Verilog Introduction

LECTURE 14-15
DIGITAL DESIGN

SARANG DHONGDI

Introduction

- Verilog is Hardware Description Language (HDL) used to design digital systems.
- Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers.

Module

- Module are the building blocks of Verilog designs.
- Design hierarchy can be created by instantiating modules in other modules.
- Modules cannot contain definitions of other modules.

Basic Syntax of Module Definition

```
module module_name (list_of_ports);
    input/output declarations;
    data declarations;
    operational statements;
endmodule
```

Data types

- Verilog Language has two primary data types.
- **Nets** - represents structural connections between components.
  - Ex. "wire" – Used for interconnection
  - Ex. supply0, supply1 - power supply connections
- **Registers** - represent variables used to store data.
  - Ex. reg, integer, time, real.

Primitive Gates

- Primitive logic gates (instantiations):
  - and G (out, in1, in2);
  - nand G (out, in1, in2);
  - or G (out, in1, in2);
  - nor G (out, in1, in2);
  - xor G (out, in1, in2);
  - xnor G (out, in1, in2);
  - not G (out1, in);
  - buf G (out1, in);

Circuit description

- *Gate level modeling*
- *Dataflow modeling*
- *Behavioral modeling*

Gate level modeling

- Logic gates can be used to design logic circuits
- Basic Logic gates defined by Verilog – Primitives
- not, and, or, xor, nand, nor, xnor, buf

Gate level modeling example

```
module Full_adder_G (S, Co, A, B, Ci);
  output S, Co;
  input A, B, Ci;
  wire w1, w2, w3;
  xor G1 (w1, A, B);
  and G2 (w2, A, B);
  xor G3 (S, w1, Ci);
  and G4 (w3, w1, Ci);
  or G5 (Co, w2, w3);
endmodule
```

Specifying Constant Values

<size>' <base> <number>

Examples:

- 8'b01110011 // 8-bit binary number
- 12'hA2D // 12-bit hexadecimal number
- 1'b0 // Logic 0
- 1'b1 // Logic 1

Logic Values

Logic Value	Meaning
0	Logic-0, low, or FALSE
1	Logic-1, high or TRUE
X	Unknown (or don't care), uninitialized, contention
Z	High impedance, floating

Truth table for primitive gates

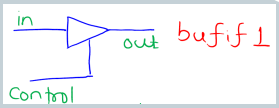
and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

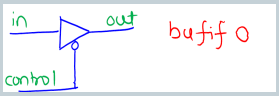
xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

not	Input	Output
	0	1
	1	0
	x	x
	z	x

Three-state gates



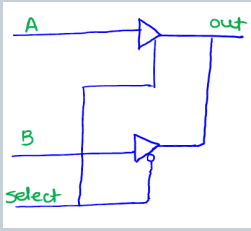
bufif1 - output=input
if Control =1,
else output = z



bufif0 - output=input
if Control =0,
else output = z

Mux with three - state output

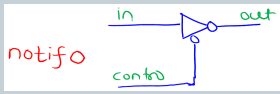
```
module mux_tri (m_out , A, B, select );
   output m_out ;
   input A, B, select ;
   tri m_out ;
   bufif1 (m_out , A, select );
   bufif0 (m_out , B, select );
endmodule
```



The outputs of three-state gates
can be connected together.

Use keyword -Tri

Notif1 – output = input'
if control = 1,
else output =z



Notif0 – output = input'
if control = 0,
else output =z

Instantiation - gate name(output, input, control);

User Defined Primitives (UDPs)

- It is declared with the keyword primitive, followed by a name and port list.
- There can be only one output, and it must be listed first in the port list and declared with keyword output.
- There can be any number of inputs. The order in which they are listed in the input declaration must conform to the order in which they are given values in the table that follows.

User Defined Primitives (UDPs)

- The truth table is enclosed within the keywords table and endtable.
- The values of the inputs are listed in order, ending with a colon (:). The output is always the last entry in a row and is followed by a semicolon (;).
- The declaration of a UDP ends with the keyword endprimitive.

```
primitive UDP_1247 (S, A, B, Ci);
   output S;
   input A, B, Ci;
   table
      0 0 0 : 0;
      0 0 1 : 1;
      0 1 0 : 1;
      0 1 1 : 0;
      1 0 0 : 1;
      1 0 1 : 0;
      1 1 0 : 0;
      1 1 1 : 1;
   endtable
endprimitive
```

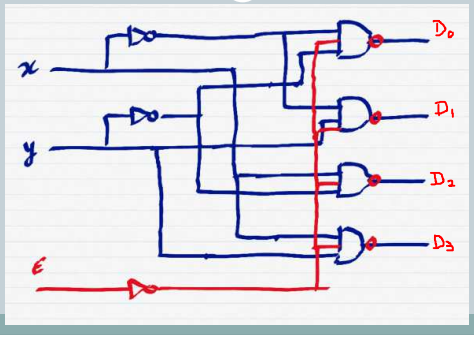
```
primitive UDP_3567 (Co , A, B, Ci);
  output Co;
  input A, B, Ci;
  table
    0 0 0 : 0;
    0 0 1 : 0;
    0 1 0 : 0;
    0 1 1 : 1;
    1 0 0 : 0;
    1 0 1 : 1;
    1 1 0 : 1;
    1 1 1 : 1;
  endtable
endprimitive
```

Full Adder using above UDPs

```
module FA_UDP (S, Co , A, B, Ci);
  output S, Co;
  input A, B, Ci;
  UDP_3567 Carry (Co , A, B, Ci);
  UDP_1247 Sum(S, A, B, Ci);
endmodule
```

Vector Statements

- Ports, wire and reg can be declared having multiple bit width.
- e.g. output [0:3] D
- wire [7:0] SUM
- Refer to individual bits or group of bits as follows:
- D[2]
- SUM[2:0]



Gate - level description of 2-to 4 line decoder

```
module decoder_2x4_gates (D, x, y, enable );
  output [0: 3] D;
  input x,y;
  input enable ;
  wire x_not ,y_not , enable_not ;
  not
    G1 (x_not , x),
    G2 (y_not , y),
    G3 ( enable_not , enable );
  nand
    G4 (D[0] , x_not , y_not , enable_not ),
    G5 (D[1] , x_not , y, enable_not ),
    G6 (D[2] , x, y_not , enable_not ),
    G7 (D[3] , x, y, enable_not );
endmodule
```

Building hierarchical description of a design

- Top-down design
- Bottom-up design

Four - bit ripple carry adder

```
module half_adder (S, C, x, y);
  output S, C;
  input x, y;
  xor (S, x, y);
  and (C, x, y);
endmodule
```

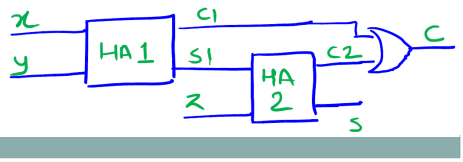
Verilog 1995 syntax

```
module half_adder ( output S, C,
input x, y);
  xor (S, x, y);
  and (C, x, y);
endmodule
```

Verilog 2001 ,2005 syntax

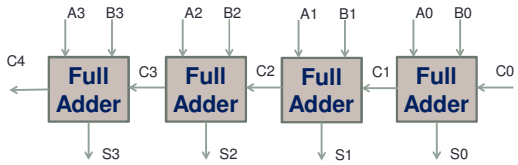
Four - bit ripple carry adder

```
module full_adder (S, C, x, y, z);
  output S, C;
  input x, y, z;
  wire S1 , C1 , C2;
  half_adder HA1 (S1 , C1 , x, y);
  half_adder HA2 (S, C2 , S1 , z);
  or G1 (C, C2 , C1);
endmodule
```



Binary adder

- Add two 4 – bit nos.
- Augend - A3A2A1A0
- Addend – B3B2B1B0



Four - bit ripple carry adder

```
module ripple_carry_4_bit_adder (Sum , C4 , A, B, C0);
  output [3: 0] Sum;
  output C4;
  input [3: 0] A, B;
  input C0;
  wire C1 , C2 , C3;
    full_adder FA0 ( Sum [0] , C1 , A[0] , B[0] , C0),
                FA1 (Sum [1] , C2 , A[1] , B[1] , C1),
                FA2 (Sum [2] , C3 , A[2] , B[2] , C2),
                FA3 (Sum [3] , C4 , A[3] , B[3] , C3);
endmodule
```

DATAFLOW MODELING

Continuous Assignment Statement with keyword **"assign"**

Dataflow modeling

Symbol	Operation	Symbol	Operation
+	binary addition		
–	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
==	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

Operators

- It should be noted that a bitwise operator (e.g., &) and its corresponding logical operator (e.g., !) may produce different results, depending on their operand.
- If the operands are scalar the results will be identical; if the operands are vectors the result will not necessarily match.
- For example, ~ (1010) is (0101), and !(1010) is 0. A binary value is considered to be logically true if it is not 0.
- In general, use the bitwise operators to describe arithmetic operations and the logical operators to describe logical operations.

- X = 4'b1010, Y = 4'b0000
- X | Y // bitwise operation. Result is 4'b1010
- X || Y // logical operation. Equivalent to 1 || 0. Result is 1.
- Examples of bitwise operators are shown below.
- X = 4'b1010, Y = 4'b1101
- ~X // Negation. Result is 4'b0101
- X & Y // Bitwise and. Result is 4'b1000
- X | Y // Bitwise or. Result is 4'b1111
- Logical operations A = 3; B = 0;
- A && B // Evaluates to 0. Equivalent to (logical-1 && logical-0)
- A || B // Evaluates to 1. Equivalent to (logical-1 || logical-0)
- !A// Evaluates to 0. Equivalent to not(logical-1)
- !B// Evaluates to 1. Equivalent to not(logical-0)

Dataflow modeling

- Identified by the keyword "assign".

```
assign a = b & c;
assign f[2] = c[0];
```

- Forms a static binding between
  - The 'net' being assigned on the LHS,
  - The expression on the RHS.
- The assignment is continuously active.
- Almost exclusively used to model combinational logic.

Dataflow modeling

- A Verilog module can contain any number of continuous assignment statements.
- For an "assign" statement,
  - The expression on RHS may contain both "register" or "net" type variables.
  - The LHS must be of "net" type.
  - A net is declared explicitly by a net keyword (such as wire) or by declaring an identifier to be an output port.

Dataflow description of 2 to 4 line decoder

```
module decoder_2x4_df ( output [0:3] D, input A, B, enable );

  assign D[0] = !((! A) && (!B) && (! enable )),
         D [1] = !(!A) && B && (! enable )),
         D [2] = !(A && (!B) && (! enable )),
         D [3] = !(A && B && (! enable ));

endmodule
```

Full Adder

```
module Full_Adder_D (S, Co , A, B, Ci);
  output S, Co;
  input A, B, Ci;
  assign S=(A&&B&&Ci) ||
  ((!A)&&(!B)&&Ci)||
  ((!A)&&B&&(!Ci))||
  (A&&(!B)&&(!Ci));
  assign Co =(A&&B)||(B&&Ci)||(A&&Ci);
endmodule
```

Dataflow description of four-bit adder

```
module binary_adder (
   output [3:0] Sum ,
   output C_out ,
   input [3: 0] A, B,
   input C_in
   );

   assign {C_out , Sum } = A+B+ C_in ;

endmodule
```

Dataflow description of a four -bit comparator

```
module mag_compare (
   output A_lt_B , A_eq_B , A_gt_B ,
   input [3: 0] A, B
   );

   assign A_lt_B = (A < B);
   assign A_gt_B = (A > B);
   assign A_eq_B = (A == B);

endmodule
```

Use of conditional operator (? :)

- condition ? true-expression : false-expression;
- assign OUT = select? A:B
  - Specifies the condition that
  - OUT = A, if select =1.
  - Else OUT = B if select = 0.

Dataflow description of 2 to 1 line multiplexer

```
module mux_2x1_df (m_out , A, B, select );
   output m_out ;
   input A, B;
   input select ;
   assign m_out = ( select )? A : B;
endmodule
```

BEHAVIORAL MODELING

Procedural assignment statements with keyword "always"

Behavioral Modeling

- Behavioral modeling represents digital circuits at a functional and algorithmic level.
- Behavioral descriptions use the keyword always, followed by an optional event control expression and a list of procedural assignment statements.
- The event control expression specifies when the statements will execute.
- The target output of a procedural assignment statement must be of the reg data type.
- A reg data type retains its value until a new value is assigned.

- The procedural assignment statements inside the always block are executed every time there is a change in any of the variables listed after the @ symbol.

Behavioral description of 2 to1 line Mux

```
module mux_2x1_beh (m_out , A, B, select );
   output m_out ;
   input A, B, select ;
   reg m_out ;

always @(A or B or select )
   if ( select == 1) m_out = A;
   else m_out = B;
endmodule
```

Target output is declared both as output and reg.

Sequential Statements in Verilog

```
1. begin
       sequential_statements
   end

2. if(expression)
   sequential_statement
   else
   sequential_statement

3. case  (expression)
       expr:  sequential_statement
       ……
       default:  sequential_statement endcase
```

Behavioral description of 4 -to -1 line Mux

```
module mux_4x1_beh (
   output reg m_out ,
   input in_0 , in_1 , in_2 , in_3 ,
   input [1: 0] select
);

always @ (in_0 , in_1 , in_2 , in_3 , select )

   case ( select )
   2' b00: m_out = in_0 ;
   2' b01: m_out = in_1 ;
   2' b10: m_out = in_2 ;
   2' b11: m_out = in_3 ;
   endcase

endmodule
```

Full Adder

```
module Full_Adder_B (S,Co , A, B, Ci);
   output S, Co;
   input A, B, Ci;
   reg S, Co;

always @(A or B or Ci)
   begin
   {Co ,S}=A+B+Ci;
   end

endmodule
```

TESTBENCH

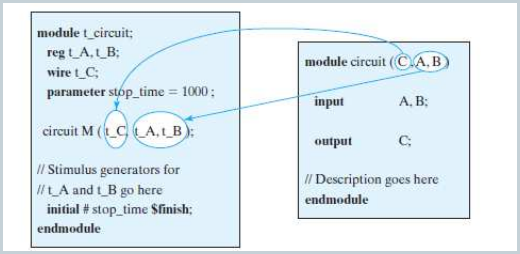
Verilog Test Bench

- What is test bench?
  - A Verilog procedural block which executes only once.
  - Used for simulation.
  - Test bench generates clock, reset, and the required test vectors

How to Write Testbench?

- Create a dummy template
  - Declare inputs to the module-under-test (MUT) as "reg", and the outputs as "wire".
  - Instantiate the MUT.
- Initialization
  - Assign some known values to the MUT inputs.
- Clock generation logic
  - Various ways to do so.
- May include several simulator directives
  - Like $display, $monitor, $dumpfile, $dumpvars, $finish.

- $display
  - Prints text or variables to stdout.
  - Syntax same as "printf".
- $monitor
  - Similar to $display, but prints the value whenever the value of some variable in the given list changes.
- $finish
  - Terminates the simulation process.
- $dumpvars
  - Starts dumping all the signals to the specified file.



- In addition to employing the always statement, test benches use the initial statement to provide a stimulus to the circuit being tested.

Example

```
initial
  begin
    A = 0; B = 0;
    #10 A = 1;
    #20 A = 0; B = 1;
  end
```

- At time 0, A and B are set to 0.
- Ten time units later, A is changed to 1.
- Twenty time units after (at t=30), A is changed to 0 and B to 1.

Example

```
initial
  begin
  D = 3' b000 ;
  repeat (7)
  #10 D = D + 3' b001 ;
  end
```

2 to 1 line mux - testbench

```
module mux_21 (m_out, A, B, select);
   output m_out;
   input A, B;
   input select;

   assign m_out = (select)? A:B;
endmodule


module mux_21_tb;
 wire mux_out_tb;
 reg A_tb, B_tb;
 reg select_tb;
 parameter stop_time = 120;
```

2 to 1 line mux - testbench

```
 mux_21 M1 (mux_out_tb, A_tb, B_tb, select_tb);

 initial #stop_time $finish;

 initial
   begin
     select_tb = 1; A_tb = 0; B_tb = 1;
     #10 A_tb = 1; B_tb = 0;
     #10 select_tb = 0;
     #10 A_tb = 0; B_tb = 1;
 end

 initial
   begin
     $monitor("time = ",$time, "select = %b A = %b B = %b Out = %b", select_tb, A_tb,
B_tb, mux_out_tb);
     end
endmodule
```