Physics 364, Fall 2014, reading due 2014-11-30.

Email your answers to ashmansk@hep.upenn.edu by 11pm on Sunday

Course materials and schedule are at positron.hep.upenn.edu/p364

Assignment: (a) First read this week's notes, starting on the next page. (b) Then email me your answers to the questions below.

1. What are the inputs and outputs of a digital multiplexer called? (There are several possible correct answers.) What does a multiplexer do?

2. What is a ROM? What does it do? What are its inputs and outputs? Why was it important for us to talk about multiplexers before we introduced ROMs?

3. What is a state machine? What handy things can you do with a state machine?

4. Is there anything from this reading assignment that you found confusing and would like me to try to clarify? If you didn't find anything confusing, what topic did you find most interesting?

5. How much time did it take you to complete this assignment?

In last week's reading, we saw how the Verilog language, whose syntax is similar to that of C, Java, Arduino, etc., can be used to write down descriptions of digital circuits that would be rather tedious to draw as schematic diagrams, and would be absurdly tedious to wire up on a breadboard. These Verilog-based circuit descriptions can, in turn, be transformed by the Xilinx compiler into the internal wiring of an FPGA. Using an FPGA allows us to build digital circuits containing many thousands of logic gates and flip-flops, without having to wire them up by hand on a breadboard.

Last week's Lab 24 mainly illustrated FPGA-based analogues of the digital circuits that you built up from discrete digital components in Lab 23 and earlier in Lab 19. One snippet of Verilog that appeared in Lab 24 without first having appeared in these notes (oops!) was the Verilog idiom for creating a D-type flip-flop (which is often called a `dff`):

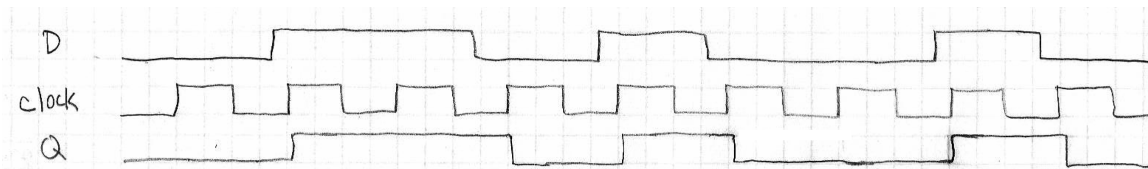
```
1  // Verilog idiom for D-type flip-flop
2  module dff (output q, input clk, input d);
3     reg q_reg;
4     always @ (posedge clk) q_reg <= d;
5     assign q = q_reg;
6  endmodule
```

In this course, you can treat the above snippet of Verilog as a recipe without learning the Verilog subtleties of "`reg`" vs "`wire`." But if you're curious, a `wire` in Verilog merely interconnects two points in a circuit, like a physical wire on your breadboard. A `wire` has no memory of anything that happened in the past. A `wire` connects the output of one part of your circuit to the input(s) of one or more other parts of your circuit. Whether the state of a wire is `0` or `1` depends only on the present output of the "upstream" part of your circuit, and not on past outputs. (Unlike a physical wire, a Verilog `wire` has no capacitance.) By contrast, a Verlog `reg` has an internal state that depends on past events, so it is analogous to a latch or a flip-flop. A `reg` is something like a single-bit version of the variables seen in ordinary programming languages, whereas a `wire` is just a name used to connect inputs of some things to outputs of other things. The line `always @ (posdege clk) q_reg <= d` means that every time there is a "positive" (LOW to HIGH) edge on the clock signal, the `reg` called `q_reg` should be updated from whatever value is present on the wire called `d` — which is exactly what a D-type flip-flop does.

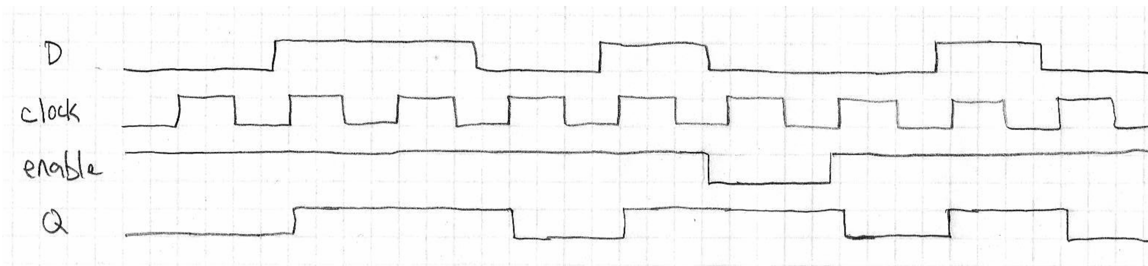
We introduced a second version of the D-type flip-flop in Lab 24, which we called `dffe`, or the D-type flip-flop *with enable*. In addition to inputs called `clk` and `D`, the `dffe` has another input called `enable`, which tells the flip-flop whether or not to update. When `enable` is HIGH, the `dffe` behaves just like a normal `dff`: the value of `D` from just before the rising edge of the clock appears, just after the rising edge of the clock, at the output `Q`. When `enable` is LOW, the `Q` output does not update — it just keeps its present value, as if the clock edge had never occurred. Here is the Verilog code for a `dffe`:

```verilog
// Verilog idiom for D-type flip-flop with enable
module dffe (output q, input clk, input d, input enable);
   reg q_reg;
   always @ (posedge clk) if (enable) q_reg <= d;
   assign q = q_reg;
endmodule
```

An ordinary D-type flip-flop (a `dff`) updates its `Q` output after *every* rising edge, so that `Q` just after the clock edge shows whatever value `D` had just before the clock edge (as shown below).



By contrast, a D-type flip-flop with enable (a `dffe`) will ignore any rising edges of the clock for which the `enable` input is LOW. It is as if the clock signal were disabled during those times when the `enable` signal is LOW. For that reason, `enable` is sometimes called the "clock enable" input. The graph below shows example waveforms for the inputs (`D`, `clk`, `enable`) and the output (`Q`) of a `dffe`. Except for the time when `enable` is 0, the `dffe` behavior is exactly the same as for a `dff`.



As you saw in Lab 24, the `dffe` is useful for making a counter. Bit 0 toggles on every positive edge of the clock. Bit 1 toggles only on those positive clock edges for which bit 0 was HIGH just before the clock edge. Bit 2 toggles only on those clock edges for which bits 0 and 1 were both HIGH just before the clock edge. And so on. This type of counter works analogously to the odometer in a car: the 1's digit is always "enabled." The 10's digit is only enabled when the 1's digit reads "9." The 100's digit is only enabled when the bottom two digits read "99." The 1000's digit is only enabled when the bottom three digits read "999," and so on.

That allowed us, in Part 6 of Lab 24, to make an 8-bit counter like this:

```verilog
// We need a wire to hold each flip-flop's Q output
wire q7, q6, q5, q4, q3, q2, q1, q0;
```

```
// each flipflop's Q is inverted, then sent into its own D input
wire d7 = ~q7,  d6 = ~q6,  d5 = ~q5,  d4 = ~q4;
wire d3 = ~q3,  d2 = ~q2,  d1 = ~q1,  d0 = ~q0;
// Here is the tricky part:  the first flip flop is always enabled.
// The second flip flop is only enabled if the first one is HIGH.
// The third flip flop is only enabled if the first two are HIGH.
// The fourth is only enabled if the first three are HIGH.  Etc.
wire e0 = 1;
wire e1 = q0;
wire e2 = q0 & q1;
wire e3 = q0 & q1 & q2;
wire e4 = q0 & q1 & q2 & q3;
wire e5 = q0 & q1 & q2 & q3 & q4;
wire e6 = q0 & q1 & q2 & q3 & q4 & q5;
wire e7 = q0 & q1 & q2 & q3 & q4 & q5 & q6;
// Instantiate the 8 flip-flops
dffe mydff0 (q0, clock, d0, e0);
dffe mydff1 (q1, clock, d1, e1);
dffe mydff2 (q2, clock, d2, e2);
dffe mydff3 (q3, clock, d3, e3);
dffe mydff4 (q4, clock, d4, e4);
dffe mydff5 (q5, clock, d5, e5);
dffe mydff6 (q6, clock, d6, e6);
dffe mydff7 (q7, clock, d7, e7);
```

It is actually pretty ugly, in the above example, to have eight separate wires called `q0`, `q1`, `q2`, ..., `q7`, and so on. We learned last week that we can use Verilog *vectors* to define groups of N wires. You can then (for example) manipulate an 8-bit-wide vector of wires as if it were an 8-bit integer, as in this example (which is equivalent to the above example).

```
// We need a wire to hold each flip-flop's Q output
wire [7:0] q;
// Instantiate the 8 flip-flops.  Flip-flop n is enabled only if
// the Q outputs of flip-flops n-1..0 are all HIGH.  The D input of
// each dffe is connected to the NOT of the flip-flop's own Q output.
dffe mydff0 (q[0], clock, ~q[0], 1);
dffe mydff1 (q[1], clock, ~q[1], q[0]);
dffe mydff2 (q[2], clock, ~q[2], q[1:0]==3);
dffe mydff3 (q[3], clock, ~q[3], q[2:0]==7);
dffe mydff4 (q[4], clock, ~q[4], q[3:0]==15);
dffe mydff5 (q[5], clock, ~q[5], q[4:0]==31);
dffe mydff6 (q[6], clock, ~q[6], q[5:0]==63);
dffe mydff7 (q[7], clock, ~q[7], q[6:0]==127);
```

By the way, the above example would probably be more clear to the reader if we used binary numbers instead of decimal numbers to represent $3 = 11_2$, $7 = 111_2$,

$15 = 1111_2$, etc. The way you represent a binary constant in Verilog is by prefixing it with `'b` as in this rewrite of the previous example:

```
// We need a wire to hold each flip-flop's Q output
wire [7:0] q;
// Instantiate the 8 flip-flops.  Flip-flop n is enabled only if
// the Q outputs of flip-flops n-1..0 are all HIGH.  The D input of
// each dffe is connected to the NOT of the flip-flop's own Q output.
dffe mydff0 (q[0], clock, ~q[0], 1);
dffe mydff1 (q[1], clock, ~q[1], q[0]);
dffe mydff2 (q[2], clock, ~q[2], q[1:0]=='b11);
dffe mydff3 (q[3], clock, ~q[3], q[2:0]=='b111);
dffe mydff4 (q[4], clock, ~q[4], q[3:0]=='b1111);
dffe mydff5 (q[5], clock, ~q[5], q[4:0]=='b11111);
dffe mydff6 (q[6], clock, ~q[6], q[5:0]=='b111111);
dffe mydff7 (q[7], clock, ~q[7], q[6:0]=='b1111111);
```

You can also use `'h` to prefix hexadecimal constants, as in `'hff` for 255 or `'hdeadbeef` for 3735928559.

There is one more very useful way in which we can generalize the D-type flip-flop: we can make its `D` input and its `Q` output be more than a single bit wide. For example, we can make an 8-bit-wide version of the `dffe`, like this:

```
module dffe_8bit (output [7:0] q, input clk, input [7:0] d, input enable);
  reg [7:0] q_reg;
  always @ (posedge clk) if (enable) q_reg[7:0] <= d[7:0];
  assign q[7:0] = q_reg[7:0];
endmodule
```

Equivalently, we could have built up an 8-bit-wide `dffe_8bit` by instantiating eight separate copies of the original `dffe`, like this:

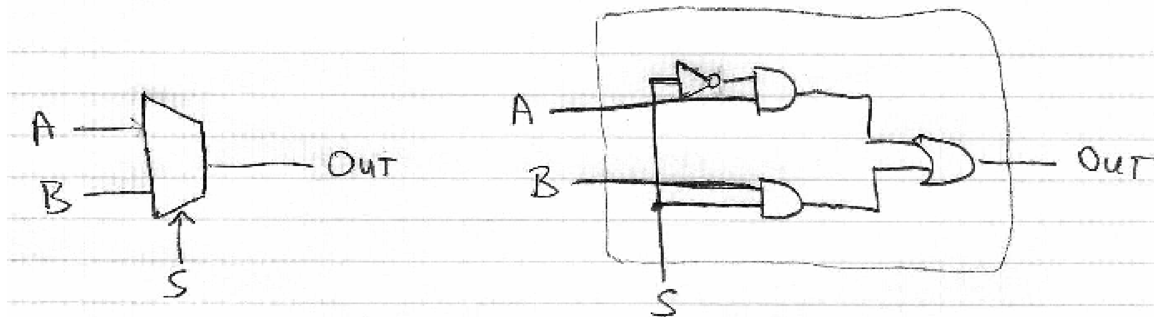
```
module dffe_8bit (output [7:0] q, input clk, input [7:0] d, input enable);
  dffe (q[0], clk, d[0], enable);
  dffe (q[1], clk, d[1], enable);
  dffe (q[2], clk, d[2], enable);
  dffe (q[3], clk, d[3], enable);
  dffe (q[4], clk, d[4], enable);
  dffe (q[5], clk, d[5], enable);
  dffe (q[6], clk, d[6], enable);
  dffe (q[7], clk, d[7], enable);
endmodule
```

Using this 8-bit-wide flip-flop, we now have a very easy way to make an 8-bit-wide counter. An 8-bit-wide adder (which Verilog is smart enough to figure out how to make out of logic gates) takes the present value of `q[7:0]` and adds 1 to it. The result

of this addition is wired into the `d[7:0]` inputs of the 8-bit-wide D-type flip-flop. Each time the clock makes a LOW-to-HIGH transition, the 8-bit value represented by flip-flop outputs `q[7:0]` is incremented by 1. So `q[7:0]` takes on successive values 00000000_2, 00000001_2, 00000010_2, 00000011_2, ..., 11111111_2, and then wraps around back to 00000000_2. In other words, it counts from 0 to 255, over and over again, incrementing once per clock tick.

```
1 wire clk = FromSomeInputPin;  // assume the clock is provided to us
2 wire enable = 1;              // these flip-flops will always be enabled
3 wire [7:0] q;                 // each flip-flop has a Q output
4 wire [7:0] d = q + 1;         // add 1 to Q and send it back to D
5 dffe_8bit myflipflop (q[7:0], clk, d[7:0], enable);
```

At the end of last week's reading, we discussed a component called a **digital multiplexer**, which is sometimes abbreviated "mux." The schematic symbol for a multiplexer usually looks like the left figure below. There are three inputs, called **A**, **B**, and **S**, and there is one output. If **S** is LOW, then the output shows the value of input **A**; if **S** is HIGH, then the output shows the value of input **B**. So a multiplexer selects between several possible inputs (just two possibilities in the simplest case), sending a copy of the selected input to the output.



The above-right figure shows how a two-to-one multiplexer can be implemented using ordinary logic gates. I remember being very surprised, when I first saw this circuit, that it actually works as a multiplexer. To try to convince you that it really works, I'll show you the truth table below, which you can check by tracing through the above-right schematic diagram.

Amazingly, as you can work through by comparing the truth table with the schematic diagram, the circuit does indeed cause **OUT** to equal **A** when **S** is LOW, and it does indeed cause **OUT** to equal **B** when **S** is HIGH.

A	B	S	$A \cdot \overline{S}$	$B \cdot S$	OUT
0	0	0	0	0	0
0	1	0	0	0	0
1	0	0	1	0	1
1	1	0	1	0	1
0	0	1	0	0	0
0	1	1	0	1	1
1	0	1	0	0	0
1	1	1	0	1	1

We have several different ways we could write Verilog code corresponding to this multiplexer. The most direct way would be to mimic the schematic diagram precisely:

```
module mux2to1 (input S, A, B, output OUT);
    wire notS, AandnotS, BandS;
    not (notS, S);
    and (AandnotS, A, notS);
    and (BandS, B, S);
    or  (OUT, AandnotS, BandS);
endmodule
```

Alternatively, we could use Verilog's C-like binary operators, in place of the built-in `and`, `or`, and `not` logic gates:

```
module mux2to1 (input S, A, B, output OUT);
    wire notS, AandnotS, BandS;
    assign notS = ~S;
    assign AandnotS = A & notS;
    assign BandS = B & S;
    assign OUT = AandnotS | BandS;
endmodule
```

As we showed in last week's reading, the `wire` statement creates a new wire, while the `assign` statement connects an existing wire. To make things more concise, we can create a new wire and connect it in the same statement, like this:

```
module mux2to1 (input S, A, B, output OUT);
    wire notS = ~S;
    wire AandnotS = A & notS;
    wire BandS = B & S;
    assign OUT = AandnotS | BandS;
endmodule;
```

Finally, we can eliminate the internal wires by combing several Verilog operators in a single expression, like this:

```
module mux2to1 (input S, A, B, output OUT);
```

```
2     assign OUT = (A & ~S) | (B & S);
3 endmodule;
```

We mentioned in last week's reading that Verilog, like C and Java, has a so-called *ternary operator*, **?:**, which is analogous to a multiplexer. It is used in statements like `assign OUT = S ? B : A;` If the first expression (S) is true, the ternary operator evaluates to the second expression (B); otherwise, it evaluates to the third expression (A). We can use this strange-looking ternary operator as yet another way to implement our multiplexer in Verilog, as follows:

```
1 module mux2to1 (input S, A, B, output OUT);
2     assign OUT = S ? B : A;
3 endmodule;
```

As we did with flip-flops, we can generalize the multiplexer so that instead of selecting between two single wires `A` and `B`, we can select between two vectors, for example the four-bit-wide vectors `A[3:0]` and `B[3:0]`. We can do that in Verilog like this:

```
1 module mux2to1_4bit (
2   input        S,
3   input  [3:0] A,
4   input  [3:0] B,
5   output [3:0] OUT
6 );
7   assign OUT[3:0] = S ? B[3:0] : A[3:0];
8 endmodule;
```

If you want to generalize the multiplexer to be N bits wide, so that there is no need to define separate modules for 4-bit-wide values, 8-bit-wide values, etc., you can use a **parameter**, as in this example. In this case, we give the parameter N a default value of 1.

```
1 module mux2to1_Nbit #(parameter N=1)
2 (
3   input          S,
4   input  [N-1:0] A,
5   input  [N-1:0] B,
6   output [N-1:0] OUT
7 );
8   assign OUT[N-1:0] = S ? B[N-1:0] : A[N-1:0];
9 endmodule;
```

To instantiate this module (within some other module whose details we omit), we use the following syntax. In this case, we are using $N = 4$.

```
1 wire [3:0] inputA, inputB, muxout;
2 mux2to1_Nbit #(4) mymux (S, inputA, inputB, muxout);
```

We will find parameters especially handy for defining N-bit-wide flip-flops, like this:

```
1  module dffe_Nbit #(parameter N=1)
2  (
3    output [N-1:0] q,
4    input          clk,
5    input  [N-1:0] d,
6    input          enable
7  );
8    reg [N-1:0] q_reg;
9    always @ (posedge clk) if (enable) q_reg <= d;
10   assign q = q_reg;
11 endmodule
```

You might have noticed, while reading through these Verilog examples, that I have not always been perfectly consistent about whether a module's inputs are listed before or after the module's outputs. In general, it can be difficult to keep track of the order in which one has chosen to write the various inputs and outputs of a module. If you write the arguments in one order when you define the module, and then you accidentally write the arguments in a different order when you use the module, you can create a "bug" (a mistake) that is very difficult to track down. Fortunately, Verilog has a mechanism for avoiding these kinds of mistakes. It is possible (and strongly encouraged) to specify module arguments *by name* rather than *by position* when you instantiate a module. So instead of writing

```
1 wire [3:0] inputA, inputB, muxout;
2 mux2to1_Nbit #(4) mymux (S, inputA, inputB, muxout);
```

we can instead write

```
1 wire [3:0] inputA, inputB, muxout;
2 mux2to1_Nbit #(.N(4)) mymux (.S(S), .A(inputA), .B(inputB), .OUT(muxout));
```

This convention avoids completely the potential problem of writing the arguments in the wrong order. If you use the arguments-by-name convention, you can use whatever order you like. This example will work just as well as the previous example:

```
1 wire [3:0] inputA, inputB, muxout;
2 mux2to1_Nbit #(.N(4)) mymux (.OUT(muxout), .A(inputA), .B(inputB), .S(S));
```

But if instead you write

```
mux2to1_Nbit #(4) mymux (muxout, inputA, inputB, S);
```

when you meant to write

```
mux2to1_Nbit #(4) mymux (S, inputA, inputB, muxout);
```

you will definitely not get the results that you expect. So it is much safer (less error-prone) to use the arguments-by-name convention. It is also much easier to read Verilog code that uses the arguments-by-name convention, because you don't need

to keep looking back and forth between the module instantation and the module definition to figure out the meanings of the first, second, third arguments, etc. It is easier to remember the names than to remember what order you wrote them in.

Using this convention, we could rewrite our last example of an 8-bit counter like this:

```
module counter (input clk, output [7:0] q);
    dffe ff0 (.clk(clk), .q(q[0]), .d(~q[0]), .enable(1));
    dffe ff1 (.clk(clk), .q(q[1]), .d(~q[1]), .enable(q[0]=='h1));
    dffe ff2 (.clk(clk), .q(q[2]), .d(~q[2]), .enable(q[1:0]=='h3));
    dffe ff3 (.clk(clk), .q(q[3]), .d(~q[3]), .enable(q[2:0]=='h7));
    dffe ff4 (.clk(clk), .q(q[4]), .d(~q[4]), .enable(q[3:0]=='hf));
    dffe ff5 (.clk(clk), .q(q[5]), .d(~q[5]), .enable(q[4:0]=='h1f));
    dffe ff6 (.clk(clk), .q(q[6]), .d(~q[6]), .enable(q[5:0]=='h3f));
    dffe ff7 (.clk(clk), .q(q[7]), .d(~q[7]), .enable(q[6:0]=='h7f));
endmodule
```

But it is actually much simpler to write an 8-bit counter like this:

```
module counter (input clk, output [7:0] q);
    dffe_Nbit #(.N(8)) myff (.clk(clk), .q(q), .d(q+1), .enable(1));
endmodule
```

More experienced Verilog coders would probably implement a counter as follows, without using the separate `dffe_Nbit` module. But I have tried my best in these examples to avoid using Verilog's `always` blocks, except in the magic incantation for creating a D-type flip-flop, to keep the amount of Verilog you need to learn down to a minimum. So you may choose to close your eyes for this example:

```
module counter (input clk, output [7:0] q);
    reg [7:0] q_reg;
    always @ (posedge clk) q_reg <= q_reg + 1;
    assign q = q_reg;
endmodule
```

We saw an example of an 8-bit-wide *shift register* in Lab 24.[1] Here is an 8-bit-wide shift register using `dffe_Nbit`:

```
module shiftreg (input clk, input d, output [7:0] q);
    wire [7:0] ff_data;
    assign ff_data[7:1] = q[6:0];
    assign ff_data[0] = d;
    dffe_Nbit #(.N(8)) ff (.clk(clk), .q(q), .ena(1), .d(ff_data));
endmodule
```

[1] https://en.wikipedia.org/wiki/Shift_register

Using the *concatenation operator* (written using curly braces **{}**) that we first saw in last week's reading, you can save a few keystrokes:

```
1 module shiftreg (input clk, input d, output [7:0] q);
2     wire [7:0] ff_data = {q[6:0],d}
3     dffe_Nbit #(.N(8)) ff (.clk(clk), .q(q), .ena(1), .d(ff_data));
4 endmodule
```

You could eliminate the `ff_data` wires, too, if you like, as in the following example. By the way, don't let yourself be confused by the fact that `shiftreg` and `dffe_Nbit` both have inputs and outputs named `d` and `q`.

```
1 module shiftreg (input clk, input d, output [7:0] q);
2     dffe_Nbit #(.N(8)) ff (.clk(clk), .q(q), .ena(1), .d({q[6:0],d}));
3 endmodule
```

One last bit of random Verilog minutia: Be sure to put the line

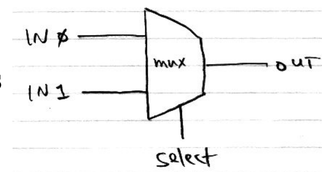
`` `default_nettype none ``

near the top of each of your Verilog source files. The first character is a backquote, which usually appears on the same keyboard key as the tilde (˜) character. (You will see this near the top of each of my Verilog files, unless I have left it out by accident.) By default, Verilog will treat an undeclared identifier as if it were a (one-bit-wide) wire. Thus, simply mistyping the name of a wire can lead to a very difficult-to-find bug. If you include the above line at the top of your program, Verilog will complain if you try to use a name that you have not declared. This is a good thing!

OK, that's all the Verilog that we need to know in order to build up the rest of the ideas in the course. One thing to emphasize is that every Verilog example that we have introduced can in some way be reduced to an equivalent set of logic gates and flip-flops. (And flip-flops can also be built up from logic gates.) So all of the Verilog examples that we have studied are really just convenient ways of describing schematic diagrams containing a large number of logic gates (AND, OR, NAND, XOR, etc.) and their interconnections.

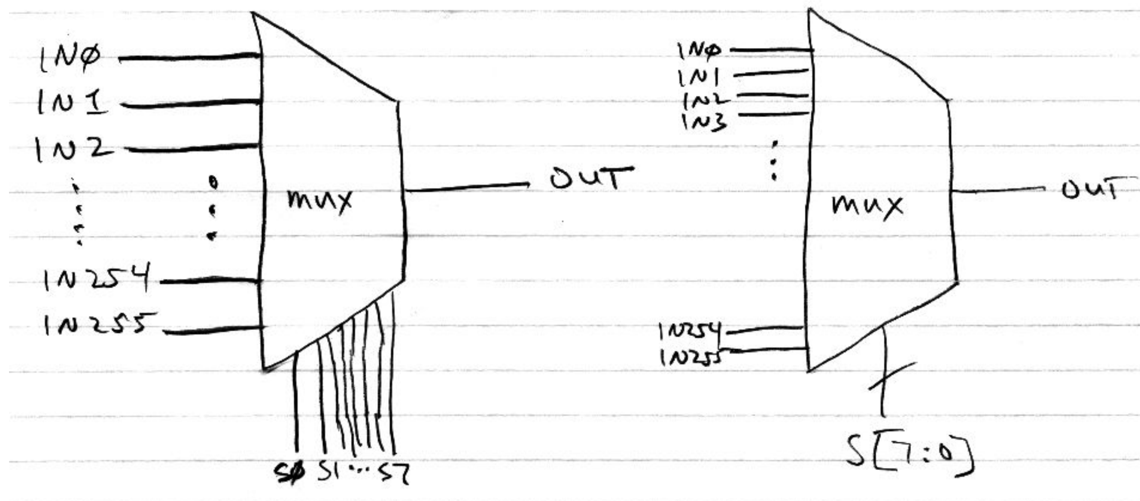
In earlier pages, we described multiplexers as digital circuits whose output is selected from two possible inputs, **A** and **B**. Which of the two inputs is selected is determined by a third input, **S**. We saw how to implement a multiplexer using logic gates. The logic-gate implementation is equivalent to the Verilog statement

`assign OUT = (IN0 & ~select) | (IN1 & select);`

where I have changed the names **A** and **B** into `IN0` and `IN1`, and I've changed the name **S** to `select`. The symbol for this multiplexer is 

.

Earlier, we generalized to the case in which **A** and **B** are N-bit vectors — which can be done simply by instantiating N parallel copies of the single-bit multiplexer circuit. Let's instead generalize to the case in which **S** (or `select`) is an M-bit vector. For example, if $M = 8$, I can select between $2^8 = 256$ different inputs, and send my choice of those 256 inputs to the output. I'll draw the schematic symbol for such a multiplexer in two different ways: one using eight separate select lines, and another using an 8-bit-wide "bus" (called a "vector" in Verilog) of select lines. Usually the latter notation is preferred, as it is easier to read.



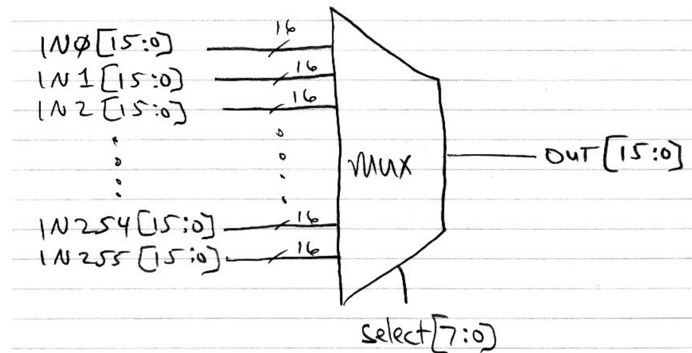
There is a surpringly straightforward (but tedious!) way to implement this 256-to-1 multiplexer using logic gates. Here is a Verilog representation, whose translation into logic gates is direct and should be easy to imagine. (I took some liberties with ellipsis (...) to avoid typing out everything literally.)

```
1  module mux256to1 (input [7:0] S, input IN0, IN1, IN2, ..., IN255, output OUT);
2    assign OUT =
3      (!S[7] & !S[6] & !S[5] & !S[4] & !S[3] & !S[2] & !S[1] & !S[0] & IN0)   |
4      (!S[7] & !S[6] & !S[5] & !S[4] & !S[3] & !S[2] & !S[1] &  S[0] & IN1)   |
5      (!S[7] & !S[6] & !S[5] & !S[4] & !S[3] & !S[2] &  S[1] & !S[0] & IN2)   |
6      (!S[7] & !S[6] & !S[5] & !S[4] & !S[3] & !S[2] &  S[1] &  S[0] & IN3)   |
7      ....
8      ( S[7] &  S[6] &  S[5] &  S[4] &  S[3] &  S[2] &  S[1] & !S[0] & IN254) |
9      ( S[7] &  S[6] &  S[5] &  S[4] &  S[3] &  S[2] &  S[1] &  S[0] & IN255) ;
10 endmodule
```

Notice that `mux256to1` is just a big pile of AND gates, OR gates, and inverters. It's something like 128 inverters, 256 nine-input AND gates, and one 256-input OR gate — so you could build it with fewer than a thousand 74HC00 (quad two-input NAND gate) chips. Notice also that simply by connecting all of the `IN0`...`IN255` to a suitable combination of 1 and 0 values, `mux256to1` can implement an arbitrary 8-to-1 truth table. It can map any desired pattern of 8 input bits into one output bit.

An arbitrary truth table is sometimes called a *lookup table*. It turns out that a 4-to-1 lookup table (usually paired with a D-type flip-flop) is the basic unit of a Xilinx FPGA.[2] So the building blocks of FPGAs are not really anything so mysterious.

Now let's repeat `mux256to1` sixteen times, so that each of the 256 input choices is not a single bit but rather a 16-bit number. The schematic symbol for this even larger multiplexer looks like this:



The corresponding Verilog code (again with some "..." abbreviations) is:

```
module mux256to1_16bit (
    input   [7:0] S,
    input  [15:0] IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8, IN9, IN10,
    input  [15:0] IN11, IN12, IN13, IN14, IN15, IN16, IN17, IN18, IN19, IN20,
    ...
    input  [15:0] IN250, IN251, IN252, IN253, IN254, IN255,
    output [15:0] OUT
);
    mux256to1(S, IN0[0], IN1[0], IN2[0], ...., IN254[0], IN255[0], OUT[0]);
    mux256to1(S, IN0[1], IN1[1], IN2[1], ...., IN254[1], IN255[1], OUT[1]);
    mux256to1(S, IN0[2], IN1[2], IN2[2], ...., IN254[2], IN255[2], OUT[2]);
    ....
    mux256to1(S, IN0[14], IN1[14], IN2[14], ...., IN254[14], IN255[14], OUT[14]);
    mux256to1(S, IN0[15], IN1[15], IN2[15], ...., IN254[15], IN255[15], OUT[15]);
endmodule
```

Now we have a device that can map an arbitrary 8-bit number into an arbitrary 16-bit number, simply by arranging for all 4096 of the `IN` wires to be connected to the desired pattern of 1 and 0 values. Such a device is known as a **Read-Only Memory**, or a **ROM**. In this case, it would be called a 256×16 ROM.

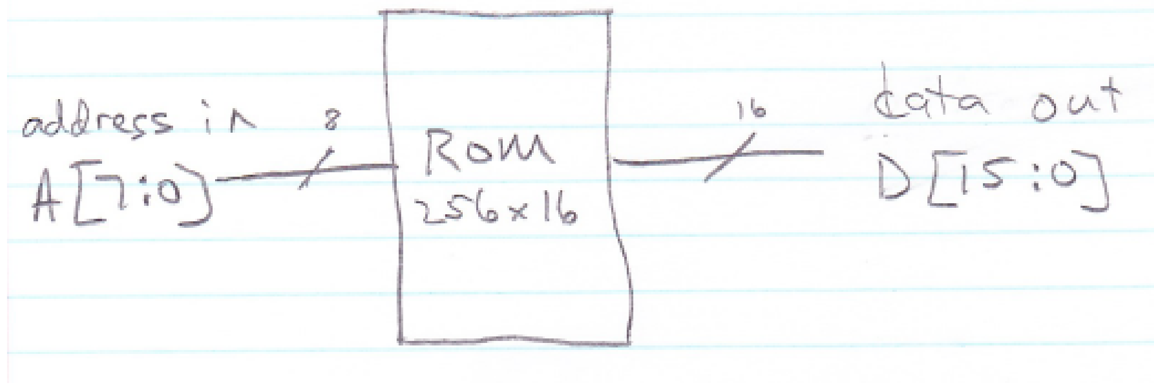
A ROM is the simplest example of a **memory**. A memory can store useful information. For instance, this ROM might store 16 bits of longitude and 16 bits of latitude of each of 128 cities you would like to visit on future vacations. (16 bits per geographic coordinate gives you about 0.6 km precision.) Or we could use this ROM to store, for example, the squares of the integers $0 \ldots 255$, or a table of the first 256 prime

[2] https://en.wikipedia.org/wiki/Field-programmable_gate_array

numbers i.e. mapping n to `Prime[n]` (in *Mathematica* notation).

So a ROM is just a big multiplexer, whose `IN[]` lines have been pre-wired to some desired pattern.

Memories come in many shapes and sizes, though the number of *data bits* is most commonly 8, 16, or 32, and the number of *storage locations* is always a power of two — it's 2^n, where n is the number of **address bits**. In a memory, the `OUT[15:0]` lines would really be called **data lines** `D[15:0]` (or in some cases `Q[15:0]`), and the `S[7:0]` "select" lines would really be called **address lines** `A[7:0]`.

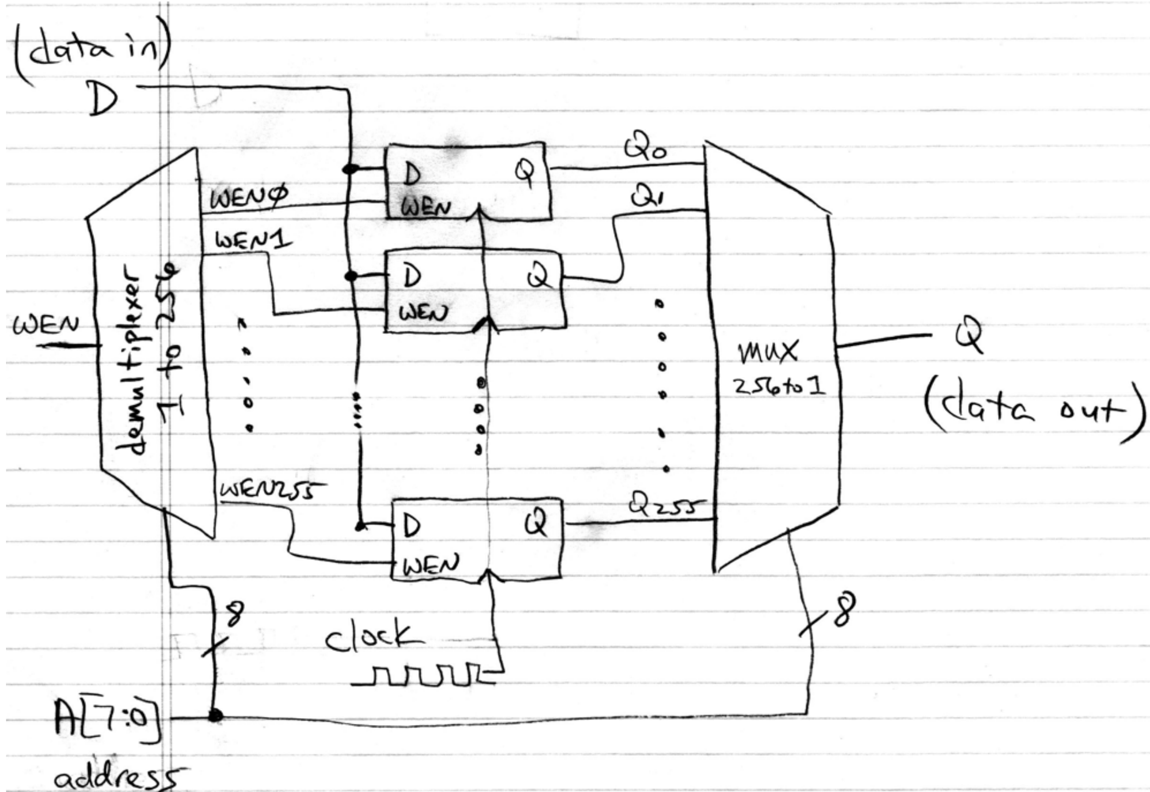


While a ROM is a handy device, memory is far more useful if your circuit can write to it as well as read from it.

A memory with which you can both read and write (i.e. you can both examine its contents and modify its contents) is normally called a **RAM** (*random access memory*), though technically the "random" in RAM refers not to read/write capability but rather to the fact that you can access the addresses in any order you like. A magnetic tape drive, by contrast, allows only sequential access. There is more discussion of what the "random" really means in this context in the Wikipedia's RAM article.[3] By the Wikipedia's definition, a ROM is a special case of a random access memory, since you can access its contents in any order. While this may be strictly true, nobody who is trying to communicate clearly would ever refer to read-only memory as a RAM.

So for our purposes, a RAM is a memory that can be both read and written quickly at arbitrary addresses in arbitrary order. The fastest and easiest-to-use RAM is called *static RAM*, or **SRAM**. An SRAM uses one flip-flop to store each memory bit. (Often the flip-flop in an SRAM is formed using two inverters instead of two NAND gates, to use fewer transistors, but conceptually each memory cell is a flip-flop.) So a 256×16 SRAM would use 4096 flip-flops. Here is a schematic drawing of a straightforward implementation of a 256×1 SRAM (i.e. 256 locations, each storing 1-bit-wide data).

[3] http://en.wikipedia.org/wiki/Random-access_memory



Here is Verilog code equivalent to the above schematic diagram:

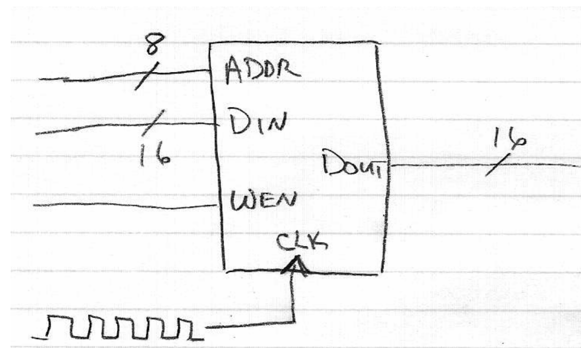
```
1  module sram256x1 (input clk, wen, D, input [7:0] A, output Q);
2      // Use address lines to demultiplex write-enable to 256 flip-flop enables
3      wire wen0 = wen && A=='b00000000;
4      wire wen1 = wen && A=='b00000001;
5      wire wen2 = wen && A=='b00000010;
6      ....
7      wire wen254 = wen && A=='b11111110;
8      wire wen255 = wen && A=='b11111111;
9      // 256 D-type flip-flops store 1 bit for each of 256 locations
10     dffe ff0 (.clk(clk), .enable(wen0), .d(D), .q(Q0));
11     dffe ff1 (.clk(clk), .enable(wen1), .d(D), .q(Q1));
12     ....
13     dffe ff255 (.clk(clk), .enable(wen255), .d(D), .q(Q255));
14     // Use address lines to multiplex 256 flip-flop outputs to data output Q
15     mux256to1 mux (.S(A), .IN0(Q0), .IN1(Q1), .IN2(Q2), ....,
16                    .IN255(Q255), .OUT(Q));
17 endmodule
```

The "write enable" line, `WEN`, is normally 0. To read from the above SRAM, you simply set the address lines `A[7:0]` to point to the desired memory location, and the bit stored at that location appears at the `Q` output. To write to the SRAM (i.e. to change its stored contents), you set `A[7:0]` to point to the memory location whose contents you want to change, you set the `WEN` line to 1, and you set the `D` input to whatever value (0 or 1) you want to store at location `A`. Then on the next rising

edge of the clock, the contents of location `A` are replaced by `D`. Since updates only happen on the rising edge of a clock, this is a *synchronous SRAM*. There are also asynchronous SRAMs, just as there are asyncrhonous flip-flops, but the synchronous type is much easier to use correctly.

(I used a *demultiplexer* above without ever telling you what one is. As described in the Wikipedia,[4] a demultiplexer takes one data input and n `select[]` inputs. It has 2^n separate outputs, corresponding to the 2^n possible values of `select[n-1:0]`. The demultiplexer forwards the data input to the selected output and simply sends '0' to all of the non-selected outputs. It does the opposite of what a multiplexer does.)

The circuit shown above is a 256×1 SRAM: it has 8 address lines and only one data line. If we wanted a 256×16 SRAM (which would probably be much more useful), we would replicate this circuit 16 times, as we did above to go from the 256×1 ROM to the 256×16 ROM. The resulting SRAM would store a 16-bit-wide value at each of $2^8 = 256$ locations. The symbol that one would use for drawing such an SRAM on a schematic diagram is shown to the right. The Verilog code to implement an 256×16 SRAM, by building it up out of 16 256×1 SRAMs, is shown below.




```
1  module sram256x16 (
2      input         clk,
3      input         wen,
4      input  [15:0] Din,
5      input   [7:0] addr,
6      output [15:0] Dout
7  );
8      sram256x1 ram0 (.clk(clk), .wen(wen), .D(Din[0]), .A(addr), .Q(Dout[0]);
9      sram256x1 ram0 (.clk(clk), .wen(wen), .D(Din[1]), .A(addr), .Q(Dout[1]);
10     sram256x1 ram0 (.clk(clk), .wen(wen), .D(Din[2]), .A(addr), .Q(Dout[2]);
11     ....
12     sram256x1 ram14 (.clk(clk), .wen(wen), .D(Din[14]), .A(addr), .Q(Dout[14]);
13     sram256x1 ram15 (.clk(clk), .wen(wen), .D(Din[15]), .A(addr), .Q(Dout[15]);
14 endmodule
```

In most real-life SRAMs, a single data bus is used for both reads and writes. Having separate read and write data-paths would waste I/O pins. (It would be like having twice as many data lines to wire up on your breadboard.) But separate D_{in} and D_{out} (or sometimes D and Q, respectively) buses are easier to use and to understand. So inside an FPGA, where there are no physical pins to worry about connecting, one

[4] https://en.wikipedia.org/wiki/Multiplexer#Digital_demultiplexers

frequently uses separate buses for reading and writing a memory.

The RAM inside your computer — whose size is nowadays measured in gigabytes — is not SRAM. It is *synchronous DRAM* (SDRAM). DRAM (*dynamic* RAM) uses a capacitor instead of a flip-flop to hold the state of each stored bit. (Billions of capacitors are cheaper and smaller than billions of flip-flops.) Since the capacitors' charge slowly leaks away, the contents of DRAM must be refreshed several times per second. In exchange for the hassle of refreshing, DRAM offers a large cost savings over SRAM: SRAM requires typically six transistors per memory cell, while DRAM requires one transistor and one capacitor per cell.

Finite State Machines

The figure to the right (figure 8.57 from Horowitz & Hill, 2nd edition) shows a generic sequential-logic circuit. (Sequential logic is logic that contains flip-flops. The word "register," used in the diagram, is an alternative name for a flip-flop.) A **Finite State Machine** (**FSM**) is a special case of sequential circuit in which one register is designated the **state number**. The FSM moves from state to state when specified conditions are met. The next state is a (deterministic) function of the present state and the machine's present input. The output is a function of the present state.

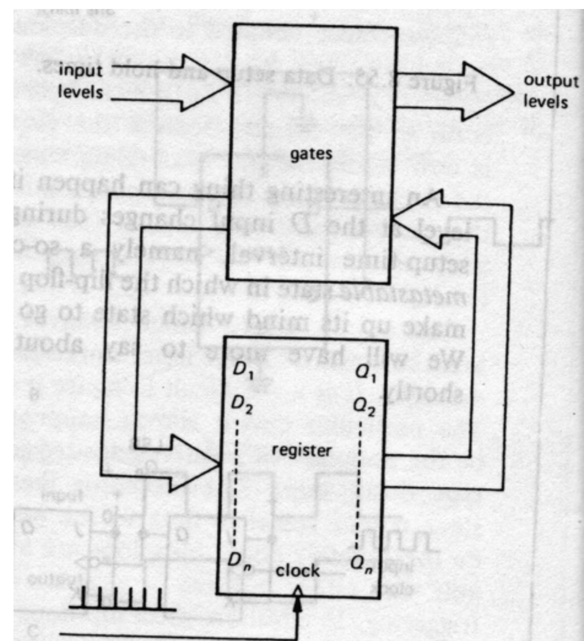
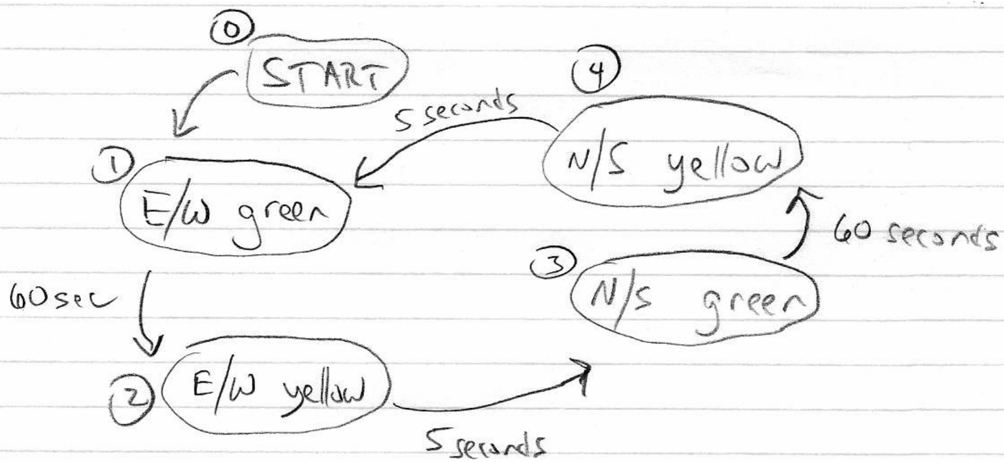


Figure 8.57. The classical sequential circuit: memory registers plus combinational logic. This scheme can be easily implemented with single-chip "registered PALs" (Section 8.27).

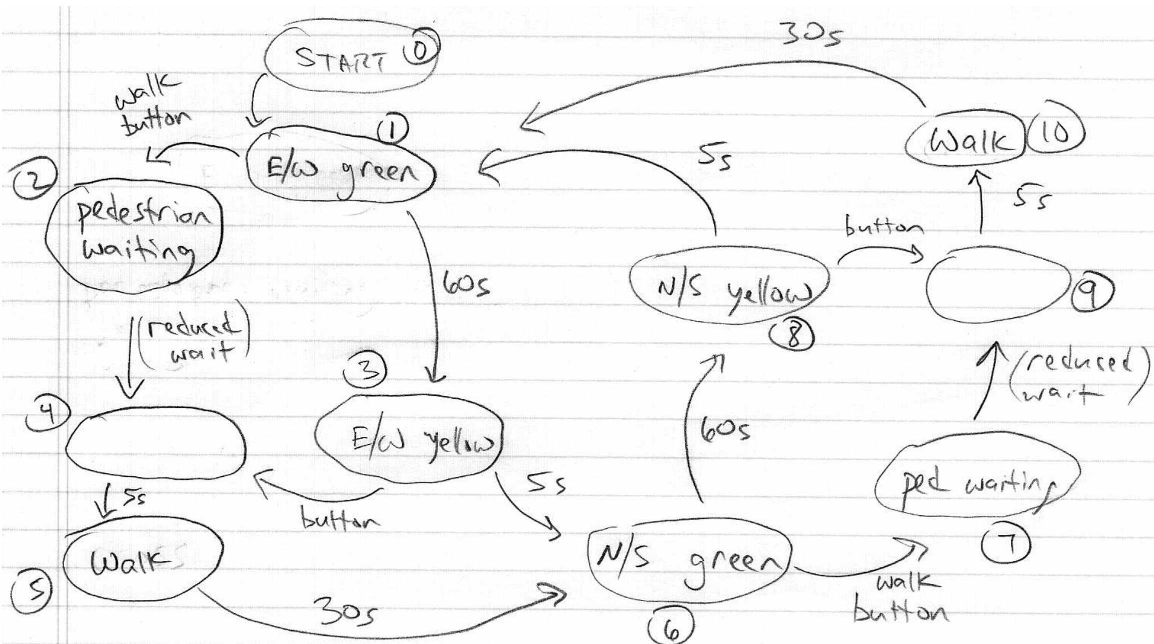
FSMs are ubiquitous — and quite handy. The two most common applications that come to mind are validating ("parsing") input — e.g. for a vending machine — and sequencing operations — e.g. for a traffic signal. A state machine allows you to do with a few D-type flip-flops and some combinational logic the sorts of things you may be accustomed to doing with a simple computer program. This can be helpful if the circuit you are building needs to carry out a sequence of steps to do its job.

A state diagram can be very helpful for describing an FSM's intended behavior. Here is a simple FSM for a traffic signal. (E/W means "east/west" and N/S means "north/south.")



state	N/S R	Y	G	E/W R	Y	G	next state
0	1	0	0	1	0	0	1
1	1	0	0	0	0	1	2
2	1	0	0	0	1	0	3
3	0	0	1	1	0	0	4
4	0	1	0	1	0	0	1

Now suppose we add a button for pedestrians to signal that they want to cross the street. The state diagram becomes more complicated:



Here, shown below, is a state table. For each state number, it shows the state of each of the seven outputs, and what the next state will be. In many cases, which state is

next will depend on external inputs.

state	N/S R	Y	G	E/W R	Y	G	WALK	next state
0	1	0	0	1	0	0	0	1
1	1	0	0	0	0	1	0	button → 2, timeout → 3
2	1	0	0	0	0	1	0	4
3	1	0	0	0	1	0	0	button → 4, time → 6
4	1	0	0	0	1	0	0	5
5	1	0	0	1	0	0	1	6
6	0	0	1	1	0	0	0	button → 7, time → 8
7	0	0	1	1	0	0	0	9
8	0	1	0	1	0	0	0	button → 9, time → 1
9	0	1	0	1	0	0	0	10
10	1	0	0	1	0	0	1	1

I will provide in Lab 26 a skeletal Verilog implementation of the first traffic signal FSM above — the simpler one. Part of Lab 26 will be for you to flesh out the missing details. If you have time, you can implement a version that includes the pedestrian signal.

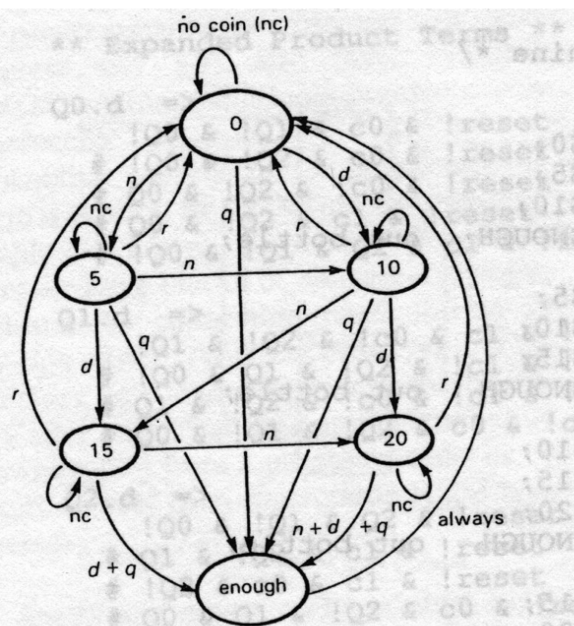
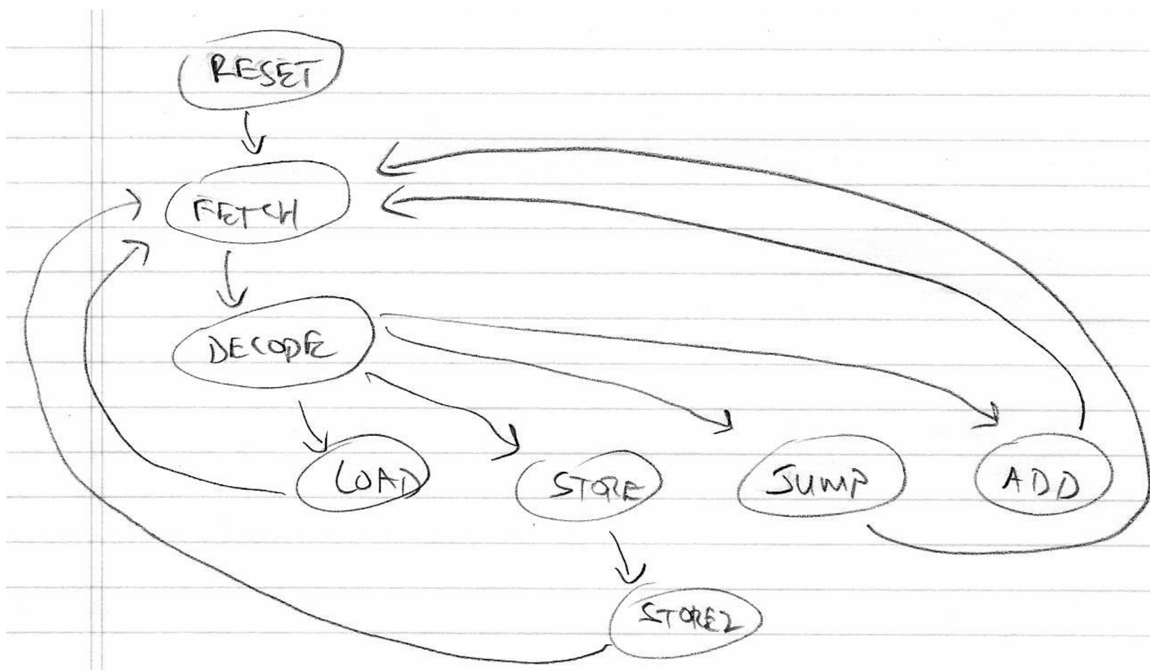


Figure 8.80. Vending machine state diagram.

Shown to the right is a state diagram from Horowitz & Hill (figure 8.80) for a vending machine. Inserting a nickel, dime, or quarter is represented on the diagram as "n," "d," or "q." In Lab 26, we will work through an implementation of this FSM. You might want to ask yourself how you would modify this machine to give back change (the correct amount) if you overpay. Also, what sequence of states would you jump through in order to handle a transaction that has been cancelled?

In the next (final) reading for the course, we will further explore the state-machine concept, and we will introduce the concept of a *microprocessor*. A microprocessor executes a sequence of *instructions*. Instructions have useful consequences, such as moving data around, performing math operations, performing input/output to the external world, etc. (This discussion will be repeated in much more detail next week.)



A microprocessor's instructions are stored in a memory. The *program counter* is the memory address of the next instruction to be executed. The program counter is one of several *registers* used by the microprocessor: an *instruction register* holds the instruction currently being executed; various *general-purpose registers* may hold temporary results of computations; sometimes there is an *accumulator* register that plays a special role in math operations. The processor has access to read/write memory for data storage. In the Von Neumann architecture (which is most common), the program and its data are stored in the same memory. One occasionally sees the alternative "Harvard architecture," in which program memory and data memory are totally separate.

Typically you write your program in C, C++, Java, the Arduino language, etc. A *compiler* then translates your program into low-level instructions for the microprocessor to execute. The microprocessor starts at the first address of your program, fetches the instruction from that memory location, decodes the instruction, executes the instruction, and then goes on to the next instruction.

The simple microprocessor we will discuss next week has this very minimal set of instructions: `LOAD`, `STORE`, `JUMP`, `JUMPZ` (jump if zero), `JUMPN` (jump if negative), `ADD`, `SUB`, `MUL`. We will see that this set of operations is sufficient for us to make a program that calculates and displays on the BASYS2 board's LEDs all prime numbers less than 9999.

A microprocessor is basically a sophisticated Finite State Machine with an attached memory and some means of exchanging data with the outside world. That's why the memory and the FSM are important ideas for us to study this week, in preparation

for making that final step. Then we will have gone from transistor all the way to (simplified) computer.