EECS151/251A Discussion 2

Quincy Le Huynh
Jan. 31, 2020
Administrivia

• Bring your laptops to discussion to get started on HW
• Changing my OH to Friday 10a–11a, right after discussion
  – Same place 340 HMMB (knock on door or follow me when discussion ends)
• Gradescope for HW1 submission
• Any questions/comments/concerns?
  – Email me (quincy.huynh@berkeley.edu) and Tan (tan.nqd@berkeley.edu)
  – I will try my best to answer lab–related questions, but Tan is the best point of contact
Reclarifying # of Boolean Input/Outputs

- Rederiving what I said last time about the number of unique Boolean functions, I made a boo-boo
Verilog

- One of several Hardware Description Languages (HDLs)
- Not a traditional programming language
- Learn the mindset from scratch

- Don’t even think about it like a traditional programming language
- Stop thinking about it like a traditional programming language
- Stop it
- This isn’t C/Java/Python/Ruby/JavaScript/...
Verilog

- When you write Verilog, you’re saying what hardware is placed and how they’re connected
- 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
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
  - any signal that appears on the right hand side of assignment statements
  - any 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
```
(Old Way) The always @ block – Sequential Logic

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

```verilog
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
module REGISTER(q, d, clk);
    parameter N = 1;
    output reg [N-1:0] q;
    input [N-1:0] d;
    input clk;
    always @(posedge clk)
        q <= d;
endmodule
Example with the Explicit Register

// 4-bit wrap-around counter reset
module counter(value, enable, reset, clk);
    output [3:0] value;
    input enable, reset, clk;
    wire [3:0] next;
    REGISTER_R #(N(4)) state (.q(value), .d(next), .rst(reset), .clk(clk));
    assign next = value + 1;
endmodule

This means override parameter N to 4 instead of the default 1
Reg vs. Wires

**wire**
- Used when connecting modules in structural Verilog
  
  ```verilog
  wire a, b, clk;
  ```
- Used for continuous assignments
  
  ```verilog
  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.

```verilog
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.

```verilog
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 (i.e. values immediately before the clock edge).
- Allows multiple registers to be written to simultaneously (order does not matter).
- Used 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
- Kind of 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

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

• Need a testbench to specify your test case
  – Typically is a module that instantiates the module you 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
Simulating Verilog

• When writing your testbench, you can kinda go back to C–level coding, since it is procedural and you’re testing a vector of values over several clock cycles.
• You have access to:
  – for loops
  – tasks (subroutines)
  – print statements
An Example Testbench

```vhdl
`timescale 1ns/1ns
module reg_tester();
reg clk;
reg [3:0] a; // a is test input
wire [3:0] b; // b is test output

//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
REGISTER #(.N(4)) DUT(.q(b), .d(a), .clk(clk));
initial begin
  $dumpfile("dump.vcd"); //Setup file dump (for waveform viewer)
  $dumpvars; //Dump signals to dumpfile
  a = 4'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 = 4'd3; //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
```
"X" means unknown

Result!

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

- 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: 3, b: 0, clk: 1
- time: 12, a: 3, b: 3, clk: 1
- time: 13, a: 3, b: 3, clk: 1
Some useful “System Tasks”

- $\text{strobe("format str", values ...)}$
  - 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”
- $\text{time}$
  - Get the current simulation time
- $\text{monitor("format str", values ...)}$
  - Prints values to the console when any of them change
  - Only 1 monitor statement can be active at a time
- $\text{finish()}$
  - End the simulation
- $\text{display("format str", values ...)}$
  - Similar to $\text{strobe}$ except it is not guaranteed to be executed at the end of the current cycle
<table>
<thead>
<tr>
<th>Format str</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d or %D</td>
<td>Decimal format</td>
</tr>
<tr>
<td>%b or %B</td>
<td>Binary format</td>
</tr>
<tr>
<td>%h or %H</td>
<td>Hexadecimal format</td>
</tr>
<tr>
<td>%o or %O</td>
<td>Octal format</td>
</tr>
<tr>
<td>%c or %C</td>
<td>ASCII character format</td>
</tr>
<tr>
<td>%v or %V</td>
<td>Net signal strength</td>
</tr>
<tr>
<td>%m or %M</td>
<td>Hierarchical name</td>
</tr>
<tr>
<td>%s or %S</td>
<td>As a string</td>
</tr>
<tr>
<td>%t or %T</td>
<td>Current time format</td>
</tr>
</tbody>
</table>
Printing every 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
REGISTER #(N(4)) DUT(.q(b), .d(a), .clk(clk));

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
initial begin
  forever begin
    $strobe("time: %4d, a: %d, b: %d, clk: %b", $time, a, b, clk);
    #1;
  end
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: 1

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
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 @(posedge clk) clk <= ~clk;

//Instantiate the DUT
REGISTER #(.N(4)) DUT(.q(b), .d(a), .clk(clk));

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

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 Lab 3
  - 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](https://www.edaplayground.com)) ── Highly, highly recommend this for Homework!!!
More Simulation Practice

- Live Demo with 4-bit counter, running an exhaustive test!
- I’ll be better this time with the demo, trust.

```vhdl
// 4-bit wrap-around counter reset
module counter(value, enable, reset, clk);
  output [3:0] value;
  input enable, reset, clk;
  wire [3:0] next;
  REGISTER_R #(.N(4)) state (.q(value), .d(next), .rst(reset), .clk(clk));
  assign next = value + 1;
endmodule
```
More Simulation Practice

- Live Demo with 4-bit counter, running an exhaustive test!
- I’ll be better this time with the demo, trust.

```verilog
// 4-bit wrap-around counter reset
module counter(value, enable, reset, clk);
    output [3:0] value;
    input enable, reset, clk;
    wire [3:0] next;
    REGISTER_R #(N(4)) state (.q(value), .d(next), .rst(reset), .clk(clk));
    assign next = value + 1;
endmodule
```
Exhaustive Test

- What is an exhaustive test, really?
- In a 4-bit wrap-around counter you’d want to make sure:
  - It counts from 0000 to 1111 (a 4-bit counter)
  - When it reaches 1111, that it actually wraps around to 0000 in the next clock cycle
Exhaustive Test (tasks and for loop)

```verilog
`timescale 1ns/1ns
module test;
    reg clk, enable, reset;
    wire[3:0] value;

    // Set the initial state of the clock
    initial clk = 0;
    // Instantiate device under test
    counter COUNTER(.value(value),
                   .enable(enable),
                   .reset(reset),
                   .clk(clk));

    integer i;
    // Every 4 timesteps (1ns/step) flip the clock
    always #(4) clk <= ~clk;
    initial begin
        // Dump waves
        $dumpfile("dump.vcd");
        $dumpvars(1, test);

        clk = 0;
        enable = 1;
        reset = 1;
        toggle_clk;
        reset = 0;
        for (i = 0; i < 32; i = i + 1)
            begin
                if (i == 16)
                    $display("wrapping around!");
                $display("time: %4d, value: %b", $time, value);
                toggle_clk;
            end
        $finish();
    end

    task toggle_clk;
    begin
        #4 clk = ~clk;
        #4 clk = ~clk;
    end
endtask
```
Subroutines, or Tasks

```
`timescale 1ns/1ns
module test;
  reg clk, enable, reset;
  wire[3:0] value;

  // Set the initial state of the clock
  initial clk = 0;
  // Instantiate device under test
  counter COUNTER(.value(value),
      .enable(enable),
      .reset(reset),
      .clk(clk));

  integer i;
  // Every 4 timesteps (1ns/step) flip the clock
  always #(4) clk <= ~clk;
  initial begin
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars(1, test);
    clk = 0;
    enable = 1;
    reset = 1;
    toggle_clk;
    reset = 0;
    for (i = 0; i < 32; i = i + 1)
      begin
        if (i == 16)
          $display("wrapping around!");
        $display("time: %4d, value: %b", $time, value);
        toggle_clk;
      end
    $finish();
  end

  task toggle_clk;
  begin
    #4 clk = ~clk;
    #4 clk = ~clk;
  end
endtask
```
For loop to test several values

```verilog
`timescale 1ns/1ns
module test;
  reg clk, enable, reset;
  wire[3:0] value;

  //Set the initial state of the clock
  initial clk = 0;
  // Instantiate device under test
  counter COUNTER(.value(value),
                   .enable(enable),
                   .reset(reset),
                   .clk(clk));

  integer i;
  // Every 4 timesteps (1ns/step) flip the clock
  always #(4) clk <= ~clk;
  initial begin
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars(1, test);

    for (i = 0; i < 32; i = i + 1)
      begin
        if (i == 16)
          $display("wrapping around!");
          $display("time: %4d, value: %b", $time, value);
        toggle_clk;
      end

    $finish();
  end

  task toggle_clk;
  begin
    #4 clk = ~clk;
    #4 clk = ~clk;
  end
endtask
```
Checking part of the test vector

```verilog
//timescale 1ns/1ns
module test;
    reg clk, enable, reset;
    wire[3:0] value;

    //Set the initial state of the clock
    initial clk = 0;
    // Instantiate device under test
    counter COUNTER(.value(value),
                        .enable(enable),
                        .reset(reset),
                        .clk(clk));

    integer i;
    //Every 4 timesteps (1ns/step) flip the clock
    always #(4) clk <= ~clk;
    initial begin
        // Dump waves
        $dumpfile("dump.vcd");
        $dumpvars(1, test);
    end

    task toggle_clk;
    begin
        #4 clk = ~clk;
        #4 clk = ~clk;
    end
endtask

clk = 0;
    enable = 1;
    reset = 1;
    toggle_clk;
    reset = 0;
    for (i = 0; i < 32; i = i + 1)
        begin
            if (i == 16)
                $display("wrapping around!");
            $display("time: %4d, value: %b", $time, value);
            toggle_clk;
        end
    $finish();
end
```

Berkeley
UNIVERSITY OF CALIFORNIA
Results

time:  8, value: 0000
time: 16, value: 0001
time: 24, value: 0010
time: 32, value: 0011
time: 40, value: 0100
time: 48, value: 0101
time: 56, value: 0110
time: 64, value: 0111
time: 72, value: 1000
time: 80, value: 1001
time: 88, value: 1010
time: 96, value: 1011
time: 104, value: 1100
time: 112, value: 1101
time: 120, value: 1110
time: 128, value: 1111
wrapping around!
time: 136, value: 0000
time: 144, value: 0001
time: 152, value: 0010
LTspice issues?

• If time allows, I will go try to demo LTspice again