UNIVERSITY OF CALIFORNIA IRVINE
COURSE: INTRODUCTION TO DIGITAL LOGIC LAB (31L) FALL 2015

Lab 3 Report - Group 33

Kelvin Phan
51197373

Patrick Skoury
75202200

Aaron Liao
90811748

Shawn Howen
60029119

Andrew Mehta
44592437

November 22, 2015

1 CREATING THE REGISTER

1.1 ARCHITECTURE

Our design for the register implemented the behavioral style of architecture used for sequential coding. Before defining our ports, we set a GENERIC called "NBIT," which was an integer that defaulted as 32, but could be changed depending on the desired number of bits to be stored in the register. Then, we defined our inputs and output with reference to the ENTITY given for the assignment. Within the architecture, we used the behavioral model of coding, which included a PROCESS statement. Before our PROCESS, we defined a signal called "dhold," which was an STD_LOGIC_VECTOR of the same length as "din." The signal was set as a vector of zeros, and is used in the PROCESS to store the previous value of "dout," in case write enable is active low and we don't want to read from "din". Our PROCESS statement had a sensitivity list that depended on "clk" and "rst_a." The two main IF statements checked whether the asynchronous reset was '1' or if the "clk" was on a rising edge. Within the "clk" statement, IF statements checked for the synchronous reset, whether increment was '0' and write enable were '1,' whether "inc" and write enable were both '1,' or if just "inc" was '1.' If "we" was '0' and we wanted to increment, "dout" was set as "dhold" plus one, rather than "din" plus one. This

differs from the case when both "we" and "inc" are '1,' which reads from "din," and disregards the previous value of "dout." If none of the IF statements are true, "dout" is given "dhold," which means there was no change to the output value. We fed in six random inputs to "din" after 14 rounds. The test bench section of the report specifies how we tested our components in further detail.

2 Creating the Register File

2.1 Architecture

Our design for the register file depended heavily on defining our own array TYPE. The first step was initializing two GENERIC integers, "NBIT" and "NSEL." "NBIT" served the same purpose as it did in the single register. "NSEL" was used to define how many registers we would have in our register file. The number of registers was equal to "2**NSEL." After GENERIC, we defined all our input and output ports. Within the architecture, a SUBTYPE called "selRange" was created as the integer range from '0' to "INTEGER'HIGH." Three signals were then created under "selRange," which would be used as indexes for reading and writing data. "Vector" was defined as a subtype of STD_LOGIC_VECTOR with the same range of "NBIT." Then, a TYPE we called "matrix" was an ARRAY with the length of "2**NSEL," containing the SUBTYPE "vector." The signal "regN" was of the TYPE "matrix" and was initialized to have all zeros. We essentially created a matrix of "2**NSEL" registers, which stored vectors with the length of "NBIT." After these declarations, we assigned our indexes by converting "raddr_1," "raddr_2," and "waddr" to integers. The PROCESS statement only depended on the "clk" signal. Within the PROCESS, we first checked whether "rst_s" was '1.' If it was, a FOR LOOP set every value in "regN" to zero. If there was no synchronous reset, our outputs "rdata_1" and "rdata_2" were set as "regN(i)" with their respective index. If write enable was active high, then "regN(w_index)" was given the "wdata" input. We fed 12 inputs into our register file in the test bench, each time reading from two addresses and writing to one address.

3 Creating the Counter

3.1 Architecture

Our design for the counter starts with defining two GENERIC variables. "NBIT" is the amount of bits that is sent into the internal counter register as well as the number of bit in the output vector. "STEP" is an integer used to increment or decrement in the counter by a certain value. After defining our inputs and outputs, we started our architecture using the behavioral style of coding. Our PROCESS statement only relied on the "clk" signal changing in the sensitivity list. Within the PROCESS, we defined a variable called "temp," which would be used for each IF case. All cases depended on "clk" having an event and equaling '1.' The first case was having the synchronous reset set "temp" to all 0's. If "preload" was '1,' then "din" would be sent to the

internal counter register. When "asc" was '1,' "temp" would increment by setting it equal to itself plus an unsigned vector with the value of "STEP" and the length of "NBIT." THe same procedure was used when "asc" was '0,' except the subtraction operator was used. After ending the IF statements, "temp" was sent to "dout." Then, our PROCESS was finished. We fed a total of two random inputs into the "din" signal over a series of 14 rounds, through "preload," in order to test the ascending and descending feature.

3.2 Errors Encountered

3.2.1 Compilation

Problem: The only issue we had with the counter was how to increment and decrement. Originally, we had used the command "temp + 1" and "temp - 1." This would not work because "temp" was an STD_LOGIC_VECTOR and 1 was an integer.

Resolution: In order to fix this problem, we converted "temp" to unsigned and used the command "to_unsigned(STEP,NBIT)" in order to have two unsigned vectors. This allowed us to add or subtract "temp" with an unsigned vector, given the value of "STEP" and the length of "NBIT." The final step was changing this unsigned vector back to an STD_LOGIC_VECTOR.

4 Creating the Testbench

4.1 Architecture

Our design for the testbench, written in SystemVerilog, allows us to verify the functionality of our components, by feeding them random inputs and testing specific cases. Our group decided to use one test bench that implemented all our components. We start the test bench by defining our signals, including those for the register, register file, counter, and the SystemVerilog design of our components. We created the same components in SystemVerilog in order to verify the outputs and compare with the expected value. After defining all signals and variables, we started our tests. Within the initial begin, we used a for loop to initialize the register file to all zeros. "Clk" was also initialized as '0.' Then, we began a series of tests with 100 ns intervals. Each round of tests checked the functionality of the register, register file, and counter. There were 10 rounds total that used each component. The 11th and 12th round did not feed inputs to the register. The final two rounds only tested the counter. After the initial begin statement, we generated two clocks with an always statement. The first clock had a period of 100 ns, while the second had a period of 96 ns. The second clock, "clk2," was used to imitate an input skew. After generating our clocks, we used an always statement that responded to the positive edge of "clk2" in order to generate random input values for the register, the register file ("wdata"), and the counter data input. We used an if statement, which depended on whether the inputs had less than 32 bits, otherwise they were given "$urandom," which defaults as 32 bits. "Clk2" was only used for feeding in random inputs, whereas "clk" was

used throughout the rest of the test bench for all other necessary always statements. Then, we instantiated our register, register file, and counter components. The next four always statements were used to create SystemVerilog equivalents of our components, which produced an expected output. Three responded to the positive edge of the clock, while one of the four was sensitive to the asynchronous reset, which was a feature of the register. The final step was to compare the VHDL components with their SystemVerilog counterparts. At the positive edge of the clock, a "$display" function was used to show important information, including the inputs, "clk" value, and "$time" during simulation. At the negative edge of the clock, an if statement checked if the expected data out of the SystemVerilog component matched the actual output of the VHDL component. Two display commands were used for each case, pass or fail. Whether the test passed or failed, the same information was displayed. This included all outputs of each VHDL component, as well as the expected output. Overall, our test bench was essential in the verification of our components and proved the correct functionality of our register, counter, and register file.