Verilog Syntax Guide – EE 214

Important Reminder: Verilog is **not** a *programming language*, like C, Java, or Python. It does not produce a program that is executed by a processor – it is a *hardware description language*, which is used to define a digital circuit (like the ones you are learning about in lecture). For this class, Vivado uses your Verilog code to generate a bitstream, which then is used to implement your circuit on the FPGA.

Top Level Module & Busses

```
module top
(
    // Port list
    input  [7:0] sw,
    output [7:0] led
);

// logic

endmodule
```

The top-level module is the one in **bold** in the design sources panel. You can set a new Top in the right-click menu for a module in the sources panel. The input and output ports of the top level module *must* match the symbols defined in the .xdc constraints file. In the master XDC for the Blackboard, the ports are defined with square brackets [] so that you can group the signals together into a **bus**. If you only want to use a single port without changing the constraints file, you can declare a 1 bit wide bus.

```
input [11:0] sw, // [MSB:LSB] - [11:0] is 12 bits wide
output [3:3] led, // 1 bit wide bus
...
assign led[3] = sw[3]; // Use in statements
```

Data Types

The two most common data types in Verilog are **wire** and **reg**. Wires do not store any data; they can be thought of as a physical wire – they connect elements together. When you declare an input or output signal in a module's port list, the default type is **wire**. So the following port lists are equivalent:

```
1.  module example1
2.  (
3.      input A, // Single wire
4.      input [3:0] B, // Bus (or group) of wires
5.      output Y
6.  );
7.  endmodule
8.
9.  module example2 // Equivalent to example1
10. (
11.     input  wire A,
12.     input  wire [3:0] B,
13.     output wire Y
14. );
15. endmodule
```

Since wires do not store a value, they cannot be assigned to in more than one place. They are used to connect elements together, so in the following code, X will *always* have the same value as A. Regs on the other hand store their previous value, so in the following code, X_reg is only initialized to A. Then, when the input w_en is set, its value changes to d_in.

```
1.  module example
2.  (
3.      input wire w_en,       // write enable
4.      input wire [3:0] d_in, // data in
5.      input wire [3:0] A
6.  );
7.
8.  wire [3:0] X;
9.  assign X = A;
10. // or
11. wire [3:0] X = A; // Can combine lines 7-8 into one statement
12.
13. reg [3:0] X_reg = A; // This initializes X_reg to A
14.
15.
16. // This line will not work, since X would be driven twice
17. // assign X = 4’b1111; // Literal number, width=4, base=binary
18.
19. always @( w_en )
20. begin                 // “begin” and “end” group statements
21.     if (w_en == 1’b1) // together, like “{“ and “}” in C
22.         X_reg = d_in; // begin and end can be omitted when the
23. end                   // code block is a single statement
24.
25. // begin and end can be omitted for always blocks too
26. always @( w_en )
27.     if (w_en == 1’b1)
28.         X_reg <= d_in;
29.
30. endmodule
```

Procedural Blocks (always/initial)

Regs can only be assigned to in a procedural block. In a procedural block, the statements execute in sequence, unlike when using the “assign” statement with a wire. Wires cannot be assigned to in a procedural block.

Procedural blocks can contain control structures such as if, if else, else, and case statements. These work similar to programming languages such as C.

Initial blocks execute once at the start of simulation, and are primarily used in test benches. **Always** blocks execute whenever a signal in their **sensitivity list** changes. When the signal in the sensitivity list is a bus, every bit in the bus is included in the sensitivity list. This module stores a 16-bit number which starts at 0, and counts up at the same frequency as the clock.

```
module example
(
    input wire clk, count_en,
    output wire [15:0] val
);

reg [15:0] count; // 16-bit reg

initial // Initializes the count reg
     count <= 0; // Single statement

// Sensitivity lists generally include a clock signal
always @( posedge(clk) ) // The posedge() function is used to start
begin                    // the block on each rising edge of clk
    count <= count + 1;
end

assign val = count; // The wire must be assigned outside of the
                    // procedural blocks
endmodule
```

The next module accomplishes the same task. It uses a reset signal to initialize count, which is a more common design pattern. It also uses the count reg as the output signal, so the assign statement isn’t needed. Note that regs cannot be used as input ports.

```
module example
(
    input wire clk, rst, count_en,
    output reg [15:0] count
);

always @( posedge(clk), rst )
begin
    if (rst == 1’b1)
        count <= 16’d0;
    else
        count <= count + 1;
end
endmodule
```

Blocking & Non-Blocking Assignment

In a procedural block, you can use blocking assignment with “=”, and non-blocking assignment with “<=”. Blocking assignment works in a similar way to programming languages like C, it executes sequentially and “blocks” other operations from happening until the assignment is finished. Non-blocking on the other hand, schedules the assignment to update the destination at the end of the procedural block. So for non-blocking assignment, the order of statements doesn’t matter.

```
reg [3:0] A, B;

initial
begin
    A = 4’d3;
    B = 4’d6;
end

// Blocking
always @( posedge(clk) )
begin
    A = 4’d2;
    B = A;
    // A and B are both 2
end

// Non-blocking
always @( posedge(clk) )
begin
    A <= 4’d2;
    B <= A;
    // A is 2, B is still 6
end
```

Combinational Logic in an Always Block

An always block can model combinational logic using “always @(*)”. The *, known as the “wildcard”, means that the sensitivity list includes all of the signals used in the block. This has the same effect as combinational logic, so the reg variables act as wires. You should use blocking assignment (=) in a combinational always block.

```
module comb_always
(
    input wire A, B
    output reg Y
)

always @(*) // Equivalent to @(A, B, C)
begin
    if (A & B)
        Y = 1’b1;
    else if (A & ~B)
```

```
        Y = 1'b0;
    else if (~A & B)
        Y = 1'b1;
    else if (~A & ~B)
        Y = 1'b0;
end
endmodule
```

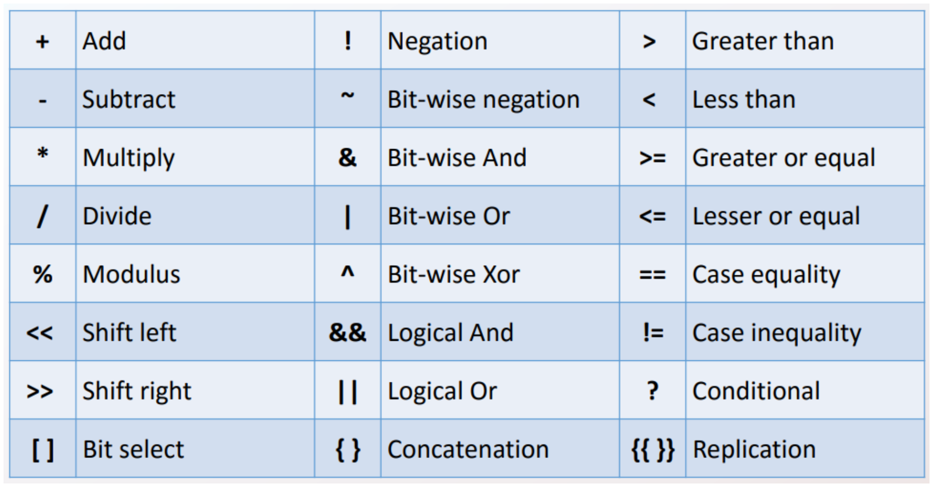
```
module mux_4_1
(
    input wire [1:0] sel,
    input wire [7:0] in0,
    input wire [7:0] in1,
    input wire [7:0] in2,
    input wire [7:0] in3,
    output reg [7:0] out
)

always @(*) // Equivalent to @(sel, in0, in1, in2, in3)
begin
    case (sel) // The case matching the value of sel
        2'b00: // is executed
        begin
            out = in0; // If sel = 2'b00, then out = in0
        end
        2'b01 : out = in1;
        2'b10 : out = in2;
        2'b11 : out = in3;
        default : out = 8'd0; // Executes if sel does not
    endcase                   // match any of the cases
end

endmodule
```

Number Literals: <width>'<base>x

- 4'b1111: binary 1111
- 4'd15: decimal 15
- 4'b1111 == 4'd15

- 5'b10010: binary for decimal 18
- 4'd18: decimal 18 (too large to fit in 4 bits!)
- 5'd18 decimal 18 (OK)

Operators



+	Add	!	Negation	>	Greater than
-	Subtract	~	Bit-wise negation	<	Less than
*	Multiply	&	Bit-wise And	>=	Greater or equal
/	Divide	|	Bit-wise Or	<=	Lesser or equal
%	Modulus	^	Bit-wise Xor	==	Case equality
<<	Shift left	&&	Logical And	!=	Case inequality
>>	Shift right	||	Logical Or	?	Conditional
[ ]	Bit select	{ }	Concatenation	{{ }}	Replication

```
1.   ///// Bit select [] /////
2.
3.   assign led[3] = sw[3];
4.
5.   wire [7:0] X, Y; // Declares 2 wires, each 8 bits wide
6.
7.   // These lines are equivalent:
8.   assign X = Y;
9.   assign X[7:0] = Y[7:0];
10.
11.  // Assigns the upper 4 bits of Y from the lower 4 bits of X
12.  assign Y[7:4] = X[3:0];
13.
14.
15.  wire A, B, C;
16.  wire [2:0] F, Z; // 3 bits wide
17.
18.  ///// Concatenation {} /////
19.
20.  assign F = { A, B, C }; // This is equivalent to lines 24-26
21.  assign F[2] = A;
22.  assign F[1] = B;
23.  assign F[0] = C;
24.
25.
26.  ///// Replication {{}} /////
27.
28.  // These lines are equivalent:
29.  assign Z = {3{ A }};
30.  assign Z = { A, A, A };
31.
32.
33.  ///// Shift << >> /////
34.
35.  result = [signal] <</>> [n] // Shifts the signal left or right by n
36.                              // bits. Fills empty bits with 0.
37.  wire [7:0] X = 8’b10011101;
```

```
wire [7:0] Y = X << 3; // Shifts X 3 bits left
wire [7:0] Z = X >> 2; // Shifts X 2 bits right
// Y = 11101000
// Z = 00100111


///// Logical vs. bit-wise operators /////

// Logical: &&, ||, !
wire en, rst;
always @( posedge(clk) )
    if (en == 1'b1 && rst == 1'b0) // Used for logical expressions
        // Executes if en is 1 and rst is 0


// Bit-wise: &, |, ~
wire [7:0] X = 8'b11011010;
wire [7:0] Y, Z;
assign Y = ~X;          // Inverts X
assign Z = X & 8'b1111; // ANDs X with 00001111
// Y = 00100101
// Z = 00001010
```

Modules

Modules are used to group logic statements together to improve organization, readability, and reusability. A module is generally defined in its own .v file, with the same name as the file name. The first part of a module is its **port list**. After the port list, the module's logic is defined by internal signal declarations, assign statements, always blocks, and the instantiation of other modules. To instantiate a module, you must connect its input and output ports to external signals with a **port connection list**.

```
// and_gate.v
module and_gate
(
    input A, B,
    output Y
);
assign Y = A & B;
endmodule
```

```
// top.v
module top
(
    input  [3:0] sw,
    input  [3:0] btn,
    output [3:0] led
);



```

```
and_gate and0 // <module name> | <instance name> (optional)
(
    .A ( sw[0]  ), // .<port name> ( <external signal> ),
    .B ( btn[0] ),
    .Y ( led[0] )
);

// Whitespace doesn’t matter in Verilog
and_gate and1 ( .A(sw[1]), .B(btn[1]), .Y(led[1]) );
and_gate and2 ( .A(sw[2]), .B(btn[2]), .Y(led[2]) );
and_gate and3 ( .A(sw[3]), .B(btn[3]), .Y(led[3]) );

endmodule
```

Test Benches

Test benches are used to specify inputs to a CUT (circuit under test) for a simulation. For more information on simulation, refer the tutorial for Project 3 Requirement 2. The CUT is the module you are trying to test. The test bench is a module without any inputs or outputs, that contains an instance of the CUT. A test bench contains 4 basic components:

- Input signals (reg type)
- Output signals (wire type)
- CUT instantiation (connects the inputs & outputs to the CUT)
- Input stimulus (defines which inputs should be triggered and when in order to properly verify the CUT’s functionality)

```
// and_gate_3_input.v

module and_gate_3_input
(
    input  wire A, B, C,
    output wire Y
);

assign Y = A & B & C;

endmodule
```

To test the and_gate_3_input module, you could use the following test bench:

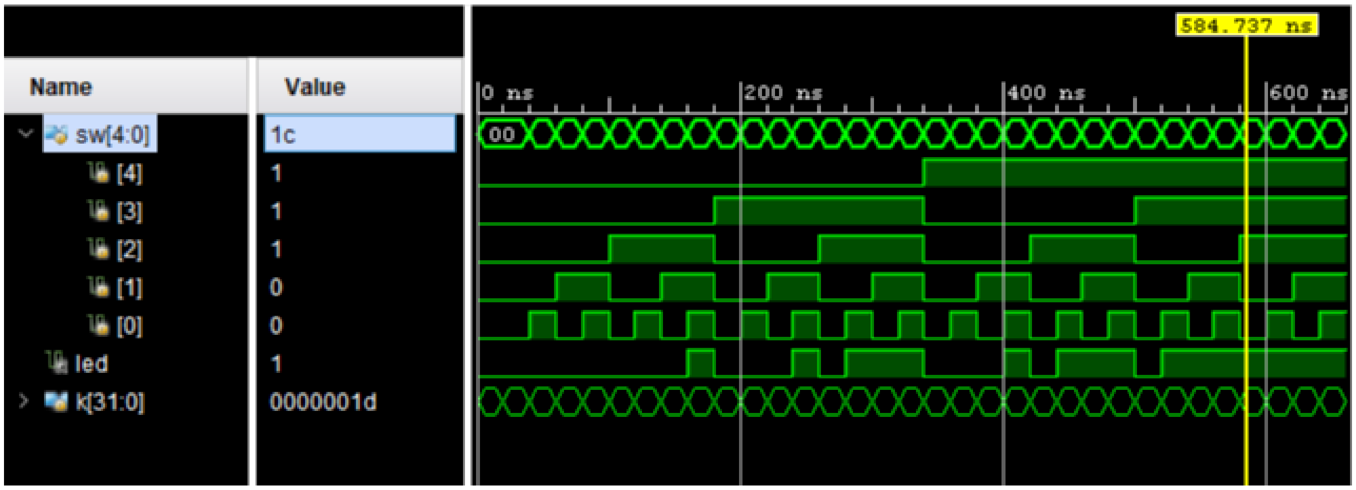
```
// and_gate_3_input_tb.v

`timescale 1ns / 1ps // Required for simulation

module and_gate_3_input_tb(); // Empty port list

// Inputs (initialized to 0)
reg A = 0;
```

```
9.   reg B = 0;
10.  reg C = 0;
11.  // Output
12.  wire Y;
13.
14.  // CUT instantiation
15.  and_gate_3_input cut
16.  (
17.      .A ( A ),
18.      .B ( B ),
19.      .C ( C ),
20.      .Y ( Y )
21.  );
22.
23.  // Input stimulus
24.  initial
25.  begin
26.      #20 // Delay for 20 ns (units defined by `timescale)
27.          A = 1;
28.      #20 A = 0; // Waits 20 ns, then sets A back to 0
29.
30.      #20
31.      A = 0; B = 1; C = 0;
32.      #20
33.      A = 0; B = 0; C = 1;
34.
35.      #20
36.      A = 1; B = 1; C = 0;
37.      #20
38.      A = 1; B = 0; C = 1;
39.      #20
40.      A = 0; B = 1; C = 1;
41.      #20
42.      A = 1; B = 1; C = 1;
43.
44.      #50 $finish // Stops the simulation
45.  end
46.
47.  endmodule
```

When the test bench is run by a simulator, you can see the value of each signal over time. Since the output Y is the result of ANDing A, B, and C, you should be able to verify that Y is not asserted until A, B, and C are all set to 1. This happens after line 38. The total delay in the initial block up to that point is 160 ns, so the simulation should show that at 160 ns, Y transitions from 0 to 1.

```
1.  // Example from Project 3 Requirement 2 tutorial
2.
3.  `timescale 1ns/ 1ps
4.
5.  module majority_of_five_tb;
6.
7.  // Inputs
8.  reg [4:0] sw; // Width of inputs and outputs must match the
9.                // width of ports on the CUT
10. // Outputs
11. wire led;
12.
13. // Instantiate the Circuit Under Test (CUT)
14. majority_of_five cut (.sw(sw),.led(led));
15.
16.  // Declare loop index variable
17. integer k;
18.
19. // Apply input stimulus
20. initial
21. begin
22.     sw = 0; // The input is initalized in the initial block,
23.             // instead of the reg declaration
24.     for (k=0; k<32; k=k+1)
25.         #20 sw = k;
26.
27.     #20 $finish;
28. end
29. endmodule
```

In the majority_of_five testbench example, the module called “majority_of_five” is being tested. To do this, the “majority_of_five_tb” module instantiates majority_of_five at line 13. The loop variable *k* is used to apply every possible combination of inputs. The CUT has a single input bus (sw) with a width of 5, so k starts at 5’b00000 (dec. 0) and goes up to 5’b11111 (dec. 31). There is a delay of 20 ns after each increment of k. When run in the simulator, you can observe the following waveform. Each combination of the inputs lasts for 20 ns, and you can see that the output (led) is asserted when at least 3 out of the 5 bits on the input are asserted.



State Machines

State machines are used in many different applications. You can find more information on them on Real Digital. The main parts of a state machine are the state registers, next state logic, and output logic. The following is an example of a Verilog implementation for a simple finite state machine:

```verilog
module FSM
(
    input clk, rst,
    input A, B,
    output Y
);

// State codes
localparam S0 = 2’b00; // localparam defines a constant
localparam S1 = 2’b01; // This allows you to name states for your
localparam S2 = 2’b10;    particular use
localparam S3 = 2’b11;

// State registers
reg [1:0] PS, NS;

always @( posedge(clk) )
begin
    if (rst)
        PS <= S0;
    else
        PS <= NS;
end

// Next state logic
always @(*)
begin
    case (PS)
        S0:
        begin
            NS = S1;
        end
        S1:
        begin
            if (A & ~B)
                NS = S2;
            else
                NS = S1;
        end
        S2:
        begin
            if (~A & B)
                NS = S3;
            else
                NS = S2;
        end


        S3:
        begin
            if (~A & ~B)
                NS = S0;
            else
                NS = S3;
        end
        default:
            NS = S0;
    endcase
end

// Output logic
assign Y = (PS == S1) || (PS == S3); // The output Y is asserted in
                                        states 1 and 3
endmodule
```