# Module 2

Click to add subtitle

You

- Verilog Models for Multiplexers, Modeling Registers and Counters Using Verilog Always Statements, Constants, Arrays, Loops in Verilog, Hazards(ming-bo-lin) Testing a Verilog Model
- **Additional topics in Verilog**: Verilog Functions, Verilog Tasks, Multivalued Logic and Signal Resolution, Built-in Primitives User-Defined Primitives, Named Association, Generate Statements , System Functions, File I/O Functions.[ Text Book 1]

- 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

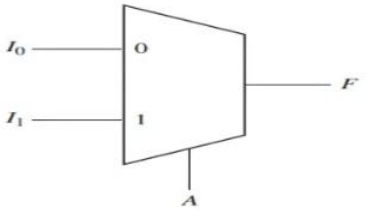
# Verilog Models for Multiplexers

## 2.13.1 Using Conditional Operator

Figure 2-36 shows a 2-to-1 multiplexer (MUX) with 2 data inputs and one control input. The MUX output is $F = A' \cdot I_0 + A \cdot I_1$. The corresponding Verilog statement is

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

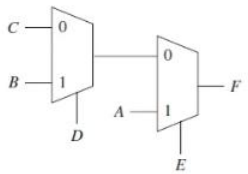
Here, the MUX can be modeled as a single concurrent signal assignment statement. Alternatively, we can represent the MUX by a conditional signal assignment statement as shown in Figure 2-36. This statement executes whenever $A$, $I_0$, or $I_1$ changes. The MUX output is $I_0$ when $A = 0$; otherwise it is $I_1$. In the conditional statement, $I_0$, $I_1$, and $F$ can be one or more bits.




```
// conditional signal assignment
statement
    assign F = (A) ? I1 : I0;
```

The general form of a conditional signal assignment statement is

```
assign signal_name = condition ? expression_T : expression_F;
```




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

You

# 4:1 Multiplexer

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

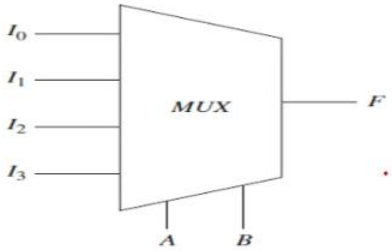
$$F = A'B'I_0 + A'B\,I_1 + A\,B'I_2 + A\,B\,I_3$$

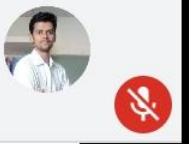
One way to model the MUX is with the Verilog statement

```
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 );
```



## 2.13.2 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 : F = I0;
     2'b01 : F = I1;
     2'b10 : F = I2;
     2'b11 : F = I3;
  endcase
```

Since this 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 in the form of <number of bits>'<base><value>. 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. All possible values of the expression must be included in the choices. If all values are not explicitly given, a default clause is required in the **case** statement. As an alternative, 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
```

One might notice that combinational circuits can be described using concurrent or sequential statements. Sequential circuits generally require an **always** statement. **Always** statements can be used to make sequential or combinational circuits.

The following are important coding practices while writing synthesizeable Verilog for combinational hardware:

(a) If possible use concurrent assignments (e.g., assign) to design combinational logic.

(b) When procedural assignments (always blocks) are used for combinational logic, use blocking assignments (e.g., "=").

(c) If Verilog 2001 or later is used, instead of specifying contents of sensitivity lists, use always@* to avoid accidental omission of inputs from sensitivity lists. The accidental omission results in incorrect hardware or deviation between simulation and synthesis.

The following are important coding practices while writing synthesizeable Verilog for combinational hardware:

(a) If possible use concurrent assignments (e.g., assign) to design combinational logic.

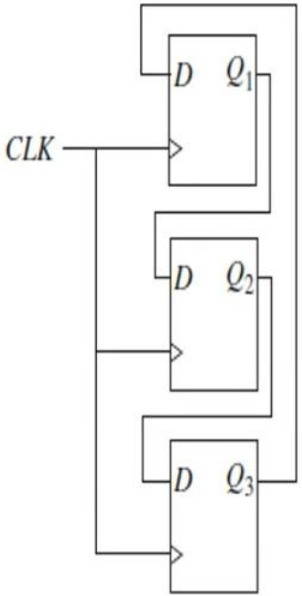
(b) When procedural assignments (always blocks) are used for combinational logic, use blocking assignments (e.g., "=").

(c) If Verilog 2001 or later is used, instead of specifying contents of sensitivity lists, use always@* to avoid accidental omission of inputs from sensitivity lists. The accidental omission results in incorrect hardware or deviation between simulation and synthesis.

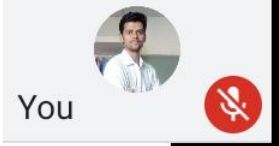
You

1.Modeling Registers and Counters Using Verilog Always Statements

# Cyclic Shift Register




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



at first, but that is the way the hardware works. At the rising edge of the clock, all of the D inputs are loaded into the flip-flops, but the state change does not occur until after a propagation delay.

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
```

You

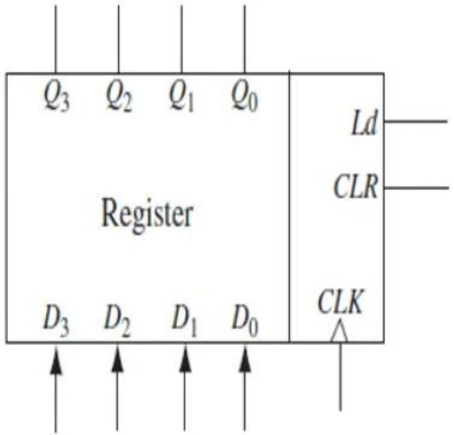
### *Example 2*

What is the hardware obtained if the following code is synthesized? Note that this is the same code as in the previous example, but with the statement order inside the always block reversed.

```
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
```

You

# Register with Synchronous Clear and Load




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

You