CprE 3810: Computer Organization and Assembly Level Programming
Project Part 1: RISC-V Single-cycle Processor

*[**Note:** This is a **TEAM** assignment and should be actively collaborated on and submitted as a single project team. Although you are still required to document your design process via a lab report, your focus should be more on creating a functioning, easy-to-understand design and testing it thoroughly than on a lengthy writeup. The TAs and instructor are still available to help during the lab sessions, office hours, and in the Lab Help channel, but there is now much more freedom in how to implement components and you are expected to come up with major aspects of the design as a project team. Fortunately, you have **four** course weeks to complete this assignment.]*

Below is the book's block diagram of the baseline RISC-V processor to give broad guidance. This is a good representation of what your project should look like, but it is incomplete relative to the set of instructions you will need to implement.

Here is my rough advice on time targets:

- *By the end of Lab section 2(week 6 of the course), I would expect you to have completed the team contract assignment part 0. I would also expect you to have completed the high-level design and begun designing the required submodules.*
- *By the start of the **second** Proj 1 lab section (i.e., week 7 of the course), I would expect most submodules – fetch unit and control unit – to be mostly implemented and under test.*
- *By the start of the **third** Proj 1 lab section (i.e., week 8 of the course), I would expect an integrated "processor" that executes an addi-only program.*
- *By the start of the **fourth** Proj 1 lab section (i.e., week 9 of the course), I would expect an integrated "processor" that executes at least one conditional branch and most data-flow instructions. You may still have bugs with respect to edge cases – signed vs unsigned slt, etc. – and still have a couple control-flow instructions left to implement – jal, jalr, other branches – but the processor is essentially complete. At least one team member should have tried synthesizing a mostly-functioning processor so any questions about synthesis can be answered (as a warning, synthesis can take an hour or more for a single-cycle processor and it can bring to the forefront issues with poor VHDL design practices).*

***Disclaimer*: Due to complexity of the assignment, this document is subject to minor change before the due date. Updates will be posted to the Lab Help channel.**

By the end of this part, your processor must implement:

`add, addi, and, andi, lui, lw, xor, xori, or, ori, slt, slti, sltiu, sll,`

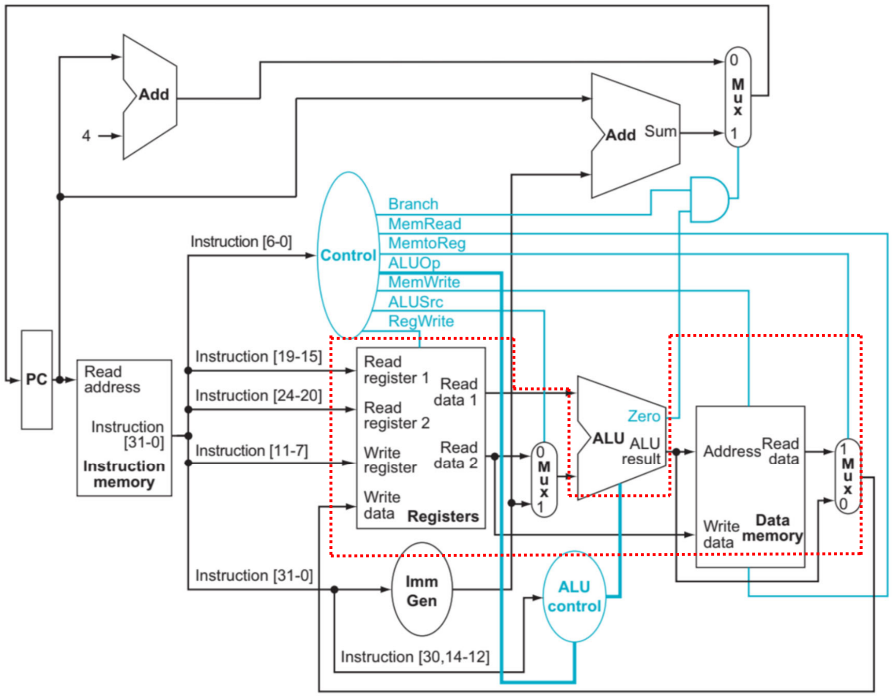

Figure 1: Baseline RISC-V processor block diagram (from textbook).

`srl, sra, sw, sub, beq, bne, blt, bge, bltu, bgeu, jal, jalr, lb, lh, lbu, lhu, slli, srli, srai, auipc, wfi (or HALT)`

For those interested, extra credit (see syllabus for full policy) is offered for implementing exceptions, coprocessor registers, DSP/SIMD instructions, or even an FPGA wrapper for the truly adventurous. Make sure to report your design approach, resources used, and annotated waveform in your report. Include your test cases in your submission.

1 Teaming

We will begin by organizing ourselves into your teams (you can find your team in Canvas at the start of your lab).

a. *Get your larger team assignment and discuss teaming logistics (e.g., meeting times, communication strategies, coding standards, and version control/code sharing).*

b. *As a team, look at the above high-level design. Skim read the lab manual and identify the varies parts (see team contract for a tabular version). Complete your team contract.*

c. *Select which code base you will start with or which components you will use from each team.*

2 RISC-V Single-cycle Processor

Through previous labs, lecture, and homework, you have now learned enough to be able to implement a RISC-V Single-Cycle Processor using mainly structural VHDL. As with the previous datapath designs that you have implemented, start by drawing a high-level schematic of the interconnection between components similar to the one provided above. You will be required to use the very brief skeleton code included with the testing framework as a base for your design (consult the "cpre3810-toolflow.pdf" manual included in the framework ZIP on generation/usage – specifically, part 2.1). 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 RARS. 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 generated file as you draw your initial schematic since it will require you to appropriately name certain signals and possibly add ports.

a. Start your schematic by drawing the components already instantiated in the skeleton processor code.

b. Sketch in the remaining high-level components you need to add for the basic instructions (see the above figure, lecture slides, or the book for more details). You are not required to keep the names from the book figure, although you must not rename the signals in the skeleton code.

c. Read through the remaining parts of the lab and add any additional components and signals required by the set of instructions not included in the book's schematic. *[I would expect these signals could change slightly throughout the course of the project as you gain more insight into your design. Update your schematic as needed throughout the project.]*

d. Include your final RISC-V processor schematic in your lab report.

e. General tips/reminders:

   (a) The RISC-V data memory is byte addressable (i.e. every 32-bit address specifies a byte in memory), while the data memory component created in Lab 2 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.

   (b) 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.

   (c) 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.

   (d) Working completely separately and integrating on the final week is a risky approach. My suggestion is to build your processor up by instruction. Start with the addi instruction that you know your datapath supports after Lab2. I've included it as an example in the template control signals spreadsheet ("Proj1_control_signals.xlsx"). I've also included

test programs as example programs in the test infrastructure (i.e. "proj/riscv/addiSeq.s" after following project initialization instructions in "cpre3810-toolflow.pdf").

3 Component Implementation

3.1 Control Logic

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 your project 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 may need to update/modify control signals (and your datapath) in order to add all of these instructions.]*

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.]*

3.2 Fetch Logic

In addition to the control logic, you will also need to implement the fetch logic that updates the PC (i.e., the address of the next instruction to fetch from memory).

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 you are required to implement.

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.]*

c. Implement your new instruction fetch logic using VHDL. Use QuestaSim 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 QuestaSim waveforms in your writeup.

3.3 Complete RISC-V ALU

You will update your ALU to meet the full requirements of the RISC-V ISA. Your RISC-V ALU will take two operand inputs (operand A and operand B) and a set of control inputs (based on your design). It will have four outputs (result F and Zero). It must support the following operations:

`add/sub, addi/subi, slt, and, or, xor, nor, sll, srl, sra, slli, srli, srai`

1. The RISC-V ISA contains several shift instructions. For this part we will consider the **barrel** shifter design discussed in class. [Add the barrel shifter ONLY once you have a basic fetch unit and control logic unit designed and implemented.]

   (a) Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does RISC-V not have a `sla` instruction? *[This may require you to look up these instructions in an ISA specification such as P&H textbook A.10.]*

   (b) Unfortunately, barrel shifters are not covered explicitly in the P&H textbook, but the following Java applet shows how an 8-bit barrel shifter can be created using cascaded 2:1 muxes (as described in lecture and shown here). Implement a 32-bit **right shifter** (both arithmetic and logical) using structural VHDL. In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations. *[There is a strong pattern to how the muxes are mapped in the conventional barrel shifter design. If you make sense of this pattern your code will be considerably simplified.]*

   (c) In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations. Integrate this functionality into your existing design. *[Hint: it can be done by trivially modifying the input and output. An additional shifting component is not necessary.]*

   (d) Use QuestaSim to test your shifter designs thoroughly to make sure they are working as expected. Describe how the execution of the different shifting operations corresponds to the QuestaSim waveforms in your writeup.

2. For any additional functional units you design, go through the design process for each component.

   (a) In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

   (b) Use QuestaSim to test your additional components thoroughly to make sure they are working as expected. Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

3. Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: How is Zero calculated? How is slt implemented? *[The answer to these questions can be found in the lecture slides. Verify with your TA before proceeding. Your design may require you to update the implementation of your adder/subtractor.]*

4. Implement this 32-bit ALU using structural VHDL. *[Leverage as many of the components you and your partner have already designed and tested as possible!]*

5. Use QuestaSim to test your 32-bit ALU thoroughly to make sure it is working as expected. Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

6. You should now be able to demonstrate using a **test program** a much wider variety of behavior than your Lab 2 datapath.

7. Integrate your new ALU into your RISC-V datapath from Lab 2.

8. Augment your previous test benches(s) relevant to arithmetic/logic operations (i.e. from the adder/subtractor) into a new testbench ("tb_alu.vhd") that tests the new functionality from this lab (e.g., `slt, or, and, nor, xor, sll, srl, sra`, etc.). Don't simply test that they work for one value, test them for multiple values including edge cases and justify why your test plan is comprehensive. Include waveforms that demonstrate your functioning test bench.

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. You should use the automated testing framework provided on Canvas. Follow the directions from the toolflow manual provided in the framework ZIP ("cpre3810-toolflow.pdf"). In brief, you must place all of your vhdl source files in the "/proj/src" directory with a subdirectory structuring of your choice after initializing a new project. Then, using the "test" command with the given shell script ("3810_tf.sh"), specify which test program(s) you'd like to simulate. Consult the toolflow manual for a more detailed understanding (part 2.2). The automated test framework will assemble, simulate (in RARS and QuestaSim), and compare your design's state (i.e., memory and architectural register file) updates with RARS 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 "/proj/riscv". *[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.]*

In your writeup, show the QuestaSim waveform 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. Create a test application that makes use of every required arithmetic/logical 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). Name this file Proj1_base_test.s.

b. Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.

c. Create and test an application that sorts an array with N elements using the Mergesort algorithm(link). Name this file Proj1_mergesort.s.

5 Synthesis

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 framework component of

the toolflow, utilize the "synth" command with the given shell script ("3810_tf.sh" – consult the toolflow manual, part 2.3, for more information). 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 from part 1. 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.]*

5.1 Turn-In Checklist

- Complete the lab report (using template Proj1_report.doc on Canvas).
- Use the toolflow to generate your submission by using the "submit" command of the shell script ("3810_tf.sh" – consult the toolflow manual, part 2.4, for more information).

5.2 File Structure

```
Canvas
├── Proj1_report.pdf
└── submit.zip
    ├── src files folder containing all of your VHDL design files
    ├── test files folder containing all of your VHDL testbench files
    ├── riscv files folder containing your test assembly programs
    └── Proj1_report.pdf
```

Submit the ZIP and report PDF files on Canvas under the "Project Part 1: Single-cycle RISC-V Processor" assignment in a single submission. **Note: Even though the report is included in the ZIP file, you must also submit it as a PDF separately from the ZIP on Canvas in the same submission as shown above in the submission structure.**

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