Verilog Testbenches

**Motivation:** Design testbenches that exercise a design to achieve complete coverage (ideally).

**Overview:**

- **Directed Testing:** Testing that exercises a design for “targeted” features.
- **Constrained Random Testing:** Testing that utilizes random stimuli to exercise a design. “Discover” new corners, reach convergence faster.
  - Layered testbenches
  - Functional coverage
  - Towards UVM

Figure 1: The ever changing composition of project teams (split along design/verification) [1].

Figure 2: Pentium is a classic example of hardware errors with serious consequences. (FDIV, F00F bugs) [2].
Device Under Test (DUT)

Basic Example:
- A simple machine that counts from 0 → 1 → 2 continuously is shown on the right.
- This is actually a Moore FSM (to be discussed next week).

```verilog
module moore_counter(
    input wire clk, rst,
    input wire en,
    output reg [1:0] out
);

// Use systemverilog enumeration for convenience
typedef enum logic [1:0] {S0 = 2'b00, S1 = 2'b01, S2 = 2'b10} state_t;
// Sequential elements
reg [1:0] state;
reg [1:0] next_state;
// Constant assignment
assign out = state;

always @ (*) begin
    case (state)
        S0: next_state = en ? S1: state;
        S1: next_state = en ? S2: state;
        S2: next_state = en ? S0: state;
        default: next_state = S0;
    endcase
end

always @(posedge clk or rst)begin
    if (rst) begin
        state <= S0;
    end
    else begin
        state <= next_state;
    end
end
endmodule
Testbench Example

timescale 1 ns / 100 ps

module moore_counter_testbench;
  // Setup a clock
  reg clk = 1’b0;
  always #0.5 clk = ~clk;
  // Instantiate DUT and wire/reg the inputs and outputs
  reg rst;
  reg en;
  wire [1:0] out;

  moore_counter dut (  
    .clk(clk),  
    .rst(rst),  
    .en(en),  
    .out(out)  
  );

  // Basic directed test
  initial begin
    $dumpvars;
    $dumpfile("test.vcd");
    rst = 1’b1;
    en = 1’b0;
    #2;
    rst = 1’b0;
    en = 1’b1;
    #1;
    assert (out == 2’b01) $display ("[Passed] Out == 2’b01");
    else $error("[Failed] Out (%b) != 2’b01", out);
    #1;
    assert (out == 2’b10) $display ("[Passed] Out == 2’b10");
    else $error("[Failed] Out (%b) != 2’b10", out);
    #1;
    assert (out == 2’b00) $display ("[Passed] Out == 2’b00");
    else $error("[Failed] Out (%b) != 2’b00", out);
    #1
    $dumpoff;
    $finish;
  end
endmodule
Better Testbench

```verilog
int passes = 0;
int out_ref = 0;
int delay;

// Some local parameters
localparam PASSES = 10;
localparam RANDOM_RANGE = 10;

// Define basic checking task
// Note: Only immediate assertions within the task
task automatic random_delay_test;
begin
  // Generate unsigned random delay
delay = $urandom% RANDOM_RANGE;

  // Activate en for 1 cycle
  en = 1'b1;
  # 1
  en = 1'b0;
  # delay

  // Update the reference
  if(out_ref == 2) begin
    out_ref = 0;
  end
  else begin
    out_ref = out_ref + 1;
  end

  // Assertions
  assert (out == out_ref) $display("[Passed] Out (%d) == Ref (%d)", out, out_ref);
  else $error("[Failed] (%d) != Ref (%d)", out, out_ref);

  // Update the passes completed
  passes = passes + 1;
end
endtask

initial begin
  $dumpfile("test.vcd");
  $dumpvars;
  // Initial Reset
  en = 1'b0;
  rst = 1'b1;
  #1;
  rst = 1'b0;

  // Loop
  while (passes < PASSES) begin
    random_delay_test;
  end
  $dumpoff;
  $finish;
end
```
Thinking About Testbenches

About Synthesis: Testbenches use powerful constructs to ease simulation. However, we need to be aware of which constructs exist only in testbenches [3], [4]:

- **Delays (#):** It has no physical representation.
- **Directives:** For convenience directives, only use to calculate static, elaboration time constants.
- **Task/Function:** Do not implement sequential logic in hardware with these.
- **Initial:** Cannot be used to set starting values in ASIC. Generally, for both ASIC, FPGA, consider how you can use `reset` to achieve an intended function.
DeMorgan’s Law: Bubble Pushing

**Overview:** A tool in logic simplification (Boolean algebra). **Why could simplification be useful?**

**Key:** \((x+y)' = x'y', \ (xy)' = x' + y'\)

**For Gates:**
Practically, the above statement is realized as:
- Swap *and* for *or* (or vice versa).
- For backwards pushes, transfer bubble to all inputs.
- For forwards pushes, transfer bubble to output.

*Figure:* Backwards push (left) and forwards push (right).
Exercise 1: A Closer Look at DeMorgan’s Law

Towards Generality: Determine if \((x+y+z)’ \neq x’y’z’\) is true.
Exercise 1.5: Bubble Pushing In Action
# Sum of Products

## Truth Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

## SoP

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
## Product Of Sums

### Truth Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### PoS

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Exercise 2: Writing a SoP

**Truth Table**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Question:** Find the SoP for the following truth table.
Karnaugh Maps

Overview: Karnaugh maps are a visual method of simplification (as opposed to Boolean algebra). When a limited number of variables are involved, K-Maps are quite convenient.

Key: Karnaugh maps are built on two properties:

- $1+A = 1$
- $A+A' = 1$

Since the minimizations above operate on a term and its inverse, we must use gray coding to setup the map. Use a binary representation where successive values vary by 1 bit.
Karnaugh Maps (Cont.)

Simple Example:
\[ f(A,B) = A'B + A' \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Map:

Result:
Value of B has no consequence on output.

We group \( A'B', A'B \) yielding simplified result \( A' \).
Karnaugh Maps (Cont.)

Some Tips: Now that we have introduced Karnaugh maps, it’s useful to consider some general tips for solving these.

- Use groups of either 1’s or powers of 2 (1, 2, 4, 8 ...).
- Use the largest cover group where possible, and as few groups as possible.
- Overlap is okay, and you can wrap across boundaries when possible.

![Karnaugh Map Diagram]
Exercise 3: Practicing K-Maps

A Three Input Table! For the following SOP, demonstrate simplification using K-Maps.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Exercise 4: Simplifying Complex Maps
Sources