COMPUTER ORGANIZATION
EECS 2021

LAB "L" REPORT
HARDWARE BUILDING BLOCKS

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.

X _____

Camillo John (CJ) D'Alimonte
212754396
Lab Section 2
11/11/14
Professor Aboelaze

Abstract:

This laboratory/experiment introduced me to some of the more complex applications that can be built through the Verilog language. I learned how to program different adders and muxes and how to test their functionality using random and exhaustive testing. I also got a chance to model an ALU in Verilog which allowed me to learn how to add and subtract bits in Verilog. I was able to design Verilog code from a given circuit diagram. By the end of the lab, I had learned some of the more complex applications that can be programmed in the Verilog language.

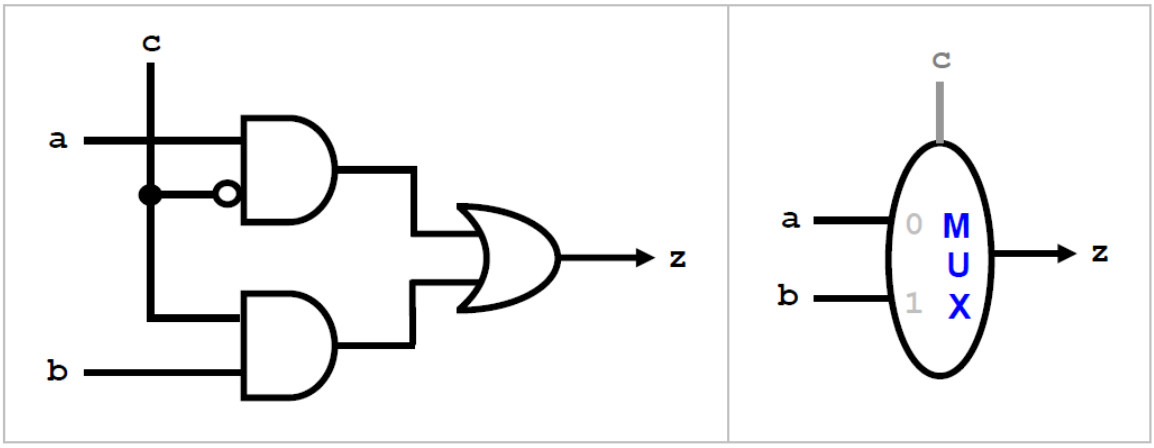
Equipment Used:

This lab was done through the UNIX environment at the Lassonde Lab. I used the pico editor in UNIX to write my code and I used the built-in Verilog compiler to run and test my programs.

Methods/Procedures:

For LabL1, I needed to use abstraction to encapsulate the provided circuit (see below) as a reusable component so that it could be used by someone who doesn't know how it works internally. I needed to create a program, yMux1.v using the following code:

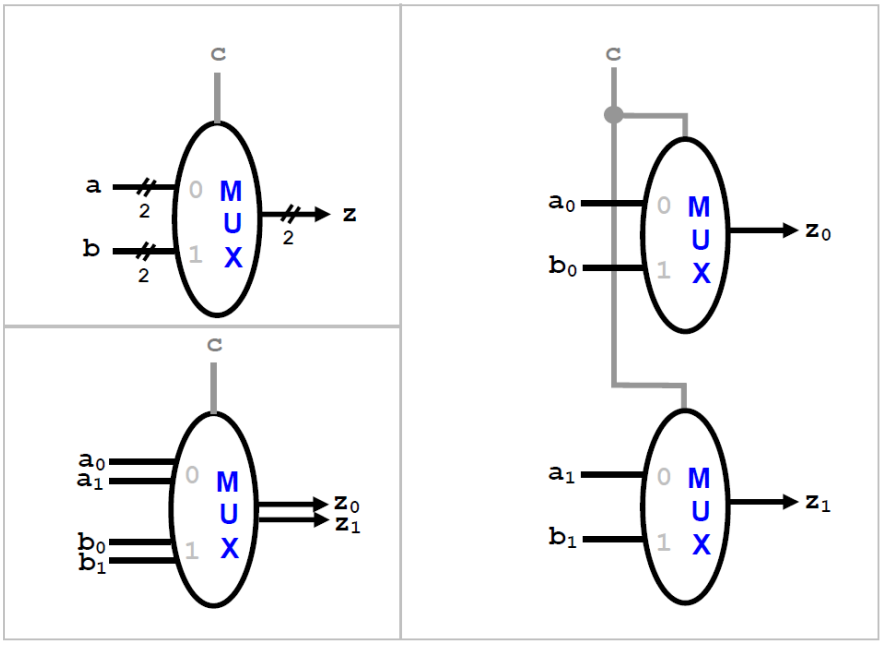
```
module yMux1(z, a, b, c);
        output z;
      input a, b, c;
  wire notC, upper, lower;
    not my_not(notC, c);
and upperAnd(upper, a, notC);
  and lowerAnd(lower, c, b);
  or my_or(z, upper, lower);
        endmodule
```



After creating yMux1, I needed to write a program that instantiates and tests yMux1. I would have to test my program using either manual testing (via command-line arguments) or through an exhaustive, three-nested loop test (which is the test method I use).

Pseudo Code
LabL1.v
	integer i, j, k;
	reg a, b, c, expect;
	wire z;
	yMux1 test(z, a, b, c);
	
	for (i → 0; i < 2; i → i + 1)
	for (j → 0; j < 2; j → j + 1)
	for (k → 0; k < 2; k → k + 1)
	a → i; b → j; c → k;
	
	expect (c→ 0) ? a : b;
	#1;
	if (expect → z)
	$display("PASS: a, b, c, z);
	else
	$display("FAIL: a, b, c, z);
	end

For LabL2, I needed to enhance my yMux1 program so that it can handle 2-bit busses instead of 1-bit wires. The control input, c, would still be 1-bit but the two data inputs and the output would become 2-bits.



I needed to create a program, LabL2.v, which would instantiate and test yMux2, the enhanced version of yMux1. Once again, I chose to use exhaustive testing.

Pseudo Code
LabL2.v
	integer i, j, k;
	reg c;
	reg [1:0] a, b, expect;
	wire [1:0] z;
	yMux2 test(z, a, b, c);
	for (i → 0; i < 4; i → i + 1)
	  for (j → 0; j < 4; j → j + 1)
	  for (k → 0; k < 2; k → k + 1)
	  a → i; b → j; c → k;
	  expect[0] = (c → 0) ? a[0] : b[0];
	  expect[1] = (c → 0) ? a[1] : b[1];
	  if (expect[0] →z[0] && expect[1] → z[1])
	  $display("PASS: a, b, c, z);
	  else
	  $display("FAIL: a, b, c, z);
	  End

For LabL3, I needed to design a 32 bit 2-to-1 mux using an array-based instantiation. We will localized the value of the bus size using the parameter statement rather than keep it hard coded in various places. I needed to write a program, LabL3.v which instantiates and tests yMux, using a bus of width 32.

```
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
```

Pseudo Code
LabL3.v
	reg c;
	reg [31:0] a, b, expect;
	wire [31:0] z;
	yMux #(32) test(z, a, b, c);
	
	repeat (500)
	  a → $random;
	  b → $random;
	  c → $random % 2;
	
	  expect = (c → 0) ? a : b;
	  if (expect != z)
	  $display("FAIL: a, b, c, z);
	end

For LabL4, I needed to design a 32-bit, 4-to1 mux; i.e. a circuit that selects amongst four 32-bit inputs, a0, a1, a2, a3, based on a 2-bit control signal c. I was able to benefit from our existing 32-bit, 2-to-1 we had previously used. I had to write a program, LabL4.v, which randomly tested my 4-to1 mux. The code for the yMuz4to1 is provided:

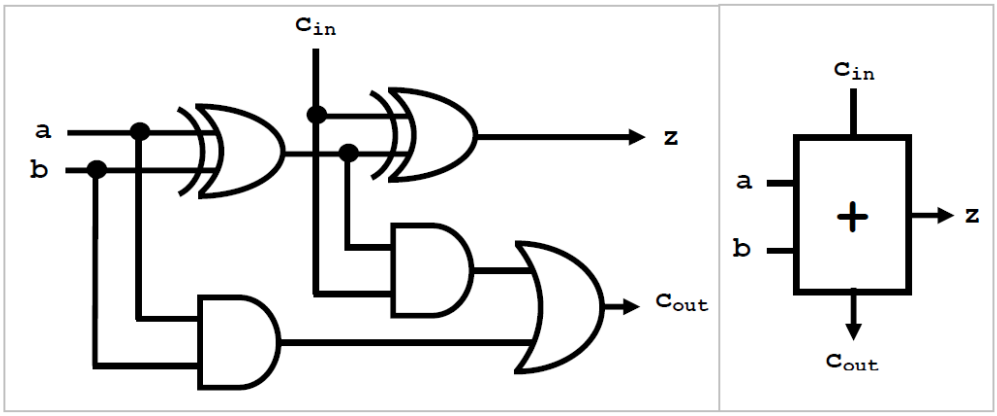
```
module yMux4to1(z, a0,a1,a2,a3, c);
        parameter SIZE = 2;
       output [SIZE-1:0] z;
  input [SIZE-1:0] a0, a1, a2, a3;
          input [1:0] c;
     wire [SIZE-1:0] zLo, zHi;
  yMux #(SIZE) lo(zLo, a0, a1, c[0]);
  yMux #(SIZE) hi(zHi, a2, a3, c[0]);
yMux #(SIZE) final(z, zLo, zHi, c[1]);
            endmodule
```

Pseudo Code LabL4.v	reg [1:0] c;
	reg [31:0] a0, a1, a2, a3, expectLo, expectHi, expect;
	wire [31:0] z;
	yMux4to1 #(32) test(z, a0, a1, a2, a3, c);
	
	repeat (500)
	a0 → $random;
	a1 → $random;
	a2 → $random;
	a3 → $random;
	c → $random % 4; //modular 4
	
	expectLo = (c[0] → 0) ? a0 : a1;
	expectHi = (c[0] → 0) ? a2 : a3;
	expect = (c[1] → 0) ? expectLo : expectHi;
	
	if (expect != z)
	$display("FAIL: a0, a1, a2, a3, c, z);
	end

For LabL5, I needed to use abstraction to encapsulate the functionality of the full adder I built in a previous lab. I would need to write a tester program to test the code provided to us, as follows below. My tester would only output if the sum a+b+cin has a MSb that is different from cout or a LSb that is different from z.

```
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
```



Pseudo Code
LabL5.v
	reg [1:0] c;
	reg [31:0] a0, a1, a2, a3, expectLo, expectHi, expect;
	wire [31:0] z;
	yMux4to1 #(32) test(z, a0, a1, a2, a3, c);
	
	repeat (500)
	a0 → $random;
	a1 → $random;
	a2 → $random;
	a3 → $random;
	c → $random % 4; //modular 4
	
	expectLo = (c[0] → 0) ? a0 : a1;
	expectHi = (c[0] → 0) ? a2 : a3;
	expect = (c[1] → 0) ? expectLo : expectHi;
	
	if (expect != z)
	$display("FAIL: a0, a1, a2, a3, c, z);
	end

For LabL6, I needed to build a 32 bit adder which is a circuit that takes two 32-bit busses, a and b, and a single bit, cin, and produces their sum z and an overall carry cout. Using random testing, I needed to write LabL6.v which instantiates and tests yAdder.

Note: I also had to finish the development of the yAdder. Changes are underlined.

```
module yAdder(z, cout, a, b, cin);
output [31:0] z;
output cout;
input [31:0] a, b;
input cin;
```

```
wire[31:0] in, out;
yAdder1 mine[31:0](z, out, a, b, in);
assign in[0] = cin;
assign in[31:1] = out[30:0];
assign cout = out[31];
```

Pseudo Code LabL6.v	reg cin, ok;
	reg [31:0] a, b, expect;
	wire [31:0] z;
	
	yAdder test(z, cout, a, b, cin);
	
	repeat (500)
	a = $random;
	b = $random;
	cin = $random%2;
	expect → a + b + cin;
	ok = 0;
	if (expect → z) ok = 1;
	if(!ok)
	$display("FAIL: a, b, z);
	end

For LabL7, I needed to re-test my adder by interpreting the random inputs as signed integers by adding the keyword signed after each reg and wire declaration.

Pseudo Code LabL7.v	reg signed [31:0] a, b, expect; wire signed [31:0] z;

For LabL8, I needed to enhance my 32-bit adder so it could subtract. Subtraction is completed using this identity:

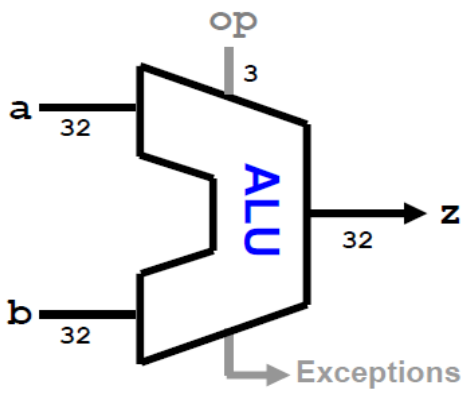
```
a – b = a + [(not b) + 1] = a + (not b) + 1
```

Pseudo Code LabL8.v	reg ctrl, ok; reg [31:0] a, b, expect; wire [31:0] z; yArith test(z, cout, a, b, ctrl); repeat (500) a → $random; b → $random; ctrl → $random%2;

	if(ctrl)
	expect →a + ~b + 1;
	else
	expect → a + b + ctrl;
	
	ok = 0;
	if (expect → z) ok = 1;
	if(!ok)
	$display("FAIL: a, b, z);

For LabL9-LabL10, I needed to build an arithmetic logic unit (ALU). An ALU is a versatile unit that performs a variety of operations based on a control input op. In addition to z, the ALU also generates a number of exception signals (or *flags*) to indicate exceptions, such as a zero result, a carry, or a signed overflow. These exceptions are meaningful only for certain operations and then only for signed or unsigned input interpretation. I needed to create a program, yAlu.v, and test it with both LabL9.v and Lab10.v

op	result
000	a & b
001	a | b
010	a + b
110	a - b
111	slt



Pseudo Code
LabL9/10.v
	reg signed [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);
	repeat (10)
	begin
	a → $random;
	b → $random;
	flag → ("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;
	else if(op → 7
	expect = (a < b) ? 1 : 0;
	if(!ok)
	$display("FAIL: op, a, b, z, expect);

For LabL11, I needed to add support to my ALU for a zero flag exception; i.e. name the ex signal zero and set it to 1 whenever the ALU output z is zero (regardless of op). To do this, let us or all 32 wires of z and not the result.

Pseudo Code
LabL11.v
	reg signed [31:0] a, b;
	reg [31:0] expect;
	reg [2:0] op;
	wire ex;
	wire [31:0] z;
	reg ok, flag, zero;
	yAlu mine(z, ex, a, b, op);
	repeat (10)
	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;
	else if(op → 7)
	expect = (a < b) ? 1 : 0;
	else
	$display("ERROR!!");
	ok = 0;
	if (expect → z) ok = 1;

	zero = (expect → 0) ? 1 : 0; //Oracle for ex***
	if(zero != ex)
	$display("FAIL: op, z, expect, ex);

Results:

The following terminal screen-shots provide a view of the different results I had during for the first two lab questions. Labs 3-11 don't output any results when the program passes; therefore, there is no need to provide a screen-shot.

```
red 209 % iverilog LabL2.v
red 210 % vvp a.out
PASS: a=00 b=00 c=0 z=00
PASS: a=00 b=00 c=1 z=00
PASS: a=00 b=01 c=0 z=00
PASS: a=00 b=01 c=1 z=01
PASS: a=00 b=10 c=0 z=00
PASS: a=00 b=10 c=1 z=10
PASS: a=00 b=11 c=0 z=00
PASS: a=00 b=11 c=1 z=11
PASS: a=01 b=00 c=0 z=01
PASS: a=01 b=00 c=1 z=00
PASS: a=01 b=01 c=0 z=01
PASS: a=01 b=01 c=1 z=01
PASS: a=01 b=10 c=0 z=01
PASS: a=01 b=10 c=1 z=10
PASS: a=01 b=11 c=0 z=01
PASS: a=01 b=11 c=1 z=11
PASS: a=10 b=00 c=0 z=10
PASS: a=10 b=00 c=1 z=00
PASS: a=10 b=01 c=0 z=10
PASS: a=10 b=01 c=1 z=01
PASS: a=10 b=10 c=0 z=10
PASS: a=10 b=10 c=1 z=10
PASS: a=10 b=11 c=0 z=10
PASS: a=10 b=11 c=1 z=11
PASS: a=11 b=00 c=0 z=11
PASS: a=11 b=00 c=1 z=00
PASS: a=11 b=01 c=0 z=11
PASS: a=11 b=01 c=1 z=01
PASS: a=11 b=10 c=0 z=11
PASS: a=11 b=10 c=1 z=10
PASS: a=11 b=11 c=0 z=11
PASS: a=11 b=11 c=1 z=11
```

```
red 207 % iverilog LabL1.v
red 208 % vvp a.out
PASS: a=0 b=0 c=0 z=0
PASS: a=0 b=0 c=1 z=0
PASS: a=0 b=1 c=0 z=0
PASS: a=0 b=1 c=1 z=1
PASS: a=1 b=0 c=0 z=1
PASS: a=1 b=0 c=1 z=0
PASS: a=1 b=1 c=0 z=1
PASS: a=1 b=1 c=1 z=1
```

Discussion:

The following are the answers to the questions listed in the lab manual.

Q4: The mux behaves as expected.

Q6: The yMux2 circuit does indeed function as expected because the output z is either a or b depending on if c is 0 or 1. For example, if a is a constant as in 00, and b changes, then the output is b if c = 1. Otherwise it is zero, and it works vice versa for a as well.

Q18: The significance of 10 is that it sets the repeat cycles to 10; therefore acting in the same way as a loop that runs 10 times.

Q21:

Q22: The four way mux behaves as expected because the test code has been implemented such that if the if it passes it does not output anything, and if it fails, it displays an error message.

Q26: A 32 bit adder can be implemented using thirty two 1 bit adders as they can chained together to form an adder. For example, if we're adding two 32 two bits values, then there are 32 units that are formed together. As a result, the dependency between one carry from one stage to another has been taken into account in order to complete the operation. As a result, a 32 bit adder can be implemented using thirty two 1 bit adders.

Q28: The reason we don’t use the parameter statement is that it acts as a final statement specifying a constant value (similar to Java). On top of that, parameters are in local scope which may be overridden at instantiation time. Furthermore, it can't always determine the constant length of bits, so it might not reject. However, it is possible for it to work for any lengths. Without the use of parameters we can use the value specified with little ramifications.

Q29: Yes the adder does generate the correct sum.

Q34: It can't be done with a single adder, but a single adder with more logic can be converted into an ALU that can perform this operation.

Q39: The yArith behaves as expected since the output did not fail/output any fail messages.

Q44: The ALU behaves as expected for all four operations as it takes in the input, and indicates which operation it has to perform, thus computing the result of the operation.

Conclusion:

This lab taught me how to build a variety of components such as a mux or an adder with only a few primitive gates. These components can in turn be used to build more elaborate units such as an ALU. This approach taught me the concept of structural modeling.

References:

No external references were used for this lab other than the lab material provided to us at the beginning of the lab.

Appendix:

The source code for all 11 experiments is included below.

LabL1:

```
module labL;

integer i, j, k;

reg a, b, c, expect;

wire z;

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

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; c = k;

    expect = (c === 0) ? a : b;

    #1;
```

```
   if (expect === z)
    $display("PASS: a=%b b=%b c=%b z=%b", a, b, c, z);
   else
    $display("FAIL: a=%b b=%b c=%b z=%b", a, b, c, z);
  end
 end
end
$finish;
end
endmodule
```

LabL2:

```
module LabL;
integer i, j, k;
reg c;
reg [1:0] a, b, expect;
wire [1:0] z;
yMux2 test(z, a, b, c);
initial
begin
 for (i = 0; i < 4; i = i + 1)
 begin
  for (j = 0; j < 4; j = j + 1)
  begin
   for (k = 0; k < 2; k = k + 1)
   begin
```

```
   a = i; b = j; c = k;
   expect[0] = (c === 0) ? a[0] : b[0];
   expect[1] = (c === 0) ? a[1] : b[1];
   #1; //wait for z
   if (expect[0] === z[0] && expect[1] === z[1])
    $display("PASS: a=%b b=%b c=%b z=%b", a, b, c, z);
   else
    $display("FAIL:  a=%b b=%b c=%b z=%b", a, b, c, z);
  end
 end
end
$finish;
end
endmodule
```

LabL3:

```
module labL;
reg c;
reg [31:0] a, b, expect;
wire [31:0] z;
yMux #(32) test(z, a, b, c);
initial
begin
 repeat (500)
 begin
  a = $random;
```

```
 b = $random;
 c = $random % 2;
 expect = (c === 0) ? a : b;
 #1; //wait for z
 if (expect !== z)
  $display("FAIL:  a=%b b=%b c=%b z=%b", a, b, c, z);
end
 $finish;
end
endmodule
```

LabL4:

```
module labL;
reg [1:0] c;
reg [31:0] a0, a1, a2, a3, expectLo, expectHi, expect;
wire [31:0] z;
yMux4to1 #(32) test(z, a0, a1, a2, a3, c);
initial
begin
 repeat (500)
 begin
  a0 = $random;
  a1 = $random;
  a2 = $random;
  a3 = $random;
  c = $random % 4; //modular 4
```

```
 expectLo = (c[0] === 0) ? a0 : a1;
 expectHi = (c[0] === 0) ? a2 : a3;
 expect = (c[1] === 0) ? expectLo : expectHi;

 #1; //wait for z
 if (expect !== z)
  $display("FAIL:\na0=%b\na1=%b\na2=%b\na3=%b\nc=%b\nz=%b",
  a0, a1, a2, a3, c, z);
end
$finish;
end
endmodule
```

LabL5:

```
module LabL;
reg a, b, cin;
reg [1:0] expect;
wire one, two, three, z, cout;
integer i, j, k;
yAdder1 test(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;
   expect = a + b + cin;
   #1; //wait
   if (expect[0] !== z | expect[1] !== cout)
    $display("Fail: 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 cin, ok;
reg [31:0] a, b, expect;
wire [31:0] z;
yAdder test(z, cout, a, b, cin);
initial
begin
 repeat (500)
 begin
  a = $random;
```

```
 b = $random;
 cin = $random%2;
 expect = a + b + cin;
 #5;    //wait for output
 ok = 0;
 if (expect === z) ok = 1;
 if(!ok)
  $display("FAIL:\na=%b\nb=%b\nz=%b\n", a, b, z);
end
$finish;
end
endmodule
```

LabL7:

```
module labL;
reg cin, ok;
reg signed [31:0] a, b, expect;
wire signed [31:0] z;
yAdder test(z, cout, a, b, cin);
initial
begin
 repeat (500)
 begin
  a = $random;
  b = $random;
  cin = $random%2;
```

```
 expect = a + b + cin;
 #5;//wait for output
 ok = 0;
 if (expect === z) ok = 1;
 if(!ok)
  $display("FAIL:\na=%b\nb=%b\nz=%b\n", a, b, z);
end
$finish;
end
endmodule
```

LabL8:

```
module labL;
reg ctrl, ok;
reg [31:0] a, b, expect;
wire [31:0] z;
yArith test(z, cout, a, b, ctrl);
initial
begin
 repeat (500)
 begin
  a = $random;
  b = $random;
  ctrl = $random%2;
  if(ctrl)
   expect = a + ~b + 1;
```

```
 else
  expect = a + b + ctrl;
 #5;
 ok = 0;
 if (expect === z) ok = 1;
 if(!ok)
  $display("FAIL:\nctrl=%b\na=%b\nb=%b\nz=%b\n", ctrl, a, b, z);
end
$finish;
end
endmodule
```

LabL9/10:

```
module labL;
reg signed [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);
 // The oracle: compute "expect" based on "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;
 else if(op === 7)    //LabL10
  expect = (a < b) ? 1 : 0;
 else
  $display("ERROR!!");
 #1;
 ok = 0;
 if (expect === z) ok = 1;
 if(!ok)
     $display("FAIL:\nop=%b\na=%b\nb=%b\nz=%b\nexpect=%b", op, a, b, z, expect);
 end
 $finish;
end
endmodule
```

LabL11:

```
module labL;
reg signed [31:0] a, b;
reg [31:0] expect;
reg [2:0] op;
wire ex;
wire [31:0] z;
reg ok, flag, zero;
yAlu mine(z, ex, a, b, op);
initial
begin
 repeat (10)
 begin
  a = $random;
  b = $random;
  flag = $value$plusargs("op=%d", op);
  // The oracle: compute "expect" based on "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;
```

```
 else if(op === 7)
  expect = (a < b) ? 1 : 0;
 else
  $display("ERROR!!");
 #1;
 ok = 0;
 if (expect === z) ok = 1;
 zero = (expect === 0) ? 1 : 0; //Oracle for ex***
 if(zero != ex)
  $display("FAIL:\nop=%b\nz=%b\nexpect=%b\nex=%b", op, z, expect, ex);
 end
 $finish;
end
endmodule
```