DIGITAL SYSTEM DESIGN USING VERILOG

COURSE CODE: 19EC5DCDSV
(3 CREDITS)

Faculty In-Charge: Dr. Dinesha P

drdinesh-ece@dayanandasagar.edu

DATAFLOW MODELLING

Features :

- Dataflow provides a powerful way to implement a design.
- Automated tools are used to crate a gate level circuit from the description of a dataflow design, called logic synthesis.
- Register transfer level (RTL) is a combination of dataflow and behavioral modeling.
- A continuous assignment is the most basic statement of dataflow modeling.

Continuous assignment:

This statement continuously drives a valve onto a net. This statement begins with the keyword assign. The syntax is

Assign net__1value=expression;

Where net__1value is a scalar or vector net or their concatenation

Assign net1 = expr1, net2= expr2, …, netn=exprn;

This statement is a compact form of n continuous assignments.

.

- Operands in this expressions can be variables or nets or function calls. Variables or net can be a scalar or vectors.
- Continuous assignments are always active. Whenever an operand in the right hand side expression changes value, the expression is evaluated and the new value is assigned to net_1value.

Example:

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

The above statement perform a 4 bit addition with carry input.

{ } is the concatenation operator.

Net declaration assignment: net data types may be a wire.

Example:

```
wire out;    // normal continuous assignment
Assign out = in1 & in2;
```

Which is equivalent to

```
Wire out = in1 & in2;
```

Implicit net declaration:

- Implicit net declaration is a feature of Verilog HDL

Example:

```
Wire   in1, in2;
Assign out = in1 & in2;
```

In the above, out is not declared as a wire, but an implicit wire declaration for out is done by simulator.

Expressions: The dataflow modeling using expression, that consists of sub expressions, operators and operands instead of primitive gates or modules used in the structural styles.

Expressions = operators +operands

The operand may be a constants, net, reg, integer, real, real time, bit select, part select, array element and call function

Delays:

- The time between any operand in the right-hand side changes and when the new value of the right-hand side is assigned to the left-hand side controlled by the delay value.
- The delay is specified after the key word assign

Syntax:

```
// for continuous assignment statement
    assign #delay net_1value = expression;
            #delay net_1value = expr1,
             #delay net_1value = expr2,
                        ……
            #delay net_1value = exprn;

// for wire
Wire in1,in2,out;

Assign #10 out  =  in1 & in2; // delay is 10 time units
```

- net declaration continuous assignment with delay

wire #10 out = in1 & in2;

- // regular continuous assignment with delay

Wire out;

Assign #10 out = in1 & in2;

Net declaration delays can also be used in gate level modeling

```
// net delays
wire #10 out;
assign out = in1 & in2;

// regular assignment delay
wire out;
assign #10 out = in1 & in2;

// regular assignment delay
wire #5 out;
assign #10 out = in1 & in2;
```

Introduction to Verilog:

Value set: The Verilog supports four-value logic system

0: logic 0, false condition

1: logic 1, true condition

x: unknown logic value

z: high impedance

Data types:

Verilog HDL has classes of data types: nets and variables. Nets mean any hardware connection points, and variable represent any data storage elements. A variable represents a data storage element. The variable data types include five different kinds: reg, integer, time, real and real-time.

Net data types:

The nets represent physical connections between structural entities, such as gates and modules. A net does not store a value.

Syntax:

Net_type [signed] [(msb : lsb)] net1, net2…….netn;

Net type may be wire or tri or wand or triand etc., generally wire keyword is used to declare the net.

reg variable data types: A reg variable holds a value between assignments; it can be used to model hardware registers such as edge sensitive and level sensitive storage elements.

A form of reg declaration is a s follows

reg [signed] [(msb : lsb)] reg1, reg2,.....regn;

Vector: Multiple bit width

OPERANDS

- The operands in an expression can be of any constants , parameters, nets, variables (reg, integer, time, real, real time), bit select, part select, array element and function call.
- Parameters is a constant and declared using a parameter or localparam declaration.
- A function call may be either a user defined function or a system function.

Constants: There are 3types of constant in Verilog HDL. These are integer, real and string.

Integer constants: Integer constants can be specified in decimal, hexadecimal, octal or binary format. There are two ways to express integer constants: simple decimal form and base format notation.

Simple decimal form: In this, a number is specified as a sequence of digits 0 through 9, with plus (+) or minus (-) unary operator.

Example: -123 // is decimal -123

1234 // is decimal 1234

Integers values represented in signed form. A negative number is represented in two's complement form.

Base format notation: In this representation, the integer is composed of 3 parts: an optional size constant, a single quote followed by a base format character and the digits representing the value of the number.

[size] ′ [s/S] [base_format] base value

Example:

4 ′ b1001 // a 4 bit binary number

16 ′ habcd // a 16 bit hexadecimal number

2006 // unsized number

4 ′ sb1001 // a 4 bit signed number, it represents -7

-4 ′ sb1001 // a 4 bit signed number, it represents -(-7) =7

Real constants: Real numbers can be specified in either decimal notation or in scientific notation. Real numbers expressed with a decimal point must have at least one digit on each side of the decimal point.

Example: 1.5 //

1256.342 //

String constant:

A string is a sequence of characters enclosed by double quotes (“ “) and may not be split into multiple lines. One character is represented as an8 bit ASCII code. Hence, a string is considered as an unsigned integer constant represented by a sequence of 8 bit ASCII codes.

Bit select and part select:

- A bit select extracts a particular bit from a vector.

Syntax of bit select is

vector_name [bit_selsct_expr]

Where vector_name can be any vector of net, reg, integer and time data types. The bit_selsct_expr can be an expression.

- In part select, a contiguous sequence or **subsequence** of a bit vector is selected.

Syntax: a constant part select

vector_name [msb_const_expr : lsb_const_expr]

Verilog Operators:

Verilog operators are similar to the operators used in C language. Predefined Verilog operators can be grouped into several classes:

1. Unary sign and reduction operators:

1 2 Unary sign operators & Reduction and (unary operator to and bits in a vector and reduce to one bit) ~& Reduction nand | Reduction or (unary operator to or bits in a vector and reduce to one bit) ~| Reduction nor ^ Reduction x

Array and memory elements:

- Arrays can be used to group elements into multi-dimensional objects.
- We know, only nets and reg can be declared as vectors, all net and reg data types are allowed to be declared as multi-dimensional arrays.
- A multi-dimensional array is declared by specifying the address ranges after the declared identifier, called dimension, one for each.
- An array element can be a scalar or a vector if the element is net or reg data type.

Syntax:

net_type [signed] [range] id [msb:lsb] {[msb: lsb]};

reg [signed] [range] id [msb:lsb] {[msb: lsb]};

integer id [msb:lsb] {[msb: lsb]};

Time id [msb:lsb] {[msb: lsb]};

Memory:

- Memory is a basic module in any digital system; in Verilog HDL, it is declared as a one-dimensional reg array.
- A memory can be used to model a read-only memory (ROM), a random access memory (RAM) and a register file.
- To access a memory word, use the following format

mem_name [addr_expr]

Where addr_expr can be any expressions. For example

reg [7:0] mema [7:0]; //one dimensional array of 8 bit vector

Operators:

Verilog HDL has a rich set of operators, including arithmetic operators, bit wise operators, logical operators, concatenation and replication operators, relational operators, equality operators, shift operators, and conditional operators.

Bit wise operators: Bit wise operators perform a bit by bit operation on two operands and produce a vector results. In bit wise operators, a z is treated as unknown x,. When two operands are not equal length, the shorter operand is zero-extended to match of the loner operand.

Symbol	Operation
~	Bitwise negation
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
~^,^~	Bitwise exclusive nor

TABLE 3.1 The summary of operators in Verilog HDL

Arithmetic	Bitwise	Reduction	Relational
+: add	~: NOT	&: AND	>: greater than
−: subtract	&: AND	|: OR	<: less than
*: multiply	|: OR	~ &: NAND	>=: greater than or equal
/: divide	^: XOR	~ |: NOR	<=: less than or equal
%: modulus	~^, ^~: XNOR	^: XOR	**Miscellaneous**
**: exponent	**Case equality**	~^, ^ ~ : XNOR	{ , }: concatenation
Shift	===: equality	**Logical**	{c{ }}: replication
<<: left shift	! ==: inequality	&&: AND	? : conditional
>>: right shift	**Equality**	||: OR	
<<<: arithmetic left shift	==: equality	!: NOT	
>>>: arithmetic right shift	! =: inequality		

TABLE 3.2 The precedence of operators in Verilog HDL

Operators	Symbols	Precedence
Unary	+ ! ~	Highest
Exponent	**	
Multiply, divide, modulus	* / %	
Add, subtract	+−	
Shift	<< >> <<< >>>	
Relational	< <= > >=	
Equality	== ! = === ! ==	
Reduction	& ~ &	
	^ ^ ~	
	| ~|	
Logical	&&	
	||	
Conditional	?:	Lowest

The bit-wise operators include five operators: & (and), |(or), ^ (xor), ^~(xnor) and ~(negation), as shown in Table 3.3. The functions of these operators are as follows:

- & (and): if any bit is 0, the result is 0, or else if both bits are 1, then the result is 1; otherwise the result is an x.
- | (or): if any bit is 1, the result is 1, or else if both bits are 0, the result is 0; otherwise the result is an x.
- ^ (xor): if one bit is 1 and the other is 0, the result is 1, or else if both bits are 0 or 1, the result is 0; otherwise the result is an x.
- ^~(xnor): if one bit is 1 and the other is 0, the result is 0, or else if both bits are 0 or 1, the result is 1; otherwise the result is an x.
- ~(negation): if the input bit is 1 the result is 0, or else if the input bit is 0, the result is 1; otherwise the result is an x.

Dataflow model for a 4:1 multiplexer

```
module mux41_dataflow(i0, i1, i2, i3, s1, s0, out);
// port declarations
input i0, i1, i2, i3;
input s1, s0;
output out;
// using basic and, or, not logic operators.
   assign out = (~s1 & ~s0 & i0) |
                (~s1 &  s0 & i1) |
                ( s1 & ~s0 & i2) |
                ( s1 &  s0 & i3) ;
endmodule
```

Procedural Assignments:

There are two types of procedural assignments in Verilog. Initial blocks execute only once at time zero, whereas always blocks loop to execute over and over again.

In other words, the initial block execution and always block execution starts at time 0. Always block waits for the event, whereas initial block just executes all the statements without waiting.

Initial Statements

An initial statement has the following basic form:

```
initial
begin
sequential-statements
end
```

Always Statements:

An always statement has the following basic form:

```
always @(sensitivity-list)
 begin
 sequential-statements
end
```

When an always statement is used, the statements between the **begin** and the **end** are executed sequentially rather than concurrently. The expression in parentheses after the word always is called a **sensitivity list**, and the process executes whenever any signal in the sensitivity list changes. The symbol "@" should be used before the sensitivity list.

For example, if the always statement has the sensitivity list @(A, B, C), then it executes whenever any one of A, B, or C changes. Whenever one of the signals in the sensitivity list changes, the sequential statements in the always block are executed in sequence one time.

- The variables on the left-hand side element of an = or ,<= in an always block should be defined as **reg** data type. Any other data type including wire is illegal.
- The assignment operator "=" indicates concurrent execution when used outside an always block.

When the statements

C = A && B; // concurrent statements

E = C || D; // when used outside always block

are used outside an always block, the order of the statements does not matter. But when used in an always block, they become sequential statements executed in the order they are written.

Blocking and Non-Blocking Assignments:

- Sequential statements can be evaluated in two different ways in Verilog—**blocking assignments** and **non-blocking assignments**.
- A blocking statement must complete the evaluation of the right-hand side of a statement before the next statements in a sequential block are executed.
- Operator "=" is used for representing the blocking assignment. The meaning of "blocking" is that a blocking assignment has to complete before the next statement starts execution (i.e., it blocks the next assignment in the sequential block from starting evaluation).
- A non-blocking statement allows assignment evaluation without blocking the sequential flow. **In other words, several assignments can be evaluated at the same time**. Operator " <= " is used for representing the non-blocking assignment.

For example:

```
always @(A, B, D)
begin
C = A && B; // Blocking operator is used
E = C || D; // Statements execute sequentially
end
```

The block executes once when any of the signals A, B, or D changes. The first statement updates the value of C before the second statement starts execution; hence, the second statement uses the new value of C as input. If C or E changes when the block executes, then the always block will not execute a second time because C is not on the sensitivity list.

The operator **"<="** is to evaluate several assignments at the same time without blocking the sequential flow. Consider the following code:

```
always @(A, B, D)
begin
C <= A && B; // Statements execute simultaneously
                    because
E <= C || D; // non-blocking operator is used
end
```

The block executes once when any of the signals A, B, or D changes. Both statements execute simultaneously with the values of A, B, C, and D at the beginning of the always block. The first statement does not update the value of C before the second statement starts execution; hence, the second statement uses the old value of C as input. If C changes when the block executes, then the always block will not execute a second time because C is not on the sensitivity list.

Operator "<=" is used for representing the non-blocking assignment inside an always statement.

- The following example shows a comparison of blocking assignments and non-blocking assignments.
- The posedge keyword is used for an edge-triggered functionality in the sensitivity list.
- Signals A and B are used for a blocking statement, and C and D are applied for a non-blocking statement.
- Assume the initial values of input signals are A=C=1'b1 and B=D=1'b0. In the case of blocking assignments, both A and B will become 1'b0.
- Since the second assignment will not be evaluated until the completion of the first assignment, the newly evaluated A signal will be assigned to B in the second assignment.
- On the other hand, non-blocking assignments will start evaluating all statements in a sequential block at the same time; the result will be independent of the assignment order. Therefore, the result of non-blocking assignments will be C=1'b0 and D=1'b1. So the signals swap in the case of C and D, but not in the case of A and B.

Example: Blocking and Non-Blocking Assignments

```
module sequential_module (A, B, C, D, clk);
input clk;
 output A, B, C, D;
reg A, B, C, D;
always @(posedge clk)
begin
A = B; // blocking statement 1
 B = A; // blocking statement 2
end
always @(posedge clk)
begin
C <= D; // non-blocking statement 1
 D <= C; // non-blocking statement 2
end
endmodule
```

- Always statements can be used for modeling combinational logic and sequential logic; however, always statements are not necessary for modeling combinational logic.
- They are, required for modeling sequential logic.
- Should be very careful when using always statements to represent combinational logic.
- If any of the input signals are accidentally omitted from the sensitivity list, there can be mismatches between synthesis and simulation and a lot of confusion.
- Hence, the common practice is of using the always @* statement if a combinational circuit is desired, which avoids accidental errors such as these.
- If the sensitivity list is "*" then the block will get triggered for any input signal changes.

- Consider the code shown in the following example, where an always statement is used to model two cascaded gates.
- D and E should be defined as reg. Also, D should be defined as output if the output of the first gate is desired externally. If inout is used for D, you will have a compile error, since D is of reg type. Normally, **input and inout ports can be only net (or wire) data type**. The statement order is important here because blocking assignment is used.

Verilog Code for Combinational Logic with Blocking Assignments in an Always Block:

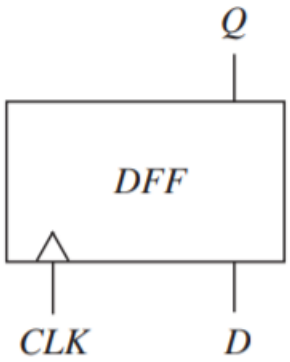
```
module two_gates (A, B, C, D, E);
 input A, B, C;
output D, E;
reg D, E;
always @(*)
Begin
 #5 D = A | | B; // blocking statement 1
 #5 E = C | | D; // blocking statement 2
 end
endmodule
```

Sensitivity List: Both combinatorial always blocks and sequential always blocks have a sensitivity list that includes a list of events. An always block will be activated if one of the events occurs. In the combinatorial logic, the sensitivity list includes all signals that are used in the condition statement and all signals on the right-hand side of the assignment. On the other hand, the sensitivity list in sequential circuit contains three kinds of edge-triggered events: clock, reset, and set signal event.

The sensitivity list can be specified using @(*) for a combinational circuit, indicating that the block must be triggered for any input signal changes. If sensitivity list is omitted at the always keyword, delays or time-controlled events must be specified inside the always block.

Modeling Flip-Flops Using Always Block:

A flip-flop can change state either on the rising or on the falling edge of the clock input. This type of behavior is modeled in Verilog by an always block.




```
always @ (posedge CLK)
begin
   Q <= D;
end
```

In above fig, on the rising edge of CLK, the always block executes once through and then waits at the start of the block until CLK changes again.

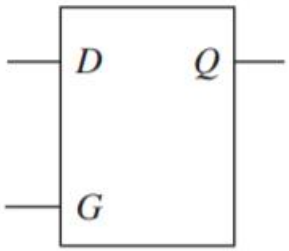
The sensitivity list of the always block tests for a rising edge of the clock, and Q is set equal to D when a rising edge occurs.

The expression posedge (or negedge) is used to accomplish the functionality of an edge-triggered device.

If CLK is changed from 0 to 1 it is a rising edge. If CLK changes from 1 to 0, it indicates a falling edge.

- If the flip-flop has a delay of 5ns between the rising edge of the clock and the change in the Q output, we would replace the statement Q <= D; with Q <= #5 D; in the foregoing always block.
- The statements between begin and end in an always block operate as sequential statements. In the previous example, Q <= D; is a sequential statement that executes only following the rising edge of CLK.

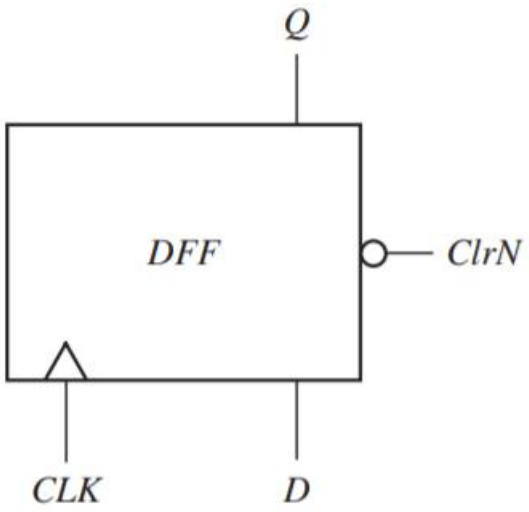
Verilog Code for a Transparent Latch:




```
always @ (G or D)
begin
 if (G)
    Q <= D;
end
```

Both G and D are on the sensitivity list since if G = 1, a change in D causes Q to change.
If G changes to 0, the always block executes, but Q does not change.
For the sensitivity list, both (G or D) and (G, D) are acceptable.

Verilog Code for a D Flip-Flop with Asynchronous Clear:




```
always @ (posedge CLK or negedge ClrN)
begin
  if (~ClrN)
    Q <= 0;
  else
    Q <= D;
end
```

- Flip-flop has an active-low asynchronous clear input (ClrN) that resets the flip-flop independently of the clock. Therefore Flip-flop executes when either CLK or ClrN changes.
- Since the asynchronous ClrN signal overrides CLK, ClrN is tested first and the flip-flop is cleared if ClrN is 0. Otherwise, CLK is tested, and Q is updated if a rising edge has occurred.

In the above examples, we have used two types of sequential statements: signal assignment statements and if statements.

The basic if statement has the form

```
if (condition)
 sequential statements1
else
 sequential statements2
```

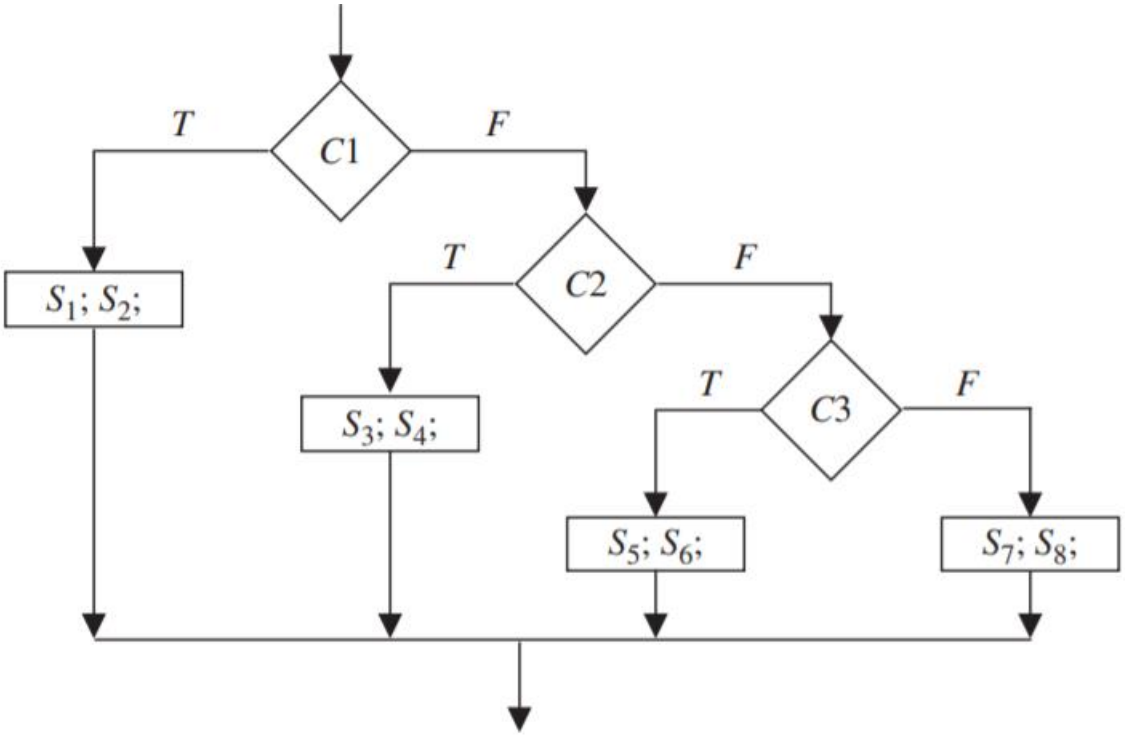
The condition is a Boolean expression that evaluates to TRUE or FALSE. If it is TRUE, sequential statements1 are executed; otherwise, sequential statements2 are executed.

Verilog if statements are sequential statements that can be used within an always block (or an initial block), but they cannot be used as concurrent statements outside of an always block. The most general form of the if statement is

```
if (condition)
sequential statements
 // 0 or more else if clauses may be included
else if (condition)
sequential statements}
[else sequential statements]
```

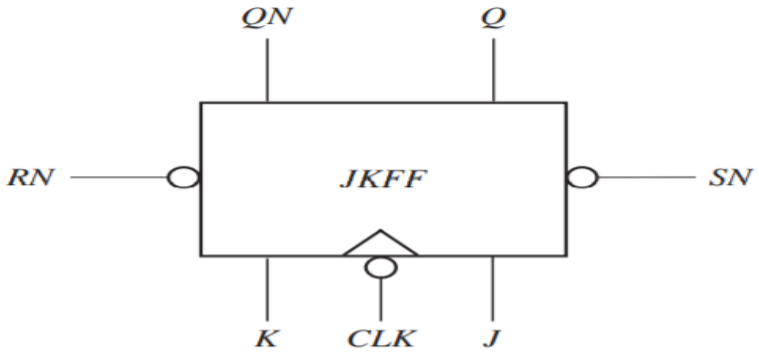

The curly brackets indicate that any number of else if clauses may be included, and the square brackets indicate that the else clause is optional.

The following example shows how a flow chart can be represented using nested if's or the equivalent using else if's. In this example, C1, C2, and C3 represent conditions that can be true or false, and S1 , S2 , . . . , S8 represent sequential statements. If more than one statement needs to be in an if block, begin and end should be used.




```
if (C1)
begin
   S1; S2;
end
else if (C2)
begin
   S3; S4;
end
else if (C3)
begin
   S5; S6;
end
else
begin
   S7; S8;
end
```

J K Flip-Flop:




```
module JKFF (SN, RN, J, K, CLK, Q, QN);
input  SN, RN, J, K, CLK;
output Q, QN;

reg Qint;

always @(negedge CLK or RN or SN)
begin
  if (~RN)
    #8  Qint <= 0;                                  // statement1
  else if (~SN)
    #8  Qint <= 1;                                  // statement2
  else
    Qint <= #10 ((J && ~Qint) || (~K && Qint)); // statement3
end

assign Q = Qint;                                    // statement4
assign QN = ~Qint;                                  // statement5

endmodule
```

In the above code,

define a reg Qint as an internal signal that represents the state of the flip-flop internal to the module.

The two concurrent statements, statement4 and statement5, transmit this internal signal to the Q and QN outputs of the flip-flop.

Because the flip-flop can change state in response to changes in SN, RN, and CLK, these three signals are in the sensitivity list of the always statement.

Both RN and SN are active low signals. If RN = 0, the flip-flop is reset, and if SN = 0, the flip-slop is set.

Since RN and SN, reset and set the flip-flop independently of the clock, they are tested first. If RN and SN are both 1, we test for the falling edge of the clock. In the if statement, both (~RN) and (RN == 1'b0) are acceptable.

The condition (negedge CLK) is TRUE only if CLK has just changed from 1 to 0. The next state of the flip-flop is determined by its characteristic equation:

$$Q1 = JQ' + K' Q$$

Always Blocks Using Event Control Statements:

An alternative form for an always block uses wait or event control statements instead of a sensitivity list. If a sensitivity list is omitted at the always keyword, delays or time controlled events must be specified inside the always block. For example,

```
always
begin
#10 clk <= ~clk;
end
```

Such an always block could look like

```
Always
 begin
rst = 1;
// sequential statements
@(posedge CLK); //wait until posedge CLK
// more sequential statements
end
```

- This always block will execute the sequential-statements until a wait (event control) statement is encountered. Then it will wait until the specified condition is satisfied. It will then execute the next set of sequential-statements until another wait is encountered. It will continue in this manner until the end of the always block is reached. Then it will start over again at the beginning of the block.
- The wait statement is used as a level-sensitive event control. The general syntax of the wait statement is

  wait (Boolean-expression)
- A procedural statement waits when the Boolean expression is FALSE. When the expression is TRUE, the statement is executed. The logic values 0, 'x', and 'z' are treated as FALSE. Logic 1 is TRUE. The following example will block the flow of the procedural block when the condition of the wait statement is FALSE.

```
always
begin
wait (WR) MEM = DATA_IN;
wait (~WR) DATA_OUT = MEM;
end
```

Delays in Verilog:

```
assign #5 D = A && B;
```

to model an AND gate with a propagation delay of 5ns (assuming its time unit is ns).

Basically, delays in Verilog can be categorized into two models: inertial delay and transport delay. The inertial delay for combinational blocks can be expressed in the following three ways:

```
// explicit continuous assignment
wire D;
assign #5 D = A && B;
// implicit continuous assignment
wire #5 D = A && B;
// net declaration
wire #5 D;
assign D = A && B;
```

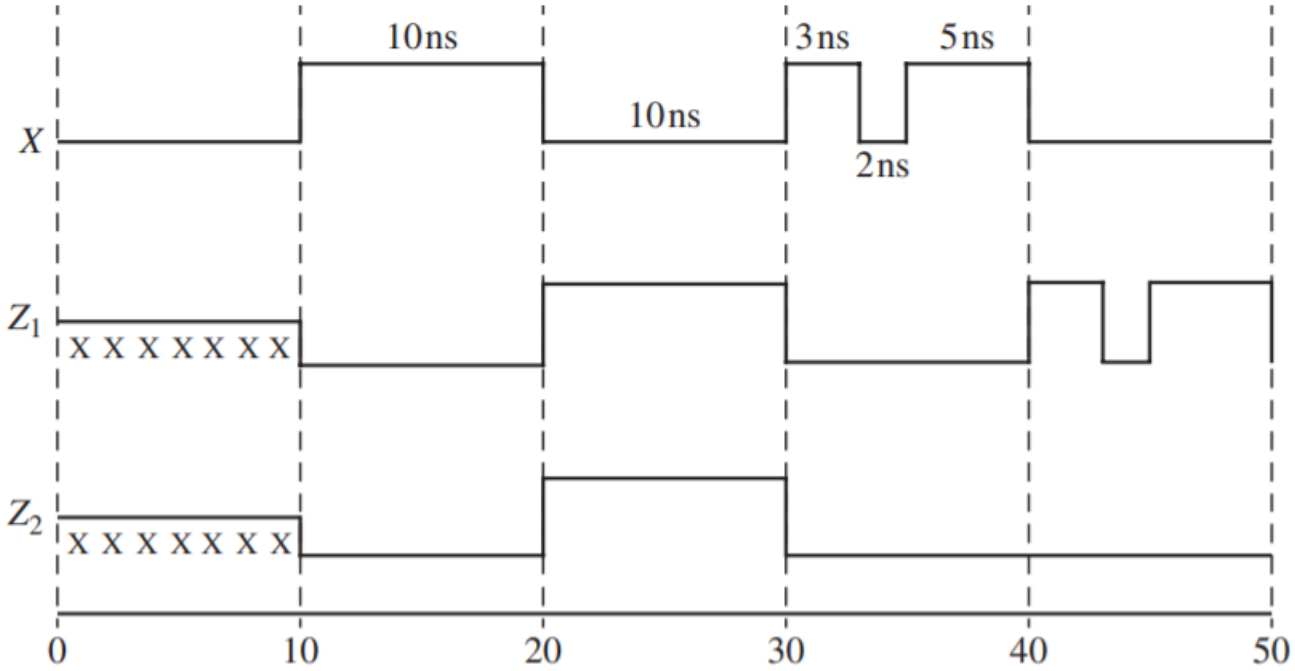
Any changes in A and B will result in a delay of 5ns before the change in output is visible. If values in A or B are changed 5ns before the evaluation of D output, the change in values will be propagated. However, an input pulse that is shorter than the delay of the assignment does not propagate to the output. This feature is called inertial delay.

Inertial delay is used to model gates and other devices that do not propagate short pulses from the input to the output. If a gate has an ideal inertial delay T, in addition to delaying the input signals by time T, any pulse with a width less than T is rejected. For example, if a gate has an inertial delay of 5ns, a pulse of width 5ns would pass through, but a pulse of width 4.999ns would be rejected.

Transport delay is intended to model the delay introduced by wiring; it simply delays an input signal by the specified delay time. In order to model this delay, a delay value must be specified on the right-hand side of the statement.

The following example illustrates transport delay and inertial delay in Verilog

```
always @ (X)
begin Z1 <= #10 (X);  // transport delay
end
assign #10 Z2 = X;      // inertial delay
```



- The first statement has transport delay while the second one has inertial delay. As shown in Figure, if the delay is shorter than 10ns, the input signal will not be propagated to the output in the second statement.
- Only one pulse (between 10ns and 20ns) on input X is propagated to the output Z2 , since it has 10ns pulse width. All other pulses are not propagated to the output Z2 .
- But Z1 has transport delay and hence propagates all pulses.
- It is assumed that the output Z1 and Z2 are initialized to 0 at 0ns.
- The delay in the statement Z1 <= #10 X; is called intra-assignment delay.
- The expression on the right hand side is evaluated but not assigned to Z1 until the delay has elapsed (also called delayed assignment). However, in a statement like #10 Z1 <= X; the delay of #10 elapses first and then the expression is evaluated and assigned to Z1 (also called delayed evaluation).

```
assign a = #10 b;
```

Verilog also has a type of delay called net delay.

Consider the code

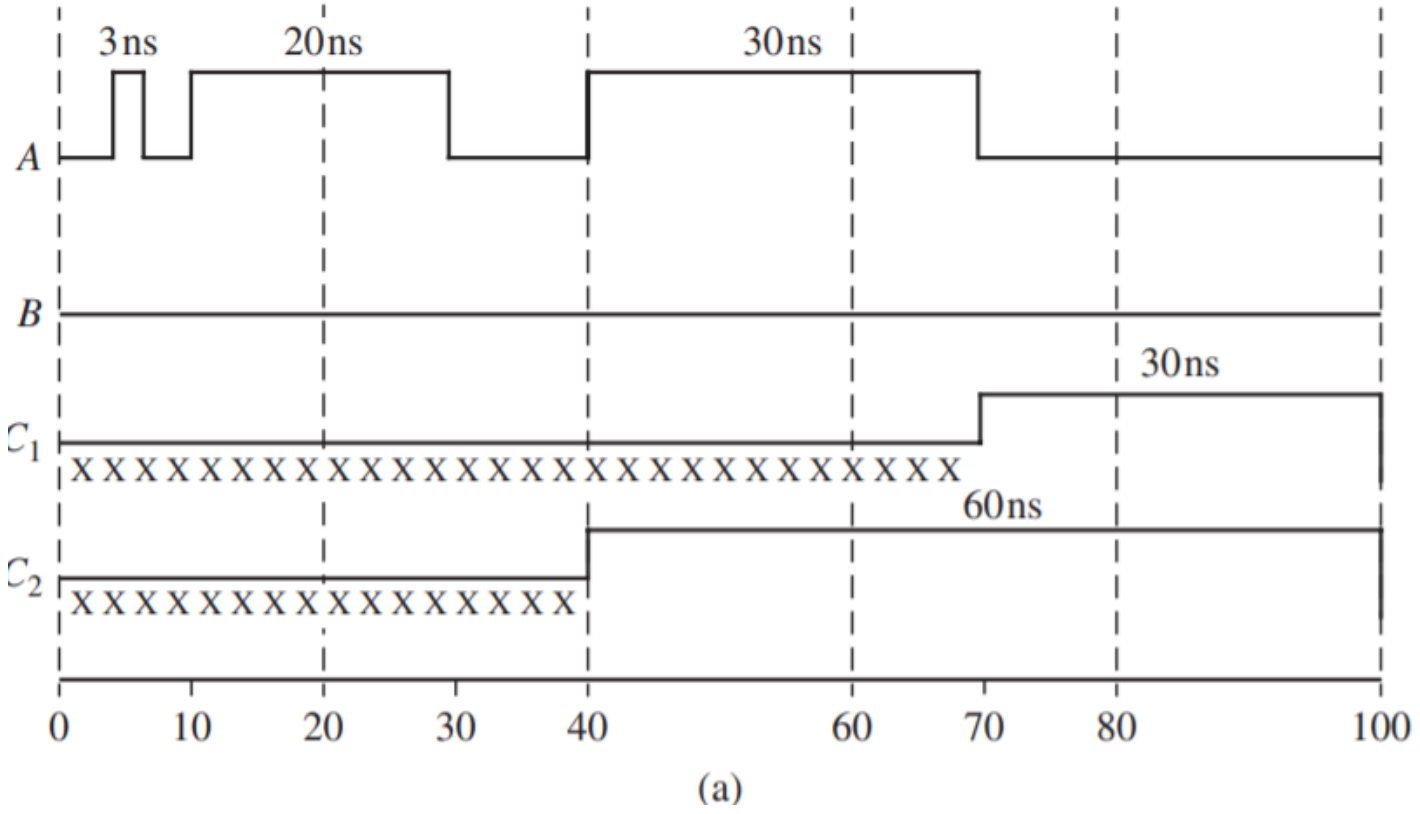
```
wire C1;
wire #10 C2; // net delay on wire C2
assign #30 C1 = A || B;    // statement 1 – inertial delay
assign #20 C2 = A || B;    // statement 2 – inertial delay will
                                be
                            // added to net delay before being
                            // assigned to wire C2
```

The wire C2 has a net delay of 10ns, whereas C1 has no such net delay.

Net delay refers to the time it takes from any driver on the net to change value to the time when the net value is updated and propagated further.

There are inertial delays of 30ns for C1 in statement 1 and 20ns for C2 in statement 2, typically representative of gate delays. After statement 2 processes its delay of 20ns, the net delay of 10ns is added to it.

- Figure indicates the difference between C1 and C2 . C1 rejects all narrow pulses less than 30ns, whereas C2 rejects only pulses less than 20 units.



(a)

consider the following two statement pairs with the Y waveform as shown in Figure.

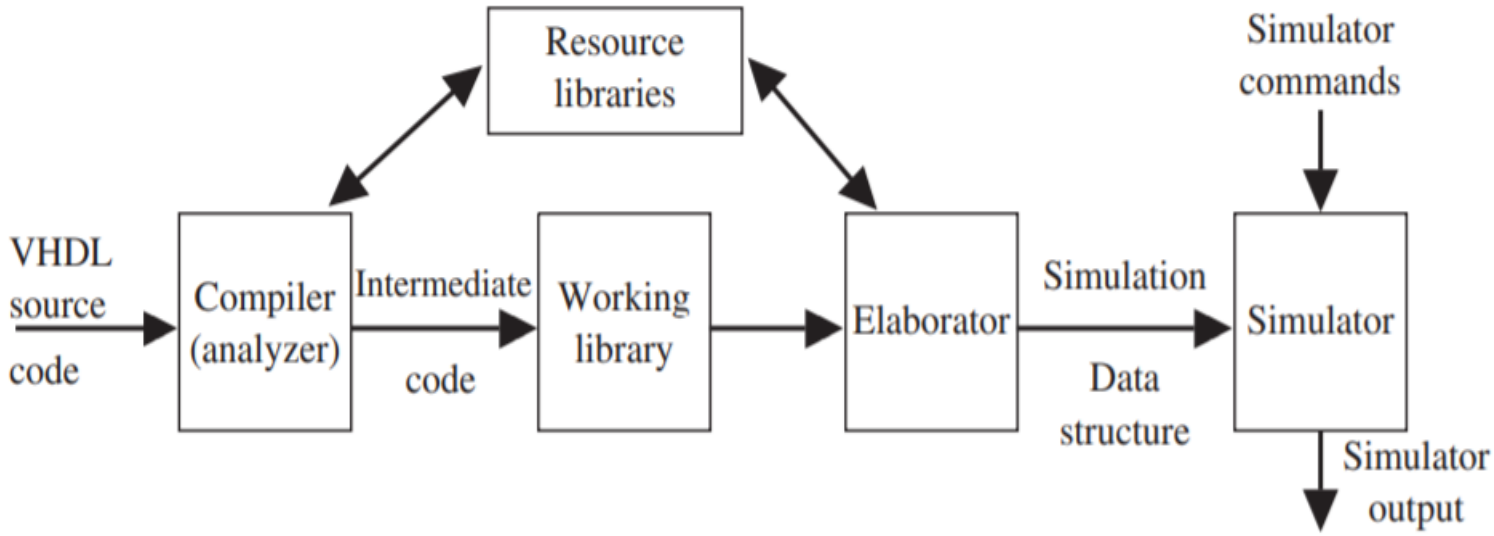
```
wire #3 D; // net delay on wire D
assign #7 D = Y; // statement 1 – inertial delay
wire #7 E; // net delay on wire E
assign #3 E = Y; // statement 1 – inertial delay
```

The assign statement for D works with a 7ns inertial delay and rejects any pulse below 7ns. Hence D rejects the 3ns, 2ns and 5ns pulses in Y. The 3ns net delay from the wire statement is added to the signal that comes out from the assign statement.

In the case of E, pulses below 3ns are rejected. Hence the 3ns pulse in Y passes through the assign statement for E, the 2ns pulse is rejected and the 5ns pulse is accepted.

Hence the 3ns and 5ns pulses get combined in the absence of the 2ns pulse to yield output on E appears as a big 10ns pulse. The 7ns net delay from the wire statement is added to the signal that comes out from the assign statement. If any pulses less than 7ns are encountered at the net delay phase, they will be rejected.

Compilation, Simulation, and Synthesis of Verilog Code

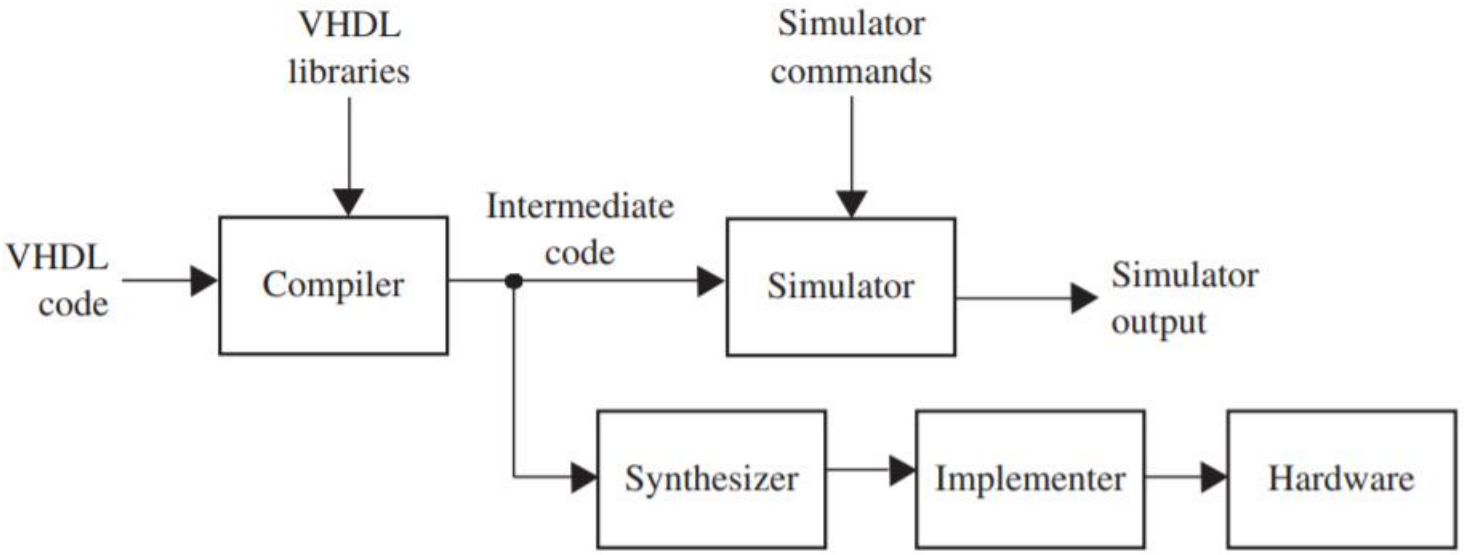


Simulation of the Verilog code is important for two reasons. First, we need to verify that the Verilog code correctly implements the intended design, and second, we need to verify that the design meets its specifications.
We first simulate the design and then synthesize it to the target technology (FPGA or custom ASIC).

There are three phases in the simulation of Verilog code: analysis (compilation), elaboration, and simulation.

- Before simulation, the Verilog code must first be compiled. The Verilog compiler, also called an analyzer, first checks the Verilog source code to see that it conforms to the syntax and semantic rules of Verilog. If there is a syntax error, such as a missing semicolon, or if there is a semantic error, such as trying to add two signals of incompatible types, the compiler will output an error message. The compiler also checks to see that references to libraries are correct. If the Verilog code conforms to all of the rules, the compiler generates intermediate code, which can be used by a simulator or by a synthesizer.
- Next, the design must have the modules being instantiated linked to the modules being defined, the parameters propagated among the various modules, and hierarchical references resolved. This phase in understanding a Verilog description is referred to as **elaboration**. During elaboration, a driver is created for each signal. Each driver holds the current value of a signal and a queue of future signal values.
- The simulation process consists of an initialization phase and actual simulation. The simulator accepts simulation commands, which control the simulation of the digital system and which specify the desired simulator output. Verilog simulation uses what is known as discrete event simulation.

One of the most important uses of Verilog is to synthesize or automatically create hardware from a Verilog description. The synthesis software for Verilog translates the Verilog code to a circuit description that specifies the needed components and the connections between the components. The initial steps (analysis and elaboration) in Figure are common whether Verilog is used for simulation or synthesis. The simulation and synthesis processes are shown in following Figurer.



- After the Verilog code for a digital system has been simulated to verify that it works correctly, the Verilog code can be synthesized to produce a list of required components and their interconnections, typically called the **netlist.**
- The synthesizer output can then be used to implement the digital system using specific hardware, such as a CPLD or an FPGA or as an ASIC. The CAD software used for implementation generates the necessary information to program the CPLD or FPGA hardware.

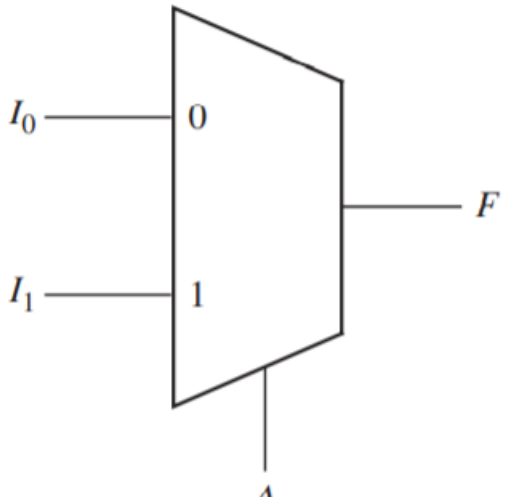
Verilog Models for Multiplexers:

A multiplexer is a combinational circuit and can be modeled using concurrent statements only or using always statements. A conditional operator with assign statement can be used to model a multiplexer without always statements. A case statement or if-else statement can also be used to make a model for a multiplexer within an always statement.

Using Conditional Operator:

Figure shows a 2-to-1 multiplexer (MUX) with 2 data inputs and one control input. The MUX output is F = A`.I_0 + A. I_1. The corresponding Verilog statement is

assign F = (~A && I_0) || (A && I_1);




```
// conditional signal assignment
statement

    assign F = (A) ? I1 : I0;
```

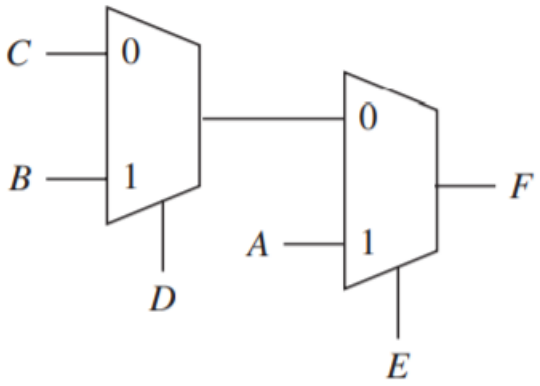
- Alternatively, we can represent the MUX by a conditional signal assignment statement as shown in Figure. This statement executes whenever A, I_0 , or I_1 changes. The MUX output is I_0 when A = 0; otherwise it is I_1 . In the conditional statement, I_0 , I_1 , and F can be one or more bits.

- The general form of a conditional signal assignment statement.

assign signal_name = condition ? expression_T : expression_F;

This concurrent statement is executed whenever a change occurs in a signal used in one of the expressions or conditions. If condition is true, signal_name is set equal to the value of expression_T. Otherwise if condition is false, signal_name is set equal to the value of expression_F.

Figure shows how two cascaded MUXes can be represented by a conditional signal assignment statement. The output MUX selects A when the condition E is true; otherwise, it selects the output of the first MUX, which is B when the condition D is true, or it is C.




```
assign F = E ? A : ( D ? B : C );
// nested conditional assignment
```

Figure 2-38 shows a 4-to-1 multiplexer (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

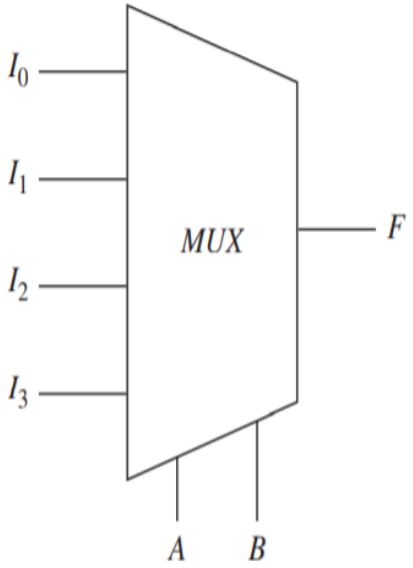
$$F = A'B'I_0 + A'B\,I_1 + A\,B'I_2 + A\,B\,I_3$$

One way to model the MUX is with the Verilog statement

```
assign F = (~A && ~B && I0) || (~A && B && I1) ||
       A && ~B && I2) || (A && B && I3);
```

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

```
assign F = (A) ? (B ? I3 : I2) : ( B ? I1 : I0 );
```



Using If-else or Case Statement in an Always Block:

If a MUX model is used inside an always statement, a concurrent statement cannot be used.

The MUX can be modeled using a case statement within an always block:

```
always @ (Sel or I0 or I1 or I2 or I3)
        case Sel
        2'b00 : = I0;
        2'b01 : F 5 I1;
         2'b10 : F 5 I2;
        2'b11 : F 5 I3;
        endcase
```

MUX has four input signals,

the selection signal, Sel should be a 2-bit signal.

The selection signals are represented as 2'b00, 2'b01, 2'b10, and 2'b11

The b represents that the base is binary here

The case statement has the general form:

```
Case
expression choice1 : sequential statements1
choice2 : sequential statements2
. . .
[default : sequential statements]
endcase;
```

The expression is evaluated first. If it is equal to choice1, then sequential statements1 are executed; if it is equal to choice2, then sequential statements2 are executed; and so forth. If all values are not explicitly given, a default clause is required in the case statement.

the MUX can also be modeled using an if-else statement within an always block:

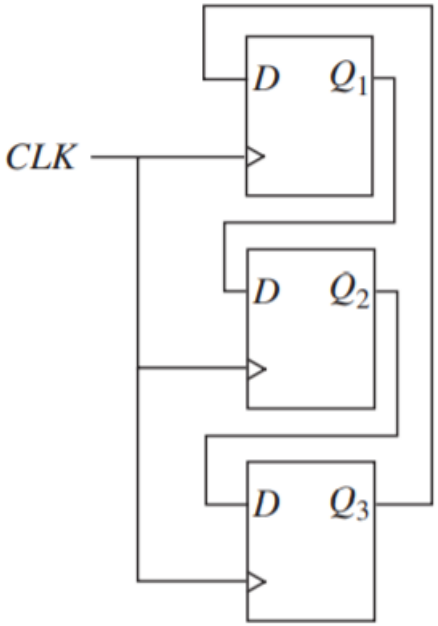
```
always @ (Sel or I0 or I1 or I2 or I3)
begin
 if (Sel == 2'b00)          F = I0;
else if (Sel == 2'b01)      F = I1;
else if (Sel == 2'b10)      F = I2;
else if (Sel == 2'b11)      F = I3;
end
```

Modeling Registers and Counters Using Verilog Always Statements:

Figure shows three flip-flops connected as a cyclic shift register (or a rotating shift register). These flip-flops all change state following the rising edge of the clock.

assumed a 5ns propagation delay between the clock edge and the output change

Immediately following the clock edge, the three statements in the always statement execute in sequence with no delay. Then the sequential statements are Q1 <= Q3; Q2 <= Q1; Q3 <= Q2;




```
always @ (posedge CLK)
begin
   Q1 <= #5 Q3;
   Q2 <= #5 Q1;
   Q3 <= #5 Q2;
end
```

- The order of the statements is not important when the non-blocking assignment operator "<= " is used. The same result is obtained even if the statements are in reverse order as shown here.

```
always @ (posedge CLK)
begin
Q3 <= #5 Q2;
Q2 <= #5 Q1;
Q1 <= #5 Q3;
end
```

Example 1: What is the hardware obtained if the following code is synthesized?

```
module reg3 (Q1,Q2,Q3,A,CLK);
input A;
input CLK;
output Q1,Q2,Q3;
reg Q1,Q2,Q3;
always @(posedge CLK)
begin
Q3 = Q2; // statement 1
 Q2 = Q1; // statement 2
Q1 = A; // statement 3
end
endmodule
```

Answer: A 3-bit shift register

In the above example,the list of statements executes from top to bottom in order. Note that the blocking operator is used. Therefore, the first statement finishes update before the second statement is executed. Synthesis results in a 3-bit shift register with serial input A, and outputs Q1 , Q2 , and Q3 .

Example2: What is the hardware obtained if the following code is synthesized?

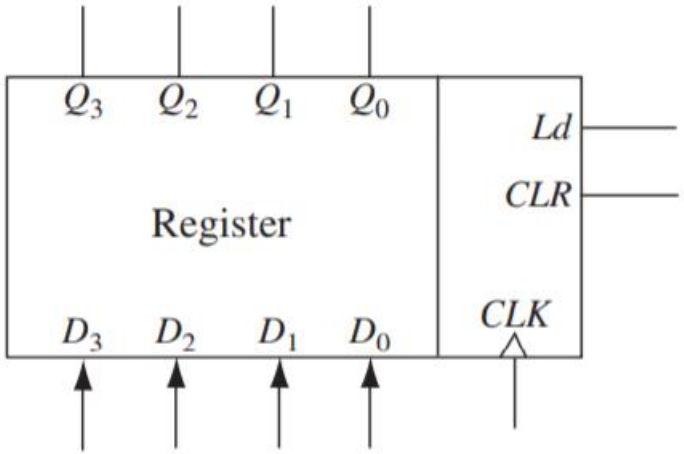
```
module reg31 (Q1,Q2,Q3,A,CLK);
input A;
 input CLK;
output Q1,Q2,Q3;
reg Q1,Q2,Q3;
always @(posedge CLK)
begin Q1 = A; // statement 1
Q2 = Q1; // statement 2
Q3 = Q2; // statement 3
end
endmodule
```

Ans: A single flip-flop

Explanation: In this example, blocking operator is used, so the list of statements executes from top to bottom in order. So the first statement finishes update before the second statement is executed. Q1 gets the value of the serial input A when statement 1 finishes. In statement 2, the same value propagates to Q2 . In statement 3, the same value propagates to Q3 . In effect, the input A has reached Q3 . Modern synthesis tools will generate a single flip-flop with input A when this code is synthesized. The outputs Q1 , Q2 , and Q3 can all be connected to the output of the same flip-flop. If the synthesizer does not have good optimization algorithms, it might generate three parallel flip-flops, each with the same input A but with outputs Q1 , Q2 , and Q3 , respectively.

Register with Synchronous Clear and Load:

Figure shows a simple register that can be loaded or cleared on the rising edge of the clock. If CLR is set to 1, the register is cleared, and if Ld = 1, the D inputs are loaded into the register. This register is fully synchronous so that the Q outputs change only in response to the clock edge and not in response to a change in Ld or CLR.




```
always @ (posedge CLK)
begin
 if (CLR)          Q <= 4'b0000;
 else if (Ld)      Q <= D;
end
```

- In the Verilog code for the register, Q and D are 4-bit vectors dimensioned [3:0]. Since the register outputs can only change on the rising edge of the clock, CLR and Ld are not on the sensitivity list. The CLR and Ld signals are tested after the rising edge of the clock.

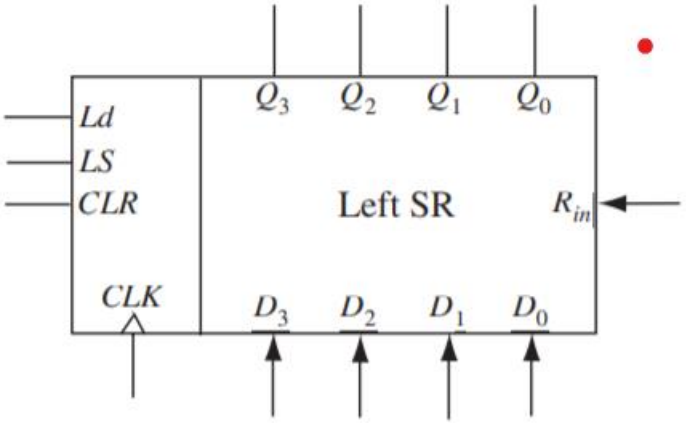
Left Shift Register with Synchronous Clear and load:

Model a left shift register using a Verilog always statement. A left shift control input (LS) is used in addition to Ld, CLR control input.

When LS = 1, the contents of the register are shifted left and the right-most bit is set equal to Rin. The shifting is accomplished by taking the rightmost 3 bits of Q, Q[2:0], and concatenating them with Rin.

For example, if Q = 1101 and Rin = 0, then {Q[2:0], Rin} = 1010, and this value is loaded back into the Q register on the rising edge of CLK.

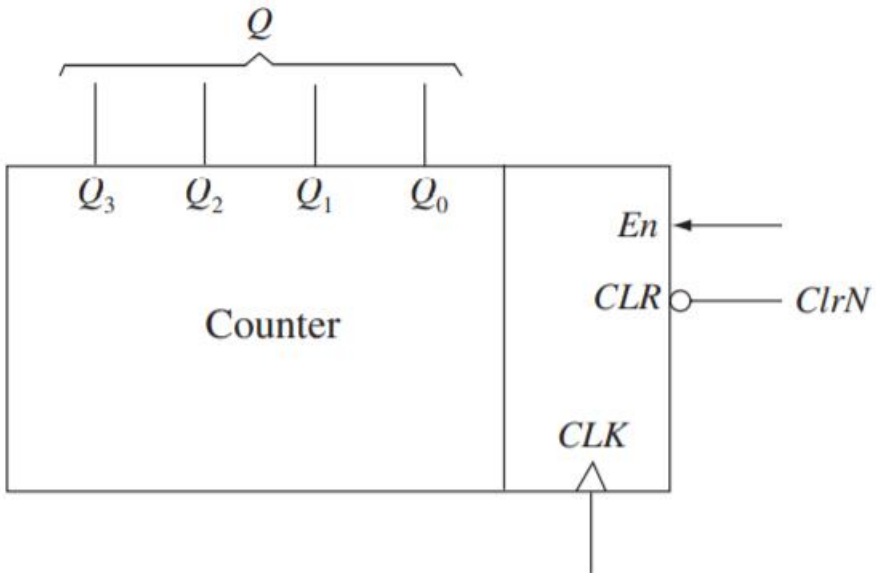
If CLR = Ld =LS = 0, then Q remains unchanged.




```
always @ (posedge CLK)
begin
 if (CLR)        Q <= 4'b0000;
 else if (Ld)    Q <= D;
 else if (LS)    Q <= {Q[2:0], Rin};
end
```

Synchronous counter: Figure shows a simple synchronous counter. On the rising edge of the clock, the counter is cleared when ClrN =0, and it is incremented when ClrN = En = 1. In this example, the signal Q represents the 4-bit value stored in the counter.

The signal Q is declared to be of type reg with length of 4 bits. Then Q <= Q+1; increments the counter. When the counter is in state 1111, the next increment takes it back to state 0000.



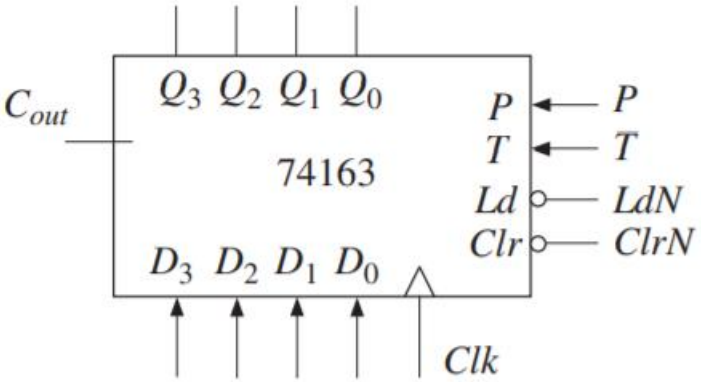

```
reg Q[3:0];

always @ (posedge CLK)
begin
 if (~ClrN)     Q <= 4'b0000;
 else if (En)   Q <= Q + 1;
end
```

Verilog model for a standard MSI counter (74163):

It is a 4-bit fully synchronous binary counter, which is available in both TTL and CMOS logic families.

It can be cleared or loaded in parallel.

All operations are synchronized by the clock, and all state changes take place following the rising edge of the clock input.



Control Signals			Next State				
ClrN	*LdN*	*PT*	Q_3^+	Q_2^+	Q_1^+	Q_0^+	
0	X	X	0	0	0	0	(clear)
1	0	X	D_3	D_2	D_1	D_0	(parallel load)
1	1	0	Q_3	Q_2	Q_1	Q_0	(no change)
1	1	1	present state + 1				(increment count)

This counter has four control inputs—ClrN, LdN, P, and T. Both P and T are used to enable the counting function.

While P is an actual enable signal to the 4-bit generic counter, T is used for a carry connection signal when cascading multiple counters.

Operation of the counter is as follows:

1. If ClrN = 0, all flip-flops are set to 0 following the rising clock edge.
2. 2. If ClrN = 1 and LdN = 0, the D inputs are transferred (loaded) in parallel to the flip-flops following the rising clock edge.
3. 3. If ClrN = LdN = 1 and P = T = 1, the count is enabled and the counter state will be incremented by 1 following the rising clock edge.

If T = 1, the counter generates a carry (Cout) in state 15; consequently

$$Cout = Q3\ Q2\ Q1\ Q0\ T$$

```
// 74163 FULLY SYNCHRONOUS COUNTER
        module c74163 (LdN, ClrN, P, T, Clk, D, Cout, Qout);
        input       LdN;
        input       ClrN;
         input       P;
        input       T;
        input       Clk;
        input       [3:0] D;
        output      Cout;
        output [3:0]  Qout;
        reg [3:0]     Q;
        assign Qout = Q;
        assign Cout = Q[3] & Q[2] & Q[1] & Q[0] & T;
        always @ (posedge Clk)
        begin
        if (~ClrN)              Q <= 4'b0000;
        else if (~LdN)          Q <= D;
        else if (P & T)         Q <= Q 1 1;
        end
        endmodule
```

Two 74163 Counters Cascaded to Form an 8-Bit Counter: