P2: RISC-V Processor

96/100 Points

10/3/2023

Offline Score: **96/100**

Add Comment

Anonymous Grading: **no**

9/20/2023

Details

ALERT: Changes to the writeup have been made in green

TODO for Spring 2024: A references section with links to all pdfs and references throughout this page (and anything else that's helpful like the prelim ref sheet)

Reminders:

- This is an individual project. Do not show your work to anyone or look at the work of anyone.
- Follow the **Submission Checklist** at the bottom before submitting your project.
- Checking the **components guide (https://canvas.cornell.edu/courses/54972/pages/logisim-component-guide)** is important to understanding how to properly build your RISCV processor. For instance, the op codes for the ALU are **NOT** the same as for Project 1. Try to understand the components before you use them so you can potentially avoid having to backtrack.

Overview

In this project, you'll implement a **single-cycle** processor in Logisim. Your processor will support a small subset of the RISCV instruction set, including arithmetic, logic and memory operations. You'll also write a test file and a small greatest common divisor program, both in RISCV, to ensure your processor's correctness.

Since this project requires you to understand the overall layout and functionality of the processor in depth, we recommend meeting with a TA during your lab section to talk through your design and answer any questions you may have.

Academic Integrity: As one of the most widely studied architectures, RISCV has a wealth of information available on the web and in textbooks. Feel free to use any RISCV documentation you find to help you learn how the instructions work--*but* make sure your design is your own. If you're unsure if it's okay to borrow from a source, ask the TAs. Please cite any significant outside source you use that wasn't provided to you by us. And, of course, make sure your information is accurate! When in doubt, the **official RISCV manual (https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf)** is your best bet. However, for the BEQ instruction, do NOT stick to the immediate format given in this manual; you should just consider the immediate to be insn[31:25] followed by insn[11:7] (don't forget to shift left).

What To Submit

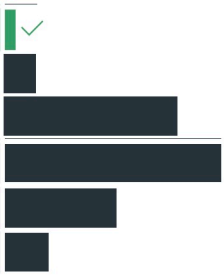
By Tuesday, October 3rd, 2023

- An assembly program containing your greatest common divisor implementation.
- A single Logisim project file (extension: *.circ*) containing your processor and all needed subcomponents.
- A text file containing your well-commented RISCV assembly test program.

Heads up! Last semester, students who filled out the survey reported the following:

How long did you (personally) spend working on this project?

0-4 hours	2 respondents	1 %
5-10 hours	14 respondents	5 %
11-20 hours	91 respondents	31 %
21-30 hours	114 respondents	38 %
31-40 hours	57 respondents	19 %
41+ hours	20 respondents	7 %



Instructions

Your processor must support the following instructions:

Format	Instructions
R-type	ADD, SUB, AND, SLT, SLL, SRA
I-type	ADDI, ANDI, LW, LB
S-type	SW, SB
U-type	LUI
B-type	BEQ

Check the **RISCV manual** (https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf) to see how these instructions are laid out in binary and what they do. Page 104 provides a helpful table for comparing the formats side by side. However, for the BEQ instruction, do NOT stick to the immediate format given in this manual; you should just consider the immediate to be insn[31:25] followed by insn[11:7] (don't forget to shift left).

We won't test your processor with any instructions not in the list above. You do *not* have to implement or account for them in any way - their behavior can be undefined and up to your discretion.

GCD Algorithm

To help you become familiar with RISC-V assembly instructions, you will create a program to compute the greatest common divisor between two numbers. This program will make use of the reduced set of RISC-V instructions that you'll implement later in your processor.

The greatest common divisor function, or GCD, is important in number theory and underlies most modern encryption algorithms. The Greek mathematician Euclid devised an algorithm to compute it more than 2,000 years ago. It relies on the facts that GCD(a, b) = GCD(b, a), GCD(a, b) = GCD(b, a-b) when a ≥ b, and GCD(a, 0) = a. You are provided with the following pseudocode for reference.

```
algorithm gcd(a, b)
        loop while a, b > 0
                if b > a
                        swap a and b
                end
                let c := a - b
                a := b
                b := c
        end
end
```

Input and Output

You may assume that neither a nor b are negative. The arguments a, b are encoded as follows. The word representing a is stored at memory address 0, retrievable using a single “lw” instruction. The word representing b is unaligned where the least significant byte is at memory address 6 (see the diagram below). Your solution is **required** to use four “lb” instructions to construct this argument.

Let b = AA|BB|CC|DD where AA is its most significant byte and DD is its least. Then the following diagram representing memory shows where each byte is stored in memory. The “..” represent irrelevant bytes of memory.

```
Address:        4  5  6  7  8  9  10 11
Memory:         |..|..|DD|CC|BB|AA|..|..|
```

In words,

- The most significant byte is located at memory location 9,
- The next byte is at address 8,
- The next byte is at 7,
- And the least significant byte is at location 6.

Place the greatest common divisor of a and b into register t0.

Suggested Workflow

Work independently on decoding the arguments and implementing the GCD algorithm.

1. Start by writing and testing the correctness of the GCD algorithm with arguments you set yourself.
2. Once decoding is complete, the GCD of the provided arguments in the starter file should yield a familiar number!

Submission

Important: Use only instructions from the reduced RISC-V above.

Warning: ##start and ##expect statements will not have any effect during autograding for GCD, so do not base your submission’s correctness on ##start statements.

Warning: We will both test your submission with multiple inputs and manually inspect your solution, so do not try to hardcode the answer in your assembly.

Please use the provided template in your submission for this problem: **P2-gcd.txt (https://canvas.cornell.edu/courses/54972/files/8897094?wrap=1)** **(https://canvas.cornell.edu/courses/54972/files/8897094/download?download_frd=1)**

Implementation

Logisim Dos and Don'ts

Download the p2 version of Logisim if you haven't already. The link is here: **Electronic Resources (https://canvas.cornell.edu/courses/54972/pages/electronic-resources?wrap=1)**

Build your RISCV processor as a single Logisim circuit file. **Do not** use Logisim's *Project* > *Load Library* > *Logisim Library* command to import circuits from other Logisim files.

Your top-level circuit must be named "RISCV" or "RISCV32" (case-sensitive).

The **CS3410 Components** library folder, documented in the **Component Guide (https://canvas.cornell.edu/courses/54972/pages/logisim-component-guide)**, contains several components you will need such as the register file, RAM, ALU, etc. Make sure to read the **Component Guide (https://canvas.cornell.edu/courses/54972/pages/logisim-component-guide)** before start implementing. You **must** use exactly

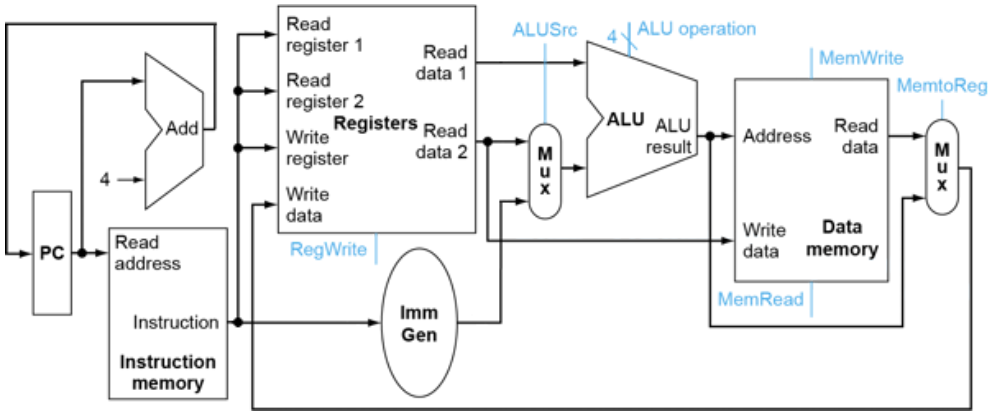
one of each component, excluding the LCD video and Core (*yes, that means only one incrementer)*. **Do not** nest these components any deeper than one subcircuit down.

Feel free to use other regular Logisim components, with the following caveats:

- The only component in the Memory folder that you should use is the register.
- The Arithmetic folder is forbidden.
- Components at the transistor level or below (e.g. transistor, power, ground, transmission gate) are forbidden.
- Constants are forbidden.
- You can use tunnels, but if you do, please use sparingly for the sake of readability.
- Avoid empty inputs/outputs on splitters and plexers.
- You may only use one clock. Subcircuits requiring a clock signal should use input pins to connect to the processor clock. Each cycle should begin on a rising clock edge

We encourage you to also take a peek at the **Not sure how to start?** and **Help and Hints** section near the bottom of this writeup before starting your circuit as you might find it to be incredibly helpful and also answer some initial questions regarding the processor.

Understanding the Processor



(This image is a simplified version of the diagram in section 4.4 of your textbook. It contains all the concepts you need for this project (except branching control logic, which is your task to figure out), and can help you visualize how all these pieces fit together in the big picture. Note that your circuit will look different in terms of exact components, inputs/outputs, etc.)

A CPU or "central processing unit" or just "processor" is a piece of hardware that executes instructions. With every tick of the computer's internal clock, the processor will do the following. Relevant *CS3410* components are italicized.

1. **Fetch** an instruction from main memory. The PC (Program Counter) holds the address of the instruction to fetch. Except in the case of a branch, it should increment by 4 after every cycle, since each instruction takes up 32 bits or 4 bytes. For this project, we simulate the portion of memory dedicated to holding instructions as the *program ROM* component, and the PC is an ordinary 32-bit register component. You will use an *incrementer* to compute PC + 4.
   - Things to consider: How to use a single incrementer to add 4. If you're stuck, think about how binary numbers work.
   - Things to consider: How to update the PC in the case of a branch instruction.
2. **Decode** the instruction into relevant parts (opcode, registers, immediates, etc.) that the processor understands, and read the requested register(s). We will use the *register file* component to simulate the 32 registers.
   - Things to consider: What info is important to extract from an instruction. How to generate the correct immediate from the data provided in the instruction (you will probably want to use a subcircuit for this, as shown in the above image). How to single out bits that differentiate instructions--what makes LW different from SW, or from SB?
3. **Execute** the instruction. The *ALU* and *blackbox comparator* components are useful here.

- Things to consider: What the inputs to the ALU should be. Whether to use the ALU result. How to generate ALU opcodes (hint: in Logisim, *Window > Combinational Analysis* or *Project > Analyze Circuit* can make a circuit if you supply input/output pins and a truth table).

4. **Access memory** (here simulated as the *RAM* component) to load/store data, if necessary.
   - Things to consider: How to get the right selector bits. How to differentiate bytes vs. words. How to simulate the upper bound of the memory.
   - The *RAM* component is one of the trickier components to understand, so thoroughly read through its section in the **components guide (https://canvas.cornell.edu/courses/54972/pages/logisim-component-guide)** for further information.
5. **Write back** a newly computed value to the register file as needed.
   - Things to consider: Whether to write to the register file at all. Which computed value to write (ALU, memory, etc).

We recommend going one step at a time and making sure each part works before moving on to the next. For example, if you think you've got fetch working, try and load a few instructions into the ROM and see if you're able to output each one with every clock tick. Then move on to decode. Test as you go. Use your subcircuits to contain your more complex logic, and do your best to keep your circuits neat!

Not sure how to start?

Consider creating a rough **design document** that serves as a high-level overview of your circuits and logic to reference as you implement and debug your project. You can bring this design doc to office hours and have a TA look it over to see whether you're on the right track. It could include:

- A small description of how you plan to implement each stage of the processor (fetch, decode, execute, memory, write-back) along with diagrams. **We recommend this even if you are not writing a design doc. It is very helpful!**
- Outlines of subcircuits and their functionality, inputs, and outputs.
- Explanations of control logic signals and what values they take on different input combinations. For example, if you use a bit to differentiate R-types from I-types, note down which opcodes would make that bit 1 and which would make it 0.
- Write explanations of anything else in your circuit that isn't easily grasped at first glance, It might help you catch bugs as you implement.

Testing

Write a test program file for Logisim containing RISCV assembly that fully tests all aspects of your processor. This will be a simple .txt file that you can run by loading into the Program ROM. If you have multiple test files, feel free to just concatenate them all together for CMS submissions.

You need to test every instruction that your processor supports. Include both random and edge cases, and clearly distinguish them using comments. For edge cases, please indicate in comments what you are targeting. Your comments can be very brief.

Additionally, don't feel the need to write a program to autogenerate assembly instructions for your random test cases. You certainly *can*, if you would like to, but you can get full points for testing without autogenerating test files.

The RISCV processor doesn't use input and output pins in the same way as the ALU, so you won't be able to test your circuit as a whole using test vectors like you did in P1. Your text file is the only option. However, test vectors may be useful for debugging some of your subcircuits.

This semester, we have added a command line interface that partially exposes the register file state to the end user. It will also give you warnings about whether or not you have things in your circuit that would make it hard or impossible for us to

grade (misnamed circuits, deeply nested register files, etc.), so please pay attention to it. You should use this tool in conjunction with GUI-based testing and development, not as the only source of processor correctness.

You may run a test file against your circuit by using the following:

```
java -jar big-red-logisim-p2.jar --test-risc <path to your test.txt file> <path to your circuit.circ file>
```

Warning: this feature is still highly experimental and has previously been only used internally among course staff. There does not (and likely will not ever) exist a GUI for this feature like for test vectors. If you find bugs or incorrect behavior, please let us know.

We encourage you to build your own semi-automated testing suites using this command-- **Python 3's subprocess module (https://docs.python.org/3/library/subprocess.html)** is a good place to start.

Anatomy of a test file:

Your assembly files will need to be in a fairly particular format, and Logisim will crash if you do not provide it with the required information. You will likely want to develop a suite of multiple test files as only 32 registers can be checked per file.

You will need:

Exactly one of the following:

- ## desc = <test description>: this should be a simple description of your test
- ## cycles = <# of cycles>: the number of cycles to run your program must be specified here. Note that this is different than the online riscv interpreter, which will simply run until a non-user specified instruction is met. Your program will be unceremoniously killed once the cycle limit is reached.

Any number (with at most 1 each per non-x0 register) of the following:

- ## start[x] = <start value>: you may set pre-execution values for any of the 31 non-x0 register file values here. If a register does not have a ## start[x] assignment, it will simply be set to 0. These values are not set via instructions! They just *magically* appear before the first PC value. You likely won't *have to* use any of these-- and note that the interpreter/GUI-internal program ROM does not respect these directives, so omitting them altogether may be easier depending on how your structure your tests.
- ## expect[x] = <end value>: you may specify what the expected values for all applicable registers are using ## expect[x]. If a register does not have a specified expected value, it is expected to be equal to 0 (this is important! Logisim will only accept 0 for such registers!). The end conditions are checked only a single time after the cycle count has been reached.

Example:

```
## desc = smoke test
## cycles = 1
# I can also write comments like this
## start[1] = 0x00000001
## expect[1] = 0x00000002
addi x1, x1, 1
```

Help and Hints

Help. Ed and office hours are your friend! We expect to see most students in OH during the course of the project. Extra hours will be scheduled as needed.

Bugs. If you suspect a bug in Logisim or the CS3410 library, ask the course staff for help.

Read the docs. Refer to the **components guide (https://canvas.cornell.edu/courses/54972/pages/logisim-component-guide)**, the **design guidelines (https://canvas.cornell.edu/courses/54972/pages/logisim-design-guidelines)**, the **tips and tricks**

(https://canvas.cornell.edu/courses/54972/pages/logisim-tips-and-tricks) page, and the **RISCV manual** (https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf) often. Other resources that might be helpful are the **RISCV greencard** (https://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscvcard.pdf) and this **overview of instructions** (https://pages.github.coecis.cornell.edu/ag974/rv8/isa.html). However, for the BEQ instruction, do NOT stick to the immediate format given in the official RISCV manual; you should just consider the immediate to be insn[31:25] followed by insn[11:7] (don't forget to shift left).

Also consider:

- **ECE4750 "Tiny RISC-V" Manual** (https://raw.githubusercontent.com/cbatten/ece4750-tinyrv-isa/master/ece4750-tinyrv-isa.txt)
  - Does not include LB/SB (which you should be familiar with), but is more digestible as a quick reference.

What's up with "simulating the upper bound of memory"?

Logisim does not support RAM components large enough to cover a full 32-bit (4GB) address space. The largest RAM component contains 64MB of data using 24-bit-wide word-addresses. Our tests will mainly rely on memory addresses in the lowest **4MB** of the address space, so your processor should contain *at least* **4MB** of RAM placed at the lowest addresses (this corresponds to a 22-bit-wide byte-address). That is, reads and writes to byte-addresses in the range 0x00000000 to 0x003fffff should work as expected.
Important: Writes to addresses that not backed by any RAM should have no effect, and the address space should not "wrap around" after **4MB**. This is considered as part of your circuit's correctness, as it is never correct for a sw to 0x80000000 to overwrite a value at 0x0000000 (for example). No-op'ing loads/stores to these addresses is likely the easiest way to do this.

Note: your processor does not need to account for non-aligned memory addresses.

RISCV (subset) Assembly Syntax

The RISCV Program ROM component has a built-in assembler that understands all of the instructions you will implement. The syntax is standard RISCV syntax. Labels are case sensitive, everything else ignores case. Anything following a pound ('`#`') is a comment. In this project, you will only use a few of the instructions listed here.

The instruction syntax is the same as given in the RISCV standard. Registers are written as `x0`, `x1`, ..., `x31`, and the destination register goes on the left, followed by source registers and immediate values on the right. Most integer arguments (immediates, shift amounts, jump targets) can be specified in hex (i.e. 0xf28), in decimal (i.e. 264 or -264), or a label.

By default, the first instruction will be placed at address 0, and subsequent instructions are placed at at addresses 4, 8, 12, etc.

Symbolic register names. The assembler built into the RISCV Program ROM accepts standard RISCV register names:`zero, ra, sp, t0-t6, s0-s11, a0-a7, x0-x31`.

Some examples of instructions are:

Immediate Arithmetic	ADDI x12, x0, PC
Register Arithmetic	ADD x13, x0, x20
Immediate Load	LUI x14, 0x123
Shifts	SLLI x13, x13, 2 SLL x15, x14, x3
Memory Load/Store	LW x12, -4(x30) SW x12, 0(x30)

Submission Checklist

Circuit

- Follows every item in Logisim Design Guidelines
- Does not have any forbidden components (e.g. constants, Logisim Adder/Shifter)
- Does not have duplicate components
- CS3410 Components should only appear once in any circuit

Note: Any circuitry that mimics the function of a forbidden or duplicated component will still be penalized.

- Does not use more components than necessary
- No overcomplicated circuitry
- No circuitry with the same functions as a Logisim component
- Does not have splitters with naked wires
- Is readable, using abstractions (subcircuits) appropriately and tunnels sparingly

Testing

- Includes comments that explain how tests were chosen
- Covers all operations
- Covers both corner and random cases

Rough Score Distribution:

- circuit: 67%
- testing: 19%
- GCD assembly code 14%
- This is the score distribution from last year. It may be subject to change.

P1
(https://canvas.cornell.edu/courses/54972/assignments/522715)
P3
(https://canvas.cornell.edu/courses/54972/assignments/522717)

(https://canvas.cornell.edu/courses/54972/modules/items/2052975)
(https://canvas.cornell.edu/courses/54972/modules/items