Lecture 3 Introduction to Verilog HDL

Outline

- 3.1 Background of Verilog HDL
- 3.2 An overview of Verilog HDL
- 3.3 Verilog Data Types
- 3.4 Basic Verilog Operators
- 3.5 Behavioral Verilog Blocks: initial, assign and always

Outline

- 3.1 Background of Verilog HDL
- 3.2 An Overview of Verilog HDL
- 3.3 Verilog Data Types
- 3.4 Basic Verilog Operators
- 3.5 Behavioral Verilog Blocks: initial, assign and always

3.1 Background of Verilog HDL

- Hardware Description Language
  - To describe an IC with millions of transistors is a big challenge in 1980's
  - HDL address the issues: RT-level design to gate-level netlist with powerful EDA tools
- Verilog (close to C and widely used by industry)
  - IEEE 1364 Standard in 1995
  - Revised version in 2001, Further version in 2005
- Language Reference Manual:
  - A complete authoritative definition of the Verilog HDL
  - We focus on a subset of the Verilog standard: synthesizable design and basic constructs to build a basic testbench.
- System Verilog
  - A huge set of extensions to Verilog
  - First published as an IEEE standard in 2005.

Outline

- 3.1 Background of Verilog HDL
- 3.2 An Overview of Verilog HDL
- 3.3 Verilog Data Types
- 3.4 Basic Verilog Operators
- 3.5 Behavioral Verilog Blocks: initial, assign and always

3.2.1 Starting with An Example of Verilog Design

- Main aspects needed to describe a circuit
  - Module-endmodule: name of the design block
  - Interfaces: in the following parathesis
  - Functions: encapsulated in the module-endmodule

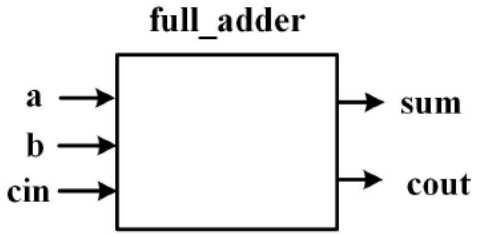
```
1  //Logic Design with Verilog-2005
2  module full_adder (input  a    ,
3                     input  b    ,
4                     input  cin  ,
5                     output sum  ,
6                     output cout);
7  assign sum  = a ^ b ^ cin                    ;
8  assign cout = (a & b) | (a & cin) | (b & cin);
9  endmodule
```

```
1  //A Full Adder Design with Verilog-1995
2  module full_adder (a, b, cin, sum, cout);
3  input              a    ;
4  input              b    ;
5  input              cin  ;
6  output             sum  ;
7  output             cout ;
8
9  assign sum  = a ^ b ^ cin                    ;
10 assign cout = (a & b) | (a & cin) | (b & cin);
11
12 endmodule
```

3.2.1 Starting with An Example of Verilog Design

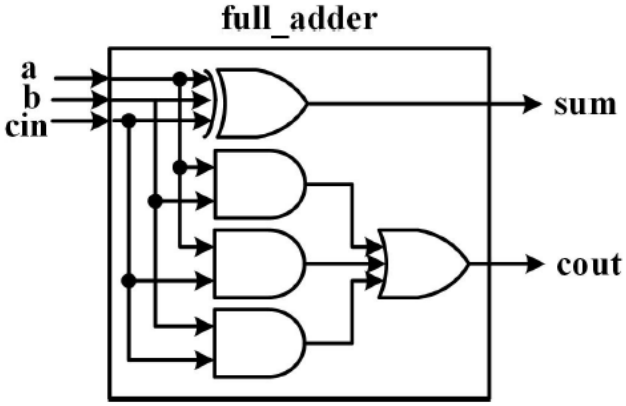
- Syntax
  - In the parentheses, both the name and the direction of each port are listed. All the ports are comma-separated except for the last one.
  - Each line of the code in a Verilog description must terminate with a semicolon

Step 1: Declare IO ports and design name within *module-endmodule*




```
module full_adder (input  a ,
                   input  b ,
                   input  cin ,
                   output sum ,
                   output cout );
endmodule
```

Step 2: Describe functions enclosed by the *module-endmodule*




```
module full_adder (input  a ,
                   input  b ,
                   input  cin ,
                   output sum ,
                   output cout );
assign sum  = a ^ b ^ cin                          ;
assign cout = (a & b) | (a & cin) | (b & cin);
endmodule
```

FIGURE 3.1
Hardware Description Procedure Using Verilog

3.2.1 Starting with An Example of Verilog Design

- Verilog template
  - Starts with *module-endmodule* to name the design and declare the ports.
  - Between *module-endmodule*, three behavioral blocks can be used to construct (and simulate) the hardware: assign, always, initial (section 3.5)
  - Within the blocks:

TABLE 3.1
Summary of Verilog HDL Description

Blocks		**Statements**	**Basic Operators**
assign	**concurrent assign**	Basic Operators	∼ (NOT), & (AND), | (OR), ∧ (XOR), + (Adder)
	conditional assign	? :	
always	**level trigger always**	if-else case-endcase	
	edge trigger always	if-else	
initial		if-else case-endcase for/while/repeat/forever	

  - Basic operators (in section 3.4)
  - Statements (in section 4.1)
  - The data types in Verilog are introduced in section 3.3.

3.2.2 Basic Verilog Description Syntax

- Basic syntax below
  - Module and signal names can be composed of characters, digits, and underscores, but they are case sensitive for characters.
    - For example, ``reset'' and ``Reset'' are considered different signals.
    - Additionally, signal names cannot start with a number. For example, ``2add'' is not a valid module or signal name.
  - Avoid using keywords as module and signal names, such as *module, input, output, endmodule, begin, end, initial, always, assign, if, else, case*, and so on.
  - Verilog ignores white space such as spaces and tabs, so it is helpful to vertically align the design code to make your program readable, concise, and easy to edit.
  - Verilog comments can be single-line or multi-line. Single-line comments are denoted by ``//'' and multi-line comments are enclosed between ``/*'' and ``*/''.

3.2.3 Recommended Coding Style

General Verilog Coding Styles

1. It is highly recommended to use vertical alignment in Verilog code for improved readability and consistency. Align assignment operators = and/or <=, commas to separate IO ports, semicolons at the end of each line, and the first letters of inputs and outputs. Utilizing the Vim editor's visual block editing feature can greatly enhance coding efficiency and overall experience when writing Verilog code in a vertically aligned manner.
2. It is advisable to list all inputs and outputs on separate lines, with each line dedicated to a single port. This practice makes the port declarations more readable and facilitates design instances and connections.
3. To enhance code clarity and understanding, use meaningful names for signals. For instance, consider using "en" for enable, "clk" for clock, and "rst" for reset.
4. To maintain consistency, ensure that the module name, instance name, and file name are related. For instance, if the design module is named "full_adder", the file name should be "full_adder.v". When instantiating the design, use the related name such as "u_full_adder" by adding "u_" (short form of unit) in front. If you need to instantiate the design multiple times, use "u1_", "u2_", and so on, to maintain a coherent and systematic naming convention.
5. At the outset of the Verilog text, include essential comments like the design title, description, author, date, version, and revision history. Providing this information helps in identifying and understanding the purpose and context of the design.
6. Within the design code, incorporate as many comments as possible to enhance comprehensibility and maintainability. Meaningful comments clarify the code's functionality, making it easier for both the original developer and others to comprehend and modify the code effectively.

Outline

- 3.1 Background of Verilog HDL
- 3.2 An Overview of Verilog HDL
- 3.3 Verilog Data Types
- 3.4 Basic Verilog Operators
- 3.5 Behavioral Verilog Blocks: initial, assign and always

3.3.1 Verilog Constant

- Verilog Constant: N'B (size and base)
  - N denotes the number of bits, and B denotes the base including b for binary, d for decimal, o for octal, and h for hexadecimal.
  - In the case of a single bit constant, the binary format is recommended. And in the case of a vector, the hexadecimal format is recommended. The counter/timer usually uses decimal as the natural format.
    - Underscore can be used to between every four binary digits

TABLE 3.2

Examples of Verilog Constants

Number	Bits	Base	Decimal	Stored
3'b101	3	binary	5	101
8'b11	8	binary	3	0000_0011
8'b1010_1011	8	binary	171	1010_1011
3'd6	3	decimal	6	110
8'hac	8	hexadecimal	172	1010_1100

3.3.2 Verilog Data Types – Wire and Reg

- Synthesizable Verilog Data Types
  - Only the wire and reg data types are recommended to synthesizable Verilog designs.
  - Notice that the data width of a vector is usually written from the most-significant-bit (MSB) to the least-significant-bit (LSB).
  - In Verilog design, a part of a vector can be used to the design logic.
    - For example, a[7] represents the MSB of the vector a, a[0] represents the LSB of the vector a, and a[3:0] indicates the least significant 4 bits of signal a.

```
1  wire  a       ;
2  reg   b       ;
3  wire  [7:0]  c;
4  reg   [7:0]  d;
```

3.3.3 Hardware Related Data Types

- Synthesis results of wire and reg
  - Wire: either an internal wire or an output of a combinational logic or a latch
  - Reg: an output of a latch or a register.
- Design rules
  - The LHS signal of an *assign* block must be declared as a wire
  - The LHS signal of an *always* or *initial* blocks must be declared as reg.
- Other data types defined by Verilog Standard
  - integer, real, time, etc.
  - Can be used to design a testbench

3.3.2 Synthesizable Verilog Data Types

- Design rules
  - Input and output ports are default wires
  - wire f = a|b; is recommended

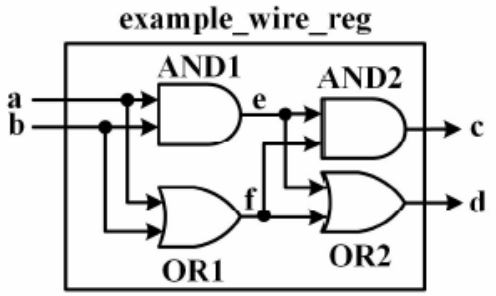


FIGURE 3.2
An Example of Wire and Reg Declaration

```
1  //An Example of Using Wire and Reg Date Typ
2  module example_wire_reg (input         a   ,
3                           input         b   ,
4                           output        c   ,
5                           output   reg  d );
6
7  wire e;
8  reg  f;
9
10 assign e  = a & b      ;  //e must be a wire;
11 always @(a, b) begin
12   f <= a | b;           // f must be a reg;
13 end
14
15 assign c  = e & f      ;  //c output is in default wire data type;
16 always @(e, f) begin
17   d <= e | f;           //d output should be declared as a reg;
18 end
19
20 endmodule
```

3.3.3 Mismatched Data Assignment

- Bit Width Mismatch
  - LHS bit width > RHS bit width, only the lower bits of LHS signal will be assigned with the value of the RHS signal.
    - Notice that the line 1 and line 5 can be combined and written as ``wire [7:0] a = 8'h12'' under Verilog-2005 Standard.
  - LHS bit width < RHS bit width , the LHS signal will be assigned with the value of the lower bits of the RHS signal.
  - Though the simulator won't report an error, the mismatched data assignment is not recommended.
  - The warning reported by the simulator won't affect the simulation, but all the warning should be taken care by the designers.

```
1  wire [7 :0]   a       ;
2  wire [15:0]   b       ;
3  wire [3 :0]   c       ;
4
5  assign a = 8'h12;
6  assign b = a     ;    // b is 16'hxx12
7  assign c = a     ;    // c is 4'h2;
```

3.3.3 Mismatched Data Assignment

Design Rules – Data Types

1. Specify the bit width and base when using Verilog constants. Although the default constant is in decimal, it is highly recommended to provide a complete representation, such as 4'd15 instead of just 15.
2. For synthesizable designs, only use *wire* and *reg* data types, although other data types like integer can be synthesized.
3. Address and rectify all the mismatched warnings that arise during simulation. Although mismatched issues may pass the compilation stage, they can trigger simulation warnings that might ultimately result in design feature failures.
4. When designing synthesizable circuits, it's crucial to avoid using unknown states for signals and IOs. All bits of signals and IOs must be explicitly represented by a specific value, which can be either binary zero or one. For tri-state buffers, you should use the high-impedance state (high-Z) where appropriate.
5. When designing a testbench, initialize all inputs of the design circuit with either binary zeros or ones. Even though some inputs may not be utilized for certain test cases, it is recommended to provide specific initial values to avoid unexpected behavior during simulation.

Outline

- 3.1 Background of Verilog HDL
- 3.2 An overview of Verilog HDL
- 3.3 Verilog Data Types
- 3.4 Basic Verilog Operators
- 3.5 Behavioral Verilog Blocks: initial, assign and always

3.4.1 Equality and Inequality

TABLE 3.3
Equality and Inequality

Operators	Verilog	Examples
Equality	==	if(a[3:0]==4'h8)
Inequality	!=	if(a[3:0]!=4'h8)

- Verilog operators
  - Mimic digital logic for operating on all data types to produce outputs.
  - Must be included in Verilog blocks.
- 3.4.1 Equality and Inequality
  - if(sel==1'b1) can be represented as if(sel)
  - if(sel!=1'b1) can be represented as if(~sel)

```
1  module example_equ_inequ();
2  wire a, b, sel;
3  reg  c, d     ;
4  always @(a, b, sel) begin
5    if (sel) begin
6      c <= a;
7    end else begin
8      c <= b;
9    end
10 end
11
12 always @(a, b, sel) begin
13   if (~sel) begin
14     d <= a;
15   end else begin
16     d <= b;
17   end
18 end
19 endmodule
```

Recommended Coding Styles – Equality and Inequality

1. It is recommended to use a simple Boolean statement "if(~ a)", to describe a condition of "if(a==1'b0)".
2. Likewise, "if(a)" is recommended to describe a condition of "if(a==1'b1)".

3.4.2 Bit-Wise Logic Operators

- 3.4.2 Bit-Wise Logic Operators
  - Single bit
  - Vectors: take each bit in one operand and perform the operation with the corresponding bit in the other operand.

TABLE 3.4
Basic Logic Operators

Operators	Verilog	Examples
Bit-wise NOT	$\sim$	$\sim a$; $\sim a[3:0]$
Bit-wise AND	&	$a\&b$; $a[3:0]\&b[3:0]$
Bit-wise OR	|	$a\|b$; $a[3:0]\|b[3:0]$
Bit-wise XOR	$\wedge$	$a\wedge b$; $a[3:0]\wedge b[3:0]$

```
1  module example_logic_op();
2  //--- Bit-Wise Logic Design Code---//
3  reg  [3:0] a,b;
4  wire [3:0] c = a & b;     // AND
5  wire [3:0] d = a | b;     // OR
6  wire [3:0] e = a ^ b;     // XOR
7  wire [3:0] f = ~(a & b); // NAND
8  wire [3:0] g = ~(a | b); // NOR
```

```
10 //--- Testbench ---//
11 initial begin
12   a = 4'b1010;
13   b = 4'b0101;
14   #10;
15   $display ("The outputs c=%b, d=%b, e=%b, f=%b, g=%b
16       with Inputs a=%b, b=%b", c, d, e, f, g, a, b);
17 /* Printed log: # The outputs c=0000, d=1111, e=1111,
18 f=1111, g=0000 with inputs a=1010, b=0101 */
19
20   a = 4'b1100;
21   b = 4'b1101;
22   #10;
23   $display ("The outputs c=%b, d=%b, e=%b, f=%b, g=%b
24       with Inputs a=%b, b=%b", c, d, e, f, g, a, b);
25 /* Printed log: # The outputs c=1100, d=1101, e=0001,
26 f=0011, g=0010 with Inputs a=1100, b=1101 */
27 end
28 endmodule
```

- 3.4.2 Bit-Wise Logic Operators
  - Though logical NOT (!), logical AND (&&), and logical OR (||) are defined by Verilog Standard, the bit-wise logic operators can be used as logic functions as well.

Recommended Coding Styles – Bit-Wise Operators

1. Bit-wise operators can be used for both bit-wise and logical operations since they have identical functions in many cases.
2. The Verilog standard allows for mixing different data types and logical operations. For example, the expression "if(a & (b==4'h4) & |c)" means that the *if* statement is TRUE only if all three conditions ("a==1'b1", "b==4'h4", and the result of a Reduction OR operation on the multi-bit signal "c" is binary one) are TRUE.

3.4.3 Reduction on Multi-Bit Signals

- 3.4.3 Reduction on Multi-Bit Signals
  - LHS is a single-bit signal and the RHS is a vector
  - The operator takes all bits in the RHS operand one by one. Below is a design example using the reduction operators.

TABLE 3.5
Reduction on Vectors

Operators	Verilog	Examples
Reduction AND	&	$\&a[3:0]$
NOT Reduction AND	$\sim\&$	$\sim\&a[3:0]$
Reduction OR	|	$\|a[3:0]$
NOT Reduction OR	$\sim\|$	$\sim\|a[3:0]$
Reduction XOR	$\wedge$	$\wedge a[3:0]$
NOT Reduction XOR	$\sim\wedge$	$\sim\wedge a[3:0]$

```
1  module example1_reduction ();
2  //--- Reduction on Multi-Bit Signals Design Code---//
3  reg [3:0] a;
4  wire b = &a;     // Reduction AND
5  wire c = |a;     // Reduction OR
6  wire d = ^a;     // Reduction XOR
7  wire e = ~&a;    // NOT Reduction AND
8  wire f = ~|a;    // NOT Reduction OR
```

3.4.3 Reduction on Multi-Bit Signals

```
module example1_reduction ();
//--- Reduction on Multi-Bit Signals Design Code---//
reg [3:0] a;
wire b = &a;     // Reduction AND
wire c = |a;     // Reduction OR
wire d = ^a;     // Reduction XOR
wire e = ~&a;    // NOT Reduction AND
wire f = ~|a;    // NOT Reduction OR

endmodule
```

```

//--- Testbench ---//
initial begin
  a = 4'b1010;
  #10;
  $display ("Outputs b=%b, c=%b, d=%b, e=%b, f=%b,
            with Input a=%b", b, c, d, e, f, a);
/*Printed log: # Outputs b=0, c=1, d=0, e=1, f=0,
with Input a=1010*/

  a = 4'b1100;
  #10;
  $display ("Outputs b=%b, c=%b, d=%b, e=%b, f=%b,
            with Input a=%b", b, c, d, e, f, a);
/*Printed log: # Outputs b=0, c=1, d=0, e=1, f=0,
with Input a=1100*/

  a = 4'b1111;
  #10;
  $display ("Outputs b=%b, c=%b, d=%b, e=%b, f=%b,
            with Input a=%b", b, c, d, e, f, a);
/*Printed log: # Outputs b=1, c=1, d=0, e=0, f=0,
with Input a=1111*/

  a = 4'b0000;
  #10;
  $display ("Outputs b=%b, c=%b, d=%b, e=%b, f=%b,
             with Input a=%b", b, c, d, e, f, a);
/*Printed log: # Outputs b=0, c=0, d=0, e=1, f=1,
with Input a=0000*/
end
endmodule
```

3.4.3 Reduction on Multi-Bit Signals

- Reduction on Multi-Bit Signals
  - Find a vector with all the binary ones or zeros
    - if(a==32'h0) can be simplified as if(~|a)
    - if(b==32'hffff_ffff) can be simplified as if(&b)
    - How to design the function that vector ``a'' not all 0s and/or not all 1s

```
1  module example2_reduction ();
2  //--- Reduction on Multi-Bit Signals Design Code---//
3  reg [31:0] a,b;
4  reg        c,d;
5  always @(a) begin
6    if(~|a) begin // Will be true when all bits are zeros
7      c <= 1'b1;
8    end else begin
9      c <= 1'b0;
10   end
11 end
12
13 always @(b) begin
14   if(&b) begin  // Will be true when all bits are ones
15     d <= 1'b1;
16   end else begin
17     d <= 1'b0;
18   end
19 end
```

3.4.3 Reduction on Multi-Bit Signals

- Reduction on Multi-Bit Signals
  - if(a==32'h0) can be simplified as if(~|a)
  - if(b==32'hffff_ffff) can be simplified as if(&b)

```
21  //--- Testbench ---//
22  initial begin
23    a = 32'h0;
24    b = 32'hffff_ffff;
25    #10;
26    $display ("Outputs c=%b, d=%b, with Input a=%h, b=%h",
27                c, d, a, b);
28  /*Printed log: # Outputs c=1, d=1,
29  with Input a=00000000, b=ffffffff*/
30
31    a = 32'h0123_4567;
32    b = 32'h89ab_cdef;
33    #10;
34    $display ("Outputs c=%b, d=%b, with Inputs a=%h, b=%h",
35                c, d, a, b);
36  /*Printed log: # Outputs c=0, d=0,
37  with Inputs a=01234567, b=89abcdef*/
38  end
39  endmodule
```

```
1  module example2_reduction ();
2  //--- Reduction on Multi-Bit Signals Design Code---//
3  reg [31:0] a,b;
4  reg        c,d;
5  always @(a) begin
6    if(~|a) begin // Will be true when all bits are zeros
7      c <= 1'b1;
8    end else begin
9      c <= 1'b0;
10   end
11 end
```

```
12
13   always @(b) begin
14     if(&b) begin  // Will be true when all bits are ones
15       d <= 1'b1;
16     end else begin
17       d <= 1'b0;
18     end
19   end
```

3.4.3 Reduction on Multi-Bit Signals

- Reduction on Vectors
  - Reduction OR operation such as ``if(|a)'' can be used to find a vector in which not all the bits are binary 1'b0.
  - And the NOT Reduction AND operation such as ``if(~&a)'' can be used to find a vector that not all the bits are binary ones.

Recommended Coding Styles – Reduction Operators

1. Use a simple NOT Reduction OR, e.g., "if(~|a)", to simplify the equality operation where all bits are zeros, e.g., "if(a==32'h0)".
2. Use a simple Reduction AND, e.g., "if(&a)", to simplify the equality operation where all bits are ones, e.g., "if(a==32'hffff_ffff)".
3. Use a simple Reduction OR, e.g., "if(|a)", to simplify the inequality operation where not all bits are zeros, e.g., "if(a!=32'h0)".
4. Use a simple NOT Reduction AND, e.g., "if(~&a)", to simplify the inequality operation where not all bits are ones, e.g., "if(a!=32'hffff_ffff)".

- 3.4.4 Relational Operators
  - greater than
  - less than
  - greater than and equal to
  - less than and equal to

TABLE 3.6
Greater Than and Less Than

Operators	Verilog	Examples
Greater Than	>	if(a > b)
Less Than	<	if(a < b)
Greater Than or Equal To	>=	if(a >= b)
Less Than or Equal To	<=	if(a <= b)

```
1 //signal a greater than 4'h0 AND less than 4'h8
2 if(a>4'h0 & a<4'h8 )
3
4 //signal a greater than or equal to 4'h0 AND
5 //less than or equal to 4'h8
6 if(a>=4'h0 & a<=4'h8 )
7
8 //signal a less than 4'ha OR b less than 4'hb
9 if(a<4'ha | b<4'hb)
```

3.4.5 Concatenation

- Concatenation
  - Combines all the signals (ports, wire, reg, and constant) in the concatenation operator together as a new vector

TABLE 3.7
Concatenation

Operators	Verilog	Examples
Concatenation	{}	{a, b, c}

```
1  module example_concatenation ();
2  //--- Concatenation Design Code---//
3  reg   [7 :0] a ;
4  reg   [15:0] b ;
5  reg   [7 :0] c ;
6  wire  [31:0] vec1 = {a, b, c}          ;
7  wire  [15:0] vec2 = {b[7:0], b[15:8]};
8  wire  [15:0] vec3 = {2{a}}             ;
9
10 wire  [9:0]   vec4 = {{2{1'b0}}, a}    ;
11 wire  [9:0]   vec5 = {a, {2{1'b0}}}    ;
12
13 //--- Testbench ---//
14 initial begin
15   a = 8'h12    ;
16   b = 16'h3456;
17   c = 8'h78    ;
18   #10;
19   $display ("Outputs vec1=%h, vec2=%h, vec3=%h, vec4=%h,
20             vec5=%h, with inputs a=%h, b=%h, c=%h",
21             vec1, vec2, vec3, vec4, vec5, a, b, c);
22 /* Printed log: Outputs vec1=12345678, vec2=5634, vec3=1212,
23 vec4=012, vec5=048 with inputs a=12, b=3456, c=78 */
24 end
25 endmodule
```

3.4.5 Concatenation

- Multiplications with a variable and a constant

```
1  module example_con_mul ();
2  //--- Concatenation Design Code---//
3  reg  [7 :0] r,g,b ;
4  wire [15:0] rx32 = {3'b0, r, 5'b0};
5  wire [15:0] gx64 = {2'b0, g, 6'b0};
6  wire [15:0] bx8  = {5'b0, b, 3'b0};
7
8  //--- Testbench ---//
9  initial begin
10   r = 8'h12;
11   g = 8'h34;
12   b = 8'h56;
13   #10;
14   $display ("Outputs rx32=%h, gx64=%h, bx8=%h with inputs
15             r=%h, g=%h, b=%h", rx32, gx64, bx8, r, g, b);
16 /*Printed log: Outputs rx32=0240, gx64=0d00, bx8=02b0,
17 with inputs r=12, g=34, b=56 */
18 end
19 endmodule
```

3.4.6 Logical Shift

- Logical Shift
  - Fill the vacated bit positions with zeros.

TABLE 3.8
Logical Shift Operators

Operators	Verilog	Examples
Logical Left shift	<<	a << 2
Logical Right shift	>>	a >> 2

```
1  module example_shift ();
2  //--- Logical Shift Design Code---//
3  reg  [3:0] a;
4  wire [3:0] lls2 = a<<2;
5  wire [3:0] lrs2 = a>>2;
6
7  //--- Testbench ---//
8  initial begin
9    a   = 4'h1;
10   #10;
11   $display ("Outputs lls2=%b, lrs2=%b
12                with inputs a=%b", lls2, lrs2, a);
13 /* Printed log: # Outputs lls2=0100, lrs2=0000
14                  with inputs a=0001 */
15 end
16 endmodule
```

3.4.7 Arithmetic Operators

- Arithmetic Operators
  - The addition operator finds widespread application in both FPGA and ASIC designs.
  - Although the Verilog standard encompasses subtraction, multiplication, and division operators, it is advisable to leverage arithmetic IPs (intellectual properties) furnished by FPGA tools or ASIC foundries for proficient integration.
  - This approach guarantees enhanced performance and optimal resource utilization specifically tailored to FPGA/ASIC implementations.

TABLE 3.9
Arithmetic Operators

Operators	Verilog	Examples
Addition	+	a + b
Subtraction	−	a - b
Multiplication	*	a * b
Division	/	a / b

Outline

- 3.1 Background of Verilog HDL
- 3.2 An overview of Verilog HDL
- 3.3 Verilog Data Types
- 3.4 Basic Verilog Operators
- 3.5 Behavioral Verilog Blocks: initial, assign and always

3.5.1 Synthesizable Block - assign

- Synthesizable Block - assign
  - Concurrent assign and Conditional assign
    - An assign block executes whenever the RHS signals change.
    - The behavior mimics the executions of combinational circuits and latches: whenever the inputs change the outputs update
  - For both concurrent assign and conditional assign
    - The LHS signals must be declared as wire;
    - Can be paralleled with other assign, always, and initial;
    - Cannot ``nested".
- Synthesizable Block – always
  - Level-Trigger always
  - Edge-Trigger always
- Unsynthesizable Block - initial

3.5.1.1 Concurrent assign

- Concurrent assign
  - Can be used to describe:
    - Simple wire connection (as shown in line 3 below)
    - A gate operation (shown in line 4)
    - Multiple gate operations (shown in line 5).
  - The LHS signals of an assign block, for both concurrent assign and conditional assign, must be declared as wire.
  - In the Verilog IEEE 2005 standard, the wire declaration and the assign block description can be combined as
    - wire c1=a; as an alternative design in the second line
    - wire c2=a&b; as an alternative design in the third line
    - wire c3=a&b|c;" as an alternative design in the fourth line.

```
1 // Examples of Concurrent Assignments
2 wire   c1, c2, c3;
3 assign c1 = a     ; // Ex1: wire c1=a     ; Wire connection ;
4 assign c2 = a&b   ; // Ex2: wire c2=a&b   ; AND Gate         ;
5 assign c3 = a&b|c; // Ex3: wire c3=a&b|c; AND and OR Gates;
```

3.5.1.2 Conditional assign

TABLE 3.10
Conditional *assign*

Operators	Verilog	Examples
Conditional	? :	assign d = a ? b: c;

- Conditional assign
  - Can be used to describe:
    - Multiplexer,
    - Tri-state buffer
    - Latch
    - etc.

```
1  //a) A 2-to1 Multiplexer Using Conditional Assignment
2  module example_conditional_mux (input  sel, a, b,
3                                   output c        );
4  assign c = ~sel ? a: b;
5  endmodule
7  //b) A Tri-state Buffer Using Conditional Assignment
8  module example_conditional_tri (input        en ,
9                                  input  [1:0] a  ,
10                                 output [1:0] b  );
11 assign b = en ? a : 2'bzz ;  // Tri-state Buffer
12 endmodule
13
14 //c) A Latch Using Conditional Assignment
15 module example_conditional_latch (input  en, d ,
16                                   output q     );
17 assign q = en ? d : q ;  // latch
18 endmodule
19
20 //d) A 2-Stage Multiplexer Using Conditional Assignment
21 module example_conditional_nested (input sel0, sel1,
22                                    input  a, b, c,
23                                    output          );
24 assign d = ~sel0 ? a :
25                ~sel1 ? b: c ;  // latch
26 endmodule
```

3.5.1.2 Conditional assign

- Conditional assign

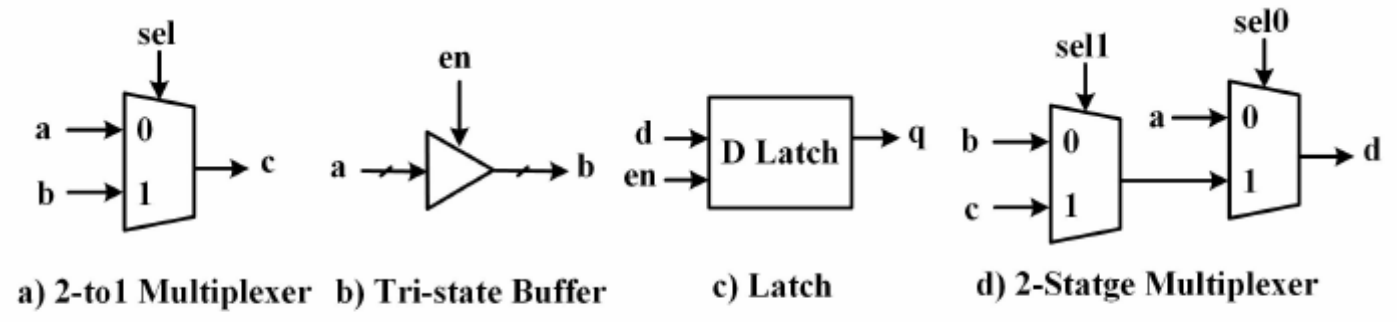


FIGURE 3.3
Examples of Using Conditional Assignment.

Recommended Coding Styles – Conditional Assignment

1. It is strongly recommended to vertically align the early-stage ":" mark with the further-stage "?" mark when using nested conditional operators. This makes the code more readable, especially for multi-stage conditional *assign* blocks.
2. The conditional *assign* block is useful for describing both combinational and sequential circuits. In combinational circuit design, it is typically used to implement multiplexers or tri-state buffers. In sequential circuit design, it can be used to create the storage element of a latch.

3.5.2 Synthesizable Block - always

- Always
  - An *always* block operates continuously. It runs when simulation starts and then restarts when reaches the end of the always block.
  - A level-trigger always
    - Can mimic the executions of combinational circuits and/or latches that the outputs update when the inputs change.
    - All level-trigger events must be included in a trigger list, denoted as @().
    - Any missing signals in the trigger list:
      - May cause simulation failure.
      - May generate unwanted latches while synthesizing the Verilog design.

TABLE 3.11
always Blocks

Blocks	Verilog	Examples
Level Trigger always	always @()	always @(a, b, c, d)
Edge Trigger always		always @(posedge clk, negedge rst)

3.5.2 Synthesizable Block - always

- Always
  - An edge-trigger always
    - Can mimic the behavior of flip-flops/registers that the output follows the input when an active clock edge occurs.
    - Only the edge events of clock and reset can be listed.
  - For both level-trigger and edge-trigger always blocks,
    - The LHS signals must be declared as reg;
    - Can be paralleled with other assign, always, and initial blocks;
    - Cannot ``nested".

TABLE 3.11
always Blocks

Blocks	Verilog	Examples
Level Trigger always	always @()	always @(a, b, c, d)
Edge Trigger always		always @(posedge clk, negedge rst)

3.5.2.1 Level-Trigger always

- Level-trigger always
  - An example in lines 1-5 with complete trigger list
    - Both signals ``a'' and ``b'' must be in the trigger list.
    - The LHS signal ``c1'' and ``c2'' must be declared as a reg.
  - An example in lines 7-11 with incomplete trigger list
    - Simulation: It will cause a simulation error when changing values for signal ``a" because the event won't trigger the AND operation.
    - Synthesis:
      - An unwanted latch may be generated
      - For most modern synthesis tools, they can make up the trigger list and fix the problem of missing signals.

```
1  // An Example of A Level Trigger List
2  reg c1;
3  always @(a, b) begin
4    c1 <= a & b;
5  end
6
7  // An Example of Missing Signals in A Level Trigger List
8  reg c2;
9  always @(b) begin
10   c2 <= a & b;
11 end
```

3.5.2.1 Level-Trigger always

- Level-trigger always
  - #5, delay by 5 timescale unit; #10, delay by 10 timescale unit
  - Nonsynthesizable
  - All the reg data must be initialized otherwise they are unknown.

```
1  // Examples of Clock Generator, Only For Testbench
2  reg clk1, clk2;
3  always  #5   clk1 = ~clk1;
4  always  #10  clk2 = ~clk2;
5
6  initial begin
7    clk1=1'b0;
8    clk2=1'b0;
9  end
```

Design Rules – Level-Trigger always Block

1. Due to its simplicity and enhanced readability, employing an *assign* block is strongly recommended for detailing combinational circuits. Nevertheless, an exception arises when dealing with an abundance of nested conditional operators. In such scenarios, utilizing a level-triggered *always* block in conjunction with a *case* statement proves to be a more appropriate approach. The upcoming sections will introduce the *case* statement utilized in *always* blocks.
2. It is crucial to include a complete trigger list when employing level-triggered *always* blocks. The absence of any signal in the trigger list may lead to simulation failures and result in the synthesis generating unwanted latches.

3.5.2.2 Edge-Trigger always

- Edge-trigger Always
  - Combined the AND gate and a flip-flop

```
1  // An Example of An Edge Trigger List
2  reg c1;
3  always @(posedge clk) begin
4    c1 <= a & b;
5  end
6
7  // A Better Coding for the Same Function
8  wire nxt_c2 = a & b;
9  reg c2;
10 always @(posedge clk) begin
11   c2 <= nxt_c2;
12 end
```

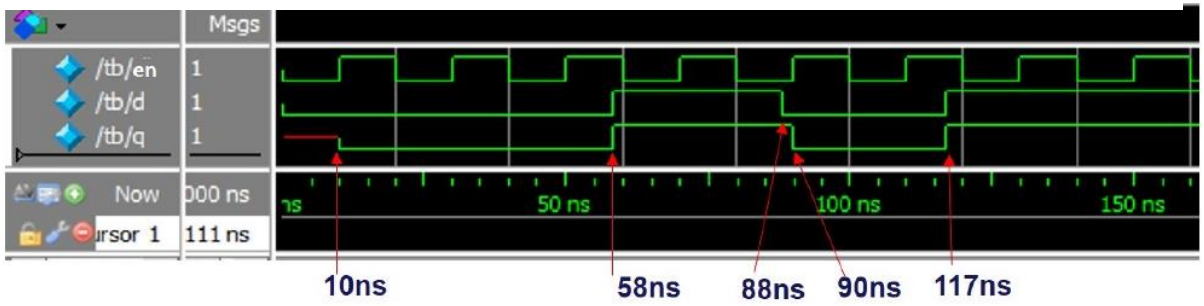
Design Rules – Edge-Triggered always Block

1. Only clock-edge and reset-edge events can be included in the edge-triggered *always* block. Listing other active signal edges in the trigger list might work for simulation purposes, but it is not permitted for practical hardware circuits.
2. It is highly recommended to segregate the design of combinational circuits and the descriptions of sequential registers into separate blocks. When working within edge-triggered *always* blocks, focus solely on the register description.

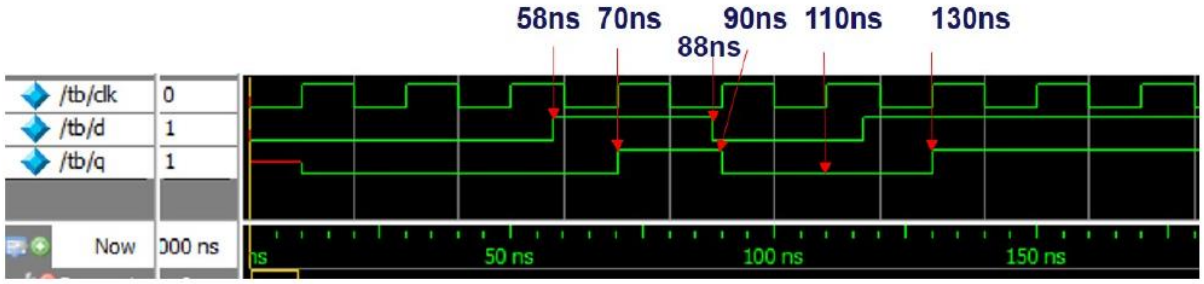
3.5.2.3 Simulation Difference Between Level Trigger and Edge Trigger

- Level trigger (D Latch Simulation)
- Edge trigger (DFF Simulation)

```
1  always @(en, d) if(en) q <= d;   // D Latch
2  always @(posedge clk)  q <= d;   // D Register
```



(a) Simulation Waveform with Level-Triggered Design



(b) Simulation Waveform with Edge-Triggered Design

FIGURE 3.4
Comparison Between Level-Triggered and Edge-Triggered always Blocks

3.5.3 Unsynthesizable Block - initial

- Initial
  - Initial is the behavioral block operating only ONCE.
  - Initial starts at time 0 (beginning of operation) and terminates when reaches the end.
  - For an initial block
    - the LHS signals must be declared as reg;
    - can be paralleled with other assign, always, and initial;
    - cannot ``nested''.
  - For example,
    - In line 2, all the signals on the LHS in the initial block, ``rst", ``clk1", and ``clk2" are declared as reg.

```
1  // An Example of Initial
2  reg  rst, clk1, clk2;
3  initial begin
4    rst = 1'b0; clk1 = 1'b0; clk2 = 1'b0;
5    #10 rst = 1'b1;
6  end
```