EECS 151/251A: Discussion 3

Verilog Simulation, Parameterized Modules

9/12/2019

Simulation Basics

Let's test a very simple Verilog module.

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

- The `adder` module is *synthesizable*; it belongs on an ASIC or FPGA
- In a testbench, we refer to the `adder` module as the *DUT* (device under test)
- Note Verilog's truncation rules: `a + b` produces a 33 bit number, which is truncated to 32 bits when assigned to `c`

The 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 Initial Block

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

- The `initial` block defines the 'entry point' of the simulator
- The code in the `initial` block is run *sequentially*, just like software
- The DUT is 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 (simulation time step)/(simulation time resolution)
```

- At the top of each testbench, there's a `` `timescale `` declaration
- The simulation time step defines how much time is advanced when running `#1;`
- The simulation time resolution defines the smallest amount time can be advanced
- In the example, `#1;` = 1ns and `#0.010;` is the smallest delay possible

Adder Testbench Demo

Demo
Run through the adder testbench. See the waveform and play with time delays.

4-State Signals in Verilog

```
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

Wait for 2 rising edges of `signal`:

```
@(posedge signal);
@(posedge signal);
```

Wait for 10 rising `clk` edges:

```
repeat (10) @(posedge clk);
```

Generate a clock signal:

```
reg clk;
initial clk = 0;
always #(10) clk <= ~clk;
```

$display

The `$display()` system function is similar to `printf()` in C.

```
$display("Wire x in decimal is %d", x);
$display("Wire x in binary is %b", x);
```

`$display()` can be used in `initial` blocks as well as inside RTL.

```
module x(input clk, input valid, input data);
  always @(posedge clk) begin
    if (valid) $display("Data is %d", data);
  end
endmodule
```

Counter Demo

Demo

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

Generate Macros

Generate macros (`generate for` and `generate if`) can be used to programmatically instantiate hardware. Stamp out `n` instances of `mod` and connected them to wires `k` and `s`.

```
genvar i;
generate for (i = 0; i < n; i = i + 1) begin
  mod name(.a(s[i]), .b(k[i+1]));
end endgenerate
```

If there's a module parameter `fast_mult` to choose multiplication speed

```
generate if (fast_mult == 1)
  assign c = a * b;
else
  shift_and_add_mult(.a(a), .b(b), .c(c));
endgenerate
```

Shift Register Example

Demo
Run through the shift register testbench. Work through the generate-for statement.

Parameterized Modules

Modules can have integer parameters to make them ‘generators’:

```
module adder #(parameter width=32)
  (input [width-1:0] a,
   input [width-1:0] b,
   output [width:0] s);
   assign s = a + b;
endmodule
```

This adder can be re-used with different parameterizations:

```
wire [31:0] a32, b32; wire [32:0] c1;
wire [63:0] a64, b64; wire [64:0] c2;
adder #(.width(32)) adder1 (.a(a32), .b(b32), .c(c1));
adder #(.width(64)) adder2 (.a(a64), .b(b64), .c(c2));
```

Shift Register Example

Demo
Make the shift register a parameterized module.

Exhaustive Testing

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

Randomized Testing

Demo

A large adder can’t be tested exhaustively since it would take too much time.

Instead test it using random stimulus.

2D Regs + Memories

```
// A memory structure that has eight 32-bit elements
reg [31:0] fifo_ram [7:0];
fifo_ram[2] // The 3rd 32-bit element
fifo_ram[5][7:0] // The lowest byte of the 6th 32-bit element
```

Demo
Test the simple memory!