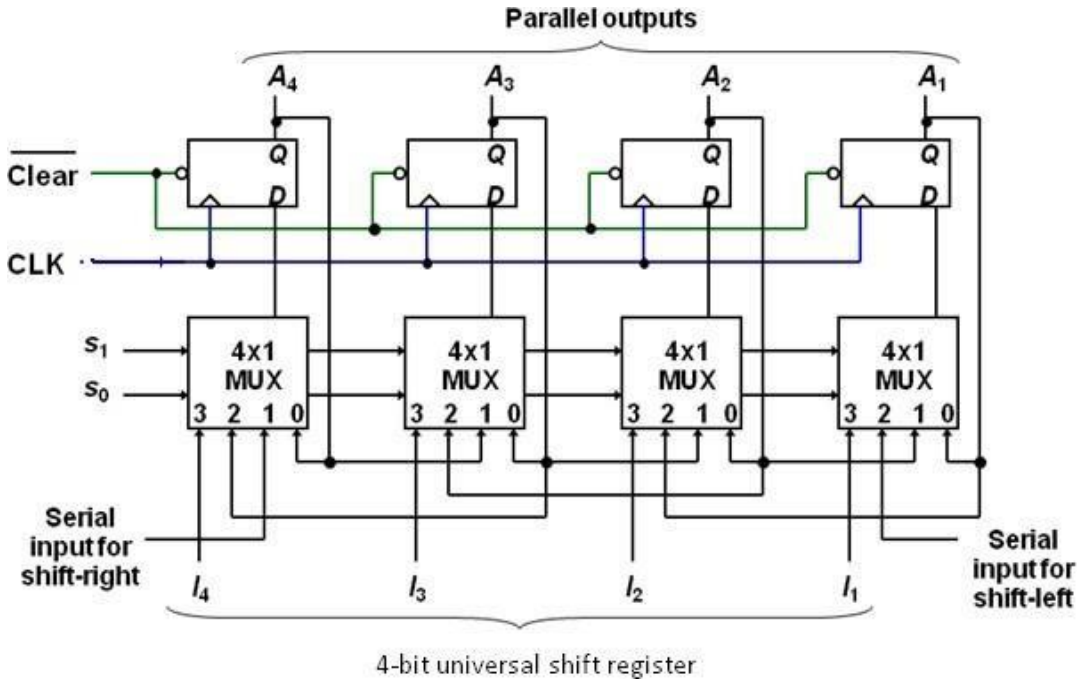


Figure: logic diagram 4-bit universal shift register

Function table for theregister

mode control		
S0	**S1**	**register operation**
0	0	No change
0	1	Shift Right
1	0	Shift left
1	1	Parallel load

Counters:

Counter is a device which stores (and sometimes displays) the number of times particular event or process has occurred, often in relationship to a clock signal. A Digital counter is a set of flip flops whose state change in response to pulses applied at the input to the counter. Counters may be asynchronous counters or synchronous counters. Asynchronous counters are also called ripple counters

In electronics counters can be implemented quite easily using register-type circuits such as the flip-flops and a wide variety of classifications exist:

- Asynchronous (ripple) counter – changing state bits are used as clocks to subsequent state flip-flops
- Synchronous counter – all state bits change under control of a singleclock
- Decade counter – counts through ten states per stage
- Up/down counter – counts both up and down, under command of a control input
- Ring counter – formed by a shift register with feedback connection in a ring
- Johnson counter – a *twisted* ring counter
- Cascaded counter
- Modulus counter.

Each is useful for different applications. Usually, counter circuits are digital in nature, and count in natural binary Many types of counter circuits are available as digital building blocks, for example a number of chips in the 4000 series implement different counters.

Occasionally there are advantages to using a counting sequence other than the natural binary sequence such as the binary coded decimal counter, a linear feed-back shift register counter, or a gray-code counter.

Counters are useful for digital clocks and timers, and in oven timers, VCR clocks, etc.

Asynchronous counters:

An asynchronous (ripple) counter is a single JK-type flip-flop, with its J (data) input fed from its own inverted output. This circuit can store one bit, and hence can count from zero to one before it overflows (starts over from 0). This counter will increment once for every clock cycle and takes two clock cycles to overflow, so every cycle it will alternate between a transition from 0 to 1 and a transition from 1 to 0. Notice that this creates a new clock with a 50% duty cycle at exactly half the frequency of the input clock. If this output is then used as the clock signal for a similarly arranged D flip-flop (remembering to invert the output to the input), one will get another 1 bit counter that counts half as fast. Putting them together yields a two-bit counter:

Two-bit ripple up-counter using negative edge triggered flip flop:

Two bit ripple counter used two flip-flops. There are four possible states from 2 – bit up-counting I.e. 00, 01, 10 and 11.

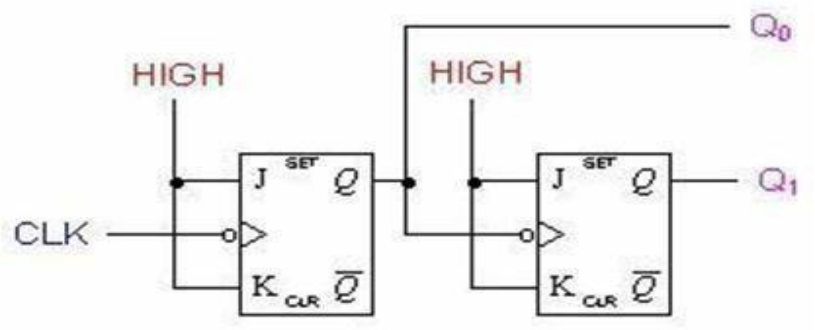
· The counter is initially assumed to be at a state 00 where the outputs of the tow flip-flops are noted as Q_1Q_0. Where Q_1 forms the MSB and Q_0 forms the LSB.

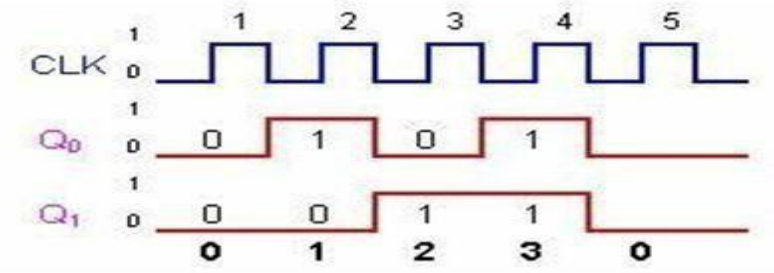
· For the negative edge of the first clock pulse, output of the first flip-flop FF_1 toggles its state. Thus Q_1 remains at 0 and Q_0 toggles to 1 and the counter state are now read as 01.

· During the next negative edge of the input clock pulse FF_1 toggles and $Q_0 = 0$. The output Q0 being a clock signal for the second flip-flop FF_2 and the present transition acts as a negative edge for FF_2 thus toggles its state $Q_1 = 1$. The counter state is now read as 10.

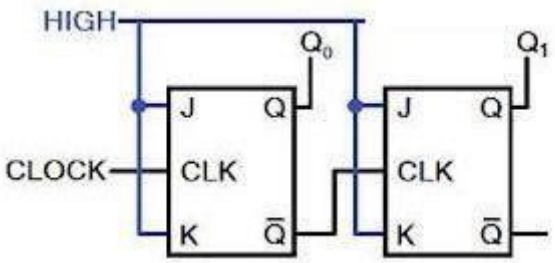
· For the next negative edge of the input clock to FF_1 output Q0 toggles to 1. But this transition from 0 to 1 being a positive edge for FF_2 output Q_1 remains at 1. The counter state is now read as 11.

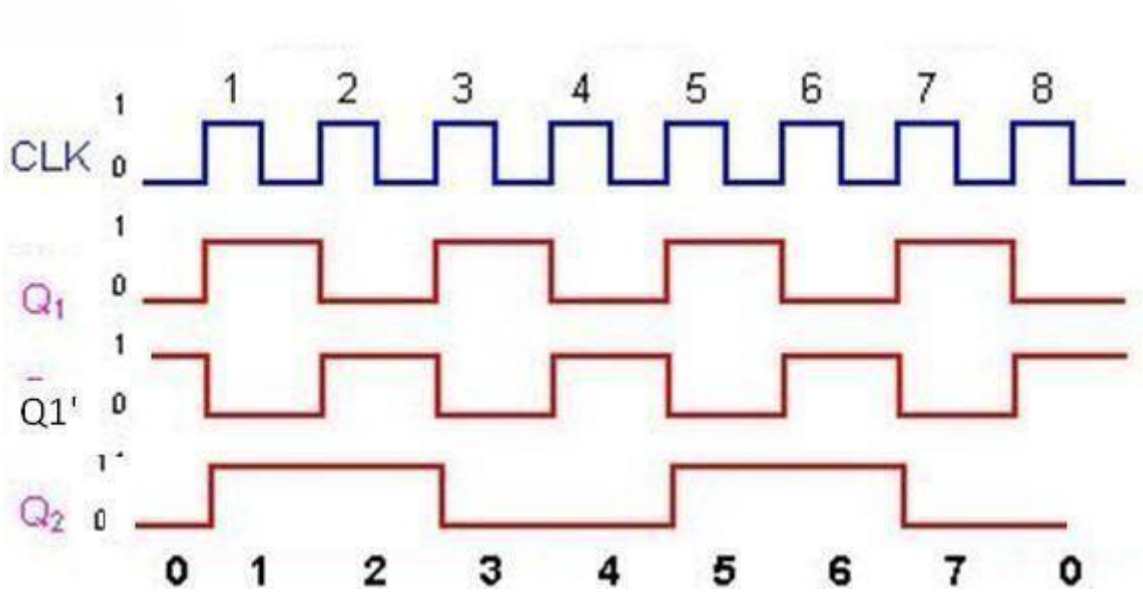
· For the next negative edge of the input clock, Q_0 toggles to 0. This transition from 1 to 0 acts as a negative edge clock for FF_2 and its output Q_1 toggles to 0. Thus the starting state 00 is attained. Figure shown below





Two-bit ripple down-counter using negative edge triggered flip flop:





A 2-bit down-counter counts in the order 0,3,2,1,0,1……..,i.e, 00,11,10,01,00,11 …..,etc. the above fig. shows ripple down counter, using negative edge triggered J-K FFs and its timing diagram.

- For down counting, Q1‘ of FF1 is connected to the clock of Ff2. Let initially all the FF1 toggles, so, Q1 goes from a 0 to a 1 and Q1‘ goes from a 1 to a 0.

- The negative-going signal at Q1‘ is applied to the clock input of FF2, toggles Ff2 and, therefore, Q2 goes from a 0 to a 1.so, after one clock pulse Q2=1 and Q1=1, I.e., the state of the counter is 11.
- At the negative-going edge of the second clock pulse, Q1 changes from a 1 to a 0 and Q1‘ from a 0 to a 1.
- This positive-going signal at Q1‘ does not affect FF2 and, therefore, Q2 remains at a 1. Hence , the state of the counter after second clock pulse is 10
- At the negative going edge of the third clock pulse, FF1 toggles. So Q1, goes from a 0 to a 1 and Q1‘ from 1 to 0. This negative going signal at Q1‘ toggles FF2 and, so, Q2 changes from 1 to 0, hence, the state of the counter after the third clock pulse is 01.
- At the negative going edge of the fourth clock pulse, FF1 toggles. So Q1, goes from a 1 to a 0 and Q1‘ from 0 to 1. . This positive going signal at Q1‘ does not affect FF2 and, so, Q2 remains at 0, hence, the state of the counter after the fourth clock pulse is 00.

Two-bit ripple up-down counter using negative edge triggered flip flop:

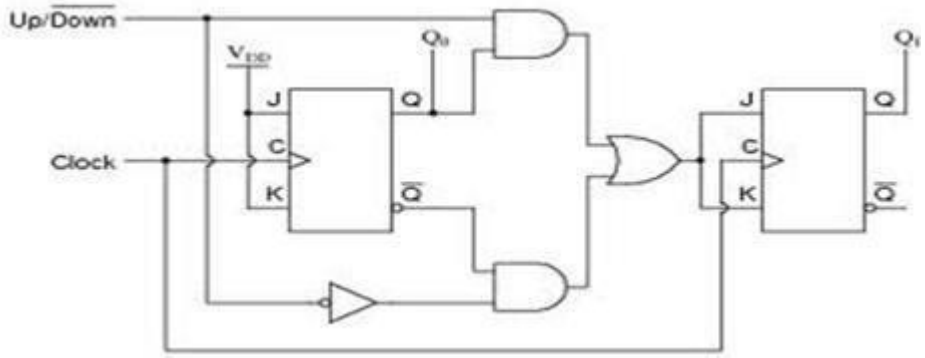


Figure: asynchronous 2-bit ripple up-down counter using negative edge triggered flip flop:

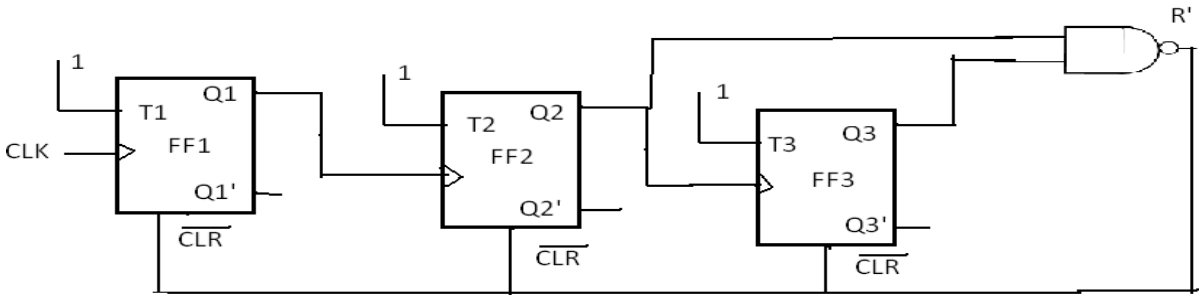
- As the name indicates an up-down counter is a counter which can count both in upward and downward directions. An up-down counter is also called a forward/backward counter or a bidirectional counter. So, a control signal or a mode signal M is required to choose the direction of count. When M=1 for up counting, Q1 is transmitted to clock of FF2 and when M=0 for down counting, Q1‘ is transmitted to clock of FF2. This is achieved by using two AND gates and one OR gates. The external clock signal is applied to FF1.
- Clock signal to FF2= (Q1.Up)+(Q1‘. Down)= Q1m+Q1‘M‘

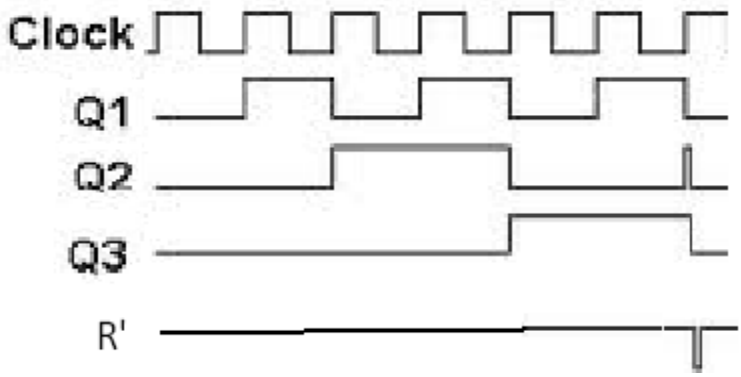
Design of Asynchronous counters:

To design a asynchronous counter, first we write the sequence , then tabulate the values of reset signal R for various states of the counter and obtain the minimal expression for R and R‘ using K-Map or any other method. Provide a feedback such that R and R‘ resets all the FF‘s after the desired count

Design of a Mod-6 asynchronous counter using T FFs:

A mod-6 counter has six stable states 000, 001, 010, 011, 100, and 101. When the sixth clock pulse is applied, the counter temporarily goes to 110 state, but immediately resets to 000 because of the feedback provided. it is –divide by-6-counter‖, in the sense that it divides the input clock frequency by 6.it requires three FFs, because the smallest value of n satisfying the condition$N \leq 2^n$ is n=3; three FFs can have 8 possible states, out of which only six are utilized and the remaining two states 110and 111, are invalid. If initially the counter is in 000 state, then after the sixth clock pulse, it goes to 001, after the second clock pulse, it goes to 010, and so on.





After sixth clock pulse it goes to 000. For the design, write the truth table with present state outputs Q3, Q2 and Q1 as the variables, and reset R as the output and obtain an expression for R in terms of Q3, Q2, and Q1that decides the feedback into be provided. From the truth table, R=Q3Q2. For active-low Reset, R‘ is used. The reset pulse is of very short duration, of the order of nanoseconds and it is equal to the propagation delay time of the NAND gate used. The expression for R can also be determined as follows.

R=0 for 000 to 101, R=1 for 110, and R=X=for111

Therefore,

R=Q3Q2Q1‘+Q3Q2Q1=Q3Q2

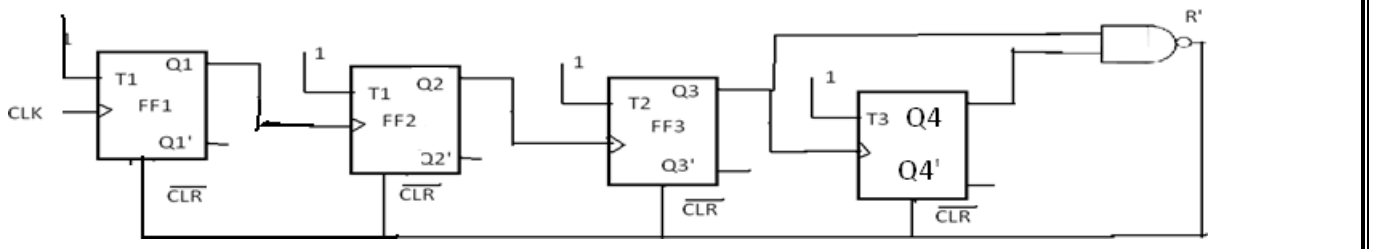
The logic diagram and timing diagram of Mod-6 counter is shown in the above fig.

The truth table is as shown in below.

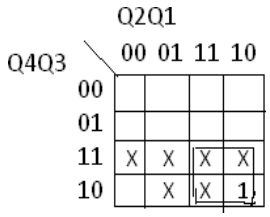
After pulses	States Q3	Q2	Q1	R
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
	↓	↓	↓	
	0	0	0	0
7	0	0	0	0

Design of a mod-10 asynchronous counter using T-flip-flops:

A mod-10 counter is a decade counter. It also called a BCD counter or a divide-by-10 counter. It requires four flip-flops (condition $10 \leq 2^n$ is n=4). So, there are 16 possible states, out of which ten are valid and remaining six are invalid. The counter has ten stable state, 0000 through 1001, i.e., it counts from 0 to 9. The initial state is 0000 and after nine clock pulses it goes to 1001. When the tenth clock pulse is applied, the counter goes to state 1010 temporarily, but because of the feedback provided, it resets to initial state 0000. So, there will be a glitch in the waveform of Q2. The state 1010 is a temporary state for which the reset signal R=1, R=0 for 0000 to 1001, and R=C for 1011 to 1111.



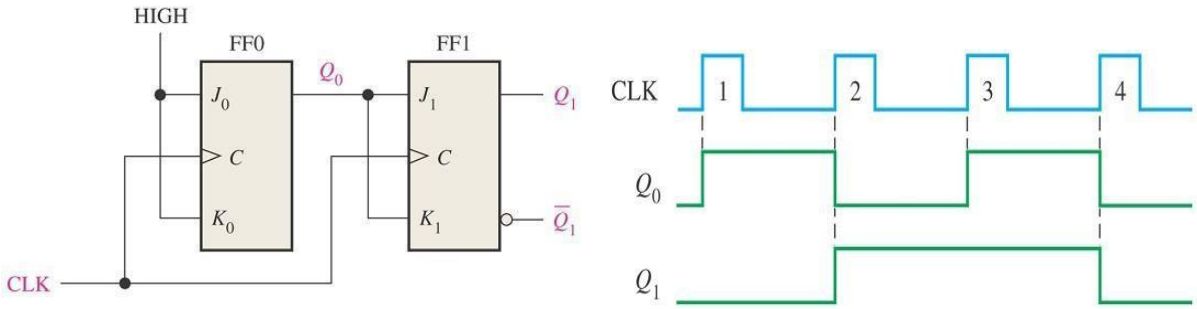
The count table and the K-Map for reset are shown in fig. from the K-Map R=Q4Q2. So, feedback is provided from second and fourth FFs. For active –HIGH reset, Q4Q2 is applied to the clear terminal. For active-LOW reset 4 2 is connected isof allFlip=flops.



After pulses	Count Q4	Q3	Q2	Q1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	0	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	0	1	0	1
10	0	0	0	0

Synchronous counters:

Asynchronous counters are serial counters. They are slow because each FF can change state only if all the preceding FFs have changed their state. if the clock frequency is very high, the asynchronous counter may skip some of the states. This problem is overcome in synchronous counters or parallel counters. Synchronous counters are counters in which all the flip flops are triggered simultaneously by the clock pulses Synchronous counters have a common clock pulse applied simultaneously to all flip-flops. A 2-Bit Synchronous Binary Counter



Design of synchronous counters:

For a systematic design of synchronous counters. The following procedure is used.

Step 1: State Diagram: draw the state diagram showing all the possible states state diagram which also be called nth transition diagrams, is a graphical means of depicting the sequence of states through which the counter progresses.

Step2: number of flip-flops: based on the description of the problem, determine the required number n of the flip-flops- the smallest value of n is such that the number of states $N \leq 2^n$--- and the desired counting sequence.

Step3: choice of flip-flops excitation table: select the type of flip-flop to be used and write the excitation table. An excitation table is a table that lists the present state (ps) , the next state(ns) and required excitations.

Step4: minimal expressions for excitations: obtain the minimal expressions for the excitations of the FF using K-maps drawn for the excitation of the flip-flops in terms of the present states and inputs.

Step5: logic diagram: draw a logic diagram based on the minimal expressions

Design of a synchronous 3-bit up-down counter using JK flip-flops:

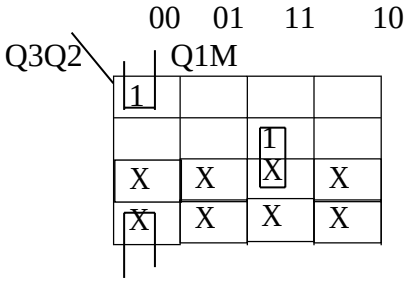
Step1: determine the number of flip-flops required. A 3-bit counter requires three FFs. It has 8 states (000,001,010,011,101,110,111) and all the states are valid. Hence no don‘t cares. For selecting up and down modes, a control or mode signal M is required. When the mode signal M=1 and counts down when M=0. The clock signal is applied to all the FFs simultaneously.

Step2: draw the state diagrams: the state diagram of the 3-bit up-down counter is drawn as

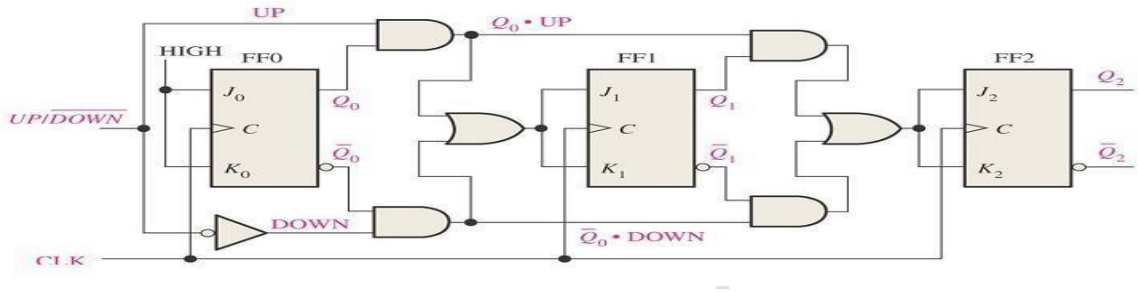
Step3: select the type of flip flop and draw the excitation table: JK flip-flops are selected and the excitation table of a 3-bit up-down counter using JK flip-flops is drawn as shown in fig.

PS			mode	NS			required excitations					
Q3	Q2	Q1	M	Q3	Q2	Q1	J3	K3	J2	K2	J1	K1
0	0	0	0	1	1	1	1	x	1	x	1	x
0	0	0	1	0	0	1	0	x	0	x	1	x
0	0	1	0	0	0	0	0	x	0	x	x	1
0	0	1	1	0	1	0	0	x	1	x	x	1
0	1	0	0	0	0	1	0	x	x	1	1	x
0	1	0	1	0	1	1	0	x	x	0	1	x
0	1	1	0	0	1	0	0	x	x	0	x	1
0	1	1	1	1	0	0	1	x	x	1	x	1
1	0	0	0	0	1	1	x	1	1	x	1	x
1	0	0	1	1	0	1	x	0	0	x	1	x
1	0	1	0	1	0	0	x	0	0	x	x	1
1	0	1	1	1	1	0	x	0	1	x	x	1
1	1	0	0	1	0	1	x	0	x	1	1	x
1	1	0	1	1	1	1	x	0	x	0	1	x
1	1	1	0	1	1	0	x	0	x	0	x	1
1	1	1	1	0	0	0	x	1	x	1	x	1

Step4: obtain the minimal expressions: From the excitation table we can conclude that J1=1 and K1=1, because all the entries for J1and K1 are either X or 1. The K-maps for J3, K3,J2 and K2 based on the excitation table and the minimal expression obtained from them are shown in fig.



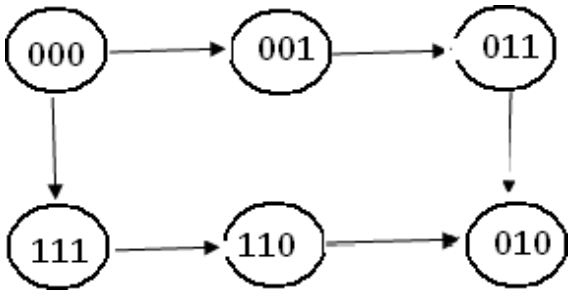
Step5: draw the logic diagram: a logic diagram using those minimal expressions can be drawn as shown in fig.



Design of a synchronous modulo-6 gray cod counter:

Step 1: the number of flip-flops: we know that the counting sequence for a modulo-6 gray code counter is 000, 001, 011, 010, 110, and 111. It requires n=3FFs ($N \leq 2^n$, i.e., $6 \leq 2^3$). 3 FFs can have 8 states. So the remaining two states 101 and 100 are invalid. The entries for excitation corresponding to invalid states are don‘t cares.

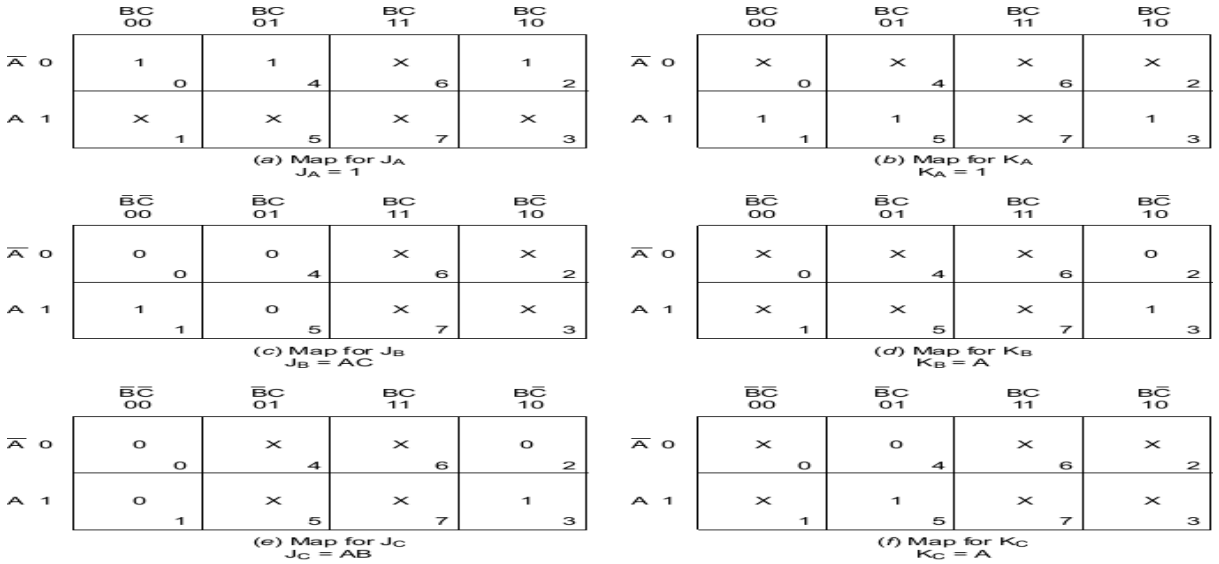
Step2: the state diagram: the state diagram of the mod-6 gray code converter is drawn as shown in fig.



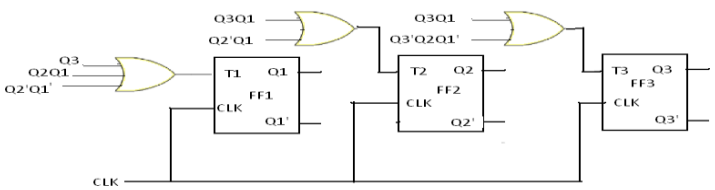
Step3: type of flip-flop and the excitation table: T flip-flops are selected and the excitation table of the mod-6 gray code counter using T-flip-flops is written as shown in fig.

PS			**NS**			**required excitations**		
Q3	**Q2**	**Q1**	**Q3**	**Q2**	**Q1**	**T3**	**T2**	**T1**
0	0	0	0	0	1	0	0	1
0	0	1	0	1	1	0	1	0
0	1	1	0	1	0	0	0	1
0	1	0	1	1	0	1	0	0
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

Step4: The minimal expressions: the K-maps for excitations of FFs T3,T2,and T1 in terms of outputs of FFs Q3,Q2, and Q1, their minimization and the minimal expressions for excitations obtained from them are shown if fig



Step5: the logic diagram: the logic diagram based on those minimal expressions is drawn as shown in fig.



Design of a synchronous BCD Up-Down counter using FFs:

Step1: the number of flip-flops: a BCD counter is a mod-10 counter has 10 states (0000 through 1001) and so it requires n=4FFs($N \leq 2^n$,, i.e., $10 \leq 2^4$). 4 FFS can have 16 states. So out of 16 states, six states (1010 through 1111) are invalid. For selecting up and down mode, a control or mode signal M is required. , it counts up when M=1 and counts down when M=0. The clock signal is applied to all FFs.

Step2: the state diagram: The state diagram of the mod-10 up-down counter is drawn as shown in fig.

Step3: types of flip-flops and excitation table: T flip-flops are selected and the excitation table of the modulo-10 up down counter using T flip-flops is drawn as shown in fig.

The remaining minterms are don't cares($\sum$d(20,21,22,23,24,25,26,37,28,29,30,31)) from the excitation table we can see that T1=1 and the expression for T4,T3,T2 are asfollows.

T4=$\sum$m(0,15,16,19)+d(20,21,22,23,24,25,26,27,28,29,30,31)

T3=$\sum$m(7,15,16,8)+d(20,21,22,23,24,25,26,27,28,29,30,31)

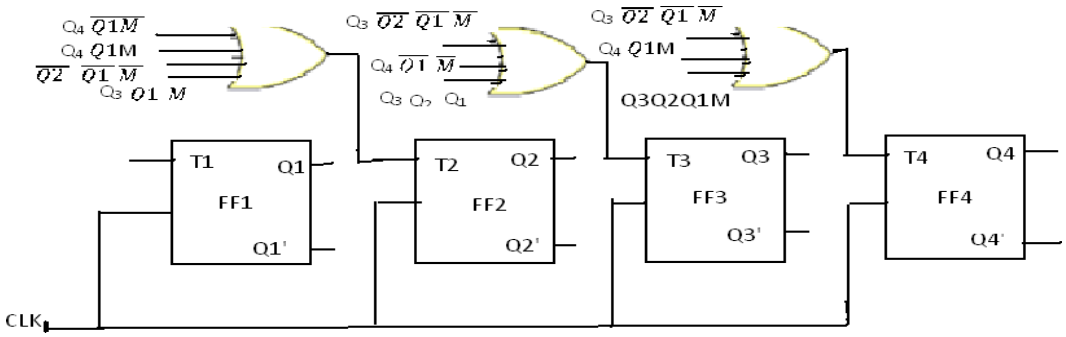
T2=$\sum$m(3,4,7,8,11,12,15,16)+d(20,21,22,23,24,25,26,27,28,29,30,31)

PS				**mode**	**NS**				**required excitations**			
Q4	**Q3**	**Q2**	**Q1**	**M**	**Q4**	**Q3**	**Q2**	**Q1**	**T4**	**T3**	**T2**	**T1**
0	0	0	0	0	1	0	0	1	1	0	0	1
0	0	0	0	1	0	0	0	1	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	0	0	0	1	1
0	0	1	0	0	0	0	0	1	0	0	1	1
0	0	1	0	1	0	0	1	1	0	0	0	1
0	0	1	1	0	0	0	1	0	0	0	0	1
0	0	1	1	1	0	1	0	0	0	1	1	1
0	1	0	0	0	0	0	1	1	0	1	1	1
0	1	0	0	1	0	1	0	1	0	0	0	1
0	1	0	1	0	0	1	0	0	0	0	0	1
0	1	0	1	1	0	1	1	0	0	0	1	1
0	1	1	0	0	0	1	0	1	0	0	1	1
0	1	1	0	1	0	1	1	1	0	0	0	1
0	1	1	1	0	0	1	1	0	0	0	0	1
0	1	1	1	1	1	0	0	0	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	1	1	0	0	1	0	0	0	1
1	0	0	1	0	1	0	0	0	0	0	0	1
1	0	0	1	1	0	0	0	0	1	0	0	1

Step4: The minimal expression: since there are 4 state variables and a mode signal, we require 5 variable kmaps. 20 conditions of Q4Q3Q2Q1M are valid and the remaining 12 combinations are invalid. So the entries for excitations corresponding to those invalid combinations are don‘t cares. Minimizing K-maps for T2 we get

T 2= Q4Q1‘M+Q4‘Q1M+Q2Q1‘M‘+Q3Q1‘M‘

Step5: the logic diagram: the logic diagram based on the above equation is shown in fig.



Shift register counters:

One of the applications of shift register is that they can be arranged to form several types of counters. The most widely used shift register counter is ring counter as well as the twisted ring counter.

Ring counter: this is the simplest shift register counter. The basic ring counter using D flip-flops is shown in fig. the realization of this counter using JK FFs. The Q output of each stage is connected to the D flip-flop connected back to the ring counter.

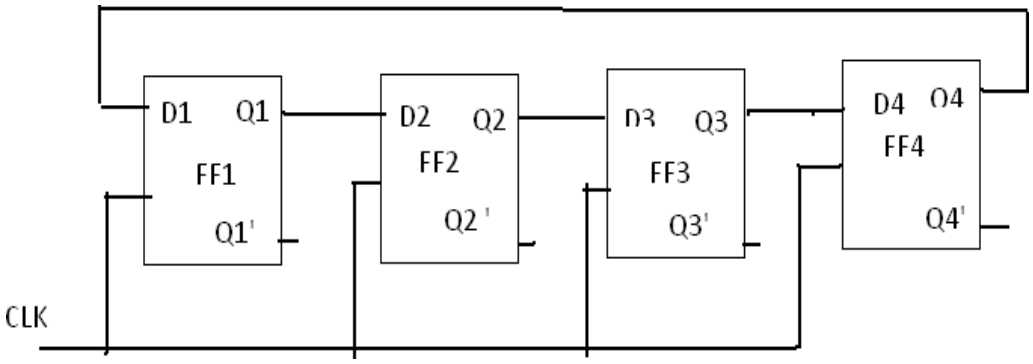
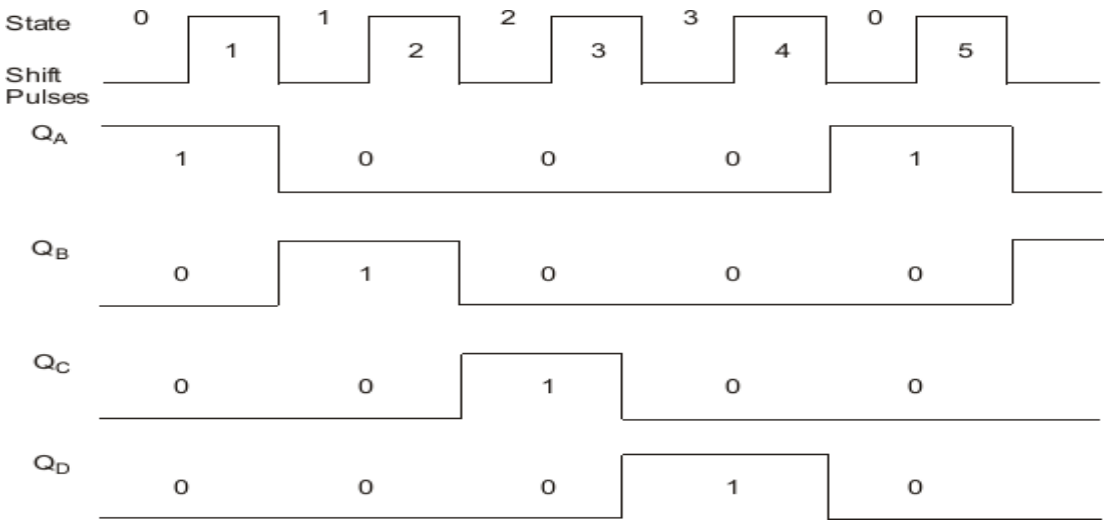


FIGURE: logic diagram of 4-bit ring counter using D flip-flops

Only a single 1 is in the register and is made to circulate around the register as long as clock pulses are applied. Initially the first FF is present to a 1. So, the initial state is 1000, i.e., Q1=1, Q2=0,Q3=0,Q4=0. After each clock pulse, the contents of the register are shifted to the right by one bit and Q4 is shifted back to Q1. The sequence repeats after four clock pulses. The number

of distinct states in the ring counter, i.e., the mod of the ring counter is equal to number of FFs used in the counter. An n-bit ring counter can count only n bits, where as n-bit ripple counter can count 2^n bits. So, the ring counter is uneconomical compared to a ripple counter but has advantage of requiring no decoder, since we can read the count by simply noting which FF is set. Since it is entirely a synchronous operation and requires no gates external FFs, it has the further advantage of being very fast.

Timing diagram:



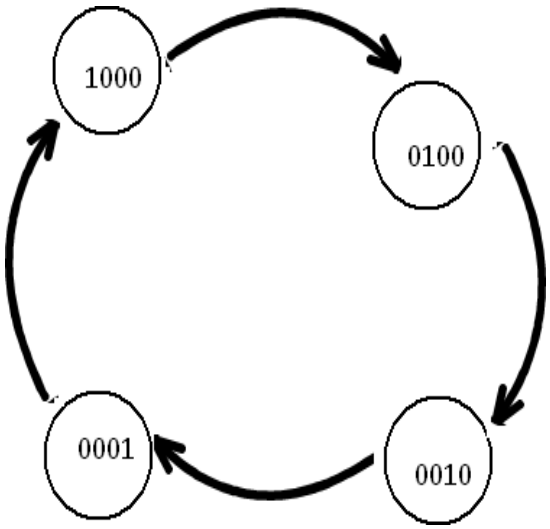


Figure: state diagram

Twisted Ring counter (Johnson counter):

This counter is obtained from a serial-in, serial-out shift register by providing feedback from the inverted output of the last FF to the D input of the first FF. the Q output of each is connected to the D input of the next stage, but the Q‘ output of the last stage is connected to the D input of the first stage, therefore, the name twisted ring counter. This feedback arrangement produces a unique sequence of states.

The logic diagram of a 4-bit Johnson counter using D FF is shown in fig. the realization of the same using J-K FFs is shown in fig.. The state diagram and the sequence table are shown in figure. The timing diagram of a Johnson counter is shown in figure.

Let initially all the FFs be reset, i.e., the state of the counter be 0000. After each clock pulse, the level of Q1 is shifted to Q2, the level of Q2to Q3, Q3 to Q4 and the level of Q4‘to Q1 and the sequences given in fig.

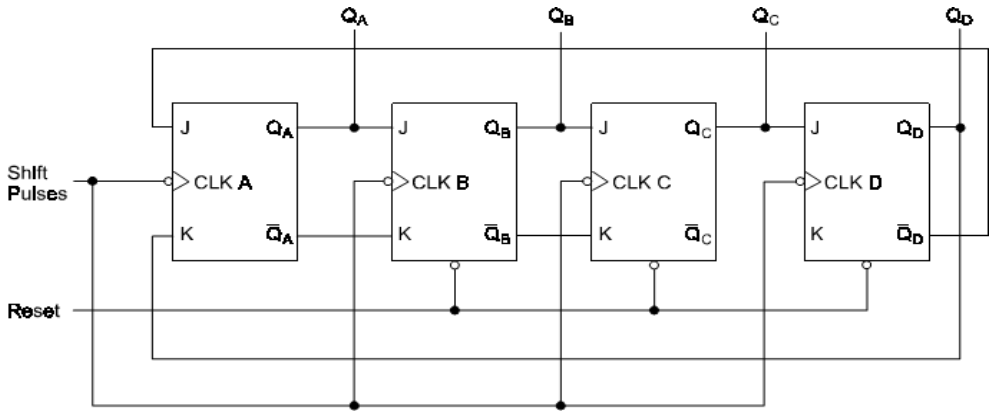


Figure: Johnson counter with JK flip-flops

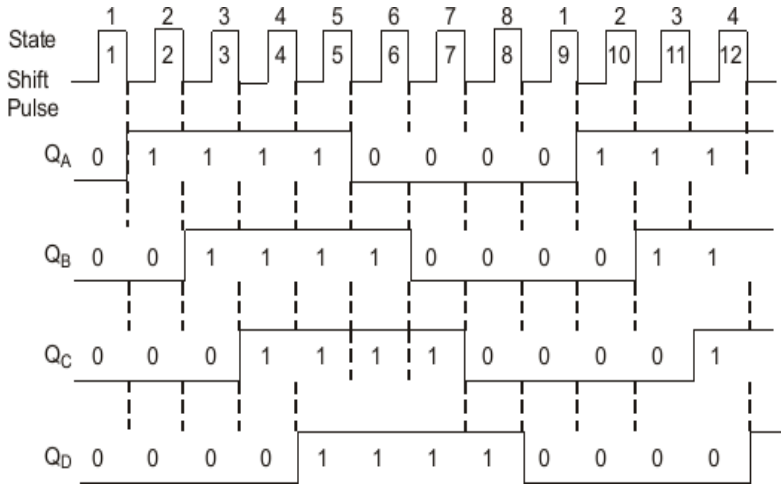
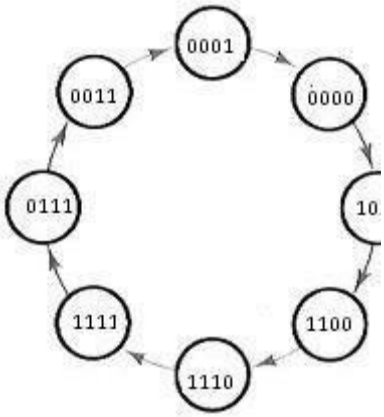


Figure: timing diagram

State diagram:



Q1	Q2	Q3	Q4	after clock pulse
0	0	0	0	0
1	0	0	0	1
1	1	0	0	2
1	1	1	0	3
1	1	1	1	4
0	1	1	1	5
0	0	1	1	6
0	0	0	1	7
0	0	0	0	8
1	0	0	0	9

Excitation table

Synthesis of sequential circuits:

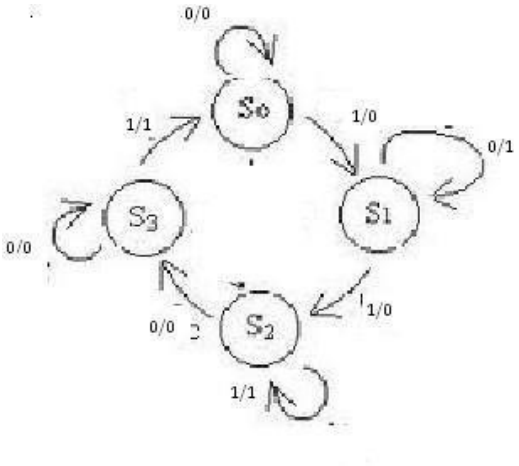
The synchronous or clocked sequential circuits are represented by two models.

1. Moore circuit: in this model, the output depends only on the present state of the flip-flops
2. Meelay circuit: in this model, the output depends on both present state of the flip-flop. And the inputs.

Sequential circuits are also called finite state machines (FSMs). This name is due to the fast that the functional behavior of these circuits can be represented using a finite number of states.

State diagram: the state diagram or state graph is a pictorial representation of the relationships between the present state, the input, the next state, and the output of a sequential circuit. The state diagram is a pictorial representation of the behavior of a sequentialcircuit.

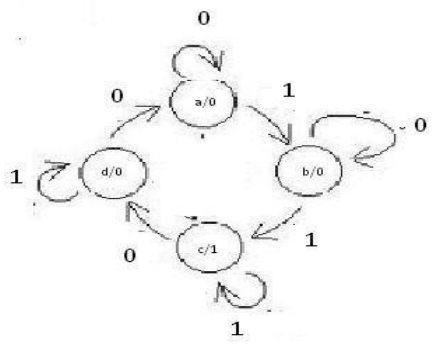
The state represented by a circle also called the node or vertex and the transition between states is indicated by directed lines connecting circle. a directed line connecting a circle with itself indicates that the next state is the same as the present state. The binary number inside each circle identifies the state represented by the circle. The direct lines are labeled with two binary numbers separated by a symbol. The input value is applied during the present state is labeled after the symbol.



	NS,O/P INPUT X	
PS	X=0	X=1
a	a,0	b,0
b	b,1	c,0
c	d,0	c,1
d	d,0	a,1

Fig :a) state diagram (meelay circuit) fig: b) state table

In case of moore circuit ,the directed lines are labeled with only one binary number representing the input that causes the state transition. The output is indicated with in the circle below the present state, because the output depends only on the present state and not on the input.



	NS INPUT X		
PS	X=0	X=1	O/P
a	a	b	0
b	b	c	0
c	d	c	1
d	a	d	0

Fig: a) state diagram (moore circuit) fig:b) state table

Serial binary adder:

Step1: word statement of the problem: the block diagram of a serial binary adder is shown in fig. it is a synchronous circuit with two input terminals designated X1and X2 which carry the two binary numbers to be added and one output terminal Z which represents the sum. The inputs and outputs consist of fixed-length sequences 0s and 1s.the output of the serial Z_i at time $t_i$is a function of the inputs $X_1(t_i)$ and $X_2(t_i)$ at that time t_{i-1} and of carry which had been generated at t_{i-1}. The carry which represent the past history of the serial adder may be a 0 or 1. The circuit has two states. If one state indicates that carry from the previous addition is a 0, the other state indicates that the carry from the previous addition is a 1

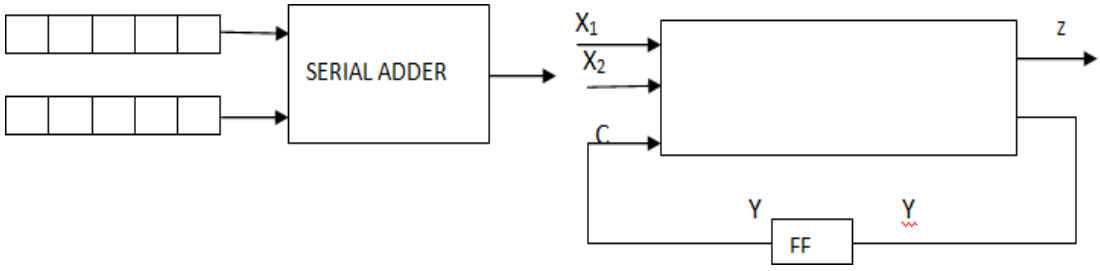
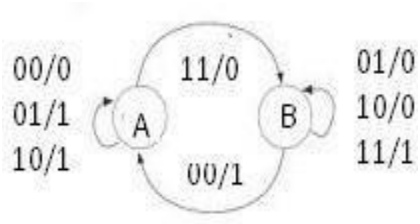


Figure: block diagram of serial binary adder

Step2 and 3: state diagram and state table: let a designate the state of the serial adder at t_i if a carry 0 was generated at t_{i-1}, and let b designate the state of the serial adder at t_i if carry 1 was generated at t_{i-1} .the state of the adder at that time when the present inputs are applied is referred to as the present state(PS) and the state to which the adder goes as a result of the new carry value is referred to as next state(NS).

The behavior of serial adder may be described by the state diagram and state table.



PS	NS ,O/P			
	X1 X2			
	0	0	1	1
	0	1	0	1
A	A,0	B,0	B,1	B,0
B	A,1	B,0	B,0	B,1

Figures: serial adder state diagram and state table

If the machine is in state B, i.e., carry from the previous addition is a 1, inputs X_1=0 and X_2=1 gives sum, 0 and carry 1. So the machine remains in state B and outputs a 0. Inputs X_1=1 and X_2=0 gives sum, 0 and carry 1. So the machine remains in state B and outputs a 0. Inputs X_1=1 and X_2=1 gives sum, 1 and carry 0. So the machine remains in state B and outputs a 1. Inputs X_1=0 and X_2=0 gives sum, 1 and carry 0. So the machine goes to state A and outputs a 1. The state table also gives the same information.

Setp4: reduced standard from state table: the machine is already in this form. So no need to do anything

Step5: state assignment and transition and output table:

The states, A=0 and B=1 have already been assigned. So, the transition and output table is as shown.

PS	NS				O/P			
	0	0	1	1	0	0	1	1
	0	1	0	1	0	1	0	1
0	0	0	0	1	0	1	1	1
1	0	1	1	1	1	0	0	1

STEP6: choose type of FF and excitation table: to write table, select the memory element the excitation table is as shown in fig.

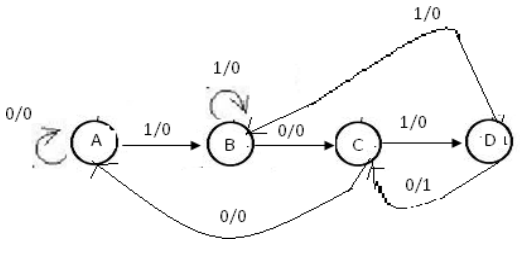
PS	I/P		NS	I/P-FF	O/P
y	x1	x2	Y	D	Z
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	1

Sequence detector:

Step1: word statement of the problem: a sequence detector is a sequential machine which produces an output 1 every time the desired sequence is detected and an output 0 at all other times

Suppose we want to design a sequence detector to detect the sequence 1010 and say that overlapping is permitted i.e., for example, if the input sequence is 01101010 the corresponding output sequence is 00000101.

Step2 and 3: state diagram and state table: the state diagram and the state table of the sequence detector. At the time t_1, the machine is assumed to be in the initial state designed arbitrarily as A. while in this state, the machine can receive first bit input, either a 0 o r a 1. If the input bit is 0, the machine does not start the detection process because the first bit in the desired sequence is a 1. If the input bit is a 1 the detection process starts.



PS	NS,Z	
	X=0	X=1
A	A,0	B,0
B	C,0	B,0
C	A,0	D,0
D	C,1	B,0

Figure: state diagram and state table of sequence detector

So, the machine goes to state B and outputs a 0. While in state B, the machinery may receive 0 or 1 bit. If the bit is 0, the machine goes to the next state, say state c, because the previous two bits are 10 which are a part of the valid sequence, and outputs 0.. if the bit is a 1, the two bits become11 and this not a part of the valid sequence

Step4: reduced standard form state table: the machine is already in this form. So no need to doanything.

Step5: state assignment and transition and output table: there are four states therefore two states variables are required. Two state variables can have a maximum of four states, so, all states are utilized and thus there are no invalid states. Hence, there are no don‘t cares. Let a=00, B=01, C=10 and D=11 be the state assignment.

	NS(Y1Y2)		O/P(z)	
PS(y1y2	X=0	X=1	X=0	X=1
A= 0 0	0 0	0 1	0	0
B=0 1	1 0	0 1	0	0
C=1 0	0 0	1 1	0	0
D=1 1 1	1 0 1	1	0	

Step6: choose type of flip-flops and form the excitation table: select the D flip-flops as memoryelements and draw the excitation table.

PS		**I/P**	**NS**		**INPUTS - FFS**		**O/P**
y1	Y2	X	Y1	Y2	D1	D2	Z
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	1	0	1	0	0
0	1	1	0	1	0	1	0
1	0	0	0	0	0	0	0
1	0	1	1	1	1	1	0
1	1	0	1	0	1	0	1
1	1	1	0	1	0	1	0

Step7: K-maps and minimal functions: based on the contents of the excitation table , draw the k- map and simplify them to obtain the minimal expressions for D1 and D2 in terms of y1, y2 and x as shown in fig. The expression for z (z=y1,y2) can be obtained directly from table

Step8: implementation: the logic diagram based on these minimal expressions

UNIT-III
Introduction to Verilog HDL

Verilog as HDL

Verilog has a variety of constructs as part of it. All are aimed at providing a functionally tested and a verified design description for the target FPGA or ASIC.

The language has a dual function – one fulfilling the need for a design description and the other fulfilling the need for verifying the design for functionality and timing constraints like propagation delay, critical path delay, slack, setup, and hold times.

Levels of Design Description

The components of the target design can be described at different levels with the help of the constructs in Verilog.

In Verilog HDL a module can be defined using various levels of abstraction. There are four levels of abstraction in verilog.
They are: 1. Circuit Level 2. Gate Level 3. Data Flow Level 4. Behavioral Level

Circuit Level

At the circuit level, a switch is the basic element with which digital circuits are built. Switches can be combined to form inverters and other gates at the next higher level of abstraction. Verilog has the basic MOS switches built into its constructs, which can be used to build basic circuits like inverters, basic logic gates, simple 1-bit dynamic and static memories. They can be used to build up larger designs to simulate at the circuit level, to design performance critical circuits.

The below Figure1 shows the circuit of an inverter suitable for description with the switch level constructs of Verilog.

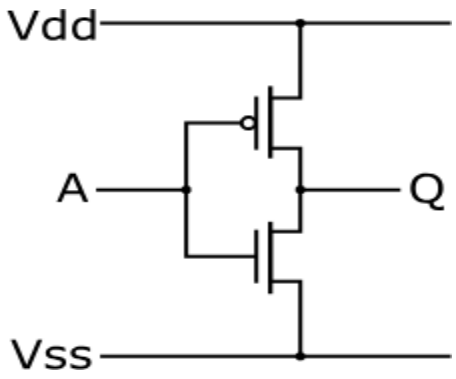


Figure 1 CMOS inverter

Gate Level

At the next higher level of abstraction, design is carried out in terms of basic gates. All the basic gates are available as ready modules called “Primitives.” Each such primitive is defined in terms of its inputs and outputs. Primitives can be incorporated into design descriptions directly. Just as full physical hardware can be built using gates, the primitives can be used repeatedly and judiciously to build larger systems.

Figure 2 shows an AND gate suitable for description using the gate primitive of Verilog.

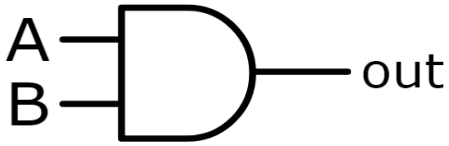


Figure 2 AND gate symbol

The gate level modeling or structural modeling as it is sometimes called is akin to building a digital circuit on a bread board, or on a PCB. One should know the structure of the design to build the model here. One can also build hierarchical circuits at this level. However, beyond 20 to 30 of such gate primitives in a circuit, the design description becomes unwieldy; testing and debugging become laborious.

Data Flow

Data flow is the next higher level of abstraction. All possible operations on signals and variables are represented here in terms of assignments. All logic and algebraic operations are accommodated. The assignments define the continuous functioning of the concerned block. At the data flow level, signals are assigned through the data manipulating equations. All such assignments are concurrent in nature. The design descriptions are more compact than those at the gate level.

Figure 3 shows an A-O-I relationship suitable for description with the Verilog constructs at the data flow level.

$$e = \overline{a.b + c.d}$$

Figure 3 An A-O-I gate represented as a data flow type of relationship.

Behavioral Level

Behavioral level constitutes the highest level of design description; it is essentially at the system level itself. With the assignment possibilities, looping constructs and conditional branching possible, the design description essentially looks like a "C" program.

A module can be implemented in terms of the design algorithm. The designer no need to have any knowledge of hardware implementation.

The statements involved are "dense" in function. Compactness and the comprehensive nature of the design description make the development process fast and efficient.

Figure 4 shows an A-O-I gate expressed in pseudo code suitable for description with the behavioral level constructs of Verilog.

If (a, b, c or d changes)
Compute e as
$$e = \overline{a.b + c.d}$$

Figure . 4 An A-O-I gate in pseudo code at behavioral level.

The Overall Design Structure in Verilog

The possibilities of design description statements and assignments at different levels necessitate their accommodation in a mixed mode. In fact the design statements coexisting in a seamless manner within a design module is a significant characteristic of Verilog. Thus Verilog facilitates the mixing of the above-mentioned levels of design. A design built at data flow level can be instantiated to form a structural mode design. Data flow assignments can be incorporated in designs which are basically at behavioral level.

Concurrency

In an electronic circuit all the units are to be active and functioning concurrently. The voltages and currents in the different elements in the circuit can change simultaneously. In turn the logic levels too can change. Simulation of such a circuit in an HDL calls for concurrency of operation.

A number of activities – may be spread over different modules – are to be run concurrently here. Verilog simulators are built to simulate concurrency. (This is in contrast to programs in the normal languages like C where execution is sequential.)

Concurrency is achieved by proceeding with simulation in equal time steps. The time step is kept small enough to be negligible compared with the propagation delay values. All the activities scheduled at one time step are completed and then the simulator advances to the next time step and so on. The time step values refer to simulation time and not real time. One can redefine timescales to suit technology as and when necessary and carry out test runs.

In some cases the circuit itself may demand sequential operation as with data transfer and memory-based operations. Only in such cases sequential operation is ensured by the appropriate usage of sequential constructs from Verilog HDL.

Simulation and Synthesis

The design that is specified and entered as described earlier is simulated for functionality and fully debugged. Translation of the debugged design into the corresponding hardware circuit (using an FPGA or an ASIC) is called “synthesis.”

The tools available for synthesis relate more easily with the gate level and data flow level modules [Smith MJ]. The circuits realized from them are essentially direct translations of functions into circuit elements.

In contrast many of the behavioral level constructs are not directly synthesizable; even if synthesized they are likely to yield relatively redundant or wrong hardware. The way out is to take the behavioral level modules and redo each of them at lower levels. The process is carried out successively with each of the behavioral level modules until practically the full design is available as a pack of modules at gate and data flow levels (more commonly called the “RTL level”).

Programming Language Interface (PLI)

PLI provides an active interface to a compiled Verilog module. The interface adds a new dimension to working with Verilog routines from a C platform. The key functions of the interface are as follows:

- One can read data from a file and pass it to a Verilog module as input. Such data can be test vectors or other input data to the module. Similarly, variables in Verilog modules can be accessed and their values written to output devices.
- Delay values, logic values, etc., within a module can be accessed and altered.
- Blocks written in C language can be linked to Verilog modules.

MODULE

Any Verilog program begins with a keyword – called a "module." A module is the name given to any system considering it as a black box with input and output terminals as shown in Figure 1. The terminals of the module are referred to as 'ports'. The ports attached to a module can be of three types:

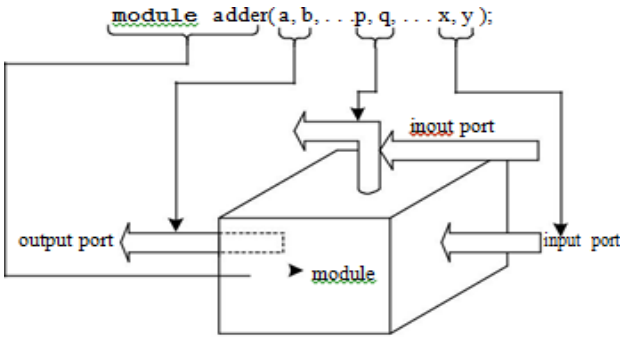


Figure 1 Representation of a module as black box with its ports.

- input ports through which one gets entry into the module; they signify the input signal terminals of the module.
- output ports through which one exits the module; these signify the output signal terminals of the module.
- inout ports: These represent ports through which one gets entry into the module or exits the module; These are terminals through which signals are input to the module sometimes; at some other times signals are output from the module through these.

Whether a module has any of the above ports and how many of each type are present depend solely on the functional nature of the module. Thus one module may not have any port at all; another may have only input ports, while a third may have only output ports, and so on.

All the constructs in Verilog are centered on the module. They define ways of building up, accessing, and using modules. The structure of modules and the mode of invoking them in a design are discussed here.

A module comprises a number of "lexical tokens" arranged according to some predefined order. The possible tokens are of seven categories:

- White spaces
- Comments
- Operators
- Numbers
- Strings
- Identifiers
- Keywords

The rules constraining the tokens and their sequencing will be dealt with as we progress. For the present let us consider modules. In Verilog any program which forms a design description is a "module." Any program written to test a design description is also a "module." The latter are often called as "stimulus modules" or "test benches." A module used to do simulation has the form shown in Figure 2. Verilog takes the active statements appearing between the "module" statement and the "endmodule" statement and interprets all of them together as forming the body of the module. Whenever a module is invoked for testing or for incorporation into a bigger design module, the name of the module ("test" here) is used to identify it for the purpose.

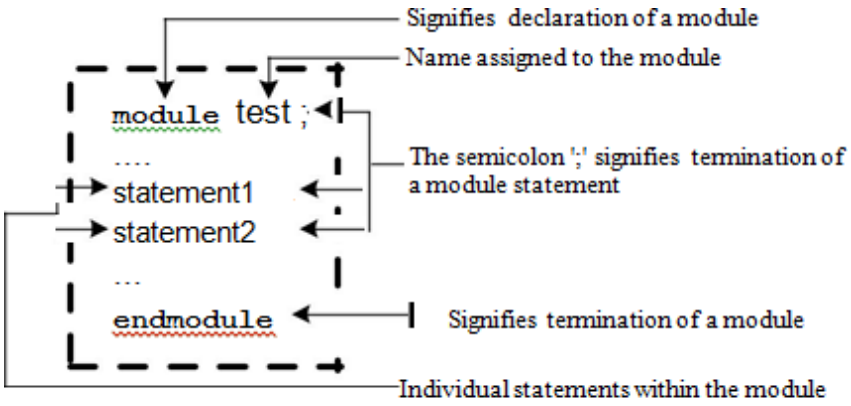


Figure 2 Structure of a typical simulation module.

LANGUAGE CONSTRUCTS AND CONVENTIONS IN VERILOG

Introduction

The constructs and conventions make up a software language. A clear understanding and familiarity of these is essential for the mastery of the language. Verilog has its own constructs and conventions [IEEE, Sutherland]. In many respects they resemble those of C language [Gottfried].

Any source file in Verilog (as with any file in any other programming language) is made up of a number of ASCII characters. The characters are grouped into sets — referred to as "lexical tokens." A lexical token in Verilog can be a single character or a group of characters. Verilog has 7 types of lexical tokens- operators, keywords, identifiers, white spaces, comments, numbers, and strings.

Case Sensitivity

Verilog is a case-sensitive language like C. Thus sense, Sense, SENSE, sENse,… etc., are all related as different entities / quantities in Verilog.

Keywords

The keywords define the language constructs. A keyword signifies an activity to be carried out, initiated, or terminated. As such, a programmer cannot use a keyword for any purpose other than that it is intended for. All keywords in Verilog are in small letters and require to be used as such (since Verilog is a case-sensitive language). All keywords appear in the text in New Courier Bold-type letters.

Examples

module -- signifies the beginning of a module definition.
endmodule -- signifies the end of a module definition.
begin -- signifies the beginning of a block of statements.
end -- signifies the end of a block of statements.
if -- signifies a conditional activity to be checked
while -- signifies a conditional activity to be carried out.

Identifiers

Any program requires blocks of statements, signals, etc., to be identified with an attached nametag. Such nametags are identifiers. It is good practice for us to use identifiers, closely related to the significance of variable, signal, block, etc., concerned. This eases understanding and debugging of any program.
e.g., clock, enable, gate_1, . . .

There are some restrictions in assigning identifier names. All characters of the alphabet or an underscore can be used as the first character. Subsequent characters can be of alphanumeric type, or the underscore (_), or the dollar ($) sign – for example

name, _name. Name, name1, name_$, . . . -- all these are allowed as identifiers

name aa -- not allowed as an identifier because of the blank (“name” and “aa” are interpreted as two different identifiers)

$name -- not allowed as an identifier because of the presence of “$” as the first character.1_name -- not allowed as an identifier, since the numeral “1” is the first character

@name -- not allowed as an identifier because of the presence of the character “@”.
A+b m not allowed as an identifier because of the presence of the character “+”.

White Space Characters

Blanks (\b), tabs (\t), newlines (\n), and formfeed form the white space characters in Verilog. In any design description the white space characters are included to improve readability. Functionally, they separate legal tokens. They are introduced between keywords, keyword and an identifier, between two identifiers, between identifiers and operator symbols, and so on. White space characters have significance only when they appear inside strings.

Comments

Comments can be inserted in the code for readability and documentation. There are two ways to write comments. A one-line comment starts with "//". Verilog skips from that point to the end of line. A multiple-line comment starts with "/*" and ends with "*/". Multiple-line comments cannot be nested. However, one-line comments can be embedded in multiple-line comments.

a = b && c; // This is a one-line comment

/* This is a multiple

linecomment */

/* This is /* an illegal */ comment */

/* This is //a legal comment */

Operators

Operators are of three types: unary, binary, and ternary. Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands.

a = ~ b; // ~ is a unary operator. b is the operand

a = b && c; // && is a binary operator. b and c are operands

a = b ? c : d; // ?: is a ternary operator. b, c and d are operands

Number Specification

There are two types of number specification in Verilog: sized and unsized.

Sized numbers

Sized numbers are represented as <size> '<base format> <number>.

<size> is written only in decimal and specifies the number of bits in the number. Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O). The number is specified as consecutive digits from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification.

4'b1111 // This is a 4-bit binary number
12'habc // This is a 12-bit hexadecimal number
16'd255 // This is a 16-bit decimal number.

Unsized numbers

Numbers that are specified without a <base format> specification are decimal numbers by default. Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).

23456 // This is a 32-bit 'hc3 // This is a 32-bit 'o21 // This is a 32-bit

decimal number by default hexadecimal number octal number

X or Z values

Verilog has two symbols for unknown and high impedance values. These values are very important for modeling real circuits. An unknown value is denoted by an x. A high impedance value is denoted by z.

12'h13x // This is a 12-bit hex number; 4 least significant bits unknown

6'hx // This is a 6-bit hex number

32'bz // This is a 32-bit high impedance number

An x or z sets four bits for a number in the hexadecimal base, three bits for a number in the octal base, and one bit for a number in the binary base. If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z. This makes it easy to assign x or z to whole vector. If the most significant digit is 1, then it is also zero extended.

Negative numbers

Negative numbers can be specified by putting a minus sign before the size for a constant number. Size constants are always positive. It is illegal to have a minus sign between <base format> and <number>. An optional signed specifier can be added for signed arithmetic.

-6'd3 // 8-bit negative number stored as 2's complement of 3 -6'sd3 // Used for performing signed integer math 4'd-2 // Illegal specification

Underscore characters and question marks

An underscore character "_" is allowed anywhere in a number except the first character. Underscore characters are allowed only to improve readability of numbers and are ignored by Verilog.
A question mark "?" is the Verilog HDL alternative for z in the context of numbers.12'b1111_0000_1010 // Use of underline characters for readability

4'b10?? // Equivalent of a 4'b10zz

Strings

A string is a sequence of characters that are enclosed by double quotes. The restriction on a string is that it must be contained on a single line, that is, without a carriage return. It cannot be on multiple lines. Strings are treated as a sequence of one-byte ASCII values.

"Hello Verilog World" // is a

string"a / b" // is a string

Value Set or Logic Values

Verilog supports four values and eight strengths to model the functionality of real hardware. The four value levels are listed in Table below.

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
Z	High impedance, floating state

Strengths

The logic levels are also associated with strengths. In many digital circuits, multiple assignments are often combined to reduce silicon area or to reduce pin-outs. To facilitate this, one can assign strengths to logic levels. Verilog has eight strength levels – four of these are of the driving type, three are of capacitive type and one of the hi-Z type.

In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits. Value levels 0 and 1 can have the strength levels listed in Table below

Strength Level	**Type**	**Degree**
supply	Driving	strongest
strong	Driving	
pull	riving	
large	Storage	↑
weak	Driving	
medium	Storage	
small	Storage	
highz	High Impedance	weakest

If two signals of unequal strengths are driven on a wire, the stronger signal prevails.
For example, if two signals of strength strong1 and weak0 contend, the result is resolved as a strong1. If two signals of equal strengths are driven on a wire, the result is unknown. If two signals of strength strong1 and strong0 conflict, the result is an x. Strength levels are particularly useful for accurate modeling of signal contention, MOS devices, dynamic MOS, and other low-level devices.

Data Types

The data handled in Verilog fall into two categories:

(i) Net data type

(ii) Variable data type

The two types differ in the way they are used as well as with regard to their respective hardware structures. Data type of each variable or signal has to be declared prior to its use. The same is valid within the concerned block or module.

Nets

A net signifies a connection from one circuit unit to another. Such a net carries the value of the signal it is connected to and transmits to the circuit blocks connected to it. If the driving end of a net is left floating, the net goes to the high impedance state. A net can be specified in different ways.

wire: It represents a simple wire doing an interconnection. Only one output is connected to a wire and is driven by that.

tri: It represents a simple signal line as a wire. Unlike the wire, a tri can be driven by more than one signal outputs.

Nets are one-bit values by default unless they are declared explicitly as vectors. The terms wire and net are often used interchangeably.

Variable Data Type

A variable is an abstraction for a storage device. It can be declared through the keyword reg and stores the value of a logic level: 0, 1, x, or z. A net or wire connected to a reg takes on the value stored in the reg and can be used as input to other circuit elements. But the output of a circuit cannot be connected to a reg. The value stored in a reg is changed through a fresh assignment in the program.

time, integer, real, and realtime are the other variable types of data; these are dealt with later.

Time

Verilog simulation is done with respect to simulation time. A special time register data type is used in Verilog to store simulation time. A time variable is declared with the keyword time. The width for time register data types is implementation-specific but is at least 64 bits. The system function $time is invoked to get the current simulation time.

```
time save_sim_time; // Define a time variable save_sim_time initial

save_sim_time = $time; // Save the current simulation time
```

Scalars and Vectors

Entities representing single bits — whether the bit is stored, changed, or transferred — are called "scalars." Often multiple lines carry signals in a cluster – like data bus, address bus, and so on. Similarly, a group of regs stores a value, which may be assigned, changed, and handled together. The collection here is treated as a "vector."
Figure below illustrates the difference between a scalar and a vector. wr and rd are two scalar nets connecting two circuit blocks circuit1 and circuit2. b is a 4-bit-wide vector net connecting the same two blocks. b[0], b[1], b[2], and b[3] are the individual bits of vector b. They are "part vectors."

A vector reg or net is declared at the outset in a Verilog program and hence treated as such. The range of a vector is specified by a set of 2 digits (or expressions evaluating to a digit) with a colon in between the two. The combination is enclosed within square brackets.

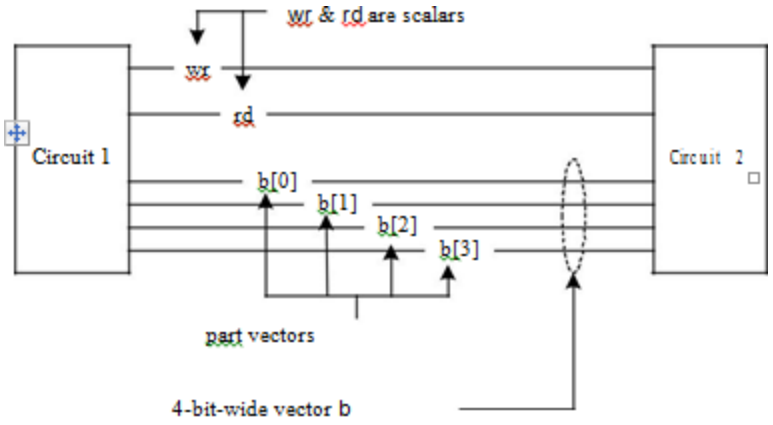


Figure Illustration of scalars and vectors.

Examples:

wire[3:0] a; /* a is a four bit vector of net type; the bits are designated as a[3], a[2], a[1] and a[0]. */

reg[2:0] b; /* b is a three bit vector of reg type; the bits are designated as b[2], b[1] and b[0]. */

reg[4:2] c; /* c is a three bit vector of reg type; the bits are designated as c[4], c[3] and c[2]. */

wire[-2:2] d ; /* d is a 5 bit vector with individual bits designated as d[-2], d[-1], d[0], d[1] and d[2]. */

Whenever a range is not specified for a net or a reg, the same is treated as a scalar – a single bit quantity. In the range specification of a vector the most significant bit and the least significant bit can be assigned specific integer values. These can also be expressions evaluating to integer constants – positive or negative.

Normally vectors – nets or regs – are treated as unsigned quantities. They have to be specifically declared as "signed" if so desired.

Examples

```
wire signed[4:0] num;// num is a vector in the range -16 to +15.

reg signed [3:0] num_1;          // num_1 is a vector in the range -8 to +7.
```

UNIT-IV

Gate Level Modeling

Introduction

Digital designers are normally familiar with all the common logic gates, their symbols, and their working. Flip-flops are built from the logic gates. All other functionally complex and more involved circuits can also be built using the basic gates. All the basic gates are available as "Primitives" in Verilog. Primitives are generalized modules that already exist in Verilog [IEEE]. They can be instantiated directly in other modules.

And Gate Primitive

The AND gate primitive in Verilog is instantiated with the following statement:

and g1 (O, I1, I2, . . ., In);

Here 'and' is the keyword signifying an AND gate. g1 is the name assigned to the specific instantiation. O is the gate output; I1, I2, etc., are the gate inputs. The following are noteworthy:

- The AND module has only one output. The first port in the argument list is the output port.
- An AND gate instantiation can take any number of inputs — the upper limit is compiler-specific.
- A name need not be necessarily assigned to the AND gate instantiation; this is true of all the gate primitives available in Verilog.

Truth Table of AND Gate Primitive

The truth table for a two-input AND gate is shown in Table below It can be directly extended to AND gate instantiations with multiple inputs. The following observations are in order here:

Truth table of AND gate primitive

		Input 1			
		0	1	x	z
Input 2	0	0	0	0	0
	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x

- If any one of the inputs to the AND gate instantiation is in the 0 state, its output is also in the 0 state. It is irrespective of whether the other inputs are at the 0, 1, x or z state.

- The output is at 1 state if and only if every one of the inputs is at 1 state.

- For all other cases the output is at the x state.

- Note that the output is never at the z state – the high impedance state. This is true of all other gate primitives as well.

Module Structure

In a general case a module can be more elaborate. A lot of flexibility is available in the definition of the body of the module. However, a few rules need to be followed:

- The first statement of a module starts with the keyword module; it may be followed by the name of the module and the port list if any.

- All the variables in the ports-list are to be identified as inputs, outputs, or inouts. The corresponding declarations have the form shown below:

f Input a1, a2;
f Output b1, b2;
f Inout c1, c2;

The port-type declarations here follow the module declaration mentioned above.

- The ports and the other variables used within the body of the module are to be identified as nets or registers with specific types in each case. The respective declaration statements follow the port-type declaration statements.

Examples:

```
wire a1, a2, c;
reg b1, b2;
```

The type declaration must necessarily precede the first use of any variable or signal in the module.

- The executable body of the module follows the declaration indicated above.

- The last statement in any module definition is the keyword “endmodule”.

- Comments can appear anywhere in the module definition.

Other Gate Primitives

All other basic gates are also available as primitives in Verilog. Details of the facilities and instantiations in each case are given in Table below. The following points are noteworthy here:

- In all cases of instantiations, one need not necessarily assign a name to the instantiation. It need be done only when felt necessary – say for clarity of circuit description.
- In all the cases the output port(s) is (are) declared first and the input port(s) is (are) declared subsequently.
- The buffer and the inverter have only one input each. They can have any number of outputs; the upper limit is compiler-specific. All other gates have one output each but can have any number of inputs; the upper limit is again compiler-specific.

Table for Basic gate primitives in Verilog with details

Gate	Mode of instantiation	Output port(s)	Input port(s)
AND	**and** ga (o, i1, i2, . . . i8);	o	i1, i2, . .
OR	**or** gr (o, i1, i2, . . . i8);	o	i1, i2, . .
NAND	**nand** gna (o, i1, i2, . . . i8);	o	i1, i2, . .
NOR	**nor** gnr (o, i1, i2, . . . i8);	o	i1, i2, . .
XOR	**xor** gxr (o, i1, i2, . . . i8);	o	i1, i2, . .
XNOR	**xnor** gxn (o, i1, i2, . . . i8);	o	i1, i2, . .
BUF	**buf** gb (o1, o2, …. i);	o1, o2, o3, . .	i
NOT	**not** gn (o1, o2, o3, . . . i);	o1, o2, o3, . .	i

Example for a typical A-O-I gate circuit

The commonly used A-O-I gate is shown in Figure 1 for a simple case. The module and the test bench for the same are given in Figure 2. The circuit has been realized here by instantiating the AND and NOR gate primitives. The names of signals and gates used in the instantiations in the module of Figure 2 remain the same as those in the circuit of Figure 1. The module aoi_gate in the figure has input and output ports since it describes a circuit with signal inputs and an output. The module aoi_st is a stimulus module. It generates inputs to the aoi_gate module and gets its output. It has no input or output ports.

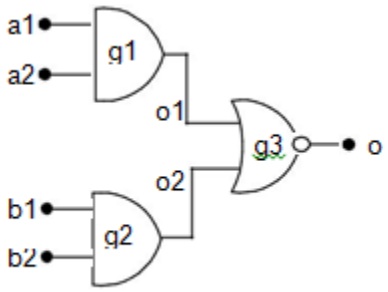


Figure for a typical A-O-I gate circuit.

```
/*module for the aoi-gate of figure 1 instantiating the gate primitives – fig 2*/module

aoi_gate(o,a1,a2,b1,b2);

input a1,a2,b1,b2;      // a1,a2,b1,b2 form the input //ports of the module

output o;               //o is the single output port of the module

wire o1,o2;             //o1 and o2 are intermediate signals //within the module

and g1(o1,a1,a2);  //The AND gate primitive has two and g2(o2,b1,b2);

                        // instantiations with assigned //names g1 & g2.

nor g3(o,o1,o2);    //The nor gate has one instantiation with assigned name g3.

endmodule

//Test-bench for the aoi_gate above
module aoi_st;
reg a1,a2,b1,b2;

//specific values will be assigned to a1,a2,b1, // and b2 and these connected
//to input ports of the gate insatntiations;
```

```
//hence these variables are declared as reg
wire o;
initial
begin
a1 = 0;
a2 = 0;
b1 = 0;
b2 = 0;
#3 a1 = 1;
#3 a2 = 1;
#3 b1 = 1;
#3 b2 = 0;
#3 a1 = 1;
#3 a2 = 0;
#3 b1 = 0;
end
initial #100 $stop;//the simulation ends after //running for 100 tu's.
initial $monitor($time , " o = %b , a1 = %b , a2 = %b , b1 = %b ,b2 = %b ",o,a1,a2,b1,b2);
aoi_gate gg(o,a1,a2,b1,b2);
endmodule
```

Tri-State Gates

Four types of tri-state buffers are available in Verilog as primitives. Their outputs can be turned ON or OFF by a control signal. The direct buffer is instantiated as
Bufif1 nn (out, in, control);

The symbol of the buffer is shown in Figure 1. We have

- out as the single output variable
- in as the single input variable and
- control as the single control signal variable.

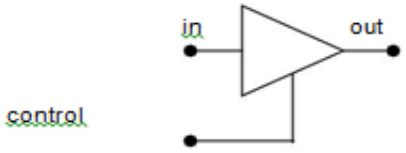


Figure 1 A tri-state buffer.

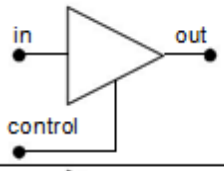
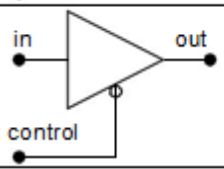
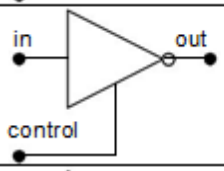
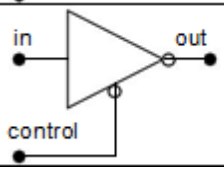
When
control = 1,
out = in.

When
control = 0,
out=tri-stated

out is cut off from the input and tri-stated. The output, input and control signals should appear in the instantiation in the same order as above. Details of bufif1 as well as the other tri-state type primitives are shown in Table 1.

In all the cases shown in Table 1, out is the output; in is the input, and control, the control variable.

Table 1 Instantiation and functional details of tri-state buffer primitives

Typical instantiation	Functional representation	Functional description
`bufif1` (out, in, control);		Out = in if control = 1; else out = z
`bufif0` (out, in, control);		Out = in if control = 0; else out = z
`notif1` (out, in, control);		Out = complement of in if control = 1; else out = z
`notif0` (out, in, control);		Out = complement of in if control = 0; else out = z

Array of Instances of Primitives

The primitives available in Verilog can also be instantiated as arrays. A judicious use of such array instantiations often leads to compact design descriptions. A typical array instantiation has the form

and gate [7 : 4] (a, b, c);

where a, b, and c are to be 4 bit vectors. The above instantiation is equivalent to combiningthe following 4 instantiations:

and gate [7] (a[3], b[3], c[3]), gate [6] (a[2], b[2], c[2]), gate [5] (a[1], b[1], c[1]), gate [4] (a[0], b[0], c[0]);

The assignment of different bits of input vectors to respective gates is implicit in the basic declaration itself. A more general instantiation of array type has the form

and gate[mm : nn](a, b, c);

where mm and nn can be expressions involving previously defined parameters, integers and algebra with them. The range for the gate is 1+ (mm-nn); mm and nn do not have restrictions of sign; either can be larger than the other.

Gate Delays

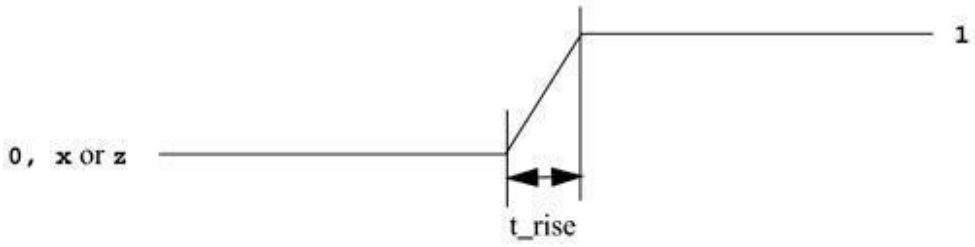
Until now, we described circuits without any delays (i.e., zero delay). In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog.

Rise, Fall, and Turn-off Delays

There are three types of delays from the inputs to the output of a primitive gate.

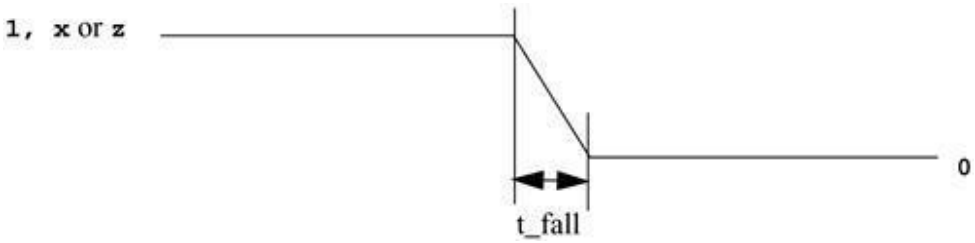
Rise delay

The rise delay is associated with a gate output transition to a 1 from another value.



Fall delay

The fall delay is associated with a gate output transition to a 0 from another value.



Turn-off delay

The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value.

If the value changes to x, the minimum of the three delays is considered.

Three types of delay specifications are allowed. If only one delay is specified, this value is used for all transitions. If two delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays. If all three delays are specified, they refer to rise, fall, and turn-off delay values. If no delays are

specified, the default value is zero. Examples of delay specification are shown in below

Example--Types of Delay Specification

```
    //Delay of delay_time for all transitionsand
#(delay_time) a1(out, i1, i2);

    // Rise and Fall Delay

Specification.and #(rise_val, fall_val)

a2(out, i1, i2);

// Rise, Fall, and Turn-off Delay Specification

bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

Examples of delay specification are shown below.

```
and #(5) a1(out, i1, i2); //Delay of 5 for all transitions and #(4,6) a2(out, i1, i2); // Rise = 4, Fall = 6

bufif0 #(3,4,5) b1 (out, in, control); // Rise = 3, Fall = 4, Turn-off = 5
```

Dataflow Modeling

Introduction

For small circuits, the gate-level modeling approach works very well because the number of gates is limited and the designer can instantiate and connect every gate individually. Also, gate-level modeling is very intuitive to a designer with a basic knowledge of digital logic design. However, in complex designs the number of gates is very large. Thus, designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level. Dataflow modeling provides a powerful way to implement a design. Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates. Later in this chapter, the benefits of dataflow modeling will become more apparent.

With gate densities on chips increasing rapidly, dataflow modeling has assumed great importance. No longer can companies devote engineering resources to handcrafting entire designs with gates. Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called logic synthesis. Dataflow modeling has become a popular design approach as logic synthesis tools have become sophisticated. This approach allows the designer to concentrate on optimizing the circuit in terms of data flow. For maximum flexibility in the design process, designers typically use a Verilog description style that combines the concepts of gate-level, data flow, and behavioral design. In the digital design community, the

term RTL (Register Transfer Level) design is commonly used for a combination of dataflow modeling and behavioral modeling.

Continuous Assignments

A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net. This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction. The assignment statement starts with the keyword assign. The syntax of an assign statement is as follows.

continuous_assign ::= assign [drive_strength] [delay3]
list_of_net_assignments ;

list_of_net_assignments ::= net_assignment { , net_assignment }

net_assignment ::= net_lvalue = expression

Notice that drive strength is optional and can be specified in terms of strength levelsThe default value for drive strength is strong1 and strong0. The delay value is also optional and can be used to specify delay on the assign statement. This is like specifying delays for gates. Delay specification is discussed in this chapter.
Continuous assignments have the following characteristics:

1. The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register.
2. Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right-hand-side operands changes and the value is assigned to the left-hand-side net.
3. The operands on the right-hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.
4. Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.

Example 6-1 Examples of Continuous Assignment

Continuous assign. out is a net. i1 and i2 are nets. assign out = i1

& i2;

Continuous assign for vector nets. addr is a 16-bit vector netaddr1

and addr2 are 16-bit vector registers.

```
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];
```

Concatenation. Left-hand side is a concatenation of a scalar

net and a vector net.

```
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

Implicit Continuous Assignment

Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared. There can be only one implicit declaration assignment per net because a net is declared only once.

In the example below, an implicit continuous assignment is contrasted with a regular continuous assignment.

```
//Regular continuous assignment

wire out;

assign out = in1 & in2;

//Same effect is achieved by an implicit continuous assignment wire out =in1
& in2;
```

Implicit Net Declaration

If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name. If the net is connected to a module port, the width of the inferred net is equal to the width of the module port.

```
wire i1, i2;

assign out = i1 & i2; //Note that out was not declared as a wire
```

//but an implicit wire declaration for out //is doneby the simulator

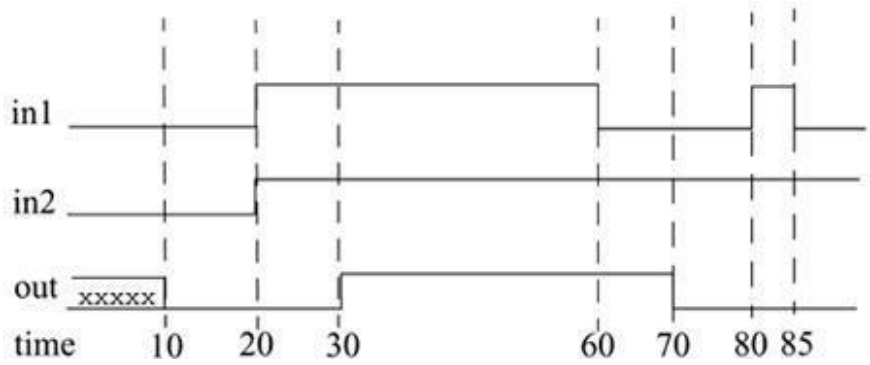
Delays

Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side. Three ways of specifying delays in continuous assignment statements are regular assignment delay, implicit continuous assignment delay, and net declaration delay.

Regular Assignment Delay

The first method is to assign a delay value in a continuous assignment statement. The delay value is specified after the keyword assign. Any change in values of in1 or in2 will result in a delay of 10 time units before recomputation of the expression in1 & in2, and the result will be assigned to out. If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of recomputation are considered. This property is called inertial delay. An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

assign #10 out = in1 & in2; // Delay in a continuous assign



\

Figure: Delays

The above waveform is generated by simulating the above assign statement. It shows the delay on signal out. Note the following change:

When signals in1 and in2 go high at time 20, out goes to a high 10 time unitslater (time = 30).

When in1 goes low at 60, out changes to low at 70.

However, in1 changes to high at 80, but it goes down to low before 10 timeunits have elapsed.

Hence, at the time of recomputation, 10 units after time 80, in1 is 0. Thus, out gets the value 0. A pulse of width less than the specified assignment delay is not propagated to the output.

Implicit Continuous Assignment Delay

An equivalent method is to use an implicit continuous assignment to specify both adelay and an assignment on the net.

```
//implicit continuous assignment delay

wire #10 out = in1 & in2;

//same as

wire out;

assign #10 out = in1 & in2;
```

The declaration above has the same effect as defining a wire out anddeclaring a continuous assignment on out.

Net Declaration Delay

A delay can be specified on a net when it is declared without putting a continuous assignment on the net. If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly. Net declaration delays can also be used in gate-level modeling.

```
//Net Delays

wire # 10 out;

assign out = in1 & in2;

//The above statement has the same effect as the following.wire

out;

assign #10 out = in1 & in2;
```

Expressions, Operators, and Operands

Dataflow modeling describes the design in terms of expressions instead of primitive gates. Expressions, operators, and operands form the basis of dataflow modeling.

Expressions

Expressions are constructs that combine operators and operands to produce a result.

Examples of expressions. Combines operands and operators a ^ b

```
addr1[20:17] + addr2[20:17] in1 | in2 ;
```

Operands

Some constructs will take only certain types of operands. Operands can be constants, integers, real numbers, nets, registers, times, bit-select (one bit of vectornet or a vector register), part-select (selected bits of the vector net or register vector), and memories or function calls (functions are discussed later).

```
integer count, final_count;

final_count = count + 1;//count is an integer operand

real a, b, c;

  c = a - b; //a and b are real operands

reg [15:0] reg1, reg2;

reg [3:0] reg_out;

reg_out = reg1[3:0] ^ reg2[3:0];//reg1[3:0] and reg2[3:0] are //part-select
                                                register operands

reg ret_value;

ret_value = calculate_parity(A, B);//calculate_parity is a //function
                                                type operand
```

Operators

Operators act on the operands to produce desired results. Verilog provides varioustypes of operators.

d1 && d2 // && is an operator on operands d1 and d2 !a[0]
// ! is an operator on operand a[0]

B >> 1 // >> is an operator on operands B and 1

Operator Types

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional. Some of these operators are similar to the operators used in the C programming language. Each operator type is denoted by a symbol. The following table shows the complete listing of operator symbols classified by category.

Table: Operator Types and Symbols

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
	>	greater than	two

Relational	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two

Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one

Shift	>> << >>> <<<	Right shift Left shift Arithmetic right shift Arithmetic left shift	Two Two Two Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

Let us now discuss each operator type in detail.

Arithmetic Operators

There are two types of arithmetic operators: binary and unary.

Binary operators

Binary arithmetic operators are multiply (*), divide (/), add (+), subtract (-), power (**), and modulus (%). Binary operators take two operands.

A = 4'b0011; B = 4'b0100; // A and B are register vectors D = 6; E

=4; F=2// D and E are integers

A * B // Multiply A and B. Evaluates to 4'b1100

D / E // Divide D by E. Evaluates to 1. Truncates any fractional part. A + B // Add Aand B. Evaluates to 4'b0111

B - A // Subtract A from B. Evaluates to 4'b0001 F = E

**F; //E to the power F, yields 16

If any operand bit has a value x, then the result of the entire expression is x. This seems intuitive because if an operand value is not known precisely, the result should be an unknown.

in1 =

4'b101x;in2 =

4'b1010;

sum = in1 + in2; // sum will be evaluated to the value 4'bx

Modulus operators produce the remainder from the division of two numbers. They operate similarly to the modulus operator in the C programming language.

13 % 3 // Evaluates to 1

16 % 4 // Evaluates to 0

-7 % 2 // Evaluates to -1, takes sign of the first operand

7 % -2 // Evaluates to +1, takes sign of the first operand

Unary operators

The operators + and - can also work as unary operators. They are used to specify the positive or negative sign of the operand. Unary + or ? operators have higher precedence than the binary + or ? operators.

-4 // Negative 4

+5 // Positive 5

Negative numbers are represented as 2's complement internally in Verilog. It is advisable to use negative numbers only of the type integer or real in expressions. Designers should avoid negative numbers of the type <sss> '<base> <nnn> in expressions because they are converted to unsigned 2's complement numbers and hence yield unexpected results.

//Advisable to use integer or real numbers -10 /

5// Evaluates to -2

//Do not use numbers of type <sss> '<base> <nnn>

-'d10 / 5// Is equivalent (2's complement of 10)/5 = (232 - 10)/5

where 32 is the default machine word width.

This evaluates to an incorrect and unexpected result

Logical Operators

Logical operators are logical-and (&&), logical-or (||) and logical- not (!). Operators && and || are binary operators. Operator ! is a unary operator. Logical operators follow these conditions:

Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), or x (ambiguous).

If an operand is not equal to zero, it is equivalent to a logical 1 (true condition). If it is 01equal to zero, it is equivalent to a logical 0 (false condition). If any operand bit is x or z, it is equivalent to x (ambiguous condition) and is normally treated by simulators as a false condition.

Logical operators take variables or expressions as operands.

Use of parentheses to group logical operations is highly recommended to improve readability. Also, the user does not have to remember the precedence of operators.

Logical operations A =
3;B = 0;

A && B // Evaluates to 0. Equivalent to (logical-1 && logical-0) A || B // Evaluates to 1. Equivalent to (logical-1 || logical-0) !A// Evaluates to 0. Equivalent to not(logical-1)

!B// Evaluates to 1. Equivalent to not(logical-0)

Unknowns

A = 2'b0x; B = 2'b10;

A && B // Evaluates to x. Equivalent to (x && logical 1)

// Expressions

(a == 2) && (b == 3) // Evaluates to 1 if both a == 2 and b == 3 are true.

// Evaluates to 0 if either is false.

Relational Operators

Relational operators are greater-than (>), less-than (<), greater-than-or-equal-to (>=), and less-than-or-equal-to (<=). If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and 0 if the expression is false. If there are any unknown or z bits in the operands, the expression takes a value x. These operators function exactly as the corresponding operators in the C programming language.

A = 4, B = 3

X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

A <= B // Evaluates to a logical

0A > B // Evaluates to a logical

1 Y >= X // Evaluates to a

logical 1

Y < Z // Evaluates to an x

Equality Operators

Equality operators are logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==) . When used in an expression, equality operators return logical value 1 if true, 0 if false. These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length. Table below lists the operators.

Table: Equality Operators

Expression	**Description**	**Possible Logical Value**
a == b	a equal to b, result unknown if x or z in a or b	0, 1, x
a != b	a not equal to b, result unknown if x or z in a or B	0, 1, x
a === b	a equal to b, including x and z	0, 1
a !== b	a not equal to b, including x and z	0, 1

It is important to note the difference between the logical equality operators (==, !=) and case equality operators (===, !==). The logical equality operators (==, !=) will yieldan x if either operand has x or z in its bits. However, the case equality operators (===, !==) compare both operands bit by bit and compare all bits, including x and z. The result is 1 if the operands match exactly, including x and z bits. The result is 0 if the operands do not match exactly. Case equality operators never result in an x.

A = 4, B = 3

X = 4'b1010, Y = 4'b1101

Z = 4'b1xxz, M = 4'b1xxz, N =

4'b1xxxA == B // Results in logical 0

X != Y // Results in logical

1X == Z // Results in x

Z === M // Results in logical 1 (all bits match, including x and z)

Z === N // Results in logical 0 (least significant bit does not match) M !== N //Results in logical 1

Bitwise Operators

Bitwise operators are negation (~), and(&), or (|), xor (^), xnor (^~, ~^). Bitwise operators perform a bit-by-bit operation on two operands. They take each bit in one operand and perform the operation with the corresponding bit in the other operand.If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand. Logic tables for the bit-by-bit computation are shown in Table. A z is treated as an x in a bitwise operation. The exception is the unary negation operator (~), which takes only one operand and operates on the bits of the single operand.

Table: Truth Tables for Bitwise Operators

bitwise and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bitwise or	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

bitwise xor	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

bitwise xnor	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

bitwise negation	result
0	1
1	0
x	x

Examples of bitwise operators are shown below.

X = 4'b1010, Y =

4'b1101Z = 4'b10x1

~X // Negation. Result is 4'b0101

X & Y // Bitwise and. Result is 4'b1000

X | Y // Bitwise or. Result is 4'b1111

X ^ Y // Bitwise xor. Result is

4'b0111 X ^~ Y // Bitwise xnor. Result

is 4'b1000X & Z // Result is 4'b10x0

It is important to distinguish bitwise operators ~, &, and | from logical operators !, &&, ||. Logical operators always yield a logical value 0, 1, x, whereas bitwise operators yield a bit-by-bit value. Logical operators perform a logical operation, not a bit-by-bit operation.

// X = 4'b1010, Y = 4'b0000

X | Y // bitwise operation. Result is 4'b1010

X || Y // logical operation. Equivalent to 1 || 0. Result is 1.

Reduction Operators

Reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (~^, ^~). Reduction operators take only one operand. Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result. The difference is that bitwise operations are on bits from two different operands, whereas reduction operations are on the bits of the same operand. Reduction operators work bit by bit from right to left. Reduction nand, reduction nor, and reduction xnor are computed by inverting the result of the reduction and, reduction or, and reduction xor, respectively.

// X = 4'b1010

&X //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|X//Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
^X//Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0

//A reduction xor or xnor can be used for even or odd parity

//generation of a vector.

The use of a similar set of symbols for logical (!, &&, ||), bitwise (~, &, |, ^), and reduction operators (&, |, ^) is somewhat confusing initially. The difference lies in the number of operands each operator takes and also the value of results computed.

Shift Operators

Shift operators are right shift (>>), left shift (<<), arithmetic right shift (>>>), and arithmetic left shift (<<<). Regular shift operators shift a vector operand to the right or the left by a specified number of bits. The operands are the vector and the number of bits to shift. When the bits are shifted, the vacant bit positions are filled with zeros. Shift operations do not wrap around. Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.

// X = 4'b1100

Y = X >> 1; //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB

position.Y = X << 1; //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB

position.

Y = X << 2; //Y is 4'b0000. Shift left 2 bits.

integer a, b, c; //Signed data typesa

= 0;

b = -10; // 00111...10110 binary

c = a + (b >>> 3); //Results in -2 decimal, due to arithmetic shift

Shift operators are useful because they allow the designer to model shift operations, shift-and-add algorithms for multiplication, and other useful operations.

Concatenation Operator

The concatenation operator ({, }) provides a mechanism to append multiple operands. The operands must be sized. Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result. Concatenations are expressed as operands within braces, with commas separating the operands. Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants.

// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110

Y = {B , C} // Result Y is 4'b0010

Y = {A , B , C , D , 3'b001} // Result Y is 11'b10010110001

Y = {A , B[0], C[1]} // Result Y is 3'b101

Replication Operator

Repetitive concatenation of the same number can be expressed by using a replication constant. A replication constant specifies how many times to replicate the number inside the brackets ({ }).

reg A;

reg [1:0] B, C;

reg [2:0] D;

A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;

Y = { 4{A} } // Result Y is 4'b1111

Y = { 4{A} , 2{B} } // Result Y is 8'b11110000

Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010

Conditional Operator

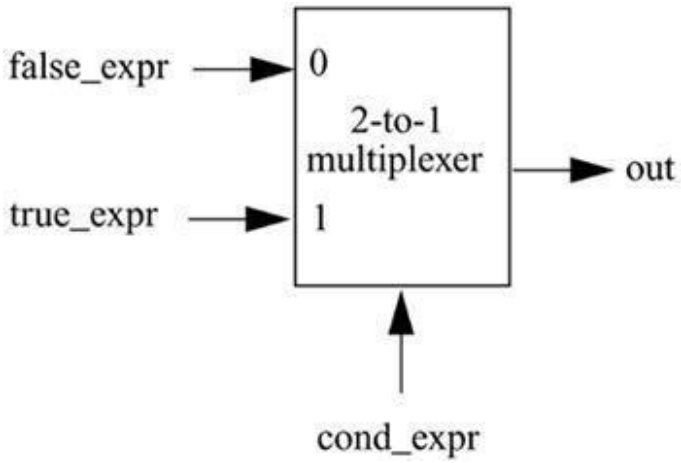
The conditional operator(?:) takes three operands.

Usage: condition_expr ? true_expr : false_expr ;

The condition expression (condition_expr) is first evaluated. If the result is true (logical 1), then the true_expr is evaluated. If the result is false (logical 0), then the false_expr is evaluated. If the result is x (ambiguous), then both true_expr and false_

expr are evaluated and their results are compared, bit by bit, to return for each bit position an x if the bits are different and the value of the bits if they are the same.

The action of a conditional operator is similar to a multiplexer. Alternately, it can be compared to the if-else expression.



Conditional operators are frequently used in dataflow modeling to model conditional assignments. The conditional expression acts as a switching control.

```
//model functionality of a tristate buffer

assign addr_bus = drive_enable ? addr_out : 36'bz;


//model functionality of a 2-to-1 mux

assign out = control ? in1 : in0;
```

Conditional operations can be nested. Each true_expr or false_expr can itself be a conditional operation. In the example that follows, convince yourself that (A==3) and control are the two select signals of 4-to-1 multiplexer with n, m, y, x as the inputs and out as the output signal.

```
assign out = (A == 3) ? ( control ? x : y ): ( control ? m : n) ;
```

Operator Precedence

Having discussed the operators, it is now important to discuss operator precedence. If no parentheses are used to separate parts of expressions, Verilog enforces the following precedence. Operators listed in Table are in order from highest precedence to lowest precedence. It is recommended that parentheses be used to separate expressions except in case of unary operators or when there is no ambiguity.

Table: Operator Precedence

Operators	**Operator Symbols**	**Precedence**
Unary	+ - ! ~	Highest precedence
Multiply, Divide, Modulus	* / %	
Add, Subtract	+ -	
Shift	<< >>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	&, ~& ^ ^~ |, ~|	
Logical	&& ||	
Conditional	?:	Lowest precedence

UNIT-V

Behavioral Modeling

Introduction

Behavioral modeling is the highest level of abstraction in the Verilog HDL. The other modeling techniques are relatively detailed. They require some knowledge of how hardware or hardware signals work. The abstraction in this modeling is as simple as writing the logic in C language. This is a very powerful abstraction technique. All that a designer need is the algorithm of the design, which is the basic information for any design.

Most of the behavioral modeling is done using two important constructs: initial and always. All the other behavioral statements appear only inside these two structured procedure constructs.

The Initial Construct

The statements which come under the initial construct constitute the initial block. The initial block is executed only once in the simulation, at time 0. If there is more than one initial block, then all the initial blocks are executed concurrently. The initial construct is used as follows:

```
initial
begin
reset=1'b0;
clk=1'b1;
end
or
initial
clk = 1'b1;
```

In the first initial block there are more than one statements hence they are written between begin and end. If there is only one statement then there is no need to put begin and end.

The always construct

The statements which come under the always construct constitute the always block. The always block starts at time 0, and keeps on executing all the simulation time. It works like a infinite loop. It is generally used to model a functionality that is continuously repeated.

```
always
#5clk=~clk
;initial
clk = 1'b0;
```

The above code generates a clock signal clk, with a time period of 10 units. The initial blocks initiates the clk value to 0 at time 0. Then after every 5 units of time it toggled, hence we get a

time period of 10 units. This is the way in general used to generate a clock signal for use in testbenches.

```
always@(posedge clk, negedge reset)
begin
a = b + c;
   d = 1'b1;
end
```

In the above example, the always block will be executed whenever there is a positive edge in theclk signal, or there is negative edge in the reset signal. This type of always is generally used in implement a FSM, which has a reset signal.

```
always
@(b,c,d)begin
   a = ( b + c
   )*d;e = b | c;
end
```

In the above example, whenever there is a change in b, c, or d the always block will be executed. Here the list b, c, and d is called the sensitivity list.

In the Verilog 2000, we can replace always @(b,c,d) with always @(*), it is equivalent to include all input signals, used in the always block. This is very useful when always blocks are used for implementing the combination logic.

OPERATIONS AND ASSIGNMENTS:

The design description at the behavioral level is done through a sequence of assignments. These are called 'procedural assignments' – in contrast to the continuous assignments at the data flow level. Though it appears similar to the assignments at the data flow level discussed in the last chapter, the two are different. The procedure assignment is characterized by the following:

- The assignment is done through the "=" symbol (or the "<=" symbol) as was the case with the continuous assignment earlier.
- An operation is carried out and the result assigned through the "=" operator to anoperand specified on the left side of the "=" sign – for example,N = ~N;
- Here the content of reg N is complemented and assigned to the reg N itself. The assignment is essentially an updating activity.
- The operation on the right can involve operands and operators. The operands can be of different types – logical variables, numbers – real or integer and so on.

wait CONSTRUCT

The wait construct makes the simulator wait for the specified expression to be true before proceeding with the following assignment or group of assignments. Its syntax has the form

wait (alpha) assignment1;

alpha can be a variable, the value on a net, or an expression involving them. If alpha is an expression, it is evaluated; if true, assignment1 is carried out. One can also have a group of assignments within a block in place of assignment1.

Example:

wait(clk) #2 a = b;

The simulator waits for the clock to be high and then assigns b to a with a delay of 2 ns. The assignment will be refreshed as long as the clk remains high.

The below code shows one version of the up-down counter module along with a test bench. It is a modification of the up down counter uses a wait construct. It has an enable input En. The counter is active and counts only when En = 1.

Verilog code:

```
module
ctr_wt(a,clk,N,En);input
clk,En;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a=4'b1111;
always
begin
wait(En
)
@(negedge clk)
a=(a==N)?4'b0000:a+1'b1;
end
endmodule
```

Test Bench

```
module tst_ctr_wt;
reg clk,En;
reg[3:0]N;
wire[3:0]a;
ctr_wt
c1(a,clk,N,En);initial
begin
clk=0;N=4'b1111;En=1'b0;#5 En=1'b1;#20 En=1'b0;
end
always
#2 clk=~clk;
initial #35 $stop;
initial $monitor($time,"clk=%h,En=%b,N=%b,a=%b",clk,En,N,a,);
 endmodule
```

Procedural Assignments

Procedural assignments are used for updating reg, integer, time, real, realtime, and memory data types. The variables will retain their values until updated by another procedural assignment.
There is a significant difference between procedural assignments and continuous assignments. Continuous assignments drive nets and are evaluated and updated whenever an input operand changes value. Whereas procedural assignments update the value of variables under the controlof the procedural flow constructs that surround them.

The LHS of a procedural assignment could be:

- reg, integer, real, realtime, or time data type.
- Bit-select of a reg, integer, or time data type, rest of the bits are untouched.
- Part-select of a reg, integer, or time data type, rest of the bits are untouched.
- Memory word.

Concatenation of any of the previous four forms can be specified.
When the RHS evaluates to fewer bits than the LHS, then if the right-hand side is signed, it will be sign-extended to the size of the left-hand side.

There are two types of procedural assignments: blocking and non-blocking assignments.

Blocking assignments: A blocking assignment statements are executed in the order they are specified in a sequential block. The execution of next statement begins only after the completionof the present blocking assignments. A blocking assignment will not block the execution of the next statement in a parallel block. The blocking assignments are made using the operator =.

```
initial
begin
   a = 1;
   b = #5 2;
   c = #2 3;
end
```

In the above example, a is assigned value 1 at time 0, and b is assigned value 2 at time 5, and c is assigned value 3 at time 7.

Non-blocking assignments: The nonblocking assignment allows assignment scheduling without blocking the procedural flow. The nonblocking assignment statement can be used whenever several variable assignments within the same time step can be made without regard to order ordependence upon each other. Non-blocking assignments are made using the operator <=.
Note: <= is same for less than or equal to operator, so whenever it appears in a expression it is considered to be comparison operator and not as non-blocking assignment.

```
initial
begin
   a <= 1;
   b <= #5 2;
   c <= #2 3;
end
```

In the above example, a is assigned value 1 at time 0, and b is assigned value 2 at time 5, and c is assigned value 3 at time 2 (because all the statements execution starts at time 0, as they are non-blocking assignments.

Conditional (if-else) Statement

The condition (if-else) statement is used to make a decision whether a statement is executed or not. The keywords if and else are used to make conditional statement. The conditional statementcan appear in the following forms.

```
if ( condition_1
   )statement_1;

if ( condition_2
   )statement_2;
else
   statement_3;

if ( condition_3
   )statement_4;
else if ( condition_4
   )statement_5;
else
   statement_6;

if ( condition_5
)begin
   statement_7;
   statement_8;
end
else
begin
   statement_9;
```

```
   statement_10;
end
```

Conditional (if-else) statement usage is similar to that if-else statement of C programminglanguage, except that parenthesis are replaced by begin and end.

Case Statement

The case statement is a multi-way decision statement that tests whether an expression matches one of the expressions and branches accordingly. Keywords case and endcase are used to make acase statement. The case statement syntax is as follows.

```
case (expression)
   case_item_1: statement_1;
   case_item_2: statement_2;
   case_item_3: statement_3;
   ...
   ...
   default: default_statement;
endcase
```

If there are multiple statements under a single match, then they are grouped using begin, andend keywords. The default item is optional.

Case statement with don't cares: casez and casex

casez treats high-impedance values (z) as don't cares. casex treats both high-impedance (z) and unknown (x) values as don't cares. Don't-care values (z values for casez, z and x values for casex) in any bit of either the case expression or the case items shall be treated as don't-care conditions during the comparison, and that bit position shall not be considered. The don't cares are represented using the ? mark.

Loop Statements

There are four types of looping statements in Verilog:

forever
repeat
while
for

Forever Loop

Forever loop is defined using the keyword forever, which Continuously executes a statement. It terminates when the system task $finish is called. A forever loop can also be ended by using the disable statement.

```
initial
begin
   clk = 1'b0;
   forever #5 clk =
~clk;end
```

In the above example, a clock signal with time period 10 units of time is obtained.

Repeat Loop

Repeat loop is defined using the keyword repeat. The repeat loop block continuously executes the block for a given number of times. The number of times the loop executes can be mention using a constant or an expression. The expression is calculated only once, before the start of loopand not during the execution of the loop. If the expression value turns out to be z or x, then it is treated as zero, and hence loop block is not executed at all.

```
initial
begin
   a = 10;
   b = 5;
   b <= #10 10;
   i = 0;
   repeat(a*b)
   begin
      $display("repeat in progress");
      #1 i = i + 1;
```

```
   end
end
```

In the above example the loop block is executed only 50 times, and not 100 times.It calculates (a*b) at the beginning, and uses that value only.

While Loop

The while loop is defined using the keyword while. The while loop contains an expression. The loop continues until the expression is true. It terminates when the expression is false. If the calculated value of expression is z or x, it is treated as a false. The value of expression is calculated each time before starting the loop. All the statements (if more than one) are mentioned in blocks which begins and ends with keyword begin and end keywords.

```
initial
begin
   a = 20;
   i = 0;
   while (i <
   a)begin
   $display("%d",i)
   ;i = i + 1;
   a = a -
   1;end
end
```

In the above example the loop executes for 10 times. (Observe that a is decrementing by one and i is incrementing by one, so loop terminated when both i and a become 10).

For Loop

The For loop is defined using the keyword for. The execution of for loop block is controlled by a three step process, as follows:

Executes an assignment, normally used to initialize a variable that controls the number of timesthe for block is executed.
Evaluates an expression, if the result is false or z or x, the for-loop shall terminate, and if it is true, the for-loop shall execute its block.
Executes an assignment normally used to modify the value of the loop-control variable and then repeats with second step.

Note that the first step is executed only once.

```
initial
begin
   a = 20;
   for (i = 0; i < a; i = i + 1, a = a - 1)
   $display("%d",i)
;end
```

The above example produces the same result as the example used to illustrate the functionalityof the while loop.

Examples:

1. Implementation of a 4x1 multiplexer.

```
module  mux4_1 (out, in0, in1, in2, in3, s0, s1);

output out;

// out is declared as reg, as default is wire

reg out;

// out is declared as reg, because we will
// do a procedural assignment to it.

input in0, in1, in2, in3, s0, s1;

// always @(*) is equivalent to
// always @( in0, in1, in2, in3, s0, s1 )

always
@(*)begin
  case ({s1,s0})
     2'b00: out = in0;
     2'b01: out = in1;
     2'b10: out = in2;
     2'b11: out = in3;
     default: out = 1'bx;
  endcase
end
endmodule
```

2. Implementation of a full adder.

```
module full_adder (sum, c_out, in0, in1, c_in);

output sum, c_out;
reg sum, c_out

input in0, in1, c_in;

always @(*)
  {c_out, sum} = in0 + in1 +

c_in;endmodule
```

3. Implementation of a 8-bit binary counter.

```
module ( count, reset, clk );

output [7:0] count;
reg [7:0] count;

input reset, clk;

// consider reset as active low signal

always @( posedge clk, negedge reset)
begin
  if(reset == 1'b0)
     count <= 8'h00;
  else
     count <= count + 8'h01;
end

endmodule
```

Implementation of a 8-bit counter is a very good example, which explains the advantage of behavioral modeling. Just imagine how difficult it will be implementing a 8-bit counter using gate-level modeling.
In the above example the incrementation occurs on every positive edge of the clock. When count becomes 8'hFF, the next increment will make it 8'h00, hence there is no need of any modulus operator. Reset signal is active low.

Block Statements

Block statements are used to group two or more statements together, so that they act as one statement. There are two types of blocks:

- Sequential block.
- Parallel

block. Sequential block:

The sequential block is defined using the keywords begin and end. The procedural statements in sequential block will be executed sequentially in the given order. In sequential block delay values for each statement shall be treated relative to the simulation time of the execution of the previous statement. The control will pass out of the block after the execution of last statement.

Parallel block:

The parallel block is defined using the keywords fork and join. The procedural statements in parallel block will be executed concurrently. In parallel block delay values for each statement are considered to be relative to the simulation time of entering the block. The delay control can be used to provide time-ordering for procedural assignments.

The control shall pass out of the block after the execution of the last time-ordered statement. Note that blocks can be nested. The sequential and parallel blocks can be mixed. Block names:

All the blocks can be named, by adding : block_name after the keyword begin or fork.

The advantages of naming a block are:

It allows to declare local variables, which can be accessed by using hierarchical name referencing.

They can be disabled using the disable statement (disable block_name;).