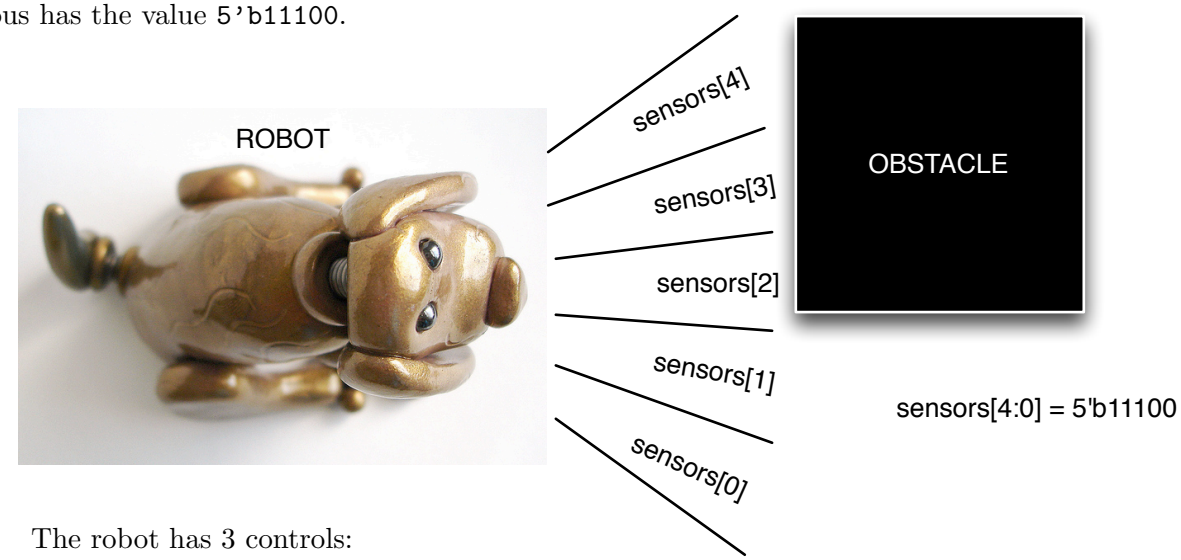
Pre-Lab: Steer the robot!

Consider a mobile robot with a depth sensing camera that it uses for navigation; we're going to build a control circuit for that robot. Specifically, this depth sensor provides five binary digits indicating whether there is an obstacle in each of five portions of the field of view in front of the robot (where the MSB is the leftmost segment and the LSB is the rightmost segment). For example, in the image below there is an obstacle in the leftmost 3/5ths of the robot's view, so the `sensors` bus has the value `5'b11100`.



The robot has 3 controls:

1. `walk`: when this signal is 1, the robot should walk; when 0 the robot should stay where it is.
2. `left`: when this signal is 1, the robot should turn to the left.
3. `right`: when this signal is 1, the robot should turn to the right. `left` and `right` should never both be 1.

Your job is to build a circuit that controls the robot, given the following rules:

1. The robot should walk, unless all three of the middle segments of its view indicate obstacles. If the middle 3 segments show obstacles, the robot should stop (*i.e.*, not walk), so that it doesn't run into anything.
2. The robot will not turn if there are no obstacles straight ahead of it (*e.g.*, if sensors[2] is 0). Otherwise it must turn; note that in some cases the robot is walking and turning.
3. If the robot must turn, it will turn in the direction that appears to be less obstructed. Specifically, there are two sensors bits in each of the left and right directions; it will turn in the direction that has fewer bits set.
4. If both directions seem equally obstructed, it will turn left.

Before your lab section, please write boolean expressions for these signals: *Feel free to define additional boolean variables if that simplfies your implementation.*

walk =

left =

right =

sensors[4]	sensors[3]	sensors[2]	sensors[1]	sensors[0]	walk	left	right
0	0	0	0	0			
0	0	0	0	1			
0	0	0	1	0			
0	0	0	1	1			
0	0	1	0	0			
0	0	1	0	1			
0	0	1	1	0			
0	0	1	1	1			
0	1	0	0	0			
0	1	0	0	1			
0	1	0	1	0			
0	1	0	1	1			
0	1	1	0	0			
0	1	1	0	1			
0	1	1	1	0			
0	1	1	1	1			
1	0	0	0	0			
1	0	0	0	1			
1	0	0	1	0			
1	0	0	1	1			
1	0	1	0	0			
1	0	1	0	1			
1	0	1	1	0			
1	0	1	1	1			
1	1	0	0	0			
1	1	0	0	1			
1	1	0	1	0			
1	1	0	1	1			
1	1	1	0	0			
1	1	1	0	1			
1	1	1	1	0			
1	1	1	1	1			

Note: in cases where there are a lot more 1's in a column than 0's, it is easier to write the inverse of the desired boolean expression and then negate it. For example, if truth table only had 2 zeros (when A=1, B=0, C=1 and when A=1, B=1, and C=0), it could be implemented by writing terms for those zero cases, summing those terms, and negating the sum. It would be written as: (AB'C + ABC')'

Learning Objectives

1. Combination logic design
2. Sequential logic design
3. Register file implementation

Work that needs to be handed in (via SVN)

1. `steering.v`: your steering circuit from the pre-lab in Verilog, using the provided module definition. We've provided a partial test suite in `steering_tb.v`.
2. `word_reader.v`: a character recognition finite state machine described in this handout using the provided module definition.
3. `word_reader_tb.v`: a testbench for testing the word reader. We've provided a shell for this file, but you'll have to design your own tests. *We won't be auto grading your test benches, but the quality of your testing will influence the assignment of parital credit.* i.e.*, no tests, no partial credit.*
4. `rf.v`: a MIPS register file. We've provided the module's interface, shown below.

```
module mips_regfile (rd1_data, rd2_data, rd1_regnum, rd2_regnum,
                     wr_regnum, wr_data, writeenable,
                     clock, reset);

   output [31:0]  rd1_data, rd2_data;
   input [4:0]    rd1_regnum, rd2_regnum, wr_regnum;
   input [31:0]   wr_data;
   input          writeenable, clock, reset;

endmodule // mips_regfile
```

   The register file has 2 32-bit read ports (`rd1_data` and `rd2_data`) which are independently controlled by `rd1_regnum` and `rd2_regnum` and 1 32-bit write port (`wr_data` controlled by `wr_regnum` and `writeenable`). It includes 31 registers; reading register $0 always returns zero. All writes are synchronous, so they should only occur the clock's rising edge.

5. `rf_tb.v`: a testbench for the register file; handed in, not autograded (see `score_keeper_tb.v`). You should test the following functionality (because we will be...):
   - That when you write a register (any of $1-$31), that future reads of that register (from either read port) return the written value and that the contents of no other register is affected.
   - That reading $0 always returns zero even if you try to write to it.
   - That reseting the register file restores all register values to 0.

New Verilog syntax

Because you are getting comfortable with the gate instantiation syntax (*e.g.*, `and a1(out, in1, in2, in3, ...)`), you can graduate to the boolean expression syntax.

Say you wanted to assign the output `X` to the expression: `AB+C'D`

You could do that with the following code:

```
output X;
input  A, B, C, D;
wire   w1, w2, w3;

or  o1(X, w2, w3);
and a1(w2, A, B);
and a2(w3, w1, D);
not n1(w1, C);
```

The good thing about the above implementation is that it is very clear that we're describing hardware and exactly what the gates are. The bad thing about it is that it is much more verbose than the boolean expression.

Verilog also allows us to describe hardware using boolean expressions directly.

```
output X;
input  A, B, C, D;

assign X = (A&B) | ((~C) &D);
```

This code is more concise and is entirely equivalent to the above. **This expression is describing a series of gates. Don't think about this as software; you will confuse yourself.**

Useful syntax:

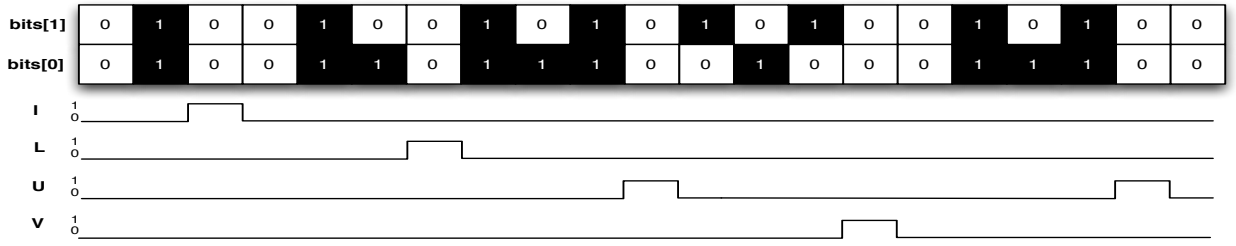
`&` - bit-wise AND `|` - bit-wise OR - bit-wise NOT // the logical not `!` also works for **one bit signals**

This code is useful for the first two parts of this Lab: the combinational design problem and the sequential design problem. **The register file and decoders should be built out of other modules.**

Finite state machine design

You are providing the smarts for a simple device to recognize printed characters. This device is swiped over the text from left to right, and every cycle determines whether the area currently being swiped is shaded on the top or bottom of the character, both, or neither; these are provided to our circuit as a two-bit signal called `bits`. Each character is recognized by observing a sequence of such marks followed by at least one blank space (i.e., a region where there are marks on neither the top or the bottom of the area currently being scanned).

For this assignment, you are being asked to build a finite state machine to recognize three characters (I, L, and U) and 15 points of extra credit is available if you can also recognize a fourth (V). As shown in the figure below, an "I" is recognized when there is a cycle (*e.g.*the first cycle) where both `bits`[1] and [0] are 0, a cycle where both `bits` are 1, followed by another cycle where both are 0. It is in this third cycle, where we know that we've seen the end of a character that we recognized that we assert the `I` signal, indicating that we've seen the I character. Recognizing the L character involves observing the sequence [`2b'00`, `2b'11`, `2b'01`, `2b'00`].



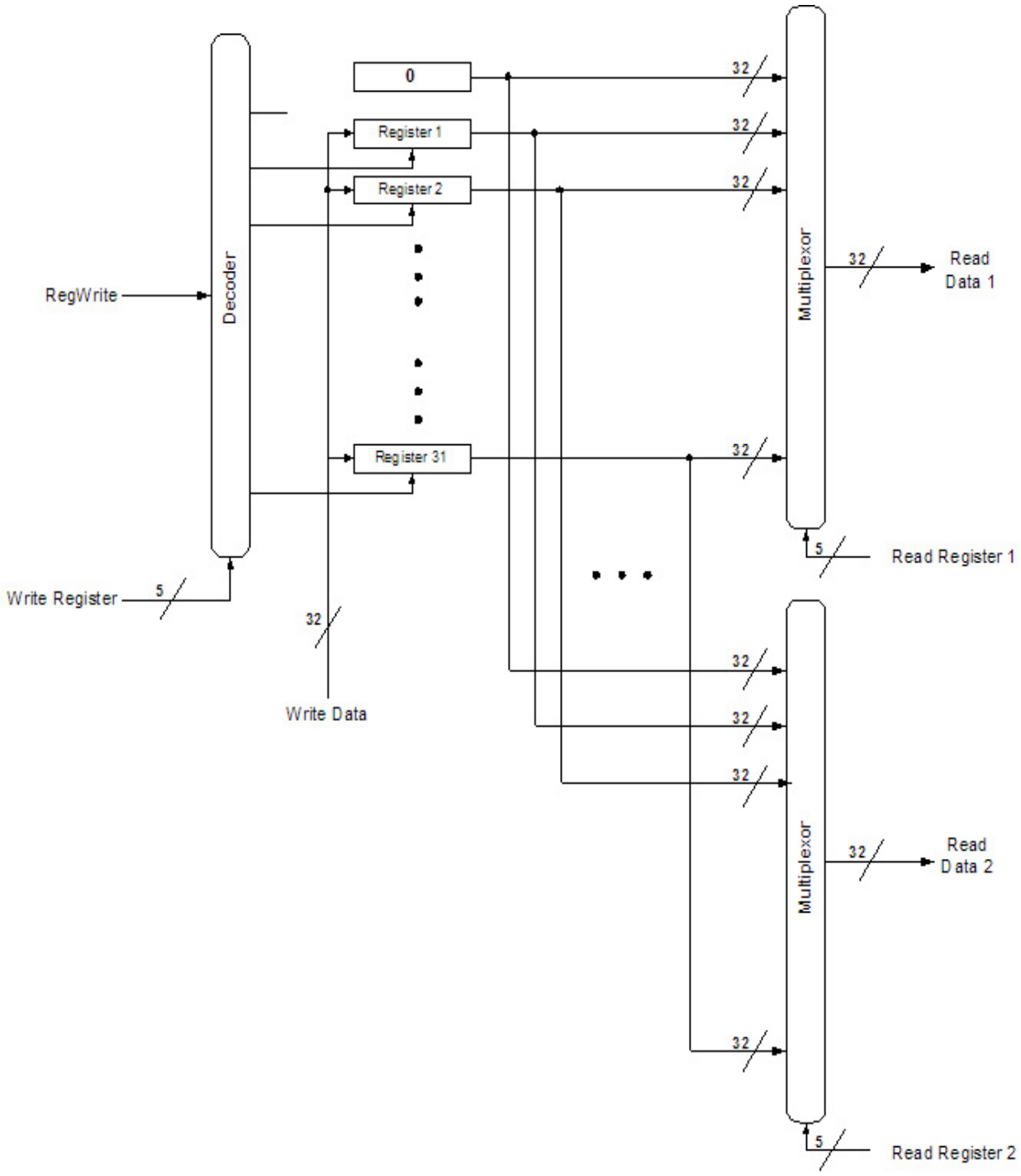
Draw a finite state machine (FSM) for an "I" detector. Make sure it correctly handles back to back I's [`2b'00`, `2b'11`, `2b'00`, `2b'11`, `2b'00`] as well as not recognizing garbage, *e.g.*[`2b'00`, `2b'01`, `2b'11`, `2b'00`]

Current State	Output			Next States
	I	L	U	

Once you've completed and tested your I detector, extend your design to identify both L and U (one at a time).

A 32 x 32-bit MIPS register file

Note: register zero always returns the value 0; we'll talk about this in lecture on Wednesday.



Instantiating registers

We're providing a *parameterized* register library, whose interface looks like this.

```
// register: A register which may be reset to an arbirary value
//
// q      (output) - Current value of register
// d      (input)  - Next value of register
// clk    (input)  - Clock (positive edge-sensitive)
// enable (input)  - Load new value? (yes = 1, no = 0)
// reset  (input)  - Asynchronous reset    (reset = 1)
//
module register(q, d, clk, enable, reset);

   parameter
            width = 32,
            reset_value = 0;

   output [(width-1):0] q;
   input [(width-1):0]  d;
   input  clk, enable, reset;
```

Most of the interface is straight-forward, but the parameter key word is new. Parameter provides a means for compile time specialization. In this case, with `width` of the register (*i.e.*, the number of bits it holds) can be set when you instantiate the register.

For example, if I wanted to create an 8-bit register, I could write:

```
   register #(8) my_reg (output, input, clk, ena, reset);
```

The `#(8)` specifies that the first parameter should be set to 8, which in turn sets the width of the output and input signals for the register. Pretty cool, hun'h?

Here's what a variable width 3-to-1 multiplexor looks like:

```
module mux3v(out, A, B, C, sel);

   parameter
     width = 32;

   output [width-1:0] out;
   input  [width-1:0] A, B, C;
   input  [1:0]       sel;
   wire   [width-1:0] wAB;

   mux2v #(width) mAB (wAB, A, B, sel[0]);
   mux2v #(width) mfinal (out, wAB, C, sel[1]);

endmodule // mux3v
```

Scaling Decoders

Decoders are an important circuit for implementing memories (like register files). Below is the truth table for a 1-to-2 decoder.

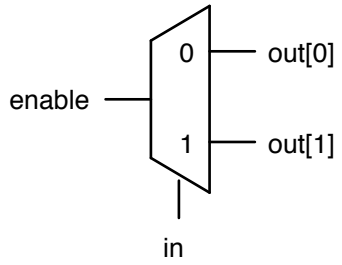


Figure 1. Decoder symbol

enable	in	out[1:0]
0	0	2'b00
0	1	2'b00
1	0	2'b01
1	1	2'b10

Figure 2. Decoder truth table

```
module decoder2 (out, in, enable);
   input     in;
   input     enable;
   output [1:0] out;

   and a0(out[0], enable, ~in);
   and a1(out[1], enable, in);
endmodule // decoder2
```

Figure 3. Decoder Verilog

Design a 2-to-4 decoder **using only 1-to-2 decoders**.

enable	in[1]	in[0]	out[3:0]
0	0	0	4'b0000
0	0	1	4'b0000
0	1	0	4'b0000
0	1	1	4'b0000
1	0	0	4'b0001
1	0	1	4'b0010
1	1	0	4'b0100
1	1	1	4'b1000

Design a 3-to-8 decoder **using only 1-to-2 and 2-to-4 decoders**.