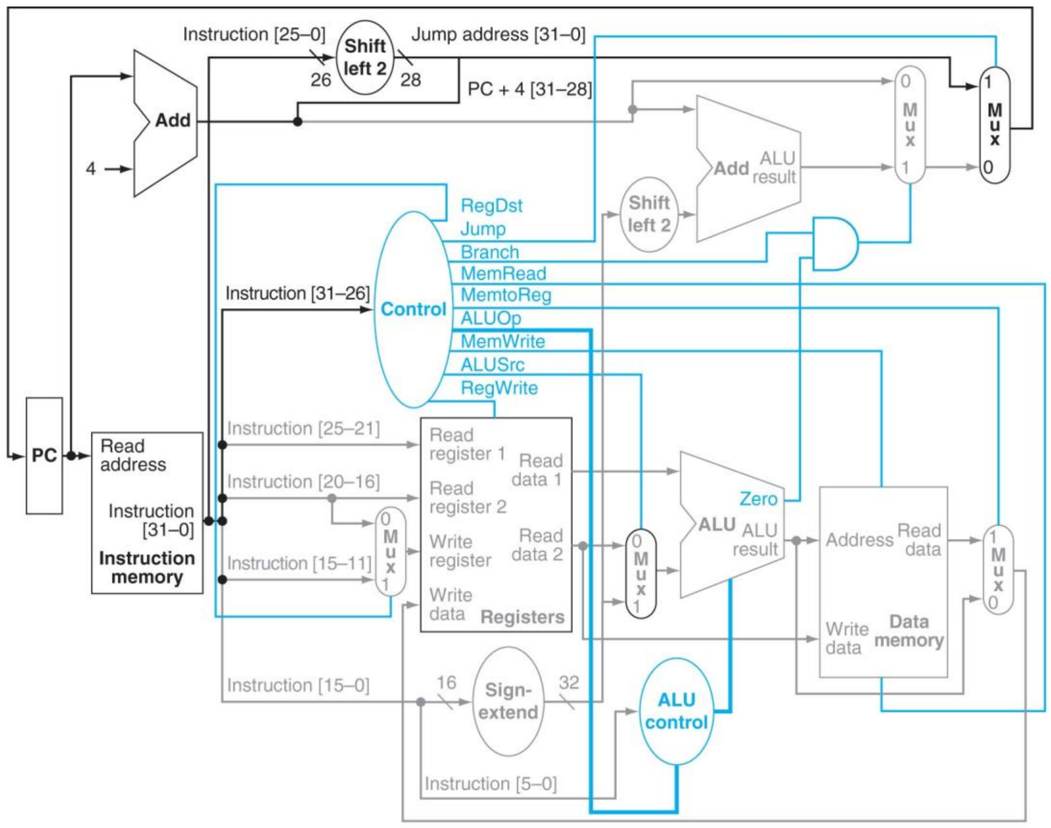
CprE 381 – Computer Organization and Assembly Level Programming

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. 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 Blackboard so please continue to check Blackboard regularly.]



0) Prelab. P&H B.10 provides an exhaustive listing of the MIPS instruction set (both literal and pseudo-instructions). Of the entire list of literal MIPS instructions in this section, which instructions will your processor be able to support, based on the components you've created in previous labs? *[I count 45 literal MIPS instructions that your processor should be able to support. Your mileage (and project score) may vary. Consider the following:*

- *The `mult` instruction variants require a significant redesign of the datapath, whereas the `mul` instruction does not. You can safely ignore all the `mult` instructions as long as you support `mul`.*

- *Supporting the 'branch-and-link' type instructions should not provide much of a burden beyond what is already required for* `jal/jalr`. *Your ALU should be able to support all the branch variants.*
- *Trap instructions need not be supported.*
- *Only worry about the load/store instructions covered in class. Move instructions and unaligned load/store need not be supported.*
- *Floating point instructions need not be supported.*
- *The* `nop` *is listed as a native instruction but it is actually encoded as a* `sll` *instruction (possible typo on P&H's part). Don't count* `nop` *as part of the 45 MIPS instructions to support.*
- *You will find* `lui` *is used quite often to help deal with 32-bit constant values. It needs to be implemented and is fairly straightforward.]*

Verify your instruction listing with your TA before proceeding.

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. Glance through P&H 4.4 before starting this problem (control logic will be eventually covered in class in more detail).

(a) Create a spreadsheet detailing the list of *M* instructions to be supported 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 *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. 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 have 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 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 case/select statements. This is covered at a high level in P&H C.3 with syntax examples in the VHDL tutorial Chapters 5 and 6.]*

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) 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 problem 1a).

(b) Draw a schematic for the instruction fetch logic. 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 problem 1a).]*

(c) 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.

(d) Implement your 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.

3) At this point, the major components should be in place for you to be able to implement the **MIPS Single-Cycle Processor** using only structural VHDL. As with the previous processors 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 270th 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.

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

4) 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) Create and test an application that makes use of every single instruction. The application need not perform any particularly useful task.

(b) Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). How you initialize memory is up to you, but the toolflow should be semi-automated.

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

5) Finally, 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 problem 4), 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!]*