EET 325
Microprocessors

Drexel University
Engineering Technology

Lab #3 – Starting the CPU module

Objectives
The objectives of this lab exercise are to introduce students to the RTL structures necessary for the creation of a soft CPU in Verilog. We will set the foundations in place for the creation of a custom soft CPU in Lab 4.

Introduction
In Lab 3, we will progress towards a working CPU that can make use of the 256-byte memory we constructed in Lab 2. We will eventually construct a CPU module that, when put into Run mode, takes over the memory system, reads from the memory, and does different things depending on what it finds there. (For now, we will leave it in Program mode.) We will also create several important registers and structures that we will use in Lab 4 to implement a working CPU.

A suggested CPU design is presented here, but from this point on, the design is increasingly your own. Try the approach suggested here, or if you want to do something differently, try that instead. Note your results in your report. ***Your eventual design should have some significant differences from mine.***

The suggested design involves two registers, A and B, and the use of four T-states. Recall from lecture that T-states correspond to the actual clock cycles provided to the CPU, since even a simple-looking instruction like **inc <address>** cannot be accomplished in one clock cycle on a simple architecture like we are considering here. If you're familiar with the "direct" or "extended" memory access from EET 401, this is similar. It is an instruction to read a specified address, increment that data by 1, and then write it back. All this cannot possibly happen in one clock cycle (on a simple 8-bit data-and-address-bus architecture); it has to be broken down into things that *can* happen in one clock cycle:

Microcode for instruction inc <address> ;Increments the data at <address> by 1
T0: [place <PC> on address bus][bring Write line low if needed]
T1: [read **inc** opcode] [place <PC+1> on address bus]
T2: [read address to be incremented] [place this address on the address bus]
T3: [read contents of address into a register][increment that register][leave address unchanged] [place updated data on the data bus][bring Write line high]

*T0 and T1 are always the same things for each instruction, in the sample design. Only T2/T3 differ. Sometimes (**nop** etc.), nothing at all is done during T3. Other times (like here), it's very busy!*

Note that if your design has registers A and B (for example), it will probably greatly confuse your users if A or B get modified when using this opcode. So, it should use a "shadow" register, not normally seen by the programmer. Or, alternately, make it clear in the datasheet that it "clobbers" data in A or B! (INC, as implemented in my design, simply uses ***memOut[7:0]=memIn[7:0]+1;*** *)*

Setup:
Open ICEStudio on your computer. Plug your TinyFPGA BX into the breadboard, if it isn't already. Load up your final project from Lab 2 and use Save As to make a copy of it as Lab 3_ex1.

(If you don't have a copy of your old code, you can use another group's, but you'll probably have to change the pinouts to match your design…)

Exercise 1: Setting up the CPU module and registers

At the end of Lab 2, you should have something like this:

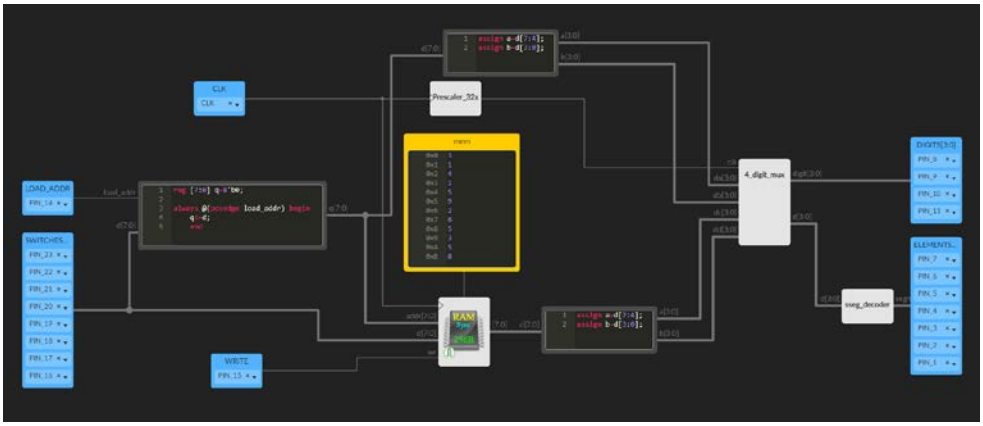

Figure 1. The final product from Lab 2: a 256-byte memory system

Make room in the center of your new Lab3 design for a largeish code module.
Create a new code module here, with:

inputs	clk, runMode, loadAddr, loadData, panel[7:0], memIn[7:0]
outputs	addr[7:0], memOut[7:0], write

Next, we will need to create registers to hold important information in this module.
You will need to translate the following into Verilog. ***Remember that bit size precedes names:***

reg [7:0]addr; //etc.

Create the following **8-bit** registers and set them (at least PC and addr) to 0x00 initially.

addr, PC, A, B, nextPC, memOut, curOpcode, curParam

Create the following **single-bit** registers:

runState, write, Z, C, N // ← Z, C, and N are flags; include them if your design does.
// (If in doubt, allocate them; they'll be implemented as needed.)

Finally, create a **two-bit register** (initialized to 0) to track the T state.
This will count from 0 to 3 and then revert back to zero in time for the next instruction.

tstate //2 bit register to track the current T state. Starts at zero.

Save your work (as Lab3_ex1), then Save As "Lab3_ex2" and continue.
Note that the above registers are sketched out and not given in proper Verilog. That's up to you.

Exercise 2: A CPU skeleton, keeping existing memory functionality

We'll need an always block to run the show, triggering the next action on the rising edge of the clock. All of the rest of the code that we write for the CPU module will go inside this block.
Here's the foundation stone of our CPU's execution structure:

```
always @ (posedge clk) begin

    //ALL of the code (except the register declarations above) goes in here

    end
```

Since some operations conclude by making the Write line high to initiate a write to memory, one of the first housekeeping tasks for every new cycle is to automatically lower the Write line to put the memory back in read-only mode. Nearly all of our code will be inside an **if** structure, but we can start off with a simple assignment (not an **assign**, but a simple non-blocking assignment):

```
write <= 1'b0;   //Lower the Write line (if not already low) to put memory in read mode
```

Next, we will write the outer control structure of the CPU. As with other aspects, you can vary this design if you like – check with me if you have any questions.
The suggested design uses the new **runMode** input:

- If **runMode** is low, then control passes to the front panel and the CPU is inactive;
- If **runMode** is high, the CPU assumes control (resetting itself as needed).

This can be done with an **if** structure, (*which goes inside the always@ block*):

```
if(runMode==0) begin
    //Program Mode code goes in here (rest of Lab 3)


    end else begin
    //Run Mode code goes in here (to be written in Lab 4)


    end
```

For now, we really only need to code up two things inside the first (Lab 3) part
Translate these to Verilog and add them to the first part:

- If **loadAddr** is high, update **addr** with the contents of **panel**; and
- If **loadData** is high, update **memOut** with the contents of **panel**, then bring **write** high.

CHECKPOINT: Double-check that you have the above code completed. (See if Verify runs OK.)

Save your work (as Lab3_ex2), then Save As "Lab3_ex3" and continue.

Exercise 3: Wiring up the new module

We now have a skeleton CPU module with enough Verilog code to work as a replacement controller for our memory project. (It also has all the groundwork for building a CPU in Lab 4, but that's for later.)
Now for the messy part – changing the (virtual) wiring over to the new design.
Make the following changes:

- Disconnect LOAD_ADDR, Write, and the switches (front panel)
- Disconnect the outputs from the load_addr code block (leftmost code block from Figure 1.)

Make the following connections from the CPU:

- Connect **MEM_OUT** on the CPU to **d** on the RAM
- Connect **Write** on the CPU to **wr** on the RAM
- Connect **q** on the RAM to **MEM_IN** on the CPU (leave it connected to the display unit, too)
- Connect the switches to **panel** on the CPU
- Connect **addr** on the CPU to the address display and to the **addr** input of the RAM.
- Connect a new toggle switch RunMode (find an open FPGA pin) to **runMode** on the CPU.
- Run the prescaled-down clock into **clk** on the CPU.
  (You may well have to scale it down further in Lab 4, but it shouldn't matter here.)

Your project should look something similar to Figure 2, below (maybe without the labels):

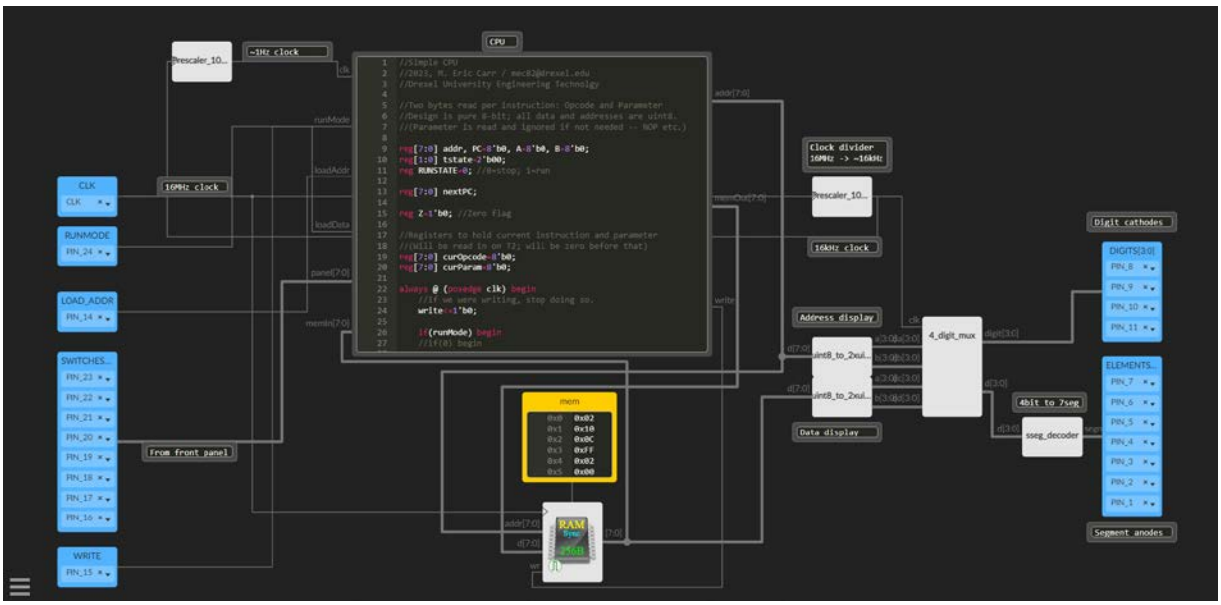

Figure 2. The CPU module, wired in and complete except for the CPU code.
(Don't mind the clocks too much at this point; they're kind of a mess, here…)

CHECKPOINT: **Project should work as it did at the end of Lab 2. (*Set runMode to LOW*)**
(Don't worry – we are in a great position to start writing CPU code, now!)

PLEASE LEAVE YOUR PROJECT ASSEMBLED; we will build on it later!
(Put your group members' names on a note attached to your project.)
PLEASE SAVE YOUR FILES – they will be very useful later!

Discussion

In your lab report, please make sure you cover the following:

- What equipment and software did you use? (Just a few sentences is fine; the idea is you want other engineers to be able to replicate and build on your work.)

- What did you set out to make the board do? How? Was it successful? If so, how does it work? If not, what do you think went wrong, and what could be done to address this? (Contact me for help if you're having problems with any lab. ICEStudio is still experimental, so we may have to develop workarounds if we run into bugs.)

- (Make sure to include any code that you create or modify, and note where it goes in the project.)

- Include a block diagram of your project. Describe the function of any modules included (such as the digit multiplexer and seven-segment decoder.)

- Include a schematic of your project (logic switches, including pull-downs; LEDs and series resistors, etc.) Include the new Write button that you wired up in Exercise 4.

- How much of the FPGA resources are we using in terms of:
    - Logic cells
    - RAM blocks
    - I/O pins (go by what we have available, not what IceStudio thinks!)
      - Notably, how many free I/O pins do we have remaining? (Any??)

END OF LAB 2