DIGITAL SYSTEM DESIGN USING VERILOG

- **Course code:** 18EC5DCDSV
- **Credits:** 03

Course Objectives:

- To study Verilog HDL in high-level synthesis of digital system designs.
- To learn Verilog HDL for modelling, simulating and synthesizing various digital modules.
- To study the additional features of Verilog HDL
- To be able to design and develop Test bench for Combinational and sequential modules
- To be able to design fixed and floating point arithmetic
- To be able to design and implement synchronous sequential circuits using FSM

Course Outcomes

CO1	**Describe & model digital blocks of computing systems using Verilog hardware description language..**
CO2	Apply hardware description language for writing test benches to check combinational & sequential modules.
CO3	Design clocked synchronous circuits and perform timing analysis
CO4	Analyze digital System and model using Verilog HDL.
CO5	Apply design Knowledge to FSM based digital Modules.
CO6	Synthesize and implement a given digital system.

Module 1

- **Computer-Aided Design , Hardware Description Languages, Module modelling styles(ming bo lin), Verilog Description of Combinational Circuits,**
- **Data flow modelling-dataflow modelling, operands, operators, (ming bo lin), Verilog Modules, Verilog Assignments, Procedural Assignments), Modeling Flip-Flops Using Always Block, Always Blocks Using Event Control Statements.[Text book 1 and 3] Delays in Verilog, Compilation, Simulation, and Synthesis of Verilog Code , Simple Synthesis Examples**
- TEXT BOOKS:Charles H Roth Jr., Lizy Kurian John, Byeong –kill-lee “Digital System Design using Verilog”, *publisher Cengage learning*
- Ming-Bo Lin, “Digital System designs and Practices using Verilog HDL and FPGAs”, *John Wiley & Sons*, 2008

Introduction to verilog

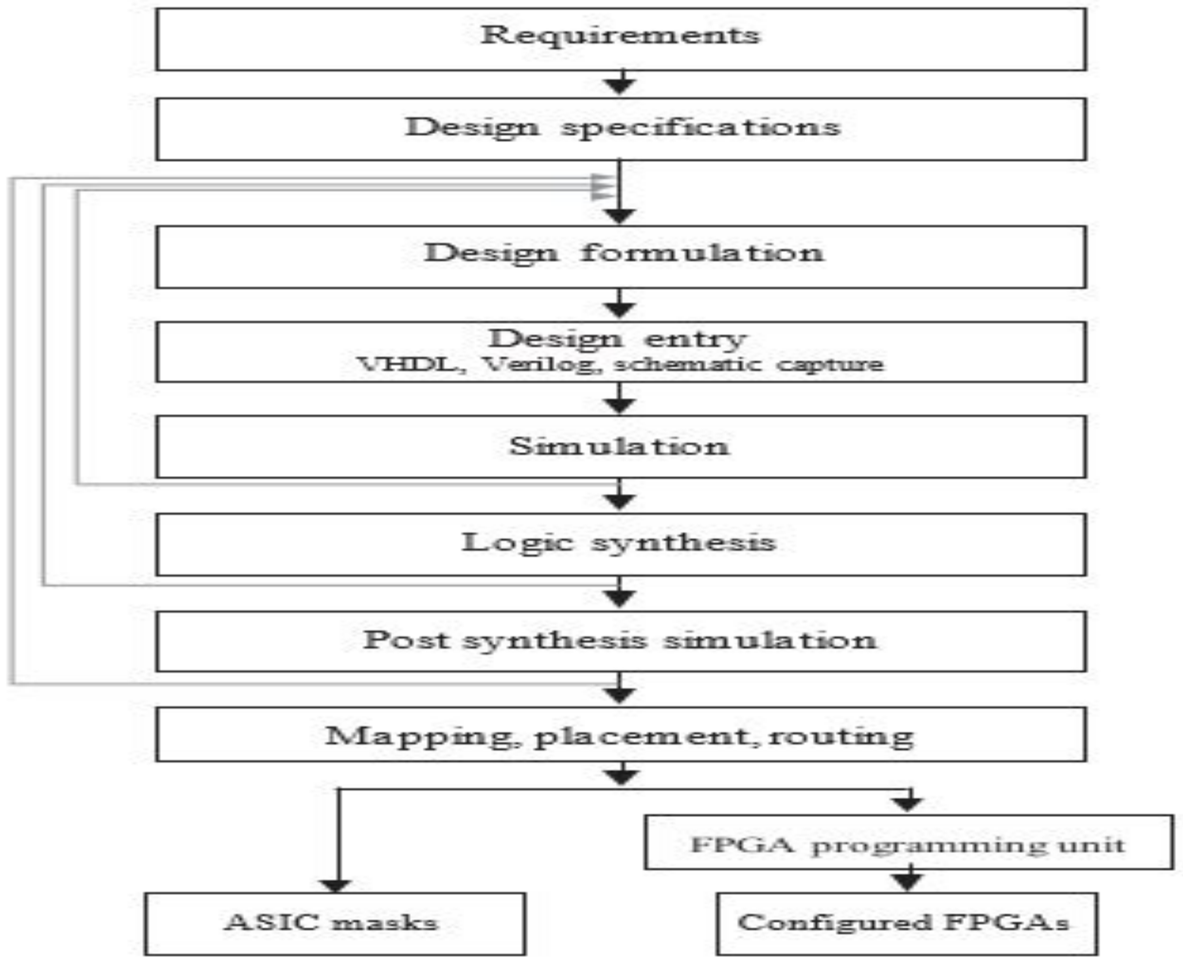
- Complexity
- Tedious and Time consuming
- Hand drawn Schematics,breadbords and wires
- More and more components on a chip,
- Improvements have advanced the **VLSI** (very large scale integration) field

- SSI (small-scale integration),
- MSI (medium-scale integration),
- LSI (large-scale integration)
- All depends on the density of integration.
- SSI referred to ICs with 1 to 20 gates
- MSI referred to ICs with 20 to 200 gates
- LSI referred to devices with 200 to a few thousand gates
- Building blocks such as adders, multiplexers, decoders, regis ters, and counters are available as MSI standard parts.
- Term VLSI was coined, devices with 10,000 gates were called VLSI chips. The boundaries between the different categories are fuzzy today. Many modern microprocessors contain more than 100 million transistors. Compared to what was referred to as VLSI in its initial days
- Modern integration capability could be described as ULSI (ultra large scale integration). Despite the changes in integration ability and the fuzzy defini- tion, the term VLSI remains popular.

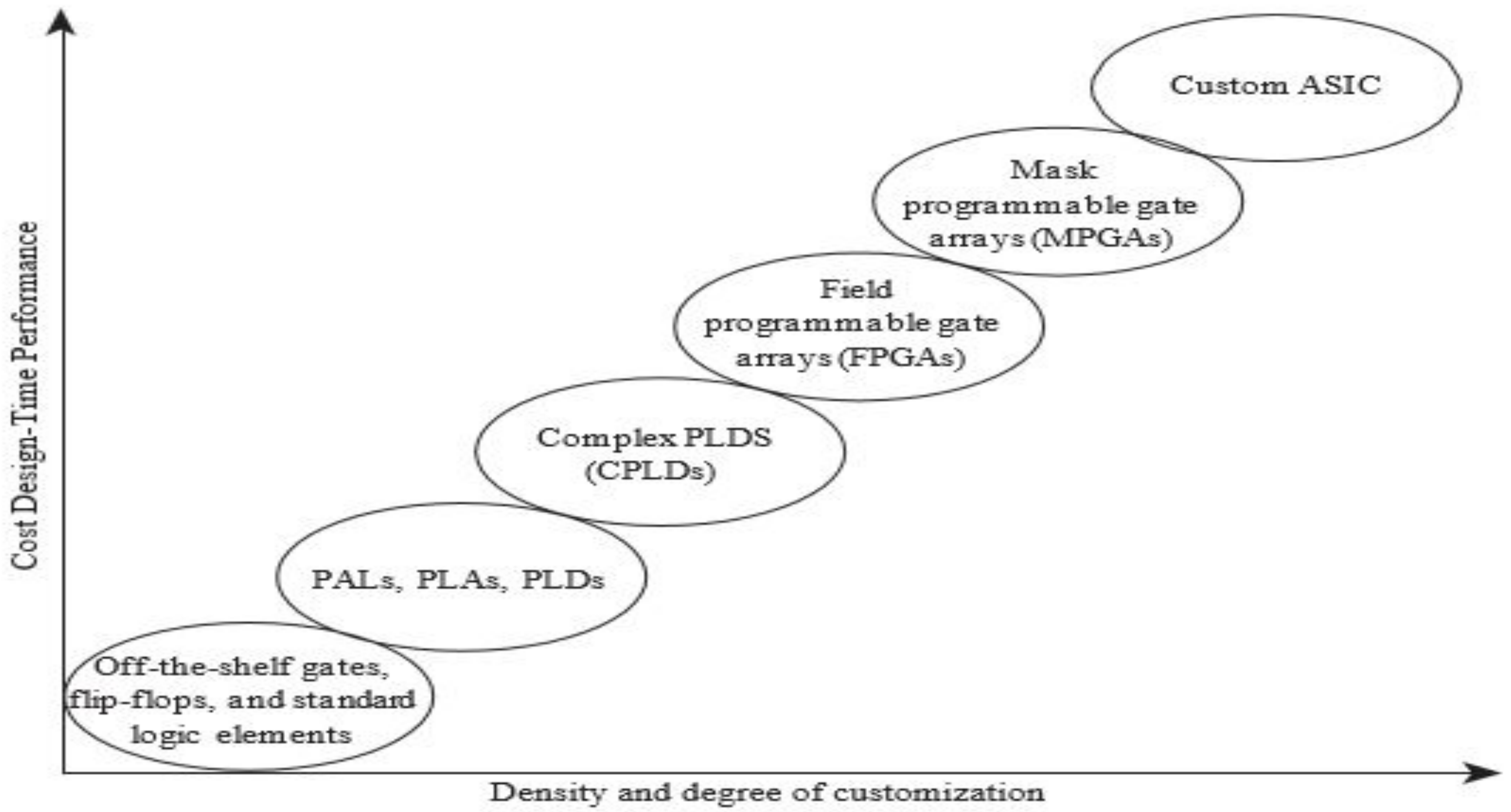
Computer-aided design

- **(CAD)** tools have advanced significantly during the past decade, and nowadays digital design is performed using a variety of software tools.
- Prototypes or even final designs can be created without discrete components and interconnection wires.

Design Flow in Modern Digital System Design



Spectrum of Design Technologies


Custom ASIC
Mask programmable gate arrays (MPGAs)
Field programmable gate arrays (FPGAs)
Complex PLDS (CPLDs)
PALs, PLAs, PLDs
Off-the-shelf gates, flip-flops, and standard logic elements
Cost Design-Time Performance
Density and degree of customization

Hardware Description Languages

- Hardware description languages (HDLs) are a popular mode of design entry for digital circuits and systems.
- There are two popular HDLs—VHDL and Verilog.
- Before the advent of HDLs, designers used graphical schematics and schematic capture tools to document and simulate digital circuits.
- A need was felt to create a textual method of documenting circuits and feeding them into simulators in the textual form as opposed to a graphic form.

Verilog

- Verilog is a hardware description language used to describe the behavior and/ or structure of digital systems
- Verilog is a general-purpose hardware description language that 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 was orig- inally developed under funding from the Department of Defense (DoD)
- Verilog was developed by the industry. It was initially developed as a proprietary language by a company called Gateway Design Automation around 1984.
- In 1990, Cadence acquired Gateway Design Automation and became the owner of Verilog. Cadence marketed it as a language and as a simulator, but it remained pro- prietary.
- At this time, Synopsis was promoting the concept of top-down design using Verilog. Cadence realized that it needed to make Verilog open in order to prevent the industry from shifting to VHDL and hence opened up the language.
- An organization called Open Verilog International (OVI) was formed, which helped to create a ven- dor-independent language specification for Verilog, clarifying many of the confusions around the proprietary specification. This was followed by an effort to create an IEEE standard for Verilog. The first Verilog IEEE Standard was created in 1995, which was revised in 2001 and 2005. Synopsis created synthesis tools for Verilog around 1988

Module Modelling Styles

- Synthesis is process that converts HDL mdules into a structural description
- It s divided into logic synthesis and high level synthesis
- RTL description into gate level netlist
- High level description into RTL results

- MODULES

 Two parts

- Interface
- Internal

Modeling the Internal of a Module

- 1.Structural Style

    -Gate Level

    -Switch Level

2.Dataflow Style

  -Module is specified as a set of continuous assignments statements

3.Behavioral or algorithmic style

4.Mixed style

  -modelling Large Designs

Port Declaration

- Interface signals of any Verilog HDL module can be of three types
  - Input
  - output
  - Inout
- The complete interface of a module is to divide it into three parts
  - Port list
  - Port Declaration
  - Data type declaration of each type

Port List style

```
module adder(x,y,c_in,sum,c_out)
  Input [3:0] x,y;
  Input c_in;
  output [3:0] sum;
  output c_out;
  reg [3:0] sum;
  reg c_out;
```

This style of port declaration is known as port list style

```
module adder(x,y,c_in,sum,c_out)
  Input [3:0] x,y;
  Input c_in;
  output  reg [3:0] sum ;
  output  reg c_out;

module adder(input [3:0]x,y, input c_in, output  reg [3:0] sum
  output  reg c_out );
```

Port Connection Rules

- Named association
- Positional association

Coding Styles

- A module cannot be declared within another module
- A module can instantiate other modules
- A module instantiation must have a module identifier (instance name) except built in primitives,gate and switch primitives and user defined primitives

Named association at the top level modules to avoid confusion

Port connection Rules

module half adder(x,y,s,c);

Input x,y;

Output ,c;

//half adder body//

//instantiate primitive gates

xor xor1(s,x,y);   (s,x,y)-Positional association

and and1 (c,x,y);   and1 (instance name is optional)

endmodule

```
module full _adder(x,y,cin,s,cout);
Input x,y,cin;
output s,cout;
wire s1,c1,c2;
// full adder body//
// instantiate the half adder
half adder ha_1(x,y,s1,c1);         // (x,y,s1,c1)-Positional  association
half adder ha_2(.x(cin), .y(s1), .s(s), .c(c2)); //ha_2(instance name is necessary)
Or (cout,c1,c2);                       //(.x(cin), .y(s1), .s(s), .c(c2))-Named association
endmodule
```

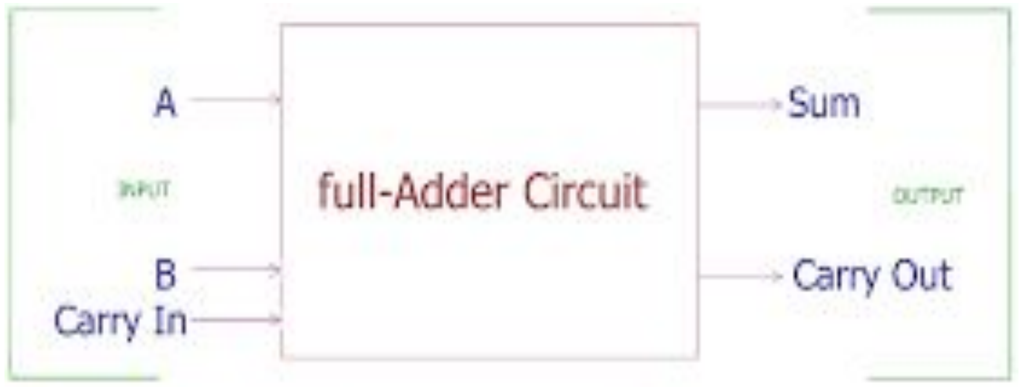
Structural Modelling

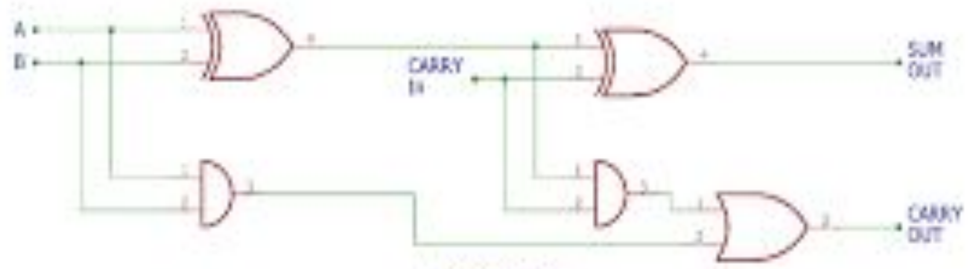
- Structural Modelling at gate level

The Half adder instantiates two gate primitives

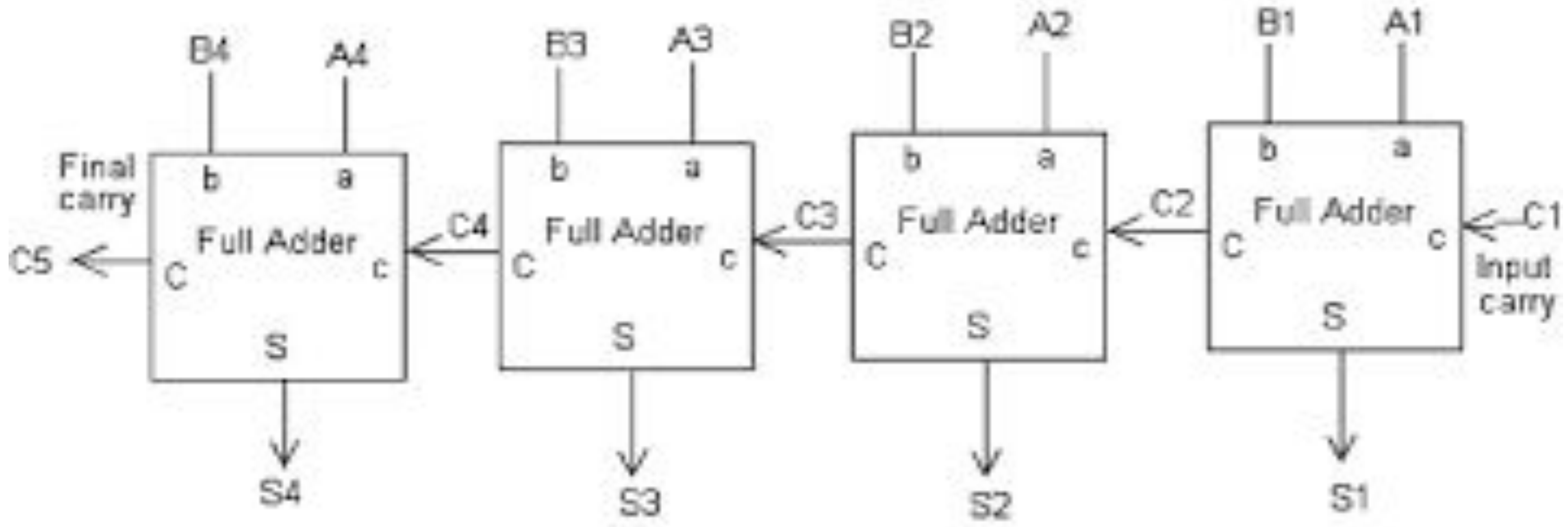
The Full adder instantiates two half modules and one gate primitive

The Four bit adder is constructed by four full adders instances





Full-Adder Circuit


B4
A4
B3
A3
B2
A2
B1
A1
Final carry
C5
C4
C3
C2
C1
Input carry
Full Adder
b
a
C
c
S
S4
S3
S2
S1

Gate level description of half adder

```
module half adder(x,y,s,c);
Input x,y;
Output s,c;
//half adder body
// instantiate primitive gates
 xor(s,x,y);
and(c,x,y);
endmodule
```

Gate level description of Full adder

```
module full adder(x,y,cin,s,cout);
input x,y ,cin;
output s,cout;
Wire s1,c1,c2;
//full adder body
//instantiate the half adder
half _adder ha_1(x,y,s1,c1);
half_adder ha_2(cin,s1,s,c2);
Or(cout,c1,c2);
endmodule
```

Gate level description of 4 bit adder

```
module 4_bit_adder(x,y,c_in,sum,c_out);
Input [3:0] x,y;
Input c_in;
output [3:0] sum;
output c_out;
Wire c1,c2,c3;
//4_bit adder body
// instantiate the full adder
```

```
full_adder fa_1(x[0],y[0],c_in,sum[0],c1);
full_adder fa_2 (x[1],y[1],c1,sum[1],c2);
full _adder fa_3(x[2],y[2],c2,sum[2],c3);
full _adder fa_4(x[3],y[3],c3,sum[3],c_out);
endmodule
```

Dataflow Modelling

- Continuous assignment
- Keyword used"assign"

syntax

assign[delay] 1_value=expression;

✓ Expression is evaluated and result is assigned to 1_value after the specified delay

✓ Default –zero delay

```
module full_adder_dataflow(x,y,c_in,sum,c_out);
//  I/O port declarations
Input x,y,c_in;
output sum,c_out;
//specify the function of full adder

assign #5 {c_out,sum} =x+y+c_in;
endmodule
```

Behavioral Modelling

- Two procedural constructs

□ Initial

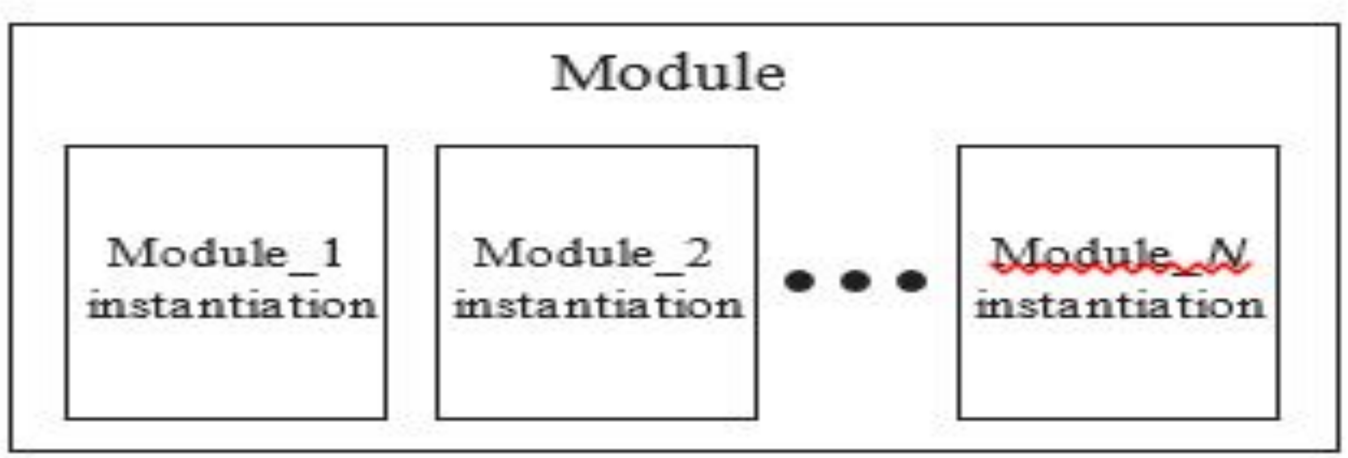
□ Always

Initial statement used to set up initial values of variable data types

Always are used to model combinational and sequential logic

```
module full_adder_behavioral(x,y,c_in,sum,c_out);
Input x,y,c_in;
output sum,c_out;
reg sum,c_out;
always @(x,y,c_in)
#5 {c-out,sum}=x+y+c_in;
endmodule
```

Verilog program structure



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

Module Declaration

module module-name (module interface list);

[list-of-interface-ports]

...

[port-declarations]

...

[functional-specification-of-module]

...

endmodule

The items enclosed in square brackets are optional. The list-of-interface- ports normally has the following form:

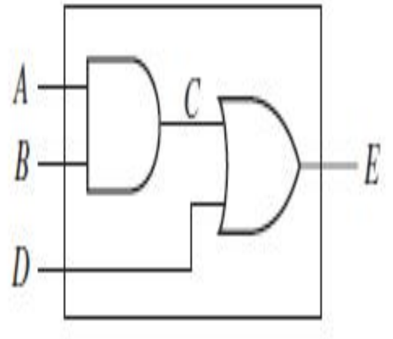
type-of-port list-of-interface-signals

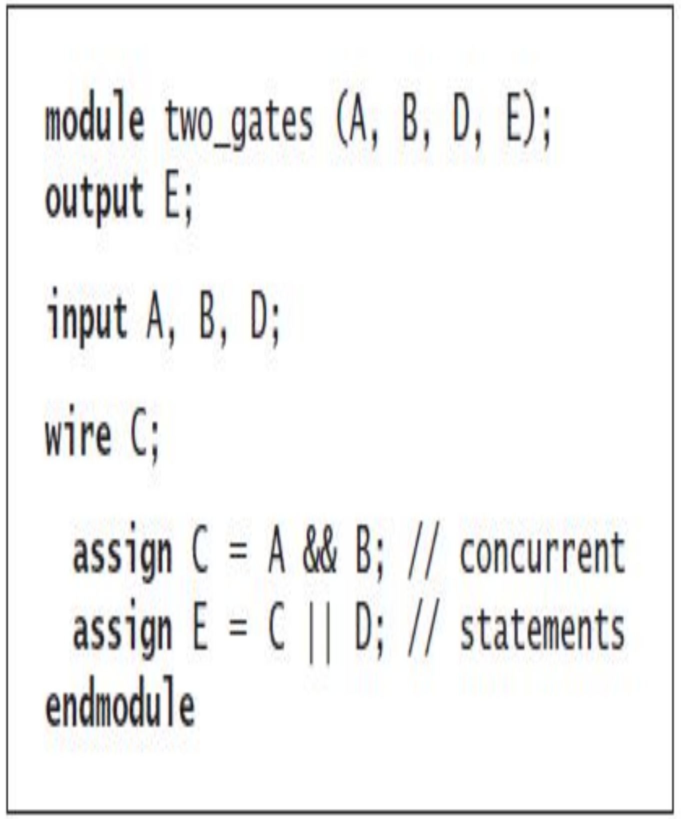
{; type-of-port list-of-interface-signals};

- The curly brackets indicate zero or more repetitions of the enclosed clause. Type-of-port indicates the direction of information; whether information is flowing into the port or out of it.
- Input port signals are of keyword **input**, output port signals are of keyword **output**, and bidirectional signals are of keyword **inout**. Also, list-of- ports can be combined with the module interface list.

Verilog Modules

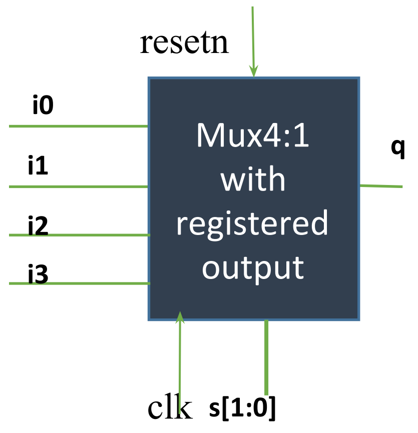
- The general structure of a Verilog code is a module description.
- A module is a basic building block that declares the input and output signals and specifies the internal operation of the module
- The **module** declara- tion has the name two_gates and specifies the inputs and outputs. *A*, *B*, and *D* are input signals, and *E* is an output signal.
- The signal *C* is declared within the module as a **wire** since it is an internal signal.
- The two concurrent statements that describe the gates are placed and the module ends with **endmodule**. All the input and output signals are listed in the module statement without specifying whether they are input or output.

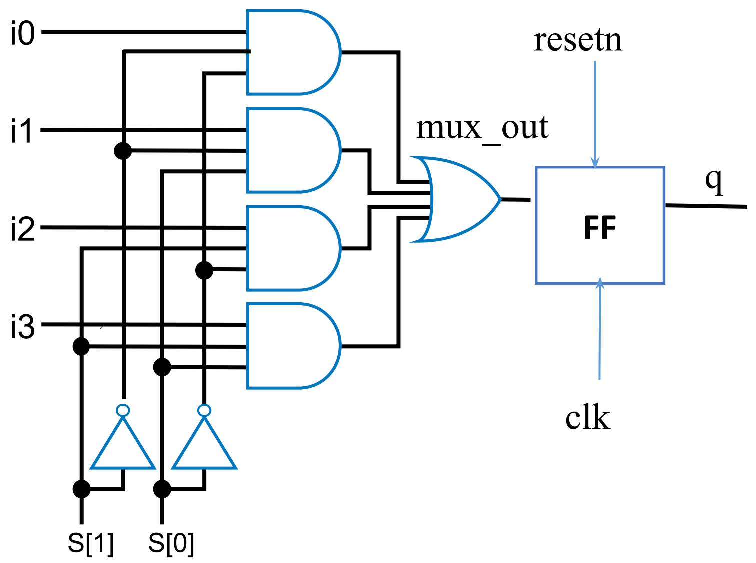




```
module two_gates (A, B, D, E);
output E;

input A, B, D;

wire C;

  assign C = A && B; // concurrent
  assign E = C || D; // statements
endmodule
```

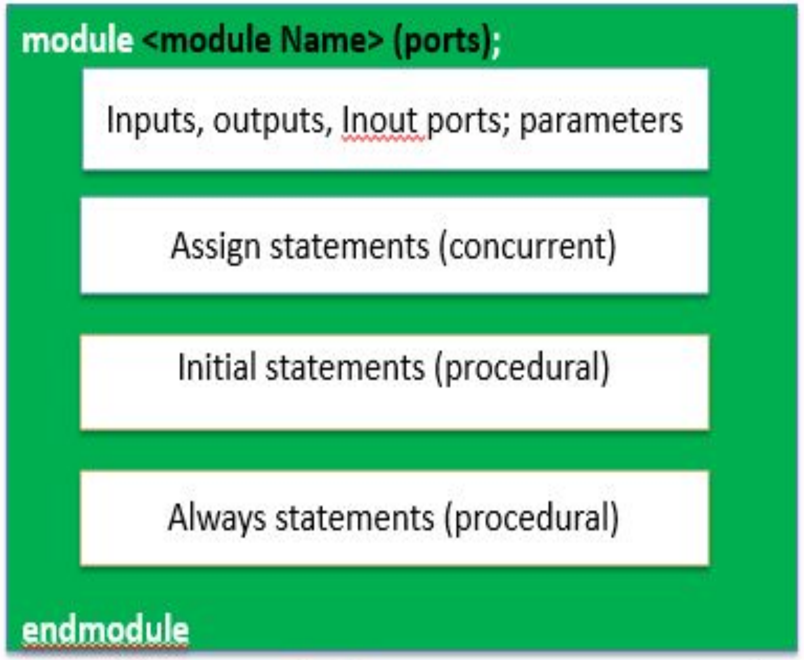
Multiplexer Design



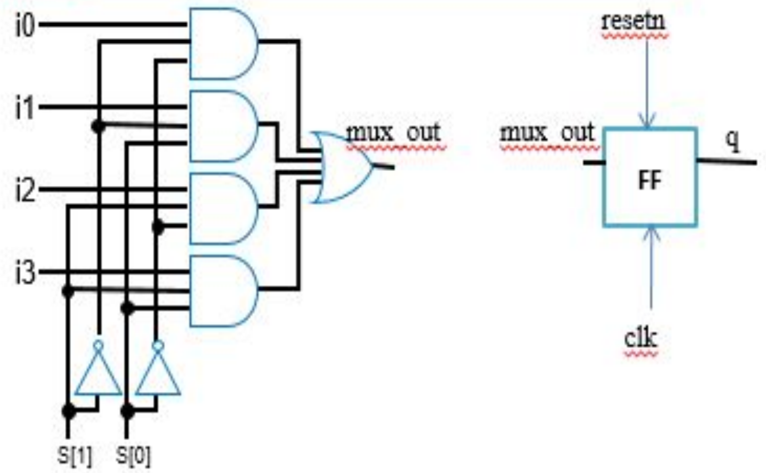


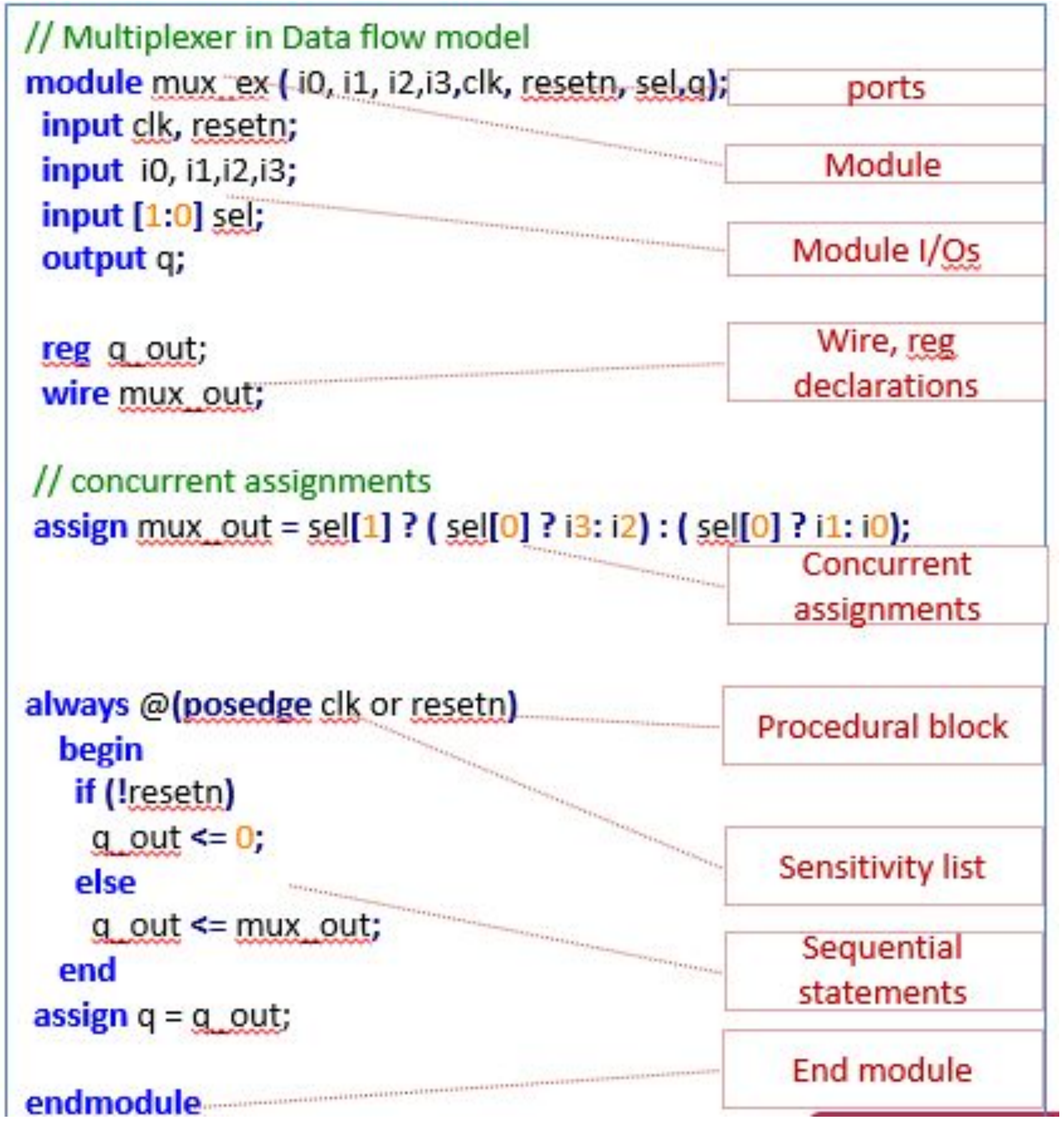
Sel[1:0]	mux_out
00	i0
01	i1
10	i2
11	i3

- A multiplexer of 2^n inputs has n select lines, are used to select which input line to send to the output.
- There is only one output in the multiplexer
- Simple 4: 1 Multiplexer with a registered output
- Based on the select input Sel[1:0], mux_out shall be any one of i0, i1, i2, i3 as shown in truth table above
- Multiplexer output is registered in the last stage



```
module <module Name> (ports);
    Inputs, outputs, Inout ports; parameters
    Assign statements (concurrent)
    Initial statements (procedural)
    Always statements (procedural)
endmodule
```





```
// Multiplexer in Data flow model
module mux_ex ( i0, i1, i2,i3,clk, resetn, sel,q);
 input clk, resetn;
 input  i0, i1,i2,i3;
 input [1:0] sel;
 output q;

 reg  q_out;
 wire mux_out;

 // concurrent assignments
 assign mux_out = sel[1] ? ( sel[0] ? i3: i2) : ( sel[0] ? i1: i0);


always @(posedge clk or resetn)
  begin
   if (!resetn)
    q_out <= 0;
   else
    q_out <= mux_out;
  end
 assign q = q_out;

endmodule
```

```
module <module Name> (ports);
    Inputs, outputs, Inout ports; parameters
    Assign statements (concurrent)
    Initial statements (procedural)
    Always statements (procedural)
endmodule
```

```
// Multiplexer in Data flow model
module mux_ex ( i0, i1, i2,i3 ,clk, resetn, sel,q);
 input clk, resetn;
 input  i0, i1,i2,i3;
 input [1:0] sel;
 output q;

 reg  q_out;
 wire mux_out;

 // concurrent assignments
 assign mux_out = sel[1] ? ( sel[0] ? i3: i2) : ( sel[0] ? i1: i0);



always @(posedge clk or resetn)
   begin
     if (!resetn)
       q_out <= 0;
     else
       q_out <= mux_out;
   end
 assign q = q_out;

endmodule
```

Verilog Assignments

- There are two types of assignment in the Verilog:

 continuous assignments

  proce- dural assignments.

Continuous assignments are used to assign values for combinational logic circuits. The **assign** keyword can be used after the net is separately declared

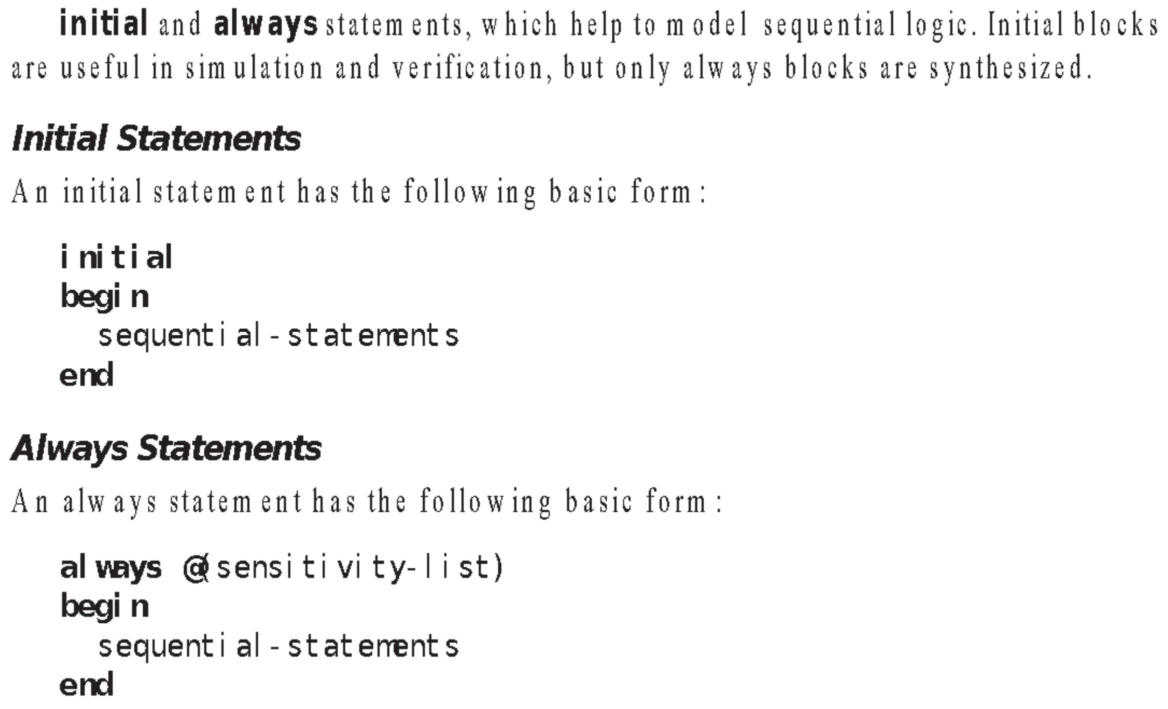
wire C;

assign C = A || B;   // explicit continuous assignment

wire D = E && F;   // implicit continuous assignment

Procedural Assignments

- Procedural assignments are used to model registers and finite state machines using the **always** keyword



initial and **always** statements, which help to model sequential logic. Initial blocks are useful in simulation and verification, but only always blocks are synthesized.

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
```

Blocking and Non-Blocking Assignments

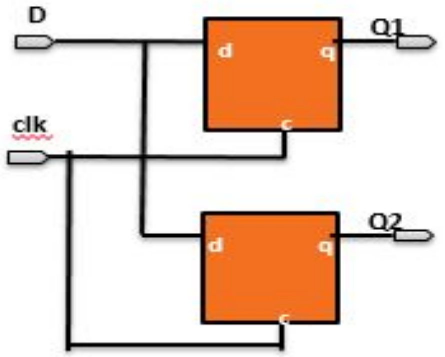
Blocking and Non-Blocking Assignments

- Sequential statements can be evaluated in two different ways 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
- A **non-blocking statement** allows assignment evaluation without blocking the sequential flow. In other words, several assignments can be evaluated at the same time.

Double synchronizer example – 2 Flip flops are connected back to back

Use of blocking statements

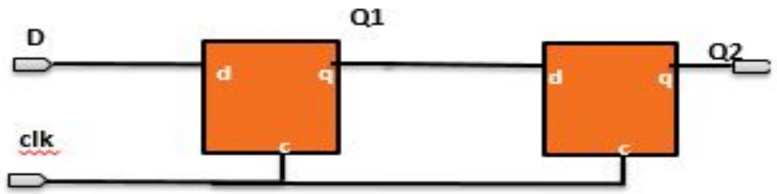
```
// D FFs with blocking assignment
module example1(D, clk,  Q2);
 input D, clk;
 output  Q2;
 reg Q1, Q2;

 always @(posedge clk)
  begin
   Q1 = D;
   Q2 = Q1;
  end
endmodule
```



Combinatorial elements must always use blocking assignments =

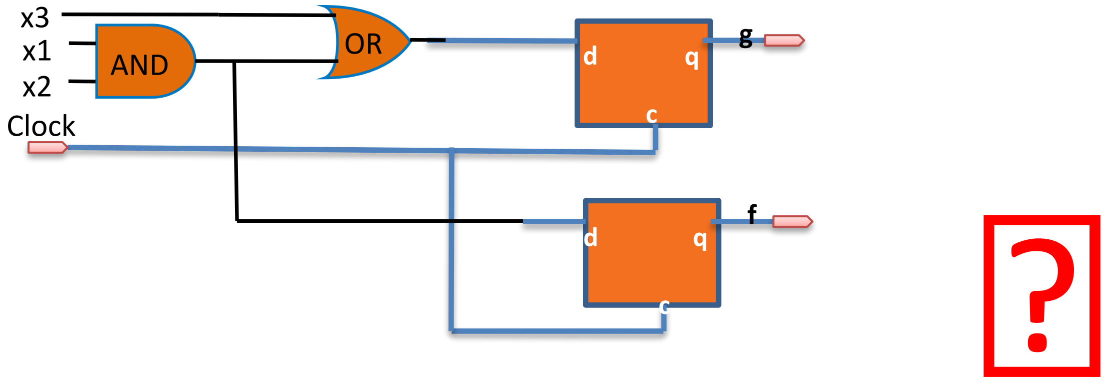
Use of Non blocking statements

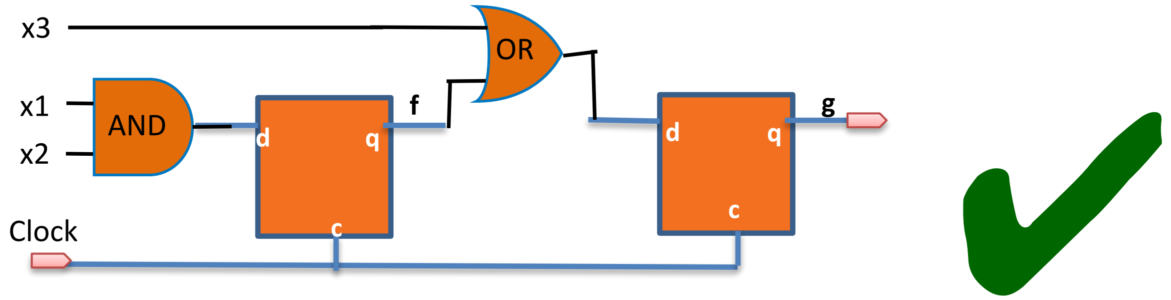
```
// D FFs with Non blocking assignment
module example1(D, clk,  Q2);
 input D, clk;
 output  Q2;
 reg Q1, Q2;
 always @(posedge clk)
  begin
   Q1 <= D; // substitute
   Q2 <= Q1; // non-blocking assignments
  end
endmodule
```



Sequential elements must always use non blocking assignments <=

```
// Example  for blocking
module example3(x1, x2, x3, Clock,  g);
 input x1, x2, x3, Clock;
 output  g;
 reg f, g;

 always @(posedge Clock)
 begin
  f = x1 & x2;
  g = f | x3;
 end
endmodule

//Example  for non  blocking

 module example3(x1, x2, x3, Clock,  g);
 input x1, x2, x3, Clock;
 output  g;
 reg f, g;

 always @(posedge Clock)
   begin
   f <= x1 & x2;
   g <= f | x3;
 end
endmodule
```





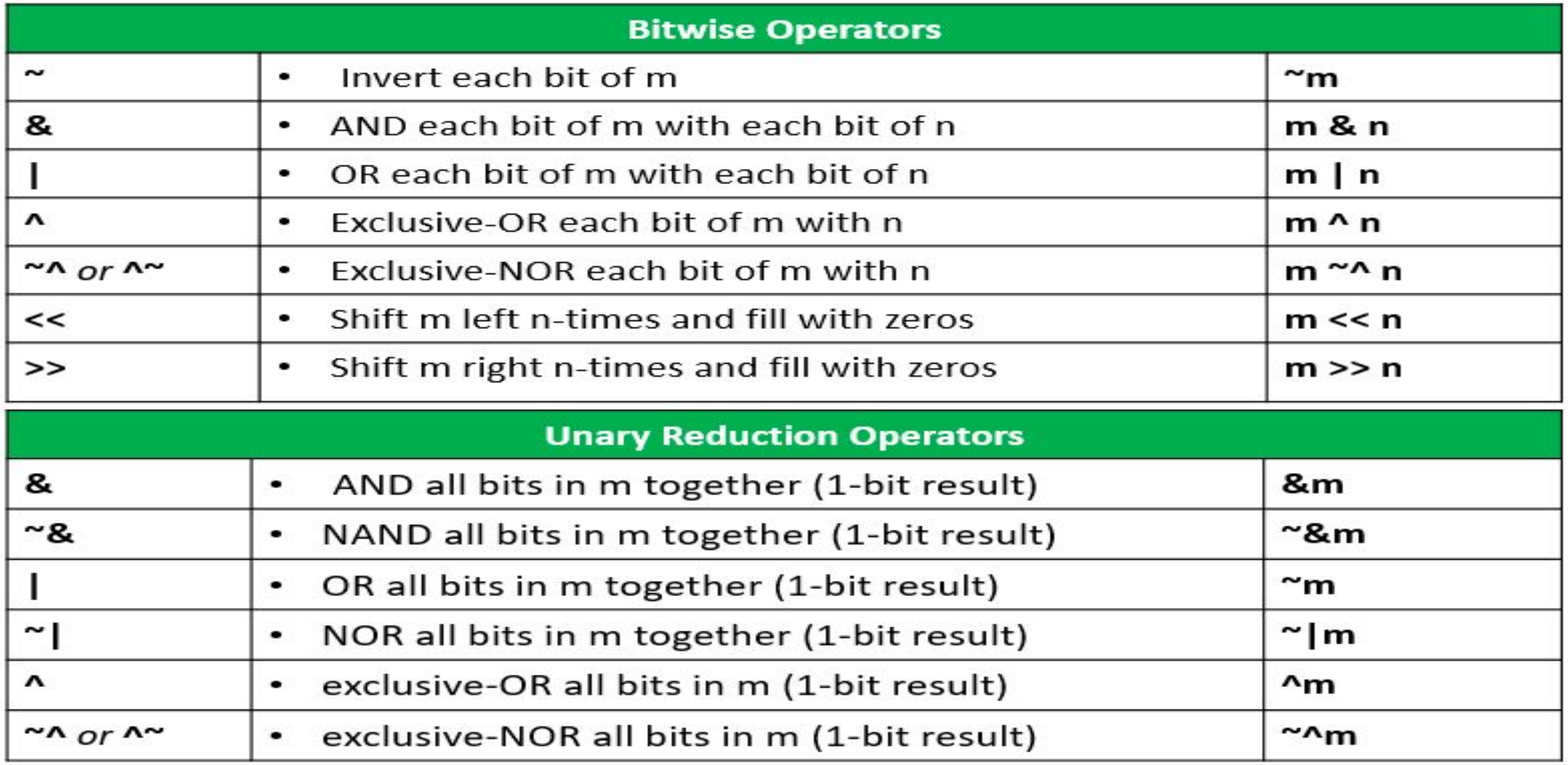
Blocking assignments (=)	Non blocking assignments (<=)
• The evaluation of the expression on the RHS is updated to the LHS variable immediately based on delay value.	• Evaluation of RHS is done but, the assignment to LHS is scheduled to occur in the next evaluation cycle and not immediately.
• When multiple blocking assignments are present in a process, the trailing assignments are blocked from occurring until current assignment is completed	• Multiple non blocking assignments can be scheduled to occur concurrently
• Recommended to use within combinatorial always block	• Recommended to use within the sequential **always** block
• Can be used in procedural assignments like **initial, always** and continuous assignments to nets like assign statements	• Can be used only in procedural block **always**

Wire and Reg

- The **wire** acts as real wires in circuit designs.
- The **reg** is similar to wires, but can store information just like registers

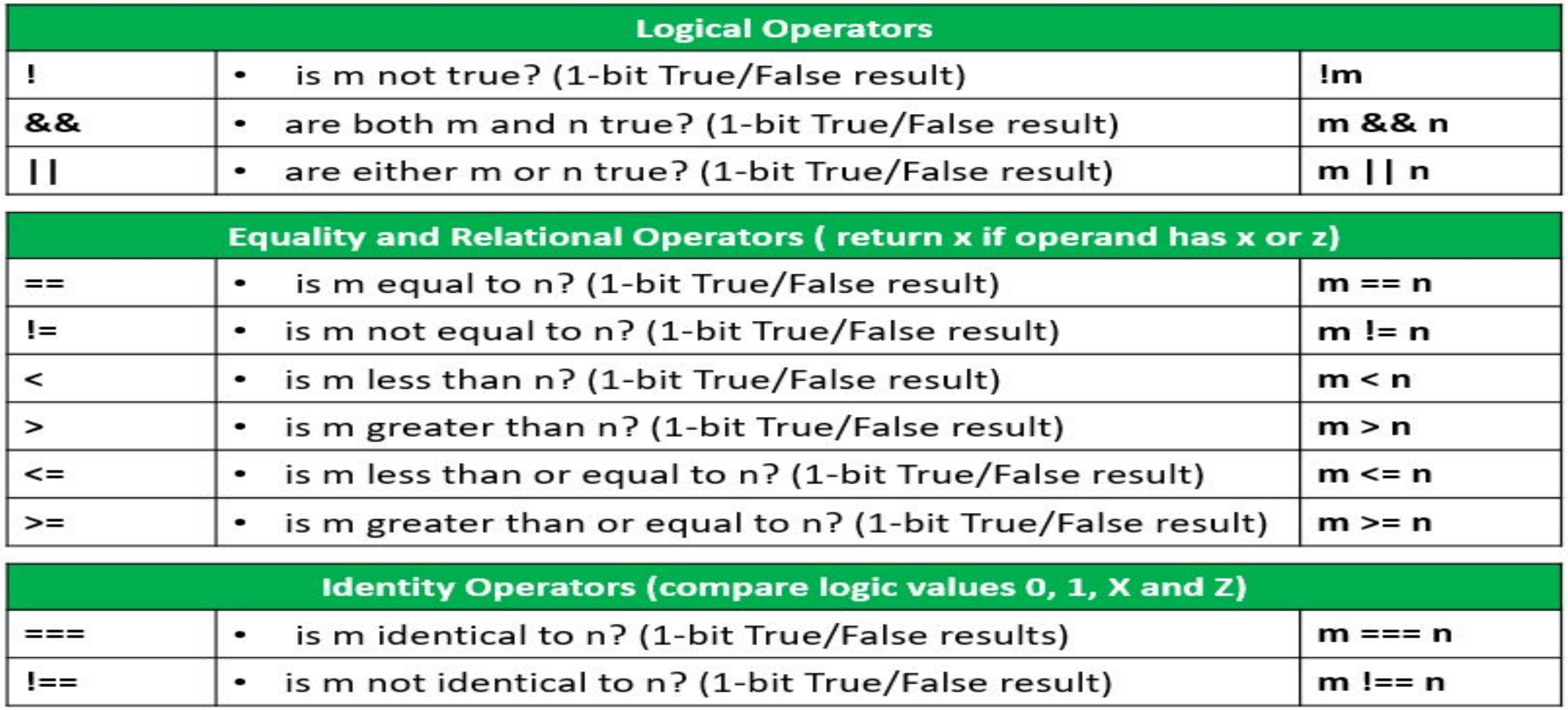
- The declarations for **wire** and **reg** signals should be done inside a module but outside any initial or always block. The initial value of a wire is *z* (high impedance), and the initial value of a reg is *x* (unknown).

Operators



Bitwise Operators		
~	• Invert each bit of m	~m
&	• AND each bit of m with each bit of n	m & n
|	• OR each bit of m with each bit of n	m | n
^	• Exclusive-OR each bit of m with n	m ^ n
~^ *or* ^~	• Exclusive-NOR each bit of m with n	m ~^ n
<<	• Shift m left n-times and fill with zeros	m << n
>>	• Shift m right n-times and fill with zeros	m >> n

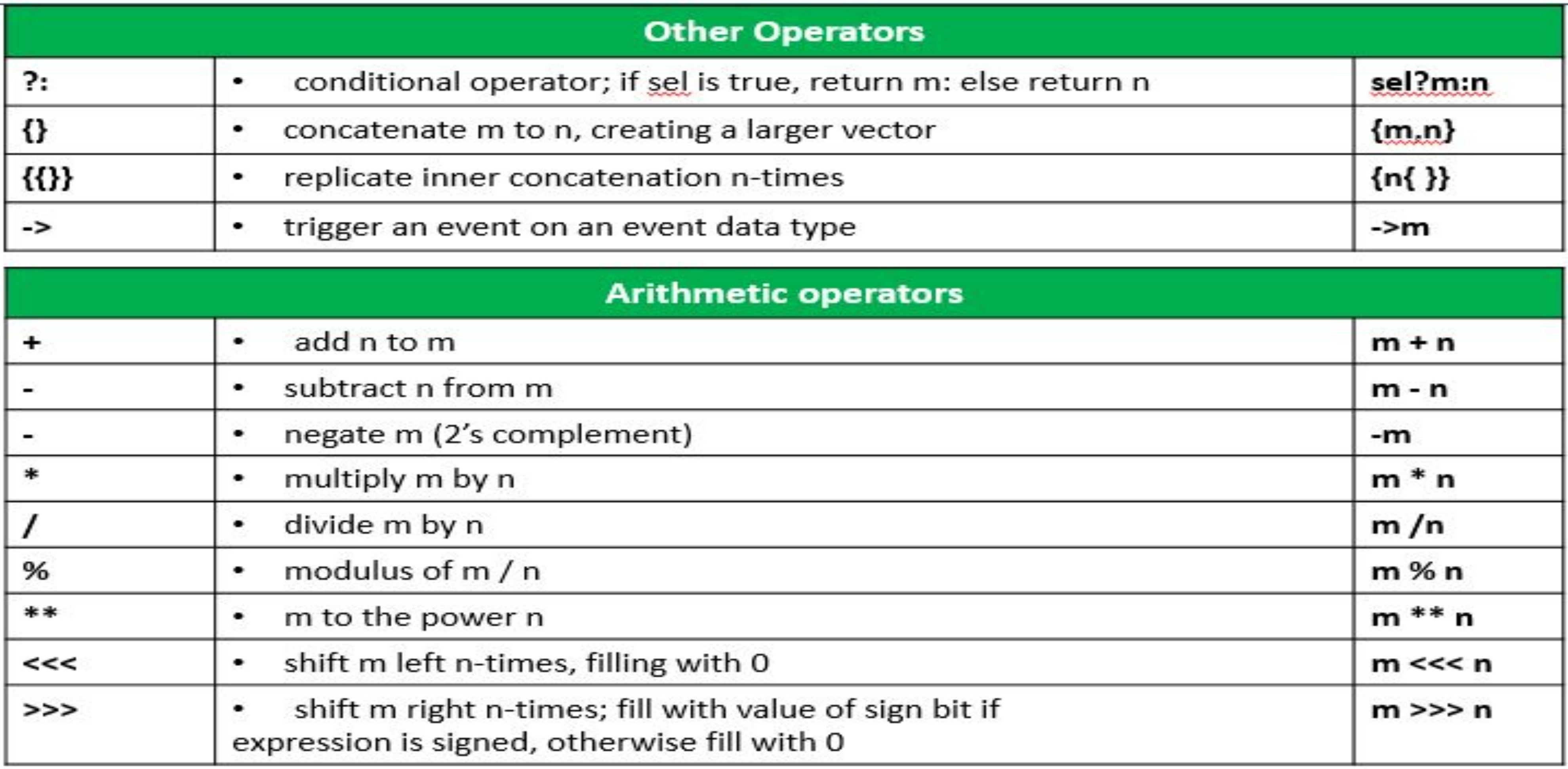
Unary Reduction Operators		
&	• AND all bits in m together (1-bit result)	&m
~&	• NAND all bits in m together (1-bit result)	~&m
|	• OR all bits in m together (1-bit result)	~m
~|	• NOR all bits in m together (1-bit result)	~|m
^	• exclusive-OR all bits in m (1-bit result)	^m
~^ *or* ^~	• exclusive-NOR all bits in m (1-bit result)	~^m



Logical Operators		
!	• is m not true? (1-bit True/False result)	!m
&&	• are both m and n true? (1-bit True/False result)	m && n
||	• are either m or n true? (1-bit True/False result)	m || n

Equality and Relational Operators (return x if operand has x or z)		
==	• is m equal to n? (1-bit True/False result)	m == n
!=	• is m not equal to n? (1-bit True/False result)	m != n
<	• is m less than n? (1-bit True/False result)	m < n
>	• is m greater than n? (1-bit True/False result)	m > n
<=	• is m less than or equal to n? (1-bit True/False result)	m <= n
>=	• is m greater than or equal to n? (1-bit True/False result)	m >= n

Identity Operators (compare logic values 0, 1, X and Z)		
===	• is m identical to n? (1-bit True/False results)	m === n
!==	• is m not identical to n? (1-bit True/False result)	m !== n



Other Operators		
?:	• conditional operator; if sel is true, return m: else return n	sel?m:n
{}	• concatenate m to n, creating a larger vector	{m,n}
{{}}	• replicate inner concatenation n-times	{n{ }}
->	• trigger an event on an event data type	->m

Arithmetic operators		
+	• add n to m	m + n
-	• subtract n from m	m - n
-	• negate m (2's complement)	-m
*	• multiply m by n	m * n
/	• divide m by n	m /n
%	• modulus of m / n	m % n
**	• m to the power n	m ** n
<<<	• shift m left n-times, filling with 0	m <<< n
>>>	• shift m right n-times; fill with value of sign bit if expression is signed, otherwise fill with 0	m >>> n

Operator Precedence				
!	~	+	– (unary)	highest precedence
{}	{{}}			
()				
**				
*	/	%		
+	–		(binary)	
<<	>>	<<<	>>>	
<	<=	>	>=	
==	!=	===	!==	
&	~&			
^	~^			
|	~|			
&&				
||				lowest precedence
?:				

Assignments

- Assign to all possible paths of the **always** block, else it infers latch.

```
always @(b or c or e or f or enable)
 begin
  if(enable)
    a <= b + c; // forgot to assign d in that case
  else
    d <= e + f; // forgot to assign a in that case
 end
```

- Use blocking assignments outside the **always** block
- Use temporary variables for separating such statements and optimize the code
- Here *one adder* is implemented instead of *two.*

```
reg [3:0] sum;
wire enable, a, b;

always @(posedge clk)
  if(enable)
    sum <= a + 1;
  else
    sum <= b + 1;
end
```

```
reg [3:0] sum;
wire enable, a, b;
wire [3:0] temp;

assign temp = enable ? a : b;

always @(posedge clk)
begin
  sum <= temp + 1;
end
```

Incomplete sensitivity list

- The sensitivity list of a combinational always block must include all the inputs used in that block.
- If you forget some signals from the sensitivity list, synthesis will silently assume they are included and generate gates.
- However, the Verilog simulator gives results as per the sensitivity list.

```
always @(b or c or e or f)
 if(enable)
  a = b + c;
 else
  a = e + f;
end
```

✗

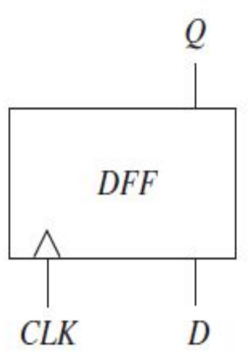
```
always @*
 if(enable)
  a = b + c;
 else
  a = e + f;
end
```

✓

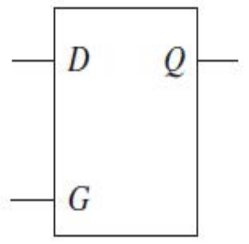
Modeling Flip-Flops Using Always Block

a.Verilog Code for Simple Flip Flop

b. Verilog code for Transparent Latch

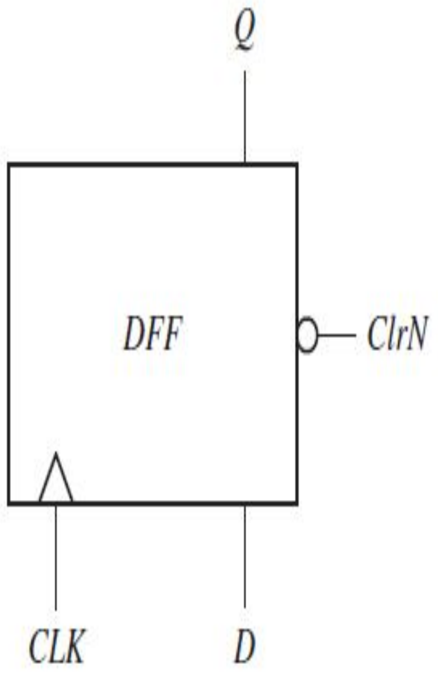


```
always @ (posedge CLK)
begin
  Q <= D;
end
```



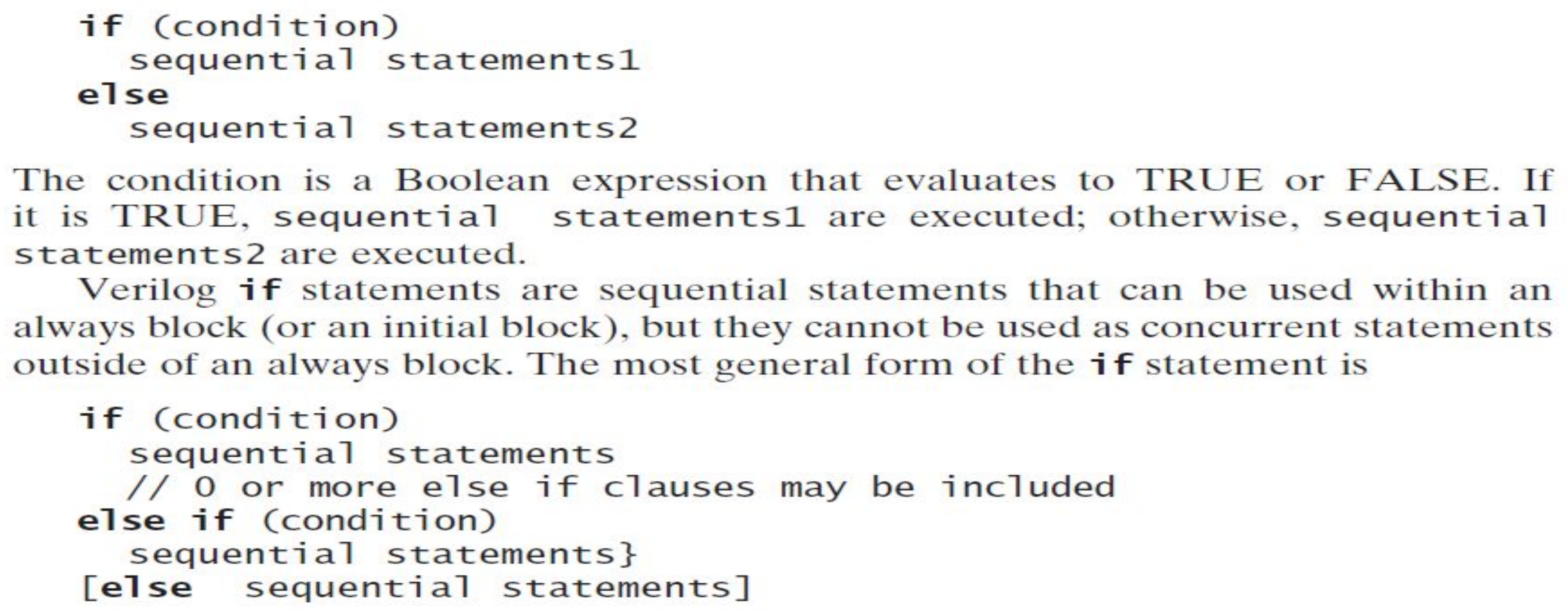
```
always @ (G or D)
begin
 if (G)
   Q <= D;
end
```

Verilog code for DFF with Asynchronous clear



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

Basic form of if



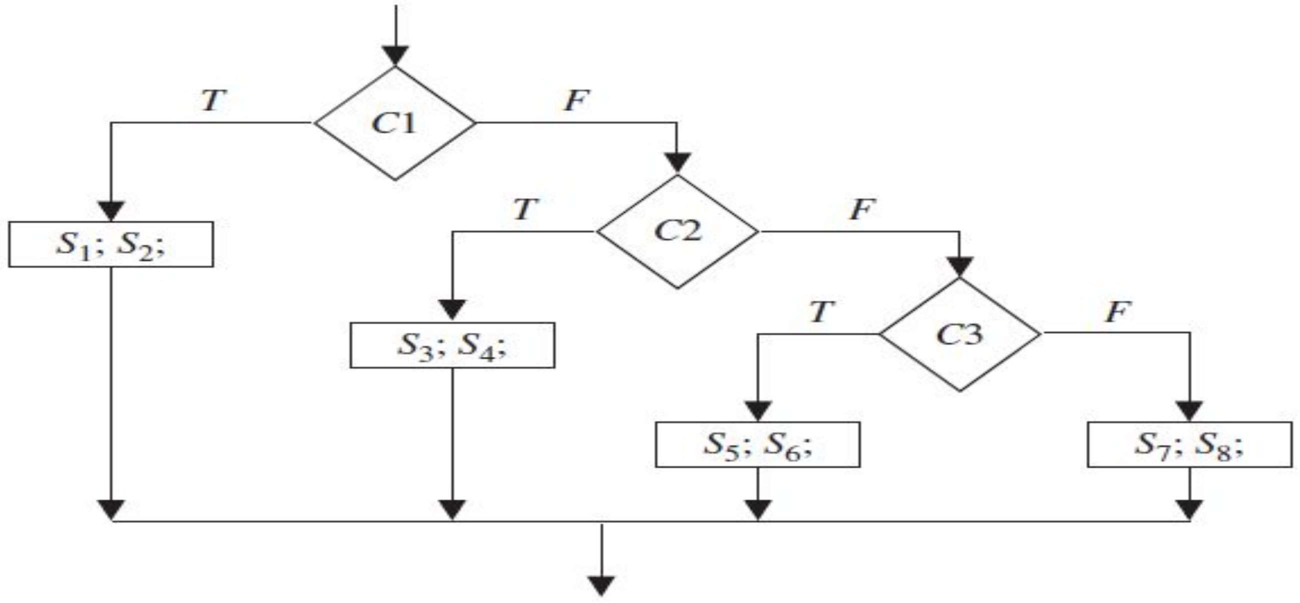
```
if (condition)
  sequential statements1
else
  sequential statements2
```

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

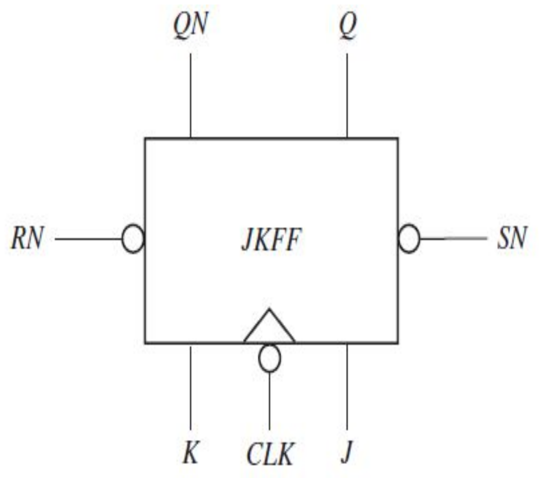
```
if (condition)
  sequential statements
  // 0 or more else if clauses may be included
else if (condition)
  sequential statements}
[else  sequential statements]
```

Nested if else loop



```
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
```

JK Flip Flop Verilog Code



```
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
```

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.

```
always
begin
   sequential-statements
   wait-statement
   sequential-statements
   wait-statement
   . . .
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.

Example

For a half adder, sum and carry can be found using the equations sum = x XOR y ; carry = x AND y. What is wrong with the following code for a half adder that must add if add signal equals 1?

```
always @(*)
begin
if (add == 1)
   sum = x ^ y;
   carry = x & y;
end
```

(a) It will compile but not simulate correctly
(b) It will compile and simulate correctly but not synthesize correctly
(c) It will work correctly in simulation and synthesis
(d) It will not even compile

Answer: (a). This code will compile but will not simulate correctly. The if statement is missing begin and end. Currently only the sum is part of the if statement. The carry statement will get executed regardless of the add signal. This can be corrected by adding begin and end for the if statement. That will result in correct simulation. It can still lead to latches in synthesis. Latches can be avoided by adding else clause or by initializing sum and carry to 0 at the beginning of the always statement.

Example

What is wrong with the following code for a half adder that must add if add signal equals 1?

```
always @(*)
begin
if (add == 1)
   sum = x ^ y;
   carry = x & y;
 else
   sum = 0;
   carry = 0;
end
```

(a) It will compile but not simulate correctly
(b) It will compile and simulate correctly but not synthesize correctly
(c) It will work correctly in simulation and synthesis
(d) It will not even compile

Answer: (d). This code will not even compile due to the missing begin and end inside the if statement. When the compiler gets to the else, it finds that the corresponding if statement is missing. Both if and else clauses need begin and end. Once that is corrected, both simulation and synthesis will work correctly.

Delays

- 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:

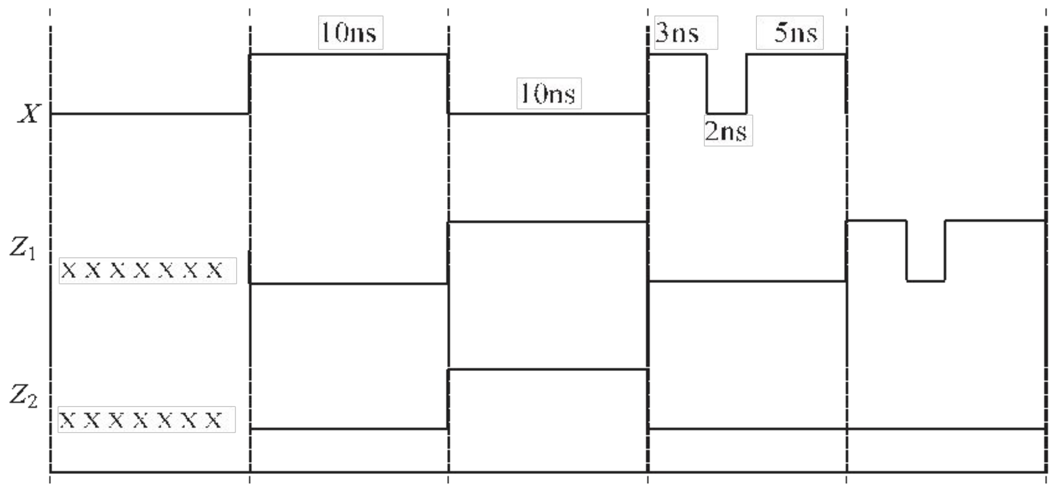
```
// explicit continuous assignment
wire D;
assign #5 D = A && B;

// implicit continuous assignment
wire #5 D = A && B;

// net declaration
wire #5 D;
assign D = A && B;
```

- **Inertial delay** is intended 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 5 ns, a pulse of width 5 ns would pass through, but a pulse of width 4.999 ns 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.

Delays in Verilog

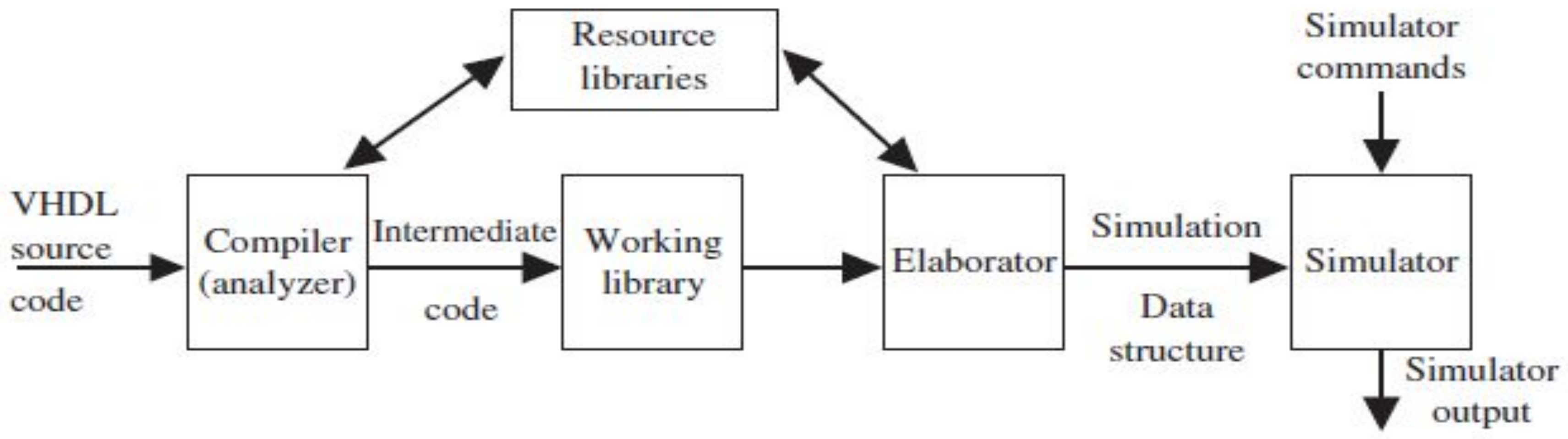
```
always @(X)
begin
  Z1 <= #10 (X);     // transport delay
end
assign #10 Z2 = X;   // inertial delay
```

FIGURE 2-23: Inertial and Transport Delays



2.10 Compilation, Simulation, and Synthesis of Verilog Code

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



- Before the Verilog model of a digital system can be simulated, 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.
- 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.

- A design consists of connected threads of execution or processes. Processes are objects that can be evaluated, that may have state, and that can respond to changes on their inputs to produce outputs. Processes include modules, initial and always procedural blocks, continuous assignments, procedural assignment statements, sys-tem tasks
- Events can occur at different times. In order to keep track of the events and to make sure they are processed in the correct order, the events are kept on an **event queue**, ordered by simulation time. Putting an event on the queue is called **scheduling an event**.
- The Verilog event queue is logically segmented into five different regions:
- **Active event region:** Events that occur at the current simulation time are in this region. Events can be added to any of the five regions but can be removed only

- **Inactive event region:** Events that occur at the current simulation time but that shall be processed after all the active events are processed are in this region. Blocking assignments with zero delays are in this region until they get moved later to the active region.
- **Non-blocking assign update region:** Events that have been evaluated during some previous simulation time but that shall be assigned at this simulation time after all the active and inactive events are processed are in this region.
- **Monitor event region:** Events that shall be processed after all the active, inac-tive, and non-blocking assign update events are processed are in this region. These are the *monitor* events.
- **Future event region:** Events that occur at some future simulation time are in this region. These are the *future* events. Future events are divided into *future inactive events* and *future non-blocking assignment update events*.

- When each Verilog statement is processed, events are added to the various queue regions according to the following convention for each type of statement:
- **Continuous assignment**—evaluate RHS and add to active region as an active update event.
- **Procedural continuous assign**—evaluate RHS and add to active region as an update event.
- **Blocking assignment with delay**—compute RHS and put into future event region for time after delay.
- **Blocking assignment with no delay**—compute RHS and put into inactive region for current time.
- **Non-blocking assignment with no delay**—compute RHS and schedule as non-blocking assign update event for current time if zero delay.
- **Non-blocking assignment with delay**—compute RHS and schedule as non- blocking assign update event for future time if zero delay.
- **$monitor** and **$strobe** system tasks—create monitor events for these system tasks. (These events are continuously reenabled in every successive time step.)

- The processing of all the active events is called a **simulation cycle**.
- For each simulation time, the following actions are performed in order:

1. Process all active update events. (Whenever there is an active update event, the corresponding object is modified and new events are added to the various event queue regions for other processes sensitive to this update.)
2. Then activate all inactive events for that time (and process them because now they are active).
3. Then activate all non-blocking assign update events and process them.
4. Then activate all monitor events and process them.
5. Advance time to the next event time and repeat from step i.

- All of these five steps happen at the same time, but the events occur in the order active, inactive, non-blocking update, and monitor events.

Simulation with Multiple Processes (Initial or Always Blocks)

```
module twoprocess
   reg A, B;

initial
begin
   A = 0;
   B = 0;
end

// process P1 always
@(B) begin
   A <= 1;
   A <= #5 0;
end

// process P2 always
@(A) begin
    if (A)
          B <= #10 ~B;
end
```

Verilog Data Types

Data Types

- Verilog has two main groups of data types: the variable data types and the net data types. These two groups differ in the way that they are assigned and hold values. They also represent different hardware structures.
- The *net* data types can represent physical connections between structural enti- ties, such as gates. Generally, it does not store values. Instead, its value is deter- mined by the values of its drivers, such as a continuous assignment or a gate. A very popular *net* data type is the **wire**. There are also several other predefined data types

- that are part of nets. Examples are **tri** (for tristate), **wand** (for wired and), **wor** (for wired or).
- The *variable* data type is an abstraction of a data storage element. A variable shall store a value from one assignment to the next. An assignment statement in a procedure acts as a trigger that changes the value in the data storage element. A very popular *variable* data type is the **reg**. There are also several other predefined data types that are part of *variables*. Examples are **reg**, **time**, **integer, real,** and **real-time.**

- Unlike VHDL, all data types are predefined by the Verilog language and not by the user. Some of the popular predefined types are
- nets connections between hardware elements (declared with keywords such as **wir**e)
- variables data storage elements that can retain values (declared with the keywords such as **reg**)
- integer an integer is a variable data type (declared with the keyword **integer**)
- real real number constants and real variable data types for floating-point number (declared with the keyword **real**)
- time a special variable data type to store time information (declared with the keyword **time**)
- vectors wire or reg data types can be declared as vectors (multiple bits) (vectors can be declared with [range1 : range2])
- In previous versions of the Verilog standard, the term **register** was used to encompass the **reg**, **integer**, **time**, **real**, and **realtime** types, but starting with the 2005 IEEE 1364Standard, that term is no longer used as a Verilog data type. A net or reg declaration without a range specification shall be considered 1 bit wide and is known as a **scalar**. Multiple bit net and reg data types shall be declared by specifying a range, which is known as a **vector**.
- While VHDL is a strongly typed language where signals and variables of differ- ent types generally cannot be mixed in the same assignment statement, Verilog uses weak typing, which means some mixing of related data types is allowed.