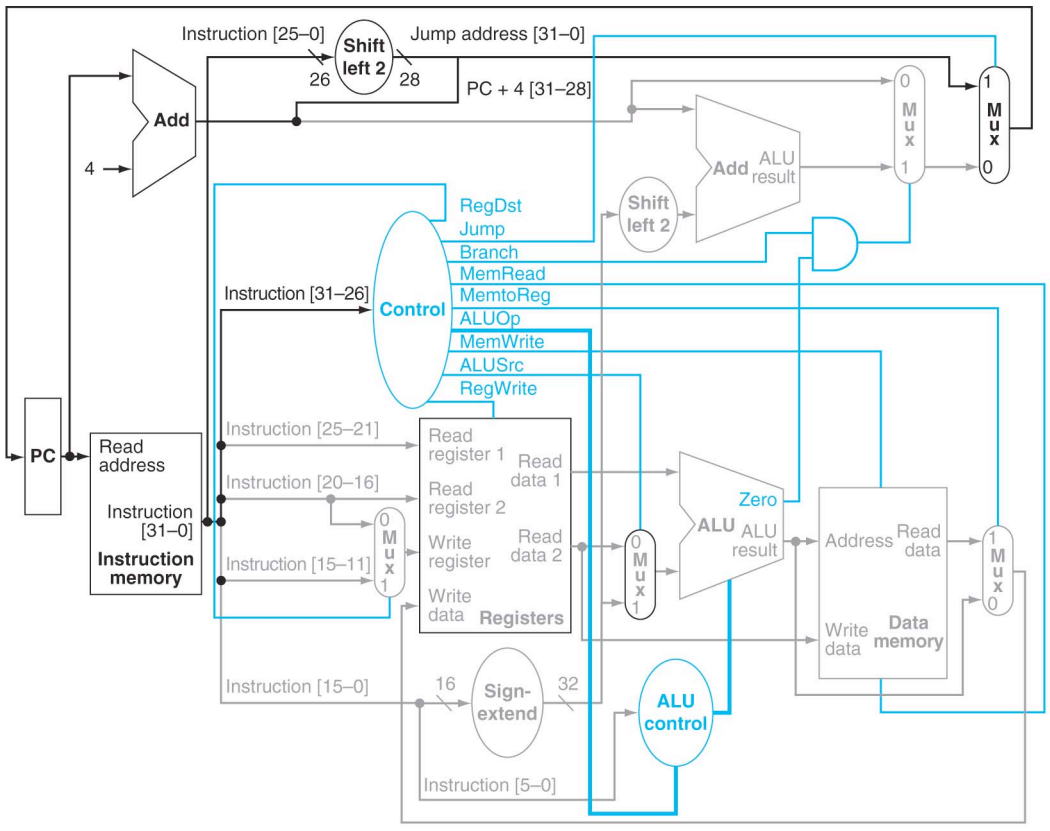
CprE 381: Computer Organization and Assembly Level Programming, Spring 2018

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 four weeks (including Spring break) total to complete this assignment. To aid in project management, I have broken the tasks down into two parts, (a) and (b). Part (a) project files and demo must be completed in two weeks, while Part (b) demo and files as well as the written report are due two weeks after part (a).
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. Should you need a reference, see P&H A.10.

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) Another component we have not yet designed in the previous labs is the **Instruction Fetch Logic**. Since the instruction memory in the single-cycle processor is a read-only memory, and since it should be read asynchronously, you should use the same generic memory component we provided for Lab #4.

(a) Draw a schematic for the basic (i.e., common case, PC+4) instruction fetch logic required by Part (a) instructions.

(b) Provide a *simple* testbench that confirms that you can instantiate and test the *mem.vhd* component as would be needed for a MIPS instruction memory. Briefly describe what ports are not needed for instruction memory, and how you wired the *mem.vhd* module accordingly.

(c) Implement your basic fetch logic using structural VHDL. Use Modelsim to test your design to make sure it is working as expected.

3) 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. Some general hints as how to proceed:

- 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.
- Also, since every instruction should execute in a single cycle, the data memory should be logically separate from the instruction memory. Although it is relatively straightforward to modify *mem.vhd* to have two parallel ports, your design will be considerably simpler if you map two separate instances instead.
- 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. Implementing this as a single control signal from a VHDL testbench will be much easier than requiring a series of "reset" instructions.
- You may need to add a couple MUXs and control signals to accommodate certain instructions (e.g., **`lui`** and variable shifts).

In your writeup, describe what challenges (if any) you faced in implementing this module.

4) Test your processor to ensure that it can fully implement all of the above data-handling instructions. This should effectively be a MIPS encoding of the testing you did for Project Part 1 with the addition of a couple instructions (i.e., **`lui`** and shift variable instructions). *[You should use the SPIM or MARS simulator to test that your applications work properly in simulation, convert the assembly to MIPS machine code, and verify that the results are consistent with what you are seeing from your VHDL implementation.]*

(a) Create a test application that makes use of every arithmetic/logical/shift instruction. 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

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

6) Given control flow instructions we also need to modify our **Instruction Fetch Logic**.

(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.

7) 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. *[You should use the SPIM or MARS simulator to test that your applications work properly in simulation, convert the assembly to MIPS machine code, and verify that the results are consistent with what you are seeing from your VHDL implementation.]*

(a) Modify your application from Part (a.4) 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). *[Since you will have to be implementing and using many test programs throughout the rest of the term project, I suggest that you aim to automate the process of generating and initializing memory contents as much as possible.]*

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

8) 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 simulation of the three test applications from Part (b.7), and you will also run a fourth test application that will only be provided to you during the demo. *[Your processors and toolflow are not expected to be perfect!]*

9) 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 same synthesis procedure as Project Part 1, report the maximum frequency your processor can run at and determine what your critical path is.