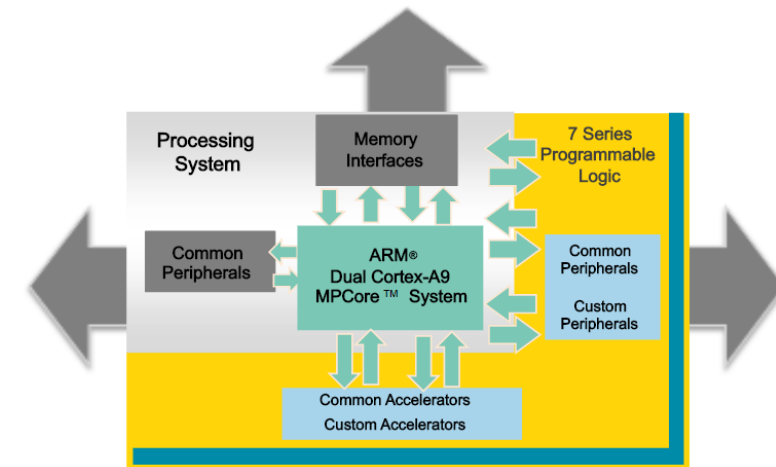
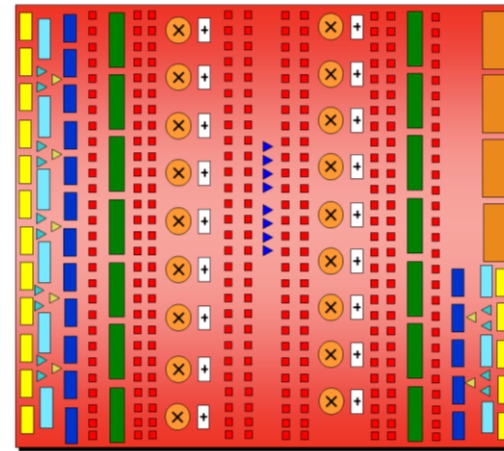




ECE 270: Embedded Logic Design



Verilog: String

- A sequence of characters enclosed in double quotes ("").
- Must be contained in a single line
- Considered as a **sequence of one-byte (8-bit) ASCII values**
- String variables should be declared as **reg** type vectors

```
reg [8*12:0] string_1;  
string_1="Hello Verilog";  
#10 string_1="Hello";  
#10 string_1="I am overloaded";
```

Accept C-like escape characters: \n = newline, \t = tab,
\b = backslash, \" = quote mark ("), %% = % sign

Verilog: String

```
reg [8*12:0] string_1;
```

```
string_1="Hello Verilog";
```

```
#10 string_1="Hello";
```

```
#10 string_1="I am overloaded";
```

```
> string_1[96:0] m overlo  
> string_1[96:0] 1011011001100101011011000110110001101111001000000101011001100101011100100
```

[illegible]

The screenshot shows a debugger window with two panes. The left pane displays the variable `string_1[96:0]`. The right pane shows a memory dump with a yellow vertical line indicating the current instruction pointer. The memory dump contains the text `m overloaded` and a long string of hex digits: `1011011010010000001101111011101100110010101110010011011000110111101100001011001000110010101100100`.

Verilog: String (Self Study)

- **reg** [8*10:1] s1, s2;

```
reg [8*14:1] s3;
```

```
reg [8*14:1] s4;
```

```
s1 = "Verilog";           // s1="000Verilog"
s2 = "-HDL";              // s2="000000-HDL"
s3= {s1,s2};              //s3= "ilog000000-HDL"
s4= {"Verilog", "-HDL"};  // s4= "000Verilog-HDL"
```

[illegible]

Verilog: String (Self Study)

```
// string variable declaration
reg [8*12:1] stringvar; // 8*12 or 96-bit wide
initial begin
    stringvar="Helloworld!";
end

// string manipulation
module string_test;
    reg [8*14:1] stringvar;
    reg [8*5:1] shortvar;
    initial begin
        stringvar = "Hello world";
        $display("%s is stored as %h", stringvar,stringvar);
        stringvar = {stringvar,"!!!"};
        $display("%s is stored as %h", stringvar,stringvar);
        shortvar = "Hello world";
        $display("%s is stored as %h", shortvar,shortvar);
    end
endmodule

// the output
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
world is stored as 776f726c64
```

Verilog: Register vs. Integer Numbers

- The **reg** register is a **1-bit** wide data type. If more than one bit is required then **range declaration** should be used (next slide).
- The **integer** register is a **32-bit** wide data type.
- Integer declarations **cannot** contain range specification.

```
module test_1(  
    input [3:0] in_1,  
    input [3:0] in_2,  
    input sel,  
    output integer [3:0] out_1  
);
```

Error: Cannot have packed dimensions of type integer

- Typically used for **constants or loop variables** in Verilog.
- Vivado will automatically trim any unused bits in integer. For e.g., if we declare an integer with a value of 255 then it will be trimmed to 8 bits.

Verilog: Real Numbers (Homework)

Verilog

Verilog: Operators

- Operators are of three types:
 - Unary
 - Binary
 - Ternary
- Unary operators precede the operand
- Binary operators appear between two operands
- Ternary operators have two separate operators that separate three operands

`a = ~ b ;` `// ~ is a unary operator. b is the operand`

`a = b && c ;` `// && is the binary operator. a and b are operands`

`a = b ? c : d ;` `// ?: is a ternary operator. b, c and d are operands`

Verilog: Operators

- Bus Operators
- Arithmetic Operators
- Bitwise Operators
- Reduction Operators
- Logical Operators
- Conditional Operators
- Relational Operators (HW)

Verilog: Bus Operators

A = 8'b10001011

Operator	Description	Example
[]	Bit/Part Select	A[0] = 1'b1; A[5:2] = 4'b0010 ;
{ }	Concatenation	{A[5:2],A[7:6],2'b01} = 8'b00101001
{ { } }	Replication	{3{A[7:6]}} = 6'b101010

Verilog: Bus Operators (Self-Study)

The concatenation operator `{ }` provides mechanism to append multiple operands. Operands *must* be sized.

Examples:

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110
```

```
Y = {B, C}; // Result Y is 4'b0010
```

```
Y = {A, B, C, D, 3'b001}; // Result Y is 11'b10010110001
```

```
Y = {A, B[0], C[1]}; // Result Y is 3'b101
```

```
assign {b [7:0], b[15:8]} = {a[15:8], a [7:0]}; // Byte swap
```

```
assign FA_out = {cout, sum}; // Full Adder output: carry out + Sum
```

Repetitive concatenation of the same number, can be expressed by using the replication constant.

```
Y = {4{A}}; // Result Y is 4'b1111
```

```
Y = {4{A}, 2{B}, C}; // Result Y is 10'b1111000010
```

Verilog: Bus Operators

A = 8'b10001011

Operator	Description	Example
[]	Bit/Part Select	A[0] = 1'b1; A[5:2] = 4'b0010 ;
{ }	Concatenation	{A[5:2],A[7:6],2'b01} = 8'b00101001
{{ }}	Replication	{3{A[7:6]}} = 6'b101010
<<	Shift left logical	A<<2 =
>>	Shift right logical	A>>3 =

Verilog: Bus Operators

A = 8'b10001011

Operator	Description	Example
[]	Bit/Part Select	A[0] = 1'b1; A[5:2] = 4'b0010 ;
{ }	Concatenation	{A[5:2],A[7:6],2'b01} = 8'b00101001
{{ }}	Replication	{3{A[7:6]}} = 6'b101010
<<	Shift left logical	A<<2 = 8'b00101100
>>	Shift right logical	A>>3 = 8'b00010001

Verilog: Bus Operators

A = 8'b10001011

- Shifting bits is a very cheap (only signal renaming) way to perform multiplication and division by powers of two
- Take the value 6 (4'b0110). If we shift it to the left one bit we get 4'b1100 or 12 and if we shift it to the right one bit we get 4'b0011 or 3.
- Now what about the value -4 (4'b1100)?
- If we shift it to the left one bit we get 4'b1000 or -8. However if we shift it to the right one bit we get 4'b0110 or 6!

Operator	Description	Example
[]	Bit/Part Select	A[0] = 1'b1; A[5:2] = 4'b0010 ;
{ }	Concatenation	{A[5:2],A[7:6],2'b01} = 8'b00101001
{ { } }	Replication	{3{A[7:6]}} = 6'b101010
<<	Shift left logical	A<<2 = 8'b00101100
>>	Shift right logical	A>>3 = 8'b00010001

Verilog: Bus Operators

A = 8'b10001011

- **Shifting bits is a very cheap** (only signal renaming) way to perform multiplication and division by powers of two
- Take the value 6 (4'b0110). If we shift it to the left one bit we get 4'b1100 or 12 and if we shift it to the right one bit we get 4'b0011 or 3.
- Now what about the value -4 (4'b1100)?
- If we shift it to the left one bit we get 4'b1000 or -8. However if we shift it to the right one bit we get 4'b0110 or 6!
- This is why we need to use the **arithmetic shift**. If we use the arithmetic shift we get 4'b1110 or -2!

Operator	Description	Example
[]	Bit/Part Select	A[0] = 1'b1; A[5:2] = 4'b0010 ;
{ }	Concatenation	{A[5:2],A[7:6],2'b01} = 8'b00101001
{ { } }	Replication	{3{A[7:6]}} = 6'b101010
<<	Shift left logical	A<<2 = 8'b00101100
>>	Shift right logical	A>>3 = 8'b00010001
>>>	Shift right arithmetic	A>>>3 = 8'b11110001

a	a >> 2	a >>> 2	a << 2	a <<< 2
0100_1111	0001_0011	0001_0011	0011_1100	0011_1100
1100_1111	0011_0011	1111_0011	0011_1100	0011_1100

Verilog: Operators

- Bus Operators
- Arithmetic Operators
- Bitwise Operators
- Logical Operators
- Reduction Operators
- Conditional Operators
- Relational Operators (HW)

Verilog: Arithmetic Operators

$A = 8'b10001011 = 139$
 $B = 8'b00001100 = 12$

Operator	Description	Example
+	Addition	$A + 12 = 151 = 8'b10010111$
-	Subtraction	$A - 10 = 129 = 8'b10000001$
*	Multiplication	$A * 3 = 417 = 9'b110100001$
/	Division	$A / 2 = 69 = 7'b1000101$
%	Modulus	$A \% 5 = 4 = 3'b100$
**	Power (exponent)	$B ** 2 = 144 = 8'b10010000$

Verilog: Arithmetic Operators

$$A = 8'b10001011 = 139$$
$$B = 8'b00001100 = 12$$

- For multiplication, many FPGAs have special resources (**DSP48**) dedicated to fast math.
- If these are available they will be used, however, if you run out of them a multiplication circuit will have to be generated (using CLB) which can be large and slow.
- **Never multiply by a power of two, but use shift instead.**
- Most tools are smart enough to do this for you but it is good practice not to rely on that.

Operator	Description	Example
+	Addition	$A + 12 = 151 = 8'b10010111$
-	Subtraction	$A - 10 = 129 = 8'b10000001$
*	Multiplication	$A * 3 = 417 = 9'b110100001$
/	Division	$A / 2 = 69 = 7'b1000101$
%	Modulus	$A \% 5 = 4 = 3'b100$
**	Power (exponent)	$B ** 2 = 144 = 8'b10010000$

Verilog: Arithmetic Operators

$$A = 8'b10001011 = 139$$

$$B = 8'b00001100 = 12$$

- When two **N-bit numbers** are added or subtracted, an **N+1-bit number** is produced.
- If you add or subtract two N-bit numbers and store the value in an N-bit number you need to think about **overflow**.
- For example, if you have a two bit number 3 (2'b11) and you add it to 2 (2'b10) the result will be 5 (3'b101) but if you store the value in a two bit number you get 1 (2'b01).
- For multiplication of two **N-bit numbers**, the result will be an **N*2-bit number**.

Operator	Description	Example
+	Addition	$A + 12 = 151 = 8'b10010111$
-	Subtraction	$A - 10 = 129 = 8'b10000001$
*	Multiplication	$A * 3 = 417 = 9'b110100001$
/	Division	$A / 2 = 69 = 7'b1000101$
%	Modulus	$A \% 5 = 4 = 3'b100$
**	Power (exponent)	$B ** 2 = 144 = 8'b10010000$

Verilog: Arithmetic Operators (Self-Study)

```
//suppose that: a = 4'b0011;  
//              b = 4'b0100;  
//              d = 6; e = 4; f = 2;  
//then,  
a + b //add a and b; evaluates to 4'b0111  
b - a //subtract a from b; evaluates to 4'b0001  
a * b //multiply a and b; evaluates to 4'b1100  
d / e //divide d by e, evaluates to 4'b0001. Truncates fractional part  
e ** f //raises e to the power f, evaluates to 4'b1111  
      //power operator is most likely not synthesizable
```

If any operand bit has a value "x", the result of the expression is all "x".
If an operand is not fully known the result cannot be either.

Verilog: Bitwise Operators

$A = 8'b10001011$

- Operate **on each bit** individually.
- When you perform a bitwise operator on multi-bit values, you are essentially using **multiple gates** to perform the bitwise operation.
- If the two values used by a bitwise operator are different in length, the shorter one is filled with zeros to make the lengths match.

Operator	Description	Example
\sim	Inverse / NOT	$\sim A = 8'b01110100$
$\&$	AND	$A[2] \& A[1] = 1'b0$
$ $	OR	$A[2] A[1] = 1'b1$
\wedge	XOR	$A[2] \wedge A[1] = 1'b1$
$\sim \wedge$	XNOR	$A[2] \sim \wedge A[1] = 1'b0$

Verilog: Logical Operators

A = 8'b10001011

Operator	Description	Example
!	NOT	!A[1] = FALSE, !A[2] = TRUE
&&	AND	A[0] && A[1] = TRUE
	OR	A[0] A[2] = TRUE
==	EQUAL	A[3:0] == 4'b1011 = TRUE
!=	NOT EQUAL	A[3:0] != 4'b1011 = FALSE
<,<=,>,>=	COMPARE	A[3:0] < 13 = TRUE

Verilog: Logical and Bitwise Operators

a	b	a&b	a b	a&&b	a b
0	1	0	1	0 (false)	1 (true)
000	000	000	000	0 (false)	0 (false)
000	001	000	001	0 (false)	1 (true)
011	001	001	011	1 (true)	1 (true)

Verilog: Reduction Operators

$A = 8'b10001011$

- **Reduce** the number of bits to one by performing the specified function on every bit.
- Similar to the bitwise operators, except they are performed on all the bits of a single value

Operator	Description	Example
$\&$	AND	$\&A = A[0] \& A[1] \& \dots A[7] = 1'b0$
$\sim\&$	NAND	$\sim\&A = \sim(A[0] \& A[1] \& \dots A[7]) = 1'b1$
$ $	OR	$ A = A[0] A[1] \dots A[7] = 1'b1$
$\sim $	NOR	$\sim A = \sim(A[0] A[1] \dots A[7]) = 1'b0$
\wedge	XOR	$\wedge A = A[0] \wedge A[1] \wedge \dots A[7] = 1'b0$
$\sim\wedge$	XNOR	$\sim\wedge A = \sim(A[0] \wedge A[1] \wedge \dots A[7]) = 1'b1$

Verilog: Operators (Self-Study)

a b	a==b	a===b	a!=b	a!==b	a&b	a&&b	a b	a b	a^b
0 0	1	1	0	0	0	0	0	0	0
0 1	0	0	1	1	0	0	1	1	1
0 x	x	0	x	1	0	0	x	x	x
0 z	x	0	x	1	0	0	x	x	x
1 0	0	0	1	1	0	0	1	1	1
1 1	1	1	0	0	1	1	1	1	0
1 x	x	0	x	1	x	x	1	1	x
1 z	x	0	x	1	x	x	1	1	x
x 0	x	0	x	1	0	0	x	x	x
x 1	x	0	x	1	x	x	1	1	x
x x	x	1	x	0	x	x	x	x	x
x z	x	0	x	1	x	x	x	x	x
z 0	x	0	x	1	0	0	x	x	x
z 1	x	0	x	1	x	x	1	1	x
z x	x	0	x	1	x	x	x	x	x
z z	x	1	x	0	x	x	x	x	x

Verilog: Operators

- Bus Operators
- Arithmetic Operators
- Bitwise Operators
- Logical Operators
- Reduction Operators
- Conditional Operators
- Relational Operators (HW)

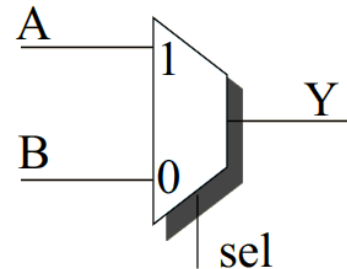
Verilog: Conditional Operator

- Can be used in place of *if* statement when one of the two or more values is to be selected for assignment

condition ? value_if_true : value_if_false

- Can be part of procedural or continuous assignment

$Y = (\text{sel})? A : B ;$



Verilog: Conditional Operator

- Design maximal circuit to return the maximum of a, b and c.

```
assign max = (a>b) ? ((a>c) ? a : c) :  
                ((b>c) ? b : c);
```

- Design 4:1 Multiplexer using conditional operator
- Design 1-bit equality comparator using conditional operator

Verilog: Operator Precedence

- If no parentheses are used to separate operands, then Verilog uses the following rules of precedence (**Good practice: use parentheses**)

Operators	Operators Symbols	
Unary	+ - ! ~	Highest Precedence
Multiply, Divide, Modulus	* / %	
Add, Subtract	+ -	
Shift	<< >> >>>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	&, ~&, ^, ^~, , ~	
Logical	&&	
Conditional	?:	Lowest Precedence

Verilog: Hint for \$signed Home Work

When you write this in Verilog:

```
wire [7:0] a;  
wire [7:0] b;  
wire less;  
assign less = (a < b);
```

the comparison between a and b is unsigned, that is a and b are numbers in the range 0-255. Writing this instead:

```
wire [7:0] a;  
wire [7:0] b;  
wire less;  
assign less = ($signed(a) < $signed(b));
```

means that the comparison treats a and b as *signed* 8-bit numbers, which have a range of -128 to +127. Another way of writing the same thing is:

```
wire signed [7:0] a;  
wire signed [7:0] b;  
wire less;  
assign less = (a < b);
```

Verilog (Three Concepts)

- Difference between Register and Wire (Next two lectures)
- **Efficient Behavioral modelling**
- Difference between blocking and non-blocking assignments

Behavioral Modeling

Behavioral Modeling

- There are two basic statements in behavioral modeling: *initial* and *always*
- All other statements appear inside these statements.
- All *initial* and *always* blocks run in parallel
- All of them start at simulation time 0.
- *Initial* block starts at time 0 and executes only once. The *initial* statement provides a means of initiating input waveforms and initializing simulation variables before the actual description/simulation begins.
- Once the statements in the *initial* are exhausted, statement becomes inactive.

Behavioral Modeling

```
initial
```

```
begin /* multiple statements, need to be grouped –  
        begin/end */
```

```
    clock = 1'b0 ; // clock initial logic state
```

```
    nrst  = 1'b0  // reset initial logic state
```

```
end
```

```
initial
```

```
begin
```

```
    # 5 a = 1'b1 ; // set a to 1 @ simulation time 5
```

```
    # 25 b = 1'b1 ; // set b to 1 @ simulation time 30
```

```
    # 50 $finish ; // end simulation after 50 time ticks
```

```
end
```

Behavioral Modeling

```
module Reg_File
    .....
    reg [31:0] RegFile[0:31] ;
    integer i ;
    initial
        begin
            for(i = 0 ; i < 32 ; i = i + 1)
                RegFile[i] = 32'h0 ;
            end
        .....
    endmodule
```

Behavioral Modeling

- *always* block starts at time 0 and executes statements continuously in a loop.
- Describes the function of a circuit.
- Can contain many statements like *if*, *for*, *while*, *case*
- Statements the *always* block are executed *sequentially* (= assignment) or in *parallel* (<= assignment)
- The *final* result describes the function of the circuit for current set of inputs.

```
// clock declaration, used mainly in Test Benches
```

```
always
```

```
# 10 clock = ~clock ; // Toggle clock every half-cycle
```


Behavioral Modeling

`always` block

- Always waiting for a change to a trigger signal
- Then executes the body

```
module and_gate (out, in1, in2) ;  
  input  in1, in2 ;  
  output reg out ;
```

```
  always @(in1 or in2)  
  begin  
    out = in1 & in2 ;  
  end  
endmodule
```



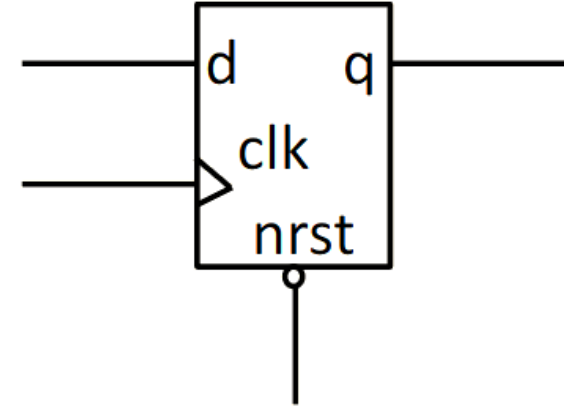
Specifies when block is executed
I.e., triggered by which signals

Behavioral Modeling

```
module full_adder (sum, cout, a, b, cin) ;  
    input a, b, cin ;  
    output reg sum, cout ; // implicit register  
  
    always @(a or b or cin) // Verilog 2001 allows (a, b, cin)  
        {cout, sum} = a + b + cin ;  
endmodule  
  
// If sensitivity list is too long, use (*), i.e. all inputs
```

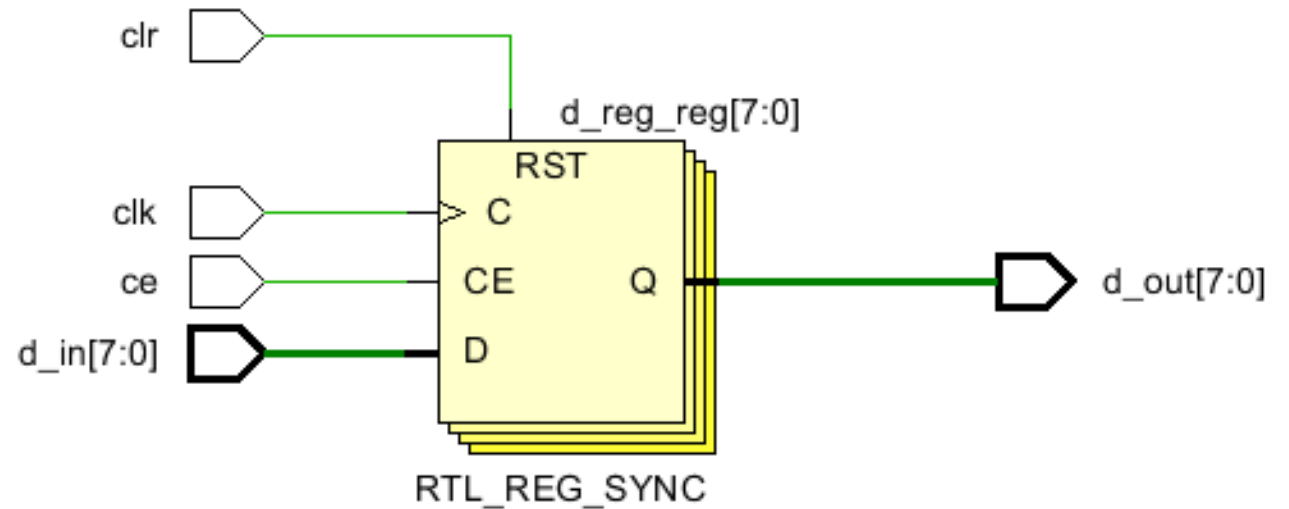
Verilog: Module (Examples)

```
module D_FF(clk, nrst, d, q) ;  
    input clk, nrst, d ;  
    output reg q ;  
    always @(posedge clk or negedge nrst)  
        // Event-based Timing Control  
        // reset state  
        if (!nrst)  
            q <= 0 ;  
        else  
            // normal operation  
            q <= d ;  
endmodule
```



Verilog: Register

```
module test_1(  
    input [7:0] d_in,  
    input ce,  
    input clk,  
    input clr,  
    output [7:0] d_out  
);  
  
    reg [7:0] d_reg;  
    always@(posedge clk)  
    begin  
        if(clr)  
            d_reg <= 8'b00000000;  
        else if (ce)  
            d_reg <= d_in;  
    end  
    assign d_out = d_reg;  
endmodule
```



Behavioral Modeling: Guidelines

- While writing Verilog code for synthesis, we need to be aware of how the various language constructs are mapped to hardware.
- **Common errors:**
 1. Variable assigned in **multiple always blocks**
 2. **Incomplete sensitivity** list
 3. **Incomplete branch** and **incomplete output** assignments