## Learning Objectives

1. Learn more Verilog features
2. Build a 32-bit ALU

## Work that needs to be handed in

Implement and test the `alu32` module and its subcomponents in Verilog. These need to be submitted **by the first deadline**:

1. `mux.v`: Complete **Build 4:1 MUX from 2:1 MUXes** on PrairieLearn.
2. `logicunit.v`: Complete **Logic Unit Truth Table to Verilog** on PrairieLearn.

These files need to be submitted **by the second deadline**:

1. `alu1.v`: your 1-bit ALU
2. `alu1_tb.v`: a test bench for your 1-bit ALU
3. `alu32.v`: your 32-bit ALU (hint: see the appendix to learn about writing code generators to help you out)
4. `alu32_tb.v`: a test bench for your 32-bit ALU
5. `generator.cpp` or `generator.py`: generator code for your 32-bit ALU

**Note**: `alu1_tb.v`, `alu32_tb.v`, `generator.cpp` or `generator.py` will not be used for initial grading but may be used for regrade requests

We are building this ALU as a step toward building a MIPS processor. The MIPS processor supports bit-wise logical operations (`and`, `or`, `nor` and `xor` instructions), two's complement addition (`add` instruction) and two's complement subtraction (`sub` instruction) among other operations. You can get more information about MIPS instructions in MIPS Green Sheet. The ALU (as we described in class) should have two 32-bit inputs (A, B) and a 32-bit output. It should perform operations based on its 3-bit control signal as shown in the table below:

| control | Operation | control | Operation |
|---|---|---|---|
| 0 | undefined | 4 | bit-wise AND |
| 1 | undefined | 5 | bit-wise OR |
| 2 | two's complement addition | 6 | bit-wise NOR |
| 3 | two's complement subtraction | 7 | bit-wise XOR |

In addition, it should output three single-bit signals: `overflow, zero, negative`. `overflow` flag is used to indicate invalid result for signed arithmetic calculation. The values of `overflow, zero` and `negative` flags used in branch instructions.

| Signal | Specificaion |
|---|---|
| negative | 1 if the ALU output interpreted as a two's complement number would be negative, 0 otherwise. |
| overflow | 1 if two's complement add or subtract operation caused an overflow, 0 otherwise. (undefined for a logical operation) |
| zero | 1 if the output is equal to zero, 0 otherwise. |

## Compiling, Running and Testing

We have included a `Makefile` in your `Lab2` directory that facilitates compilation. The following examples illustrate its use:

```
make mux4              - compiles the mux4 circuit and runs its test bench
make logicunit         - compiles the logic unit circuit and runs its test bench
make alu1              - compiles the alu1 circuit and runs its test bench
make alu32             - compiles the alu32 circuit and runs its test bench
make example_generator - compiles the example C++ generator
make generator         - compiles your C++ generator for the 32-bit ALU
make clean             - removes all compiled and vcd files
```

## Incremental Testing

The ***wrong*** way to do this assignment: (or any)

Step 1. Write all of the code
Step 2. Compile all of the code (and debug the compiler errors, which is especially fun with iverilog...)
Step 3. Debug all of the code

One ***right*** way to do this assignment: (there are many)

Step 1. Implement the `mux4` module in `mux.v` **(done on PrairieLearn)**.

Step 2. Compile and debug this circuit. We've provided a test bench, `mux4_tb.v`. Given that `mux4` is a combinational circuit and has a small number of inputs, we can **exhaustively** test it (which means give it all combination of inputs, so we can see what it does in all circumstances). To compile and run the testbench, do the following:

```
make mux4
```

In the provided testbench, every 16 cycles we give all combinations of A, B, C, and D. Also, every 16 cycles we change the setting of control, so `out` should mirror `A` the first 16 cycles, `B` the second 16, etc. Use **gtkwave** to be able to quickly visually verify the correctness; trust us, it is faster than trying to verify console output.

```
gtkwave mux4.vcd &
```

Step 3. Implement and test `logicunit` **(done on PrairieLearn)**. Again, since this circuit is small you should exhaustively test it (by adapting the test in `mux4_tb.v`). **Warning:** `logic` is a type in Verilog, so don't try to use that as a module or instance name, otherwise you'll get weird compiler errors.

Step 4. Implement and (exhaustively) test `alu1`. This circuit is a little tricky because you are building a circuit that has to work for all of the bit slices, which is not the same as the one bit adder. Specifically, when the control input is 3, the two-bit output (`carryout`, `out`) should equal the magnitude of ($A + B' +$ carryin), **not** the result of subtraction. *Don't* hardwire carryin to 1 for subtraction inside the `alu1`; we'll do that outside in `alu32`. For now, just manually feed it. It will actually be given value from the control unit when the whole system is built.

Step 5. Implement `alu32`. We've provided a prototype for it in `alu32.v`.

Step 6. Compile and debug the complete design. This module is too big to exhaustively test; with 67 input bits, you'd need to test $2^{67}$ patterns to exhaustively test it (so we won't do that..). Instead, you should test enough of the operations that you are confident that it works. In particular, it is important to include "corner cases" among your tests. Corner or boundary cases are around inputs where errors like off-by-one errors are likely; for our ALU the most obvious corner cases occur right around the point that overflow occurs. To test this corner case, you should test one of the largest additions that doesn't overflow and one of the smallest additions that does overflow, to make sure overflow occurs at the right place.

You are responsible for writing (and submitting) your own test bench for your 32-bit ALU. We've only provided a skeleton in the file `alu32_tb.v`, which is in the git repository. Here are suggested tests:

1. test that each of the operations (add, sub, and, or, nor, xor) work with some input
2. test that subtracting a number from itself gives zero (and raises the zero signal)
3. test that subtracting a very large positive number (*e.g.*, 0x7ffffff0) from a small positive number (*e.g.*, 36) produces the right answer and has negative=1 and overflow=0
4. verify that overflow is asserted when adding two very large positive numbers, when adding two negative numbers of large magnitude, and when subtracting a negative number of large magnitude from a large positive number.

## Appendix: Writing code generators

A useful technique in computing is writing code to write code, resulting in what is called "machine generated code." When implementing regular structures in Verilog, this can be much less error prone than manual copying and editing. As an example, consider this circuit that computes whether a bus is all zeros:

```
input [7:0] in;
wire  [7:1] chain;

or  o1(chain[1], in[1], in[0]);
or  o2(chain[2], in[2], chain[1]); // Note how lines from here to
or  o3(chain[3], in[3], chain[2]);
or  o4(chain[4], in[4], chain[3]);
or  o5(chain[5], in[5], chain[4]);
or  o6(chain[6], in[6], chain[5]);
or  o7(chain[7], in[7], chain[6]); // here are basically the same
not n0(zero, chain[7]);
```

One can write a generator in any language (below is C), run it, and **cut/paste the result into your verilog file**.

```
// This function generates the repeated part of the circuit
int main() {
  for (int i = 2 ; i < width ; i ++) {
    printf("   or o%d(chain[%d], in[%d], chain[%d]);\n", i, i, i, i-1);
  }
  return 0;
}
```

---

You should be able to write a loop (in the language of your choice) to instantiate a 32-bit ALU out of 1-bit ALUs and correctly connect the carry chain