Content

• FPGA Architecture
• ASIC Architecture
• Shift Registers
• Verilog Testbenches
• Problems
FPGA Architecture

- Lookup Tables (LUT) are the fundamental building block
  - The inputs index a table loaded with values
  - Programming write the LUTs
  - Many variations (ex. LUT–2, LUT–4, LUT–6 (most common), LUT–8 (rare))
- Logic block (CLB) connected by programmable interconnect
  - Programming configures MUXes
LUT Mapping

- Process assigning logical functions to LUTs
- Efficiently mapping is important, but placement is even more critical
- Considering the Boolean function: $AB + BC + AC + D$
  - This maps to six LUT–2s
  - Or, a single LUT–4
ASIC Architecture

- Transistors at this level
- Standard cells are pre-made logic cells
  - All have the same height
  - Can have different widths
- The die area is composed of rows and standard cells are placed in row efficiently to make routing easier.
Shift Registers

- Daisy-chained registers
- Each register can be multibit
- A fundamental hardware structure
  - Common in serial RX and TX interfaces
  - LFSR (pseudo-random number generator)
- Shift registers with 1-bit width can be implemented with a single multibit register

Borrowed from: http://tinyurl.com/ukctskza
Testbenches (Quickly)

- Testbenches are Verilog modules which instantiate your DUT to drive inputs and verify outputs
  - Testbenches have no inputs or outputs
- File name should match module name
- Use delays (#) or timing event @ (posedge clk) to sequence events
- Things to have:
  1. Simulated clock (should be reg)
  2. A reset signal (should be reg)
  3. Initial block (always assert reset first), several test cases
  4. Instantiated DUT
  5. Process to verify outputs or print to console ($display)
Problems

• First, we will work on each problem alone for 5ish minutes
• Then break into groups of 3–4 and discuss your solutions for 5 minutes
• I’ll have one the groups share their answer and we’ll go over as a class
Problem 1:  LUTs and Functions

Lookup Tables (LUTs) are the fundamental building blocks of FPGA architectures. A LUT-\( N \) can implement any \( N \) input logic function (ex. a LUT-4 can implement any logic function with four logical inputs). On an FPGA, LUTs can be connected together through special routing to implement functions with even more inputs. Consider the following arrangement of five LUT-4 blocks:

How many logic functions can this chain of LUT-4’s implement?
Problem 1: Solution

Solution:

Let’s break it down. A function is a logical expression. How many functions can single bit implement? 4. Bit A can be either 0 or 1, and can map to 0 or 1.

1. $A \rightarrow 0$
2. $A \rightarrow 1$
3. $\bar{A} \rightarrow 0$
4. $\bar{A} \rightarrow 1$

From this we can establish a general rule: $2^n$ for a LUT-N. Therefore, each LUT-4 can implement $2^4 = 2^{16} = 65536$ functions. Since the inputs to the last LUT-4 is another LUT-4, each bit represents $2^4$ functions. Therefore, the total number of functions is $(2^{16})^5$. 

Berkeley
UNIVERSITY OF CALIFORNIA
Problem 2: Self Starting Ring Counter

A ring counter is a special counter composed of flip-flops daisy-chained together to form a shift register and the output of the last flip-flop is connected to the output of the first. Below is a variant of self starting ring counter. (Note: the last flip-flops output is not the input to the first, but it's close enough to call it a ring counter :-)). It is self starting because there is no reset. The counter will reset itself! The counter is read out such that the first register is the LSb of the count value.

1. How does this self-initialize itself?
2. What type of counter is this?
3. Assume the register are initialized as 0, 1, 0, 1, 0. Create a table showing clock cycle, input to the chain, value of each register. Provide a waveform diagram for the first 10 cycles after initialization.
4. How does the circuit behave in steady state (steady state means after hundreds of cycle)?
5. This behavior can be create using a regular incrementing counter and a decoder. Write Verilog for this implementation.
Solution:
The NOR gate takes as input all *except* the last flip-flop in the shift register. Therefore, the NOR output is 1 only if all of the first four registers are zero. Since, the NOR output is the input of the shift register, wherever the previous condition occurs the shift register is initialize with a 1. Note this occurs regardless of the value in the last flip-flop.

1. The input to the shift register is 1 when the first four flip-flops are zero.

2. A one-hot counter

3.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>NOR Output</th>
<th>Reg0</th>
<th>Reg1</th>
<th>Reg2</th>
<th>Reg3</th>
<th>Reg4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

4. It counts in powers of two: 1, 2, 4, 8, 16, 1, 2, 4, ...
Problem 2: Solution

5. module ring_cnt(
    input clk,
    output reg [4:0] cnt;

    integer j; // Used for bit reversal

    wire [4:0] reg_in; // Input to register
    reg [5:0] reg_out; // Output of register

    // Create shift register
    genvar i;
    generate
        for (i=0; i<5; i=i+1) begin
            REGISTER regX ( .clk(clk),
                .d([reg_in]),
                .q(reg_out[i]));
        end
    endgenerate

    // Procedural Assignment
    always @(*) begin
        for (j=0; j<5; j=j+1) begin
            cnt[j] = reg_out[4-j];
        end
    end

    // Signal Assignment
    assign reg_in = {reg_out[3:0], ~reg_out[3:0]};
endmodule
Problem 3:

Problem 2: Decoder-Based Multiplexer

(a) Design a 4-to-1 multiplexer using one of the decoders you designed above. The select signals must be input to the decoder and must not be used anywhere else. Provide an exhaustive test.

(b) What could be a potential benefit of using this decoder-based multiplexer against the following design:

*borrowed from EECS 151/251A Spring 2023 hw3
Problem 3: Solution

Solution:

(a) Diagram:
Problem 3: Solution

Design:

`include "decoder.v"

module multiplexer(
    input [1:0] s,
    input [3:0] in,
    output    out
);

wire [3:0]        x;
wire              t0, t1, t2, t3;

decoder1 dc1(.s(s), .x(x));

and(t0, in[0], x[0]);
and(t1, in[1], x[1]);
and(t2, in[2], x[2]);
and(t3, in[3], x[3]);
or(out, t0, t1, t2, t3);

endmodule
module multiplexer_tb;
    reg [1:0] s;
    reg [3:0] in;
    reg expected;
    wire out;

    // loop variables
    integer i, j;

    // instantiate dut
    multiplexer mux1(s, in[0], out);

    // expected outputs
    always @(s) begin
        case (s)
            2'ba0: expected = in[0];
            2'ba1: expected = in[1];
            2'bla0: expected = in[2];
            2'bla1: expected = in[3];
            default: expected = 1'bx;
        endcase
    end

    // begin test
    initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
        for(j = 0; j < 16; j = j + 1) begin
            in = j;
            for(i = 0; i < 4; i = i + 1) begin
                s = i;
                #1;
                $display("s: %b, in: %b, out: %b, expected: %b", s, in, out, expected);
            end
            if(out !== expected) begin
                $display("FAILED, expected %b, got %b", expected, out);
                $finish();
            end
        end
        $display("ALL TESTS PASSED!");
        $finish();
    end
endmodule