Eric Chuong

Rebecca Grob

ECE425L

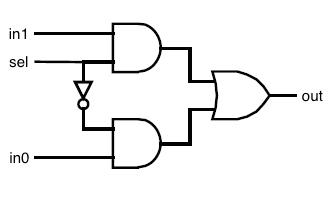
Introduction and Objective

There are three problems to this lab and for each problem we create a different product. The objective is to create a 16 bit 2-to-1 MUX, 3-to-8 Decoder, 2’s complement adder, and a 4-bit shift register. The 4-bit shift register must be able to perform load, hold, right shift, and left shift functions.

Problem 1

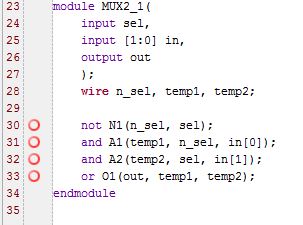
**a.** 16 bit 2-to-1 MUX

2-to-1 MUX logic diagram

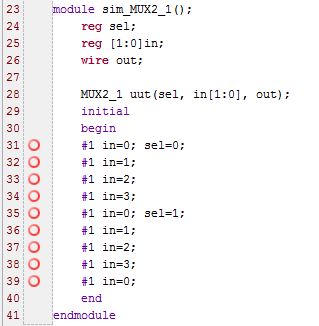


Module Code

2-to-1 MUX



Test Bench Code



Simulation Results



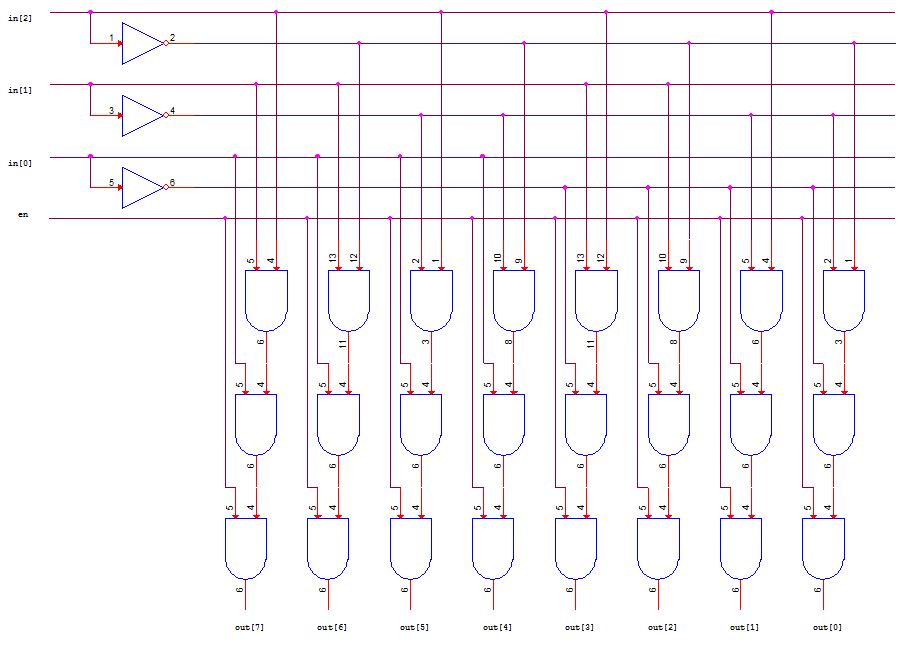
Problem Analysis

We weren’t sure how to make the 2-to-1 MUX 16 bits but we know that it would have to use a normal 2-to-1 MUX so we created that. The coding for the MUX is fairly simple. We have three inputs total which are sel, in1, and in0, and there is only one output called out. We wire three things to simplify our code which are n\_sel, temp1, and temp2. N\_sel represents the inverse of sel. Temp1 is the output of the AND gate with inputs n\_sel and in0 whereas temp2 is the output of the AND gate with inputs sel and in1.Then to get the output of the MUX we can OR temp1 and temp2. For our test bench, we wanted to show every output of the MUX so we began with all values at 0. Every 1ns, the values will change. In the end, we will have displayed every combination of the following truth table.

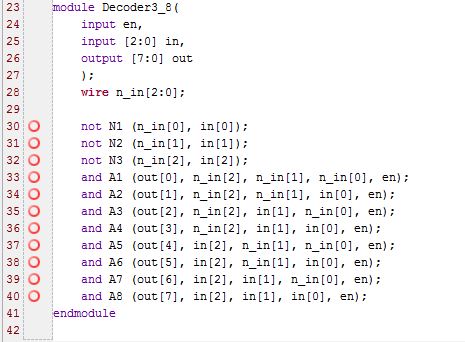
|  |  |  |  |
| --- | --- | --- | --- |
| s | in1 | in0 | out |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

**b.** 3-to-8 Decoder

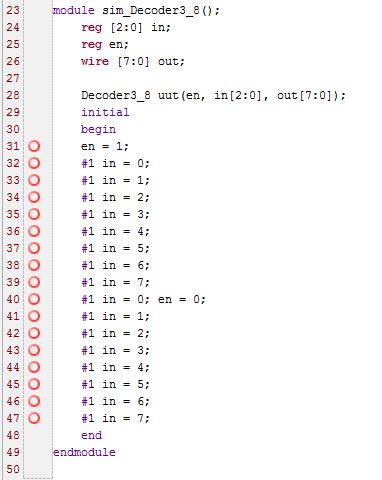
3-to-8 Decoder schematic



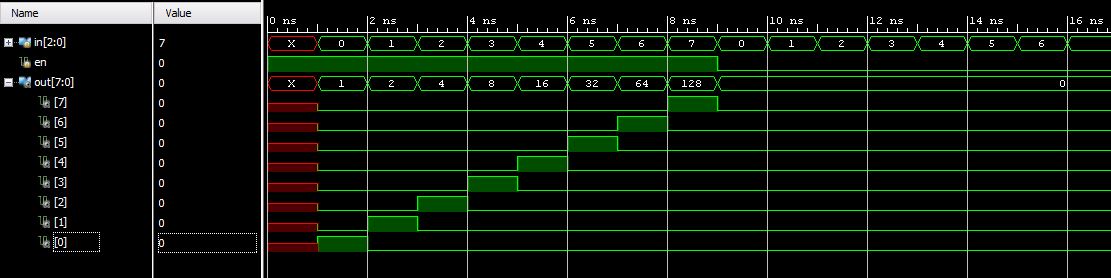
Module Code



Test Bench Code



Simulation Results



Problem Analysis

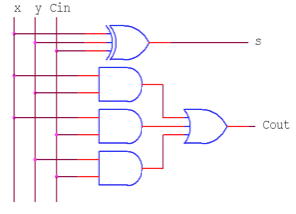
For the 3-to-8 decoder, we have 4 total inputs including the enable and 8 total outputs. The decoder will generate an output if the enable is on, however if it is off then the decoder will only output 0. The idea is that the decoder will read the 3-bit binary value of the input, in, and output 2^(in) in 8-bits. For example, if we have input 011, or 3, then we will get an output of 2^3 = 8, or 00001000. To accomplish this we use various combinations of NOT and AND logic gates which you can see from the block diagram. The final step is to incorporate the enable function so we just include it in each AND gate. For our test bench we wanted to show every possible combination of the 3-to-8 decoder. We first show what happens when the enable is turned on, so en = 1. We start with in = 000 and we increase by 1 every 1ns. When we hit in=7, which is the maximum value because we are restricted to 3-bits, we toggle enable to 0. We repeat the values for in starting again at 000. You can see from the simulation results that because en=0, the decoder will only generate an output of 0.

Problem 2

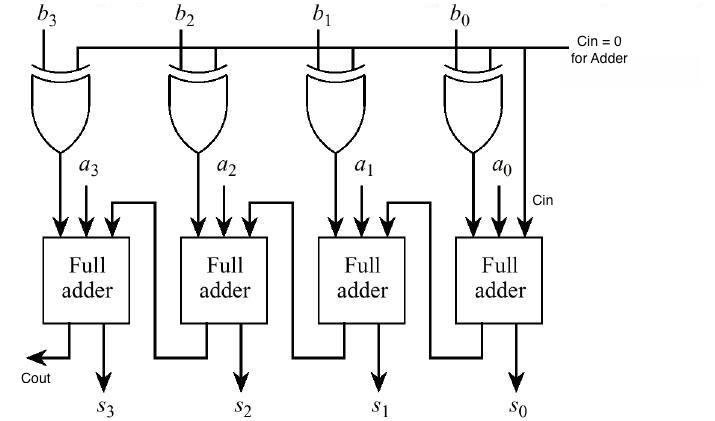
2s Complement Adder

Schematics

Full Adder

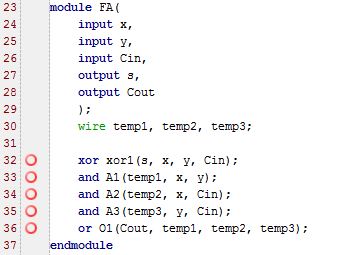


2s Complement Adder

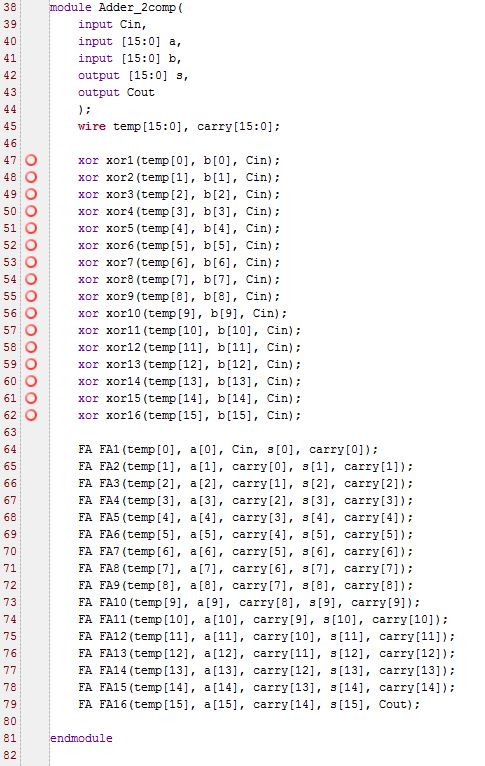


Module Codes

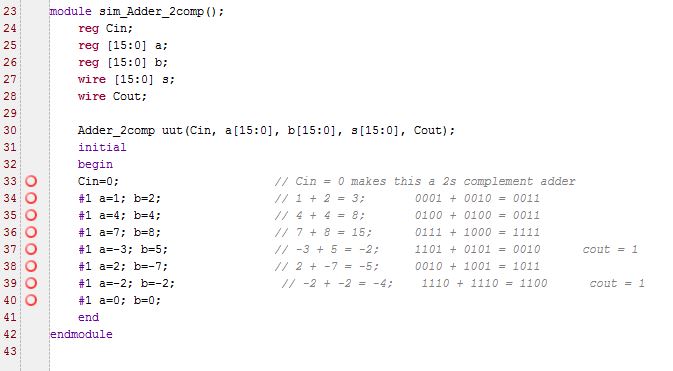
Full Adder



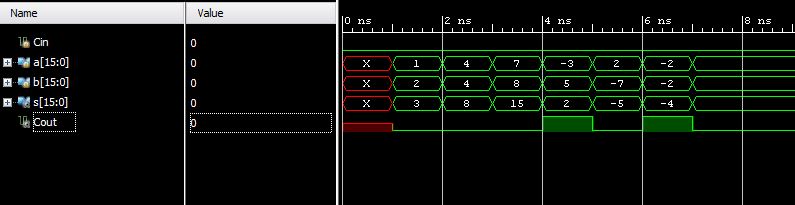
2s Complement Adder



Test Bench



Simulation Results

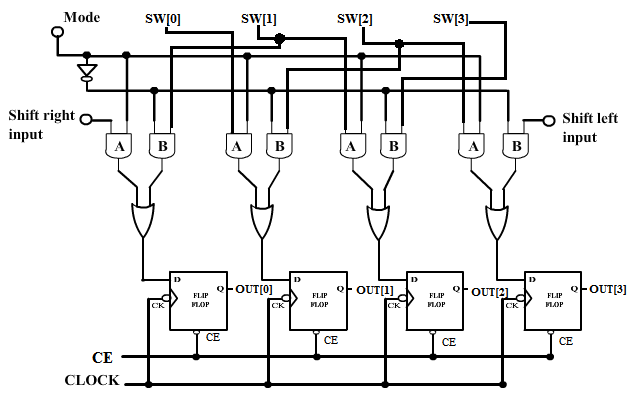


Problem Analysis

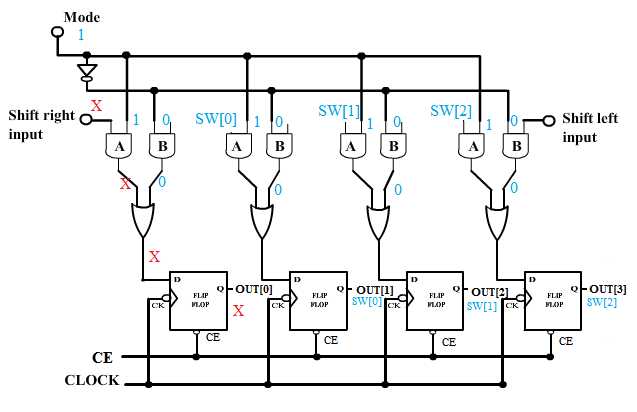
For the two’s complement adder, we need to implement the full adder. Have two three inputs which are the Cin and the two 16-bit numbers that we are adding. Our two outputs are the Cout and the sum. The sum will also be 16-bits. Notice that we must call the Full Adder for each bit that we add. Also, for the purpose of this adder, we are only using Cin = 0. For the test bench we began with showing some simple addition of two positive numbers. Then we show some negative values and the waveform demonstrates the sum perfectly. In addition, it shows the Cout if there is one.

Problem 3

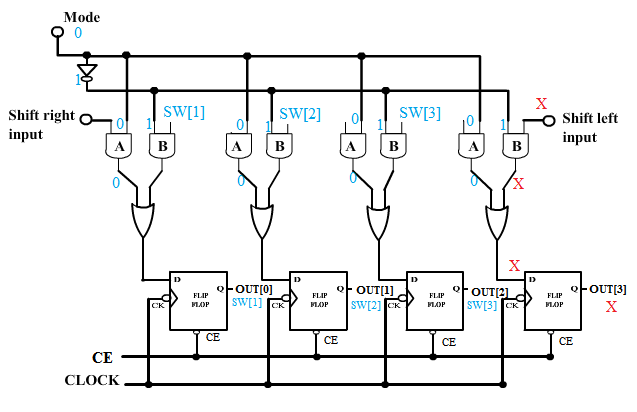
*Schematic*



If the Mode is equal to 1, it is a right shift register.



If the Mode is equal to 0, it is a left shift register.



*Verilog Code*

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 18:10:58 04/11/2016

// Design Name:

// Module Name: ShiftRegisterSourceFileV2p1

// Project Name:

// Target Devices:

// Tool versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module ShiftRegisterSourceFileV3p2(CE,SLI, SRI, clk, sw, dataout);

input [1:0]CE;

/\* 00: hold 0

01: load 1

10: Lshift 2

11: Rshift 3 \*/

//input mode; //left or right shift

input SLI; //SHIFT LEFT INPUT

input SRI; //SHIFT RIGHT INPUT

input clk; //CLOCK

input [3:0]sw; //input the swithches (4-bits)

output [3:0]dataout; //Output

wire [3:0]temp;

/\*reg [1:0]Tce;\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//wire [3:0]Tempsw;

reg [3:0] Tsw;

//Tsw = 0;

D\_Flip\_Flop DFF\_zero (clk,CE,0,Tsw);

D\_Flip\_Flop DFF\_Tce (clk,CE,CE,Tce);

always @(clk)

case(Tce)

0: begin D\_Flip\_Flop DFF\_H (clk,CE,Tsw,dataout); end//HOLD

1: begin

D\_Flip\_Flop DFF\_L1 (clk,CE,sw,Tsw);

D\_Flip\_Flop DFF\_L2 (clk,CE,Tsw,dataout);

end//LOAD

default://SHIFT ANDANDXOR(clk, CE, A1, A2, B1, B2, Q);

begin

ANDANDXOR aax1(clk, CE, SLI, CE[0], ~CE[0], Tsw[1], temp[0]);

D\_Flip\_Flop DFF1 (clk,CE,temp[0],dataout[0]);

ANDANDXOR aax2(clk, CE, Tsw[0], CE[0], ~CE[0], Tsw[2], temp[1]);

D\_Flip\_Flop DFF2 (clk,CE,temp[1],dataout[1]);

ANDANDXOR aax3(clk, CE, Tsw[1], CE[0], ~CE[0], Tsw[3], temp[2]);

D\_Flip\_Flop DFF3 (clk,CE,temp[2],dataout[2]);

ANDANDXOR aax4(clk, CE, Tsw[2], CE[0], ~CE[0], SRI, temp[3]);

D\_Flip\_Flop DFF4 (clk,CE,temp[3],dataout[3]);

end

endcase\*/\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

begin

ANDANDXOR aax1(clk, CE, SLI, CE[0], ~CE[0], sw[1], temp[0]);

D\_Flip\_Flop DFF1 (clk,CE,temp[0],dataout[0]);

ANDANDXOR aax2(clk, CE, sw[0], CE[0], ~CE[0], sw[2], temp[1]);

D\_Flip\_Flop DFF2 (clk,CE,temp[1],dataout[1]);

ANDANDXOR aax3(clk, CE, sw[1], CE[0], ~CE[0], sw[3], temp[2]);

D\_Flip\_Flop DFF3 (clk,CE,temp[2],dataout[2]);

ANDANDXOR aax4(clk, CE, sw[2], CE[0], ~CE[0], SRI, temp[3]);

D\_Flip\_Flop DFF4 (clk,CE,temp[3],dataout[3]);

end

endmodule

---------------------------------------------------------------------------------------------------

`timescale 1ns / 1ps

module ANDANDXOR(clk, CE, A1, A2, B1, B2, Dtemp);

input clk; //clock

input CE; //enable

input A1;

input A2;

input B1;

input B2;

//output Q; //output

output Dtemp; //output

wire [0:1]temp;

//wire Dtemp;

and

andA(temp[0],A1,A2), //AND A

andB(temp[1],B1,B2); //AND B

//XOR A B

or orAB(Dtemp,temp[0],temp[1]);

//D\_Flip\_Flop DFF(clk,CE,Dtemp,Q);

endmodule

---------------------------------------------------------------------------------------------------

`timescale 1ns / 1ps

module D\_Flip\_Flop(CLK,CE,D,Q);

input CLK;

input CE;

input D;

output reg Q = 0;

always @(posedge CLK)

if (CE) Q <= D;

endmodule

---------------------------------------------------------------------------------------------------

*Test Bench*

`timescale 1ns / 1ps

module BenchTest\_SRV3t1();

// Inputs

reg [1:0]CE;

reg SLI;

reg SRI;

reg clk;

reg [3:0] sw;

// Outputs

wire [3:0] dataout;

/\* 00: hold

01: load

10: Lshift

11: Rshift\*/

// Instantiate the Unit Under Test (UUT)

ShiftRegisterSourceFileV3p2 uut (

.CE(CE),

.SLI(SLI),

.SRI(SRI),

.clk(clk),

.sw(sw),

.dataout(dataout)

);

initial begin

// Initialize Inputs

clk = 0;

repeat(100)

#1 clk = ~clk;

end

initial begin

// Initialize Inputs

CE = 0;

SLI = 0;

SRI = 0;

sw = 0;

// Wait 100 ns for global reset to finish

#10; //the delay

CE = 1;//2'b01;//begin

sw = 7;//2'b0111;

#10;

sw = dataout;

SRI = 1;

#20;

sw = dataout;

SRI = 0;

#20

sw = dataout;

CE = 0;

#20;

SLI = 1;

sw = dataout;

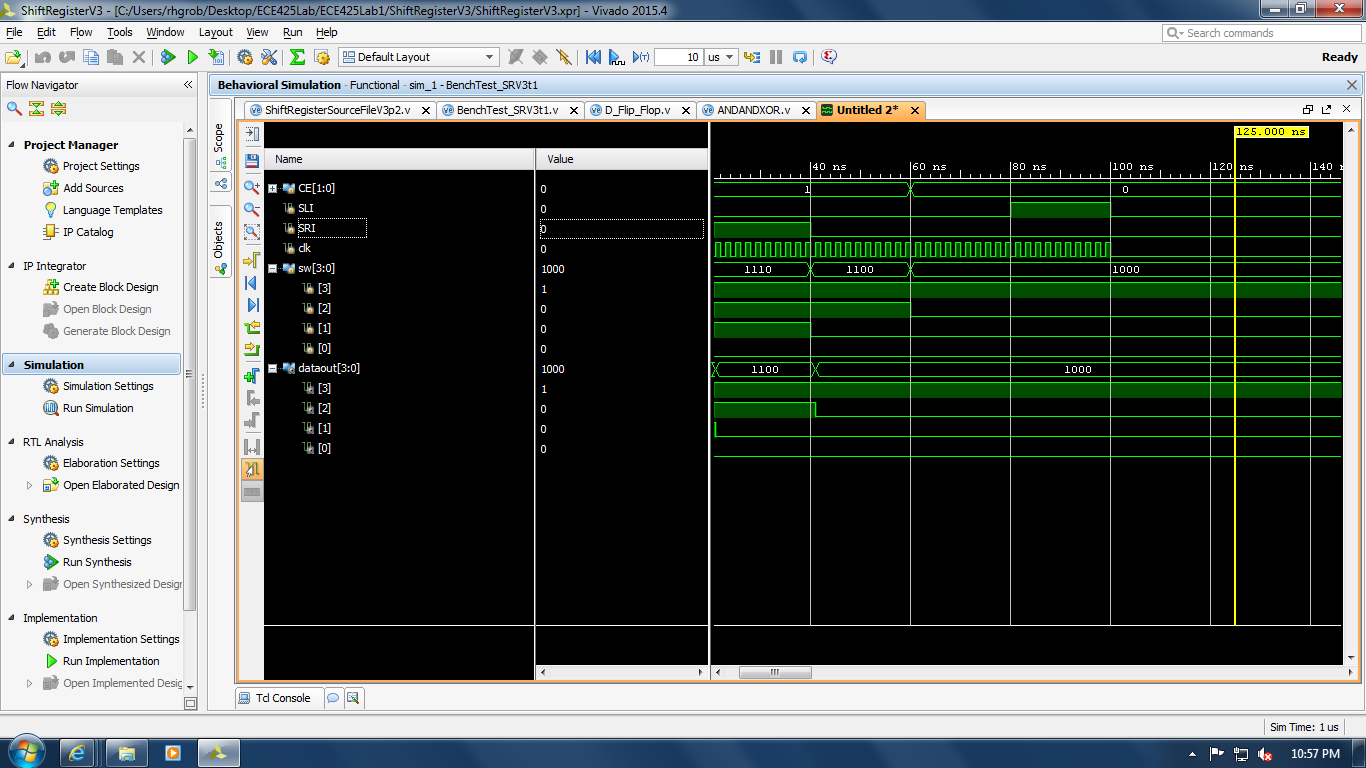
#20;

SLI = 0;

sw = dataout;

end

endmodule

*Simulator/Waveform*

*Explanation of test vectors*

The left and right shift registers are fully functional and may be changed with the last bit value of CE. The conditional statements provided some difficulty in implementing the hold and load functions (00 and 01 respectively). However, the logic functions were successfully implemented in the test bench.

Discussion of Results

Overall, we felt the lab went pretty well. There were a little bit of issues that we ran into along the way but we were able to get the main idea of the lab. Most of our work is accurate and it can be proved by looking at the waveforms generated.