Physics 364, Fall 2014, Lab #26 **Name:** ____________________

(Field Programmable Gate Arrays: 3)

Wednesday, December 3 (section 401); Thursday, December 4 (section 402)

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

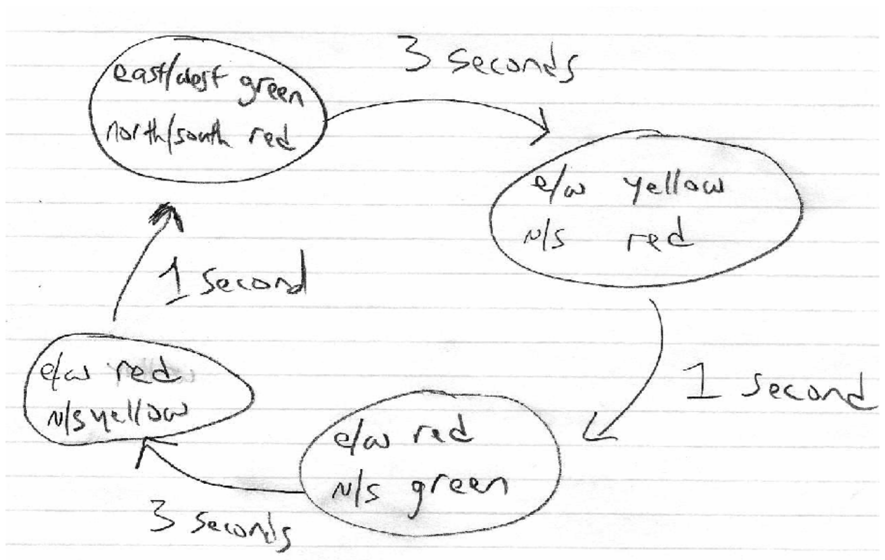
This file: http://positron.hep.upenn.edu/wja/p364/2014/files/lab26.pdf

Useful files: http://positron.hep.upenn.edu/wja/p364/2014/files/?C=M;O=D

Part 1

So far, the things we have been able to do with logic gates and flip-flops have been somewhat dull in comparison to the things that the Arduino can do. The way to make digital logic start to do things that begin to resemble the things that a computer can do is by implementing a **finite-state machine** (FSM). http://en.wikipedia.org/wiki/Finite-state_machine

The aim is that by the end of Lab 27, we will have implemented an FSM that is sophisticated enough to give you some sense of how a real computer does its work. But we have to start somewhere, so let's start by designing a state machine to control a traffic signal. Here is a **state diagram** for a simple traffic light. "E/W" means "east/west," and "N/S" means "north/south."



Initially, the east/west traffic can go, and the north/south traffic is stopped. After a few seconds, the E/W light turns yellow. Then after just one second, the E/W light turns red, and the N/S light turns green. After a few seconds, the N/S light then turns yellow. Finally, after one more second, we return to the initial state.

This is an absurdly fast traffic signal, but if we made it too realistic, it would be pretty boring to watch in the lab.

Later, we will see a more formal way to design a state machine, which is useful for complicated problems. But it turns out that the easiest way to implement a relatively simple FSM is just to use the value of a counter to tell you which state you are currently in.

We'll start from Lab 25's eight-bit counter that increments once per second, which we called `count1Hz[7:0]`. We can just use the lowest 3 bits of `count1Hz`, i.e. we can use the bit-vector `count1Hz[2:0]`, to count over and over again from 0 to 7, updating once per second. For this very simple traffic signal, this is just what we want.

You can write something like `wire [2:0] ticks = count1Hz[2:0];` and then use the value of `ticks` to control the traffic light display. Here is a possible mapping from the value of `ticks` to the state of the traffic signal:

`ticks`	east/west	north/south
0	green	red
1	green	red
2	green	red
3	yellow	red
4	red	green
5	red	green
6	red	green
7	red	yellow

From the parts supply, find your breadboard, six 100 Ω resistors, two red LEDs, two yellow LEDs, two green LEDs, and a length of breadboard wire, at least 10 cm or so.

You don't need any external power for your breadboard. The BASYS2 board will provide all necessary power. At the top of the BASYS2 board are four 6-pin connectors for "user I/O" connections (in other words, your own inputs and outputs).

From left to right, the four connectors are labeled JA, JB, JC, JD.

From left to right, the pins of (for example) the right-hand connector are `jd[1]`, `jd[2]`, `jd[3]`, `jd[4]`, ground V_{CC}, where in this case V_{CC} means the +3.3 V power that is used by the FPGA's input/output pins.

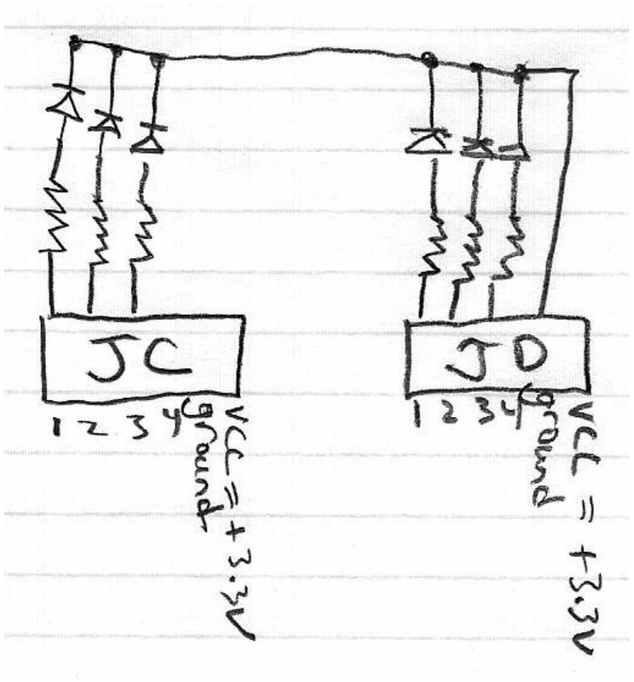
A logic LOW output of an FPGA pin is 0.0 V (ground). A logic HIGH output of an FPGA pin (for this FPGA) is +3.3 V. (This is lower than the +5 V used by the Arduino.)

You want to wire up the six LEDs so that each LED lights up when the corresponding FPGA output pin is driven HIGH. I suggest using `jc[1]`, `jc[2]`, `jc[3]` for the north/south traffic signal, and using `jd[1]`, `jd[2]`, `jd[3]` for the east/west traffic signal. But it's up to you.

Push one lead of a 100 Ω resistor into each of the connector holes corresponding to `jc[1]`, jc[2], jc[3], jd[1], jd[2], jd[3], and push the other lead of each resistor into a separate column of your breadboard.

Push one end of your 10 cm length of breadboard wire into the "ground" hole for either the JC or the JD connector. Note that "ground" is second from the end, while "V_{CC}" is on the end.

Wire up each LED in series with a resistor. Connect the negative end of each LED to the long "ground" strip on your breadboard. Connect the breadboard's ground to the BASYS2 board's ground with the breadboard wire.



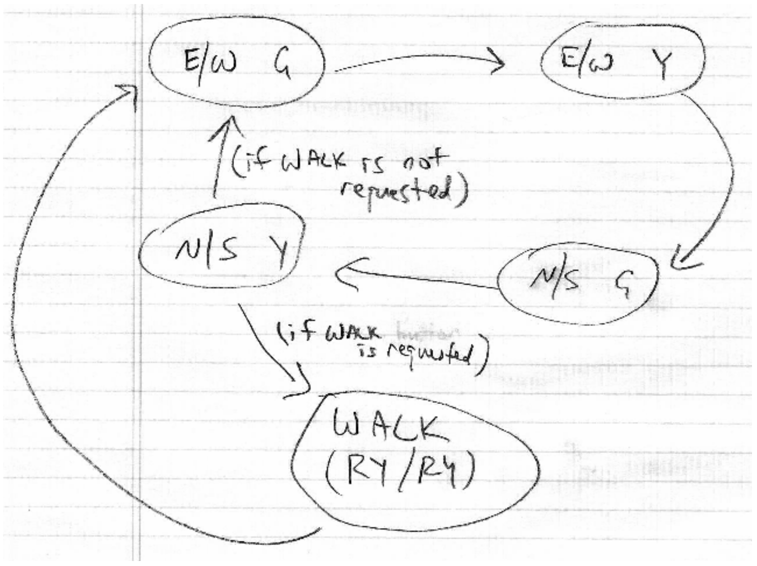
Start with the zipped Xilinx project provided at this link:
http://positron.hep.upenn.edu/wja/p364/2014/files/lab26.zip
If you want to look at just the Verilog code, it's at this link:
http://positron.hep.upenn.edu/wja/p364/2014/files/lab26_part1.v
I have pre-wired (in the FPGA code) the outputs jc[4:1] to sliding switches sw[7:4] and the outputs jd[4:1] to sliding switches sw[3:0] so that you can test out your LED connections before working on your Verilog program. Give them a try and make sure that they light up as you expect.

Now assign each of the corresponding FPGA output pins to HIGH or LOW depending on the value of ticks in your Verilog code. For instance, the east/west yellow LED should only light up if (ticks==3) .

Congratulations! You've just built a working traffic signal! Briefly sketch below how you solved this problem:

Part 2

The state machine from Part 1 never has to change what it does in response to external input. A more interesting traffic signal would respond to the press of a WALK button. Let's add a WALK feature to our traffic light, like this:



Now there are two possible next states from the "north/south yellow" state: if there is a pedestrian waiting, then the next state should be WALK. If there is no pedestrian waiting, then the next state should be "east/west green." In the WALK state, cars from all directions should see a red light, and a blue LED (alas, it didn't occur to us to buy white LEDs) lights to tell pedestrians that it is safe to cross the street.

I suggest using `jc[4]` for the blue LED. The easiest form of the "pedestrian waiting" signal is just to use a sliding switch, e.g. `sw[7]`, even though a button would be nicer. If you prefer to use a button, you need to combine it with a set/reset latch, so that the button "sticks" until the next walk state has occurred.

Here is a straightforward way that we can use a 4-bit counter to include the walk phase of the light:

`ticks`	east/west	north/south	walk (blue)
0	green	red	off
1	green	red	off
2	green	red	off
3	yellow	red	off
4	red	green	off
5	red	green	off
6	red	green	off
7	red	yellow	off
8	red	red	ON
9	red	red	ON
10	red	red	ON
11	red	red	ON
12	red	red	ON
13	red	red	ON
14	red	red	ON
15	red	red	ON

To implement this, you need to change `ticks` from 3 bits to 4 bits wide, and you need to update the LED logic to incorporate all 16 possible values of `ticks`. Note that you can use the standard less-than (`<`) and greater-than (`>`) operators in Verilog with integer values, or you can pick apart the bits of `ticks[3:0]` if you find that easier.

You also need to change the logic for `count1Hz_next`. Instead of simply

```
  wire [7:0] count1Hz_next = count1Hz + 1;
```

you need something like

```
  wire [7:0] count1Hz_next =
    ((count1Hz==7 && !sw[7]) || (count1Hz==15)) ? 0 : count1Hz + 1;
```

The result is that you go from `ticks==7` to `ticks==8` if the walk request `sw[7]` is HIGH, but you go from `ticks==7` back to `ticks==0` if the walk request is LOW. Also, you should go from `count1Hz==15` back to `count1Hz==0` so that you don't confuse yourself with the fact that `count1Hz` has 8 bits while `ticks` has only 4 bits.

In retrospect, it was a bit inelegant of me to use separate names for `count1Hz` and for `ticks`. It makes things unnecessarily confusing.

Sketch briefly how you modified your program to include the separate WALK phase.

If you have time, use an SR latch (the code for which is at the bottom of the Verilog file) that is SET when you push `btn[3]` and is RESET when `ticks[3]` is HIGH. Now use the Q output of this SR latch for the walk request signal. Also, make one of the decimal point LEDs light up while the walk-requested signal is true, so that you can see that a walk has been requested. If you do this extra part, please show us your results!

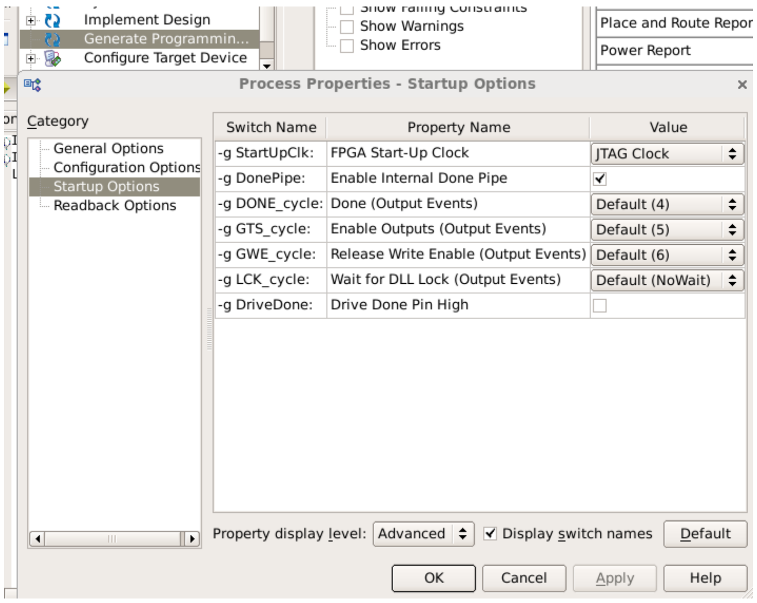
Part 3

This next part does not require you to do any programming, but there are some parts of my program that I want you to try to follow. Here is the Verilog file for Part 3:
http://positron.hep.upenn.edu/wja/p364/2014/files/lab26_part3.v

This time I won't paste the whole Verilog source code into this document, because I don't want you to be overwhelmed by staring at 200 lines of program code all at once. There are only a few parts that you need to study.

Make and compile an ISE project using the above Verilog file and the usual "user constraints" file, `basys2.ucf`
http://positron.hep.upenn.edu/wja/p364/2014/files/basys2.ucf

The FPGA type is Spartan3E / XC3S100E / CP132 / -4. One annoying thing you need to do after creating your new ISE project is to right-click on "Generate Programming File," and click "Process Properties." Then in the dialog box, click "Startup Options." Make sure the "FPGA Start-Up Clock" is set to "JTAG Clock."



Process Properties - Startup Options

Category
- General Options
- Configuration Options
- Startup Options
- Readback Options

Switch Name	Property Name	Value
-g StartUpClk:	FPGA Start-Up Clock	JTAG Clock
-g DonePipe:	Enable Internal Done Pipe	✓
-g DONE_cycle:	Done (Output Events)	Default (4)
-g GTS_cycle:	Enable Outputs (Output Events)	Default (5)
-g GWE_cycle:	Release Write Enable (Output Events)	Default (6)
-g LCK_cycle:	Wait for DLL Lock (Output Events)	Default (NoWait)
-g DriveDone:	Drive Done Pin High	

Property display level: Advanced ✓ Display switch names

By the way, if you actually look through the `basys2.ucf` file in your web browser or in the Xilinx ISE software, you'll see that its main purpose is to assign human-usable names (like `led`, `sw`, `btn`, `seg`) to the FPGA's built-in input/output pins. The bottom of the FPGA looks something vaguely like the above-right picture. (This is not exactly the same pattern as the FPGA on your board, but it illustrates the same idea.)

The UCF file contains lines that map the (row,column) pin names of the FPGA's physical package into mnemonic names that are easier to use inside your own logic circuit. For example, here are the pin assignments for the 7-segment display:

```
# Pin assignment for DispCtl
# Connected to Basys2 onBoard 7seg display
NET "seg<0>" LOC = "L14"; # Bank = 1, Signal name = CA
NET "seg<1>" LOC = "H12"; # Bank = 1, Signal name = CB
NET "seg<2>" LOC = "N14"; # Bank = 1, Signal name = CC
NET "seg<3>" LOC = "N11"; # Bank = 2, Signal name = CD
NET "seg<4>" LOC = "P12"; # Bank = 2, Signal name = CE
NET "seg<5>" LOC = "L13"; # Bank = 1, Signal name = CF
NET "seg<6>" LOC = "M12"; # Bank = 1, Signal name = CG
NET "dp" LOC = "N13"; # Bank = 1, Signal name = DP
```

In this case, the nice folks at Digilent, Inc., also included (as a comment) the pin names used by the manufacturer of the 4-digit 7-segment LED display. ("CA" for "cathode A," etc.)

The program that you are compiling for Part 3 contains a small RAM (Random Access Memory). This RAM has 8 address lines and 8 data lines. So it can store 256 distinct values, each of which is 8 bits wide. Here is the Verilog code fragment that instantiates the RAM into the top-level module:

```
    wire [7:0] ram_dataout, ram_datain, ram_address;
    wire       ram_write;
    ram256x8 myram ( ram_dataout,  // read back RAM contents at given address
                     clock,        // updates are synchronous to clock
                     ram_write,    // update RAM contents if this is asserted
                     ram_address,  // address at which to read/write the RAM
                     ram_datain);  // new data input in case ram_write is TRUE
    assign ram_datain = sw[7:0];
    assign led[7:0] = ram_dataout;
```

Here is the Verilog code that actually tells the Xilinx compiler how to create the RAM:

```
// This is one example of a Verilog idiom for making a RAM that the
// Xilinx compiler understands how to convert into an FPGA memory; we
// haven't studied the Verilog language in enough detail for this to
// make sense to you, but if you're curious, I can point you to where
// you can learn more.
module ram256x8 ( output [7:0] dataout,
                  input        clock,
                  input        writeenable,
                  input  [7:0] address,
                  input  [7:0] datain );
    reg [7:0] memory [0:255];
    assign dataout = memory[address];
    always @ (posedge clock)
      if (writeenable) memory[address] <= datain;
endmodule
```

The first thing to notice here is that the RAM has 8 input pins called **address**. This gives the RAM 256 different places at which it can store potentially useful information.

The next thing to notice is that there are 8 output pins called **dataout**. These 8 lines read back whatever value is currently stored in the RAM at the address given by **address**.

In normal computer programming, this would be like having an array caled something like **memory**. When you change the value of **address**, the **dataout** lines get the current value of **memory**[**address**], i.e. they get whatever value is stored at that memory location.

The way I have set up this RAM, it has a separate set of 8 pins called **datain**. There are also single input pins called **clock** and **writeenable**. The way this works is that most of the time **writeenable** is FALSE, and the RAM is just reading back whatever is already stored at location **address**.

But if **writeenable** is TRUE on the rising edge of the **clock**, then the contents of the memory stored at **address** will be updated with whatever 8-bit value is presented on the **datain** lines.

Most memory chips that are used in making real computers don't have separate `datain` and `dataout` pins; the same pins are used for both purposes. But that requires introducing the added complication of having wires that are sometimes used as inputs and are sometimes used as outputs. On a printed circuit board, where wires are a precious resource, it makes sense to go to that hassle. Inside the guts of an FPGA, where inputs and outputs are free, it is easier to use separate wires for **datain** and **dataout**.

Another thing to notice while compiling this program is that the Xilinx compiler was smart enough to "infer" from the Verilog code that it needs to wire up one 256×8 RAM, one 26-bit counter, an 8-bit adder/subtracter unit, a small Read Only Memory, and some multiplexers. This is nothing very profound, but it's further evidence that the Verilog program is not really a program; it's just a description of a schematic diagram for the circuit that the FPGA will be internally wired up to implement.

```
Advanced HDL Synthesis Report
Macro Statistics
# RAMs                                                   : 1
 256x8-bit dual-port block RAM                           : 1
# ROMs                                                   : 1
 16x7-bit ROM                                            : 1
# Adders/Subtractors                                     : 1
 8-bit addsub                                            : 1
# Counters                                               : 1
 26-bit up counter                                       : 1
# Registers                                              : 8
```

```
 Flip-Flops                                            : 8
# Multiplexers                                         : 2
 1-bit 4-to-1 multiplexer                              : 1
 4-bit 4-to-1 multiplexer                              : 1
```

Another thing potentially to notice in the big stream of console output from the compiler is that this program uses a mere few percent of the possible resources of this FPGA. This is one of the smallest Xilinx FPGAs, and the design we are compiling here uses only a few percent of the chip.

```
Logic Utilization:
  Number of Slice Flip Flops:            33 out of   1,920    1%
  Number of 4 input LUTs:                70 out of   1,920    3%
Logic Distribution:
  Number of occupied Slices:             49 out of     960    5%           : 1
```

OK, on to the real work. If you skim through the main module of the Verilog file (called `lab26_part3`), you'll see that the action is controlled by a roughly 3 Hz clock, which is obtained by dividing the 50 MHz built-in clock by 2^{24}.

Notice the 8-bit-wide flip-flop called **ramaddrff** that holds the current value of the RAM's **address** lines. Notice the program code that updates the RAM's address:

```verilog
   wire [7:0] ramaddr_next;
   dffe_8bit ramaddrdff ( ram_address,
                          clock,
                          ramaddr_next,
                          1, 0);
   assign ramaddr_next = btn[0] ? ram_address+1 :
                         btn[1] ? ram_address-1 :
                         btn[2] ? sw[7:0]       :
                                  ram_address   ;  // default: unchanged
   assign ram_write = btn[3];
```

See if you can follow the logic that does this:

- If you hold down button 0, the address ticks up by one for each tick of the 3 Hz clock.
- If you hold down button 1, the address ticks down by one for each tick of the 3 Hz clock.
- If you hold down button 2, the address jumps (on the next clock cycle) to the 8-bit value that is entered by the 8 sliding switches.

- If you hold down button 3, the contents of the memory at the current address are updated to contain the 8-bit value that is entered by the 8 sliding switches.
- The left-hand decimal point LED blinks once per clock tick so that you can see when the updates are going to happen.
- At any given time, the two left digits of the 7-segment display show you the 8-bit **ram_address** value, and the two right digits of the 7-segment display show you the 8-bit **ram_dataout** value.

Work with this circuit for a while and see if you can load up the contents of the memory so that, for example, the memory locations 00, 01, 02, 03, 04, 05, 06, 07, 08, 09 contain the squares of the first ten integers.

- You can do it either in hex or in decimal, but it's probably easier to read if you do it in decimal.
- Enter the ten values at the ten different addresses, then go back and check that they are all still stored where you wrote them.
- Actually, you probably want to check after the first one or two writes that the memory works as you expect, so that you don't waste time doing operations whose result is not what you might expect.

Keep tinkering with this circuit and asking us questions until you understand the idea of a Random Access Memory. http://en.wikipedia.org/wiki/Random-access_memory

Part 4

Next, we're going to implement this state machine for a vending machine:

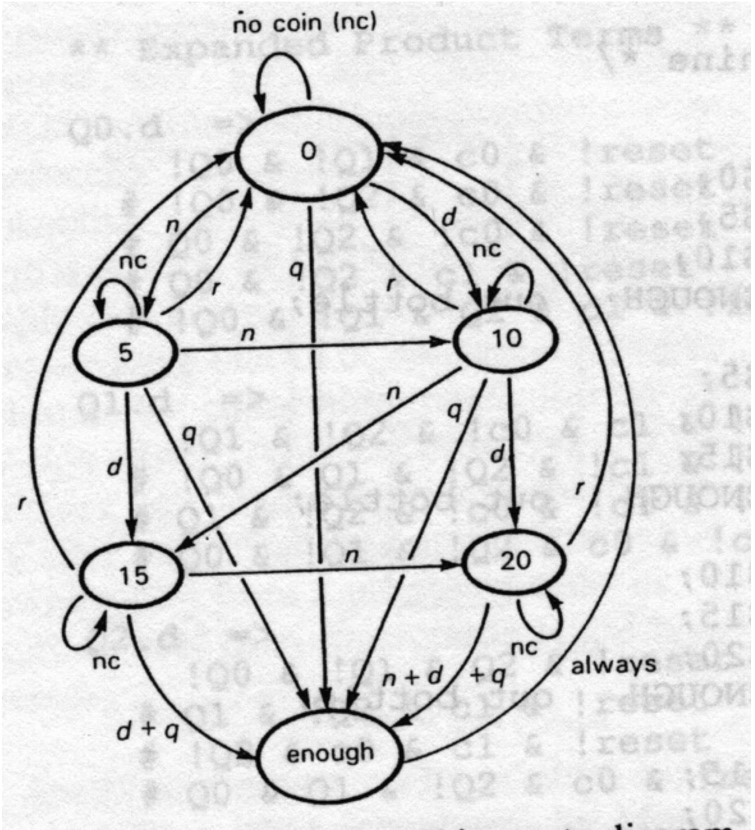


Figure 8.80. Vending machine state diagram.

We'll use the right-hand button to indicate **nickel** (i.e. you insert a 5-cent coin), then dime, then quarter. The left-hand button will reset the vending machine to its initial state, just in case.

I have provided the next-state logic for the **reset** and the **nickel** buttons. Your mission **(!)** is to extend this logic to include the **dime** and **quarter** buttons as well.

Here is my Verilog code:
http://positron.hep.upenn.edu/wja/p364/2014/files/lab26_part4.v

Be sure to bug us with questions about anything that you don't understand!

Once your vending machine is working, briefly outline here the changes that you made to `lab26_part4.v`.

Challenge: If you have extra time, add two more states called VEND1 (state 6) and VEND2 (state 7).

- The next state from ENOUGH should be VEND1 (unconditionally)
- The next state from VEND1 should be VEND2
- The next state from VEND2 should be GOT0
- Make the `8888` pattern and the LEDs go on for the ENOUGH and the VEND2 states but make them do something boring for the VEND1 state, so that you get two flashes when a soda can is dispensed.

Bigger challenge: If you have lots of time, find a way to make a sequence of states after ENOUGH that blinks the LEDs in an even more interesting pattern, e.g. zooming from right to left like the Knight Rider car's front lights: http://en.wikipedia.org/wiki/KITT