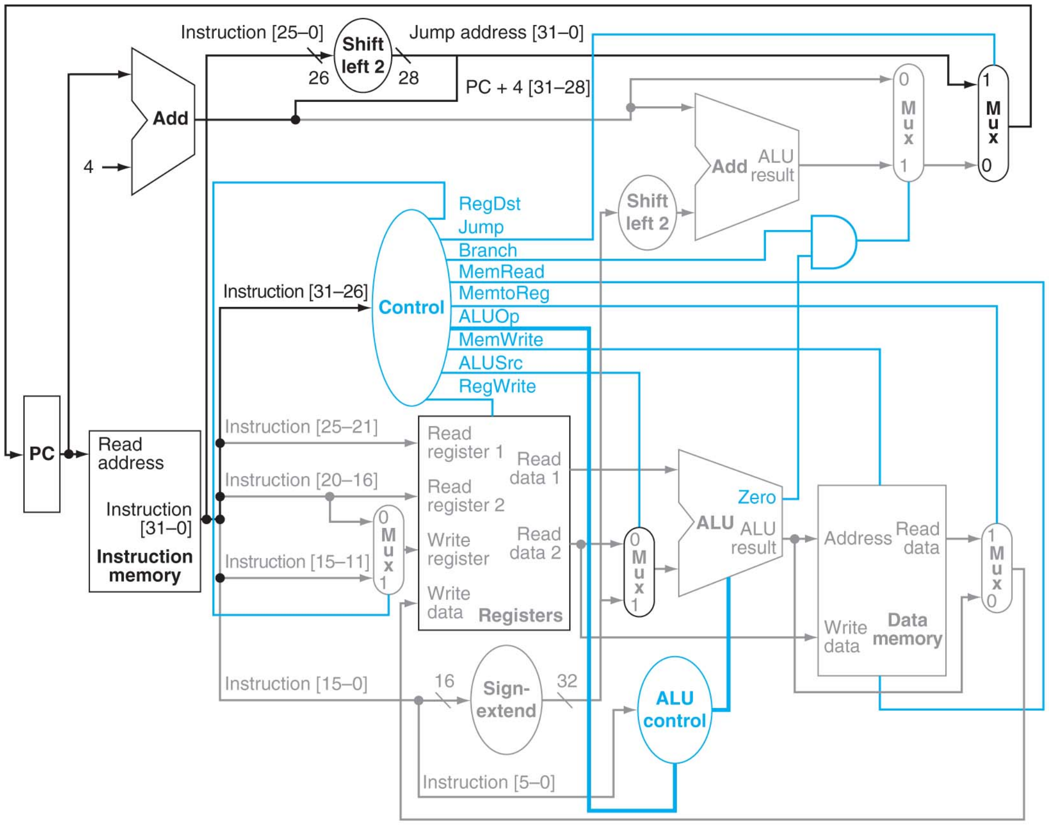
CprE 381: Computer Organization and Assembly Level Programming, Spring 2018

Project Part 2

[Note: this single-cycle processor is the second part of the term project assignment, and will involve substantial design, implementation, integration, and test tasks. You have three in-class weeks to complete this assignment (and also the week of spring break). To aid in project management, I have broken the tasks down into two parts, (a) and (b). Part (a) project files and demo should be completed in two weeks as a mandatory progress report. Part (b) files and final report are due the week after spring break. Your final grade will be based solely on the final files and written report.
***Disclaimer: Due to the complexity of the assignment, this document is subject to minor change between now and the due date – I will post any updates to Canvas so please continue to check Canvas regularly**.]*



0) Prelab. Sleep, exercise, shower, breakfast. Come to lab ready to work. Make sure you are familiar with the functionality of the following instructions:

add, addi, addiu, addu, and, andi, lui, lw, nor, xor, xori, or, ori, slt, slti, sltiu, sltu, sll, srl, sra, sllv, srlv, srav, **sw,** sub, **subu, beq,** bne, **j,** jal, **jr**

These are the instructions that you will have to fully implement for Part (a) and Part (b), respectively. Pale instructions are optional for teams of two and mandatory for teams of three. Should you need a reference, see P&H A.10.

[Hint: this lab can seem overwhelming at first – you are implementing your first processor. One way to reduce that feeling is to start by implementing and testing one instruction entirely during lab when you can ask your TA for assistance. My suggestion is to start with the addi instruction that you know your datapath supports since Lab3. I've included it as an example in the template control signals spreadsheet ("Proj2_control_signals.xlsx"). I've also included the test program as an example program in the test infrastructure ("MARsWork/Examples/addiSeq.asm"). I expect that you should be able to get this single instruction up and running within the first week, if not the first lab period given teamwork and guidance from TAs. After that, adding additional instructions effectively boils down to adding muxes, associated control signals, and associated decode logic.]

Part (a): Data-Handling Instructions

1) Up to this point in lab and your project, we have focused mainly on designing the datapath and manually generating control signals for testing. Now we will design the control logic portions to the processor. It may help to review your lecture notes as well as P&H 4.4 before starting this problem.

(a) Create a spreadsheet detailing the list of *M* instructions to be supported in this part alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your datapath implementation. The end result should be an *N*M* table where each row corresponds to the output of the control logic module for a given instruction. *[The control signals listed in P&H 4.4 will not be sufficient and may differ slightly from your datapath design. I suggest annotating the spreadsheet with a description of each instruction's purpose, as well as the purpose each of your control signals. You may find that you need to update/modify control signals (and your datapath) in order to add all of these instructions. Please keep updating this spreadsheet as you make modifications and work on Part (b).]*

(b) Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually, and show that your output matches the expected control signals from problem 1(a). *[There are many different ways to do this. While a large lookup table might be the easiest from a coding perspective, keep in mind that since this is a single-cycle processor the control logic must be combinational. Large control tables are commonly implemented using Programmable Logic Arrays (PLAs), which in VHDL you can emulate using with/select statements. This is covered at a high level in P&H D.2 with syntax examples in Free Range VHDL Chapters 4 and 5.]*

2) At this point, the major components should be in place for you to be able to implement the **MIPS Straight-Line Single-Cycle Processor** using only structural VHDL. As with the previous datapath designs that you have implemented, start with a high-level schematic drawing of the interconnection between components (you will continue to add to this in Part b). You will be required to use the very brief skeleton code included with the testing framework as a base for your design ('ModelSimWork/src/MIPS_Processor.vhd'). First, it allows the automated testing framework access to the relevant architectural components within your design. Specifically, it will track the sequence of register and memory writes to compare with those from MARS. Second, it instantiates the memory under known labels so that the instruction memory can be loaded automatically. Third, it implements halting behavior (so your simulation ends). You should inspect this file as you draw your initial schematic since it will require you to appropriately name certain signals and likely add ports to your modules (e.g., “v0” from your register file). Some general hints as how to proceed:

- Start your schematic by drawing the components already instantiated in the skeleton processor code.
- The MIPS data memory is byte addressable (i.e. every 32-bit address specifies a byte in memory), while the data memory component created in Lab #4 is word addressable. Consequently, a load request for address `0x104` should return the 65th element in your initialization file, not the 260th element in that file.
- Your processor may need to be “reset”, in the sense that the register file is cleared and the PC is set to some predetermined initialization address. The testbench assumes that the iRST input causes all registers and flipflops to be reset.
- You may need to add a couple MUXs and control signals to accommodate certain instructions (e.g., **`lui`** and shifts) that are not shown in the textbook implementation – DON’T PANIC, you can do this!

3) Test your processor to ensure that it can fully implement all of the above data-handling instructions. You should use the automated testing framework provided on Canvas. Follow the directions provided in the framework Zip (“CprE 381 Toolflow manual.pdf”). In brief, you must place all of your vhdl source files in the “ModelSimWork/src” directory. Then double-click on “Run_Test_Framework.bat”. The automated test framework will assemble, simulate (in MARS and Modelsim), and compare your design’s state (i.e., memory and architectural register file) updates with MARS to test your design. You will need to specify assembly programs that fully test your processor to increase your confidence in it. Simple test programs are included in “MARsWork/Examples/”. *[Hint: use the simple examples containing only a single instruction type, such as addi, to learn to use the framework and to test/debug your initial design.]*

(a) Create a test application that makes use of every arithmetic/logical/shift instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 1 per cycle while data written into registers can be effectively retrieved and used by later instructions). Provide a waveform demonstrating this test application running correctly on your processor.

Part (b): Control Flow Instructions and High-Level Testing

4) Update your control logic table and implementation from Part (a.1) to include control flow instructions.

5) Given control flow instructions you also need to modify your instruction fetch logic from only being PC+4.

(a) What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions from Part (a.1.a).

(b) Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed? *[Figure 4.24 (reproduced above) does not consider all of the necessary possibilities. Your answer to this problem should be consistent with that of Part (b.5).]*

(c) Implement your new instruction fetch logic using structural VHDL. Use Modelsim to test your design thoroughly to make sure it is working as expected. Describe how the execution of the control flow possibilities corresponds to the Modelsim waveforms in your writeup.

6) Testing your processor will require more effort than what you have done for past labs, as the interaction between instructions now potentially affects every single component. In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

(a) Modify your application from Part (a.3) to include tests for each control flow instruction in addition to the data-handling instructions. Confirm your processor is completely functional.

(b) Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link).

(c) **[Optional]** Create and test an application that sorts an array with *N* elements using the MergeSort algorithm (link).

7) You will be expected to **demo** your single-cycle implementation to the TAs by the project due date. Each member of the project group will be required to be present for the demo, which will take place during regular lab hours. During this time, you will describe the various components of your design and how they work together, you will show a waveform (the automated testing framework dumps the waveform at "ModelSimWork/vsim.wlf") of the three test applications from Part (b.6), and you will also run a fourth test application that will only be provided to you during the demo.

8) As we are learning about in lecture, performance (runtime for our purposes) depends on several factors: # of instructions that will be dynamically executed by an application, # of cycles each instruction takes to execute on average, and the length of the cycle. Up to this point you know the first two factors for our applications. Now you will **synthesize** our design to the DE2 board's FPGA to determine the maximum cycle time. Using the automated synthesis script provided in the testing framework ("Run_Synthesis.bat"), report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematics. What components would you focus on to improve the frequency? *[As a note, synthesis may take 50+ minutes on 2050 Coover machines and 2.5+ hours on VDI instances. Please plan accordingly and recognize that your processor must functionally work in simulation before you synthesize it.]*

Credit: Parts of this project description were originally created by Dr. Joe Zambreno.