Syllabus: **MODULE-4**

Introduction to VHDL: VHDL description of combinational circuits, VHDL Models for multiplexers, VHDL Modules.

Latches and Flip-Flops: Set Reset Latch, Gated Latches, Edge-Triggered D Flip Flop 3, SRFlip Flop, J K Flip Flop, T Flip Flop, Flip Flop with additional inputs, Asynchronous Sequential Circuits

Introduction to VHDL:

VHDL is a **hardware description language** that is used to describe the behavior and structure of digital systems. The acronym VHDL stands for VHSIC Hardware Description Language, and VHSIC in turn stands for Very High Speed Integrated Circuit. However, VHDL is a general-purpose hardware description.

Language can be used to describe and simulate the operation of a wide variety of digital systems, ranging in complexity from a few gates to an interconnection of many complex integrated circuits. VHDL can describe a digital system at several different levels — behavioral, data flow, and structural.

For example, a binary adder could be described at the

- Behavioral level in terms of its function of adding two binary numbers, without giving any implementation details.
- Dataflow level by giving the logic equations for the adder.
- Structural level by specifying the interconnections of the gates which make up the adder.

VHDL Description of Combinational Circuits

A VHDL signal is used to describe a signal in a physical system. Every signal is of type bit, which means it can have a value of '0' or '1'. (Bit values in VHDL are enclosed in single quotes to distinguish them from integer values.)

Consider, The gate circuit of Figure 10-1 has five signals: A, B, C, D and E.

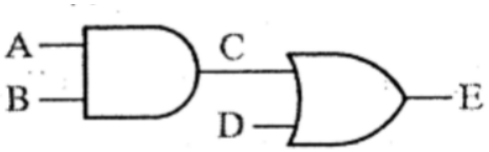


Figure 10-1: Gate Circuit

The symbol **"<="** is the signal assignment operator which indicates that the value computed on the right-hand side is assigned to the signal on the left side.

- *behavioral* description of the circuit in Figure 10-1 is

E <= D or (A and B);

Parentheses are used to specify the order of operator execution.

- *Dataflow* description of the circuit where it is assumed that each gate has a 5-ns propagation delay.

```
C <= A and B after 5 ns;
E <= C or D after 5 ns;
```

When the statements in Figure 10-1 are simulated,

- The first statement will be evaluated any time A or B changes, and the second statement will be evaluated any time C or D changes.
- Suppose that initially A 1, and B = C= D = E= 0. If B changes to 1 at time 0, C will change to 1 at time 5 ns. Then, E will change to 1 at time 10 ns.
- *Structural description*: VHDL code to do so requires that a two-input AND-gate component and a two-input OR-gate component be declared and defined.
- **Instantiation** statements are used to specify how components are connected.
- An instantiation statement is a concurrent statement that executes anytime one of the input signals in its port map changes.
- The circuit of Figure 10-1 is described by instantiating the AND gate and the OR gate as follows:

Gate1: AND2 **port map** (A, B, D);

Gate2: OR2 **port map** (C, D, E);

- Whenever A or B changes, these changes go to the Gate1 inputs, and then the component computes a new value of D. Similarly, the second statement passes changes in C or D to the Gate2 inputs, and then the component computes a new value of E. This is exactly how the real hardware works.
- A function returns a new value whenever it is called, but an instantiated component computes a new output value whenever its input changes.

VHDL signal assignment statements: The expression is evaluated when the statement is executed, and the signal on the left side is scheduled to change after delay. The square brackets indicate that after delay is optional.

In general, a signal assignment statement has the form

signal_name <= expression [after delay];

The VHDL simulator monitors the right side of each concurrent statement, and any time a signal changes, the expression on the right side is immediately re-evaluated. The new

value is assigned to the signal on the left side after an appropriate delay. Eg:

C<= A **and** B;

E<= C **or** D;

This implies that the propagation delays are 0 ns. In this case, the simulator will assume an infinitesimal delay referred to as (delta).

Assume that initially A= 1 and B= C= D= E= 0.

If B is changed to 1 at time 1 ns,

Then, C will change at time 1+Δns and E will change at time 1+2Δns .

Repetition in VHDL: Even if a VHDL program has no explicit loops, concurrent statements may execute repeatedly as if they were in a loop.

- Figure 10-2 shows an inverter with the output connected back to the input.
  - ✓ If the output is '0', then this '0' feeds back to the input and the inverter output changes to '1' after the inverter delay, assumed to be 10 ns.
  - ✓ Then, the '1' feeds back to the input, and the output changes to '0' after the inverter delay.
  - ✓ The signal CLK will continue to oscillate between '0' and '1', as shown in the waveform. The corresponding concurrent VHDL statement will produce the same result.

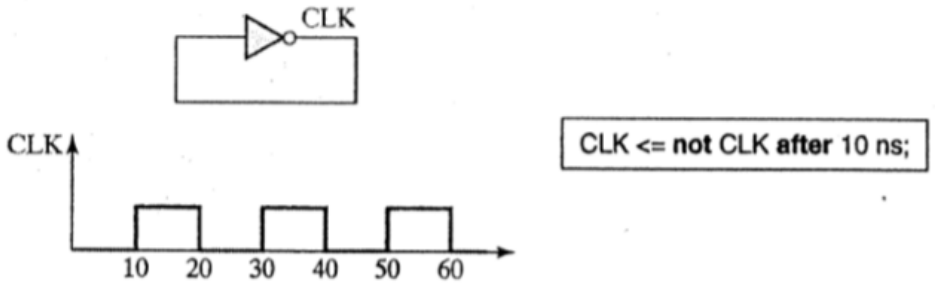


FIGURE 10-2: Inverter with Feedback

The statement in Figure 10-2 generates a clock waveform with a half period of 10 ns. On the other hand, the concurrent statement

CLK<=**not** CLK;

Will cause a runtime error during simulation. Because there is 0ns delay, the value of CLK will change at times 0+Δ, 0+ 2Δ, 0+ 3Δ etc. Because is an infinitesimal time, time will never advance to 1 ns.

In general, VHDL is not case sensitive, that is, capital and lower case letters are treated the same by the compiler and the simulator. Thus, the statements

Clk <= **NOT** clk **After** 10 nS;

and CLK <= **not** CLK **after** 10 ns; Would be treated exactly the same.

Signal names and other VHDL identifiers naming and other syntax Rules:

- May contain letters, numbers, and the underscore character (_).
- An identifier must start with a letter, and it cannot end with an underscore.
  - ✓ Thus, C123 and ab_23 are legal identifiers, but 1ABC and ABC_ are not.
- Every VHDL statement must be terminated with a semicolon.
- Spaces, tabs, and carriage returns are treated in the same way.
- In a line of VHDL code, anything following a double dash (--) is treated as a comment.
- Words such as **and**, **or**, and **after** are reserved words (or keywords) which have a special meaning to the VHDL compiler

Circuit with 3 Gates Example: consider circuit Figure 10-3, with three gates that have the signal A as a common input and the corresponding VHDL code.

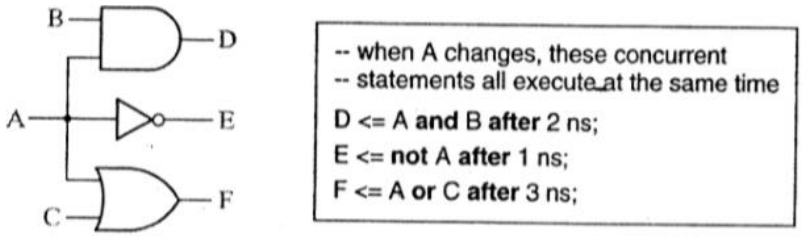


FIGURE 10-3: Three Gates with a Common Input and Different Delays

- The three concurrent statements execute simultaneously whenever A changes, just as the three gates start processing the signal change at the same time.
- However, if the gates have different delays, the gate outputs can change at different times.
  - ✓ If the gates have delays of 2 ns, 1 ns, and 3 ns, respectively, and A changes at time 5 ns, then the gate outputs D, E, and F can change at times 7 ns, 6 ns, and 8 ns, respectively.
  - ✓ If no delays were specified, then D, E, and F all will be updated at time 5+Δns.

Input as group of signals:

In digital design, we often need to perform the same operation on a group of signals. A one-dimensional array of bit signals is referred to as a bit-vector.

If a 4-bit vector named B has an index range 0 through 3, then the four elements of the bit vector are designated B(0), B(1), B(2), and B(3). The statement B “0110” assigns ‘0’ to B(0), ‘1’ to B(1), ‘1’ to B(2), and ‘0’ to B(3).

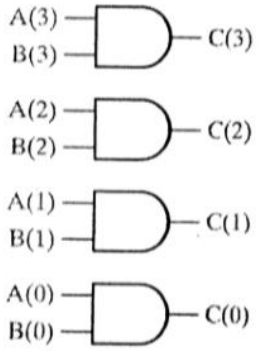


FIGURE 10-4: Array of AND Gates

- Figure 10-4 shows an array of four AND gates. The inputs are represented by bit-vectors A and B, and the outputs by bit-vector C.
- we can write four VHDL statements to represent the four gates,

```
-- the hard way
C(3) <= A(3) and B(3);
C(2) <= A(2) and B(2);
C(1) <= A(1) and B(1);
C(0) <= A(0) and B(0);
```

- It is much more efficient to write a single VHDL statement that performs the **and** operation on the bit-vectors A and B.

```
-- the easy way
C <= A and B;
```

- When applied to bit-vectors, the **and** operator performs the **and** operation on corresponding pairs of elements.

Statements with inertial delay:

Signal assignment statements containing "**after delay**" create what is called an **inertial** delay model.

C <= A **and** B **after** 10 ns;

- Consider a device with an inertial delay of D time units. If an input change to the device will cause its output to change, then the output changes D time units later.
- If the device receives two input changes within a period of D time units output does not change in response to either input change.

Eg: Initially Assume A = B=1,

- ✓ A changes to 0 at 15 ns, to 1 at 30 ns, and to 0 at 35 ns.
- ✓ Then C changes to 1 at 10 ns and to 0 at 25 ns,
- ✓ But C does not change in response to the A changes at 30 ns and 35 ns because these two changes occurred less than 10 ns apart.

- A device with an inertial delay of D time units filters out output changes that would occur in less than or equal to D time units.

VHDL can also model devices with an **ideal** (**transport**) delay. the output changes occur even if inputs changes within D time units. The VHDL signal assignment statement that models ideal (transport) delay is

signal_name <= **transport** expression **after** delay;

As an example, consider the signal assignment

C <= **transport** A **and** B **after** 10 ns;

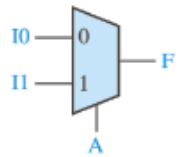
InitiallyAssume A = B=1,

- Changes to 0 at 15 ns, to 1 at 30 ns, and to 0 at 35 ns.
- Then C changes to 1 at 10 ns, to 0 at 25 ns, to 1 at 40 ns, and to 0 at 45 ns. Note that the last two changes are separated by just 5 ns (i.e. <=10ns).

VHDL Models for Multiplexers

Figure 10-5 shows a 2-to-1 multiplexer (MUX) with two data inputs and one control input.

FIGURE 10-5
2-to-1 Multiplexer



The MUX output is F = A' I0+ A I1. The corresponding VHDL statement can be written in 2 ways

F<= (**not** A **and** I0) **or** (A **and** I1);

Or

By using a conditional signal assignment statement

```
F <= I0 when A = '0' else I1;
```

- This statement executes whenever A, I0, or I1 changes.
- The MUX output is I0 when A = '0', and else it is I1. In the conditional statement, I0, I1, and F can either be bits or bit-vectors.

The general form of a conditional signal assignment statement is

signal_name <= expression1 **when** condition1

else expression2 **when** condition2

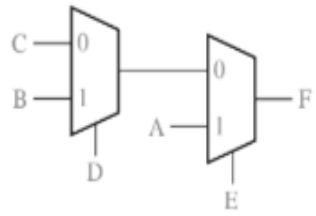
[**else** expressionN]

- This concurrent statement is executed whenever a change occurs in a signal used in one of the expressions or conditions.

- If condition1 is true, signal_name is set equal to the value of expression1, or else if condition2 is true, signal_name is set equal to the value of expression2, etc.
- The line in square brackets is optional.

3 inputs and one output using cascaded 2:1 Muxes

The output MUX selects A when E= '1'; or else it selects the out-put of the first MUX, which is B when D = '1', or else it is C.




```
F <= A when E = '1'
     else B when D = '1'
     else C;
```

4:1 Mux

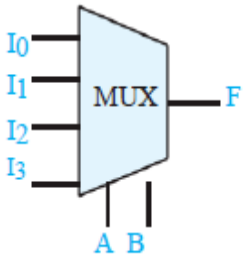


Figure shows 4-to-1 MUX with four data inputs and two control inputs, A and B. The control inputs select which one of the data inputs is transmitted to the output.

The logic equation for the 4-to-1 MUX is

$$F = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$$

There are 3 ways of writing the code

(a) one way to model the MUX is with the VHDL statement directly logic Equation

F <= (**not** A **and not** B **and** I0) **or** (**not** A **and** B **and** I1)
or (A **and not** B **and** I2) **or** (A **and** B **and** I3);

(b) Another way to model the 4-to-1 MUX is to use a conditional assignment statement:

F<=I0 **when** A&B = "00"
else I1 **when** A&B ="01"
else I2 **when** A&B = "10"
else I3;

The expression A&B means A concatenated with B, that is, the two bits A and B are merged together to form a 2-bit vector. This bit vector is tested, and the appropriate MUX input is selected.

For example, if A = '1' and B = '0', A&B = "10" and I2 is selected.

We could use a more complex condition avoiding A&B:

F <= I0 **when** A = '0' **and** B ='0'
else I1 **when** A='0' **and** B='1'
else I2 **when** A='1' **and** B='0'
else I3;

(c) A third way to model the MUX is to use a **selected signal assignment statement**,

- First set *Sel* equal to A&B. The value of Sel then selects the MUX input that is assigned to F.

```
sel <= A&B;
-- selected signal assignment statement
with sel select
  F <= I0 when "00",
       I1 when "01",
       I2 when "10",
       I3 when "11";
```

The general form of a selected signal assignment statement is

```
with expression_s select
   signal_s <= expression1 [after delay-time] when choice1,
               expression2 [after delay-time] when choice2,
               . . .
               [expression_n [after delay-time] when others];
```

- This concurrent statement executes whenever a signal changes in any of the expressions.
- First, expression_s is evaluated. If it equals choice1, signal_s is set equal to expression1; if it equals choice2, signal_s is set equal to expression2; etc.
- If all possible choices for the value of expression_s are given, the last line should be omitted; otherwise, the last line is required.
- When it is present, if expression_s is not equal to any of the enumerated choices, signal_s is set equal to expression_n.
- The signal_s is updated after the specified delay-time.

VHDL Modules

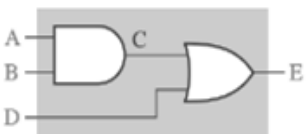
Question: Explain architecture of VHDL Module

To write a complete VHDL module, we must declare

- all of the input and output signals using an **entity** declaration
- And then specify the internal operation of the module using an **architecture** declaration.

As an example, consider Figure 10-8,

FIGURE 10-8
VHDL Module with Two Gates



- The entity declaration gives the name "two_gates" to the module. The port declaration specifies the inputs and outputs to the module. A, B, and D are input signals of type bit, and E is an output signal of type bit.
- The architecture is named "gates". The signal C is declared within the architecture because it is an internal signal. The two concurrent statements that describe the gates are placed between the keywords **begin** and **end**

```
entity two_gates is
    port (A,B,D: in bit; E: out bit);
end two_gates;
architecture gates of two_gates is
    signal C: bit;
begin
    C <= A and B; -- concurrent
    E <= C or D; -- statements
end gates;
```

An entity and an architecture at the top level, and also specify an entity and architecture for each of the component modules that are part of the system must be specified.

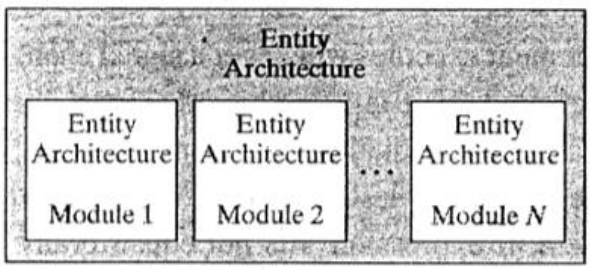


FIGURE 10-9: VHDL Program Structure

Each entity declaration includes a list of interface signals that can be used to connect to other modules or to the outside world. General form:

entity entity-name **is**

[**port**(interface-signal-declaration);]

end [**entity**] [entity-name];

The items enclosed in square brackets are optional.

The interface-signal-declaration normally has the following form:

list-of-interface-signals: mode type [:= initial-value]

Input signals are of mode **in**, output signals are of mode **out**, and bi-directional signals are of mode **inout**.

The optional initial-value is used to initialize the signals on the associ-ated list; otherwise, the default initial value is used for the specified type. For exam-ple, the port declaration

Eg: port(A, B: **in** integer : = 2; C, D: **out** bit);

- A and B are input signals of type integer that are initially set to 2,
- and C and D are output signals of type bit that are initialized by default to '0'.

Associated with each entity is one or more architecture declarations of the form

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

[declarations]

begin

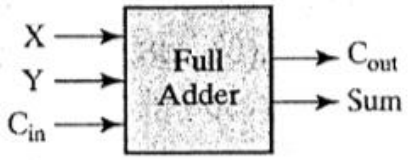
architecture bodys

end [**architecture**] [architecture-name];

- In the declarations section, we can declare signals and components that are used within the architecture.
- The architecture body contains statements that describe the operation of the module.

Full Adder:

- Entity and architecture for a full adder module.
- The entity specifies the inputs and outputs of the adder module, as shown in Figure 10-10. The port declaration specifies that X, Y and Cin are input signals of type bit, and that Cout and Sum are output signals of type bit.




```
entity FullAdder is
   port (X,Y,Cin: in bit;          -- Inputs
         Cout, Sum: out bit);  -- Outputs
end FullAdder;
```

FIGURE 10-10: Entity Declaration for a Full Adder Module

The operation of the full adder is specified by an architecture declaration:

```
architecture Equations of FullAdder is
begin– concurrent assignment statements
      Sum <= X xor Y xor Cin after 10 ns;
      Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
end Equations;
```

In this example, the architecture name (Equations) is arbitrary, but the entity name (FullAdder) must match the name used in the associated entity declaration.

The VHDL assignment statements for Sum and Cout represent the logic equations for the full adder.

Four-Bit Full Adder

FullAdder module defined above is used as a component in a system which consists of four full adders connected to form a 4-bit binary adder (Figure 10-11).

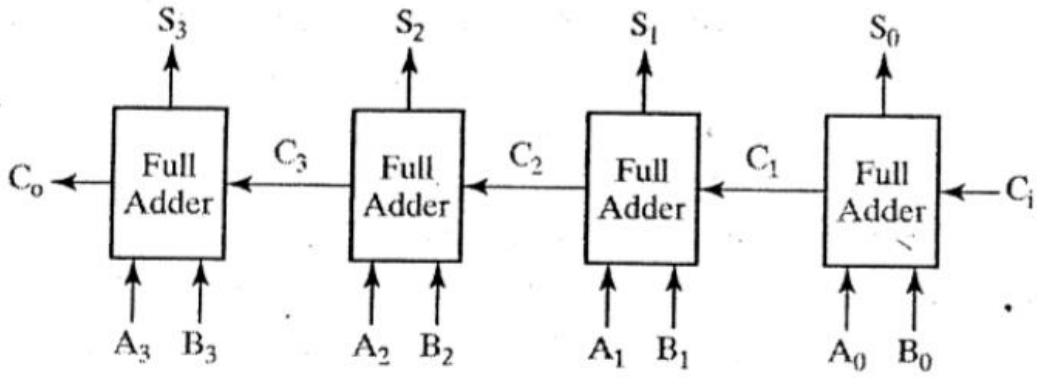


FIGURE 10-11: 4-Bit Binary Adder

- **Entity Declaration:** 4-bit adder as an entity FullAdder4 (see Figure 10-12). Because the inputs and the sum output are four bits wide, we declare them as bit_vectors which are dimensioned 3 **downto** 0. (We could have used a range 1 **to** 4 instead.)
- Next, we specify the FullAdder as a component within the **architecture of Adder4** (Figure 10-12). The component specification is very similar to the entity declaration for the full adder, and the input and output port signals correspond to those declared for the full adder. Following the component statement, we declare a 3-bit internal carry signal C.
- In the body of the architecture, we create several instances of the FullAdder component.
  - ✓ Each copy of FullAdder has a name (such as FA0) and a port map.
  - ✓ The signal names following the port map correspond one-to-one with the signals in the component port. The order of the signals in the port map must be the same as the order of the signals in the port of the component declaration.

Note: In preparation for simulation, we can place the entity and architecture for the FullAdder and for Adder4 together in one file and compile.

Alternatively, we could compile the FullAdder separately and place the resulting code in a library which is linked in when we compile Adder4.

```
entity Adder4 is
        port (A, B: in bit_vector(3 downto 0); Ci: in bit; – Inputs
                S: out bit_vector(3 downto 0); Co: out bit); – Outputs
end Adder4;
architecture Structure of Adder4 is
component FullAdder
        port (X, Y, Cin: in bit; – Inputs
                Cout, Sum: out bit); – Outputs
end component;
signal C: bit_vector(3 downto 1);
begin – instantiate four copies of the FullAdder
        FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
        FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
        FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
        FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
end Structure;
```

FIGURE 10-12: Structural Description of 4-Bit Adder

General form of Components

The 4-bit adder module demonstrates the use of VHDL components to write structural VHDL code. Components used within the architecture are declared at the beginning of the architecture, using a component declaration of the form

component component-name

port (list-of-interface-signals-and-their-types);

end component;

The port clause used in the component declaration has the same form as the port clause used in an entity declaration.

label: component-name **port map** (list-of-actual-signals);

The list of actual signals must correspond one-to-one to the list of interface signals specified in the component declaration.

Latches and Flip-Flops

- Sequential switching circuits have the property that the output depends not only on the present input but also on the past sequence of inputs. In effect, these circuits must be able to “remember” something about the past history of the inputs in order to produce the present output. **Latches and flip-flops** are commonly used memory devices in sequential circuits.
- Latches and flip-flops are memory devices which can assume one of two stable output states and which have one or more inputs that can cause the output state to change.

- A memory element that has no clock input is often called a **latch**
- a memory device that changes output state in response to a clock input and not in response to a data input is called **flip-flop.**

Prove that all Sequential circuit must contain feedback but all feed back circuits are not sequential:

- **feed back circuit which is not sequential:**

For example, consider the circuit in Figure 11-1(a). If at some instant of time the inverter input is 0,

- ✓ This 0 will propagate through the inverter and cause the output to become 1 after the inverter delay.
- ✓ This 1 is fed back into the input, so after the propagation delay, the inverter output will become 0. When this 0 feeds back into the input, the output will again switch to l, and so forth.
- ✓ The inverter output will continue to oscillate back and forth between 0 and 1, as shown in Figure 11-1(b), and it will never reach a stable condition.

FIGURE 11-1

(a) Inverter with feedback (b) Oscillation at inverter output

- **Sequential circuit:** consider a feedback loop which has two inverters in it, as shown in Figure 11-2(a).

FIGURE 11-2

(a) (b)

In this case, the circuit has two stable conditions, often referred to as **stable states.**

- If the input to the first inverter is 0, its output will be 1. Then, the input to the second inverter will be 1, and its output will be 0. This 0 will feed back into the first inverter, but because this input is already 0, no changes will occur. The circuit is then in a stable state. As shown in Figure 11-2(b),
- A second stable state of the circuit occurs when the input to the first inverter is 1 and the input to the second inverter is 0.

A simple loop of 2 inverters lacks any external means of initializing the state to ne of the stable states. ***Set - Reset*** *latches have input for this initializations*

Set-Reset Latch

Circuit is normally represented as cross-coupled connection as shown and its symbol shown in Figure 11-5(b).

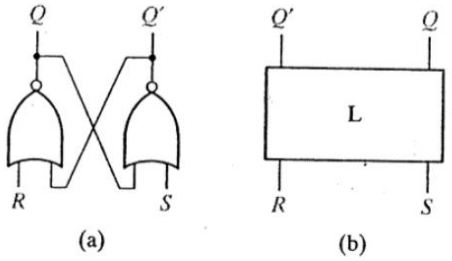


FIGURE 11-5: S-R Latch

Operation summary:

$S = R = 0$	No state change
$S = 1, R = 0$	Set Q to 1 (after active Ck edge)
$S = 0, R = 1$	Reset Q to 0 (after active Ck edge)
$S = R = 1$	Not allowed

Question**: Why Inputs S=1 R=1 are invalid in S R Latch?**

If $S = R = 1$, the latch will not operate properly, as shown in Figure 11-6.

The notation 1 → 0 means that the input is originally 1 and then changes to 0.

- Note that when *S* and *R* are both l, *P* and *Q* are both 0. Therefore, *P* is not equal to *Q* , and this violates a basic rule of latch operation P = Q'.
- Furthermore, if *S* and *R* are simultaneously changed back to 0, *P* and *Q* may both change to 1.
- If $S = R = 0$ and $P = Q = 1$, then after the 1's propagate through the gates, *P* and *Q* will become 0 again, and the latch may continue to oscillate if the gate delays are equal.

$P = Q = 1$, then after the 1's propagate through the gates, *P* and *Q* will become 0 again, and the latch may continue to oscillate if the gate delays are equal.

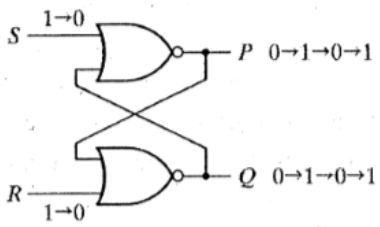


FIGURE 11-6: Improper S-R Latch Operation

Timing Diagram for S-R latch:

Figure 11-7 shows a timing diagram for the S-R latch.

- When *S* changes to 1 at time t_1, *Q* changes to 1 a short time (ε) later. (ε represents the response time or delay time of the latch.)

- At time t_2, when *S* changes back to 0, *Q* does not change.(S=R=0 → No change)
- At time t_3, *R* changes to 1, and *Q* changes back to 0 a short time (ε) later.
- If *S* 1 for a time less than ε, the gate output will not change and the latch will not change state.

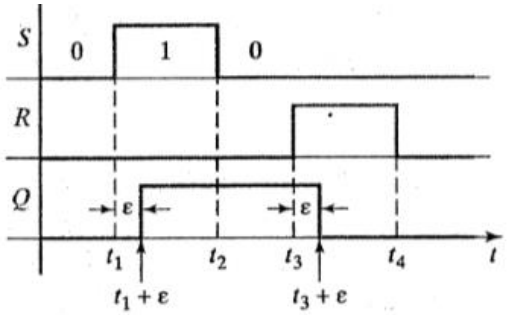


FIGURE 11-7: Timing Diagram for S-R Latch

Meta Stable State:

- Theoretically Latch has only 2 states.
- Situation where the voltage level at the output of two inverters is approximately half way between the voltage levels for logic 0 and logic 1. This is referred to as Metastable State.
- Certain events can cause the latch to enter metastable state for sshort time.
- Simultaneous change in S and R from 1→ 0 can cause latch to enter metastable State.

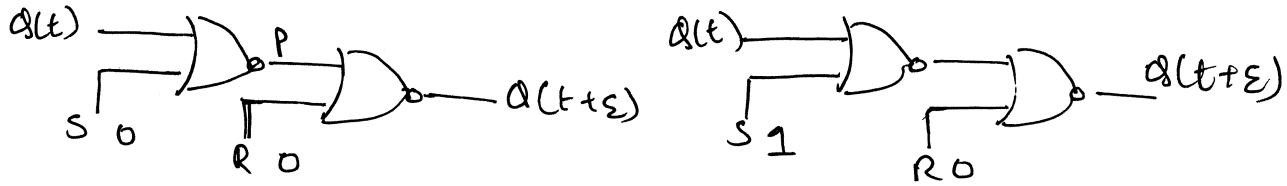
- Meta Stable State must be avoided because
  (i) Stable State entered from metastable state is unpredictable.
  (ii) any gates or latches with an input that is in the metastable will respond unpredictably.

Deriving Characteristic Equation for an S-R latch

Present state & Next State of latches & flipflops

- Present state denote state of Q output of the latch or flipflop at the time any input signal changes ($Q(t)$)
- Next State denote the state of Q output after latch or flipflop has reacted to input change and stabilized. ($Q(t+\varepsilon)$)

Method
a) Considering (a) & (b) of latches by conceptually breaking the feed back, Considering $Q(t)$ as input & $Q(t+\varepsilon)$ as o/p.



we have,

$$Q(t+\varepsilon) = R(t)'[S(t) + Q(t)]$$
$$= R(t)'S(t) + R(t)'Q(t)$$

and Present output $P = S(t)'Q(t)'$

Next State equation without including time explicitly using Q as present state we have

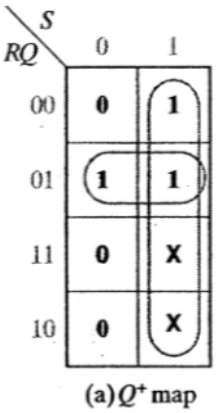
$$Q^+ = R'S + R'Q$$
$$P = S'Q'$$

From above, we have table

Present State Q	Next State Q^+ SR 00	SR 01	SR 11	SR 10	Present Output P SR 00	SR 01	SR 11	SR 10
0	(0)	(0)	(0)	1	1	1	0	0
1	(1)	0	0	(1)	0	0	0	0

For all Stable States $P = Q'$ except $S = R = 1$

∴ making $S = R = 1$ don't care conditions & Simplifying next State Equation we have,



(a) Q^+ map

S	R	Q	Q^+	
0	0	0	0	
0	0	1	1	
0	1	0	0	
0	1	1	0	
1	0	0	1	
1	0	1	1	
1	1	0	–	Inputs not allowed
1	1	1	–	

(b) Truth table

$S + R'Q \quad (SR = 0)$

FIGURE 11-8: Derivation of Q+ for an S-R Latch

- An Equation that expresses the next state of a latch in terms of its present state & inputs is referred to as next-state equation or <u>characteristic Equation</u>.

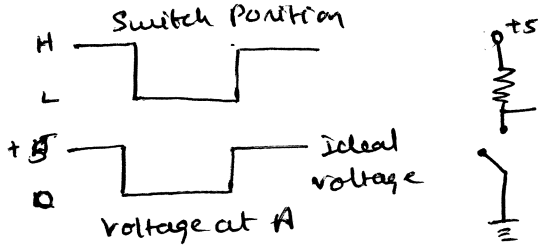
Method

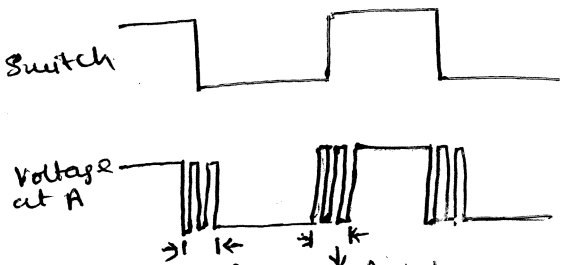
b) Characteristic Equation can be derived based on constructing a truth table for next state of Q. & plotting Q^+ on a k-map derives same Equation.

Application of S-R latch (debouncing Switches)

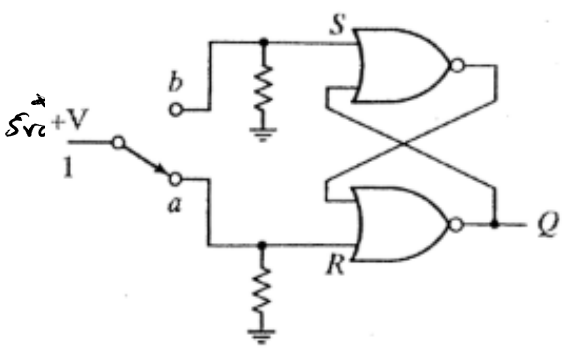
- When Mechanical Switch is opened or closed, the switch contacts tends to vibrate or bounce open & closed several times before settling down to final position. This produces noisy transitions which appears in the circuit operation. This Problem is known as <u>key bounce</u>

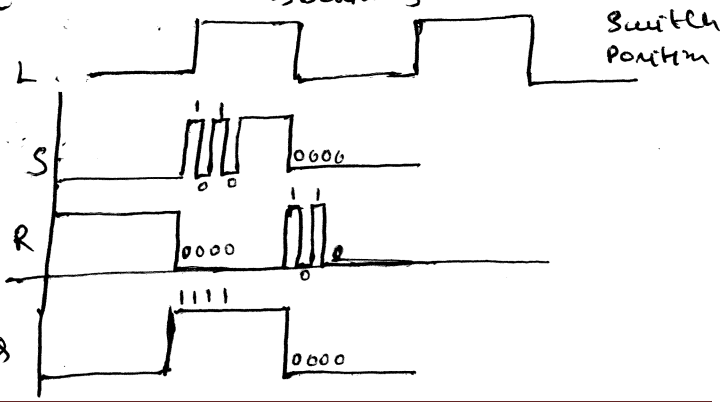
One way to avoid bouncing problem is to use SR-latch and this circuit is called switch debounce.





S R latch debounce circuit





- When switch is moved to position H,

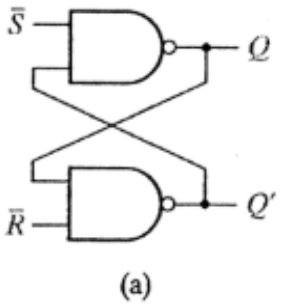
R=0, S=1 Bouncing occurs at S input & latch sees this as a series of high/low inputs resulting with high value. FF will be set with Q=1.

- when switch bounces R=S=0 flip flop remains with Q=1
- when it regains the contact R=0 S=1 causes it to set again.

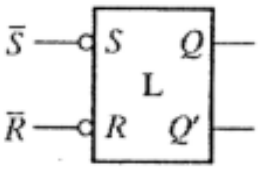
- When switch is moved to position L, S=0, R=1 and bouncing occurs at R input and flip flop sees this as high & low inputs.
  - when switch bounces R=S=0 it remains with Q=0.
  - when switch regains contact R=1, S=0 causes Q=0 again.

The result is a clean high to low signal at flip flop output.

An alternative form of S-R latch uses NAND gates



(a)



(b)

$\bar{S}$	$\bar{R}$	Q	Q^+	
1	1	0	0	
1	1	1	1	
1	0	0	0	
1	0	1	0	
0	1	0	1	
0	1	1	1	
0	0	0	–	Inputs not allowed
0	0	1	–	

(c)

- $\bar{S}=0$ will set Q to 1
- $\bar{R}=0$ will reset Q to 0

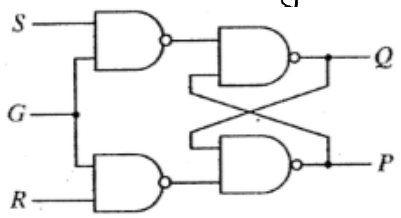
If $\bar{S}$ & $\bar{R}$ are 0 at same time both Q & Q' outputs are forced to 1.

∴ $\bar{S}=\bar{R}=0$ is not allowed.

Gated Latches: Gated latches have an additional input called the gate or enable input.

- when gate input is inactive, the state of latch cannot be changed.
- when gate input is active, the latch is controlled by other inputs (S & R).

Consider NAND-gate version of gated S-R latch



The next-state equation is

$$Q+ = SG + Q(R' + G')$$

and the equation for the P output is

$$P = Q' + RG$$

FIGURE 11-11: NAND-Gate Gated S-R Latch

Next State Q^+

Present State Q	G = 0 SR 00	SR 01	SR 11	SR 10	G = 1 SR 00	SR 01	SR 11	SR 10
0	⓪	⓪	⓪	⓪	⓪	⓪	1	1
1	①	①	①	①	①	0	①	①

(a)

Present Output P

Present State Q	G = 0 SR 00	SR 01	SR 11	SR 10	G = 1 SR 00	SR 01	SR 11	SR 10
0	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	0

(b)

TABLE 11-2: Next-State and Output of Gated S-R Latch

The above shows next state & output P tables.

- when G=0 circuit is always in a stable state (from (a))
- when G=1, S=1 sets the latch and
  G=1, R=1 resets the latch.
- for all the stable states is Q^+ except G=S=R=1 satisfies the condition $P = Q'$

∴ S=R=1 input combination is not allowed.

3 ways proving S=R=1 as invalid inputs (i.e. circled)

a) From above (a) & (b) tables all states (stables) in Q^+ except G=S=R=1 satisfies condition $P = Q'$

∴ S=R=1 is not allowed.

b) Consider the circuit for S=R=1 G ⇒ 1→0

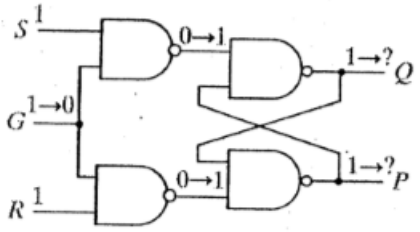
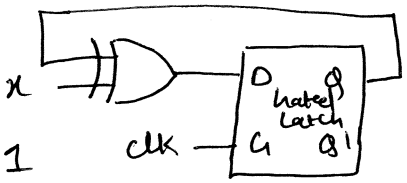


FIGURE : Race Condition in the Gated S-R Latch

- when G changes, both inputs to S-R latch changes from 0→1
- This causes both gates of basic S-R latch to attempt to change from 1→0 which violates $P = Q'$ rule

- Due to propogation delay gate determines whether latch stabilizes with Q=0 or Q=1.

Why Gated latches could not used as flip flop Where the clock signal is connected to gate inputs of the latches?



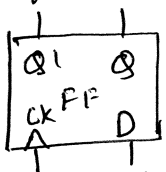
consider

- when $x=1$, $clk=1$ $D=Q'$, causes Q to change and if clk remain 1, the change in Q will feed back and cause Q to change again.
- If clk remains at 1, Q will oscillate (as intended)
- If clk remains at 1 for short time, just to allow Q to change but not feed back that causes a second change. (will not operate as intended)
- With several latches, variations in gate delays would make impossible to provide the correct clock width to all latches.

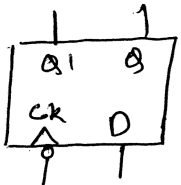
To avoid this timing problem we have edge-triggered flip flops. where flip flop outputs are changed on an edge of clock and outputs cannot change at other times even input changes.

Edge-Triggered D-flip flop: A D flip flops has two inputs, D (Data) & Ck (clock). The small arrowhead on the flip flop symbol identifies the clock input.

- The flip flop output changes only in response to the clock, not a a change in D.
- If output response to a 1 to 0 transition on the clock input → flip flop is triggered on falling edge(↓) of the clock. (bubble on the clock indicates ↓)
  for 0 to 1 → flip flop is triggered on rising edge (↑) of the clock.
- active edge refers to clock edge → Rising edge trigger or falling edge trigger



(a) Rising-edge trigger



(b) Falling edge trigger

D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

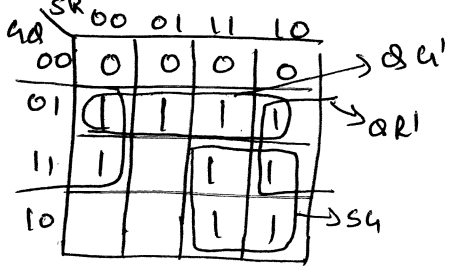
Truth table

$Q^+ = D$

8) we know $Q^+ = SG + Q(R' + G')$

$= SG + QR' + QG'$

Consider k-map of Q^+

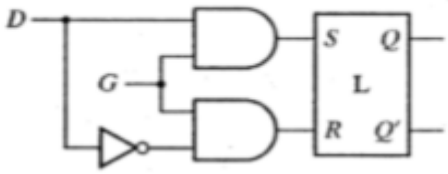


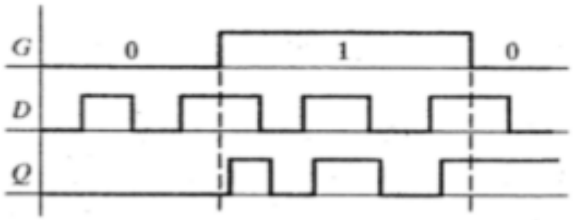
From k-map, it is evident that Q^+ has a static 1-hazard for inputs $S=R=G=Q=1$ and $G=0$ $S=R=Q=1$

- when $S=1$ $R=1$ output becomes unstable leading to race ~~error~~ condition.

- Because of static 1 hazard, for G changinging from $0 \to 1$ and again $1 \to 0$ may changes Q to 0, but because $S=R=1$ Q is forced back to 1. which causes the latch to not to stabilize with $Q=0$. which is interpreted as race condition.

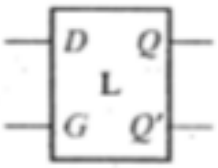
Gated D-Latch - It can be obtained from gated S-R latch by connecting S to D & R to D'.





- Q of the D latch is remains unchanged while G is inactive
  i.e. If $G=0$, D → memory (no changes)
- when G is active Q becomes equal to D after some delay. i.e. if $G=1$, $Q=G$ after delay.

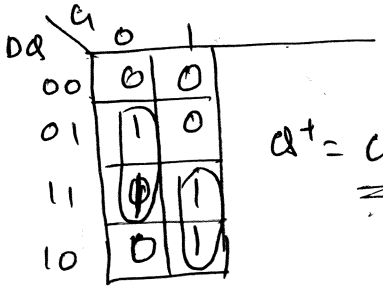
Symbol



Truth table

G	D	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

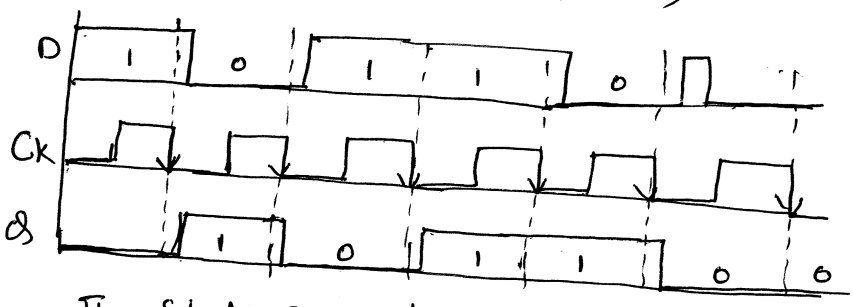
k-map



$Q^+ = G'Q + GD$

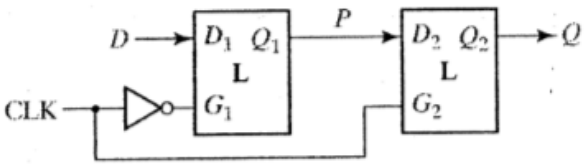
$Q^+ = G'Q + GD$

Timing for D-flip flop (Falling-Edge Trigger or Negative edge triggered)



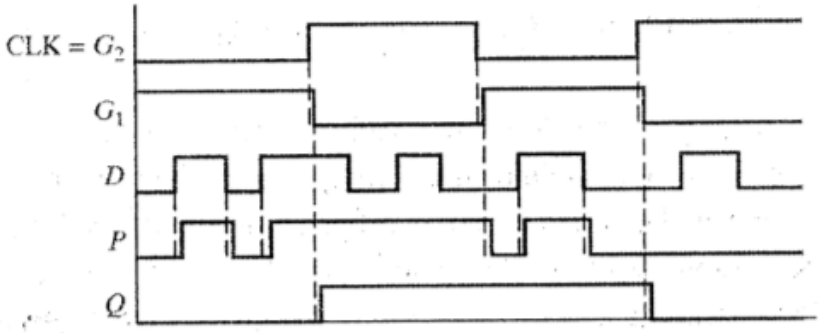
The State of D flip flop after active clock edge (Q+) is equal to the input (D) before the active edge.

Construction of rising-edge triggered D-flip flop using two gated D Latches and an inverter



(a) Construction from two gated D latches

- When CLK=0, G1 = 1 first Latch is transparent so that P output follows the D input. Because G2 = 0, Second latch holds current value of Q.
- When CLK = 1, G1 = 0 ⇒ Current value of D is stored in first latch. Because G2 = 1 the value P flows through the second latch to the Q output.
- When CLK changes back to 0, the second latch takes on the value of P and hold it and then first Latch starts following the D input again.
- If first Latch starts following the D input before the second latch takes on the value of P, the flip flop will not work properly.
- The value of D at the time of rising edge of clock determines the value of Q, and any extra changes in D that occur between rising clock edges have no effect on Q.

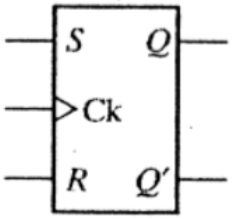


(b) Timing analysis

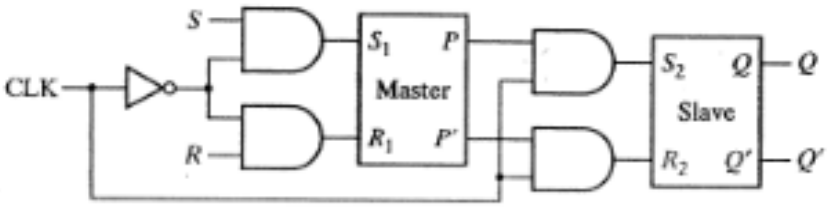
S-R Flip-Flop

An S-R flip-flop (Figure 11-22) is similar to an S-R latch in that S = 1 sets the Q output to 1, and $R = 1$ resets the Q output to 0. The essential difference is that the flip-flop has a clock input, and the Q output can change only after an active clock edge. The truth table and characteristic equation for the flip-flop are the same as for the latch, but the interpretation of Q^+ is different. For the latch, Q^+ is the value of Q after the propagation delay through the latch, while for the flip-flop, Q^+ is the value that Q assumes after the active clock edge.

Figure 11-23(a) shows an S-R flip-flop constructed from two S-R latches and gates. This flip-flop changes state after the rising edge of the clock. The circuit is often referred to as a master-slave flip-flop. When CLK = 0, the S and R inputs set the outputs of the master latch to the appropriate value while the slave latch holds the previous value of Q. When the clock changes from 0 to 1, the value of P is held in the master latch and this value is transferred to the slave latch. The master latch holds the value of P while CLK = 1, and, hence, Q does not change. When the clock changes from 1 to 0, the Q value is latched in the slave, and the master can process new inputs. Figure 11-23(b) shows the timing diagram. Initially, $S = 1$ and Q changes to 1 at t_1. Then $R = 1$ and Q changes to 0 at t_2.

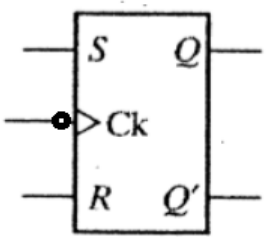


SR FF
(Rising edge/ Postitive Edge Triggered)



(a) Implementation with two latches

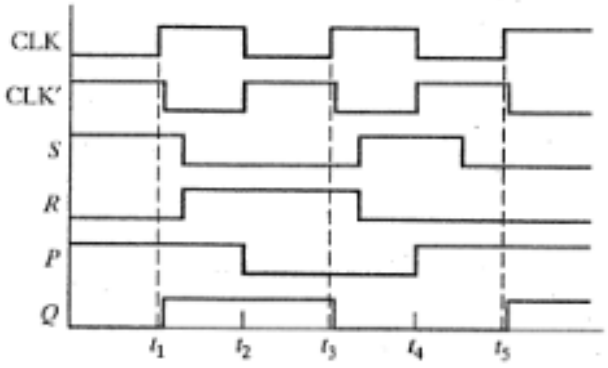
Rising Edge Triggered SR FF



SR FF
(Falling edge/ Negative Edge Triggered)

Operation summary:

$S = R = 0$	No state change
$S = 1, R = 0$	Set Q to 1 (after active Ck edge)
$S = 0, R = 1$	Reset Q to 0 (after active Ck edge)
$S = R = 1$	Not allowed



(b) Timing analysis

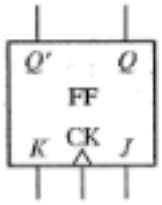
Rising Edge Triggered SR FF

FIGURE 11-23: S-R Flip-Flop Implementation and Timing

At first glance, this flip-flop appears to operate just like an edge-triggered flipflop, but there is a subtle difference. For a rising-edge-triggered flip-flop the value of the inputs is sensed at the rising edge of the clock, and the inputs can change while the clock is low. For the master-slave flip-flop, if the inputs change while the clock is low, the flip-flop output may be incorrect. For example, in Figure 11-23(b) at t_4, $S = 1$ and $R = 0$, so P changes to 1. Then S changes to 0 at t_5, but P does not change, so at t_5, Q changes to 1 after the rising edge of CLK. However, at t_5, $S = R = 0$, so the state of Q should not change. We can solve this problem if we only allow the S and R inputs to change while the clock is high.

J-K Flip-Flop

The J-K flip-flop (Figure 11-24) is an extended version of the S-R flip-flop. The J-K flip-flop has three inputs—J, K, and the clock (CK). The J input corresponds to S, and K corresponds to R. That is, if $J = 1$ and $K = 0$, the flip-flop output is set to $Q = 1$ after the active clock edge; and if $K = 1$ and $J = 0$, the flip-flop output is reset to $Q = 0$ after the active edge. Unlike the S-R flip-flop, a 1 input may be applied simultaneously to J and K, in which case the flip-flop changes state after the active clock edge. When $J = K = 1$, the active edge will cause Q to change from 0 to 1, or from 1 to 0. The next-state table and characteristic equation for the J-K flip-flop are given in Figure 11-24(b).

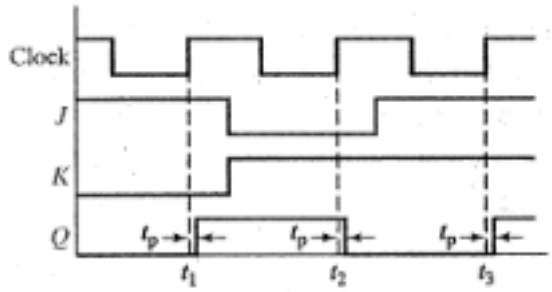


(a) J-K flip-flop

J K Q	Q^+
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	0

$$Q^+ = JQ' + K'Q$$

(b) Truth table and characteristic equation



(c) J-K flip-flop timing

FIGURE 11-24: J-K Flip-Flop (Q Changes on the Rising Edge)

© Cengage Learning 2014

Figure 11-24(c) shows the timing for a J-K flip-flop. This flip-flop changes state a short time (t_p) after the rising edge of the clock pulse, provided that J and K have appropriate values. If $J = 1$ and $K = 0$ when Clock = 0, Q will be set to 1 following the rising edge. If $K = 1$ and $J = 0$ when Clock = 0, Q will be set to 0 after the rising edge. Similarly, if $J = K = 1$, Q will change state after the rising edge. Referring to Figure 11-24(c), because $Q = 0$, $J = 1$, and $K = .0$ before the first rising clock edge, Q changes to 1 at t_1. Because $Q = 1$, $J = 0$, and $K = 1$ before the second rising clock edge, Q changes to 0 at t_2. Because $Q = 0$, $J = 1$, and $K = 1$ before the third rising clock edge, Q changes to 1 at t_3.

One way to realize the J-K flip-flop is with two S-R latches connected in a masterslave arrangement, as shown in Figure 11-25. This is the same circuit as for the S-R master-slave flip-flop, except S and R have been replaced with J and K, and the Q and Q' outputs are feeding back into the input gates. Because $S = J{\cdot}Q'{\cdot}CLK'$ and $R = K{\cdot}Q{\cdot}CLK'$, only one of S and R inputs to the first latch can be 1 at any given time. If $Q = 0$ and $J = 1$, then $S = 1$ and $R = 0$, regardless of the value of K. If $Q = 1$ and $K = 1$, then $S = 0$ and $R = 1$, regardless of the value of J.

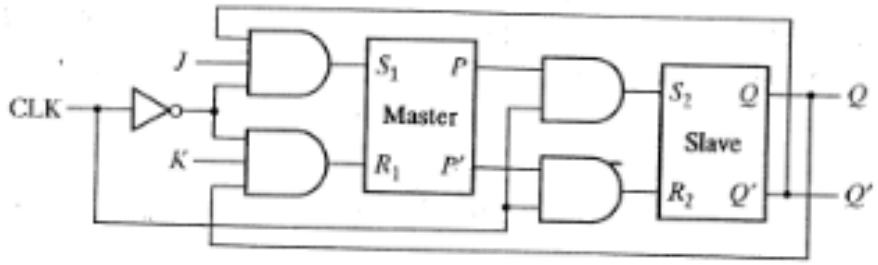
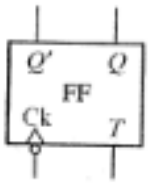


FIGURE 11-25: Master-Slave J-K Flip-Flop (Q Changes on Rising Edge)

T Flip-Flop

The T flip-flop, also called the toggle flip-flop, is frequently used in building counters. Most CPLDs and FPGAs can be programmed to implement T flip-flops. The T flip-flop in Figure 11-26(a) has a T input and a clock input. When $T = 1$ the flip-flop changes state after the active edge of the clock. When $T = 0$, no state change occurs. The nextstate table and characteristic equation for the T flip-flop are given in Figure 11-26(b). The characteristic equation states that the next state of the flip-flop (Q^+) will be 1 iff the present state (Q) is 1 and $T = 0$ or the present state is 0 and $T = 1$.



(a)

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

(b)

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

FIGURE 11-26: T Flip-Flop

Figure 11-27 shows a timing diagram for the T flip-flop. At times t_2 and t_4 the T input is 1 and the flip-flop state (Q) changes a short time (t_p) after the falling edge of the clock pulse. At times t_1 and t_3 the T input is 0, and the clock edge does not cause a change of state.

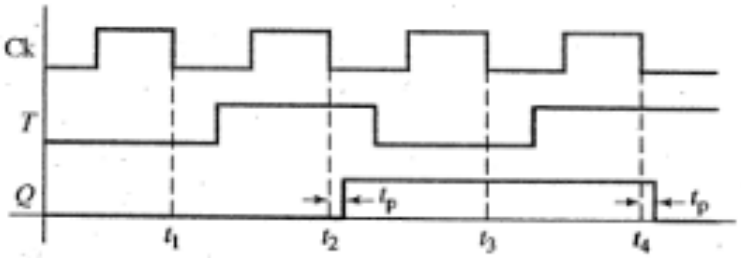
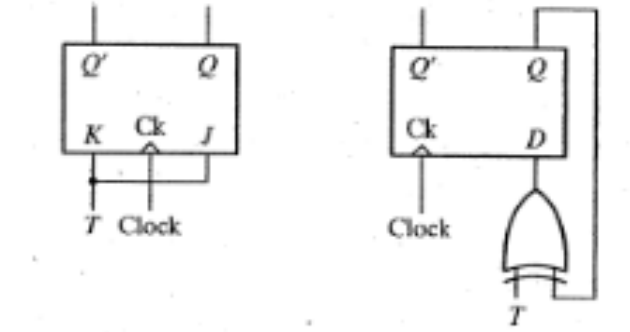


FIGURE 11-27: Timing Diagram for T Flip-Flop (Falling- Edge Trigger)

© Cengage Learning 2014



(a) Conversion of J-K to T (b) Conversion of D to T

FIGURE 11-28: Implementation of T Flip-Flops

© Cengage Learning 2014

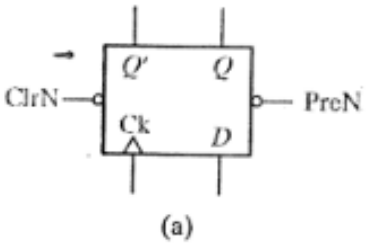
One way to implement a T flip-flop is to connect the J and K inputs of a J-K flip-flop together, as shown in Figure 11-28(a). Substituting T for J and K in the J-K characteristic equation gives

$$Q+ = JQ' + K'Q = TQ' + T'Q$$

which is the characteristic equation for the T flip-flop. Another way to realize a T flip-flop is with a D flip-flop and an exclusive-OR gate (Figure 11-28(b)). The D input is $Q \oplus T$, so $Q^+ = Q \oplus T = TQ' + T'Q$, which is the characteristic equation for the T flip-flop.

Flip-Flops with Additional Inputs

Flip-flops often have additional inputs which can be used to set the flip-flops to an initial state independent of the clock. Figure 11-29 shows a D flip-flop with clear and preset inputs. The small circles (inversion symbols) on these inputs indicate that a logic 0 (rather than a 1) is required to clear or set the flip-flop. This type of input is often referred to as *active-low* because a low voltage or logic 0 will activate the clear or preset function. We will use the notation ClrN or PreN to indicate active-low clear and preset inputs. Thus, a logic 0 applied to ClrN will reset the flip-flop to $Q = 0$, and a 0 applied to PreN will set the flip-flop to $Q = 1$.



(a)

Ck	D	PreN	ClrN	Q^+
x	x	0	0	(not allowed)
x	x	0	1	1
x	x	1	0	0
↑	0	1	1	0
↑	1	1	1	1
0,1,↓	x	1	1	Q (no change)

(b)

FIGURE 11-29: D Flip-Flop with Clear and Preset

These inputs override the clock and D inputs. That is, a 0 applied to the ClrN will reset the flip-flop regardless of the values of D and the clock. Under normal operating conditions, a 0 should not be applied simultaneously to ClrN and PreN. When ClrN and PreN are both held at logic 1, the D and clock inputs operate in the normal manner. ClrN and PreN are often referred to as *asynchronous* clear and preset inputs because their operation does not depend on the clock. The table in Figure 11-29(b) summarizes the flip-flop operation. In the table, ↑ indicates a rising clock edge, and X is a don't-care. The last row of the table indicates that if Ck is held at 0, held at 1, or has a falling edge, Q does not change. The clear and preset inputs can also be synchronous, i.e., the clear and set operations occur on the active edge of the clock. Often a synchronous clear or preset will override the other synchronous inputs, e.g., a D flip-flop with a synchronous clear input will clear on the active edge of the clock when the clear input is active independent of the value on D.

Figure 11-30 illustrates the operation of the clear and preset inputs. At t_1, ClrN = 0 holds the Q output at 0, so the rising edge of the clock is ignored. At t_2 and t_3, normal state changes occur because ClrN and PreN are both 1. Then, Q is set to 1 by PreN = 0, but Q is cleared at t_4 by the rising edge of the clock because $D = 0$ at that time.

In synchronous digital systems, the flip-flops are usually driven by a common clock so that all state changes occur at the same time in response to the same clock edge. When designing such systems, we frequently encounter situations where we want some flip-flops to hold existing data even though the data input to the flip-flops may be changing. One way to do this is to gate the clock, as shown in Figure 11-31(a). When $E_n = 0$, the clock input to the flip-flop is 0, and Q does not change. This method has two potential problems. First, gate delays may cause the clock to arrive at some flip-flops at different times than at other flip-flops, resulting in a loss of synchronization. Second, if En changes at the wrong time, the flip-flop may trigger due to the change in En instead of due to the change in the clock, again resulting in loss of synchronization. Rather than gating the clock, a better way is to use a flip-flop with a clock enable (CE). Such flip-flops are commonly used in CPLDs and FPGAs.

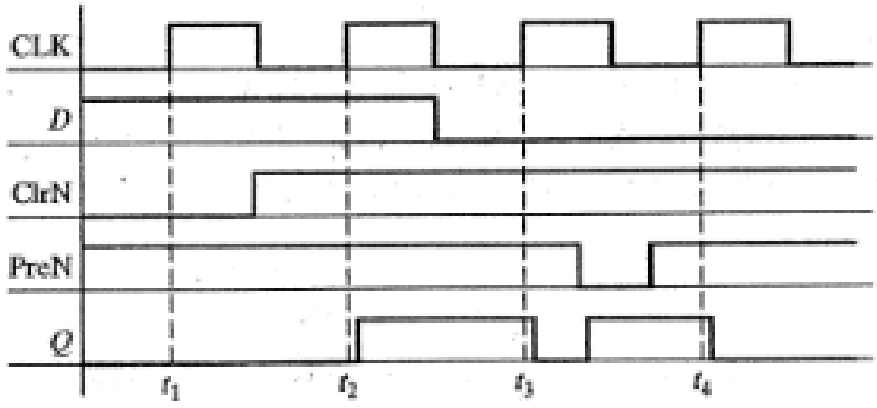
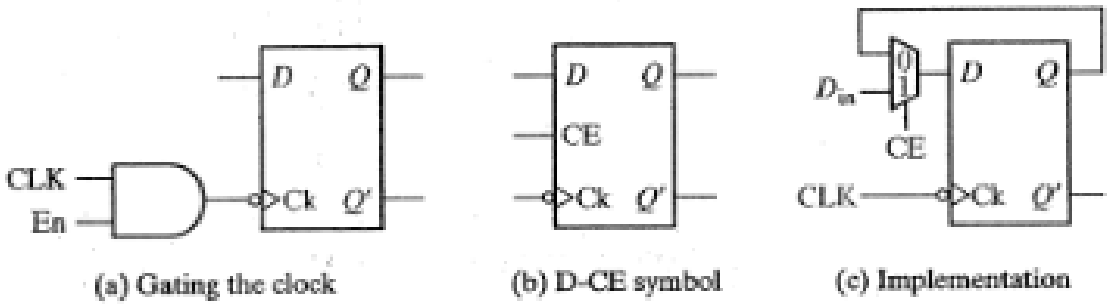


FIGURE 11-30: Timing Diagram for D Flip-Flop with Asynchronous Clear and Preset



(a) Gating the clock (b) D-CE symbol (c) Implementation

FIGURE 11-31: D Flip-Flop with Clock Enable

Figure 11-31(b) shows a D flip-flop with a clock enable, which we will call a D-CE flip-flop. When $CE = 0$, the clock is disabled and no state change occurs, so $Q^+ = Q$. When $CE = 1$, the flip-flop acts like a normal D flip-flop, so $Q^+ = D$. Therefore, the characteristic equation is $Q^+ = Q{\cdot}CE' + D{\cdot}CE$. The D-CE flip-flop is easily implemented using a D flip-flop and a multiplexer (Figure 11-31(c)). For this circuit, the MUX output is

$$Q+ = D = Q{\cdot}CE' + D_{in}{\cdot}CE$$

Because there is no gate in the clock line, this cannot cause a synchronization problem.

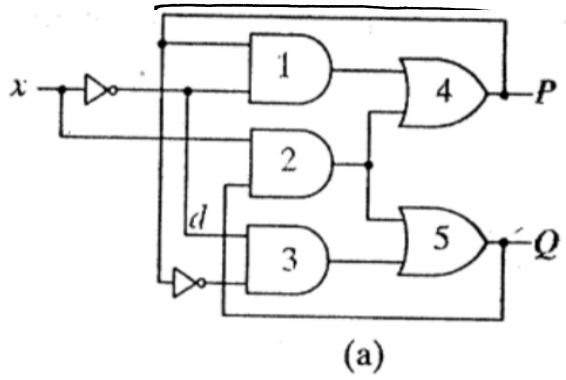
The characteristic equations for the latches and flip-flops discussed in this chapter are:

$Q^+ = S + R'Q\ (SR = 0)$	(S-R latch or flip-flop)
$Q^+ = GD + G'Q$	(gated D latch)
$Q^+ = D$	(D flip-flop)
$Q+ = D{\cdot}CE + Q{\cdot}CE'$	(D-CE flip-flop)
$Q+ = JQ' + K'Q$	(J-K flip-flop)
$Q+ = T \oplus Q = TQ' + T'Q$	(T flip-flop)

Asynchronous Sequential Circuits:

- In Sequential circuit, feed back signals defines the state of the circuit.
- In Asynchronous sequential circuits the state of the circuit can change whenever any input changes (like latches).
- In order for asynchronous circuits to operate in a well defined manner, they must satisfy several restrictions.

Consider the circuit



(a)

PQ	x = 0	x = 1
00	01	(00)
01	(01)	11
11	10	(11)
10	(10)	00

(b)

FIGURE 11-32: Asynchronous Circuit

- Since 2 feed back loops four states.
- Break the feed back loop at outputs of gate 4 and 5. Label the gate outputs as P^+ & Q^+

∴ next State Equations,

$$P^+ = x'P + xQ$$

$$Q^+ = x'P' + xQ$$

- Both Equation contains Static 1 Hazards

P^+ contain Static-1 Hazard for $xPQ = 111 \leftrightarrow 011$

(b) shows circuits next-State table with Stable states circled.

Consider $xPQ = 111$ and x changing to 0.

from table $PQ = 11$ & $x: 1 \to 0$

P does not change & Q changes to 0.

Assuming inverter for x has a delay large compared to gate delays.

- output of Gate 2 changes to 0 causing outputs of Gate 4 and 5 to change to 0.
- when inverter output changes to 1, P=0 prevents output of Gate 1 from changing to 1 so P remains 0.
- Gate 3 changes to 1 and Q becomes 1
- Circuit Stabilizes in State P=01.

This incorrect State transition causes Static 1 Hazard in P^+. This problem can be eliminated by designing the circuits to be free of Hazards.

- Even if the circuit is free of Hazards, delays in "wrong" places in the circuit can cause incorrect State transitions.
- The incorrect operation can be detected by examining next state table.

  ⇒ If next State after a single change in an input is different from the next state after Three changes in that input. The table is said to contain <u>essential hazard</u>.

Eg:- • Starting in total Stable State $xPQ = 010$ } next State after changing $x \to 1$ produces $xPQ = 100$ } <u>single change</u>

- next State after 3 changes

  (i) $xPQ = 010$ Change1 x to 1 $xPQ = 100$ $PQ = 00$

  Change2 x to 0 $xPQ = 001$

next-State after <u>3 changes</u>: Change3 x to 1 $xPQ = 111$ $PQ = 11$

- Essential Hazards are properties of next-State table, they cannot be eliminated by modifying the circuit's logic.
- To Prevent essential hazards it is necessary to control the delays in the circuit.

  For above <u>example</u> delays should be inserted at gate 4 and 5 where we have feedback loops.

Qu) <u>Explain race problem when two (or more inputs) to an asynchronous circuit ~~can~~ is changed at the same time.</u>

or

<u>Explain Multiple input change in Asynchronous circuit with example.</u>

<u>Consider next-State table</u>

PQ \ xy	00	01	11	10
00	(00)	(00)	01	(00)
01	11	00	(01)	11
11	(11)	10	(11)	(11)
10	00	(10)	11	00

FIGURE : Multiple Input Change Example

- Consider starting in total state $xyPQ = 1011$

Changing both x & y to 0 at same time $PQ = 11$ (From table)

- If y change propogates first, $xyPQ = 1011$

Then, x change propogates total State $xyPQ = 0011$

- If x change Propogates first, $xyPQ = 0011$

Then, y change Propogates total State, $xyPQ = 0000$.

Final States depends upon which input change propogates fastest and hence final state may not be $PQ = 11$ as indicated in next-state table.

Multiple-State variable change Example

- Similar race problem can occur if more than one state variable changes at same time.

Consider next-state table

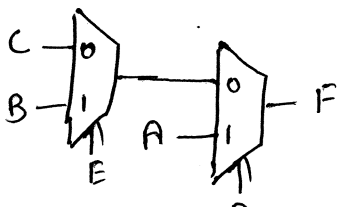
PQR	x = 0	x = 1
000	(000)	011
001	101	(001)
011	101	(011)
010	000	011
110	000	(110)
111	101	(111)
101	(101)	110
100	(100)	110

- The total State $xPQR = 0000$ has next State $PQR = 011$ when $x: 0 \to 1$.
- If Q changes first, circuit will have transition to state $PQR = 010$ & then to $PQR = 011$.
- However, If R changes first the circuit take transition to state $PQR = 001$ & may stabilize here.
- There is a race between Q and R changes and since circuit can enter two different states depending which signal changes first, The race is called a critical race.
- If total State $xPQR = 1011$ & $x: 1 \to 0$ circuit will enter state $PQR = 101$ no matter which variable changes first. so the race is not critical.
- It is not possible to fix the problem with races between state variables by inserting delay. At best asynchronous circuits must not contain any critical races.

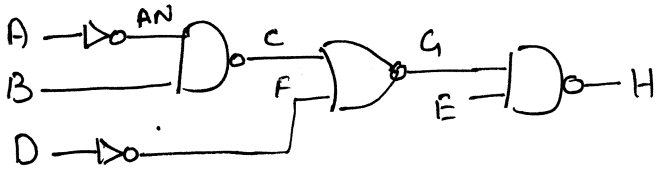
Module 4 Exercise Problems (VHDL)

1) Implement the following VHDL conditional statement using two 2-to-1 MUXes.

F <= A when D = '1' else B when E = '1' else C;



2) Write VHDL single concurrent statement to represent the following circuit. Do not use parantheses in statement



(b) Write individual statements to represent the circuit of part (a). Assume all NAND gates have delay 10ns & all NOR gates have delay 15ns & inverters delay of 5ns.

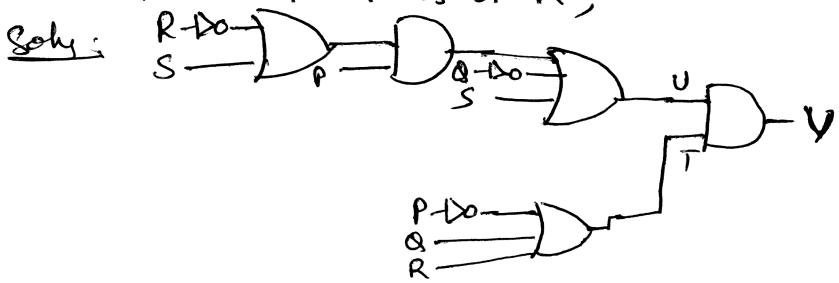
(a) H <= not A nand B nor not D nand E;

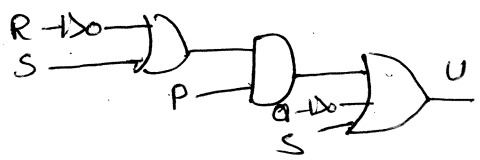
(b) AN <= not A after 5ns;
C <= AN nand B after 10ns;
F <= not D after 5ns;
G <= C nor F after 15ns;
H <= G nand E after 10ns;

3) Draw a circuit that implements the following VHDL code

V <= T and U;
U <= not R or S and P or not Q or S;
T <= not P or Q or R;

Soln:





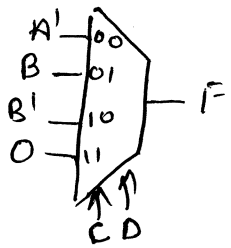
4) Simple Tristate Buffer

B=1 C=A

B=0 C=Z

```
signal A, B, C : std_logic
C <= A when B = '1' else 'Z';
```

5) Write a selected signal assignment statement to represent the 4 to 1 mux shown below. Assume that there is an inherent delay in the MUX of 15ns. Use conditional signal assignment statement.




```
begin
  with C&D select
  F <= not A after 15ns when "00",
       B after 15ns when "01",
       not B after 15ns when "10",
       '0' after 15ns when "11";
```