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 in Verilog. When you are finished, these files should be submitted via svn:

1. `alu32.v`: your 32-bit ALU
2. `alu32_tb.v`: a test bench for your 32-bit ALU
3. `example_generator.c` or `example_generator.py`: generator code for your 32-bit ALU

The ALU (as we described in class) should have 2 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	unused	4	bit-wise AND
1	unused	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`.

Signal	Specificaion
negative	1 if the ALU output interpreted as a two's complement number would be negative, 0 otherwise.
overflow	1 if the 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.

New Verilog syntax

Because we're building more elaborate modules in Verilog, we'll be taking advantage of more advanced features of the language. These features are described below.

Defining multi-bit signals:
```
wire [31:0] bus;                // define a 32-bit bus of wires
```
Selection from buses:
```
or o12(out, bus[1], bus[0]);
```
Assignment statements:
```
assign x = bus[7];              // connect wires to other wires
assign wirex[3:0] = bus[12:9];
```
Constants:
```
8'hd7           // 8-bit number with value 215 (0xd7) – hexadecimal
5'b11001        // 5-bit number with value 25 – binary
10'd978         // 10-bit number with value 978 – decimal
```
Define:
```
`define ACONSTANT 3'h2 // create a 3 bit constant
// named "ACONSTANT" and assign to it the value 2
```
Comparison statements:
```
wirex == 4'b1010                // returns a one-bit signal
```

Incremental Testing

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

Step 1. Write all of the code
Step 2. Compile all of the code (and debug the compiler errors)
Step 3. Debug all of the code

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

Step 1. Implement the `mux4` module in alu32.v.
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 run the testbench, do the following:

```
iverilog -o mux4test mux4_tb.v alu32.v
.\mux4test
```

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 test.vcd
```

Step 3. Implement and test `logicunit`. Again, since this circuit is small you should exhaustively test it (by adapting the test in `mux4_tb.v`).
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 (subtraction), the two-bit output (`carryout`, `out`) should equal the magnitude of (`A + B' + carryin`). Don't hardwire carryin to 1 for subtraction inside the `alu1`, we'll do that outside in `alu32`.
Step 5. Implement `alu32`. We've provided a prototype for it (commented out) in `alu32.v`.
Step 6. Compile and debug the complete design. This module is too big to exhaustively test; with 69 input bits, you'd need to test $2i$ 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 your SVN repository. Here are suggested tests:

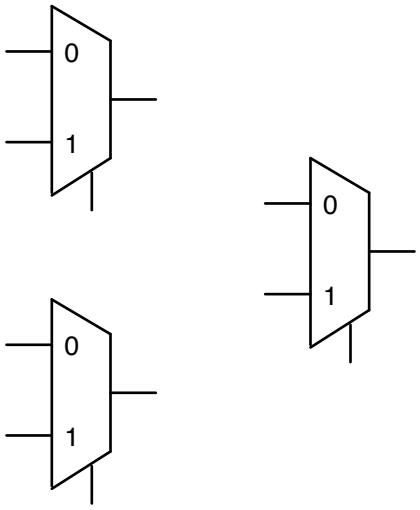
1. testing that each of the operations (add, sub, and, or, nor, xor) work with some input
2. test that subtracting a number from itself gives zero (and raises the zero signal)
3. test that subtracting a very large positive number (*e.g.*, 0x7ffffff0) from a small positive number (*e.g.*, 36) produces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

Building a 4-input Multiplexor

```
// output = A (when control == 0) or B (when control == 1)
module mux2(out, A, B, control);   // a 2-input multiplexor
   output      out;
   input       A, B;
   input       control;
   wire        wA, wB, not_control;

   not n1(not_control, control);
   and a1(wA, A, not_control);
   and a2(wB, B, control);
   or  o1(out, wA, wB);
endmodule // mux2
```

Connect the 2-input multiplexors to implement a 4-input multiplexor:




```
// output = A (when control == 00) or B (when control == 01) or
//          C (when control == 10) or D (when control == 11)
module mux4(out, A, B, C, D, control);   // a 4-input multiplexor
   output      out;
   input       A, B, C, D;
   input [1:0] control;            // <----  multiple bit signal (2-bits)

   // use  control[0]  and   control[1]  as inputs to multiplexors




endmodule // mux4
```

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:

```
// This function generates a circuit (albeit a slow one) to compute whether
// a bus is all zeros.
int
main() {
  int width = 8;

  printf("   input [%d:0] in;\n", width - 1);
  printf("   wire  [%d:1] chain;\n\n", width - 1);

  printf("   or o1(chain[1], in[1], in[0]);\n");
  for (int i = 2 ; i < width ; i ++) {
    printf("   or o%d(chain[%d], in[%d], chain[%d]);\n", i, i, i, i-1);
  }
  printf("   not n0(zero, chain[%d]);\n", width - 1);
  return 0;
}
```

Which when run generates:

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

   or o1(chain[1], in[1], in[0]);
   or o2(chain[2], in[2], chain[1]);
   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]);
   not n0(zero, chain[7]);
```

Write a loop to instantiate your 1-bit ALUs and correctly connect the carry chain: