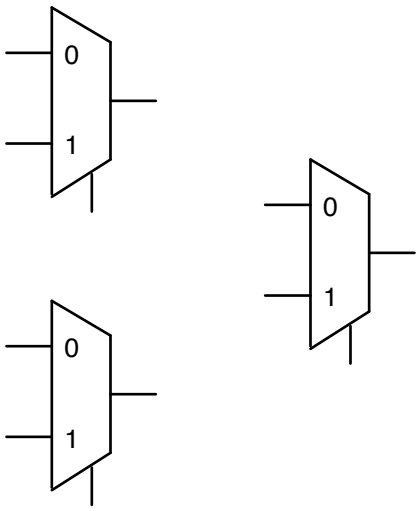
Building a 4-input Multiplexor

```
// 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
```

New Verilog syntax

Draw a circuit for the following module definition. Note that `‘define` and `‘BLUE` use a backtick while `1’b1` uses an apostrophe.

```
`define BLUE 1'b1;

module discussion3(z, x, y);
   output [3:0] z;
   input  [2:0] x;
   input  [1:0] y;
   wire         control;

   or o1(z[0], x[2], x[1], x[0]);

   assign control = x[1:0] == y[1:0];
   mux2 blue_mux(z[1], x[0], `BLUE, control);

   assign z[3:2] = x[2:1];

endmodule // discussion3
```

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;
}
```

Assuming that a 1-bit ALU has the following module interface:

```
module alu1(out, carryout, A, B, carryin, control);
```

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:

Overflow

You need to write examples to check for overflow in two's complement binary addition. For each case, assume 8-bit two's complement binary numbers. Perform the sum using the binary representation of the numbers, but make sure you know how to represent these numbers using the hexadecimal notation.

- Write an example where the sum of two positive numbers results in overflow.

- Write an example where the sum of two negative numbers results in overflow.

- Can overflow occur when summing a positive and a negative number? Justify your answer.