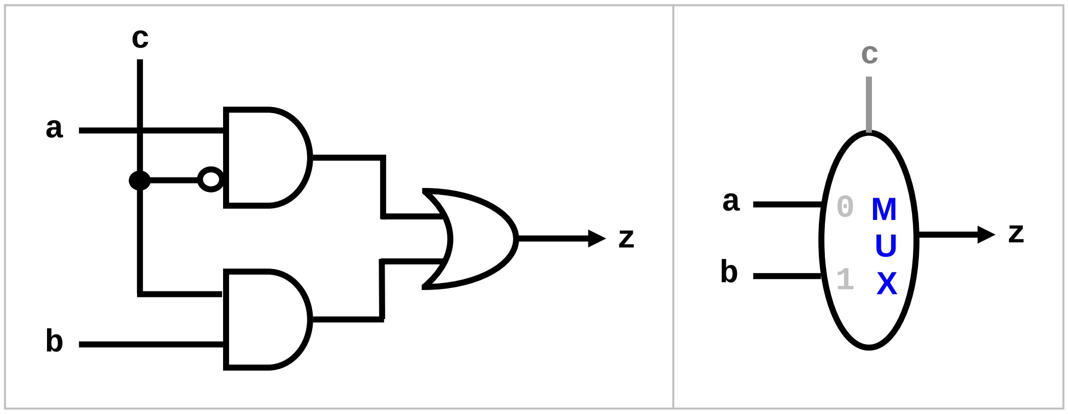
LAB L

Hardware Building Blocks

Perform the following groups of tasks:

LabL1.v

1. In a previous lab we created the **2-to-1 mux** shown in the left part of the figure below and found that it acts as an `if` statement.



In this lab we will use abstraction to encapsulate this circuit as a reusable component so that it can be used by someone who doesn't know *how* it works internally. Let us adopt the diagram shown in the right part of the figure as an abstraction of *what* this circuit does: a box with two doors, labelled 0 and 1, and a *control* input **`c`**. The control is so named because it actually controls the circuit: if it is =0 then Door-0 would open and **`z`** would be **`a`**. Otherwise **`z`** would be **`b`**. We will call the component **`yMux1`**.

2. Create the program yMux1.v as follows:

```
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
```

Compare this with the testing modules we wrote earlier. Note that the circuit's ports are listed in the `module` statement and that a special declaration is used to indicate their in or out status. Note also the absence of any `initial` block. In a way, this module to a testing module is like a library class to an app with a `main` method.

3. Create the program LabL1.v that instantiates and tests `yMux1`. Treat this component the same as a built-in one. You can do manual testing, via command-line arguments, or an exhaustive, three-nested-loop test. Compile your program using the command:

```
iverilog LabL1.v
```

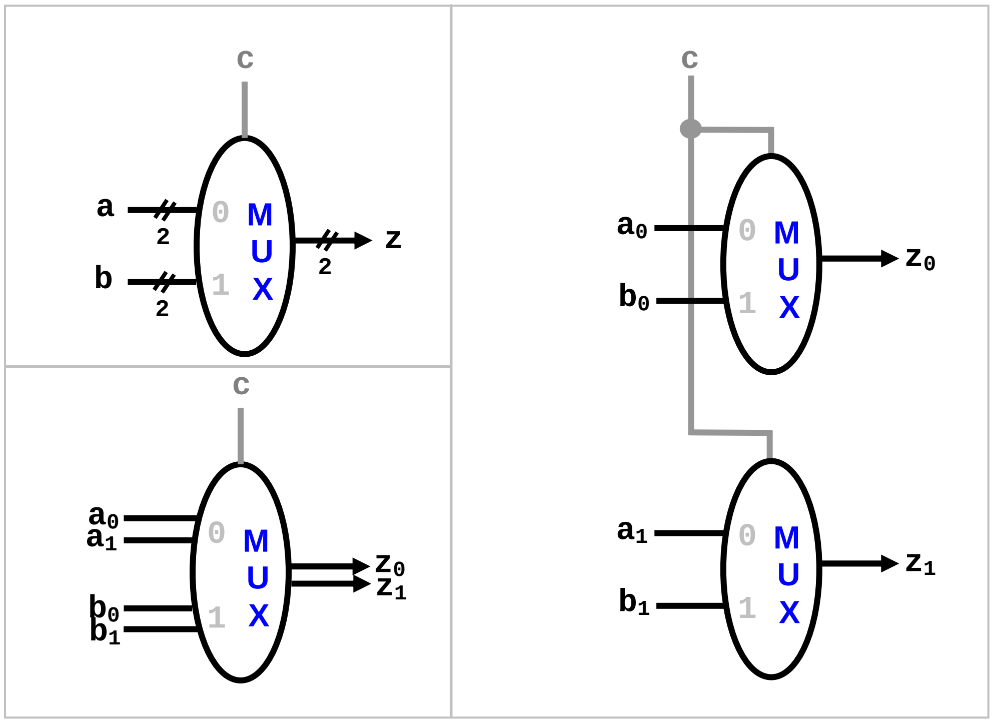
This command searches for a module that defines `yMux1` and compiles it, together with LabL1.v, to produce a single executable file named `a.out`. Alternatively, you can specify the names of the needed modules explicitly on the command line:

```
iverilog LabL1.v yMux1.v
```

4. Run your program and ensure that the mux behaves as expected.

LabL2.v

5. We seek to enhance our mux so it can handle 2-bit busses instead of 1-bit wires, as shown in the upper left diagram in the figure below. The control input, `c`, is still 1-bit but the two data inputs and the output have become 2-bit each. The diagram in the lower left side of the figure shows the wires explicitly. Call the new mux `yMux2`.



The right side of the figure shows how to build `yMux2` using two `yMux1`'s.

6. Argue that the shown circuit of **yMux2** does indeed function as expected; i.e. its output z is indeed either **a** or **b** depending on whether **c** is 0 or 1.

7. Create the program yMux2.v as follows:

```
module yMux2(z, a, b, c);
output [1:0] z;
input [1:0] a, b;
input c;

yMux1 upper(z[0], a[0], b[0], c);
yMux1 lower(z[1], a[1], b[1], c);

endmodule
```

8. Create the program LabL2.v that instantiates and tests **yMux2**. You can conduct manual testing, via command-line arguments or an exhaustive one using three-nested-loop test. Note, however, that the input space has become larger now: there are 4 possible values for **a**, 4 for **b**, and 2 for **c**, for a total of 32.

9. Compile and run your program and ensure that the mux behaves as expected.

LabL3.v

10. Our implementation in yMux2.v is correct but is not scalable because it has as many instantiation lines as there are wires in the bus. Our next task is to design a 32-bit 2-to-1 mux and it would be cumbersome to write 32 mux instantiation lines. In order to overcome this, we switch to *array-based instantiation:* replace the two **yMux1** instantiations with:

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

Verilog auto replicates this statement so that the code becomes equivalent to the two instantiations above. Note that **a** is replicated as **a[0]** and **a[1]** whereas **c**, which is declared as a single bit, is replicated as copies of itself. Save the program as yMux.v

11. And to further the extensibility of our implementation, let us localize the value of the bus size rather than keep it hard coded in many places. The **parameter** statement (which is similar to, but more powerful than, Java's `final` statement) achieves this:

```
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
```

This is our final implementation of the **2-to-1 mux**. The bus width in it is set to 2 but, as we shall see next, this implementation works as-is for *any* bus width.

12. The power of the **parameter** statement lies in the fact that a client of a component can *change the value of the parameter upon instantiation*. We can instantiate a 64-bit, 2-to-1 mux, for example, using the statement:

    ```
    yMux #(64) my_mux(z, a, b, c);
    ```

    The syntax `#(n)` means the parameter value is to be set to `n`. If more than one parameter is involved, use the syntax: `#(.NAME(n))`. For example:

    ```
    yMux #(.SIZE(64)) my_mux(z, a, b, c);
    ```

    A third, more verbose but quite explicit, syntax is available:

    ```
    defparam my_mux.SIZE = 64;
    yMux my_mux(z, a, b, c);
    ```

    The above three variations are equivalent.

13. Save LabL2.v as LabL3.v and modify it so it instantiates and tests `yMux` using a bus width of 32. Remember to change all the [1:0] declarations to [31:0], and the `yMux2` instantiation to `yMux`. And when you instantiate `yMux`, use any of the above three syntax variations to set the `SIZE` parameter to 32.

14. Compile and run LabL3. It should work exactly as before.

15. The input space for LabL3 is so large (=2^{65}) that it is practically impossible to do an exhaustive test. We therefore switch to random testing. The `$random` system task returns a randomly chosen 32-bit integer. Here is one approach:

    ```
    repeat (10)
    begin
       a = $random;
       b = $random;
       c = $random % 2;
       #1;

       // compare z with the expected output

    end
    ```

    Note that since `c` is single-bit, we used the `%` operator together with the `$random` task to generate random 0/1 values.

16. Complete the above code by adding an oracle. Have the program output PASS or FAIL depending on whether `yMux` behaves as expected or not.

17. Compile and run LabL3.

18. What is the significance of `(10)` in the code above? Change it to (500) and modify the testing code so it does not output anything except if a test fails.

LabL4.v

19. We like 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`. For example, if `c` is 2 (i.e. 10 in binary) then the circuit's output `z` must be `a2`. Call it yMux4to1.v.

20. Rather than reinventing the wheel, we can benefit from our existing 32-bit, 2-to-1 yMux. This circuit claims to have the desired functionality:

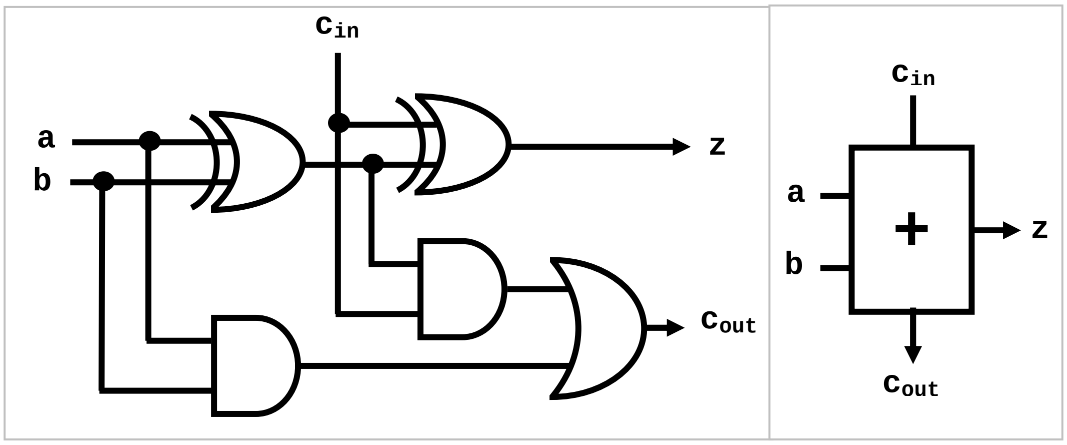
```
module yMux4to1(z, a0,a1,a2,a3, c);
parameter SIZE = 2;
output [SIZE-1:0] z;
input [SIZE-1:0] a0, a1, a2, a3;
input [1:0] c;
wire [SIZE-1:0] zLo, zHi;

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
```

21. Draw the circuit's diagram and use it to argue that this implementation does indeed have the correct functionality of a 4-to-1 mux.

22. Create LabL4.v so it instantiates and tests yMux4to1. Use random testing to create various test cases and an oracle to verify the functionality. Does the four-way mux behave as expected?

LabL5.v

23. In a previous lab we created the **full adder** shown in the left part of the figure below and found that it computes the sum $a+b+c_{in}$ in $\{\{c_{out}\},z\}$.



We now like to use abstraction to encapsulate this functionality. To that end, we re-package the adder as a component named `yAdder1` and having the block diagram shown in the right part of the figure (it operates on 1-bit wires and thus the `1` suffix).

24. Create the program yAdder1.v as follows:

```
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
```

25. Create the program LabL5.v that instantiates and tests **`yAdder1`**. Note that the input space is small: two possibilities for **`a`**, two for **`b`**, and two for **`cin`**, for a total of 8. It is therefore reasonable to use exhaustive testing. Set up the test so that an output is generated only if the sum **`a+b+cin`** has a MSb that is different from **`cout`** or a LSb that is different from **`z`**.

LabL6.v

26. We seek to build a **32-bit adder**, a circuit that takes two 32-bit busses, **`a`** and **`b`**, and a single-bit **`cin`** (which we will set to zero), and produces their sum **`z`** and an overall carry **`cout`** (which is the carry out from the MSb). Take a moment to visualize building such an adder. Can it be done using thirty two 1-bit full adders?

27. Here is a skeleton for yAdder.v. Complete its development.

```
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[1] = out[0];
```

28. When we built the 2-to-1 mux, we used the **`parameter`** statement so that the resulting mux can work for *any* size of the input; i.e. the two data inputs can be 1-bit wires, 2-bit busses, 64-bit busses, or indeed any size set upon instantiation. In the case of the full adder, however, we built one for 1-bit inputs and one for 32-bit inputs. Why? Why not use the **`parameter`** statement so the adder can work with any size input? Does this have any ramification vis-à-vis the latency of the built component?

29. Create the program LabL6.v that instantiates and tests **`yAdder`**. Use random testing for **`a`** and **`b`** and fix **`cin`** to zero. Let us avoid the signed/unsigned issue by not testing **`cout`**; i.e. we just ensure that the obtained sum is correct. Here is a template:

```
...
reg [31:0] expect;
reg ok;
...

initial
begin
   ...
   expect = a + b + cin;
   ok = 0;
   if (expect === z) ok = 1;
   ...
```

Does the adder generate the correct sum?

LabL7.v

30. Note that this adder adds bits and, hence, is not concerned with our interpretation of the inputs as signed or unsigned integers (or anything else). The answer it produces is correct regardless of interpretation. Issues such as signed or unsigned overflows, for example, can be settled by examining the circuit's inputs and outputs; they have no bearing on how the circuit works.

31. To verify the above remark, let us re-test our adder by interpreting the random inputs as signed integers. We do that by adding the keyword `signed` after each `reg` and `wire` declaration.

32. Save the program LabL6.v as LabL7.v and modify it to show signed integers. All the multi-bit declarations need to be changed:

```
module labL;
reg signed [31:0] a, b;
reg signed [31:0] expect;
reg cin;
...
```

33. Run the LabL7 test. Note that the *same* randomly-generated test cases will appear but they are now interpreted as signed integers. The correctness of the adder circuit is independent of interpretation.

LabL8.v

34. We seek to enhance our **32-bit adder** so it can also **subtract**. At first glance, you may think this will require considerable change since; after all, addition and subtraction involve distinct operations. Recall, however, the two's complement identity:

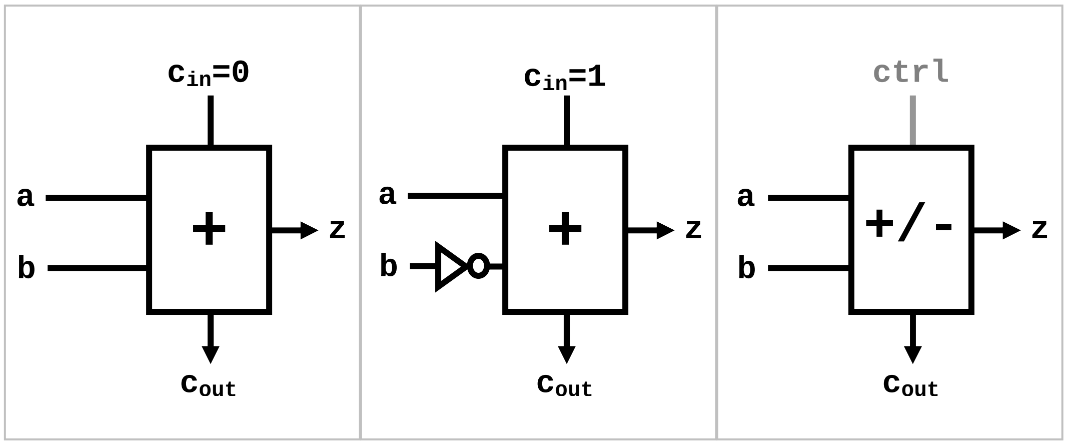
```
-x = (not x) + 1
```

Let us use this identity to rewrite subtraction as addition:

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

This implies that we can subtract by adding twice. First we add **a** and **not b** and then we add 1 to the sum. This *seems* to imply we need to instantiate *two* adders in order to build a subtractor. Can it be done with just *one* adder?

35. Recall the origin of the **cin** signal in the adder. We included it in the design in order to allow the carry to propagate from one bit position to the next. But we don't need it for bit number 0. That is why we grounded it (set it to 0) when we tested the adder. But what if we set it to 1 instead? This will effectively add 1 "for free"! The figure below captures the idea: the left part shows an adder and the middle part shows a subtractor. What we seek is to build the combined adding/subtracting circuit shown in the right part of the figure. Let us call it **yArith**. Its **z** output is **a+b** if **ctrl=0**, and it is **a-b** if **ctrl=1**.



36. Create the program yArith.v as follows:

```
module yArith(z, cout, a, b, ctrl);
// add if ctrl=0, subtract if ctrl=1
output [31:0] z;
output cout;
input [31:0] a, b;
input ctrl;
wire[31:0] notB, tmp;
wire cin;

// instantiate the components and connect them
// Hint: about 4 lines of code

endmodule
```

37. Create the program LabL8.v that instantiates and tests **yArith**.

38. Use random testing for **a** and **b** and **ctrl**, with **ctrl** generated as follows:

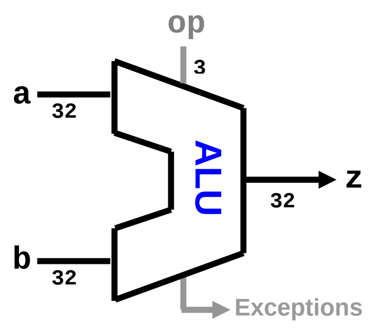
```
ctrl = $random % 2;
```

39. Does the **yArith** behave as expected? Make sure you switch the operation of the oracle based on the randomly generated value of **ctrl**.

LabL9.v

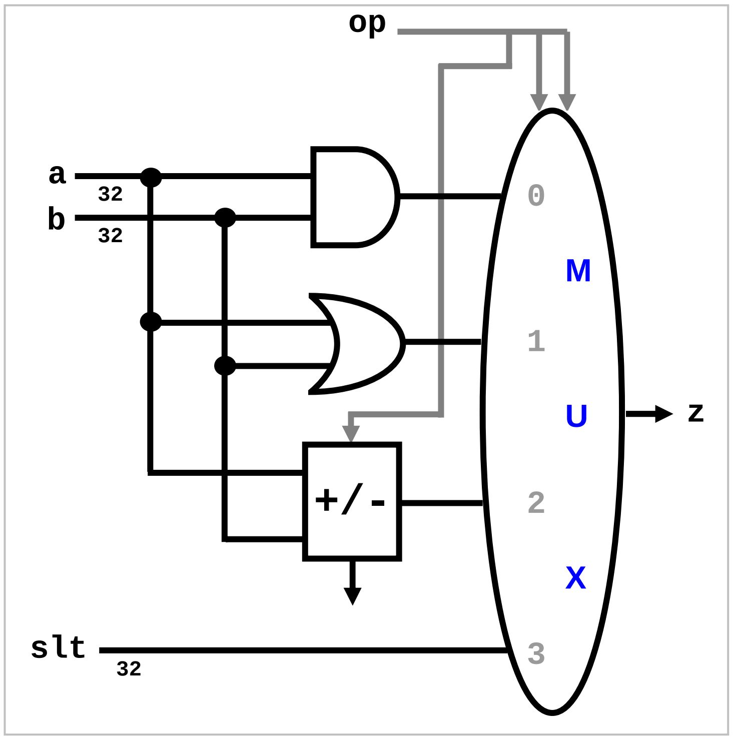
40. The last component that we like to build is the arithmetic-logic unit, the **ALU**, whose diagram is shown below. This is a versatile unit that performs a variety of operations based on a control input **op** (see the table in the diagram). In addition to **z**, the ALU also generates a number of 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.

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



41. Let us build this unit in increments. We will for now ignore the **slt** operation and not support any exception.

42. Building the ALU is easier than one might initially think because we already have all the needed components. We have built-in components for the **and** / **or** operations and we have just built own **yArith**. Hence, all we need is to put these components together and select the output based on **op**, as outlined in this diagram:



As you can see in the diagram, we used the most significant bit of **op**, i.e. **op[2]**, as a control signal for the adder / subtractor. The other two bits, i.e. **op[1:0]**, are used to control the mux—to select the component whose output is to emerge as the output of the ALU.

43. Create the program yAlu.v as follows:

```
module yAlu(z, ex, a, b, op);
input [31:0] a, b;
input [2:0] op;
output [31:0] z;
output ex;

assign slt = 0;  // not supported
assign ex = 0;   // not supported
// instantiate the components and connect them
// Hint: about 4 lines of code
```

44. Create the program LabL9.v that instantiates and tests **yAlu**. Use random testing for **a** and **b** and use a command-line argument to read **op**. Here is a template:

```
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);

    // The oracle: compute "expect" based on "op"

    #1;

    // Compare the circuit's output with "expect"
    // and set "ok" accordingly


    // Display ok and the various signals

    $finish;
end

endmodule
```

Does the ALU behave as expected for all four operations? If not, debug your circuit for the ALU (perhaps by adding $display or $monitor statements to see intermediate signals) until the problem is isolated and fixed.

LabL10.v

45. We like to add support for **`slt`** in our ALU. Recall that this instruction assumes that its operands are signed integers. Its logic can be represented as follows:

    ```
    slt = (a < b) ? 1 : 0;
    ```

    Hence, all we need to do is determine whether the signed value in **`a`** is less than that in **`b`**. This seems trivial: why don't we subtract the two and check the sign? This is not always correct. In computing **`a-b`**, it could be that **`a`** is positive and very large (say 2 billion) and **`b`** is negative and also very large in magnitude (say minus 2 billion). The subtraction leads to an answer which does not fit in 32 bits as a signed integer and, hence, we cannot rely on the answer.

46. We observe that **`a-b`** will overflow the signed range only if **`a`** is positive and **`b`** is negative, or if **`a`** is negative and **`b`** is positive. But in these two cases it is obvious which value is less: the negative one!

47. The above leads to the following algorithm for supporting **`slt`**:

    ```
    if (a and b have different signs)
       slt = 1 if a < 0 else 0

    else

       slt = 1 if (a-b) < 0 else 0
    ```

48. The above can be rewritten as follows:

    ```
    if (a[31] != b[31])
       slt = a[31]
    else
       slt = (a-b)[31]
    ```

49. But how do we generate the following condition via hardware?

    ```
    (a[31] != b[31])
    ```

    Answer: We generate this 0/1 signal using **`xor`**:

    ```
    xor(condition, a[31], [31]);
    ```

50. Re-factor your yAlu.v so that it supports **`slt`** through the above algorithm. Note that **`slt`** is a 32-bit signal and that the algorithm determines whether its least significant bit is 0 or 1 signal. The upper bits, i.e. bits 1 through 31, are 0 in both cases. Hence, the new yAlu module would set **`slt`** along the following lines:

```
module yAlu(z, ex, a, b, op);
input [31:0] a, b;
input [2:0] op;
output [31:0] z;
output ex;

assign slt[31:1] = 0;  // upper bits are always 0
assign ex = 0;         // not supported for now

// instantiate a circuit to set slt[0]
// Hint: about 2 lines of code

// Same code as before
```

51. Save your LabL9.v as LabL10.v and add an oracle for `slt`:

```
else if (op == 3'b111)
     expect = (a < b) ? 1 : 0;
```

Note that you must declare `a` and `b` as `signed` for the comparison (`a < b`) to work as expected in the oracle.

LabL11.v

52. We like to add support 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. Think about this algorithm and argue that it leads to an answer that is 1 only if all 32 bits of `z` are 0.

53. You don't have to do this or'ing sequentially. Instead, either use the unary reduction operator or proceed in a merge-sort manner and benefit from array instantiation:

```
or or16[15:0] (z16, z[15:0], z[31:16]);
or or8[7:0]  (z8, z16[7:0], z16[15:8]);
...
...
```

The first line above or's the left half of `z` with its right half and culminates in a 16-bit bus. The second line or's the two halves of this bus and culminates in an 8-bit bus. Complete the above circuit. *Hint: 4 more lines are needed.*

54. Save your LabL10.v as LabL11.v and add an oracle for `zero`:

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

55. In order to trigger test cases in which the ALU output is indeed zero, make the two operands occasionally equal supply `111` (subtraction) for `op`. Here is one way :

```
a = $random;
b = $random;
tmp = $random % 2;
if (tmp == 0) b = a;
```

cpu.v

56. The components that you created in this lab will be needed in later labs. Rather than keeping them in separate files, and be forced to copy them one by one later, we like to combine them all in one file named **cpu.v**. Use your operating system commands, or your editor, to concatenate all seven components into one file. For example, here is one way to do this under Linux:

```
cat yMux1.v yMux.v yMux4to1.v > cpu.v
cat yAdder1.v yAdder.v >> cpu.v
cat yArith.v yAlu.v >> cpu.v
```

57. To verify that your library was created correctly, create a new directory and copy to it `cpu.v` and any testing module that uses it such as `LabL9.v`. After wards, switch to the newly created directory and issue the command:

```
iverilog LabL9.v cpu.v
```

    Note that in addition to specifying the name of the tester module, you also need to specify the library filename (we did not need to do that before because we had one component per file and the two had the same name).

58. Make sure you keep a copy of `cpu.v`. It will be used for following labs.

LAB L

Notes

- This lab demonstrates that, starting with a few primitive gates, one can build a variety of components such as a mux or an adder. These components can in turn be used to build more elaborate units such as the ALU. This approach to constructing circuits is known as **structural modeling**.

- In structural modeling, a component module *never* uses operators, such as `+` or `<`, to implement a functionality. It only uses instantiation and `assign`; i.e. it implements its functionality by merely creating and connecting subcomponents.

- You *must* use structural modeling in all your component modules.

- The components built in this lab are known as **combinational**. The key characteristic of a combinational component is that it has no memory—its output depends only on its input. A **sequential** component, on the other hand, does have memory—it stores something, a state, and its output depends not only on its input but also on is state. We will introduce sequential components in the next lab.

- Some ALU implementations generate a `carry` flag exception. This 1-bit signal is 1 if a carry was generated out of the MSb of the arithmetic unit. This flag is quite trivial to implement: simply feed the `cout` output of `yArith` to the `ex` output of `yAlu`.

- Some ALU implementations generate an `overflow` flag exception. This 1-bit signal is 1 if the operands of `yArith` are interpreted as signed integers and the result of the addition or subtraction does not fit in the representation size (e.g. requires more than 32 bits as a signed integer). Overflow does not occur if the two operands have different signs and we are adding or have the same sign and we are subtracting.

- Some ALU implementations generate a `zero` flag exception. This 1-bit signal is 1 if the output `z` is 0 (i.e. all its bits are 0) and is 0 otherwise. It is very useful after a subtraction to detect equality.

- In order to accommodate multiple exceptions, we can treat `ex` (or `zero`) as a multi-bit signal; e.g. `ex[0]` is a zero flag, `ex[1]` is a carry flag, and so on.

- Unlike a built-in component such as `and`, a user-defined component must have an instance name upon instantiation; i.e. the name is *not* optional.