EECS151/251A Discussion 2

Christopher Yarp

Feb. 1, 2019
Verilog
Hardware Description Language

• The lecture, the FPGA lab, and the ASIC lab all involve describing hardware designs!

• How do we describe a hardware design?
  • Hardware Description Languages (HDL)

• Many HDLs Exist:
  • Verilog
  • VHDL
  • SystemVerilog
  • SystemC
  • Chisel
  • …
HDLs are not Like C

• People commonly say that Verilog has a C like syntax
  • Operators (+, -, &&, ||, ...) are generally the same
  • Both use semicolons at the ends of statements
  • Single line comments start with //, block comments are enclosed in /* ... */
• That is pretty much where the similarities end
• HDLs are languages for describing a hardware design
  • Combinational logic will be running in parallel ... at all times!
  • If any input to combinational logic changes, it will immediately begin producing the new result (output will come after some delay)
  • Multiple state elements can change state simultaneously at a clock edge
Using HDLs

• Many HDLs originated as simulation languages
• Not all Verilog is “synthesizable” (can be interpreted by the FPGA or ASIC tools to represent a design)
• You will be working with a subset of the Verilog language
• Look over the “Verilog Primer” Slides on the Website
  • Under “Resources”
The always @ block

• always @ blocks have the following syntax:
  
  ```verilog
  always @(sensitivity list) begin ... end
  ```

• At a high level, you are telling Verilog the statements contained in the always @ block should only change when a signal in the sensitivity list has changed
The always @ block – Combinational Logic

• For combinational logic, you should list any referenced signal
  • a signals that appears on the right hand side of assignment statements
  • a signal used in a conditional statement
• Supplying an incomplete sensitivity list can result in unexpected behavior
• Verilog allows you to specify always @(*)
  • With this, Verilog will determine the proper combinational logic sensitivity list for you!
  • **Use this** whenever you want to use an always block to describe combinational logic.

```verilog
reg out;
wire, a, b, c;
always @(a, b, c) begin
  if(a) begin
    out = b;
  end else begin
    out = c;
  end
end
```
The always @ block – Sequential Logic

• The sensitivity list allows us to describe sequential logic (registers)
  • Use posedge or negedge to describe an edge triggered flip-flop (register)

```vhdl
wire clk, D;
reg Q;
always @(posedge clk) begin
    Q <= D;
end
```

• In this case, when there is a 0->1 transition of clk, the body of the always block occurs.
  • In this case, it assigns Q to the value of D
Reg vs. Wires

wire
- Used when connecting modules in structural Verilog
  wire a, b, clk;
- Used for continuous assignments
  wire a, b, c;
  assign a = b | c;

reg
- Despite what it’s name implies, reg types are **not** always registers
- Any assignment made inside an always block must be to a reg
  - Including always @(*) blocks
- If the sensitivity list contains an edge event (ex. posedge clk), reg types will likely be inferred as registers*
  - *depending on the type of assignment statement used
Multiple Assignments

- You cannot assign a wire more than once

```markdown
assign a = b;
assign a = c; //Bad!
```

- This creates a “multi-driver” net which is not allowed by most synthesis tools.
  - What happens when \( b \) is 0 and \( c \) is 1? Is ‘a’ 0 or 1?
  - What happens if \( b \) is 1 and \( c \) is 1? Is ‘a’ 0, 1, or 2?

- However, you can assign a reg in multiple places within an always @ block

```markdown
always @(posedge clk) begin
  a <= b;
  if(d>16'd5) begin
    a <= c;
  end
end
```

- The last assignment statement ”executed” will be the one that ultimately assigned.
  - \( a <= c \) if \( d > 16'd5 \)
  - Otherwise \( a <= b \)

- Can be used to set a “default value” for a reg
Blocking Vs. Nonblocking Assignments

**Blocking Assignment (=)**
- The assignment takes place *immediately* (with respect to other assignments).
- Any reference to the assigned reg in a later statement will see its new value.
- Use this for combinational logic

```verilog
reg c, out;
wire a, b, d;
always @(*) begin
    c = a | b;
    out = c & d;
end
```

Equivalent to: `out = (a | b) & d`

**Nonblocking Assignment (<=)**
- The assignment is *deferred* until the end of the time step (until all the right hand sides have been evaluated).
- Logic can reference the values of registers *before* they are updated in this cycle (ie. values immediately before the clock edge).
- Allows multiple registers to be written to simultaneously (order does not matter).
- Use this for sequential logic

```verilog
reg c, out;
wire a, b, clk;
always @(posedge clk) begin
    //use val of 'out' before clk edge
    c <= out | a;
    //use val of 'c' before clk edge
    out <= c & b;
end
```
Generate for loops are *not* like C loops

- You are *not* describing iterations of execution
  - Cannot store a value in a temporary variable to be read and overwritten in the next iteration
- Think of it like writing a little program that writes Verilog

```verilog
genvar i;
wire [2:0] a;
wire [3:0] b;
generate
  for(i = 0; i<3; i = i+1) begin:loop
    mod inst(.a(a[i]), .b(b[i]), .c(b[i+1]));
  end
endgenerate
```

- The generate loop is structurally equivalent to

```verilog
wire [2:0] a;
wire [3:0] b;
mod inst_loop_0(.a(a[0]), .b(b[0]), .c(b[1]));
mod inst_loop_1(.a(a[1]), .b(b[1]), .c(b[2]));
mod inst_loop_2(.a(a[2]), .b(b[2]), .c(b[3]));
```

- If you want a C like loop using a single instance of a module, you need to construct the control logic to manage the multi-cycle execution yourself – generate for will *not* do it for you
Simulating Verilog
Simulating Verilog

• ASIC Lab has started talking about this already
• FPGA Lab will cover it in Lab 3
• I’ll show you a simple example today

• Need a testbench to specify your test case
  • Typically is a module that instantiates the module your are interested as the DUT (device under test)
  • Typically uses an “initial” block to manipulate signals
    • Initial blocks are run at the start of the simulation
An Example Testbench

```verilog
`timescale 1ns/1ns

module reg_tester();

reg clk;
reg [4:0] a;
wire [4:0] b;

//Set the initial state of the clock
initial clk = 0;

//Every 4 timesteps (1ns/step) flip the clock
always #(4) clk <= ~clk;

//Instantiate the DUT
five_bit_flip_flop dut(.clk(clk), .d(a), .q(b));

initial begin

   $dumpfile("dump.vcd"); //Setup file dump (for waveform viewer)
   $dumpvars; //Dump signals to dumpfile
   a = 5’d0; //Set inputs
   //Print these values at the end of the current simulation step
   $strobe("time: %4d, a: %d, b: %d, clk: %b", $time, a, b, clk);
   #4; //Go for 4 ns
   $strobe("time: %4d, a: %d, b: %d, clk: %b", $time, a, b, clk);
   #1; //Go for 1 ns
   $strobe("time: %4d, a: %d, b: %d, clk: %b", $time, a, b, clk);
   #1; //Go for 1 ns
   a = 5’d2; //Set inputs
   $strobe("time: %4d, a: %d, b: %d, clk: %b", $time, a, b, clk);
   #6; //Go for 6 ns
   $strobe("time: %4d, a: %d, b: %d, clk: %b", $time, a, b, clk);
   #1; //Go for 1 ns
   $strobe("time: %4d, a: %d, b: %d, clk: %b", $time, a, b, clk);
   #8; //Run for another clock cycle + 1ns
   $strobe("time: %4d, a: %d, b: %d, clk: %b", $time, a, b, clk);
   $finish(); //End the simulation

end
endmodule
```
Result!

"X" means unknown

time: 0, a: 0, b: X, clk: 0

time: 4, a: 0, b: 0, clk: 1

time: 5, a: 0, b: 0, clk: 1

time: 6, a: 2, b: 0, clk: 1

time: 12, a: 2, b: 2, clk: 1

time: 13, a: 2, b: 2, clk: 1
Some useful “System Tasks”

- \$strobe("format str", values ...)
  - Prints values to the console. Is executed at the very end of the current cycle (after all changes have propagated)
    - In Verilog parlance, this executes after “all simulation events have occurred for the simulation time”
- \$time
  - Get the current simulation time
- \$monitor("format str", values ...)
  - Prints values to the console when any of them change
  - Only 1 monitor statement can be active at a time
- \$finish()
  - End the simulation
- \$display("format str", values ...)
  - Similar to \$strobe except it is not guaranteed to be executed at the end of the current cycle
`timescale 1ns/1ns

module reg_tester();
reg clk;
reg [4:0] a;
wire [4:0] b;

//Set the initial state of the clock
initial clk = 0;

//Every 4 timesteps (1ns/step) flip the clock
always #(4) clk <= ~clk;

//Instantiate the DUT
five_bit_flip_flop dut(.clk(clk), .d(a), .q(b));

initial begin
$dumpfile("dump.vcd"); //Setup file dump (for waveform viewer)
$dumpvars; //Dump signals to dumpfile

a = 5'd0; //Set inputs
#6; //Go for 6 ns
a = 5'd2; //Set inputs
#15; //Go for 6 ns
$finish(); //End the simulation
end

endmodule
Result!

time: 0, a: 0, b: x, clk: 0

time: 1, a: 0, b: x, clk: 0

time: 2, a: 0, b: x, clk: 0

time: 3, a: 0, b: x, clk: 0

time: 4, a: 0, b: 0, clk: 1

time: 5, a: 0, b: 0, clk: 1

time: 6, a: 2, b: 0, clk: 1

time: 7, a: 2, b: 0, clk: 1

time: 8, a: 2, b: 0, clk: 0

time: 9, a: 2, b: 0, clk: 0

time: 10, a: 2, b: 0, clk: 0

time: 11, a: 2, b: 0, clk: 0

time: 12, a: 2, b: 2, clk: 1

time: 13, a: 2, b: 2, clk: 1

time: 14, a: 2, b: 2, clk: 1

time: 15, a: 2, b: 2, clk: 1

time: 16, a: 2, b: 2, clk: 0

time: 17, a: 2, b: 2, clk: 0

time: 18, a: 2, b: 2, clk: 0

time: 19, a: 2, b: 2, clk: 0

time: 20, a: 2, b: 2, clk: 1


```verilog
`timescale 1ns/1ns

module reg_tester();

reg clk;

reg [4:0] a;

wire [4:0] b;

//Set the initial state of the clock
initial clk = 0;

//Every 4 timesteps (1ns) flip the clock
always #(4) clk <= ~clk;

//Instantiate the DUT
five_bit_flip_flop dut(.clk(clk), .d(a), .q(b));

initial begin
    $dumpfile("dump.vcd"); //Setup file dump (for waveform viewer)
    $dumpvars; //Dump signals to dumpfile
    a = 5'd0; //Set inputs
    #6; //Go for 6 ns
    a = 5'd2;
    #15; //Go for 6 ns
    $finish(); //End the simulation
end

initial begin
    $monitor("time: %4d, a: %d, b: %d, clk: %b", $time, a, b, clk);
end
endmodule
```
Results

time: 0, a: 0, b: x, clk: 0
time: 4, a: 0, b: 0, clk: 1
time: 6, a: 2, b: 0, clk: 1
time: 8, a: 2, b: 0, clk: 0
time: 12, a: 2, b: 2, clk: 1
time: 16, a: 2, b: 2, clk: 0
time: 20, a: 2, b: 2, clk: 1
Simulating in this class

• VCS
  • Introduced in ASIC Lab 2
  • Installed on Cory 125 Computers

• ModelSim
  • Introduced in FPGA Lab3
  • Installed on Cory 125 Computers
  • Educational version (PE) available for Windows

• Vivado Simulator
  • Introduced in FPGA Lab 3
  • Installed on Cory 125 Computers
  • Free version (WebPack) can be installed on Linux and Windows (Mac needs a VM)
    • Install Vivado 2017.4 if you are in the FPGA Lab

• EDA Playground (https://www.edaplayground.com)
  • Free Web Based Simulator
  • Requires Registration