# Multiplexers (Mux)

## 2-n-1 Multiplexer (Mux)

In this lab we will use abstraction to encapsulate the 2-to-1 multiplexer in Figure 1 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 1 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 `c` is =0 then `Door-0` would open and `z` would be `a`. Otherwise `z` would be `b`. We will call the component `yMux1`.

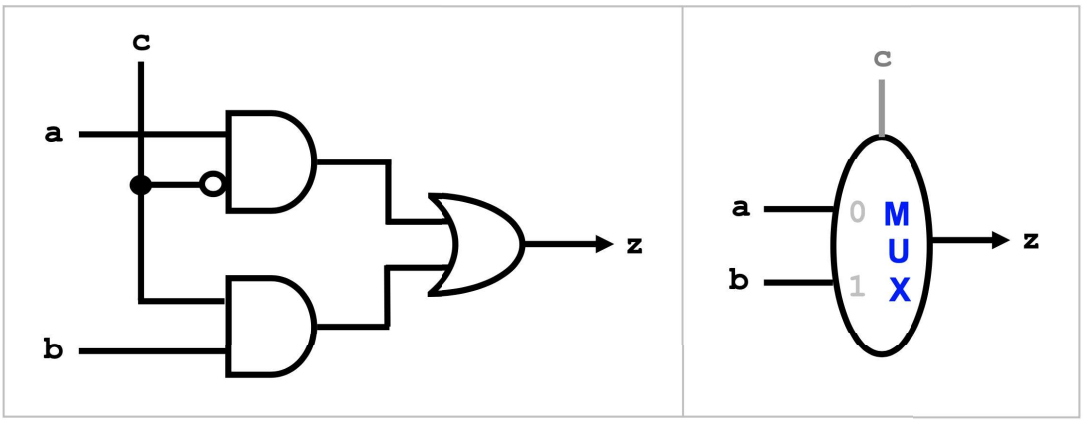



Figure 1: The 2-to-1 Multiplexer The multiplexer act as an if statement: if $c = 0$ then $z$ would be $a$. Otherwise $z$ would be $b$.

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
```

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.

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
```

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

### Combine two Mux's to control a "Bus"

We seek to enhance our `mux` so it can handle `2-bit` buses instead of `1-bit` wires, as shown in the upper left diagram in the Figure 3 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`.

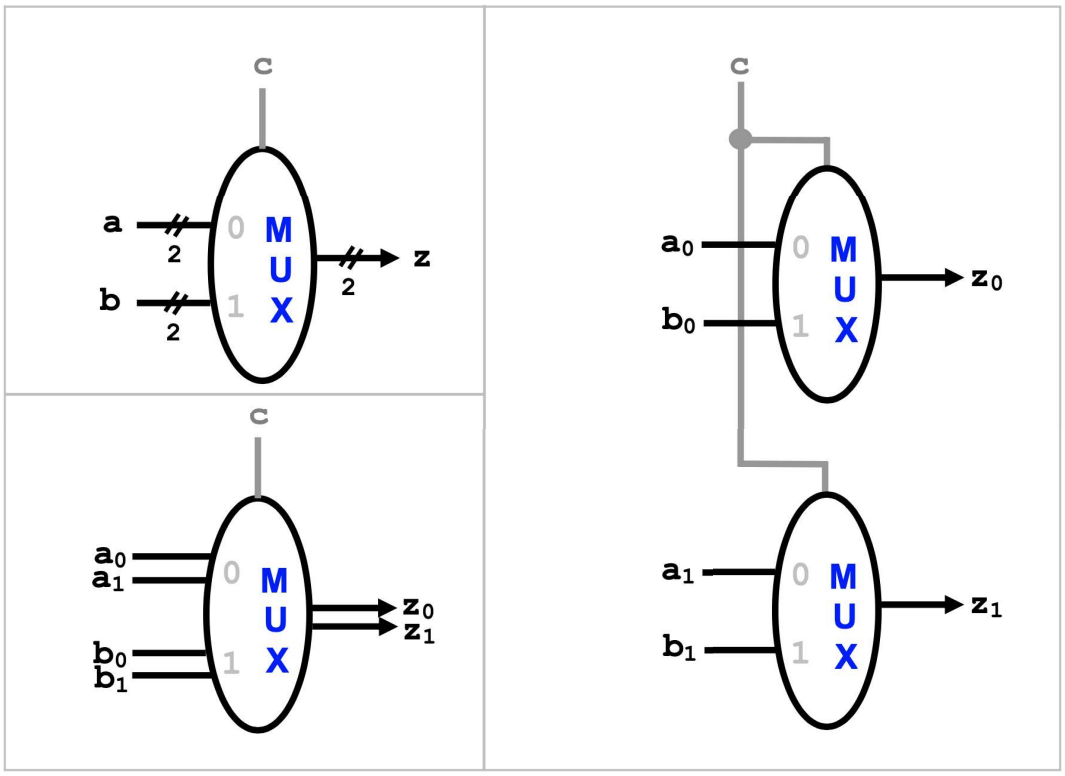



Figure 2: The right side of the figure shows how to build yMux2 using two yMux1's. Combine two Muxes to allow control signal c to shunt the a and b signals (a0, b0, as well as, b0 and b1) to output lines z0 and z1.

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
```

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

### Array-based instant'n multi-wire buses: 32-bit 2to1 Mux

Our implementation in **yMux2.v** is correct but is not scalable because it has as many instantiating 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 instantiating 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** 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.
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. 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`.

Compile and run **LabL3**. It should work exactly as before. 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. Complete the above code by adding an oracle. Have the program output `PASS` or —FAIL— depending on whether yMux behaves as expected or not. Compile and run **LabL3.**

## The 4-to-1 Multiplexor (MUX)

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**.

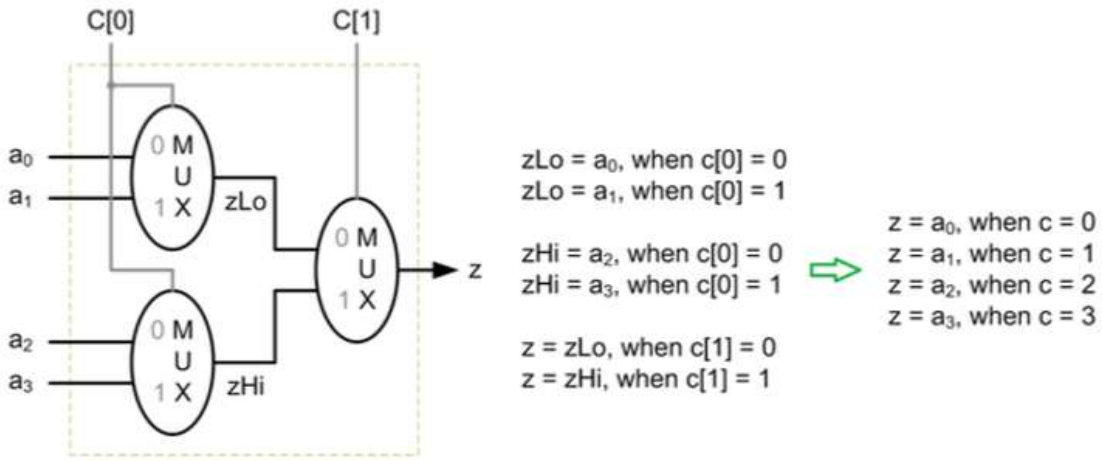



Figure 3: Combining three yMux blocks. The right side is the output analysis. You will need to modify your Verilog model to match this.

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
```

To better understand what's going on consider drawing the circuit's diagram to be confident that above Verilog implementation does indeed have the correct functionality of a 4-to-1 mux.

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?**

## The Full Adder

Create the program **LabL4.v** so that it simulates the circuit in Figure 4.

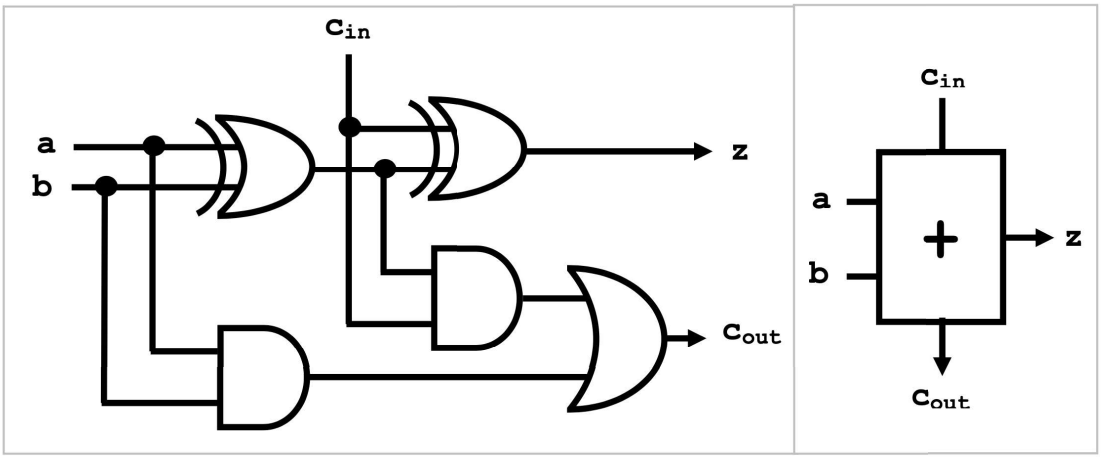



Figure 4: The adder in schematic and symbolic form, complete with inputs (a, b), carry in (input), carry out (output) and summation (z).

Note that the circuit has three `1-bit` inputs and two `1-bit` outputs. It is made up of two `XOR` gates, two and gates, `AND` an `OR` gate. 48. It is up to you to use `assign` or `shared` names to simulate the circuit but make sure you connect its ports correctly. Either way, you may find it helpful to label the wires on the diagram and/or to give them meaningful names.

Note that if $c_{in}$ is zero then $z$ is nothing but the sum of `a` and `b` with a carry of $c_{out}$. That is why this circuit is known as a full adder.

Do an exhaustive test using three nested loops and verify the addition functionality of the circuit. This functionality can be expressed as:

```
expect[0] === z && expect[1] === cout
```

where `expect` is a two-bit vector (i.e. declared as `reg[1:0]`) computed using the statement: `expect = a + b + cin`. We now repackage the adder as a component named **yAdder1** and having the block diagram shown in the right part of the Figure 4 (it operates on 1-bit wires and thus the 1 suffix).

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
```

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 $c_{in}$, 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 $a + b + c_{in}$ has a `MSb` that is different from $c_{out}$ or a `LSb` that is different from z (i.e. a "fail").

## Multi-bit (32-bit) Adder

We seek to build a `32-bit adder`, a circuit that takes two `32-bit` bus, `a` and `b`, and a single-bit $c_{in}$ (which we will set to zero), and produces their sum z and an overall carry $c_{out}$ (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?** Here is the code for **yAdder1.v**.

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

        // inputs
        input [31:0] a, b;
        input cin;

        // interconnects
        wire[31:0] in, out;

        // yAdder1 is defined in yAdder1.v
        yAdder1 mine[31:0](z, out, a, b, in);

        assign in[0] = cin;
        assign in[31:1] = out[30:0];
endmodule
```

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` bus, `64-bit` bus, or indeed any size set upon instantiation.

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

```
      ...
      reg [31:0] expect;
      reg ok;
      ...
      // The "Oracle" testing system
      initial
      begin
         ...
         expect = a + b + cin;
         ok = 0;
         if (expect === z) ok = 1;
         ...
endmodule
```

Does the adder generate the correct sum? 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. To check, 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.

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;
      ...
endmodule
```

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.

## Enhance the 32-bit Adder

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?

Recall the origin of the $c_{in}$ 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 5 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`.

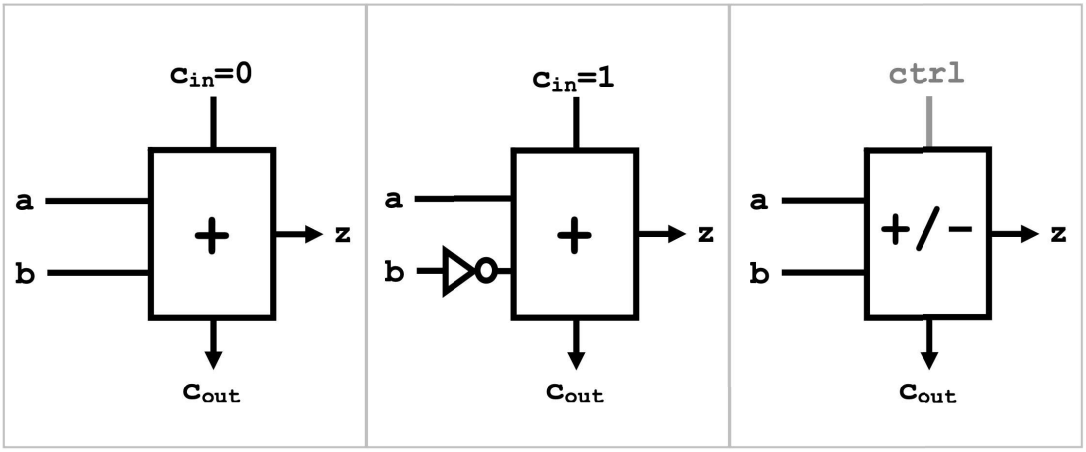



Figure 5: Enhanced Adder

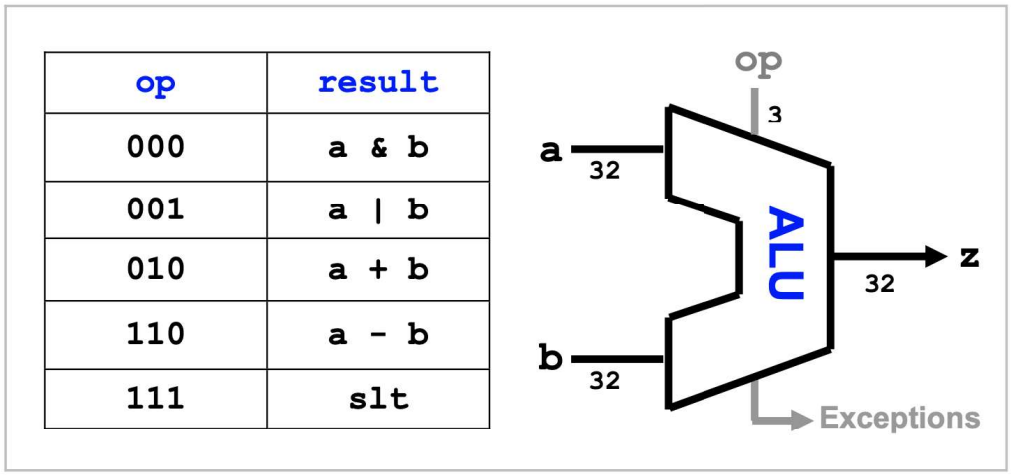
Create the program **yArith.v** as follows:

```
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
```

Create the program **LabL8.v** that instantiates and tests **yArith**. Use random testing for `a` and `b` and `ctrl`, with `ctrl` generated as follows: `ctrl = $random % 2`; Does the **yArith** behave as expected? Make sure you switch the operation of the oracle based on the randomly generated value of `ctrl`.

## The Arithmetic Logic Unit (ALU)

ALU is a versatile unit that performs a variety of operations based on a control input op (see the table in the Figure fig:ALU). 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 |



Figure 6: The Arithmetic Logic Unit (ALU)

Let us build this unit in increments. We will for now ignore the slt operation and not support any exception. We have all the needed components to build an ALU: 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 the Figure 7.

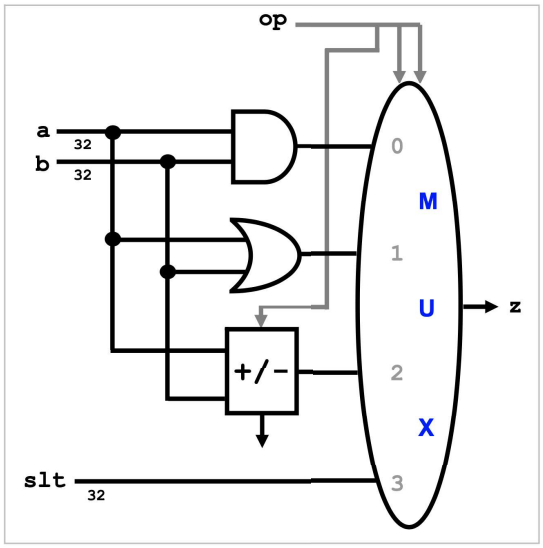




Figure 7: The Arithmetic Logic Unit (ALU) diagram

As you can see in the diagram in Figure 7, 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`.

Create the program **yAlu.v** as follows:

```
module yAlu(z, ex, a, b, op);
    // op=000: z=a AND b, op=001: z=a|b, op=010: z=a+b, op=110: z=a-b
    input [31:0] a, b;
    input [2:0] op;
    output [31:0] z;
    output ex;

    wire [31:0] zAnd, zOr, zArith, slt;

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


endmodule
```

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.

## Extending the ALU

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

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!

This 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
```

The above can be rewritten as follows:

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

But how do we generate the `(a[31] != b[31])` via hardware? **ANSWER:** We generate this 0/1 signal using `xor`:

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

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
```

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.

### Zero Flag Exception Support

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.

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

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

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;
```

## The CPU

Rather than keeping the components in separate files, we can 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
```

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).

## 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 sub- traction 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.