EET 325
Microprocessors

Drexel University
Engineering Technology

Lab #4 – Writing the CPU RTL*

*(Register Transfer Logic)

Objectives
In Lab 4, we will write RTL for a few basic instruction opcodes, creating a working CPU.

Introduction
In Lab 4, we will add the final pieces of a working CPU – the code that implements the common fetch (T0/T1) operations, as well as the individual opcodes (T2 and often T3).

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 significant differences from mine.***

The suggested design involves two registers, A and B, and the use of four T-states.
As mentioned in Lab 3 where we set up the registers, T-states correspond to the actual clock cycles provided to the CPU, since most instructions cannot possibly be executed in one clock cycle.
(Refer to Lecture 4 for details about the suggested CPU design.)

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 3 and use Save As to make a copy of it as Lab 4_ex1.

Exercise 1: Writing the common code for T0 and T1

The processor starts off each instruction in state T0, which has one important job – to place the number in the program counter register **PC** onto the address bus. (It does a few other minor tasks, too.)

We will be writing all of our code inside the IF statement controlled by the RunMode variable.

Inside the "Run mode" part of the IF statement, create a case structure to handle each T-state:

```
case(tstate)
        2'b00: begin
                //T0 code goes here
                end

        2'b01: begin
                //T1 code goes here
                end

        2'b10: begin
                //T2 code goes here
                end

        2'b11: begin
                //T3 code goes here
                end
```

Exercise 1, continued:

Next, we need to describe what should happen during T0 and T1. Recall that T0 simply copies the value in the program counter register to the address bus, so that T1 can load the opcode:

```
2'b00: begin
        //T0 code goes here
        addr = PC; //Put the value in the program counter on the address bus
        nextPC = PC + 2; //Set the next-program-counter value to PC+2 by default.
                         //(This can be changed by the instruction, if needed.)
        end
```

That's it for T0. Once we have the value of the program counter copied to the address bus, we need to wait for the memory to fetch the instruction. This takes a finite but nonzero amount of time, so we will simply end the state and fetch it in T1.

Next, we write the code for T1, which is responsible for copying the opcode from memory and putting the address for the parameter (PC+1) on the address bus:

```
2'b01: begin
        //T1 code goes here
        curOpcode = memIn; //Copy the opcode from the memory data bus
        addr = PC+1; //Put PC+1 on the address bus, to read the parameter.
        end
```

That's it for T1. We'll get to actually executing the instruction in T2 (and T3 if needed).

Save your work (as Lab4_ex1), then Save As "Lab4_ex2" and continue.

Exercise 2: Deciding on opcodes

Prep time. At this point, you need to decide what the first few opcodes will be, and what numbers (bytes) will correspond to each one. Slide 16 of Lecture 4 provides some useful instructions, so we'll implement two of these as a demo, and then you'll be on your own to implement others. (As always, ask me if you have questions or are stuck.)

Let's let opcode 00 be NOP (No Operation). We actually don't have to write anything for this; we can let it be handled by the "default" option that we will write for both T2 and T3. (It's always a good idea to have a default case in a case structure, in Verilog.)

As for the other opcodes, choose several that you want to implement, and pick opcodes for them. If you were designing a CPU by putting hardware modules together, it would be easiest to use "orthogonal" opcodes, where families of similar opcodes share common parts of the byte. For instance, maybe LDAI is opcode 02, and LDBI is opcode 04.

Once you know what numbers will correspond to at least a few of your opcodes, create a **casex** structure inside the T2 case. (We will also create one for T3). The **casex** structure works like **case**, but also allows for don't-care entries ('x'). We'll start off with opcode 01 for Load A, Immediate, and we will also include a "default" case, to handle NOP and unknown opcodes. (Leave "default" at the bottom.)

```
2'b10: begin
        //T2 code goes here
        curParam = memIn;    //Copy the memory output to the curParam register
        casex(curOpcode)     //Choose what to do, based on opcode

                //Opcode 02: Load A, Immediate
                8'bxxxx0010: //LDAI (Load A, Immediate)
                A=curParam; //Load register A from the current parameter
                z=(A==0?1:0); //Set zero flag to 1 if A==0; set it to 0 otherwise.

                default: //Default case (NOP or unknown opcodes): Do Nothing.
                end
        end
```

You can add other opcodes here easily enough – just create new casex options and add the code.
(This is still under T2, showing LDBI added after the existing LDAI code):

```
casex(curOpcode)      //Choose what to do, based on opcode

        //Opcode 02: Load A, Immediate
        8'bxxxx0010: //LDAI (Load A, Immediate)
        A=curParam; //Load register A from the current parameter
        z=(A==0?1:0); //Set zero flag to 1 if A==0; set it to 0 otherwise.

        //Opcode 04: Load B, Immediate
        8'bxxxx0100: //LDBI (Load B, Immediate)
        B=curParam; //Load register B from the current parameter
        z=(B==0?1:0); //Set zero flag to 1 if B==0; set it to 0 otherwise.

        default: //Default case (NOP or unknown opcodes): Do Nothing.
        end
end
```

Create a similar **casex** structure inside the T3 code section, since some opcodes will need that.
Leave a default case under T3; many opcodes (LDAI, LDBI, INCA, INCB etc.) will simply sit out T3.

```
2'b11: begin
        //T3 code goes here
        casex(curOpcode):

                default: //Default case (NOP or unknown opcodes): Do Nothing.
                end

        end
```

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

Save your work (as Lab4_ex2), then Save As "Lab4_ex3" and continue.

Exercise 3: More opcodes

Following the above example, create some more opcodes that you can use to write programs. Remember, you will be writing programs in this language to do basic tasks, so make sure you have useful opcodes to draw on. (Don't take it too seriously, though, since you can always add more!)

Here are some suggested opcodes that are relatively easy to implement, as well as useful:

~~LDAI (Load A, Immediate)~~ *Done, above*
~~LDBI (Load B, Immediate)~~ *Done, above*
LDAM (Load A, Memory) *(We'll do this next…)*
LDBM (Load B, Memory)
ADDI (Add Immediate to A)
ADDM (Add Memory to A)
WRA (Write A to memory)
JMP (Jump)
JNZ (Jump if Not Zero)

Let's implement LDAM, which needs to execute during both T2 and T3, to see how that works.

Inside the T2 section, add code for another opcode (change the number if you like):

```
//Opcode 03: Load A, Memory (T2)
8'bxxxx0011: //LDAM (Load A, Memory)
addr=curParam; //Put the current parameter on the address bus;
                    //we will read data from this location into A (in T3)
```

That's all that LDAM can do in T2, so the rest of the LDAM code goes in T3:

```
//Opcode 03: Load A, Memory (T3)
8'bxxxx0011: //LDAM (Load A, Memory)
A=memIn;      //T2 set up the address, so now we can read the contents into A
z=(A==0?1:0); //Set zero flag to 1 if A==0; set it to 0 otherwise.
```

Implement a few more opcodes (go through Slide 16 of Lecture 4 for some ideas.)
You don't have to implement the same opcodes that I do – and you should have at least a few new ones of your own – but come up with a set of instructions that you can use to do programming tasks.

CHECKPOINT: You have enough opcodes implemented to write a program with them, and your code compiles with no errors.

Save your work (as Lab4_ex3), then Save As "Lab4_ex4" and continue.

Exercise 4:

Once you have written enough opcodes to do some task, try them out. Write a program in the new assembly language that you are creating, to add the numbers from 0 to 10, and store the result in memory location 0xFF (so we can stop the CPU and read it back.) Here's an example.
(Note that the instructions are in even addresses and the corresponding opcodes are in odd ones.)

The easiest way to do this, once you have the opcodes, is to enter the bytes into the parameter for the RAM block, so this program is automatically copied into memory when your CPU starts up. (It's possible to toggle it in through the front panel if you want to kick it really old-school style.)

Opcodes shown are from my prototype design; refer to your list of opcodes that you have written!
(Make new ones up if you need them…)

//Sum the numbers 1 through 10. (Set A to 0 and B to 10; add B to A and decrement B until B is zero,
//then write the contents of A to 0xFF.)

Address	Contents	Notes
0x00	0x02	LDAI 0 (Load A, Immediate with the number 0)
0x01	0x00	
0x02	0x04	LDBI 10 (Load B, Immediate with the decimal number 10)
0x03	0x0A	
0x04	0x06	ABA (Add B to A) (ABA needs no parameter, so it ignores it.)
0x05	0x00	
0x06	0x13	DECB (Decrement B) (DECB also ignores its parameter)
0x07	0x00	
0x08	0x0F	JBNZ (Jump if B Not Zero, to address 0x04: go back to the ABA)
0x09	0x04	
0x0A	0x0C	WRA 0xFF (Write A to memory location 0xFF)
0x0B	0xFF	
0x0C	0x08	JMP 0x0C (Jump to the same address to create an infinite loop)
0x0D	0x0C	

Run the program and let it execute – you should eventually see it alternating between 0x0C and 0x0D. When it does, it has finished. *(Feel free to modify this program to use different opcodes if you like.)*

CHECKPOINT: **Your program can sum the numbers 1 through 10.**
You should see 0x37 (decimal 10) in address 0xFF, if it worked correctly.

PLEASE LEAVE YOUR PROJECT ASSEMBLED; we will build on it later!
(The final lab – Lab 5 – and report will count as your final project.)
(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??)

- Finally, based on the number of logic cells, how big do you think this would be if we had attempted to breadboard it? (Assume that the RAM is a single 28-pin chip.)

END OF LAB 4