Verilog 101

DATA TYPES, MODULE & CONTROL STRUCTURE, OPERATORS, AND CONSTRAINTS

WHAT IS VERILOG?

Verilog is a hardware Description Language (HDL) that consists of digital logic. It is intended to be used for simulations, timing analysis, testing, and synthesis

Data Types: WIRE

- Wire represents a physical wire in a circuit. It can be used as "assign" and to make connections between modules.

- Input, output, and inout are types of wires that enter and exit the module. "Input wire" and "output wire" are the same as "input" and "output," respectively.

- Output's can also be reg's.

```
module test(
    input [1:0] data,
    output Y
);

wire myWire;
assign myWire = data[0] & data[1];
assign Y = myWire;

endmodule
```

Data Types: REG

- reg is a register that can be assigned (flip-flop). It is used in an "always" block

```
module flipflop(
    input [1:0] data,
    input clk,
    output reg Y0,
    output reg Y1
);

always @(posedge clk) begin
    Y0 <= data[0];
    Y1 <= ~data[1];
end

endmodule
```

Data Types: VECTOR

- *Vectors are multi-bit buses of wires or reg's*

```
reg[3:0] myRegs;
wire[11:0] myBus;
```

// bus of 4 reg's
// bus of 12 wires

- *Integer is a 32 bit signed variable*

```
integer div;
```

// bus of 32 reg's

- *All data types are bit vectors that can have 4 values: 0, 1, X(don't care), Z(high impedance)*

- *Vectors can be represented as binary, hex, decimal*

```
wire [7:0] data0, data1, data2;  // buses of 8 wires each
assign data0 = 8'b101010; // binary
assign data1 = 8'hF2; // hex
assign data2 = 8'd255; // decimal
```

Operators: LOGICAL, BITWISE

Logical:

- *! (NOT)*
- *&& (AND)*
- *|| (OR)*
- *== (Equals)*

Bitwise:

- *~ (NOT)*
- *& (AND)*
- *| (OR)*
- *^ (XOR)*
- *~^ (XNOR)*

Operators: ASSIGNMENT

There are 3 types of assignment operators: "assign", "non-blocking", and "blocking"

- Blocking vs Non-blocking

Sum <= A+B; (Non-blocking)

Sum = A+B; (Blocking)

Nonblocking assignments happen in parallel, blocking assignments wait for previous assignment to finish before starting

Operators: CONCATENATION

- *{ } can be used to concatenate wires*
- *shifter*

```
module top(
input I0, I1, I2, I3,
input [5:0] data_in,
output [9:0] concatenated
);

assign concatenated = {data_in, I0, I1, I2, I3};

endmodule
```

Control Structure: COMBINATIONAL BLOCKS

- always block executes always. Triggered by an event in the sensitivity list.

```
always@(SW or data or sel) begin
        ..... elements ....
end
```

*- * can be used to trigger whenever any signal within the block changes*

```
always@(*) begin
        ..... elements ....
end
```

Can only assign regs and integers (wires must be assigned outside of always blocks)

Control Structure: SEQUENTIAL BLOCKS

- *always@(posedge(clk))*

- *always@(negedge(clk))*

- *always@(posedge(clk) or posedge(rst))*
  **asynchronous reset*

Can only assign regs and integers (wires must be assigned outside of always blocks)

Control Structure: IF, ELSE, CASE

- If, else if, else, and case statements need to be in an "always" block

```
module mux (
    input I0,
    input I1,
    input I2,
    input I3,
    input [1:0] Sel,
    output reg Y0,
    output reg Y1
);
```

```
always @(Sel,I0,I1,I2,I3)
begin
    if (Sel == 2'd0)
        begin
        Y0 = I0;
        Y1 = 1'b1;
        end
    else if (Sel == 2'd1)
        begin
        Y0 = I2;
        Y1 = 1'b0;
        end
    else if (Sel == 2'd2)
        begin
        Y1 = I3;
        Y1 = 1'b1;
        end
    else
        begin
        Y1 = 1'b0;
        Y0 = I3;
        end

end
```

In case the if statement or case statement has more than one assignment (in this case Y0 = I0; and Y1 = 1'b1;) then "begin" and "end" are needed; otherwise, both of these assignments won't be executed

Module Structure: DECLARATION

- Below are two different ways of declaring modules in verilog, both of these declarations have the same result

```
module main(
    input I0,
    input I1,
    input SW,
    output Y
);

// Structural Description of the module
endmodule
```

```
module main(
    I0, I1, SW, Y
);
input I0, I1, SW;
output Y;

// Structural Description of the module
endmodule
```

Module Structure: CALLING

- Below are two different ways of calling modules in verilog, both of these callings have the same result. (module "main" can be found on the previous slide)

```
module wrapper(
    input [1:0] data_in,
    input SW,
    output Y
);

main mywrapper(
    .I0(data_in[0]),
    .I1(data_in[1]),
    .SW(SW),
    .Y(Y)
);

endmodule
```

For this method, the order of wire connections made within the called module doesn't matter.

```
module wrapper(
    input [1:0] data_in,
    input SW,
    output Y
);

main mywrapper(data_in[0], data_in[1], SW, Y);

endmodule
```

To call modules using this method, the wires listed have to be exactly in the same order as the wires/reg's within the module that's being called or wrapped together. In this case wrapper module wires are in the same order as the wires in the "main" module within the called module.

INITIALIZING REGISTERS

```
module controller(
    input start,
    input stop,
    input inc,
    input rst,
    output reg Cout = 0,
    input clk
);
```

```
reg [1:0] state = 0;
```

STATE MACHINES

```
`timescale 1ns / 1ps
module controller(
    input start,
    input stop,
    input inc,
    input rst,
    output reg Cout = 0,
    input clk
);

// Define State Codes
`define idle 0
`define start 1
`define increment_1 2
`define increment_2 3

reg [1:0] state = 0;

//stopwatch state machine
```

STATE MACHINES

```
always @ (posedge clk) begin
if(rst) begin
    Cout <= 0;
    state <= `idle;
end else begin
    case (state)
        `idle:begin
            Cout<=0;
            if (start == 1'b1) begin
                state <= `start;
                Cout<=1;
            end
            if (start == 1'b0 && inc == 1'b1) begin
                state <= `increment_1;
                Cout<=1; //output value
            end
        end
```

```
        `start: begin
        Cout<=1;
            if (stop == 1'b1) begin
                state <= `idle;
                Cout<=0; //output value
            end
        end
        `increment_1: begin
            Cout<=0;
            state <= `increment_2;
        end
        `increment_2:begin
            Cout<=0;
            if (inc == 1'b0)
                state <= `idle;
        end
        default:
            state <= `idle;
    endcase end
end
endmodule
```

WHAT ARE CONSTRAINTS?

Constraints are used to map physical pins on your prototyping board (i.e. Basys 3, ZYBO, Nexys 4DDR) to logical pins in your Verilog design.

Each board has a different way of how the pins are named and where they are mapped on the circuit board.

Constraints: EXAMPLE

```
set_property PACKAGE_PIN U16 [get_ports {led0}]
set_property IOSTANDARD LVCMOS33 [get_ports {led0}]
```

Basys 3 led0 is connected to the FPGA on pin U16 (the pin name can be seen on the Basys 3 board or it can be found in the Basys 3 datasheet)