Project Checkpoint 2 Specification

*Many thanks to Dr. Prateek Bhakta, who designed the original version of this project. Credit for anything that works belongs to him; blame for any mistakes belongs to Dr. Dollak.*

In the second part of your project, you will begin to design a processor in Logisim that will process the binary encoded instructions that are output by your assembler from part 1. You should save your work in a Logisim save file called Project2.circ.

For part 2 of the project, you need to implement a register file, an ALU, and a control circuit.

Your solution should correctly handle the following instructions:

* add, addi, sub
* mult, div, mflo, mfhi
* sll, srl
* slt

## Register File

A register file is a collection of registers, multiplexers, demultiplexers, and logic gates that facilitate reading from and writing to individual registers. A register file stores the contents of all the registers in a CPU, and allows you to read the value of two registers (per clock cycle) and write to one register (per clock cycle). It contains 31 real registers numbered 1-31 and one simulated register 0, which is hardcoded to always output 0 and is not a register that can be written to.

Inputs:

* 32 bit value rd-data
* 5 bit rs-select
* 5 bit rt-select
* 5 bit rd-select
* 1 bit write-enable
* 1 bit clock (implicit input, not one that you should add in logisim as an input)

Outputs:

* 32 bit value rs-data
* 32 bit value rt-data

Internal State:

* 31 or 32 32-bit registers

Functionality:

The output on rs-data always corresponds to the 32 bit data stored in the register #(rs-select). If #(rs-select) is 0, then output 0 always.

The output on rt-data always corresponds to the 32 bit data stored in the register #(rt-select). If #(rt-select) is 0, then output 0 always.

If write-enable is 1 when the clock rises from 0 to 1, store the value of rd-data into register #(rd-select). If #(rd-select) is 0, then do not write to any register, even if write-enable is 1.

## ALU

An **arithmetic logic unit**, or ALU, takes two 32-bit inputs and outputs the result of some arithmetic computation on those inputs (addition, subtraction, multiplication, division, shift, comparison, equality, movefromhi, movefromlo). All these computations are done simultaneously, and one or more multiplexors controlled by a signal from the control determine what the output will be.

Inputs:

* 32-bit value IN1 (from registers or control)
* 32-bit value IN2 (from registers or control)
* Signal(s) from Control

Outputs:

* 32-bit value OUT

Internal State:

* One 32-bit register “LO”
* One 32-bit register “HI”

Functionality:

Outputs the desired arithmetic operation specified in the signals from Control on the operands IN1, IN2. You get to chose how many signal wires you want into the ALU/ how to interpret them.

On a multiply/divide instruction, store the results of the computation inside the ALU internal registers “HI/LO”.

On a “mflo” instruction, the output from the ALU should be the value of the ALU’s internal register “LO”. On a “mfhi” instruction, the output from the ALU should be the value of the ALU’s internal register “HI”.

## Control

Inputs:

* A 32-bit instruction

Outputs:

* Signal / selection wires to the register file/ALU/Multiplexors that controls how data will “flow” and which registers, if any, will get written to.

## Abridged Truth Table

A good analogy for how to think about the processor is that the data is “flowing” through the channels/wires you’ve created for it, and the multiplexors act like “switches” that control that flow in situations where there are multiple inputs/choices for a wire. Once you have set up the processor outside of the control box, all that’s left is to appropriately control those various switches.

To help plan your control box, you should start by writing a *partial* truth table for this control box. The input is the 32-bit instruction, and the output is the various multiplexor/demultiplexer switch signals coming out of the control. A true truth table would have rows, which is unreasonable, so instead we can build an abdridged truth table. Here we have one row per instruction type. Most of the control signals can be determined from the opcode alone (The ALU select signal needs the function code as well). The rest of the instruction is data that can be directed to the various outputs with multiplexors.

I have provided a starter file for your abridged truth table with a few lines filled in. Note that the grey rows are instructions that you do not have to handle for this checkpoint; however, you may find it helpful to consider them now, since you will need to handle them in the next checkpoint.

The numerical code you use for ALU select is up to you. You simply need your Control to provide a code that your ALU interprets correctly. You will want to refer to your partial abridged truth table as a reference when you are building your circuits to make sure that happens.

## Tips

This is a collection of tips from previous iterations of this class.

* You do NOT need to optimize for circuit size, gate delay, or anything else in the control unit. If it works, it works.
* You are FAR MORE LIKELY to get the control unit to work and sucessfully debug issues with a manageable size, well organized circuit layout.
* One way to approach the control is to start by using appropriate AND gates to make a set of wires, one per instruction, so that on an "add" instruction, the "add" wire becomes 1 and all others are 0, on an "addi" instruction, the "addi" wire becomes 1 and all others are 0, etc.
* In most situations, you do not need to ensure that the output of a wire is when it is unused. The rest of the circuit will just ignore it.
* Remember to do a “signed” extension of the immediate from 16 bits to 32 bits when used as an input for the ALU, so that negatives are properly extended.

## Order

This is the recommended order in which you do this project.

* Start by building a register file. It's a bit tedious, but you know everything you need to do it.
* Make your abridged truth table. Do not skip this step. For best results, also include future instructions (e.g., branches and jumps) in your truth table, so that you don’t have to change your control box as much for part 3 of the project.
* Get your general datapath set up (connect the boxes like in the in-class diagram) and connect all wires to appropriate multiplexers, etc.
* Using your truth table, make the control for the project, using the various instruction wires to act as multiplexers for data when needed. Do the ALU codes last, and try to connect the ALU.
* Making the ALU is pretty straightforward except for the HI and LO registers. Other than that, it’s mostly just built-in arithmetic boxes and one large multiplexer. The order in which you attach each output to the multiplexer depends on the ALU code you made. (ex. 0 for add, 1 for sub, etc.). It helps if the ALU code for add is 0, because many operations use add in the ALU (branches, lw, sw, etc.)

## Testing

You should test your circuit using the 32-bit instruction input into the control. Then simulate a clock cycle by pressing “Manual Tick Full Cycle” or F9, and ensure that the values of the registers are correctly updating according to the rule specified on the instruction. Make sure you test all required instructions thoroughly.

Make a spreadsheet of testcases. For each testcase, note any state that should be set before running the testcase, the assembly instruction, the corresponding binary instruction, and the expected result of running the test. Submit your spreadsheet on Blackboard.

Hint: Use the .asm and .txt files from your Checkpoint 1 testcases to get assembly and corresponding binary.

# Challenges

**Challenge Requirements for Project Checkpoint 2:**

Teams must complete at least 8 stars' worth of challenges for this checkpoint.

**Available Challenges for Project Checkpoint 2:**

You can earn a maximum of 26 stars on this assignment.

2 stars:

* Implement the following additional instructions in your CPU: and, or, nor, xor, andi, ori, xori, lui. 2 stars per instruction.

5 stars:

* Implement clo and clz in your CPU. You will have to add custom made circuits that compute these functions to your ALU. 5 stars per instruction.

# What to submit

Submit Project2.circ, your abridged truth table, and your testcases spreadsheet on Blackboard.

Only one person per team needs to submit on Blackboard.

# Interview

After you submit part 2 of your project, schedule an interview with me to discuss how your project works, what design choices you made and why, etc. I’ll also give my comments to you about your project and make recommendations that you should change/fix before part 3 of the project.

I will assign your project checkpoint grade (incomplete, retry, or achieved) based on the interview. To get an “achieved”, you need to show me that your simplified CPU works as described in this document *and* that you fully understand how it works.