EE 531: ADVANCED VLSI DESIGN

Testbench Basics for Combinational Logic

Nishith N. Chakraborty

January, 2025

OUTLINE

- Displaying simulation results – more than viewing waveforms
- Exhaustive testing of basic combinational circuits
- Testing logic via comparison of different implementations
- Testbenches with enumerations

TESTBENCH BASICS

- A virtual laboratory bench – with virtual logic analyzer and signal generator
- Testbench written in code (e.g., SystemVerilog) to test design
- Testbench:
    - Exercises internal variables that stimulate design under test (DUT)
    - May print or return values from DUT useful for debugging

- A testbench is not synthesized!
- Typically, you do not include a port list as part of a testbench

PRINTING VALUES – `$MONITOR( … )`

- Concurrently acting statement
- Prints when one of its inputs changes
- Values printed are those from end of the simulation time step
  - Intermediate values changing in an always block during time step not printed
- There can only be one `$monitor(...)` active

```
initial begin
  // prints " count=..., y=..." when cnt or result changes
  //   %b indicates the value printed is a binary vector
  $monitor($time, " count=%b, y=%b", cnt, result);

  ...
end
```

CAL POLY
SAN LUIS OBISPO

PRINTING VALUES – `$DISPLAY( … )`

- Like a print statement in a programming language
- When always block (e.g., `initial, always_comb`) executes, it prints
- Values at instant of execution are printed, even intermediate during execution of always block – not necessarily consistent with value at end of time step
- Useful for printing information about a procedural block as it is executing

```
always_comb begin
  // prints " count=..., y=..." for values of
  //   cnt and result at this instant of execution
  $display(" count=%d", cnt); // prints old cnt
  cnt = cnt + 1;
  $display(" count=%d", cnt); // prints new cnt
  ...
end
```

PRINTING VALUES – `$STROBE( … )`

- Differs from display only by when it prints
- Prints when always block executed, regardless of change on inputs
- Only prints at end of the time step

```
always_comb begin
  // prints " count=..., y=..." for values of
  //   cnt and result at end of always_comb execution
  $strobe(" count=%b", cnt); // prints new cnt at end
  cnt = cnt + 1;
  ...
end
```

MESSAGE PRINTING EXAMPLE

```
module my_tb;
  logic [3:0] a,b;
  integer     i;

  initial begin
    $monitor($time, "monitor a:%h b:%h", a, b);
    for(i=0; i<4; i=i+1) begin
      $strobe("strobe a:%h b:%h", a, b);
      $display("display a:%h b:%h", a, b);
      case(i)
        0: a = 4;
        1: b = 1;
        3: {a,b} = 9;
      endcase
      $display("display a:%h b:%h", a, b);
      #1;
    end
  end
endmodule: my_tb
```

QuestaSim Output:

```
VSIM 7> run -all
# display a:x b:x
# display a:4 b:x
#                    0monitor a:4 b:x
# strobe a:4 b:x
# display a:4 b:x
# display a:4 b:1
#                    1monitor a:4 b:1
# strobe a:4 b:1
# display a:4 b:1
# display a:4 b:1
# strobe a:4 b:1
# display a:4 b:1
# display a:0 b:9
#                    3monitor a:0 b:9
# strobe a:0 b:9
```

BASIC TESTBENCH FOR COMBINATIONAL LOGIC

- Very straightforward approach, useful for testing very basic designs (exhaustive)
- Module being tested is the *design under test* (DUT)
- Consider the following as the DUT:

```
module if2
   (input  logic a,b,c,
    output logic f);

   always_comb begin
     if(a&b) f=1;
     else if(c & a ^ b)
       f=1;
     else f=0;
   end
endmodule: if2
```

BASIC TESTBENCH FOR COMBINATIONAL LOGIC

- SystemVerilog testbench for the `if2` DUT:

```
module top;
  logic [2:0] count;
  logic       result;

  // instantiate the design under test (DUT)
  if2 dut(count[2],count[1],count[0],result);

  initial begin
    $monitor($time, " abc=%b,f=%b",
             count, result);
    for(count=0; count != 3'b111; count++)
      #1;
    #1 $finish;
  end
endmodule: top
```

QuestaSim Output:

```
VSIM 13> run -all
#                    0 abc=000,f=0
#                    1 abc=001,f=0
#                    2 abc=010,f=1
#                    3 abc=011,f=1
#                    4 abc=100,f=0
#                    5 abc=101,f=1
#                    6 abc=110,f=1
#                    7 abc=111,f=1
```

IMPORTANT NOTES ABOUT BASIC TESTBENCH EXAMPLE

- The testbench presents all possible input values to the design (exhaustive)
- Behavior of the testbench is entirely captured in the `initial` block
  - Depends on delays (e.g., #1) and other procedural statements not allowed for design
- Description of the `if2` DUT does not include any debugging code
- Testbench doesn't use `program` construct – useful to SystemVerilog verification

TESTING MORE COMPLEX SYSTEMS

- Consider the following adder as a DUT

```
module adder
   (input  logic a,b,cI,
    output logic s,cO);

   assign s  = a ^ b ^ cI,
          cO = (a&b) | (a&cI) | (b&cI);
endmodule: adder

module fourBitAdder
   (input  logic [3:0] a,b,
   output logic [3:0] sum,
    input   logic      cIn,
   output   logic      cOut);
   logic [2:0] c;

   adder b0(a[0],b[0],cIn,sum[0],c[0]);
   adder b1(a[1],b[1],c[0],sum[1],c[1]);
   adder b2(a[2],b[2],c[1],sum[2],c[2]);
   adder b3(a[3],b[3],c[2],sum[3],cOut);
endmodule: fourBitAdder
```

TESTING MORE COMPLEX SYSTEMS

- SystemVerilog testbench for the adder DUT:

```
module test;
  logic [3:0] s;
  logic       cOut;
  logic [9:0] count;

  // instantiate the design under test (DUT)
  fourBitAdder
    add0(count[7:4],count[3:0],s,count[8],cOut);

  initial begin
    for(count=0; count != 3'b111; count++)
      #1 if({cOut,s} !==(count[7:4] + count[3:0] + count[8]))
        $display("  oops! %d != %d + %d + %d",
          {cOut,s}, count[7:4], count[3:0], count[8]);
    $finish;
  end
endmodule: test
```

TAKEAWAYS FROM THE ADDER EXAMPLE

- Shows how alternative methods for a design specification can be used to test an implementation via comparison
  - Here, structural `fourBitAdder` is compared to behavioral a + b + cin
- Parts of a counter variable provide a good way to test all combinations
- The #1 delay is needed (even though combinational) to allow the counter to generate and test for different input combinations
- Case equality/inequality useful when there is a possibility of one or both operands being X or Z values

TESTING LOGIC CONTAINING ENUMERATIONS

- Consider the following DUT with enumerations

```
typedef enum bit [2:0] {
  ADD = 3'b100,
  SUB = 3'b010,
  AND = 3'b001,
  OR  = 3'b110,
  XOR = 3'b011} aluFun_t;

module aluWithEnums
  (input  logic [7:0] a,b,
   output logic [7:0] result,
   input  aluFun_t    op);
  always_comb
    unique case (op)
      ADD: result = a+b;
      SUB: result = a-b;
      AND: result = a&b;
      OR:  result = a|b;
      XOR: result = a^b;
    endcase
endmodule: aluWithEnums
```

TESTING LOGIC CONTAINING ENUMERATIONS

- A testbench ready to handle enumerations of the ALU DUT

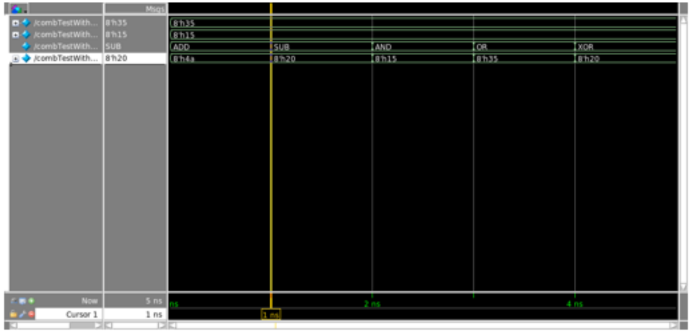
```
module combTestWithEnums;
  aluFun_t  op;
  bit [7:0] a, b, result;

  // .* -- connect all names that match
  aluWithEnums dut(.*);

  initial begin
    $monitor($time,
             “  %h = %h %s %h”,
             result, a, op.name, b);
    for (op=op.first; 1; op=op.next) begin
      a = 8'h35; b = 8'h15;
      #1 if (op == op.last) break;
    end
  end
endmodule: combTestWithEnums
```

QuestaSim Output:

```
VSIM 4> run -all
#                    0  4a = 35 ADD 15
#                    1  20 = 35 SUB 15
#                    2  15 = 35 AND 15
#                    3  35 = 35 OR 15
#                    4  20 = 35 XOR 15
```



USEFUL METHODS IN ENUMERATED VARIABLES

- `name` – provides a string that can be printed with $monitor or $display
  - Corresponds to the enumerated string value, for example the op AND
- `first` – returns value of the first enumerated label
- `next` – returns value of the next enumerated label in the list
- `last` – returns value of the last enumerated label in the list

Thank you!