CMPE 415

Homework 4

CPU Design, Simulation and Hardware Implementation

Phase 1 Due: Saturday March 14th , 11 PM

Phase 2 Due: Saturday March 28th, 11 PM

I. Introduction and Goals

This project involves the building of a simple, extremely streamlined CPU. It was designed in the interests of smooth implementation, rather than accuracy, so there are certain inherent design flaws in it, mostly related to high-speed timing; for our purposes, however, it works quite well.

I highly recommend reading this document carefully. If I saw a need to spend time writing something, it'll probably help you. With that said, there are many ways to implement a design project, some of which are objectively better than what's been tested, so keep an open mind.

I strongly advise you to write a testbench for each module, but the only ones which are strictly required will be listed in the deliverables section. You are going to need separate Block Designs for synthesis and simulation for the reasons described below in Simulation Considerations.

Also, read section XI of this document please. Those aren't binding instructions, but will help you quite a bit should you choose to follow them.

The goals of this project are:

- Develop high-level design skills while working on a relatively modestly sized system
- Practice creating, simulating, and synthesizing Vivado Block Designs
- Learn to use IP; both predefined Xilinx IP and your own packaged designs.
- Use Verilog to develop a register simulation
- Demonstrate the ability to perfect behavioral designs and multiplexing in Verilog (the meaning of this will become clear later in this document)
- Understand the function and implementation of basic CPU arithmetic instructions
- 7-Segment Display Control and Clock Division
- Button Debouncing and Input Filtering

II. ISA

When building a CPU, the first thing that must be decided is the number of bits the CPU controls and the ***Instruction Set Architecture***, or ISA. The ISA can be loosely defined as "what do I want my CPU to do", and normally includes arithmetic instructions (ADD, SUB, etc.), memory access instructions (LD, MOV from memory, etc.), IO instructions(IN, OUT, DDR, etc.) and control flow instructions (branch, jump etc.). For your sanity, we will only implement arithmetic instructions and extremely basic load functions; no external memory, IO, or control flow.

Some examples of ISAs most of you know are x86 (the assembler in CMPE 310), AVR (311), and MIPS (certain 411 sections would have seen that). As you are aware, the register structure and command syntax is significantly different. This CPU is probably closest to AVR of the 3 of them. (Don't stress about writing assembly – ***<u>you aren't going to</u>***)

When assembly programs are compiled, they essentially are compiled to binary or hex files. Essentially, every command consists of various binary bits for opcodes, source/destination registers, immediates (that means raw integers, like ADD EAX, 13 in x86 for example), memory addresses, and other useful info. **That** is the format of the instructions your CPU will run. More on this later.

This will essentially be a 16-bit CPU, which means the instruction size and register size are 16 bits. There are 16 available registers (just a convenient # for this format, not for any particular esoteric reason), which means every register address will be 4 bits (R0-R15 essentially)

All arithmetic operations will be performed on the 16 bit values stored in the registers. The one thing to note is that due to running out of switches, you cannot directly load in numbers greater than 8 bits. Don't worry too much about this.

The required commands and format are listed in table 1 below. O means opcode, D is the destination register # (between 0 and 15), S is source register (same), I is 8-bit immediate (-128 to 127), and – means don't care (doesn't affect the instruction):

Instruction	Format	Opcode(binary)	Description
MOV	OOOO----DDDDSSSS	0000	Moves Contents of RS into RD
LDI	OOOODDDDIIIIIIII	0001	Loads 8 bit # into RD
ADD	OOOO----DDDDSSSS	0010	RD = RD + RS
SUB	OOOO----DDDDSSSS	0011	RD = RD - RS
MUL	OOOO----DDDDSSSS	0100	RD = RD * RS (**careful)**
DIV	OOOO----DDDDSSSS	0101	RD = RD / RS
MOD	OOOO----DDDDSSSS	0110	RD = RD % RS
AND	OOOO----DDDDSSSS	0111	RD = RD & RS
OR	OOOO----DDDDSSSS	1000	RD = RD | RS
XOR	OOOO----DDDDSSSS	1001	RD = RD ^ RS

Here's an example. It loads 29 into R0 and 14 into R7 and adds them into R7, leaving R7 = 43 and R0 = 29:

```
LDI R0, 29          =>      0001 0000 00011101

LDI R7, 14          =>      0001 0111 00001110

ADD R7, R0          =>      0010 0000 0111 0000
```

III. Core Functions

The functionality is as follows. The user will arrange the 16 Nexys switches into the positions matching the instruction they would like to execute. ***The instructions will NOT immediately be executed, as this would require somewhat complex timing factors such as locking the CPU until division is finished, which takes about 20 clock cycles.***

Rather, to deal with this problem, the instructions will be processed by pushing BTNC, the center button of the “joypad” of your FPGA. This will require debouncing modules, which will be provided for you. When the button is pressed, the CPU will “execute” the instructions. Since you cannot conceivably push the button less than about 100,000 clock cycles (~10 ms) a 20 clock cycle timing delay will not be much of an issue.

As far as the 7 segment Displays are, the 7 segments will show the contents of the Destination and Source Registers read straight off the switches. Take the example above. When the ADD R7, R0 command is on the switches (**BEFORE THE BUTTON PRESS)** the 7-Segments will show 001C (29 in hex) on the rightmost 4 SSDs and 000E (14 in hex) on the left 4 SSDs. When the button is pressed the left 4 LEDs will show 2B (43 in hex)

The one exception is that in the case of LDI (SW[15:12] = 0001), instead of a Source Register the right SSDs will show 2 0s then the 8-bit immediate instead of the source register value (since LDI has no source register). This is important, as it complicates the SSD design a bit.

Your simulation, however, will not have the 7-Segment displays, as the outputs are not meaningful when examined in a sim. You may need to simulate them if they aren’t working however. You can find tutorials online to explain how SSD control on a Basys/Nexys works; feel free to base your work on those, but what you need to do is slightly more complex.

Sections IV – IX describe the blocks involved in my ALU; you don’t **have** to do it this way, but it’s probably in your best interests to.

A word about Block Diagrams – whenever you change your Verilog code for a block already in the BD, you need to go to the BD, right click on that module, and hit “Refresh Module”. ***Be very careful to do this; you’ll regret it if you aren’t.***

IV. ALU

You will not be writing the ALU – signed Multiplication and Division algorithms are incredibly difficult, even for fully trained engineers. The ALU will be built using preexisting Xilinx IPs following a step-by-step tutorial provided outside this document.

You will want to do this step first. You will need to add a block design and testbench for this part individually to make sure you got it correct, and it will give you a better understanding of the interconnecting parts here.

V. Register File

The register file will be extremely similar to the BRAM in HW2. It just has 2 read ports w/ read enables for RD and RS and 1 write port. The write register data and enable will be controlled by the MUXed output of the ALU.

One caveat is the register file will need a ***valid*** output. Just always set valid to 1 for this project. The reason will be clear from the ALU – you need to hook it up to the division module.

You are going to need to write a testbench for the register file by itself. This should be fairly similar to HW2 Q5, so you can adapt that one according to our specs.

VI. ISA Decoding

The 16 bit instruction will need to be decoded to find the ALU Operation and the Source/Destination Registers or Immediate. You will need to write a module to decide which alu op the ALU will write to the register and which register it will write to.

VII. MUX

The way the ALU is set up currently, for simplicity the result of any operation can be accessed at any time. This means that for ADD R1, R0, the outputs for MUL R1,R0, AND R1, R0, and all other R1,R0 instructions are available. You will use a MUX which will select the appropriate output to write to the register. This is quite like part of HW2 conceptually.

VIII. SSD

The overall function is described above. You'll need to pull the RD and RS data by connecting 16 bit inputs to the Register Module above (it makes the most sense, trust me) and pull the Immediate from the instruction decoder most likely. You need to conditionally display RS or IMM as above.

The one thing I will tell you, is that to display different numbers you need to set the Anode of the individual LED to 0 as such:

```
AN <= 8'b01111111; //sets the leftmost LED on
```

And the segments you would like on to 0 as such:

```
4'b0000: ssd <= 7'b0000001; (for 0, sets all segments to ON except the center one)
```

You will need to switch LEDs on a slow enough clock to see the difference. You need to use a clock divider to obtain a clock between 60 Hz and 1 kHz (doesn't matter too much as long as it's in the range) and change the value of AN to the individual SSD to 0 and the rest to 1.

IX. Debouncing

The plan is to provide you with the modules for the button debouncing. It's in your best interest to examine the modules, since you may see questions about debouncing and such on an exam or elsewhere and they are not complex.

The 1 thing you need to do is that after putting Filter.v in the BD, right click and select block properties or double click it it and change the params listed to : Bound = 64000 Wd = 16 N = 65535

X. Deliverables

1. Vivado project (Copy the ENTIRE outer directory into a ZIP folder)
2. Report. Includes pictures of BDs, simulation waveforms from the required parts, and high-level description of each module you built. Also include anything that doesn't work properly. If for some good reason your IO ports are not identical to that described in Section III above, **describe that in detail**.
3. Testbenches for ALU, Register Module, and whole design except SSDs and debouncing.

XI. Friendly Advice

If you're a bit lost as to where to start, I would advise the following order:

1. Create the project and build the ALU as in the tutorial accompanying this document
2. Write a thorough testbench for ALL ALU ops, being mindful that division will take **a lot** of clock cycles (20 as per the tutorial, so you need a very long wait statement to account for that) and multiplication will take 4-5 clock cycles. Try to only multiply 8 bit #s, because otherwise the overflow makes the numbers really difficult to read.
3. Build the register file very similarly to the block RAM from HW2. Simply set the VALID output to 1 all the time (it seems weird, but it's the easiest way)
4. Test the register file by itself, using the 2 read and 1 write enable and 2 read 1 write address and data ports
5. Hook the Register up to the ALU in a block diagram. You should hook RD and RS data up to A and B in the ALU (see tutorial), Write Data to the lowest 16 bits of the ALU output (you can only do MOD not DIV at this point, but don't stress it). Control the addresses and enables manually by adding ports in the BD for them.
6. Write a testbench for the Registers and ALU to show they are connected properly.
7. Write and test the Instruction Decoder and the MUX (they're heavily meshed with each other)
8. Hook up the MUX and Decoder to the Register file and ALU. You'll want the **32-bit** ALU output going into the MUX and a 16 bit output going out.
9. **At this point, your simulation block diagram is done.** All other modules will not really add much to this. Copy the BD (ctrl-A ctrl-C will work), create a new BD, and right click and paste. The 2nd BD will be the one you will be synthesizing.
10. Add ports for your inputs – a port to simulate the button, clock obviously, a 16-bit input to simulate the instruction switches, and RD, RS, and WRITE data outputs to see what's in the registers.
11. **Thoroughly test the simulation BD.** You want that to simulate flawlessly before you start worrying about hardware issues.

12. In your hardware BD, stick the debouncing modules between the button input and the module they go into (hint: probably the instruction decoder). These mess up simulation timing, which is why you need separate BDs.
13. Design the SSD and figure out where to attach it
14. Run, fail, and figure out what's wrong. The better your sims, the easier this will be to fix.
15. Write on resume you've implemented an FPGA CPU with a custom ISA using Xilinx IP

Deliverables:

Phase 1:

- Design the ALU with all the IP blocks as explained in the Tutorial.
- Write a testbench to test your ALU block
- Write a report to explain your ALU and all the functions
- In the report, explain what is your plan for the next phase. A Block Diagram of your initial proposed design is needed.

Phase 2:

- Submit the complete CPU design with all the modules
- Write a testbench to test the CPU
- Show the demo on FPGA
- Write a report and explain your design with a block diagram and details, which parts are working or not working