CSCB58 Lab 4
Part A: A Simple Processor

Introduction

This is the final Logisim lab, in which you will finally create a simple 8-bit processor, capable of executing real machine-code.

You'll be building the datapath as well as the control unit for this basic processor. A processor's "data-path includes all of the circuits that store and manipulate the data a program uses, and the control unit stores and manipulates the instructions in the program. Those instructions are used by the control unit to tell the datapath what to do.

To complete the components of the datapath, you will construct the arithmetic logic unit (ALU). This ALU will support two operations of your choosing, that can be used to manipulate data in programs. Then, you will construct the instruction decoder, part of the control unit. Finally, you will hook up the ALU and instruction decoder blocks you just built to a register file, and test the resulting processor by manually feeding it instructions in our custom machine code.

You are provided a starter file `lab04_base.circ` which can be downloaded from Quercus.

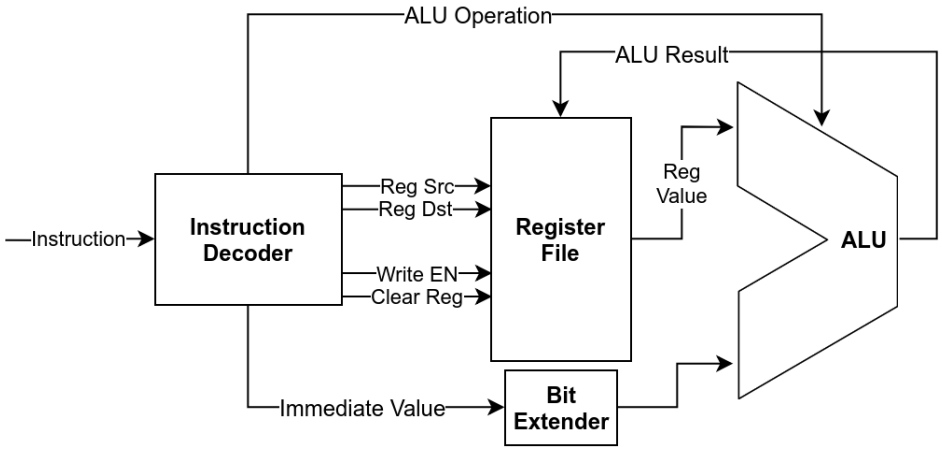
[TASK] Download the starter file and use it as the base for your solution.

This starter circuit contains an implementation of a register file from the previous week with an additional clear input. It also has "skeletons" (the input and output pins) of the sub-circuits you need to build.

Notes: For this lab, you may any Logisim component from **Wiring**, **Plexers**, **Gates**, **Arithmetic**, and **Input/Output** blocks. You may also use **Registers**, from the **Memory** block. You can also use the circuit analyzer (although it's not hugely helpful here).

Processor Structure

Below is a schematic of the processor you will implement. Note this processor is simply enough that all instructions can be done in exactly one clock cycle, and there is no need for an entire control unit FSM. Instead, we will use a combinatorial instruction decoder to determine the control signals.



- The instruction decoder, one of the blocks you will be constructing, is a combinational circuit that decodes an 8-bit instructions fed as input, and outputs the correct values on the control signals (ALU Operation, Reg Src, Reg Clear, etc.) to make the operation happen.
- The register file in the diagram is similar to the one you saw in the lectures, with one read port and one write port. It has four 8-bit registers. The register file accepts an 8-bit data input, two 2-bit select inputs to select which register to read from ("read_reg") and write to ("write_reg"), the clock, a write-enable input, and a reset input. Use Logisim registers to implement it.

  The one difference from what we saw in class: this register file features a Clear Register input ("clear"): if it is high on the positive edge of the clock and if write_enable is high, the register file will ignore "data_write" and will instead write the value 0 to the register.

  [TASK] Implement the register file inside the "register_file" subcircuit using the input and output pins already there.
- The ALU receives two 8-bit inputs, and a 1-bit input to control what operation it runs on those inputs. The output is sent back into the register file. There are no output flags.
- Since the immediate value is 2-bit, and the ALU accepts 8-bit inputs, it needs to be **zero-extended** to 8-bit. You can use the Logisim Bit Extender for this (found **Wiring** in the Design panel), but make sure to configure the extension type to be zero extension, not sign extension.

ALU Sub-Circuit

Given two 8-bit inputs A and B, your ALU sub-circuit will perform two operations of your choosing, selected using a 1-bit control input "op". You do not need to implement these operations yourself; instead, you will select from the built-in **Arithmetic** blocks in Logisim-Evolution. Make sure to set the width of the operations to 8-bits to match the inputs and the output "data_out". Once again, you do not need to output any flags.

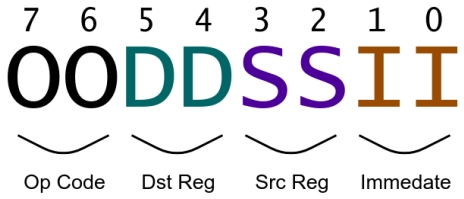
[TASK] Implement the ALU in the "alu" subcircuit using the input and output pins already there.

Some historical context

Some early processors contained a huge number of special purpose circuits. Since then, we have scaled back to a more limited set of functionality, but even today, most ALUs can compute dozens of different functions. It turns out that it is CPUs that support a limited set of instructions can be made to run faster and more efficiently. Complex functionality can then be implemented in software. In fact, modern CPUs support more assembly instructions then actually implemented by the ALU. Instead, some instructions are performed by running using multiple ALU steps over several cycles.

Instruction Decoder Sub-Circuit

The decoder receives an 8-bit input with the instruction. It outputs the 2-bit destination register index, 2-bit source register index, 2-bit immediate value, ALU operation (1-bit), write enable (1-bit), and register clear (1-bit).



For this processor, there is only one type of instruction, in the format shown above. Each instruction contains a 2-bit operation code (op-code), a destination register (indexed from 0 to 3), a source register, and a 2-bit immediate value. **This instruction set has been chosen to require very little logic to decode.**

The two ALU operations are operations of **your** choosing. Remember, the ALU's input consists of the register file's output and the immediate value from the instruction, and the output is fed into the register file. It is up to you to determine what the control lines should be to execute these four instructions. The table below shows the specific meaning of each op-code. An example instruction is `01110111` which performs ALU Operation A on the value in register 1 and the immediate value 3, and stores the result in register 3

Opcode	Operation	Description
00	No-Op	do nothing
01	ALU Operation A	Dst reg ← src reg (OP-A) immediate
10	ALU Operation B	Dst reg ← src reg (OP-B) immediate
11	Clear Reg	Dst reg ← 0

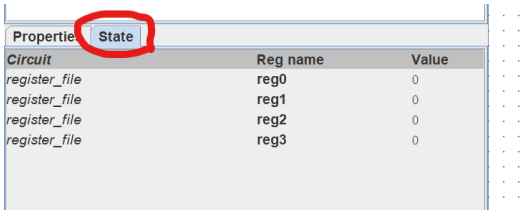
[TASK] Build the decoder unit in the "instr_decoder" subcircuit using the input and output pins already there.

When implementing, you have a choice of either using a truth-table, using the circuit analyzer, or even just looking at the table and figuring things out on your own. It is entirely your choice. **However this instruction set has been chosen to require very little logic to decode, and you may be marked down a little if the decoder is overly complicated** (example too many or redundant logic gates).

Wiring it together

Now that you have an ALU, a register file, and an instruction decoder, creating the processor in the schematic on the first page should be straightforward. Create a "main" circuit, and wire all of the sub-circuits together according to the schematic. Use Logisim buttons for the CLK and Reset. You may find the **Bit Extender** component (in Wiring) useful in wiring up the immediate to the ALU. You should choose zero extension.

Once you've done this, test a few instructions manually by editing the instruction input and running a clock cycle. To quickly inspect the state of the registers, beside the `Properties` tab, there is a `Registers` tab, which lists the four registers and their values at all times. See figure to the right.



[TASK] Create a "main" circuit, wire up the processor, test it, and upload to Quercus.

Summary of tasks

1. Implement the register file, decoder, and ALU in their respective subcircuits.
2. Create the main circuit and wire everything up.
3. Demonstrate your processor to your TA.

Part B: Intro to Assembly

Introduction

From this week, we are leaving Logisim behind and starting something new – programming in assembly. We will be learning about a specific architecture called MIPS. MIPS is a RISC (Reduced Instruction Set Computer) family of processors originally introduced in the early 1980's. The MIPS assembly language is well-known for being concise and logically structured, and because of their simplicity and energy efficiency, MIPS processors are still in use as embedded processors in devices like cell phones and portable game players.

Since the computers we use today are not MIPS machines, we will need to simulate one. The simulator we are using is called MARS, and it is written in JAVA. You can easily download it from the MARS web page: http://courses.missouristate.edu/kenvollmar/mars/download.htm.

Note some students encountered a bug that sometimes causes MARS to freeze when debugging code. You can download a fixed version from Quercus: https://q.utoronto.ca/courses/291126/files/25051677?module_item_id=4517621.

Additionally, some students last year reported issues with MARS not finding .asm files on Mac. Try opening the jar with CLI:

```
java -jar path-to-jar
```

Note: Take your time on this lab and future lab to get used to MARS, since we'll be using it in all of the remaining labs as well as the project. Play with the code, and also run the lecture and review code available on Quercus. As always, you will need to submit your solution code (as `.s` or `.asm` files) to Quercus before the start of your practical session.

An Assembly Program

Every assembly program we write will look very similar. Here is an outline of a typical program:

```
# Anything after # is a comment. Document your code!
# Write your name and UTorID at the top of your lab submissions!

.text   # This tells the assembler that the following are instructions.

.globl main
main:   # This is a label ()which is simply a name for an address).
        # The main label tells the OS where to start execution.

        # Write your main program code here...

        # This tells the OS when your program ends.
        li $v0, 10  # Uses system call 10 to exit program
        syscall     # More on this in future weeks

# Write functions here...
```

Every program is separated into two parts: a data section and a code section indicated by `.text`. We will learn about the data section in future weeks. Your code should be placed in the main block. The `main` label specifies where the program's main function starts (where MARS should start executing code). You may create other labels as you like; each label is used to name a specific line of code so that you can branch or jump to it. We'll use labels a lot when we implement control flow like branches, loops, and function calls.

The keyword *syscall* asks the operating system (or the simulator, in this case) to intervene to run a privileged instruction. We will learn about it in future weeks. Here we are using it to tell the OS to end the program

Note: Make sure to always document your assembly code using comments.

MIPS Reference

Before you become an assembly programming guru, you almost always need a reference card at hand, since some instructions implicitly work with specific registers (e.g., *syscall* implicitly uses *$v0* and *$a0*); and some register values are assigned to specific operations (like different system calls), which we don't want to memorize.

Download the **MIPS reference card from Qerucus** – it has the basic information that you need while programming MIPS assembly. You will find the reference card, as well as other useful Assembly resources such as detailed MIPS assembly references, here: https://q.utoronto.ca/courses/291126/pages/mips-assembly-resources .

MARS Basics

Download one of the lecture MIPS assembly programs from Quercus/OneDrive. Load the file into MARS and then "Assemble" it (under the "Run" menu, on the toolbar, or using the F3 key). You cannot run your code until it assembles correctly. To test this, add a few random characters to the code in the "Edit" window, then try to assemble again. You will see an error message printed in the "MARS Messages" window at the bottom of the screen.

Now, take a moment to familiarize yourself with the layout of MARS. Fix the code, and then re-assemble it. You'll be taken to an "Execute" series of windows.

- The top left window contains the text segment – the code in your assembly program. That window provides you with the addresses of the various instructions, the machine-code value that is stored at that address, the assembly equivalent, and finally the line of code in the source file that generated that assembly instruction. You'll see that some lines of source code generate multiple lines of assembled code. In some cases, the original assembly instruction is a pseudo-instruction. In other cases, extra operations are required because of the simulated machine's architecture (hardware).
- The middle left window (above the message window) contains a window into memory. This will be more useful in future weeks when we start defining variables. Initially, the memory window is set to shopw the data segment – the variables defined in your assembly program. However, the pulldown bar lets you select other segments including the heap or stack. Note that you can also scroll through memory and select how the values are interpreted. ASCII mode is particularly useful for checking strings.

- The pane on the right contains information about all of the registers in the machine. By default, it shows the registers of the processor: register file, lo, hi, and PC. "Coproc 1" (co-processor 1) is the floating point unit, and we will not use it in this course. "Coproc 2" supports the execution of interrupts, which we may briefly touch on in future weeks.

Now, step through the code line by line. You can also execute the entire program, if you just wish to see the result, or step backward, if you wish to investigate a particular instruction more carefully. Take a few moments to familiarize yourself with how MARS uses highlighting to indicate the currently executing instruction and the registers that are being accessed. You can reset the simulator through the "Run" menu or by clicking on the "rewind" button in the toolbar.

Compute the Number of Solutions to a Quadratic Equation

Let a, b, c and x be some integers. Consider the quadratic equation: $ax^2 + bx + c = 0$. The number of real solutions for this equation can be found using the discriminant Δ:

$$\Delta = b^2 - 4ac \, .$$

If $\Delta < 0$ there are zero real solutions for x, if $\Delta = 0$ there is one solution, if and $\Delta > 0$ there are two.

Write an assembly program that receives the coefficients a, b, c in registers `$t5`, `$t6`, `$t7` (respectively) and computes the number of solutions. At the beginning of your program, initialize these registers to some values. When your program ends, `$t0` should contain the number of solutions: 0, 1, or 2. You can assume results fit in 32 bits. **Test your program well!**

Note it is fine to overwrite the registers we used for input (or output) during the run of your program, as long as at the beginning you take it from the right registers, and the end the result is stored in the right register.

[TASK] Submit a file `lab04a.asm` that implements the above. Make sure your code is well-documented.

[TASK] Write your name and UTorID at the top of the file as a comment!

Use Euclid's Algorithm to Compute Greatest Common Divisor

Let a and be b be two positive, non-zero integers. Their *greatest common divisor* is the largest positive, nonzero integer that divides both a and b without leaving a remainder. For example if $a = 46$ and $b = 20$ their greatest common divisor is $gcd(46, 20) = 2$ since $46/2 = 23$ and $20/2 = 10$ but no larger integer divides both. Other examples: $gcd(60, 15) = 15$, $gcd(16, 60) = 4$, $gcd(60, 48) = 12$, $gcd(25, 64) = 1$.

Euclid provided a simple iterative algorithm to compute the GCD, given here in Python:

```
while a != b:
    if a > b:
        a = a - b
    else:
        b = b - a
```

At the end of the run, a will contain the GCD.

Implement Euclid's algorithm in assembly. Your program will first initialize `$t8` and `$t9` to two positive numbers, and then implement the algorithm above. When your program ends, the GCD of `$t8` and `$t9` should be stored in `$t0`.

[TASK] Submit a file `lab04b.asm` that implements the above. Make sure your code is well-documented.

[TASK] Write your name and UTorID at the top of the file as a comment!

Hints:

- First test the algorithm in a high level language (e.g., Python) so you can understand how it works.
- You will need a loop, and also some conditionals. This means several branches, jumps, and labels.

Summary of tasks

1. Write a program `lab04a.asm` that computes the number of solutions to a quadratic equation.
2. Write a program `lab04b.asm` that computes the GCD using Euclid's algorithm.
3. Make sure to document your assembly code using comments, and to write your name at the top.
4. Submit the programs to Quercus as a zip file. **Do so before the deadline!**
5. Demonstrate your solutions to your TA.

Evaluation

As always, marks are based not only on submitted work but also on oral examination by TA. You need to be able to make reasonable explanations about any details to the TA.

Solution	Submitting everything on time	1 mark
	Decoder	1 mark
	Register file	1 mark
	ALU	1 mark
	main circuit and wireup	1 mark
	Code documentation (comments)	1 mark
	Number of solutions	2 marks
	GCD	1 mark
Understanding	TA oral score	1 to 4

Final lab marks (up to 9) are determined by multiplying your solution subtotal by the oral score (1–4) and dividing by 4:

$$\text{total} = \text{solution marks} \times \frac{\text{oral score}}{4}$$