Project Checkpoint 3 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.*

For part 3 of the project, you need to add instruction memory and data memory to your project, together with the additional circuitry/multiplexors/control signals needed to hook it all together. The diagram from the lecture notes should give you a good guide to how the datapath should be connected.

You will need to modify the controller from part 2 to handle the new instructions and send control signals to the appropriate multiplexors.

Your solution should add functionality for the following instructions:

* lw, sw
* beq, bne
* j, jal, jr, jalr
* syscall

## Data Memory

Logisim provides a random-access memory (RAM) that can be both read from and written to, like your register file. However, you can only read OR write to one address at a time.

For lw instructions, the appropriate value in memory should be accessed. For sw instructions, the appropriate value in memory should be written to. Please play around with the logisim RAM module to see what it can do.

You should set the memory module settings to:

* Address width: 16
* Data width: 32
* Allow misaligned: no
* Enables: use line enables
* Data Bus: Separate Data bus

You will need to downshift the 32-bit byte-addressed address that your MIPS process works with to a 16-bit word-addressed address that your logisim RAM module uses. This requires the appropriate combination of bit shifting and extending to make this happen (the order matters!).

## Instruction Memory and Program Counter

The instruction memory block will serve as read-only memory (ROM) that stores the instructions for your CPU, encoded in MIPS format.

The Program Counter (PC) is a register that stores the current instruction number. This counter serves as the address for the instruction memory. In the absence of jumps or branches, the value of PC becomes (PC + 4) at the beginning of every clock cycle.

For jumps and jal, the next value of PC should be the address specified in the lower 26 bits of the instruction. jal instructions also cause the value of PC + 4 to be stored in register , and jalr instructions cause the value of PC+4 to be stored in register specified by the instruction. For jump register instructions, one of the outputs of the register file needs to replace the current value of the PC.

For branches, the next value of the PC should be the address of (PC + 4 + offset). Keep in mind that the 16 bit offset is word addressed, while 32 bit PC is byte-addressed. Use a signed extender and shifter appropriately.

You should set the ROM memory modules settings to:

* Address width: 16
* Data width: 32
* Allow misaligned: no

The input 32-bit instruction to the control should now come from the instruction memory output.

## Syscall

Syscall should function identically to jalr $0, $k0 (go to , store PC+4 into ). The syscall encoding that I chose should make this easy for you. This will be necessary for when we implement syscall in project part 5.

## More Control

Naturally, you will have to add to your control unit to handle the extra signals to multiplexors and extra instruction types. Some of these instructions will also need to set the write\_enable signal to the register file (lw, jal, etc.), while others won’t (sw, jr, etc.).

Inputs:

* A 32 bit Instruction

Outputs:

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

## Tips

* Make a copy of your Project2.circ file *before* you start changing things for this checkpoint.
* You will have to switch between things that are byte adressed (immediates in lw/sw, addresses in registers, PC) and things that are word addressed (immediates in jumps/branches, the logisim addresses to RAM/ROM). Make sure you keep track of which is which when testing.

## Testing

You should test your output by taking MIPS assembly code, using your Project part 1 assembler to translate it into binary, and then loading the instructions into the ROM instruction memory, and loading the static memory (if there is any) into the RAM memory.

For now (before we actually implement syscall), you may want to add an "end" block at the end of your assembly like:

end:  
 j end

The above loop can simulate the end of your program for Logisim, and you can compare the value of your registers at the end of your program with whatever QTSPIM would output.

I recommend testing at least a program that uses every instruction, some recursive function, like a Fibonacci implementation, and something that uses static memory. Since we haven’t implemented syscall yet, we can’t test anything with syscall, like heap allocation, but you can still do stack allocation.

# Interview

After project part 3 is submitted, you (and whoever you worked with) will have an interview with me, so that we can determine the state of your project. I will mostly be checking to make sure that

* You have finished the project correctly.
* If not, what steps you still need to do to get parts 1-3 working before starting part 4.
* All members of the group *understand* and contributed to the submitted work. If I find that all members of the group do not understand or did not contribute to the work, this will likely result in individual meetings and adjusted grades.

# Challenges

**This checkpoint has no new challenges**. Checkpoints 4 and 5 have many, many stars' worth of challenges to look forward to.

# What to Submit

* Project3.circ
* Update your abridged truth table for the control box that you are submitting for part 3.

Only one person per team needs to submit on Blackboard.