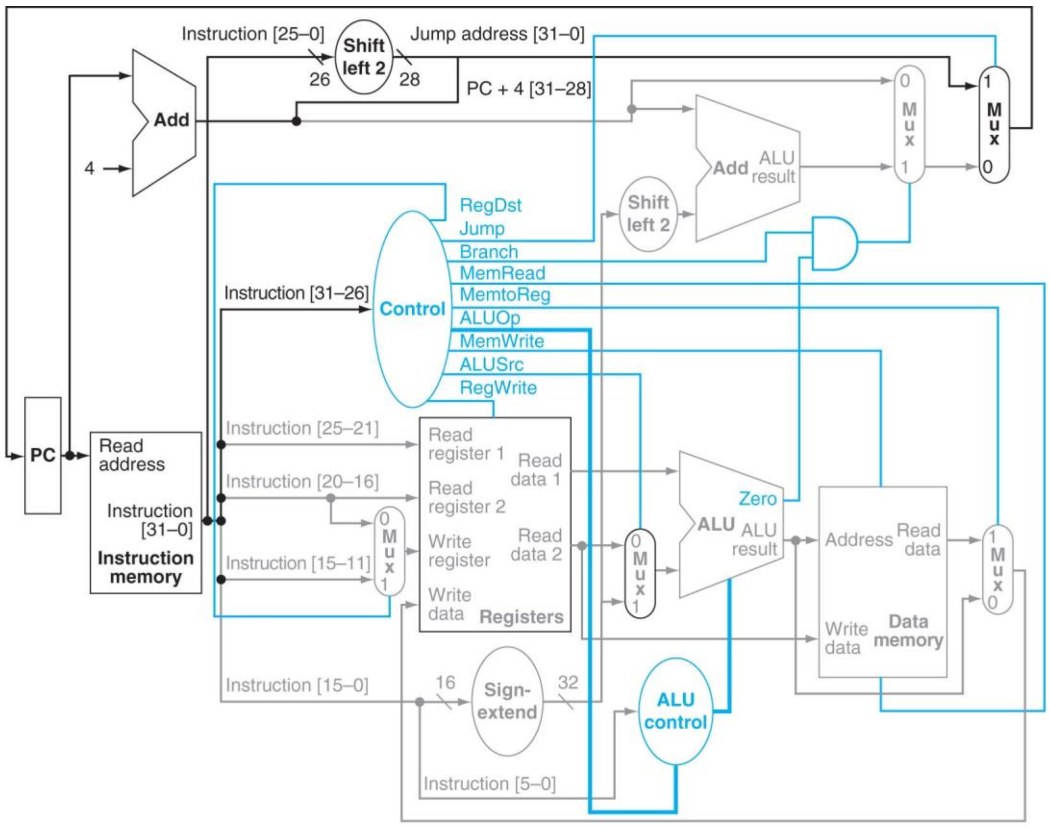
CprE 381 – Computer Organization and Assembly Level Programming

Project Part B

[Note from Joe: this single-cycle processor is the second part of the team project assignment, and will involve substantial design, implementation, integration, and test tasks. You have four weeks to complete this assignment, and I expect that most teams will need the entire allotted time. To aid in project management, I have broken the tasks down into two phases, ***I*** *and* ***II****, each of which should take approximately two weeks. 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. Download and review the "CprE 381 Processor Lab Toolflow" and supporting documentation from Canvas. In your writeup, answer the following questions, referencing individual files in the toolflow as needed. Your answers may be brief, but please be precise:

- How are instruction and data memory initialized in the simulation? How does MARS interface with ModelSim?
- In the MIPS skeleton VHDL, how is a halting / termination condition detected? What MIPS assembly instruction does this correspond to?

In addition, make sure your team is 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 **Phase I** and **Phase II**, respectively. Should you need a reference, see P&H A.10.

Phase I: Data-Handling Instructions

1) We have focused mainly on datapath elements to this point in the course, but the **Control Logic** plays an equally important role in any processor implementation scheme. For previous labs we have manually generated the necessary control signals, but from this point forward we must design and implement logic to automate this task. It may help to review your lecture notes as well as P&H 4.4 before starting this problem.

(a) Modify the provided spreadsheet to include the list of *M* instructions to be supported in this phase alongside their binary Instruction 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 in your single-cycle processor implementation. The end result should be an *M*N* 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. I suggest annotating the spreadsheet with a description of each instruction's purpose, as well as the purpose of your control signals. It is likely that your team will need to update/modify control signals (and your datapath) in order to add all of these instructions, and that you will have to continue to edit/update this spreadsheet several times over the next few weeks.]*

(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 part 1a). *[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 describe using with/select statements. This is covered at a high level in P&H D.2 with syntax examples in the VHDL tutorial 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 Phase II). Some general hints as how to proceed:

- Start your schematic by drawing the components already instantiated in the skeleton processor code provided in the testing framework.
- 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-04 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 will 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.
- As previously mentioned, you will need to add controls signals and corresponding muxes to accommodate certain instructions (e.g., **`lui`** and the various shifts) that are not shown in the textbook implementation.

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

3) Test your processor to ensure that it can fully implement all of the instructions for this phase. You are required to use the automated testing tooflow, so please follow the directions provided in the tooflow documentation. Assuming your VHDL source files are in the appropriate directory, the automated test framework will assemble, simulate (in both MARS and ModelSim), and compare your design's state changes (i.e. updates to memory and register file values). Simple test programs are included in `"MARsWork\Examples"`. *[I strongly suggest you start with the simple examples containing only a single instruction type, such as* **`addi`***, to learn how to use the framework and how to test/debug your initial design.]*

Create a test application that makes use of every **Phase I** 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 and memory can be read and used by later instructions). Attempt to consider edge cases. Include this file in your submission as `Proj-B_test1.s`.

Phase II: Control Flow Instructions and High-Level Testing

4) Instructions that modify program control flow will require non-trivial updates to your **Instruction Fetch Logic**, to support cases other than PC = PC + 4.

(a) What are the control flow possibilities that your instruction fetch logic must support? In your writeup, describe these possibilities as a function of the different **control flow-related instructions** included above.

(b) Update your control logic spreadsheet, corresponding VHDL implementation, and testbench from Part 1) to include these control flow instructions.

(c) Update your processor schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. In your writeup, describe what additional control signals are needed. *[Figure 4.24 (reprinted above) does not consider all of the necessary possibilities. Your answer to this problem must be consistent with that of your updated spreadsheet from Part 4.b)]*

(d) Implement your updated schematic 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.

In your writeup, provide your schematic for this part, and describe what challenges (if any) you faced in implementing control flow instructions.

5) Fully **Debugging** your processor will require more effort than what you have done for past labs, as the interaction between instructions for meaningful applications 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 3) to include tests for each **Phase II** instruction in addition to the data-handling instructions. Name this file `Proj-B_test2.s`. Confirm your processor is completely functional before proceeding.

- **(b)** Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file `Proj-B_test3.s`.

- **(c)** **[BONUS POINTS]** Create and test an application that sorts an array with *N* elements using the MergeSort algorithm (link). Name this file `Proj-B_test4.s`.

6) 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 or office hours. During this time, you will describe the various components of your design and how they work together, you will show simulation of the test applications from Part 5), and you will also run an additional test application that will only be provided to you during the demo.

7) 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, we have explicitly designed the first two factors for our processor and software applications, and have assumed the third. Now you will use a **Synthesis** tool to map your design to the DE2 board's FPGA to determine the maximum cycle time. Using the automated synthesis script provided in the testing toolflow ("`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 schematic. In your writeup, briefly discuss your critical path results. What components would you focus on to improve the frequency? *[Note: synthesis may take 50+ minutes on the 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.]*