Colt Thomas

March 4, 2014

Lab 7 – Register File

Procedure:

We are to first implement a 1-bit register using Verilog. We then must test the functionality with a .tcl file and then simulate it. Below is the code in Verilog, the .tcl file and then the waveform simulation.

Verilog code-

Below are the modules that I used for a one bit register:

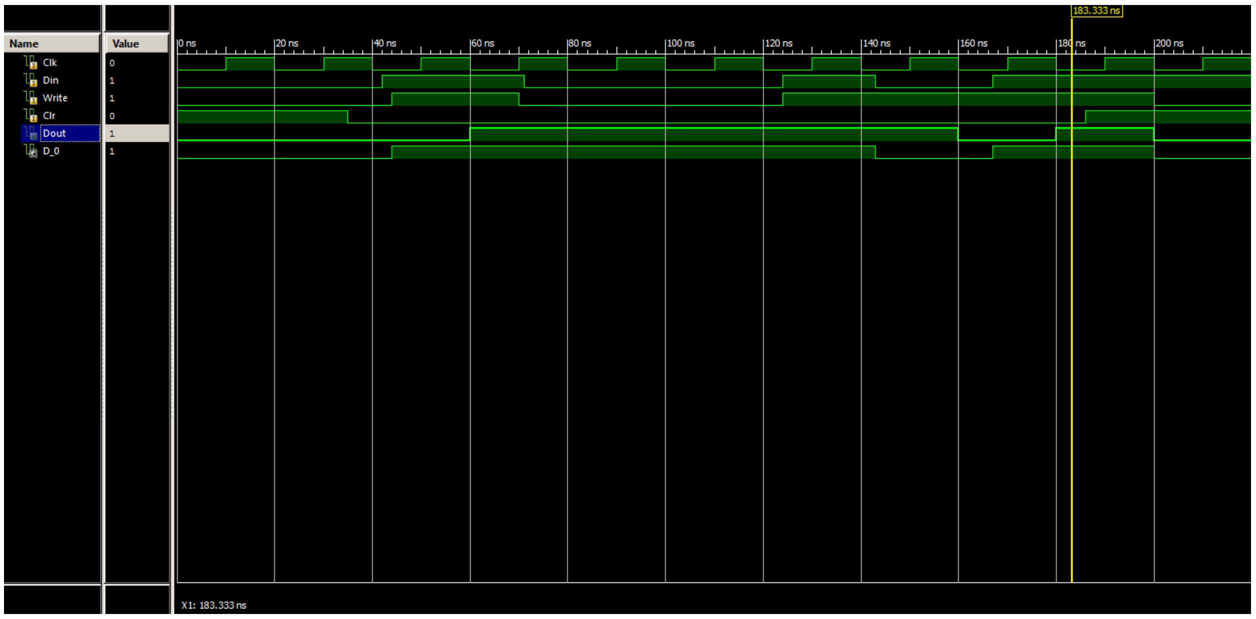
```
module mux21(
   input a,
   input b,
   input sel,
   output q
   );

        assign q = sel ? b:a;
endmodule


module FlipFlopD(
   input clk,
   input clr,
   input d,
   output reg q
   );
        always @(negedge clk)  //this is a falling edge FlipFlopD

        if(clr) q <=0 ;
        else q <= d;

endmodule



module RegisterOneBit(
   input Din,
   input Write,
   input Clk,
   input Clr,
   output Dout
   );
            wire D_0;
        mux21 mx_1(Dout , Din, Write , D_0);
        FlipFlopD fl_d(Clk, Clr, D_0 , Dout);
endmodule
```

.TCL file-

Here is the .tcl file that I used to test out the 1-bit register, with the simulation below:

```
wave add / -radix hex
#runs the clock on 10ns intervals
isim force add Clk 0 -time 0 -value 1 -time 10 ns -repeat 20ns
#forces the input data
isim force add Clr 1 -time 0  -value 0 -time 35ns -value 1 -time 186ns
isim force add Din 0 -time 0 -value 1 -time 42ns -value 0 -time 71ns -value 1 -time 124ns -value 0 -time
143ns -value 1 -time 167ns
isim force add Write 0 -time 0  -value 1 -time 44ns -value 0 -time 70ns -value 1 -time 124ns -value 0 -
time 200ns
run 220ns
```

Simulation-



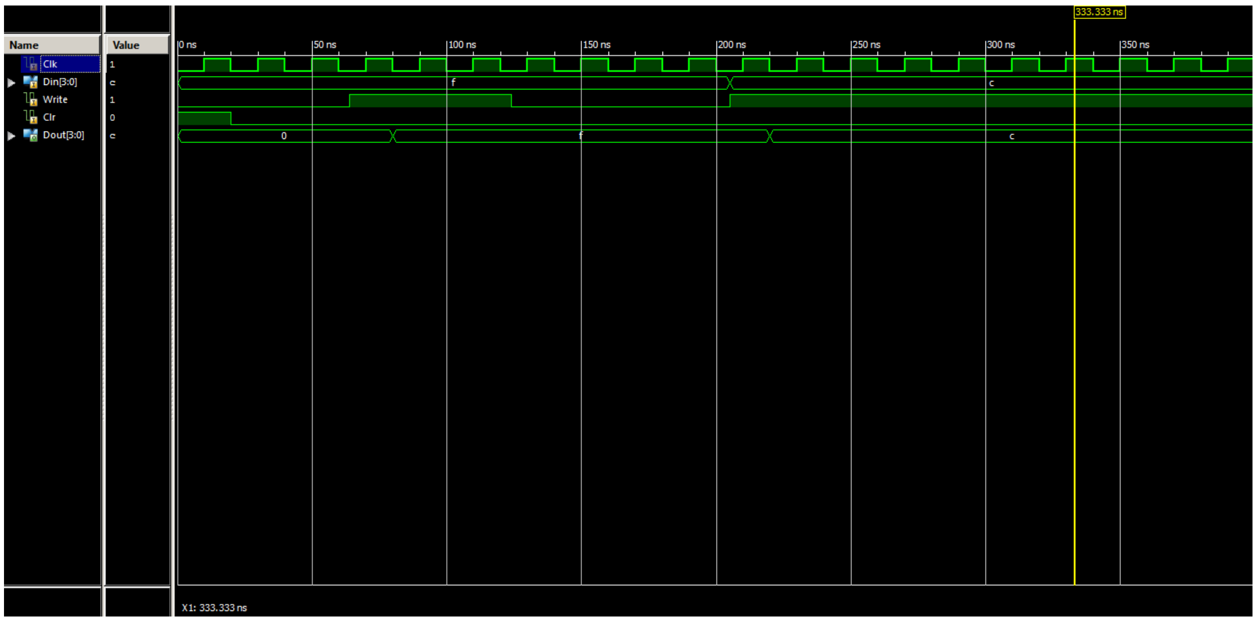
Next we were to implement a four-bit register using the one bit registers. We then had to simulate it like we did to the one bit register. Below is the Verilog code:

4-Bit Register Verilog

```
module RegisterFourBit(
  input [3:0] Din,
  input Write,
  input Clr,
  input Clk,
  output [3:0] Dout
  );


        RegisterOneBit R0(Din[0], Write, Clk, Clr, Dout[0]);
        RegisterOneBit R1(Din[1], Write, Clk, Clr, Dout[1]);
        RegisterOneBit R2(Din[2], Write, Clk, Clr, Dout[2]);
        RegisterOneBit R3(Din[3], Write, Clk, Clr, Dout[3]);
Endmodule
```

4-bit register simulation

We were instructed to implement a simulation where we forced our 4 data inputs to a ‘1’ and not change them. Then we were to verify that our bits weren’t getting swapped in the register by switching the 4 data inputs to 1100.



4-bit register .tcl file

```
wave add / -radix hex

#runs the clock on 10ns intervals

isim force add Clk 0 -time 0 -value 1 -time 10 ns -repeat 20ns

isim force add Clr 1 -time 0  -value 0 -time 20ns
isim force add Din 1111 -time 0 -value 1100 -time 205ns
isim force add Write 0 -time 0  -value 1 -time 64ns -value 0 -time 124ns -value 1 -time 205ns

run 400ns
```

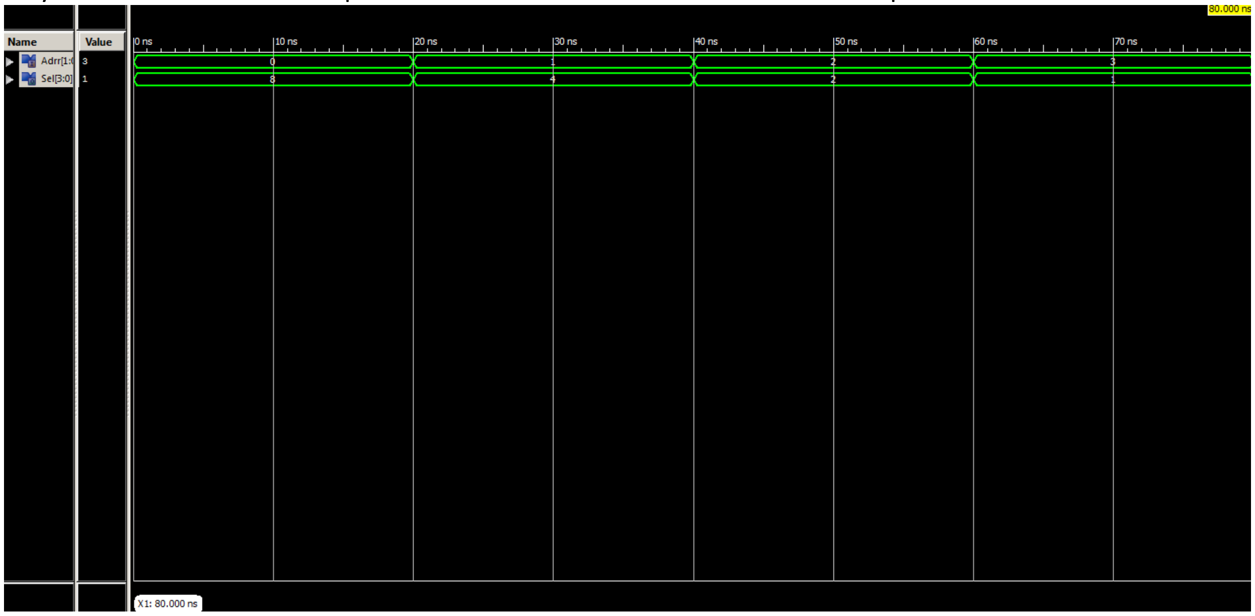
2:4 Decoder

Now we get to implement a 2:4 decoder in Verilog and then make a simulation of it. Below is the Verilog:

```
module Decoder24(
   input [1:0] Adrr,
   output [3:0] Sel
   );

        assign Sel =        (Adrr==2'b00) ?4'b1000:
                            (Adrr==2'b01) ?4'b0100:
                            (Adrr==2'b10) ?4'b0010:
                            4'b0001;
endmodule
```

2:4 Decoder –Simulation

As you can see that all the inputs turn on one of each of the bits of the output.



2:4 Decoder - .tcl

```
wave add / -radix hex

#shows all possible outputs of the address input
isim force add Adrr 00 -time 0  -value 01 -time 20ns -value 10 -time 40ns -value 11 -time 60ns

run 80ns
```

FourWordRegister – Verilog

```
module FourWordRegister(
   input [3:0] Din,
   input Write,
   input [1:0] Adrr,
   input Clk,
            input Clr,
   input [15:0] Dout
   );
               wire[3:0] reg_3 , reg_2 , reg_1 , reg_0 , Sel;
               wire write_3 , write_2 , write_1 , write_0;

               //this is our decoder which selects which 4-bit reg to write to
               Decoder24 Decode(Adrr , Sel);
```

```
                    //this is the logic to select the write operation
                    and(write_3 , Sel[3] , Write);
                    and(write_2 , Sel[2] , Write);
                    and(write_1 , Sel[1] , Write);
                    and(write_0 , Sel[0] , Write);

                    //these are the 4-bit registers
                    RegisterFourBit Reg3(Din, write_3, Clr, Clk, reg_3);
                    RegisterFourBit Reg2(Din, write_2, Clr, Clk, reg_2);
                    RegisterFourBit Reg1(Din, write_1, Clr, Clk, reg_1);
                    RegisterFourBit Reg0(Din, write_0, Clr, Clk, reg_0);

                    //we concatenate the outputs into a 16 bit output
                    assign Dout = {reg_3 , reg_2 , reg_1 , reg_0};


endmodule
```

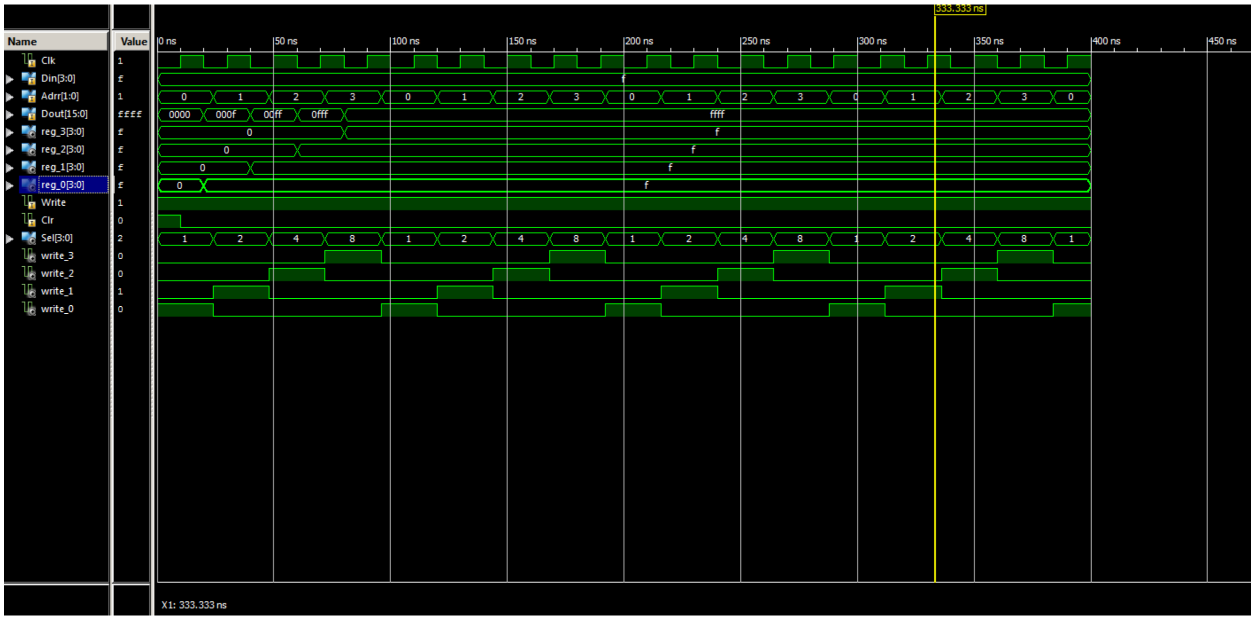
FourWordRegister - .tcl

I made a file that would just test to see if the word loaded to the appropriate register, and that the output concatenated accordingly

```
wave add / -radix hex

#this is the clock timer for the simulation:
isim force add Clk 0 -time 0 -value 1 -time 10 ns -repeat 20ns

#we will always have the input to be 1111 for Din
isim force add Din 1111 -time 0

#now we test the circuit to see if '1111' can be written to the correct register
isim force add Adrr 00 -time 0  -value 01 -time 24ns -value 10 -time 48ns -value 11 -time 72ns -repeat 96ns
isim force add Clr 1 - time 0 -value 0 -time 10ns
isim force add Write 1 - time 0


run 400ns
```

FourWordRegister - simulation

We see that the register store the appropriate words depending on the address. Note that when the address is 0, hex value ‘f’ is stored in bit 1, and so on to bit 4



4:1 4-bit MUX – Verilog

This is the verilog code for my 4:1 4-bit MUX that will be used to select the output of any particular register:

```
module mux16to4(RegFile_out , Adrr , Reg_3 ,Reg_2 ,Reg_1 ,Reg_0 );
        parameter WID = 4;
  input [WID-1:0] Reg_3;
  input [WID-1:0] Reg_2;
  input [WID-1:0] Reg_1;
  input [WID-1:0] Reg_0;
  input [1:0] Adrr;
  input [WID-1:0] RegFile_out;

        assign RegFile_out =     (Adrr==2'b11)?Reg_3:
                                                        (Adrr==2'b10)?Reg_2:
                                                        (Adrr==2'b01)?Reg_1:
                                                        Reg_0;

endmodule
```

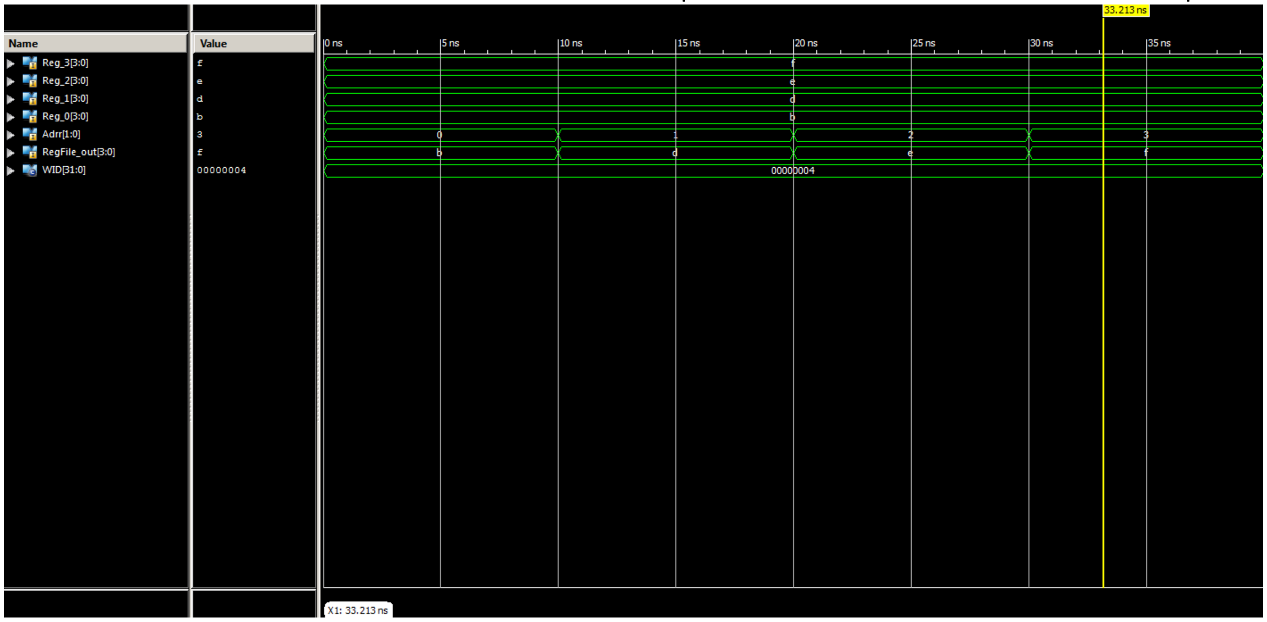
4:1 4-bit MUX - .tcl

This is the .tcl file I used for the 4:1 4-bit MUX:

```
wave add / -radix hex

#just a MUX simulation.  All the inputs are different hex values (represented as binary below)
isim force add Adrr 00 -time 0 -value 01 -time 10ns -value 10 -time 20ns -value 11 -time 30ns
isim force add Reg_3 1111 -time 0
isim force add Reg_2 1110 -time 0
isim force add Reg_1 1101 -time 0
isim force add Reg_0 1011 -time 0

run 40ns
```

4:1 4-bit MUX – simulation

You will notice below that the MUX selects the correct inputs based on the values of the Address input:



Register File Verilog Code:

This code implements the Register file:

```
module RegisterFile(
   output [3:0] Dout,
   input [3:0] Din,
   input Clk,
   input Write,
   input [1:0] Adrr_Read,
            input [1:0] Adrr_Write
   );
         wire [15:0] Reg_Out;
```

```
        wire [3:0] Reg_3, Reg_2, Reg_1, Reg_0;
        FourWordRegister Reg(Din , Write , Adrr_Write, Clk, Clr, Reg_Out);

        assign Reg_3 = {Reg_Out[15], Reg_Out[14], Reg_Out[13], Reg_Out[12]};
        assign Reg_2 = {Reg_Out[11], Reg_Out[10], Reg_Out[9], Reg_Out[8]};
        assign Reg_1 = {Reg_Out[7], Reg_Out[6], Reg_Out[5], Reg_Out[4]};
        assign Reg_0 = {Reg_Out[3], Reg_Out[2], Reg_Out[1], Reg_Out[0]};
        mux16to4(Dout , Adrr_Read , Reg_3, Reg_2, Reg_1, Reg_0);

endmodule
```

TestBench Verilog Code:

Below is the TestBench code that I had used so that way we can take it and implement it on the Xilinx board:

```
module TestBench(
        input [3:0] Switch_Input,
        output [3:0] LED_Ind,
        input [1:0] Adrr_Write,
        input [1:0] Adrr_Read,
        output[3:0] Reg_Out,
        input Write,
        input Clr,
        input Clk
   );

        RegisterFile Main(Reg_Out, Switch_Input , Clk , Clr, Write , Adrr_Read , Adrr_Write);
        buf(LED_Ind, Switch_Input);

endmodule
```

Testbench UCF file:

Below is the UCF file lines that I implemented:

```
## clock pin for Nexys 2 Board
NET Clk   LOC = "B8"; # Bank = 0, Pin name = IP_L13P_0/GCLK8, Type = GCLK, Sch name = GCLK0

## Leds
```

```
NET Reg_Out[0]  LOC = "J14"; # Bank = 1, Pin name = IO_L14N_1/A3/RHCLK7, Type = RHCLK/DUAL, Sch name = JD10/LD0
NET Reg_Out[1]  LOC = "J15"; # Bank = 1, Pin name = IO_L14P_1/A4/RHCLK6, Type = RHCLK/DUAL, Sch name = JD9/LD1
NET Reg_Out[2]  LOC = "K15"; # Bank = 1, Pin name = IO_L12P_1/A8/RHCLK2, Type = RHCLK/DUAL, Sch name = JD8/LD2
NET Reg_Out[3]  LOC = "K14"; # Bank = 1, Pin name = IO_L12N_1/A7/RHCLK3/TRDY1, Type = RHCLK/DUAL, Sch name = JD7/LD3
NET LED_Ind[0]  LOC = "E17"; # Bank = 1, Pin name = IO, Type = I/O, Sch name = LD4
NET LED_Ind[1]  LOC = "P15"; # Bank = 1, Pin name = IO, Type = I/O, Sch name = LD5
NET LED_Ind[2]  LOC = "F4";  # Bank = 3, Pin name = IO, Type = I/O, Sch name = LD6
NET LED_Ind[3]  LOC = "R4";  # Bank = 3, Pin name = IO/VREF_3, Type = VREF, Sch name = LD7

## Switches
NET Adrr_Read[0] LOC = "G18"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW0
NET Adrr_Read[1] LOC = "H18"; # Bank = 1, Pin name = IP/VREF_1, Type = VREF, Sch name = SW1
NET Adrr_Write[0] LOC = "K18"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW2
NET Adrr_Write[1] LOC = "K17"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW3
NET Switch_Input[0] LOC = "L14"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW4
NET Switch_Input[1] LOC = "L13"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW5
NET Switch_Input[2] LOC = "N17"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW6
NET Switch_Input[3] LOC = "R17"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = SW7

## Buttons
NET Clr LOC = "B18"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = BTN0
#NET "btn<1>" LOC = "D18"; # Bank = 1, Pin name = IP/VREF_1, Type = VREF, Sch name = BTN1
#NET "btn<2>" LOC = "E18"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = BTN2
NET Write LOC = "H13"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name = BTN3
```

Anomalies

This was one of the smoothest labs that I have had so far. The only problem that I had was just getting my inputs and outputs lined up for the .bit file and .ucf file, but it was only because I was near the end of the lab and ready to pass off. Once I took the time to read the last bit, things worked on the Xilinx board. It was rather cool! I enjoyed this lab. This gives me a lot of ideas of things to do on my own time. It did help me to do the simulations and make sure that everything worked as I progressed through the lab. Had I not done that, I would have spent hours debugging rather than the 10 minutes that I spent fixing my .ucf file.