CSE 141L: Introduction to Computer Architecture Lab
SystemVerilog Verification

Pat Pannuto, UC San Diego
ppannuto@ucsd.edu
Milestone 2 is due in 9 days

• What to submit?
  – SOMETHING

• M1 feedback
  – Should be released by Wednesday
  – Pay attention to things that should be revised for M2

• M2 is about proving individual components work
  – How would you prove to your manager your component works?
Today’s Objectives: Validation & Verification in Hardware Design

• Real-world hardware design process
  – And where we are cutting corners to simplify for class

• Mapping verification in theory to verification in practice
The hardware design process

Validation: Have we built the right thing?
Verification: Have we built the thing right?

Spec (English language)

Model (higher-level language [C, Python, etc])

RTL Logic Design (HDL, e.g. Verilog)

Physical Design (layout; VLSI/ASIC, FPGA)

Manufactured Part (actual hardware)

Validation: Does the model satisfy the spec?
Does the RTL match the model?

Verification: Does the physical design work?

Test: Does the part work?
V&V is standard practice across engineering disciplines
So what is verification to us?

• Complete validation of all functionality of Device Under Test (DUT)
  – Q: Did the ALU testbench example last week do this?
    • How do you know that it did / didn’t?

• Mechanistically: Stimulating DUT with all possible inputs
Conceptual view of Verification

Stimulus Generator

\{\text{say, an adder}\}

DUT

“Golden results”

Reference Implementation

Compare & VERIFY

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Pragmatic view of Verification [for this class!]

Stimulus Generator

A

B

“Golden results”

Reference Implementation

adder_tb.sv

DUT

{say, an adder}

C

S

Compare & VERIFY

C

S

adder.sv
Can we find these pieces in last week’s `alu_tb.sv`?

```verilog
`timescale 1ns / 1ps

module ALU_tb;

// Signals to interface with the ALU module
logic [7:0] INPUTA; // data inputs
logic [7:0] INPUTB;
logic [2:0] op; // ALU opcode
bit SC_IN = 'b0;
wire[7:0] OUT;
wire Zero;

// Define a helper wire for comparison
logic [7:0] expected;

// Loop variables
integer i, j;

// Instantiate and connect Unit Under Test
ALU uut(
  .InputA(INPUTA),
  .InputB(INPUTB),
  .SC_in(SC_IN),
  .OP(op),
  .Out(OUT),
  .Zero(Zero)
);

// The actual testbench logic
initial begin
  $display("STarting!");
  INPUTA = 1;
  INPUTB = 1;
  op = 'b000; // ADD
  test_alu_func; // void function call
  #5;

  INPUTA = 4;
  INPUTB = 1;
  op = 'b100; // AND
  test_alu_func; // void function call
  #5;

  op = 'b011; // XOR
  for (i=0; i<256; i++) begin
    for (j=0; j<256; j++) begin
      INPUTA = i;
      INPUTB = j;
      test_alu_func;
      #5;
    end // j end
  end // i end

  $display("End: all test cases passed.");
end // initial begin's end

endmodule
```

task test_alu_func; begin
  case (op)
    0: expected = INPUTA + INPUTB; // ADD
    1: expected = {INPUTA[6:0], SC_IN}; // LSH
    2: expected = {1'b0, INPUTA[7:1]}; // RSH
    3: expected = INPUTA ^ INPUTB; // XOR
    4: expected = INPUTA & INPUTB; // AND
    5: expected = INPUTA - INPUTB; // SUB
  endcase
  #1;
  if(expected == OUT) begin
    $display("$t YAY!! inputs = %h %h, opcode = %b, zero %b",$time, INPUTA,INPUTB,op, Zero);
  end else begin
    $display("$t FAIL! inputs = %h %h, opcode = %b, zero %b",$time, INPUTA,INPUTB,op, Zero);
    $stop;
  end
end
endtask
```

Straight talk: Some of the ‘little tests’ feel silly

• But I promise it feels worse when a bug was a typo in an ‘easy’ module

• Take advantage of groups
  – One of you implement your ALU, according to your specification
  – Someone else implement the ALU testbench, according to your specification
  – ... does it actually match?
Pragmatic considerations for verification

• A few slides back...
  – “Mechanistically: Stimulating DUT with all possible inputs”

• What defines “all possible inputs” for
  – A half-adder?
  – Our example ALU?
  – Your processor?
So what can we do to actually test well?

- Exhaustive coverage?
- Principled, corner-case test design?
- Randomized coverage?
- All of the above?
Questions on anything so far?
Then hopping over to look at some examples of `rand`

- SystemVerilog random testing?

```verilog
rand bit [7:0] inputA // `rand` picks random values independently
rand bit [7:0] inputB // and can repeat choices throughout the run
randc bit [2:0] opc // “cycle” `rand` won’t repeat until all seen

constraint legal_ops { opc < 6; /* can add more here */ }
```

We’ll fix this Wednesday with alternative approach!
Open Q&A on testbench design, more live examples