Physics 364, Fall 2014, reading due 2014-12-07.

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 states in the music-machine (or player-piano) state machine described in the notes, and (very briefly) what is the purpose of each state?

2. In the state diagram for the simplified computer described in the notes, what is the role of the `FETCH`, `DECODE`, `LOAD`, `STORE`, `ADD`, and `JUMP` states?

3. In the simplified computer, why does the existence of the `JUMPZ` and `JUMPN` states (and the corresponding opcodes) make the possible behavior of this machine so much more interesting than if there were only `JUMP` (but not `JUMPZ` or `JUMPN`)?

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. If you're willing (an answer is not required), please share your thoughts on how the course content could be improved for future years. Would you change the balance between analog and digital? Are there topics on which we should spend less time, more time, or topics to add or delete? Should the workload or time spent per week be changed? Note that this question is only about course topics and structure. (How we did as teachers is something you can review anonyously in your course evaluation.)

Administrative note: By popular vote, what we originally called the "take-home final exam" will now instead be a "final homework assignment." The main difference is that it is OK, for this final homework assignment, if you want to discuss the questions with other students. The work that you turn in must be your own, but if you want to talk with people about how you approached a problem, that is acceptable. My goal is that this final assignment will help you to review the course material so that the key ideas will stick with you for the future. The assignment will be available online by December 10 and will be due by the end of the term on December 19.

Last week, we introduced state machines and memories. In Lab 27, the last day of class, we will explore two examples of state machines that are much more complicated than the traffic signal and vending machine that we studied in Lab 26. But the key ideas will be the same as in the simpler state machines.

The first state machine we will study in Lab 27 is a kind of player piano. It reads from a memory a sequence of notes and durations, and plays these notes on a speaker, in order to play a tune. 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.

To store the desired information, we will use a ROM (Read Only Memory). The size and shape of this ROM will be the same as we studied in last week's reading: 256×16, 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. Here, I will use a Verilog description that is a bit verbose, but has the advantage that it maps directly onto last week's discussion of a ROM as a special case of a multiplexer. Here it is:

```
// 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
```

The big `assign` statement is using the "ternary conditional operator" `?:` to say,

```
if (address == 0) then
    dataout will equal 1000 (decimal)
else if (address == 1) then
    dataout will equal 1517 (decimal)
else if (address == 2) then
    dataout will equal 1000
else if (address == 3) then
    dataout will equal 1703
...
else if (address == 255) then
    dataout will equal 0
else (in case we missed any possibilities)
    dataout will equal 0
end if
```

After a moment's thought, you can see that this is describing a multiplexer that selects between 256 possible values. The repeated use of `?:` is confusing at first. We saw last week how to write

```
    assign myvalue = (condition ? first : second);
```

to multiplex between two possible values. We can extend this to multiplex between four possible values, like this:

```
  assign myvalue =
      (A==0 ? (first) :
              (A==1 ? (second) :
                      (A==2 ? third : fourth)));
```

If $A = 0$ then `myvalue` gets the `first` possibility. If $A \neq 0$ then we need to consider whether or not $A = 1$. If $A = 1$ then `myvalue` gets the `second` possibility. If $A \neq 1$ then we need to consider whether or not $A = 2$. If $A = 2$ then `myvalue` gets the third possibility; otherwise, `myvalue` gets the fourth possibility.

It turns out that we can remove all of the above parentheses and get the exact same result, more neatly formatted:

```
  assign myvalue = A==0 ? first  :
                   A==1 ? second :
                   A==2 ? third  : fourth ;
```

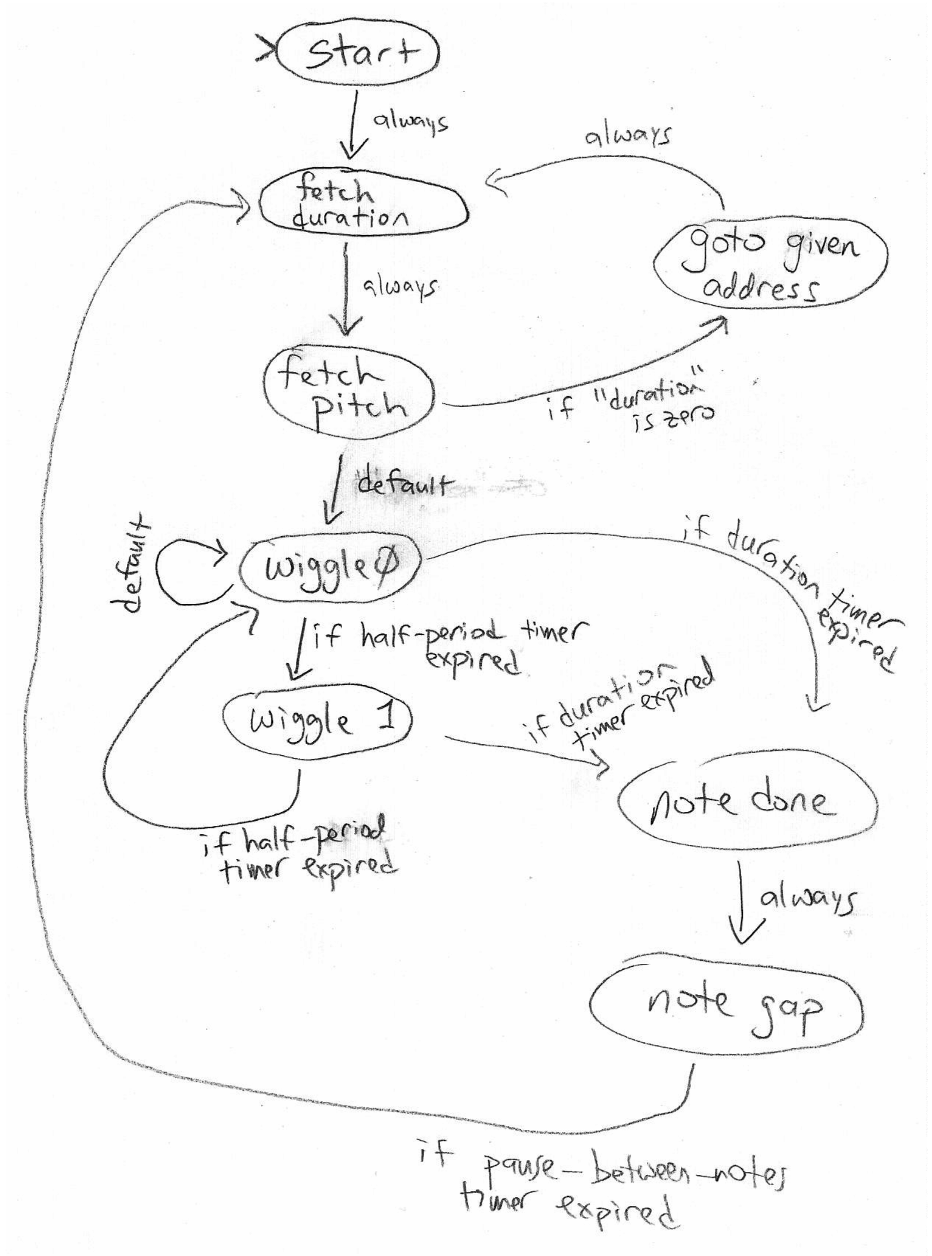
This trick works in C, in Java, and in Verilog. We will use it many times in Lab 27.

Notice that I stored `duration` in even-numbered addresses and `halfperiod` in odd-numbered addresses. So we will need to read from two successive addresses to play a given note. Notice also (e.g. at $A = 27$) that I used the special case `halfperiod==0` to represent a musical "rest," i.e. to represent silence.

I use one more special case, which is trickier to understand. In the special case `duration==0`, I use the corresponding `halfperiod` value to inducate the address from which the machine should subsequently start reading. This allows me to do a "GOTO" operation, so that, for example, at the end of a tune, I can go back and play the same tune again. It is worth taking a few minutes to understand the point of this "GOTO" operation, because it is a key idea that will help you to understand how a computer does its work: the idea is that I can store into the ROM a value which is then used as a subsequent address into that same ROM. Whoa!

The diagram Shown below is my state diagram for the machine that will read this ROM to play out a tune. I 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
fetch pitch
always
goto given address
if "duration" is zero
default
wiggle0
default
if duration timer expired
if half-period timer expired
wiggle 1
if duration timer expired
if half-period timer expired
note done
always
note gap
if pause-between-notes timer expired

Since there are 8 states, I use a 3-bit-wide D-type flip-flop to hold the current state of the machine, which is called `state`. The logic to decide which state to enter on the next clock cycle will manipulate the wires called `nextstate`. Notice that I used trick introduced in last week's reading to write a general-purpose N-bit flip-flop module. When I instantiate it, I use `(.N(3))` to indicate that the number of bits (N) should be 3 in this instance. Another thing I am doing differently is connecting the top-level wires to the module input/output wires **by name** instead of by position. Then I can write the inputs/outputs in whatever order I wish, and Verilog will not get confused about which top-level wires connect to which module inputs and outputs. It also makes it more clear to the person reading the program which argument is doing what.

Here is the instantiation of the 3-bit-wide D-type flip-flop that holds the present state of the music 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
```

Now, the really complicated part of this machine is the next-state logic. The `nextstate` wires determine which state we will 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 user (that’s 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.

Whew! That takes some effort to digest.

Next, the ROM connections. The memory address from which to read is stored in an 8-bit D-type flip-flop. Notice that 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: it needs to know its address (input) and where to send its 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 (described below) contains the next address from which we should continue reading.

I said in last week's reading that the "state register" may be one of several registers (flip-flops) used in a state machine, but I didn't elaborate. Here is an example.

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 really important 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. We will also have two 16-bit counters. One of them increments once per microsecond. (This turns out to be easy to do, because we will 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.

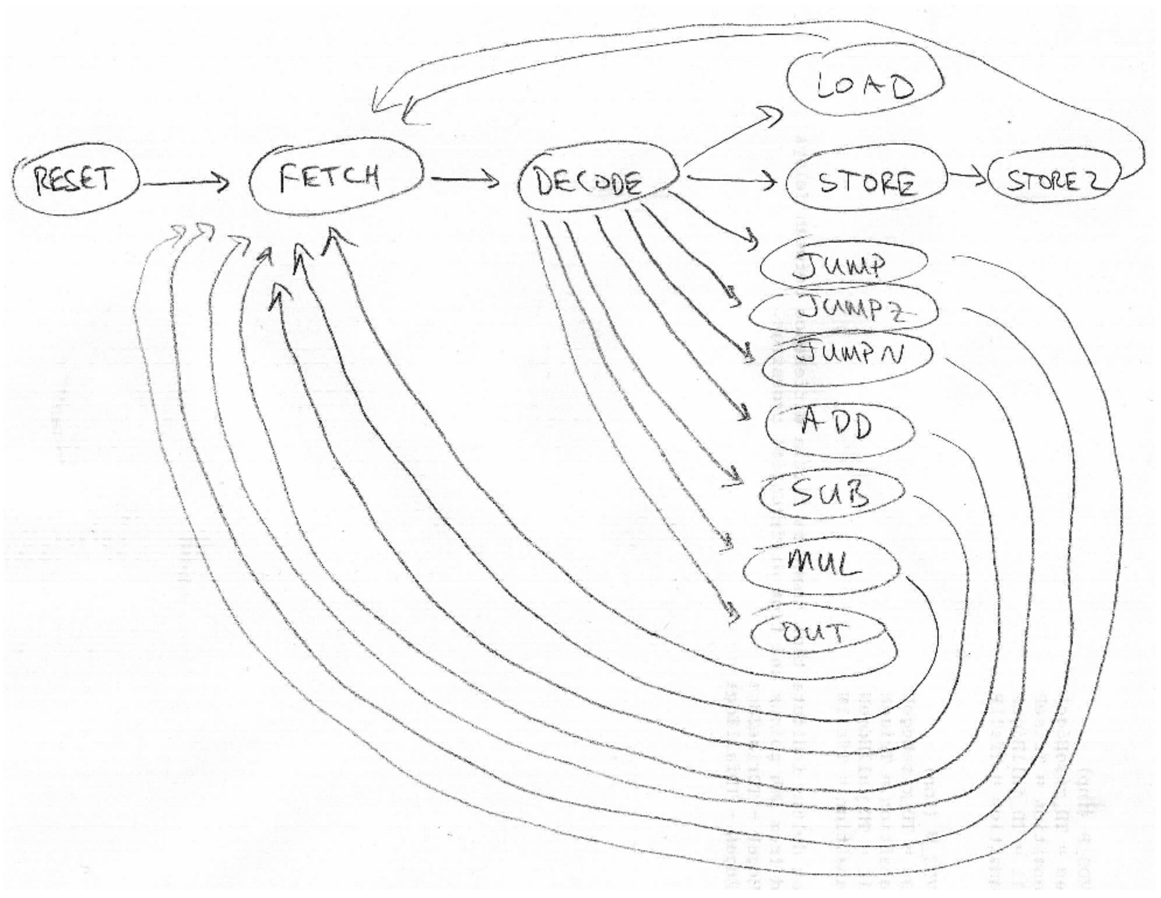
```
    // Use a 16-bit counter to count off microseconds until the next
    // time the wire driving the speaker needs to wiggle up or down
    wire [15:0] count_usec;
    wire        reset_usec;
    counter_Nbit #(.N(16)) myuseccounter
      (.q(count_usec),
       .clock(clock),        // clocked 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
```

I will be finishing up the detailed writeup for Lab 27 over the weekend, so you can't yet see it in its final form. But I suggest that you quickly look through the Fall 2012 lab that covers this player-piano state machine, at this link:
http://positron.hep.upenn.edu/wja/p364/2012/lab11f.html

The music machine is the first of two rather complicated state machines we will study in Lab 27. The second one is even a bit more complicated than the music machine, but it expresses the main point that we have been building up to in these past few weeks — that you understand enough digital logic to be able to see how to build a computer out of building blocks that you understand down to the transistor level.

The main goal of this second part of Lab 27 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.

Here (shown below) is the state diagram that I quickly flashed at the end of last week's reading (with minor modifications). This state diagram actually implements a tiny computer!



Before we look in detail at the implementation of this state machine, please read (the text and figures, not so much the code) the short textbook chapter linked below, which describes the design of the tiny computer on which the above state diagram is based. I will show you in detail my own implementation of the same computer (with minor modifications). Unlike the textbook's version, my implementation uses only the subset of Verilog that we have learned in this course. (So the many `always` blocks that appear in the textbook code won't be needed in our version.)

http://positron.hep.upenn.edu/wja/p364/2014/files/reading15_supplement.pdf

The state diagram above is basically the same one as in the textbook chapter, except that I created several additional instructions:

- `JUMPZ` : jump ("goto") if `AC==0`
- `JUMPN` : jump if `AC<0`
- `SUB` : subtract memory contents from accumulator (`AC`)
- `MUL` : multiply accumulator contents by memory contents
- `OUT` : display accumulator on 7-segment LED display

Here is what this list of states (i.e. the association from word-like names to integer state numbers) looks like in Verilog:

```verilog
    // 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
      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 13 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:

```verilog
    // 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:

```verilog
    // 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]==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 textbook chapter described, the CPU points its "Program Counter" (`PC`) at the next address in memory, `FETCH`es the instruction from `memory[PC]`, and then `DECODE`s 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.

This CPU uses a read/write memory, a.k.a. a Random Access Memory (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`.[1] In case you're curious, 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'll see in a moment how to make sense of the data stored in the RAM. To connect this memory to our CPU state machine, we need to 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;
```

The `memory_dataout` lines are an input to the CPU (they are an output of the memory); we will see below how they are used.

[1] http://positron.hep.upenn.edu/wja/p364/2012/asm.hex

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:

```verilog
     // 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.
    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)  ? AC * memory_dataout :
                      (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 **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:

```verilog
    // 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;
```

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:

```
    // 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;
```

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`, and `JUMPN` opcodes can overwrite the contents of the Program Counter, thus changing the flow of the program.

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.

It turns out that if you want to figure out whether $B > A$, you subtract B from A and then see whether or not the result is negative. That is the usual way that computers make comparisons of two numbers. Isn't that clever of them? You can also figure out if $B = A$ by subtracting B from A and then testing to see whether the result is zero.

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
    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==JUMPN && AC[15]));
```

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 at the beginning of today's reading. 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;
```

Everything else in the top-level module (called `lab11g` in the Fall 2012 materials, but will be called `lab27_part2` this year) is stuff that has been there since earlier labs: the counter to make a 1 MHz clock from the on-board 50 MHz clock; the `display4digits` module and its connections, to handle the ping-pong updating of the 4 separate digits of the 7-segment display.

If you look at the complete Verilog program from Fall 2012 at this URL:
http://positron.hep.upenn.edu/wja/p364/2012/lab11g_part2.v
you will see that there is really not that much to it.

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. Stare at it for long enough to convince yourself that it can indeed identify prime numbers. (If you don't know C, then just try to follow the comments instead of the code.)

```c
#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;
}
```

When I run the above program on my Mac, I see this output (boring parts suppressed):

```
2
3
5
7
11
13
17
19
23
29
```

```
....
9887
9901
9907
9923
9929
9931
9941
9949
9967
9973
```

Regrettably, we do not have a 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. Here we go! (Note that everything to the right of a '#' symbol is a comment.)

```
        #
        # 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.  The only purpose
        # of the code below is to convert i from binary into decimal so
        # that the LED display can be understood by human observers.
        #
jdone:  load    zero    #  'outnum' will hold the binary-coded decimal result
        store   outnum  #  outnum := 0

... (uninteresting stuff suppressed) ...

done:   jump    start   #  go back and start counting again from i==2
disply: load    outnum  #  send most recent prime to the 7-segment display
    out     #
        jump    iloop   #  go back up to try next candidate i
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'
l:      .data   0       #  store the loop variable 'l'
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)
Ndelay: .data   f000    #  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/p364/2012/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). If I didn't do this, then the prime numbers would print out in hexadecimal, which would make the program seem less 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 simply as a loop: tell the computer to count to about 60000 as a way to waste time.

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`.[2] The annotated version is at `prime_assembled.txt`.[3] The file looks like this (with boring parts suppressed):

```
        mem['h00] = 'h006b;  //  start:   load  istart
        mem['h01] = 'h015b;  //           store i
        mem['h02] = 'h005b;  //  iloop:   load  i
        mem['h03] = 'h055a;  //           add   one
        mem['h04] = 'h015b;  //           store i
        mem['h05] = 'h006a;  //           load  d9999
        mem['h06] = 'h065b;  //           sub   i
        mem['h07] = 'h0455;  //           jumpn done
        mem['h08] = 'h005a;  //           load  one
        mem['h09] = 'h015c;  //           store j
        mem['h0a] = 'h005c;  //  jloop:   load  j
        mem['h0b] = 'h055a;  //           add   one
        mem['h0c] = 'h015c;  //           store j
        mem['h0d] = 'h005b;  //           load  i
        mem['h0e] = 'h065c;  //           sub   j
        mem['h0f] = 'h031f;  //           jumpz jdone
        mem['h10] = 'h005c;  //           load  j
        mem['h11] = 'h065a;  //           sub   one
        mem['h12] = 'h015d;  //           store k
        mem['h13] = 'h005d;  //  kloop:   load  k
        mem['h14] = 'h055a;  //           add   one
        mem['h15] = 'h015d;  //           store k
        mem['h16] = 'h065b;  //           sub   i
        mem['h17] = 'h030a;  //           jumpz jloop
        mem['h18] = 'h005c;  //           load  j
        mem['h19] = 'h075d;  //           mul   k
        mem['h1a] = 'h015f;  //           store prod
        mem['h1b] = 'h065b;  //           sub   i
        mem['h1c] = 'h0302;  //           jumpz iloop
```

[2] http://positron.hep.upenn.edu/wja/P364_2012/asm.hex
[3] http://positron.hep.upenn.edu/wja/P364_2012/prime_assembled.txt

```
        mem['h1d] = 'h0413;  //           jumpn kloop
        mem['h1e] = 'h020a;  //           jump  jloop
        mem['h1f] = 'h0059;  //  jdone:   load  zero
        mem['h20] = 'h0160;  //           store outnum
...
        mem['h55] = 'h0200;  //  done:    jump  start
        mem['h56] = 'h0060;  //  disply:  load  outnum
        mem['h57] = 'h0800;  //           out
        mem['h58] = 'h0202;  //           jump  iloop
        mem['h59] = 'h0000;  //  zero:    .data 0
        mem['h5a] = 'h0001;  //  one:     .data 1
        mem['h5b] = 'h0000;  //  i:       .data 0
        mem['h5c] = 'h0000;  //  j:       .data 0
        mem['h5d] = 'h0000;  //  k:       .data 0
        mem['h5e] = 'h0000;  //  l:       .data 0
        mem['h5f] = 'h0000;  //  prod:    .data 0
        mem['h60] = 'h0000;  //  outnum:  .data 0
        mem['h61] = 'h0000;  //  remain:  .data 0
        mem['h62] = 'h0000;  //  hdigit:  .data 0
        mem['h63] = 'h1000;  //  h1000:   .data 1000
        mem['h64] = 'h0100;  //  h100:    .data 100
        mem['h65] = 'h0010;  //  h10:     .data 10
        mem['h66] = 'h2710;  //  d10000:  .data 2710
        mem['h67] = 'h03e8;  //  d1000:   .data 3e8
        mem['h68] = 'h0064;  //  d100:    .data 64
        mem['h69] = 'h000a;  //  d10:     .data a
        mem['h6a] = 'h270f;  //  d9999:   .data 270f
        mem['h6b] = 'h0001;  //  istart:  .data 1
        mem['h6c] = 'hf000;  //  Ndelay:  .data f000
        mem['h6d] = 'h0000;
        mem['h6e] = 'h0000;
        mem['h6f] = 'h0000;
        mem['h70] = 'h0000;
...
```

So the memory contents that cause the computer to calculate this big sequence of prime numbers look like this: 006b 015b 005b 055a 015b 006a 065b 0455 005a 015c 005c 055a 015c 005b 065c 031f 005c 065a 015d 005d 055a 015d 065b 030a 005c 075d 015f 065b 0302 0413 020a 0059 0160 005b 0161 0666 0426 0255 0059 0162 0061 0667 0430 0161 0062 0563 0162 0228 0062 0560 0160 0059 0162 0061 0668 043d 0161 0062 0564 0162 0235 0062 0560 0160 0059 0162 0061 0669 044a 0161 0062 0565 0162 0242 0062 0560 0561 0160 006c 015e 005e 065a 015e 0356 0250 0200 0060 0800 0202 0000 0001 0000 0000 0000 0000 0000 0000 0000 0000 1000 0100 0010 2710 03e8 0064 000a 270f 0001 f000 0000 0000 . . . (more zeros)

There's not much to it! It's shorter than the player piano's musical score.

Just in case you're eager to know how I managed to convert the first ("assembly") format into the second ("machine" / hexadecimal) format, here is the Python program that I wrote to do the conversion:

http://positron.hep.upenn.edu/wja/P364_2012/sasm.py.txt

Such a program is called an **assembler**, because it converts human-readable **assembly language** into hexadecimal (or binary) **machine code**.

In Lab 27, we will look again through the implementation of this CPU, and you will have a chance to tinker with it a bit on your BASYS2 board. If you're interested in a preview, you can see how this worked out in the Fall 2012 lab at this link:
http://positron.hep.upenn.edu/wja/p364/2012/lab11g.html

This reading was probably somewhat exhausting to get through! And it touched on several topics that may be completely unfamiliar to you if you have not formally studied Computer Science. But I hope that I succeeded, nevertheless, at convincing you that a computer is really just a (somewhat complicated) state machine with some attached memory. And I hope that you were already convinced that a state machine and a memory are things that can be built up from logic gates and flip-flops, and that these, in turn, are things that we know how to build up from transistors. This conceptual reduction of the computer to transistor-level circuit components is the capstone of the digital portion (the last $\frac{1}{3}$) of this course.