Physics 364, Fall 2014, Lab #27 **Name:** ______________________

(Field Programmable Gate Arrays: 4)

Monday, December 8 (section 401); Tuesday, December 9 (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/lab27.pdf

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

Part 1

The first state machine we will study today is a kind of player piano. It reads from a memory a sequence of notes and durations, and plays the tune on a speaker. For each note to be played, the memory contains a 16-bit duration (in milliseconds) and a 16-bit half-period (in microseconds). One FPGA output pin will be connected to a speaker. To play a given tone, the FPGA will drive this output pin HIGH (+3.3 V) for one half-period, then LOW (0 V) for one half-period, then HIGH for one half-period, then LOW for one half-period, and so on. It continues to play this tone until the desired duration has elapsed. So to play the A note ($f = 440$ Hz, $T = \frac{1}{f} = 2272.7\ \mu$s) above middle C for a duration of one second, we would drive the output pin HIGH for 1136 μs, then LOW for 1136 μs, and so on, until 1000 ms have elapsed.

A `.zip` archive of the files for Part 1, including a pre-made ISE project file, are at http://positron.hep.upenn.edu/wja/p364/2014/files/lab27.zip

If you want to look at the Verilog file on its own, you can either view it within the ISE software or follow this link:
http://positron.hep.upenn.edu/wja/p364/2014/files/lab27_part1.v

A key goal of today's lab is for you to see how much more powerful a state machine becomes when it is combined with a memory that contains a list of tasks for it to carry out. This concept may inspire you to go in your own direction. Feel free at any time to modify this program to do something slightly different, or to ask us for help with modifying it to suit your interests. If you're uninspired by today's lab, feel free to ponder project ideas.

ROM

To store the desired information, we use a 256×16 ROM (Read Only Memory), i.e. 256 storage locations, each of which is 16 bits wide. That means that the input to our ROM is 8 address lines, `address[7:0]`, and the output of our ROM is 16 data lines, `dataout[15:0]`. There are several ways to describe a ROM in Verilog. The method used here maps directly onto last week's discussion of a ROM as a special case of a multiplexer.

```
// 256x16 ROM, i.e. 256 storage locations, each of which is 16 bits wide;
// how exactly you get Verilog to infer a ROM, a RAM, etc. is somewhat
// idiomatic and depends on the FPGA vendor's software (e.g. Xilinx);
// this is one acceptable way to tell Xilinx that you want a ROM
```

```
module rom256x16 ( output [15:0] dataout,
                   input  [7:0]  address );
    wire [7:0] A = address;  // abbreviation to reduce typing
    // white piano keys starting from middle C:
    //    note:     C    D    E    F    G    A    B   C
    // f    (Hz):   262  294  330  349  392  440  494 523
    // T/2 (us):  1911 1703 1517 1432 1276 1136 1012 956
    assign dataout =
      //    duration       halfperiod
      A==  0 ? 1000  :  A==  1 ? 1517  :
      A==  2 ? 1000  :  A==  3 ? 1703  :
      A==  4 ? 1000  :  A==  5 ? 1911  :
      A==  6 ? 1000  :  A==  7 ? 1703  :
      A==  8 ? 1000  :  A==  9 ? 1517  :
      A== 10 ? 1000  :  A== 11 ? 1517  :
      A== 12 ? 1000  :  A== 13 ? 1517  :
      A== 14 ? 1000  :  A== 15 ? 1703  :
      A== 16 ? 1000  :  A== 17 ? 1703  :
      A== 18 ? 1000  :  A== 19 ? 1703  :
      A== 20 ? 1000  :  A== 21 ? 1517  :
      A== 22 ? 1000  :  A== 23 ? 1276  :
      A== 24 ? 1500  :  A== 25 ? 1276  :
      A== 26 ? 1000  :  A== 27 ?    0  : // rest 1s
...
      A==250 ?    0  :  A==251 ?    0  :
      A==252 ?    0  :  A==253 ?    0  :
      A==254 ?    0  :  A==255 ?    0  : 0 ;
endmodule
```

We store `duration` in even-numbered addresses and `halfperiod` in odd-numbered addresses. So we need to read from two successive addresses to play a given note. We use the special case `halfperiod==0` to represent a musical "rest," i.e. to represent silence, e.g. at $A = 27$.

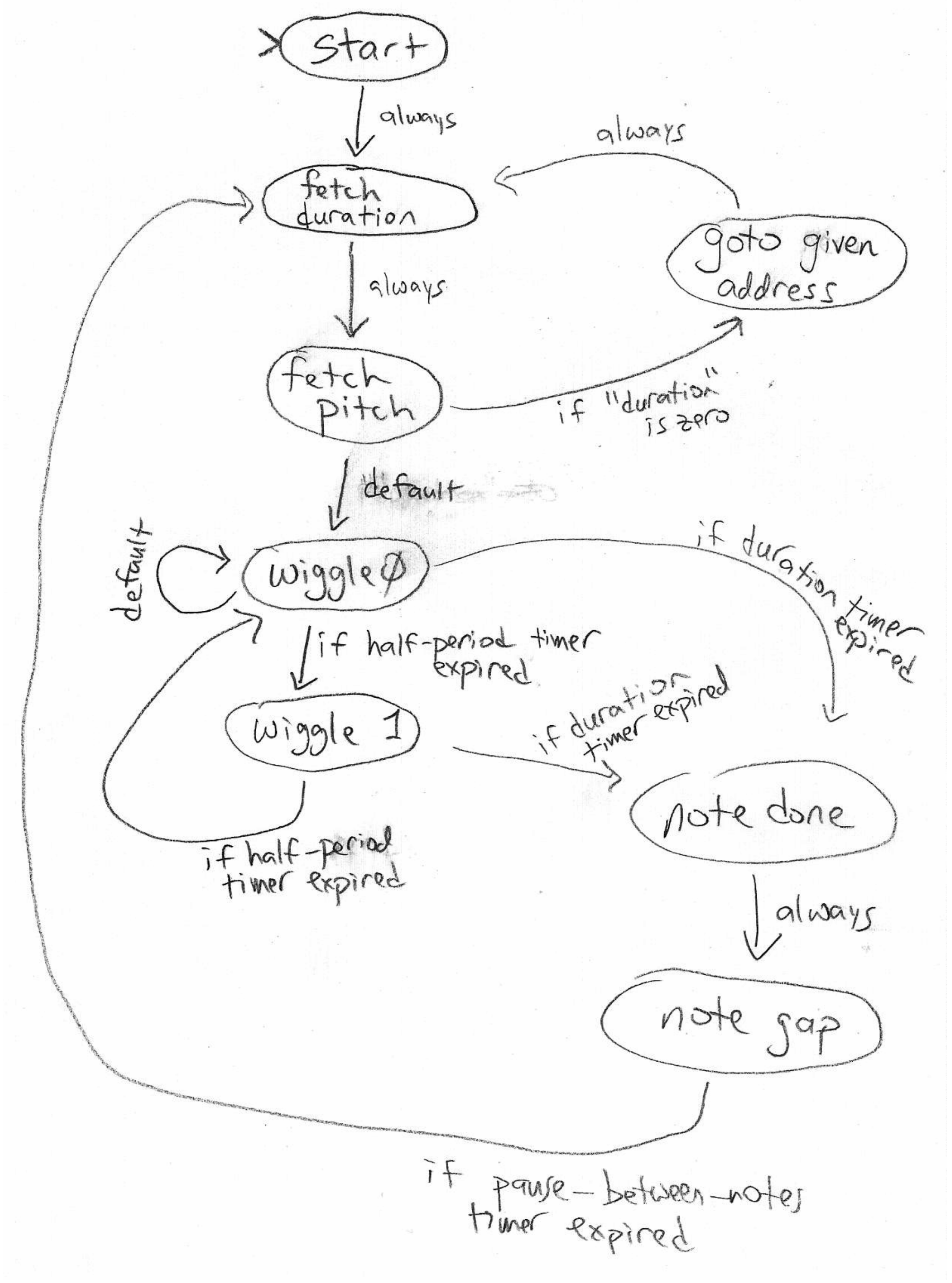
In the special case `duration==0`, we use the corresponding `halfperiod` value to inducate the address from which the machine should subsequently start reading. This allows us to do a "GOTO" operation, so that at the end of a tune, we can go back and play the same tune again.

State diagram

Shown below is a state diagram for the machine that reads this ROM to play out a tune. We use a Verilog statement called "`localparam`" to assign state names such as `START`, `FETCHDURA`, `FETCHPITCH`, etc., to the integer state numbers 0, 1, 2, etc.

```
    // Enumerate the possible states of our state machine
    localparam
      START=0,        // initial state: reset goes here
      FETCHDURA=1,    // fetch next note's duration (unit=millisecond)
      FETCHPITCH=2,   // fetch next note's half-period (unit=microsecond)
      GOTO=3,         // special case (duration==0): GOTO new memory address
      WIGGLE0=4,      // output bit is LOW for one-half period
```

```
        WIGGLE1=5,      // output bit is HIGH for one-half period
        NOTEDONE=6,     // finished playing a note
        NOTEGAP=7;      // pause briefly before moving on to next note
```


start
always
fetch duration
always
goto given address
always
fetch pitch
if "duration" is zero
default
wiggle∅
default
if duration timer expired
if half-period timer expired
wiggle 1
if duration timer expired
note done
if half-period timer expired
always
note gap
if pause-between-notes timer expired

State representation in Verilog
Since there are 8 states, we use a 3-bit-wide D-type flip-flop to hold the current state of the machine, called `state`. The logic to decide which state to enter on the next clock cycle manipulates the wires called `nextstate`.

Here is the 3-bit-wide D-type flip-flop that holds the present state of the machine:

```
    // We'll use a 3-bit D-type flip-flop to hold the state of the FSM
    wire [2:0] state, nextstate;
    dffe_Nbit #(.N(3)) mystatedff
      (.q(state),        // on each rising edge of the clock, the flip-flop
       .d(nextstate),    //   copies 'nextstate' (D) to 'state' (Q)
       .clock(clock),    // clocked at 1 MHz
       .enable(1),       // always enabled
       .reset(reset));   // reset to initial state by pushing button
```

Next-state logic

The `nextstate` logic determines which state we go into on the next clock cycle:

```
    // Compute next state based on current state and pertinent conditions
    assign nextstate =
      (reset || stopthenoise)             ? START      :  // go here if reset
      (state==START || forcenewaddress)   ? FETCHDURA  :
      (state==FETCHDURA)                  ? FETCHPITCH :
      (state==FETCHPITCH && zeroduration) ? GOTO       :  // dur==0 means GOTO
      (state==FETCHPITCH)                 ? WIGGLE0    :
      (state==GOTO)                       ? FETCHDURA  :
      (state==WIGGLE0 && durationup)      ? NOTEDONE   :  // end of this note?
      (state==WIGGLE0 && zeroperiod)      ? WIGGLE0    :  // rest vs. tone
      (state==WIGGLE0 && wigglenow)       ? WIGGLE1    :  // wiggle up & down
      (state==WIGGLE1 && durationup)      ? NOTEDONE   :
      (state==WIGGLE1 && wigglenow)       ? WIGGLE0    :
      (state==NOTEDONE)                   ? NOTEGAP    :
      (state==NOTEGAP && notegapdone)     ? FETCHDURA  :  // pause betw. notes
      /* default: stay in same state */     state      ;
```

If you compare the above code snippet with the state diagram, you should see that the code is expressing the same state transitions as the diagram.

- If we press the **reset** button (`btn[0]`, the right-hand button), or if we slide the **stopthenoise** switch (`sw[0]`, the right-hand switch) upward, then the machine goes to the `START` state. This is also where it starts when we first program the FPGA.
- From the `START` state, the next state is always the `FETCHDURA` state (where we fetch from ROM the duration of the next note)
  - The `FETCHDURA` state goes to the next ROM address (which should be an even-numbered address) and records the 16-bit "duration" (in milliseconds) of the next note to play.
  - We will also go into the `FETCHDURA` state if the **forcenewaddress** button is pressed (which is `btn[1]`, the second button from the right). This button allows the you to directly modify the machine's address, so that the machine subsequently starts playing music from that address.
- From the `FETCHDURA` state, the next state is always the `FETCHPITCH` state.

  - The `FETCHPITCH` state goes to the next ROM address (which should be an odd-numbered address) and records the 16-bit “halfperiod” (in microseconds) of the next note to play.

- From the `FETCHPITCH` state, there are two possible next states:
  - By far the most common transition is to go to the `WIGGLE0` state, which is used along with `WIGGLE1` to make the output bit oscillate (wiggle) back and forth to make the desired musical tone.
  - But in the special case in which the “duration” value is zero, we interpret this to mean that the “halfperiod” value actually represents the next memory address from which we should continue reading musical notes. In that case, the next state will be the `GOTO` state, to handle this change of address.

- If we happen to have wound up in the `GOTO` state, the next state from there is always the `FETCHDURA` state, from which the process begins of fetching from memory the next musical note to play.
  - Again, the point of the `GOTO` state is to make the memory address jump discontinuously to a new location. Normally the memory address just increases by one step at a time, incrementing once to find a new duration and then incrementing again to find a new pitch.
  - We will look at the address-update logic in a moment.

- While a note is playing, we go back and forth between the `WIGGLE0` state and the `WIGGLE1` state.
  - If we’re in `WIGGLE0`, the output bit (to the speaker) is LOW; if we’re in `WIGGLE1`, the output bit is HIGH.
  - We reset a microsecond counter to zero before entering the `WIGGLE0` or `WIGGLE1` state. When the counter reaches the `halfperiod` value, we know it is time to transition to the other state, so that the output pin goes back and forth between LOW and HIGH at the correct oscillation period.
  - Meanwhile, in the `FETCHPITCH` state, we reset a millisecond counter to zero. If the number of milliseconds elapsed reaches the `duration` value, we know it is time to transition to the `NOTEDONE` state, to finish up playing this note and move on.
  - There is one more special case: if `halfperiod` is zero, indicating that we should play silence (a rest) rather than a note, then we just stay in the `WIGGLE0` state instead of going back and forth between `WIGGLE0` and `WIGGLE1`.

- From the `NOTEDONE` state, we always go to the `NOTEGAP` state.
  - The purpose of this state is to put a small (currently 25 milliseconds) gap between notes, so that your ear can hear the difference between two consecutive eighth-notes and a single quarter-note at the same pitch.
  - I didn’t realize that I needed this state until I heard how terrible *Mary Had a Little Lamb* sounded without it.

- If not otherwise specified, we just remain in the same state for the next clock cycle.

ROM address & instantiation

Next, the ROM connections. The memory address from which to read is stored in an 8-bit D-type flip-flop. We again use the generalized `dffe_Nbit` with `#(.N(8))` to indicate an 8-bit-wide flip-flop.

```
    // Use an 8-bit D-type flip-flop to hold the memory address
    // from which the state machine is reading
    wire [7:0] memory_addr, memory_nextaddr;
    dffe_Nbit #(.N(8)) memaddrff
      (.q(memory_addr),
       .d(memory_nextaddr),
       .clock(clock),
       .enable(1), .reset(0));
```

Next, we instantiate the ROM and make connections for address (input) and data (output):

```
    // We will store the desired tune to be played in a ROM (Read-Only
    // Memory) with 256 locations, each of which can store a 16-bit number;
    // declare wires to store the 'dataout' value from the memory, as well
    // as the 'duration' and 'halfperiod' that will be copied (via flip-flops)
    // from the 'memory_data' wires
    wire [15:0] memory_data, duration, halfperiod;
    rom256x16 myrom1
      (.dataout(memory_data),   // data comes out
       .address(memory_addr));  // address goes in
```

Here is the logic to decide what ROM address to read from on the next clock cycle:

```
    // Compute what memory address we will read from on next clock cycle
    assign memory_nextaddr =
      (btn[1])            ? sw[7:0]       :  // button1: goto switch address!
      (state==START)      ? 0             :  // start reading from zero
      (state==FETCHDURA)  ? memory_addr+1 :  // after reading duration or
      (state==FETCHPITCH) ? memory_addr+1 :  //   pitch, increment address
      (state==GOTO)       ? halfperiod    :  // special GOTO command
      /* default: same */   memory_addr   ;  // otherwise stay unchanged
```

As a special case, if we hold down `btn[1]`, the address is loaded from the `sw[7:0]` sliding switches. On the `START` state, we start out at address zero. If we are in either the `FETCHDURA` state or the `FETCHPITCH` state, we want to increment the address by one so that the next read from the ROM will happen from the next memory loacation after this one. If we are in the `GOTO` state, then the `halfperiod` value contains (instead of a musical note) the next address from which we should continue reading.

Duration and halfperiod flip-flops

We store the duration and the half-period for the current note in a pair of 16-bit-wide flip-flops. The `duration` wires and the `halfperiod` wires connect the outputs of these flip-flops, respectively. The `duration` flip-flop is only enabled in the `FETCHDURA` state; and the `halfperiod` flip-flop is only enabled in the `FETCHPITCH` state. In both cases, the **D** (data) inputs of the flip-flops are from the **memory_data** outputs of the ROM. We're just writing down the value that we read from the ROM in either the `FETCHDURA` or the `FETCHPITCH` state. The fact that `memory_data` (the ROM output) is connected to the **D** inputs of these two flip-flops is a key point — if you don't see why it is connected this way, please ask.

```
// Use a 16-bit D-type flip-flop to hold the desired duration (in ms)
// the note we are playing (or are about to play)
dffe_Nbit #(.N(16)) durationff
  (.q(duration),              // output goes to 'duration' wires
   .d(memory_data),           // input data come from memory output data
   .enable(state==FETCHDURA), // enabled only in the FETCHDURA state
   .clock(clock), .reset(0));
assign zeroduration = (duration==0);  // indicates special 'GOTO' command

// Use a 16-bit D-type flip-flop to hold the half-period (in us)
// of the note we are playing (or are about to play)
dffe_Nbit #(.N(16)) halfperiodff
  (.q(halfperiod),              // output goes to 'halfperiod' wires
   .d(memory_data),             // input data come from memory output data
   .enable(state==FETCHPITCH),  // enabled only in the FETCHPITCH state
   .clock(clock), .reset(0));
assign zeroperiod = (halfperiod==0);  // indicates rest (silence) vs. note
```

Counting microseconds and milliseconds.
Now we have two 16-bit counters. One of them increments once per microsecond. (This is easy, because we set up our master clock to run at 1 MHz.)

The second counter increments once per millisecond. But we still clock it with the same 1 MHz clock. We use a once-per-millisecond pulse called `pulse_1kHz` to enable the millisecond counter. We do this because we want all of the logic in our state machine to be synchronous to a single clock.

```
// Use a 16-bit counter to count off microseconds until the next
// time the wire driving the speaker needs to wiggle up or down
wire [15:0] count_usec;
wire        reset_usec;
counter_Nbit #(.N(16)) myuseccounter
  (.q(count_usec),
   .clock(clock),        // clocked by 1 MHz clock
   .enable(1),           // always enabled
   .reset(reset_usec));  // reset to zero when starting new half-period
// We start a new half-period immediately after FETCHPITCH or once
// the number of microseconds exceeds the desired half-period
assign reset_usec = (state==FETCHPITCH || wigglenow);
assign wigglenow  = (count_usec==halfperiod);

// Use a 16-bit counter to count off milliseconds until the end
// of the note that we are playing (or are about to play)
```

```
    wire [15:0] count_msec;
    wire        reset_msec = (state==FETCHPITCH || state==NOTEDONE);
    counter_Nbit #(.N(16)) mymseccounter
      (.q(count_msec),       // elapsed millisecs (i.e. counter value)
       .clock(clock),        // counter operates from 1 MHz clock
       .enable(pulse_1kHz),  // enable only once per millisecond
       .reset(reset_msec));  // reset when starting or ending a note
    assign durationup = (count_msec==duration);
    assign notegapdone = (count_msec==25);  // 25 msec gap between notes
```

Other connections

Finally, notice that the left-hand pin on the **JC** connector is the wire that wiggles between LOW and HIGH when the state machine goes back and forth between the `WIGGLE0` and `WIGGLE1` states. You will want to connect this pin (the left-hand pin of **JC**) to one side of a small speaker. Connect the other side of the speaker to the GND wire (the 2nd pin from the right on the JC connector).

Once you compile and load the FPGA program, notice that the LEDs display in binary what note we are currently playing (i.e. its half-period).

If you hold down `btn[1]`, then the 7-segment LEDs display the 16 bits of `memory_data`, so that you can see what is stored inside the ROM. Also, when you hold down `btn[1]`, the 8 bits of `memory_address` are updated to contain whatever value is on the sliding switches. These two features together allow you to inspect the ROM contents at a given address.

If you are not holding down `btn[1]` (normally you won't be), then the digits do this:

- Digit 0 (on the right) indicates how much time is left until the current note is finished playing.
- Digit 1 indicates what state the machine is in. It looks as if it is displaying "9" all the time, but really it is just going back and forth between "4" and "5" (`WIGGLE0` and `WIGGLE1`).
- Digits 3 and 2 display the 8-bit `memory_address` from which the machine is currently playing.

```
    // Connect useful values to the inputs of 'myd4d1' (displaydigits module)
    wire [15:0] timeleft = duration-count_msec;
    assign digit0 = btn[1] ? memory_data[3:0]   : timeleft[14:7];
    assign digit1 = btn[1] ? memory_data[7:4]   : state;
    assign digit2 = btn[1] ? memory_data[11:8]  : memory_addr[3:0];
    assign digit3 = btn[1] ? memory_data[15:12] : memory_addr[7:4];
    assign dots[2:0] = btn[3:0];
    assign led = halfperiod[7:0];

    // Use 4 JC pins and 4 JD pins as outputs to drive LEDs, etc.
    assign jc[1] = (state==WIGGLE1);
    assign jc[4:2] = 0;
    assign jd[4:1] = {clock,pulse_1kHz,count_msec[0],count_msec[15]};
```

Spend some time checking out the features described above, looking through the ROM contents by moving the switches, and annoying your neighbors with my horrible rendition of *Mary Had a Little Lamb.* Go through the first several lines of the ROM contents and see how I managed to code in E, D, C, D, E, E, E. D, D, D. E, G, G, and so on.

Other tunes: the pips

If you slide the switches to contain the decimal value 64 (binary 01000000) and momentarily hold down `btn[1]`, then you will hear over and over again, every ten seconds, the pips[1] that are played e.g. by the BBC World Service on the hour. See if you can understand how this is coded into the ROM:

```
      A== 64 ? 1000  :  A== 65 ?    0  : // BBC news GMT "pips"
      A== 66 ?  100  :  A== 67 ?  500  : // 1 kHz for 100 ms at :59:55
      A== 68 ?  900  :  A== 69 ?    0  :
      A== 70 ?  100  :  A== 71 ?  500  : // 1 kHz for 100 ms at :59:56
      A== 72 ?  900  :  A== 73 ?    0  :
      A== 74 ?  100  :  A== 75 ?  500  : // 1 kHz for 100 ms at :59:57
      A== 76 ?  900  :  A== 77 ?    0  :
      A== 78 ?  100  :  A== 79 ?  500  : // 1 kHz for 100 ms at :59:58
      A== 80 ?  900  :  A== 81 ?    0  :
      A== 82 ?  100  :  A== 83 ?  500  : // 1 kHz for 100 ms at :59:59
      A== 84 ?  900  :  A== 85 ?    0  :
      A== 86 ?  500  :  A== 87 ?  500  : // 1 kHz for 500 ms at :00:00
      A== 88 ? 4500  :  A== 89 ?    0  :
      A== 90 ?    0  :  A== 91 ?   64  : // goto address 64
```

Do you see now why we need a `GOTO` instruction so that after reading from addresses 90 and 91 the machine can be told to loop back to address 64?

Other tunes: invention 13

Here (in the figure below) is a much more interesting tune. I actually transcribed the whole thing a few years ago (and made a couple of mistakes), but there was a limit to what would fit into the BASYS2 board's available RAM.

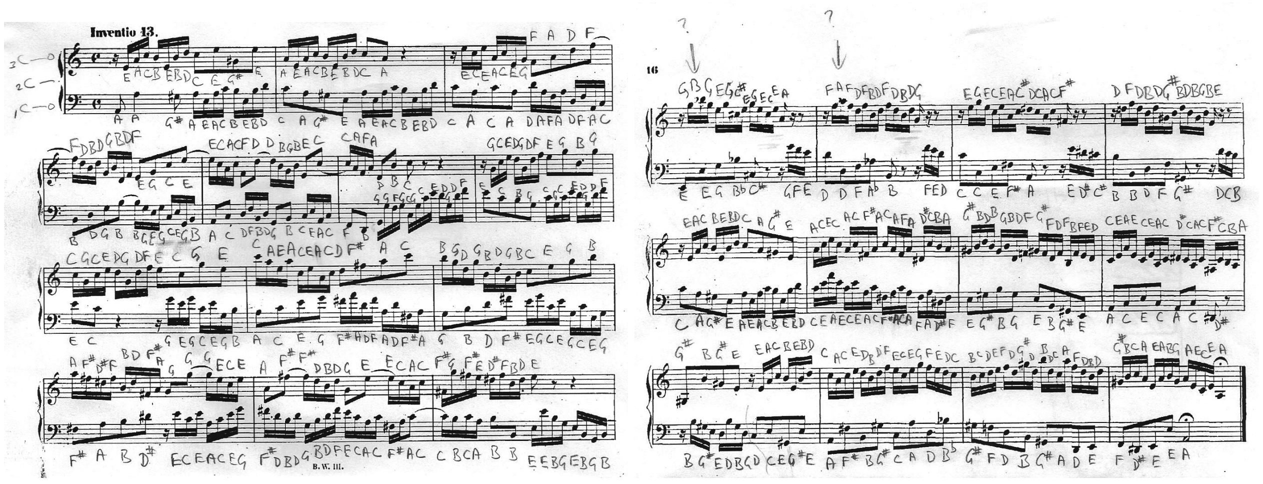
It starts at ROM address 96 (decimal), or 01100000 (binary). If you key this into the `sw[7:0]` and hold down `btn[1]` for a moment, it should start to play.

By the way, if you get tired of the music at some point, you can slide `sw[0]` up and your machine should sit quietly in the `START` state.

```
      A== 96 ?  125  :  A== 97 ?    0  : // start of RHS of invention 13
      A== 98 ?  125  :  A== 99 ? 1516  :
      A==100 ?  125  :  A==101 ? 1136  :
      A==102 ?  125  :  A==103 ?  955  :
```

[1] http://en.wikipedia.org/wiki/Greenwich_Time_Signal

```
      A==104 ?  125  :  A==105 ? 1012  :
      A==106 ?  125  :  A==107 ? 1516  :
      A==108 ?  125  :  A==109 ? 1012  :
```



Part 2

I hope that by now you get the idea that a state machine is what allows a digital logic circuit to step through the kind of sequence of operations that you might normally expect a computer to carry out. The state machine from Part 1 was much more complicated than the vending-machine or traffic-signal state machines from last week, though it does not really have such a huge number of distinct states. The main thing that makes this machine capable of doing something so complicated is that it is essentially reading a program from a memory and carrying out the instructions given in that memory.

Here is the Verilog file for Part 2:
http://positron.hep.upenn.edu/wja/p364/2014/files/lab27_part2.v
If you prefer to avoid compiling you own copy, you can use this pre-compiled `.bit` file:
http://positron.hep.upenn.edu/wja/p364/2014/files/lab27_part2.bit

The main point of this part is to remind you that a Verilog program is not really a program at all, but just a representation of a schematic diagram. I've taken the state machine from Part 1 and encapsulated it into a module called `playerpiano`, which you can look through in the Verilog source code, either in your web browser or in the ISE software.

```
// put the state machine logic from Part 1 into a module, so that we can
// instantiate TWO music machines to play at the same time!
module playerpiano
  ( output        wigglewire,    // output wire wiggles to play music
    input         clock,         // 1 MHz clock
    input         reset,         // reset button (start play from addr 0)
    input         gotobutton,    // button to start playing from 'startaddr'
    input         stopthenoise,  // switch to shut off the music
    input         pulse_1kHz,    // pulse (1 us duration) once per ms
    output [7:0]  memaddr,       // address at which to read from ROM
    input  [15:0] memdata,       // the data value returned by the ROM
    input  [7:0]  startaddr );   // addr to jump to if 'gotobutton' is pressed
```

Then on the top level of the schematic diagram, I've instantiated two copies of `playerpiano`, reading from two separate ROMs and sending their output music to two separate FPGA pins: `JC[1]` (the left-hand pin on connector JC) and `JD[1]` (the left-hand pin on connector JD).

If you load the program and plug your speaker from `JC[1]` to `GND`, you will hear the same tune as at the end of Part 1. If you move your speaker wire from `JC[1]` to `JD[1]`, then you will hear the corresponding left-hand part of Invention 13.

Notice that the two tunes are playing at the same time! The 7-segment display is showing the left-hand memory address with digits 3,2 and the right-hand memory address with digits 1,0.

If you have time (if you reach this point by 3:30pm or so), then **try the following:**

Using one opamp and several resistors, wire up a circuit that allows you to combine the audio signals from the two FPGA output pins and send the combined (summed) signal to a single speaker. As an option, you might also contemplate a way to mix the two inputs with a ratio other than 1:1. Maybe you want the right-hand tune to be twice as loud as the left-hand tune, for instance. **Please try very hard not to expose the FPGA output pins to any voltage lower than 0 V or higher than $+3.3$ V, to avoid cooking the FPGA.**

Fallback option: If you want to try something quicker, just borrow a second speaker and use linear superposition of sound waves to sum the two signals.

If you want to see this circuit playing through a big speaker (borrowed from Bill Berner) via a transistor push-pull buffer, here's a video from Fall 2012:
`https://www.youtube.com/watch?v=tHcDawYmWtE` .

Part 3

The main idea for this part is go to a step beyond the music machine, to see that the microprocessor at the heart of your Arduino, iPhone, notebook computer, etc. is really just a fancy state machine connected to a large memory.

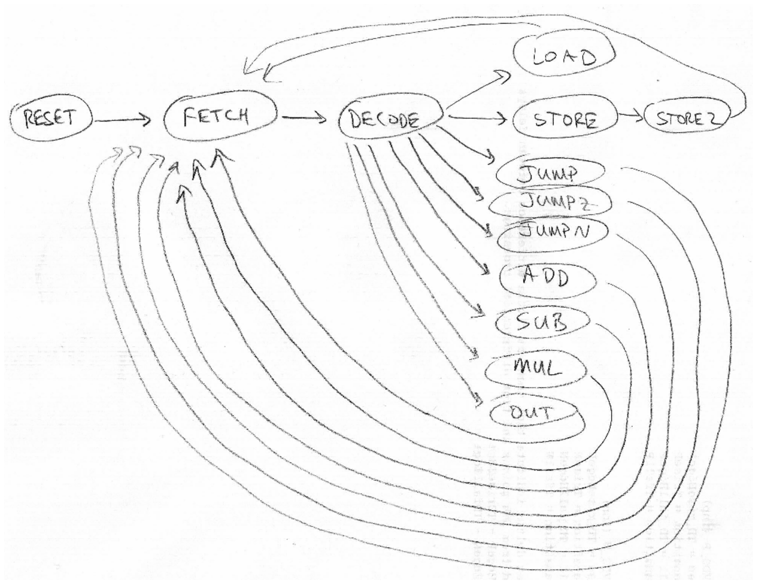
There's no Verilog programming for you to do here, but I'd like you to study and tinker with this circuit enough that you can follow how it works. In Part 4 you will modify the FPGA's memory contents to alter the program's behavior.

If you're feeling ambitious, in Part 4 you can load the FPGA's memory with an entire instruction sequence of your own design, but that's probably only feasible if you've encountered assembly-language programming somewhere in the past.

Feel free to modify my Verilog program to do something different, if you like. Don't feel obligated to follow the script. Ask us for help if you have an idea you want to try.

Here is the Verilog file for Part 3:
http://positron.hep.upenn.edu/wja/p364/2014/files/lab27_part3.v
If you decide to compile this for yourself, then you will also need to download
http://positron.hep.upenn.edu/wja/p364/2014/files/asm.hex
and put it into the same ISE project directory that contains `lab27_part3.v`. Or you can avoid compiling your own copy by using this pre-compiled `.bit` file:
http://positron.hep.upenn.edu/wja/p364/2014/files/lab27_part3.bit
If you use ADEPT to load this `lab27_part3.bit` file into your BASYS2 board, you will see your board display a sequence of prime numbers. (Make sure the sliding switches are all in the DOWN positions.)

This state diagram for our simple computer should look familiar by now:



Here is what this list of states (i.e. the association from word-like names to integer state numbers) looks like in Verilog. I just added a JUMPNZ state ("jump if nonzero"), which hasn't yet made it into the state diagram or the reading.

```
// Enumerate the possible states of CPU's state machine
localparam
  RESET  =  0, // initial state: reset goes here
  FETCH  =  1, // fetch next instruction from memory
  DECODE =  2, // decode instruction: what are my orders?!
  LOAD   =  3, // execute LOAD:   AC := memory[argument]
  STORE  =  4, // execute STORE:  memory[argument] := AC
  STORE2 =  5, //   store gets an extra clock cycle for write to finish
  JUMP   =  6, // execute JUMP:   PC := argument
  JUMPZ  =  7, // execute JUMPZ:  if (AC==0) PC := argument
  JUMPN  =  8, // execute JUMPN:  if (AC<0) PC := argument
  JUMPNZ = 13, // execute JUMPNZ: if (AC!=0) PC := argument
  ADD    =  9, // execute ADD:    AC := AC + memory[argument]
  SUB    = 10, // execute SUB:    AC := AC - memory[argument]
  MUL    = 11, // execute MUL:    AC := AC * memory[argument]
  OUT    = 12; // execute OUT:    display AC on 7-segment LEDs
```

Since there are 14 states (more than 8 but fewer than 16), we use a 4-bit-wide D-type flip-flop to hold the current value of the CPU state:

```
// Use a 4-bit D-type flip-flop to hold the state of the CPU's FSM
dffe_Nbit #(.N(4))  state_ff (.q(state), .d(state_next),
                              .clock(clock), .enable(run), .reset(reset));
```

and here is the next-state logic to set up the transitions from state to state:

```
// Compute next state based on current state and pertinent conditions
assign state_next =
  reset         ? RESET                  :  // reset line => RESET
  state==FETCH  ? DECODE                 :  // FETCH => DECODE
  state==DECODE ? (IR[15:8]==0 ? LOAD    :  // DECODE => execute decoded
                   IR[15:8]==1 ? STORE   :  //   instruction (LOAD, STORE,
                   IR[15:8]==2 ? JUMP    :  //   JUMP, etc.)
                   IR[15:8]==3 ? JUMPZ   :
                   IR[15:8]==4 ? JUMPN   :
                   IR[15:8]==9 ? JUMPNZ  :
                   IR[15:8]==5 ? ADD     :
                   IR[15:8]==6 ? SUB     :
                   IR[15:8]==7 ? MUL     :
                   IR[15:8]==8 ? OUT     :
                                 FETCH)  :  // unknown => FETCH
  state==STORE ? STORE2                  :  // STORE => STORE2
  /* default */  FETCH                   ;  // default => FETCH
```

As the reading described, the CPU points its "Program Counter" (PC) at the next address in memory, FETCHes the instruction from memory[PC], and then DECODEs it to figure out what to do next. The upper 8 bits of the instruction contain the "opcode," i.e. they determine whether the instruction is a LOAD operation, a STORE operation, an ADD operation, etc. The lower 8 bits of the instruction are the "argument" for the operation, and generally refer to a memory address.

Random Access Memory

This CPU uses a read/write memory, a.k.a. a RAM, both to store instructions that it will execute and to store the intermediate results of its calculations. The module that defines the RAM begins like this:

```
// 256x16 RAM, i.e. 256 storage locations, each of which is 16 bits wide
module ram256x16 (
    input         clock,        // clock (pertinent for writes only)
    input         writeenable,  // write-enable
    input  [7:0]  address,      // address at which to read/write
    input  [15:0] datain,       // data to store next clock (if write-enabled)
    output [15:0] dataout       // current memory contents at address A
);
```

The other details of this `ram256x16` module are not worth studying, except to note that the initial ("power-up") contents of the memory, when the FPGA is first configured by the ADEPT software, are read from a file called `asm.hex`.[2] (This file is read when the FPGA program is compiled to produce the `.bit` file, not when the FPGA is loaded.) Here's the rest of the `ram256x16` module:

```
    reg [15:0] memory [255:0];
    always @ (posedge clock) begin
        if (writeenable) memory[address] <= datain;
    end
    assign dataout = memory[address];
    // power-up memory contents come from text file 'asm.hex'
    initial $readmemh("asm.hex", memory);
 endmodule
```

We connect the 8-bit `memory_addr` lines, the 16-bit `memory_datain` lines, the single `memory_write` line (which is HIGH only when we want to update the value stored in the memory at the current address), and the 16-bit `memory_dataout` lines. The first three of these are outputs of the CPU state machine (they are inputs to the memory), so they are connected like this:

```
    // This CPU uses a RAM consisting of 256 16-bit words.  In the
    // FETCH state, the memory address is the Program Counter, so that
    // we can fetch the next instruction.  Otherwise, the memory
    // address is the "argument" of the decoded instruction, i.e. the
    // low 8 bits of the IR.
    assign memory_addr = (state==FETCH) ? PC : IR[7:0];

    // The memory is only written in the STORE state; the data written
    // to the memory always come from the accumulator (AC).
    assign memory_write  = (state==STORE);
    assign memory_datain = AC;
```

[2]http://positron.hep.upenn.edu/wja/p364/2014/files/asm.hex

The memory_dataout lines are an input to the CPU (they are an output of the memory). The logic for memory_addr is worth thinking about for a moment. If the CPU is FETCHing the next instruction to be run, then the relevant memory address is contained in the Program Counter (PC). Otherwise, the relevant memory address is contained in the low 8 bits of the Instruction Register IR[7:0], because each instruction consists of an opcode (upper 8 bits) and an address (lower 8 bits). The opcode is "what operation to perform," and the address indicates "what data to operate on" (in addition to the accumulator).

Accumulator

The **accumulator** is the register that does nearly all of the CPU's work. It is stored in a 16-bit-wide D-type flip-flop. Here is the Verilog code that instantiates the flip-flop and connects it:

```
     // Accumulator (AC) is this CPU's primary register; all math
     // instructions operate on the accumulator.
     //
     // Acccumulator next-value logic:
     //   ADD    =>  AC := AC + memory
     //   SUB    =>  AC := AC - memory
     //   MUL    =>  AC := AC * memory
     //   LOAD   =>  AC := memory
     //   RESET  =>  AC := 0
     //
     // Note that the multiply happens in a single clock cycle, so it
     // will compile to an entirely combinational multiplier -- the
     // one you would write down using an adder and a multiplexer for
     // each bit of the multiplicand.
      wire [15:0] product =
        (AC*memory_dataout > 'hffff) ? 'hffff : AC*memory_dataout;
    dffe_Nbit #(.N(16)) AC_ff (.q(AC), .d(AC_next), .clock(clock),
                              .enable(AC_enable), .reset(reset));
    assign AC_next  = (state==ADD)  ? AC + memory_dataout :
                      (state==SUB)  ? AC - memory_dataout :
                      (state==MUL)  ? product :
                      (state==LOAD) ? memory_dataout      : 0;
    assign AC_enable =  run &&
            (state==ADD  ||
             state==SUB  ||
             state==MUL  ||
             state==LOAD ||
             state==RESET );
```

It is actually really neat (I think) to see how the contents of the accumulator are transformed when the CPU is in the various states like LOAD, ADD, SUB, MUL. The expression AC + memory_dataout represents a 16-bit-wide adder, whose two inputs are the present contents of the accumulator and the present contents of memory_dataout. The present contents of memory_dataout are whatever happens to be stored at the memory address referred to by memory_addr on the previous page. As described above, memory_addr is normally the low 8 bits of the Instruction Register, i.e. the "operand" or "argument" of whatever instruction is currently being carried out by the CPU. If the current state is ADD, then on the next clock cycle, the contents of the accumulator will be replaced by the output of the adder. The net result of this will be `AC` $\leftarrow$ `AC + memory[IR[7:0]]`, i.e. whatever is stored in the memory address pointed to by the low 8 bits of the Instruction Register will be added to the accumulator. If you don't fully understand this, then now is the time to stop one of us and demand a better explanation!

Similarly, the expression `AC - memory_dataout` represents a 16-bit subtraction. To subtract two numbers, you take the first number and add to it the negative of the second number. To compute the negative of a number, in two's complement representation, you first invert all of the bits (all 1's become 0's and vice-versa) and then add 1 to the result. So the logic that Verilog needs to generate for subtraction is similar to that for addition.

Accumulator's multiplication logic Multiplication of two binary numbers is less complicated than you might think. Suppose you want to compute $9 \times 5 = 45$, which in binary would be $1001_2 \times 101_2 = 101101_2$. You can do this using the binary version of the "long multiplication" technique that you learned (in decimal!) as a kid:

```
      1001
   x  0101
      ----
      1001
     0000
    1001
 + 0000
   -------
   0101101
```

So multiplying together two 16-bit numbers requires 16 multiplexers (the thing to be added in each row is either zero or a shifted copy of the first multiplicand) and 15 adders. In general, the result of multiplying two 16-bit integers together is a 32-bit integer, so in general the answer won't fit back into the accumulator. A real computer needs a mechanism to detect **overflow** conditions in a way that the programmer can handle correctly. To keep things simple, my `product` logic uses this crude workaround: if the product is too big to fit into 16 bits, then I simply report 65535 (the largest 16-bit number) as the answer. That kludge is implemented by multiplexing the output of the multiplier with FFFF_{16}:

```
    wire [15:0] product =
        (AC*memory_dataout > 'hffff) ? 'hffff : AC*memory_dataout;
```

Output register

The **output register** is how this little CPU communicates its results to the outside world. When the `OUT` instruction is executed, the current contents of the accumulator are copied to the `OUT` register, whose contents are always displayed on the 4-digit 7-segment LED display.

On a real computer, a mechanism similar to this would be used to interface the computer to a digital-to-analog converter, or to external devices like printers, network interfaces, disk drives, etc. A real computer would also have an opcode like `IN` to receive input from external devices like keyboards, mice, analog-to-digital converters, disk drives, etc. Anyway, here is the Verilog code for the output register:

```
    // The output register is this CPU's way to report its results to
    // the outside world.  The only path to the 'out' register is from
    // the accumulator.  The 'out' register is only enabled while the
    // OUT instruction is executing.
    dffe_Nbit #(.N(16)) out_ff (.q(out), .d(AC), .clock(clock),
                                .enable(out_enable), .reset(reset));
    assign out_enable = (state==OUT) && run;
```

Instruction Register

The **Instruction Register** (IR) holds the 16-bit instruction that is currently being executed, as described in the textbook chapter. Here is the Verilog code for the IR. Notice that the IR is a flip-flop that is only enabled during the `FETCH` state. Fetching the next instruction from the memory into the IR is analogous to the music machine's fetching the next *duration* and *halfperiod* from the memory into the corresponding registers (a.k.a. flip-flops). We need the IR to hold a copy of the instruction we are currently executing, because the process of carrying out the current instruction will in general require us to read or write other memory locations.

```
    // The Instruction Register (IR) holds the instruction that is
    // currently being executed.  The only path into the IR is from
    // the memory; the IR is only enabled in the FETCH state, i.e.
    // while fetching the next instruction from memory.
    dffe_Nbit #(.N(16)) IR_ff (.q(IR), .d(memory_dataout), .clock(clock),
                               .enable(IR_enable), .reset(reset));
    assign IR_enable = (state==FETCH) && run;
```

Program Counter

The **Program Counter** (PC) holds the memory address from which the next instruction will be read, the next time the CPU enters the `FETCH` state. Normally the Program Counter just increases by one on each subsequent instruction, corresponding to running a program that has no `GOTO`s, no loops, etc. But the `JUMP`, `JUMPZ`, `JUMPN`, and `JUMPNZ` opcodes can overwrite the contents of the Program Counter, thus changing the flow of the program.

If you have ever traced through the sequence of operations carried out by a computer program, either by running your pencil down a program listing line-by-line as your mind mimics the computer's actions, or by using a "symbolic debugger" go watch your program run line-by-line, then Program Counter corresponds to "what line of the program is running now." Since each instruction of an assembly-language program is stored at its own memory address, the Program Counter holds the address in memory corresponding to the line of the program that is currently running. (More precisely, the PC actually holds the address of the *next* line of the program to run, since the PC is incremented at the end of the `FETCH` state.)

The **conditional** jump instructions are the most interesting ones, as they are what permit the computer to make decisions: it can e.g. (JUMPZ instruction) go to a different address if the accumulator currently equals zero, or else continue along its current path if the accumulator contents are non-zero. Similarly, we can check whether the accumulator is negative (i.e. the highest bit is set), and jump or not jump accordingly. **Important point:** Without these conditional jump instructions, you couldn't write programs that use **IF** or **WHILE** or **FOR** to make decisions or to loop until some condition occurs.

Here is the Verilog code for the Program Counter:

```
    // The Program Counter (PC) holds the address from which the next
    // instruction will be fetched.  Here is the program counter
    // update logic:
    //    RESET    =>  PC := 0
    //    FETCH    =>  PC := PC+1  (after fetching from PC, point to PC+1)
    //    JUMP     =>  PC := low byte of IR
    //    JUMPZ    =>  PC := low byte of IR if AC == 0, else unchanged
    //    JUMPN    =>  PC := low byte of IR if AC <  0, else unchanged
    //    JUMPNZ   =>  PC := low byte of IR if AC != 0, else unchanged
    dffe_Nbit #(.N(8)) PC_ff (.q(PC), .d(PC_next), .clock(clock),
                              .enable(PC_enable && run), .reset(reset));
    assign PC_next  = (state==RESET) ? 0 :
                      (state==FETCH) ? PC+1 : IR[7:0] ;
    assign PC_enable = run &&
                       ((state==RESET)           ||
                        (state==FETCH)           ||
                        (state==JUMP)            ||
                        (state==JUMPZ && AC==0)  ||
                        (state==JUMPNZ && AC!=0) ||
                        (state==JUMPN && AC[15]));
```

Top-level connections

That's basically all it takes to make a simplified computer. You can see that it's just a state machine connected to a memory, not so different from the music machine we studied earlier today. To make it easy for you to see all of the computer's inputs and outputs, I put the guts of the computer (a.k.a. CPU, Central Processing Unit) into a module called `simple_cpu`. Here are the connections needed from the top-level module:

```
    // These wires connect to the inputs/outputs of the CPU module
    wire [15:0] memory_dataout, memory_datain, IR, AC, out;
    wire [7:0]  PC, memory_addr;
    wire [3:0]  state;
    wire        memory_write;

    // Determine when the CPU will run (do its normal thing) and when
    // it will pause to wait for the user.
    wire run = !sw[0];

    // Button 1 will reset the CPU to its initial state.
    // function
    wire reset = btn[1];

    // Instantiate the CPU and its memory
    simple_cpu cpu (.clock(clock), .reset(reset), .run(run),
                    .PC(PC), .AC(AC), .IR(IR), .state(state),
                    .memory_dataout(memory_dataout), .memory_addr(memory_addr),
                    .memory_datain(memory_datain), .memory_write(memory_write),
                    .out(out));
    ram256x16 ram (.clock(clock),
                   .writeenable(memory_write),
                   .address(memory_addr),
                   .datain(memory_datain),
                   .dataout(memory_dataout));

    // The green LEDs will display the Program Counter
    assign led = PC;

    // The 7-segment display will show the OUT register, i.e. the most
    // recent prime number.
    wire [15:0] leddat = out;
....
    assign {digit3,digit2,digit1,digit0} = leddat;
```

If you look at the complete Verilog program at this URL:
http://positron.hep.upenn.edu/wja/p364/2014/files/lab27_part3.v
you will see that there is really not that much to it.

Now what can this computer do?
To prove to you that this computer is capable of doing a real computation, I coded the stupidest imaginable algorithm for calculating all of the prime numbers from 2 to 9973. Here is how the algorithm would look if I were to write it in the C programming language.

```
#include <stdio.h>

int main(void)
{
  int i, j, k, product;
  // loop 'i' over candidate prime numbers, from 2 to 9999
  for (i = 2; i<10000; i = i+1) {
    // loop 'j' over possible first factors, from 2 to i-1
    for (j = 2; j<i; j = j+1) {
      // loop 'k' over possible second factors, from j to i-1
      for (k = j; k<i; k = k+1) {
        product = j*k;
        // if j*k equals i, then i must not be prime: jump to 'iloop'
        if (product==i) goto iloop;
        // if j*k > i, then skip the rest of the k loop
        if (product>i) break;
      }
    }
    // if we reach this point, then i is prime: print it out
    printf("%d\n", i);
  iloop:;  // this label 'iloop' allows the 'goto' to jump here
  }
  return 0;
}
```

On my Mac, the program outputs 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, . . ., 9887, 9901, 9907, 9923, 9929, 9931, 9941, 9949, 9967, 9973. Since we have no C compiler for our home-made computer, we have to write our program directly in the computer's **assembly language**, i.e. using the opcodes LOAD, STORE, ADD, JUMP, etc.

```

        #
        # prime.sasm
        # coded 2010-11-11 by Bill Ashmanskas, ashmansk@hep.upenn.edu
        #
        # The purpose of this program is to demonstrate that the CPU
        # implemented in simple_cpu.v is capable of carrying out a
        # non-trivial computation.
        #
        # This is probably the dumbest imaginable algorithm to compute
        # prime numbers.  Its execution time scales as the third power
        # of the number of candidates to evaluate.  For every candidate i,
        # loop over possible factors j and k, testing whether i==j*k.  If
        # no such j and k are found, then display i on the LEDs.
        #
        # Note that with storage for a mere 5000 boolean values (which
        # I could have easily cooked up), one can use a much more efficient
        # algorithm, the Sieve of Eratosthenes.  It scales (with number of
        # candidate integers N) as N*log(N)*log(log(N)), while my algorithm
        # scales as N**3.  I point this out only so that you don't think
        # that I think the N**3 algorithm is a good way to compute primes.
        #
start:  load    istart  #
        store   i       #  i := istart (nominally 1)
iloop:  load    i       #  loop i from istart+1 to 9999
        add     one     #
        store   i       #    i := i+1
        load    d9999   #
        sub     i       #
        jumpn   done    #    if (i>9999) goto done
        load    one     #
        store   j       #    j := 1
jloop:  load    j       #    loop j from 2 to i-1
        add     one     #
        store   j       #      j := j+1
        load    i       #
        sub     j       #
        jumpz   jdone   #      if (j==i) goto jdone
        load    j       #
        sub     one     #
        store   k       #      k := j-1
kloop:  load    k       #      loop k from j to i-1
        add     one     #
        store   k       #        k := k+1
        sub     i       #
        jumpz   jloop   #        if (k==i) goto jloop
        load    j       #
        mul     k       #
        store   prod    #        product := j*k
        sub     i       #        // if j*k==i then i is not prime
        jumpz   iloop   #        if (product==i) goto iloop  // skip to next i
        jumpn   kloop   #        if (product<i)  goto kloop  // keep looping k
                        #        // k exceeds i/j, so skip to next j
        jump    jloop   #        goto jloop
        #
        # If we reach here, then i is a prime number.  Display it.
        #
jdone:
... (uninteresting stuff suppressed) ...
        jump    iloop   #  go back up to try next candidate i
done:   jump    start   #  go back and start counting again from i==2
#
# This is where we define all of the constants and variables that
# our program will use when it runs.
#
zero:   .data   0       #  store the constant '0'
one:    .data   1       #  store the constant '1'
i:      .data   0       #  store the loop variable 'i' (prime number cand.)
j:      .data   0       #  store the loop variable 'j'
k:      .data   0       #  store the loop variable 'k'
prod:   .data   0       #  store the product 'prod' = j*k
outnum: .data   0       #  compute/store binary-coded-decimal conversion of i
remain: .data   0       #  store remainder used in BCD computation
hdigit: .data   0       #  store hex value used to display one decimal digit
h1000:  .data   1000    #  store hexadecimal constant 0x1000
h100:   .data   100     #  store hexadecimal constant 0x100
h10:    .data   10      #  store hexadecimal constant 0x10
d10000: .data   2710    #  store decimal constant 10000
d1000:  .data   3e8     #  store decimal constant 1000
d100:   .data   64      #  store decimal constant 100
d10:    .data   a       #  store decimal constant 10
d9999:  .data   270f    #  store decimal constant 9999 (= 270f in hexadecimal)
istart: .data   1       #  starting value for i (i.e. first prime to check)
Jdelay: .data   1000    #  delay factor (in hexadecimal)
Kdelay: .data   300     #  additional delay factor (in hexadecimal)
```

The part of the code that is shown above does the prime number calculation. The whole program, including the parts that I omitted above, is at
http://positron.hep.upenn.edu/wja/p364/2014/files/prime.sasm
There are two parts that I didn't show:

First, the conversion of the prime number from a 16-bit hexadecimal integer into four decimal digits (thousands, hundreds, tens, ones), so that the primeness of the prime numbers looks more convincing to a human observer.

Second, the brief delay before displaying each new prime number, so that the numbers do not overwrite each other too quickly for you to see. The delay is implemented as a "nested loop:" the outer loop repeats $1000_{16} = 4096$ times, and the inner loop repeats $300_{16} = 768$ times for each repetition of the outer loop. So it's just wasting time by counting up to about 3 million. The nested loop is needed because our tiny computer's integers are only 16 bits wide.

Machine-readable format

The program above is still in human-readable form. We need to convert the instructions into hexadecimal memory contents. Here is the output of that process, i.e. an annotated file that is identical in content to `asm.hex`.[3] The annotated version is at `prime_assembled.txt`.[4] The file looks like this (with boring parts suppressed):

```
        mem['h00] = 'h006e;  //  start:   load istart
        mem['h01] = 'h015f;  //           store i
        mem['h02] = 'h005f;  //  iloop:   load i
        mem['h03] = 'h055e;  //           add one
        mem['h04] = 'h015f;  //           store i
        mem['h05] = 'h006d;  //           load d9999
        mem['h06] = 'h065f;  //           sub i
        mem['h07] = 'h045c;  //           jumpn done
        mem['h08] = 'h005e;  //           load one
        mem['h09] = 'h0160;  //           store j
        mem['h0a] = 'h0060;  //  jloop:   load j
        mem['h0b] = 'h055e;  //           add one
        mem['h0c] = 'h0160;  //           store j
        mem['h0d] = 'h005f;  //           load i
        mem['h0e] = 'h0660;  //           sub j
        mem['h0f] = 'h031f;  //           jumpz jdone
        mem['h10] = 'h0060;  //           load j
        mem['h11] = 'h065e;  //           sub one
        mem['h12] = 'h0161;  //           store k
        mem['h13] = 'h0061;  //  kloop:   load k
        mem['h14] = 'h055e;  //           add one
        mem['h15] = 'h0161;  //           store k
        mem['h16] = 'h065f;  //           sub i
        mem['h17] = 'h030a;  //           jumpz jloop
        mem['h18] = 'h0060;  //           load j
        mem['h19] = 'h0761;  //           mul k
        mem['h1a] = 'h0162;  //           store prod
        mem['h1b] = 'h065f;  //           sub i
        mem['h1c] = 'h0302;  //           jumpz iloop
        mem['h1d] = 'h0413;  //           jumpn kloop
        mem['h1e] = 'h020a;  //           jump jloop
        mem['h1f] = 'h005d;  //  jdone:   load zero
...
        mem['h5c] = 'h0200;  //  done:    jump start
        mem['h5d] = 'h0000;  //  zero:    .data 0
        mem['h5e] = 'h0001;  //  one:     .data 1
```

[3] http://positron.hep.upenn.edu/wja/p364/2014/files/asm.hex
[4] http://positron.hep.upenn.edu/wja/p364/2014/files/prime_assembled.txt

```
        mem['h5f] = 'h0000;  //  i:       .data 0
        mem['h60] = 'h0000;  //  j:       .data 0
        mem['h61] = 'h0000;  //  k:       .data 0
        mem['h62] = 'h0000;  //  prod:    .data 0
        mem['h63] = 'h0000;  //  outnum:  .data 0
        mem['h64] = 'h0000;  //  remain:  .data 0
        mem['h65] = 'h0000;  //  hdigit:  .data 0
        mem['h66] = 'h1000;  //  h1000:   .data 1000
        mem['h67] = 'h0100;  //  h100:    .data 100
        mem['h68] = 'h0010;  //  h10:     .data 10
        mem['h69] = 'h2710;  //  d10000:  .data 2710
        mem['h6a] = 'h03e8;  //  d1000:   .data 3e8
        mem['h6b] = 'h0064;  //  d100:    .data 64
        mem['h6c] = 'h000a;  //  d10:     .data a
        mem['h6d] = 'h270f;  //  d9999:   .data 270f
        mem['h6e] = 'h0001;  //  istart:  .data 1
        mem['h6f] = 'h1000;  //  Jdelay:  .data 1000
        mem['h70] = 'h0300;  //  Kdelay:  .data 300
```

So the memory contents that cause the computer to calculate this big sequence of prime numbers look like this: 006e 015f 005f 055e 015f 006d 065f 045c 005e 0160 0060 055e 0160 005f 0660 031f 0060 065e 0161 0061 055e 0161 065f 030a 0060 0761 0162 065f 0302 0413 020a 005d 0163 005f 0164 0669 0426 025c 005d 0165 0064 066a 0430 0164 0065 0566 0165 0228 0065 0563 0163 005d 0165 0064 066b 043d 0164 0065 0567 0165 0235 0065 0563 0163 005d 0165 0064 066c 044a 0164 0065 0568 0165 0242 0065 0563 0564 0163 0800 006f 0160 0070 0161 0061 065e 0161 0953 0060 065e 0160 0951 0202 0200 0000 0001 0000 0000 0000 0000 0000 0000 0000 1000 0100 0010 2710 03e8 0064 000a 270f 0001 1000 0300 0000 0000 . . . (more zeros). There's not much to it! To convert the first ("assembly") format into the second ("machine") format, I wrote this Python program:
`http://positron.hep.upenn.edu/wja/p364/2014/files/sasm.py.txt`
which is called an **assembler**, because it converts human-readable **assembly language** into hexadecimal (or binary) **machine code**.

You can run the assembler yourself from your web browser, at this link:
`http://positron.hep.upenn.edu/wja/p364/2014/files/assembler.html`

Again, for Part 3, all you really need to do is to load this pre-compiled `.bit` file:
`http://positron.hep.upenn.edu/wja/p364/2014/files/lab27_part3.bit`
into your BASYS2 board with ADEPT and to see your board display a sequence of prime numbers. (Make sure the sliding switches are all in the DOWN positions.)

$\boxed{\textbf{Part 4}}$

For this final part, the idea is for you to watch the tiny computer step through some of its operations in detail. You can also write your own programs in the tiny computer's assembly language, if you like. To make the operation of the computer less opaque, I've coded in some features for watching and changing what the computer is doing. To get all of these debug features, you'll want to load
`http://positron.hep.upenn.edu/wja/p364/2014/files/lab27_part4.bit`
into your BASYS2 board. The Verilog source code is at
`http://positron.hep.upenn.edu/wja/p364/2014/files/lab27_part4.v`
And if you want to make changes, you will probably find it easiest to start from this complete Xilinx ISE project in `.zip` form:
`http://positron.hep.upenn.edu/wja/p364/2014/files/lab27_part4.zip`

Reset button. If you push down `btn[1]`, you will reset the CPU to its initial state, and it will restart whatever program it was running. (If it's the prime-number program, you'll see it restart counting out prime numbers from 0002.)

Pause switch. If you slide `sw[0]` up, you will pause the CPU's activity.

Program Counter display. The 8 green LEDs display the current value of the Program Counter.

New functions for 7-segment LEDs:

- If you slide `sw[7]` up, then the left two digits will display the Program Counter, and the right two digits will display the `memory_addr`, i.e. the address that is currently sent into the RAM.
- If you slide `sw[6]` up (but not `sw[7]`), then the four digits will display the Instruction Register.
- If you slide `sw[5]` up (but not `sw[7:6]`), then the four digits will display `memory_dataout`, i.e. the data that are currently read out of the RAM.
- If you slide `sw[4]` up (but not `sw[7:5]`), then the four digits will display the Accumulator contents.
- If you slide `sw[3]` up (but not `sw[7:4]`), then the right-hand digit will display the state number of the CPU's finite-state machine, i.e. 0=`RESET`, 1=`FETCH`, 2=`DECODE`, etc.
- If you press `btn[3]`, it will rotate between **four modes** of operation for the CPU.
  - The decimal-point dots will display `---*`, `--*-`, `-*--`, `*---` respectively for modes 0,1,2,3.
  - **In mode 0**, the CPU runs freely as it did in Part 3. You can use `sw[0]` to pause the CPU and `btn[1]` to reset it.

- **In mode 1**, the CPU will pause at each `OUT` instruction until you press `btn[0]` to continue. This results in each prime number being displayed until you push `btn[0]`. While the CPU is stopped, you can use `sw[7]` through `sw[3]` to explore the contents of 7=(PC,`memaddr`), 6=IR, 5=`memdataout`, 4=AC, 3=`state`.
- **In mode 2**, the CPU will pause at each `DECODE` state until you press `btn[0]`. You can single-step through individual instructions and examine the CPU's registers using `sw[7]` through `sw[3]` as above.
- Also in mode 2, if `sw[0]` is up, then the CPU will pause on every clock cycle, no matter what state it is in. You can watch the CPU's finite state machine go from `FETCH` to `DECODE` to (`LOAD`, `STORE`, `ADD`, etc.).
- In mode 0, 1, or 2, pressing `btn[2]` will copy the current 8-bit value represented by the sliding switches into a register called `useraddr`. When you are in mode 1 (not mode 2, this is mode 1 again), if the Program Counter equals the 8-bit value in `useraddr`, then the CPU will pause until you press `btn[0]`. This allows you to explore what the CPU is doing at a particular section of the program code. (If you are a programmer, it is like setting a breakpoint in the debugger.)
- So you can use the **useraddr** feature in mode 1 to make the program stop next time it reaches e.g. the instruction at address 18 (hexadecimal), then switch to mode 2 to single-step through the subsequent instructions.
- **In mode 3**, the CPU does nothing, and you can read/write the memory.
  - If all of the sliding switches are down, then `btn[0]` will step forward through the memory one address at a time.
  - In **memory mode (mode 3)**, the 8 green LEDs show the address, and the digits show the memory data stored at that address.
  - While you are are holding down `btn[0]`, the digits will momentarily show the address also.
  - If the switches are all in the down (zero) position, then pressing `btn[0]` will add 1 to the address. This makes it easy for you to step sequentially through the entire memory to check its contents.
  - If the switches are set to a non-zero position, then pressing `btn[0]` will use the switches as the address into the memory. This address is remembered once the button is released, so you can then move the switches without changing the address.
  - If you push `btn[1]`, the 8-bit value from the switches will replace the lower 8 bits of the memory contents at the current memory address.
  - If you push `btn[2]`, the 8-bit value from the switches will replace the upper 8 bits of the memory contents at the current memory address.
  - The "current memory address" in memory mode (mode 3) is stored in the same `userdata` register mentioned above.
  - This allows you to look through the memory contents, compare them with the assembler output, and even modify the program (or more likely the .data words).

If you want to write your own program, you will probably find it convenient to run it in

mode 1 so that you can display a sequence of values on the 7-segment LED displays using the `OUT` instruction, then push `btn[0]` to let the program run until the next `OUT` instruction.

A few annoying sources of potential confusion:

- I recently modified `lab27_part3.v` so that the CPU is clocked at 50 MHz, but I did not have time to make the same modification to `lab27_part4.v` (because making that modification would break other features in `lab27_part4.v`). So the `Kdelay` constant at the bottom of `prime.sasm` needs to be much smaller (e.g. `Kdelay: .data 10`) for `lab27_part4.v` than in `lab27_part3.v` (where the value is 300). So if you use the `asm.hex` from Part 3 and compile it into the Part 4 Verilog program, the delay between prime numbers will be nearly one minute, unless you reduce this `Kdelay` value.
- The file `asm.hex` must contain exactly 256 hexadecimal values, one value per line. The Xilinx compiler is not very smart (or very forgiving) about the `.hex` file format. If your file is the wrong length, the compiler will not even warn you; it will just initialize the memory contents with all zeros.
- If you change the `asm.hex` file and want to generate a new `lab27_part4.bit` file to reflect this change, you need to right-click "Generate Programming File" and then choose "ReRun All." The Xilinx compiler isn't smart enough to know what steps need to be re-run when the `asm.hex` file is changed, as it does not consider it to be a source-code file.
- For these last two reasons, I think that in the future I will avoid using the `.hex` memory file. I only recently learned just how limited the Xilinx `$readmemh` support is. Sorry!

Your Challenge (if you have the time and energy):
Here is a very short program that just counts up, starting from 1, and `OUT`puts each number to the 7-segment display. About half of the program is there just to delay about a second after displaying each number. You can leave those lines out if you prefer to use the "mode 1" feature described above.

```
start:  load   zero     #  we'll do a loop over i
        store  i        #  i:= 0
iloop:  load   i
        add    one
        store  i        #  i := i+1
        out             #  send accumulator to the 7-segment display

# these next six lines are just here to pause after showing each number
        load    Kdelay  #  loop k from Kdelay downto 1
        store   k       #  k := Kdelay
kdelay: load    k       #  loop k from Kdelay downto 1
        sub     one     #
        store   k       #    k := k-1
        jumpnz  kdelay  #    if (k!=0) keep looping over k

        jump    iloop   #  go back up to loop over the next value of i
```

```

zero:   .data   0       #  store the constant '0'
one:    .data   1       #  store the constant '1'
i:      .data   0       #  store the loop variable 'i'
j:      .data   0       #  store the loop variable 'j'
k:      .data   0       #  store the loop variable 'k'
Kdelay: .data   f000    #  delay factor (in hexadecimal)
```

Copy this source code (there is an easier-to-copy version at
`http://positron.hep.upenn.edu/wja/p364/2014/files/lab27_part4_asm.txt`
) into the **Assembler Code** window at
`http://positron.hep.upenn.edu/wja/p364/2014/files/assembler.html`
and then click **Assemble it**. Then copy and paste the 256 lines of "`.hex` format" values to replace the current contents of `asm.hex`, which you can edit using the Xilinx ISE source code editor. Then do "Generate Programming File" → "ReRun All" (you **must** select "**ReRun All**" to make Xilinx ISE use the updated `asm.hex` contents) and load the resulting `.bit` file into your board. You should see the board counting (in hexadecimal).

Now modify the assembler source code in some way that interests you. You could make the counter count up in steps of 3 instead of steps of 1. Or you could display the square of each integer (in hexadecimal) instead of the integer itself. Or you could take one more stab at displaying the Fibonacci sequence! The idea is just to give yourself a chance to write a very short assembly-language program and to load it into the tiny computer that we programmed into the BASYS2 board. Ask us for help if you have an idea that you're not sure how to implement.

If you plan to spend more than a few minutes editing the assembler source code, then you should take precautions to **avoid losing your work**. Use some sort of text editor (e.g. the ISE source code editor, or even NotePad) to make your changes, and then copy/paste them into the web browswer window. (The web browser window is not smart enough to save your file anywhere. Your text is only there until you close or reload the web page.)

If you make only a tiny change to the `asm.hex` contents (e.g. only affecting one or two memory locations), then you might find it quicker to use the "mode 3" feature described above, to avoid the time (about 1 minute) needed to recompile your Xilinx project.