LE/EECS 2021 4.00   Computer Organization

LAB L REPORT

Michael Williams 211087798

Section E

November 17rd 2014

The work in this report is my own. I have read and understood York University academic dishonesty policy and I did not violate the senate dishonesty policy in writing this report.

ABSTRACT

In this lab, we were taught to use the basics and fundamentals of Verilog programming and incorporate them further into means of creating multiplexors to perform different tasks, based on the simple versions of the same multiplexors that were made in the previous lab. This meant taking simplistic circuits and manipulating them to incorporate a wide variety of inputs, even as big as full 32 bit words. This also meant developing an understanding on how module functions work in Verilog, as well as their similarities and differences to method construction in java programming. It also meant understanding just what the limitations were to creating module functions and more so, figuring out ways to work around standard Verilog coding that was no longer available in a module function. The purpose of this lab was to teach us how to efficiently and correctly create modules for instantiation, using the practises mentioned above, as well as to use multiple modules together, and create new complex multiplexors capable of performing a wide variety of functions given limited input, in order to create a foundation of understanding that would become essential for upcoming labs, and understanding how to program module instantiations and create multiplexors in general. For the most part, the code was rather simple, as it was mostly just following instructions. There were a few problems that involved thinking outside the box and large amounts of testing until understanding the limitations when creating module instances. However, with the exception of those few instances, the lab itself was average difficulty, albeit far more challenging that what we had previous been required to do. In conclusion, we were shown and given the means to practise Verilog to a point of proper understanding so that we might be able to apply to future labs, and future understandings of more difficult concepts to be discussed, regarding Verilog.

EQUIPMENT

-laptop, where most of my work was done, with the exception of a few minor corrections in the lab

-putty, which allowed me access to the eecs servers to continue working on my labs, even from home

-xming, allowing me to open visual windows of the server programs

-gedit, used for the editing and writing of the actual coding seen in the appendix section.

METHODS/PROCEDURES

For the most part, the lab was above average difficulty, but I still managed to complete it. I simply followed the given instructions but at times this did mean making some alterations to the provided code. I especially began to have increased difficulty in programs 6, 8, 9, and 10. Throughout the rest of the labs we were provided the equations and methods needed for creating the program. However in the programs mentioned above, it become my task of filling in the blanks left behind, which proved to be quite troublesome at times. For example, in lab 8 when developing the yArith module, one very big problem was figuring out a way to send either b or ~b depending on the control value, due to the limitations on the module itself. After much rigorous testing, I was able to do so using the statement

“xor my\_xor[31:0] (notB, b, ctrl)”.

Another example was while developing the ALU module, and having to come up with a statement that would do the same function as an if statement, again because of the limitations of the module itself. Luckily I was able to come up with the statement

“assign z\_slt[0] = (tmp === 1) ? a[31] : yArith[31]”.

Luckily, the methods and variables used in the main module testers were still basic and all from the same information learned in lab K. As a result, these testers were quite easy to implement efficiently and effectively. For a better understanding of the methods and procedures used in the programs of this lab, please go to the appendix section, where the entirety of the code itself is shown. You can also refer to the results section to see what the outputs are supposed to look like under the given circumstances.

RESULTS

The results shown below are comprised of both the example inputs given for programs, as well as my own inputs for instances where we were not provided an input. Input is only provided for programs that specifically needed input or were given input. However, because of the nature of this lab the results are, for the most part, obtained through random testing or exhaustive testing. There should be more than enough test examples below to ensure that the programs run and output as they are supposed to, not to mention that the oracle is also available in each program to ensure that the program outputs correctly, right or wrong.

LABL1

Input: a = 0;

b = 1;

c = 1;

Output: a=0 b=1 c=1 z=1

LABL2

Input: a = 1;

b = 3;

c = 0;

Output: a=01 b=11 c=0 z=01

LABL3

Input: Input was randomly created for these examples. However, the randomly generated values a, b, and c are provided to ensure that the program outputs properly.

Output:

PASS: a=00010010000101010011010100100100

b=11000000100010010101111010000001

c=1

z=11000000100010010101111010000001

PASS: a=10110001111100000101011001100011

b=00000110101110010111101100001101

c=1

z=00000110101110010111101100001101

PASS: a=10110010110000101000010001100101

b=10001001001101110101001000010010

c=1

z=10001001001101110101001000010010

PASS: a=11100011001101110010010011000110

b=11100010111101111000010011000101

c=0

z=11100011001101110010010011000110

PASS: a=11110100000000000111101011101000

b=11100010110010100100111011000101

c=0

z=11110100000000000111101011101000

LABL4

Input: Input was randomly created for these examples. However, the randomly generated values a0, a1, a2, a3, and c are provided to ensure that the program outputs properly.

Output:

PASS: a0=00010010000101010011010100100100

a1=11000000100010010101111010000001

a2=10000100100001001101011000001001

a3=10110001111100000101011001100011

c=01

z=11000000100010010101111010000001

PASS: a0=01000110110111111001100110001101

a1=10110010110000101000010001100101

a2=10001001001101110101001000010010

a3=00000000111100111110001100000001

c=01

z=10110010110000101000010001100101

PASS: a0=00111011001000111111000101110110

a1=00011110100011011100110100111101

a2=01110110110101000101011111101101

a3=01000110001011011111011110001100

c=01

z=00011110100011011100110100111101

PASS: a0=11100011001101110010010011000110

a1=11100010111101111000010011000101

a2=11010101000100111101001010101010

a3=01110010101011111111011111100101

c=11

z=01110010101011111111011111100101

PASS: a0=10001001001100101101011000010010

a1=01000111111011001101101110001111

a2=01111001001100000110100111110010

a3=11100111011101101001011011001110

c=00

z=10001001001100101101011000010010

LABL5

Input: Input was generated using exhaustive testing in this program. The generated values a, b, and cin are provided to ensure that the program outputs properly.

Output: a=0 b=0 cin=1 z=1 cout=0

a=0 b=1 cin=0 z=1 cout=0

a=0 b=1 cin=1 z=0 cout=1

a=1 b=0 cin=0 z=1 cout=0

a=1 b=0 cin=1 z=0 cout=1

a=1 b=1 cin=0 z=0 cout=1

LABL6

Input: Input was randomly created for these examples. However, the randomly generated values a, b, and cin are provided to ensure that the program outputs properly.

Output:

a=00010010000101010011010100100100

b=11000000100010010101111010000001

cin=0 z=11010010100111101001001110100101

ok=1

a=10000100100001001101011000001001

b=10110001111100000101011001100011

cin=0 z=00110110011101010010110001101100

ok=1

a=00000110101110010111101100001101

b=01000110110111111001100110001101

cin=0 z=01001101100110010001010010011010

ok=1

a=10110010110000101000010001100101

b=10001001001101110101001000010010

cin=0 z=00111011111110011101011001110111

ok=1

a=00000000111100111110001100000001

b=00000110110101111100110100001101

cin=0 z=00000111110010111011000000001110

ok=1

LABL7

Input: Input was randomly created for these examples. However, the randomly generated values a0, a1, a2, a3, and c are provided to ensure that the program outputs properly.

Output:

a=00010010000101010011010100100100

b=11000000100010010101111010000001

cin=0

z=11010010100111101001001110100101

ok=1

a=10000100100001001101011000001001

b=10110001111100000101011001100011

cin=0

z=00110110011101010010110001101100

ok=1

a=00000110101110010111101100001101

b=01000110110111111001100110001101

cin=0

z=01001101100110010001010010011010

ok=1

a=10110010110000101000010001100101

b=10001001001101110101001000010010

cin=0

z=00111011111110011101011001110111

ok=1

a=00000000111100111110001100000001

b=00000110110101111100110100001101

cin=0

z=00000111110010111011000000001110

ok=1

LABL8

Input: Input was randomly created for these examples. However, the randomly generated values a0, a1, a2, a3, and ctrl are provided to ensure that the program outputs properly.

Output:

PASS: a= 303379748 b=-1064739199 ctrl=1

z= 1368118947 expect= 1368118947

PASS: a=-1309649309 b= 112818957 ctrl=1

z=-1422468266 expect=-1422468266

PASS: a=-1295874971 b=-1992863214 ctrl=1

z= 696988243 expect= 696988243

PASS: a= 114806029 b= 992211318 ctrl=1

z= -877405289 expect= -877405289

PASS: a= 1993627629 b= 1177417612 ctrl=1

z= 816210017 expect= 816210017

PASS: a= -482925370 b= -487095099 ctrl=0

z= -970020469 expect= -970020469

PASS: a= 1924134885 b=-1143836041 ctrl=0

z= 780298844 expect= 780298844

PASS: a= 1206705039 b= 2033215986 ctrl=0

z=-1055046271 expect=-1055046271

PASS: a= -201295128 b= -490058043 ctrl=0

z= -691353171 expect= -691353171

PASS: a= -561108803 b=-1767155667 ctrl=1

z= 1206046864 expect= 1206046864

LABL9

Input: Input was randomly created for these examples. However, the randomly generated values a, and b are provided to ensure that the program outputs properly. In each example, a different value of op is given so that each of the 4 available computations may be tested.

Output:

ok=1

a=00010010000101010011010100100100

b=11000000100010010101111010000001

z=00000000000000010001010000000000

expect=00000000000000010001010000000000

op=0

ok=1

a=00010010000101010011010100100100

b=11000000100010010101111010000001

z=11010010100111010111111110100101

expect=11010010100111010111111110100101

op=1

ok=1

a=00010010000101010011010100100100

b=11000000100010010101111010000001

z=11010010100111101001001110100101

expect=11010010100111101001001110100101

op=2

ok=1

a=00010010000101010011010100100100

b=11000000100010010101111010000001

z=01010001100010111101011010100011

expect=01010001100010111101011010100011

op=6

LABL10

Input: Input was randomly created for these examples. However, the randomly generated values a, and b are provided to ensure that the program outputs properly. In each example, a different value of op is given so that each of the 4 available computations may be tested.

Output:

ok=1

a=00010010000101010011010100100100

b=11000000100010010101111010000001

z=00000000000000010001010000000000

expect=00000000000000010001010000000000

op=0

ok=1

a=10000100100001001101011000001001

b=10110001111100000101011001100011

z=10000000100000000101011000000001

expect=10000000100000000101011000000001

op=0

ok=1

a=00000110101110010111101100001101

b=01000110110111111001100110001101

z=00000110100110010001100100001101

expect=00000110100110010001100100001101

op=0

ok=1

a=00010010000101010011010100100100

b=11000000100010010101111010000001

z=11010010100111010111111110100101

expect=11010010100111010111111110100101

op=1

ok=1

a=10000100100001001101011000001001

b=10110001111100000101011001100011

z=10110101111101001101011001101011

expect=10110101111101001101011001101011

op=1

ok=1

a=00000110101110010111101100001101

b=01000110110111111001100110001101

z=01000110111111111111101110001101

expect=01000110111111111111101110001101

op=1

ok=1

a=00010010000101010011010100100100

b=11000000100010010101111010000001

z=11010010100111101001001110100101

expect=11010010100111101001001110100101

op=2

ok=1

a=10000100100001001101011000001001

b=10110001111100000101011001100011

z=00110110011101010010110001101100

expect=00110110011101010010110001101100

op=2

ok=1

a=00000110101110010111101100001101

b=01000110110111111001100110001101

z=01001101100110010001010010011010

expect=01001101100110010001010010011010

op=2

ok=1 a=00010010000101010011010100100100

b=11000000100010010101111010000001

z=01010001100010111101011010100011

expect=01010001100010111101011010100011

op=6

ok=1

a=10000100100001001101011000001001

b=10110001111100000101011001100011

z=11010010100101000111111110100110

expect=11010010100101000111111110100110

op=6

ok=1

a=00000110101110010111101100001101

b=01000110110111111001100110001101

z=10111111110110011110000110000000

expect=10111111110110011110000110000000

op=6

ok=1 a=00010010000101010011010100100100

b=11000000100010010101111010000001

z=00000000000000000000000000000000

expect=00000000000000000000000000000000

op=7

ok=1

a=10000100100001001101011000001001

b=10110001111100000101011001100011

z=00000000000000000000000000000001

expect=00000000000000000000000000000001

op=7

ok=1

a=00000110101110010111101100001101

b=01000110110111111001100110001101

z=00000000000000000000000000000001

expect=00000000000000000000000000000001

op=7

LABL11

Input: Input was randomly created for these examples. However, the randomly generated values a, and b are provided to ensure that the program outputs properly. In each example, a different value of op is given so that each of the 4 available computations may be tested.

Output:

ok=1

a=11100010111101111000010011000101

b=11010101000100111101001010101010

z=11000000000100111000000010000000

expect=11000000000100111000000010000000

op=0

ok=1

a=01110010101011111111011111100101

b=10111011110100100111001001110111

z=00110010100000100111001001100101

expect=00110010100000100111001001100101

op=0

ok=1

a=11100010111101111000010011000101

b=11010101000100111101001010101010

z=11110111111101111101011011101111

expect=11110111111101111101011011101111

op=1

ok=1

a=01110010101011111111011111100101

b=10111011110100100111001001110111

z=11111011111111111111011111110111

expect=11111011111111111111011111110111

op=1

ok=1

a=11100010111101111000010011000101

b=11010101000100111101001010101010

z=10111000000010110101011101101111

expect=10111000000010110101011101101111

op=2

ok=1

a=01110010101011111111011111100101

b=10111011110100100111001001110111

z=00101110100000100110101001011100

expect=00101110100000100110101001011100

op=2

ok=1

a=11100010111101111000010011000101

b=11010101000100111101001010101010

z=00001101111000111011001000011011

expect=00001101111000111011001000011011

op=6

ok=1

a=01110010101011111111011111100101

b=10111011110100100111001001110111

z=10110110110111011000010101101110

expect=10110110110111011000010101101110

op=6

ZERO FAIL: ok=1

a=11100010111101111000010011000101

b=11010101000100111101001010101010

z=00000000000000000000000000000000

expect=00000000000000000000000000000000

op=7

ok=1

a=01110010101011111111011111100101

b=10111011110100100111001001110111

z=00000000000000000000000000000000

expect=00000000000000000000000000000000

op=7

ZERO FAIL: ok=1

a=01110010101011111111011111100101

b=10111011110100100111001001110111

z=00000000000000000000000000000000

expect=00000000000000000000000000000000

op=7

DISCUSSION

Some of the programs in this lab proved to be quite difficult. In particular, I had a lot of trouble with the programs associated with labL6, labL8, labL9 and labL10. With these programs, a lot of trouble stemmed from my limited knowledge of the limitations faced by programmers when creating module functions. The explanation given in the lab, I felt, was insufficient in explaining these limitations, or the variety of methods or procedures available for working around these limitations. For example, while developing the yArith module, I was completely unaware that we could not create if statements. This made distinguishing whether or not the function was adding or subtracting the second input that much harder to evaluate. Even in the cases where we were required to fill in the blanks for programs 9 and 10, I did not feel enough information was given to entirely understand what it was that was supposed to be included, or for that matter, the correct and most efficient ways of doing so. Unfortunately, I found that the only option was to rigorously and continuously test different methods, until finding one that worked. However, upon figuring out these few strategies, the program itself became less difficult. It was still difficult, but at the very least I was able to build on what I learned and now have some idea as to how to make certain implementations when creating module functions. The only other slight problem I encountered was when we I was creating the first mux value, but soon came to realize that the reason it hadn’t been working properly was because I had been instantiating in the wrong place within the tester.

LABL1

The lab itself behaved as expected. The outputs were the same as the outputs from the previous lab. Despite the entirety of the program itself being made into a module function, it was quite easy to replicate the results from the 2-to-1 MUX. See the results section for further inspection of the output or the appendix section to refer to the coding. They generally follow the same coding schematics, with a few minor exceptions.

LABL2

Given the visual diagram of the circuit being created for this lab, it is easy to understand that yMux2 does indeed function as it is expected to. This is because it behaves just like yMux1, which up to this point had been confirmed to work properly, in the sense that it opens a gate in the mux dependent on the input value of c. In other words, because all bits of input a and b are still reliant on the input of c, the same gates open under the same circumstances, no matter the size of the input. An easier way to understand this is simply by looking at the right side of the given diagram in the labL2 section. As it shows, the 2 bit buses can be broken down into 2 separate Mux modules, 1 for each bit. Hence, we are essentially running 2 of the same yMux1 modules that we know run as expected and combining their results. Therefore, this will run as expected. Upon running and testing yMux2 this can also be confirmed.

LABL3

Even when changing to array based instantiation, the program still behaves just as expected. Now, given the specific size value when instantiating the code, yMux can effectively run any number of bits and still produce the correct output. Because there are so many possible values, exhaustive testing is not a good idea. Therefore, we proceed to use random testing, in which random values are created and used as input. Combined with a given number of repeat operations, we can output whatever number of random values we want. In the code

repeat (10)

begin

a = $random;

b = $random;

c = $random % 2;

#1;

// compare z with the expected output

end

“(10)” has a large significance. This ensures that the code within the following statement block will repeat itself 10 times, effectively producing 10 very different inputs and outputs. This value can be changed to anything.

LABL4

In this lab, the previously made yMux was used to create a 4-to-1 Mux, which would essentially replicate the processes of the original mux, only this time around the controlling value c would be given up to 4 inputs, being 0, 1, 2, or 3 and depending on which input was given, 1 of 4 different gates would open. For example, if c is 0, the input corresponding to 0 would be the output. In its simplest form, it is just a yMux with 2 more gates present. When created and tested, this new mux does indeed behave as expected; taking 4 different inputs and, depending on the input c, producing a result that corresponds to 1 of the inputs: 0 for a0, 1 for a1, 2 for a2, and 3 for a3. For proof of this, refer to the results section. Refer to the next page in the lab to view the diagram of the circuit. This diagram shows that the implementation does indeed have the correct functionality of a 4-to-1 mux. As shown in the diagram, given four different inputs, dependent on the combination of the 2-bit c input, only 1 input will be produced in the resultant wire z. More specifically, this diagram shows that:

If C = 00, then C(0) = 0 and C(1) = 0 which would make A(0) the resultant

If C = 01, then C(0) = 1 and C(1) = 0 which would make A(1) the resultant

If C = 10, then C(0) = 0 and C(1) = 1 which would make A(2) the resultant

If C = 11, then C(0) = 1 and C(1) = 1 which would make A(3) the resultant

Therefore, the 4-to-1 Mux works as expected to.

LABL5

In this lab, we created an encapsulation of the full adder created in the previous lab, so that it would work from a module function and when instantiated. This module functioned as expected and because the input consisted of a small domain, using exhaustive testing was the best means of testing it. To see this exhaustive testing please refer to lab5 of the results section.

LABL6

In this lab, a 32-bit adder was created as a module function, taking 2 32-bit busses as input as well as a 1-bit cin, that was set to 0 for the proportion of this program. This adder would produce the sum of the first 2 inputs. This can be done with 32 1-bit adders, because that is essentially what the module is being made to do. Just like the variations of yMux, we simply use array based instantiation, and sent an entire 32-bit value as input, where each bit is taken through the full adder, and computed.

We built this program specifically so that it could take 32-bit inputs. The reason for this, instead of altering the program to take any number of bits as input, like in ymux, was because of the way the full adder itself is built. It is made so that the carry in of one adder corresponds to the carry out of another adder. Therefore it takes the 1-bit adder a small period of time to return a value. As a result, the net latency is proportional to the amount of columns we have to compute, or the number of bits in the input. This can have serious ramifications with the program computation time if the user is allowed to specify the size of the output. Therefore, we simply adopt a 32-bit adder. More so, values can only be properly expressed in terms of 32 bits. Adding more bits to the question would further complicate the circuit itself. This adder does however, still compute the correct sum, given any random input of variables. This can be tested using random input as shown in the results section covering labL6.

LABL7

For this lab, we were required to simply add “signed” to each register and wire variable, so that the adder would also work under signed values, or in other words, with positive and negative values. Because nothing in the actual coding changes, the program itself runs as it should. See results of labL7 for examples of the change and to verify that it still works as it should.

THE FOLOWING PAGE CONTAINS THE DIAGRAM SPECIFIED IN THE LAB4 SECTION

LABL8

For this lab, we wish to enhance our adder so that it can also subtract values. To do this, there is no need to instantiate 2 versions of the adder., with one doing the adding and the other doing the subtracting. This can be accomplished using 1 line: “xor xorB[31:0](notB, b, ctrl);”.

This will keep the value of b the same if ctrl is 0, and negate it if ctrl is 1. The resultant value is stored in notB. From there we simply call the adder function and use notB as the secondary input b.

This causes the program to compute as expected, adding if ctrl = 0, and subtracting if ctrl = 1. We know subtracting works as it is supposed to because we essentially add the negative value + 1, which effectively turns input b into a negative, whenever ctrl = 1. Thus, when adding, the resultant simulates subtracting. Further proof of this can be seen from the results section.

LABL9

In this lab, we built an ALU or arithmetic logical unit, that given an input value c, did 1 of 4 various computations and returned the result. If the value of c were 0, 1, 2, or 6, it would produce a & b, a | b, a+b, or a – b, respectively. The ALU does indeed run as expected and produces the output of the program, given the specific input code, for any of the 4 operations. Some examples of this are included in the results portion of the lab

LABL10

For this lab, we simply added to the previous built ALU the option to compute the such less than, or slt operation, as well as the previous operation options. This operation executes as it is supposed to, just as the other operations work correctly as well. For proof that this computes as it is supposed to, please refer to the results section. The specifics behind how to implement the slt operation can also be seen in the appendix section under lab10.

LABL11

For this lab we simply seek to enhance our ALU one last time, this time adding support for a zero flag exception, or in other words, to raise an exception whenever the output of z, is 0, regardless of the operation taking place. We do this by oring the 32 wires of z and then using not on the result. This will effectively result in 1 whenever the result of z is 0. This value is then stored in the exp output so that when running a tester, we can easily extract the value and determine whether an exception was thrown or not. Examples of this can be seen in the results section.

CPU.V

Because future labs will be dependent on the modules created in this lab, it is much easier and efficient to simply create one program that contains the implementation for all the needed modules. When a specific module is called, it is simply linked to the file containing all the implementations and the specific implementation requested is computed. This collection is exactly what cpu.v is, and will provide a much greater alternative to recopying each and every file into a new directory for future labs.

CONCLUSION

In conclusion, the lab was definitely the hardest I have come across, albeit not many problems arose. It provided me a lot of challenge much in terms of understanding, and implementation. However, I did learn from it, and now understand how to implement module functions, as well as the limitations of creating module functions and some of the strategies to work around these limitations. This meant taking simplistic circuits and manipulating them to incorporate a wide variety of inputs, even as big as full 32 bit words. This also meant developing an understanding on how module functions work in Verilog, as well as their similarities and differences to method construction in java programming. There were a few problems that involved thinking outside the box and large amounts of testing until understanding the limitations when creating module instances. However, with the exception of those few instances, the lab itself was average difficulty, albeit far more challenging that what we had previous been required to do. In conclusion, we were shown and given the means to practise Verilog to a point of proper understanding so that we might be able to apply to future labs, and future understandings of more difficult concepts to be discussed, regarding Verilog, giving me knowledge and a basis for understanding the future labs regarding Verilog and its many processes

APPENDIX

LABL1

module LabL;

reg a, b, c;

wire z;

yMux1 test(z, a, b, c);

initial

begin

a = 0;

b = 1;

c = 1;

#1 $display("a=%b b=%b c=%b z=%b", a, b, c, z);

$finish;

end

endmodule

LABL2

module LabL;

reg [0:1] a, b;

reg c;

wire [0:1] z;

yMux2 test(z, a, b, c);

initial

begin

a = 1;

b = 3;

c = 0;

#1 $display("a=%b b=%b c=%b z=%b", a, b, c, z);

end

endmodule

LABL3

module LabL;

reg [31:0] a, b, expect;

reg c;

wire [31:0] z;

yMux #(.SIZE(32)) my\_mux(z, a, b, c);

initial

begin

repeat(10) //repeat(10)

begin

a = $random;

b = $random;

c = $random % 2;

#1;

if(c == 0)

expect = a;

else

expect = b;

if(expect == z)

$display("PASS: a=%b b=%b c=%b z=%b", a, b, c, z);

if(expect != z)

$display("FAIL: a=%b b=%b c=%b z=%b", a, b, c, z);

end

end

endmodule

LABL4

module LabL;

reg [31:0] a0, a1, a2, a3, expect;

reg [1:0] c;

wire [31:0] z;

yMux4to1 #(.SIZE(32)) my\_mux(z, a0, a1, a2, a3, c);

initial

begin

repeat(5)

begin //diagram will look like the one at the start of the lab, only with more inputs

a0 = $random;

a1 = $random;

a2 = $random;

a3 = $random;

c = $random % 4;

#1;

if(c == 0)

expect = a0;

else if(c == 1)

expect = a1;

else if(c == 2)

expect = a2;

else if(c == 3)

expect = a3;

if(expect == z)

$display("PASS: a0=%b a1=%b a2=%b a3=%b c=%b z=%b", a0, a1, a2, a3, c, z);

else

$display("FAIL: a0=%b a1=%b a2=%b a3=%b c=%b z=%b", a0, a1, a2, a3, c, z);

end

$finish;

end

endmodule

LABL5

module LabL;

reg a, b, cin, expect;

wire z, cout;

integer i, j, k;

yAdder1 my\_adder(z, cout, a, b, cin);

initial

begin

for(i = 0; i < 2; i = i + 1)

begin

for(j = 0; j < 2; j = j + 1)

begin

for(k = 0; k < 2; k = k + 1)

begin

a = i; b = j; cin = k;

#1;

expect = a+b+cin;

if(expect !=cout || expect !=z)

$display("a=%b b=%b cin=%b z=%b cout=%b", a, b, cin, z, cout);

end

end

end

$finish;

end

endmodule

LABL6

module LabL;

reg [31:0] a, b, expect;

reg cin, ok;

wire [31:0] z;

wire cout;

yAdder #(.SIZE(32), .SIZE2(1)) my\_adder(z, cout, a, b, cin);

initial

begin

cin = 0;

repeat(5)

begin

a = $random;

b = $random;

expect = a + b + cin;

ok = 0;

#1;

if(expect === z)

ok = 1;

$display("a=%b b=%b cin=%b z=%b ok=%b", a, b, cin, z, ok);

end

$finish;

end

endmodule

LABL7

module LabL;

reg signed [31:0] a, b, expect;

reg cin, ok;

wire signed [31:0] z;

wire cout;

yAdder #(.SIZE(32), .SIZE2(1)) my\_adder(z, cout, a, b, cin);

initial

begin

cin = 0;

repeat(5)

begin

a = $random;

b = $random;

expect = a + b + cin;

ok = 0;

#1;

if(expect === z)

ok = 1;

$display("a=%b b=%b cin=%b z=%b ok=%b", a, b, cin, z, ok);

end

$finish;

end

endmodule

LABL8

module LabL;

reg signed [31:0] a, b, expect;

reg ctrl, ok;

wire signed [31:0] z;

wire cout;

yArith my\_adder(z, cout, a, b, ctrl);

initial

begin

repeat(10)

begin

a = $random;

b = $random;

ctrl = $random % 2;

#1;

if(ctrl === 0)

expect = a + b;

else

expect = a - b;

#1;

if(expect === z)

$display("PASS: a=%d b=%d ctrl=%d z=%d expect=%d", a, b, ctrl, z, expect);

else

$display("FAIL: a=%d b=%d ctrl=%d z=%d expect=%d", a, b, ctrl, z, expect);

end

$finish;

end

endmodule

LABL9

module labL;

reg [31:0] a, b;

reg [31:0] expect;

reg [2:0] op;

wire ex;

wire [31:0] z;

reg ok, flag;

yAlu mine(z, ex, a, b, op);

initial

begin

repeat(10)

begin

a = $random;

b = $random;

flag = $value$plusargs("op=%d", op);

if(op === 0)

expect = a & b;

else if(op === 1)

expect = a | b;

else if (op === 2)

expect = a + b;

else if (op === 6)

expect = a - b;

#1;

if(z === expect)

ok = 1;

else

ok = 0;

$display("ok=%b a=%b b=%b z=%b expect=%b op=%d", ok, a, b, z, expect, op);

$finish;

end

end

endmodule

LABL10

module labL;

reg signed [31:0] a, b;

reg signed [31:0] expect;

reg [2:0] op;

wire ex;

wire [31:0] z;

reg ok, flag;

yAlu mine(z, ex, a, b, op);

initial

begin

repeat(5)

begin

a = $random;

b = $random;

flag = $value$plusargs("op=%d", op);

if(op === 0)

expect = a & b;

else if(op === 1)

expect = a | b;

else if (op === 2)

expect = a + b;

else if (op === 6)

expect = a - b;

else if (op === 3'b111)

expect = (a < b) ? 1 : 0;

#1;

if(z === expect)

ok = 1;

else

ok = 0;

$display("ok=%b a=%b b=%b z=%b expect=%b op=%d", ok, a, b, z, expect, op);

end

end

endmodule

LABL11

module labL;

reg signed [31:0] a, b;

reg signed [31:0] expect, zero;

reg [2:0] op;

wire ex;

wire [31:0] z;

reg ok, flag;

yAlu mine(z, ex, a, b, op);

initial

begin

repeat(10)

begin

a = $random;

b = $random;

flag = $value$plusargs("op=%d", op);

if(op === 0)

expect = a & b;

else if(op === 1)

expect = a | b;

else if (op === 2)

expect = a + b;

else if (op === 6)

expect = a - b;

else if (op === 3'b111)

expect = (a < b) ? 1 : 0;

zero = (expect == 0) ? 1 : 0;

#1;

if(z === expect)

ok = 1;

else

ok = 0;

$display("ok=%b a=%b b=%b z=%b expect=%b op=%d", ok, a, b, z, expect, op);

if(zero !== ex)

$display("ZERO FAIL: ok=%b a=%b b=%b z=%b expect=%b op=%d", ok, a, b, z, expect, op);

end

end

endmodule

CPU.V

module yMux1(z, a, b, c);

output z;

input a, b, c;

wire notC, upper, lower;

not my\_not(notC, c);

and upperAnd(upper, a, notC);

and lowerAnd(lower, c, b);

or my\_or(z, upper, lower);

endmodule

//-------------------------------------------------------------

module yMux(z, a, b, c);

parameter SIZE = 2;

output [SIZE - 1:0] z;

input [SIZE - 1:0] a, b;

input c;

yMux1 mine[SIZE - 1:0](z, a, b, c);

endmodule

//------------------------------------------------------------------

module yMux4to1(z, a0, a1, a2, a3, c);

parameter SIZE = 2;

output [SIZE - 1:0] z;

input [SIZE - 1:0] a0, a1, a2, a3;

input [1:0] c;

wire [SIZE - 1 :0] zLo, zHi;

yMux #(SIZE) lo(zLo, a0, a1, c[0]);

yMux #(SIZE) hi(zHi, a2, a3, c[0]);

yMux #(SIZE) final(z, zLo, zHi, c[1]);

endmodule

//------------------------------------------------------------------

module yAdder1(z, cout, a, b, cin);

output z, cout;

input a, b, cin;

xor left\_xor(tmp, a, b);

xor right\_xor(z, cin, tmp);

and left\_and(outL, a, b);

and right\_and(outR, tmp, cin);

or my\_or(cout, outR, outL);

endmodule

module yAdder(z, cout, a, b, cin);

parameter SIZE = 32;

parameter SIZE2 = 1;

integer i;

output [SIZE - 1:0] z;

output [SIZE2 - 1:0] cout;

input [SIZE - 1:0] a, b;

input [SIZE2 - 1:0] cin;

wire [SIZE - 1:0] in, out;

yAdder1 mine[SIZE -1:0] (z, out, a, b, in);

assign in[0] = cin;

assign in[1] = out[0];

assign in[31:2] = out[31:1];

endmodule

//------------------------------------------------------------------

module yArith(z, cout, a, b, ctrl);

//if ctrl = 1, use not b

output signed [31:0] z;

output cout;

input signed [31:0] a, b;

input ctrl;

wire signed [31:0] notB, tmp;

wire cin;

xor xorB[31:0](notB, b, ctrl); //using xor will make notB = b is ctrl = 0, else it will

//negate the value when ctrl = 1 to simuulate subtraction

yAdder mine1(z, cout, a, notB, ctrl);

endmodule

//------------------------------------------------------------------

module yAlu(z, ex, a, b, op);

input [31:0] a, b;

input [2:0] op;

output [31:0] z;

output ex;

wire [31:0] z\_arith, z\_and, z\_or, z\_slt;

wire tmp;

wire[15:0] or16;

wire[7:0] or8;

wire[3:0] or4;

wire[1:0] or2;

wire cout;

assign z\_slt[31:1] = 0;

assign ex = 0; //not supported

xor (tmp, a[31], b[31]);

assign z\_slt[0] = (tmp === 1) ? a[31] : z\_arith[31];

yArith my\_arith[31:0] (z\_arith, ex, a, b, op[2]);

and my\_and[31:0] (z\_and, a, b);

or my\_or[31:0] (z\_or, a, b);

yMux4to1 #(.SIZE(32)) my\_mux(z, z\_and, z\_or, z\_arith, z\_slt, op[1:0]);

or or\_16[15:0] (or16, z[31:16], z[15:0]);

or or\_8[7:0] (or8, or16[15:8], or16[7:0]);

or or\_4[3:0] (or4, or8[7:4], or8[3:0]);

or or\_2[1:0] (or2, or4[3:2], or4[1:0]);

nor nor\_1 (ex, or2[1], or2[0]);

endmodule