EECS 151/251A
Discussion 3

Zhaokai Liu
9/14, 9/15 and 9/20
About Me (Zhaokai Liu)

- 5th year PhD student advised by Bora
- Research in mixed-signal design, ADC, analog circuit automation, ...
- Took this class 4 years ago

Office hour
- Friday 2-3pm
Agenda

- Administrivia
- Verilog
- Simulating Verilog
Administrivia

- Homework 2 due 11:59pm, Friday, 9/17
- Homework 3 out this Thursday
Verilog
module this_is_wrong_1 ( 
    input a, 
    input b, 
    input reg rst 
); 
    wire x; 
    always @(a or b) begin 
        assign x = a + b; 
    end 
    always @(rst) begin 
        assign x = 1'b0; 
    end 
endmodule

module this_is_wrong_2 ( 
    input [N-1:0] a, 
    input b, 
    output reg c 
); 
    wire [N:0] x; 
    generate 
        for (i=0; i<N; i=i+1) begin 
            SubMod submod(.in0(a[i]), .in1(x[i]), .out(x[i+1])) 
        end 
    endgenerate 
    assign c = x[N]; 
endmodule
Race Conditions: Synthesis vs. Simulation

Want: a register pipeline

Determine:
1. Does it **synthesize** correctly?
2. Does it **simulate** correctly?
   - Note: always blocks may simulate in any order
3. Is it good coding practice?

<table>
<thead>
<tr>
<th>Candidate #1:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>always @(posedge clk) begin</code></td>
</tr>
<tr>
<td><code>q1 = d;</code></td>
</tr>
<tr>
<td><code>q2 = q1;</code></td>
</tr>
<tr>
<td><code>q3 = q2;</code></td>
</tr>
<tr>
<td><code>end</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Candidate #2:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>always @(posedge clk) begin</code></td>
</tr>
<tr>
<td><code>q3 = q2;</code></td>
</tr>
<tr>
<td><code>q2 = q1;</code></td>
</tr>
<tr>
<td><code>q1 = d;</code></td>
</tr>
<tr>
<td><code>end</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Candidate #3:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>always @(posedge clk) begin</code></td>
</tr>
<tr>
<td><code>q1 &lt;= d;</code></td>
</tr>
<tr>
<td><code>q2 &lt;= q1;</code></td>
</tr>
<tr>
<td><code>q3 &lt;= q2;</code></td>
</tr>
<tr>
<td><code>end</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Candidate #4:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>always @(posedge clk) q1 = d;</code></td>
</tr>
<tr>
<td><code>always @(posedge clk) q2 = q1;</code></td>
</tr>
<tr>
<td><code>always @(posedge clk) q3 = q2;</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Candidate #5:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>always @(posedge clk) q3 = q2;</code></td>
</tr>
<tr>
<td><code>always @(posedge clk) q2 = q1;</code></td>
</tr>
<tr>
<td><code>always @(posedge clk) q1 = d;</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Candidate #6:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>always @(posedge clk) q1 &lt;= d;</code></td>
</tr>
<tr>
<td><code>always @(posedge clk) q2 &lt;= q1;</code></td>
</tr>
<tr>
<td><code>always @(posedge clk) q3 &lt;= q2;</code></td>
</tr>
</tbody>
</table>
Simulating Verilog
Adder

We can test RTL via simulation before putting it on the FPGA of fabricating an ASIC. Let’s see a 32 bit adder.

```verilog
module adder(
    input [31:0] a,
    input [31:0] b,
    output [31:0] c
);
    assign c = a + b;
endmodule

or

module adder(a, b, c);
    input [31:0] a, b;
    output [31:0] c;
    assign c = a + b;
endmodule
```

- The adder module is **synthesizable**. It can be implemented on ASIC (gates, optimized by compiler) or FPGA (LUTs).
- Note that 32-bit $a + 32$-bit $b$ produces a **33-bit** result, which is truncated to 32-bit when assigning to $c$.
- In the test bench, we will refer to the adder module as DUT (design under test).
Testbench

module adder_tester();
  reg [31:0] a, b;
  wire [31:0] c;
  adder dut(.a(a), .b(b), .c(c)); // device under test
  // ... initial block of adder_tester
endmodule

- The `adder_tester` module is not synthesizable. It is executed by an RTL simulator.
- Note we have created reg nets to drive the DUT’s inputs, and wire nets to sense the DUT’s outputs.
- Testbench is super important! Make sure you have covered the circumstances as many as possible. Basically you can do nothing with the bugs on a tape-out chip.
Testbench initial block

initial begin
  a = 32'd1;
  b = 32'd2;
  #1; // wait for 1 timestep
  if (c != 32'd3) begin
    $display("FAILED");
  end
  a = 32'd5;
  b = 32'd10;
  #1;
  if (c != 32'd15) begin
    $display("FAILED");
  end
  $finish();
end

● The initial block defines the ‘entry point’ of the simulator. It executes only once at time zero
● The code in initial block runs sequentially, just like other languages (C, python)
● The input signals are driven using blocking (=) assignments
● #(n) is used to advance simulation time by n timesteps.
● $display and $finish are system functions
Timescales and timesteps

```
timescale 1ns/10ps
```

- `timescale declaration is at the top of each testbench`
- Simulation time step defines how much time is advanced when running #1
- Simulation time resolution defines the smallest amount of time can be advanced
- In the example, #1 = 1ns; #0.01 is the smallest possible delay
Delay

- Delay at LHS: Everything happens after the time step
- Delay at RHS: RHS is evaluated first, and assigned to the left at the end of that time step
- Pay attention if blocking/non-blocking assignments are both used (not recommended)

```verilog
reg [7:0] r;
reg a, b;
wire c, d;
initial begin
    a = 0;
    b = 1;
    #1 r = 8'd20;
    r = #1 8'b0001_0100;
end

assign #3 c = a & b;
and #3 (d, a, b);
// an AND gate with 3 timestep delay
```
Blocking vs Non-blocking + Delay

```verilog
initial begin  initial begin  initial begin
  a <= 0;    a <= 0;    a <= 0;
  b <= 0;    b <= 0;    b <= 0;
  q <= 0;    q <= 0;    q <= 0;
  #5 a <= 1;  #5 a <= 1;  #5 a = 1;
  b <= 1;    b <= 1;    b = 1;
  q <= a & b; q <= #5 a & b; q <= #5 a & b;
end        end        end
/* Output: */  /* Output: */  /* Output: */
  t=0:  a=0, b=0, q=0  t=0: a=0, b=0, q=0  t=0:  a=0, b=0, q=0
  t=5:  a=1, b=1, q=0  t=5: a=1, b=1, q=0  t=5:  a=1, b=1, q=0
  t=10: a=1, b=1, q=1  No change after t=5!  t=10: a=1, b=1, q=1
*/        /*

• This article is helpful for understanding!
```
Adder Demo

https://www.edaplayground.com/x/fHwj
Exhaustive Testing Demo

A small adder can be tested exhaustively by using nested for loops.

https://www.edaplayground.com/x/JEZg
A large adder can’t be tested exhaustively since it would take too much time. Instead test it using random stimulus.

https://www.edaplayground.com/x/mEVz
module counter(input clk);
    reg [3:0] count;
    always @(posedge clk) begin
        count <= count + 'd1;
    end
endmodule

- All *registers* begin with an initial value of `x' in simulation unless otherwise specified
- Every signal in Verilog has 4 potential states: 0, 1, `x` (unknown), `z` (high-impedance/unconnected)
- We can set initial values of *registers* with `initial count = 0`
- Initial values are synthesizable on some FPGAs but not on ASICs, so using a *reset* is recommended
Simulation Constructs

@ (posedge signal);
@ (posedge signal);
// wait for 2 rising edges of signal
repeat (10) @(negedge clk);
// wait for 10 falling edge of clk

reg clk;
initial clk = 0;
always #(10) clk <= ~clk;
// an easy way to create a clock in testbench
$display

$display("Wire x in decimal is %d", x);
$display("Wire x in binary is %b", x);
// Similar to printf() in C

module x(input clk, input valid, input data);
    always @ (posedge clk) begin
        if (valid) $display("Data is %d", data);
    end
endmodule
// $display can be used inside RTL as well
Counter Demo

Run through the counter testbench. Deal with unknown values, initialize registers, add a reset. Use $display in the DUT.

https://www.edaplayground.com/x/mmHN
Generate Macros

```verilog
genvar i;
generate for (i = 0; i < n; i = i + 1) begin
  full_adder u0(.a(a[i]), .b(b[i]), .cin(c[i]), .s(s[i]), .cout(c[i+1]));
end
endgenerate

// there is a module parameter named mult_option
generate if (mult_option == 1)
  assign z = x * y;
else
  assign z = x + y; // will you see a multiplier and an adder on the hardware
endgenerate // at the same time?
```

- Generate macros (generate for and generate if) can be used to programmatically instantiate hardware
2D Regs + Memories

// A memory structure that has eight 32-bit elements
reg [31:0] mem [7:0];

mem[2]
// The 3rd 32-bit element

mem[5][7:0]
// The lowest Byte (8-bit) of the 6th 32-bit element