Project Checkpoint 4 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 fourth part of your project, you will add memory-mapped I/O functionality to your processor in Logisim through a central I/O bus. You will also add *controller* circuits for the keyboard, monitor, terminal, and memory that connect these peripherals to the bus.

# Setting up the bus

For part 4 of the project, you need to change your current configuration from one that interacts directly with data memory to one that interacts with a bus. This bus connects memory, I/O registers, and peripherals like your keyboard and monitor together.

The blocks visible in the “main” layer for your main project should be:

* CPU
* Memory
* LogisimKeyboard
* KeyboardController
* LogisimRGBMonitor
* RGBController
* LogisimTTYTerminal
* TerminalController

Each of the processor, memory, keyboard controller, and terminal controller are connected to the I/O Bus, which will be in the main circuit. The bus has three wires.

* 26-bit Address
* 32-bit Data
* 1-bit read/write

## BusWriteEnable

An important part of this setup is the tristate buffer, which controls which part of the circuit is “writing” to the bus at any given time. See the first part of Section 6.5 of the zyBook for more on how this circuit works. The given Logisim file called BusLayout.circ shows how everything should be connected.

The BusWriteEnable signal coming out of a box is a signal that comes from the device to the tristate buffer to get it to write to the bus. Multiple devices should **not** be writing to the bus at the same time.

On a sw instruction (CPU writes to bus), the CPU should be the only entity that writes to the data bus.

On a lw instruction (CPU reads from bus), it is up to appropriate device for that address to write to the bus.

## Processor

The processor is the circuit that you made in parts 2-3 of the project with two modifications:

1. Instead of being the main circuit, it will instead be a subcircuit.
2. Instead of including the RAM and accessing memory directly, it will use input and output pins and connect to the bus in main as shown in the bus layout file. Essentially, you delete memory from your project part 3, and then you connect the loose wires to the bus through input/output pins.

The processor should have inputs:

* DataFromBus (32 bits)

and have outputs:

* Address (26 bits)
* DataToBus (32 bits)
* WriteEnableMem (1 bit)
* WriteToBus (1 bit)

## CS 301 Memory Map

For this assignment, the address ranges we will use are NOT the standard MIPS memory ranges, but are chosen to make your experience with Logisim easier. Firstly, unlike real MIPS, the logisim memory module can support up to 24-bit *word*-addressed memory, so any *byte* address meant to go to memory will never go above 26-bits (0x4000000).

For our project, addresses strictly less than are meant for the RAM. Addresses to (256 bytes) are meant for I/O.

Your static memory will go from to however much static memory you allocate. The heap comes IMMEDIATELY after that, and grows upward.

We will allocate addresses for our reserved "OS" memory in part 5. The stack pointer will start at and grow downward.

We will make use of the fact that the highest 6 bits of the address are truncated when doing some of our math. Because the higher order bits are removed, the CPU/Assembly can treat addresses like as equivalent to , which is , and the rest of the circuit won’t know the difference.

## Initialization

Your CPU will need to do some initialization to make things work properly. When you load/reset Logisim, the stack pointer register $sp will have a default value of 0. This means when you first lower the stack pointer, it will start storing information at , which is in our reserved I/O range after truncation ()

You must initialize $sp to / / at the beginning of your programs so that this doesn’t happen.

For now, just remember to include a line like at the beginning of each program that you use for testing in project part 4. We will do a more robust "boot up" process in part 5 that includes this line.

## The Terminal

First, play around with the TTY terminal in Logisim. You will need to attach a write enable, a 7-bit data field, and a clock. Make sure you can write characters to the terminal, one per clock tick.

Your terminal controller will be the simplest controller to write, so you should start here. Most of the wires in this controller will simply be direct connections between the bus and memory.

It will have inputs:

From bus:

* AddressFromBus (26 bits)
* DataFromBus (32 bits)
* WriteEnableFromBus (1 bit)

To the Terminal:

* DataToTerminal (7 bits)
* WriteToTerminal (1 bits)

For the terminal, we will use address .

The behavior is:

* If the Address on the bus is 0x3FFFF00 and WriteEnableFromBus is 1,
  + write the lower 7 bits of DataFromBus to the terminal.
* If the address on the bus is **not** 0x3FFFF00 and/or WriteEnableFromBus is 0, do nothing.

This should only require a comparator and an extender and a few logic gates.

## Main Memory

Your memory module controller is the second simplest controller to write.

It will have inputs:

From bus:

* AddressFromBus (26 bits)
* DataFromBus (32 bits)
* WriteEnableFromBus (1 bit)

From memory:

* DataFromMem (32 bits)

and have outputs:

To Bus:

* DataToBus (32 bits)
* WriteToBus (1 bit)

To Memory:

* AddressToMem (24 bits)
* DataToMem (32 bits)
* WriteEnableToMem (1 bit)

Memory should NOT be written to and you should NOT write to the bus if the address being requested is in the I/O range (i.e. 0x3FFFF00 or greater).

To summarize:

* If the Address on the bus is less than 0x3FFFF00:
  + If WriteEnableFromBus is 1 (sw), then write the value in DataFromBus to memory in address AddressFromBus.
  + If write enable is 0 (lw): then write the value in memory in address AddressFromBus to DataToBus
* Otherwise, do nothing

This should only require a comparator, shifter, extender, and a few extra basic logic gates.

## KeyboardController

Your KeyboardController should have the same connections to the bus as memory, as well as another series of inputs to connect to the keyboard in Logisim. The keyboard itself should be in the main circuit of your project, and the controller should be a self-contained box connected to the bus.

This is the third easiest controller to write.

It will have inputs:

From bus:

* AddressFromBus (26 bits)
* DataFromBus (32 bits)
* WriteEnableFromBus (1 bit)

From keyboard:

* keyboardStatus (1 bit)
* keyboardData (7 bits)

and have outputs:

To Bus:

* DataToBus (32 bits)
* WriteToBus (1 bit)

To Keyboard:

* KeyboardRead (1 bit)

The keyboard controller doesn’t need internal registers, because the Logisim keyboard stores state for you. The keyboard is allocated two memory addresses, . If your assembly code tries to read/write from this address, it should look from the perspective of the processor that it was loading/storing from memory like any other address.

However, what your controller will actually do is:

* If the Address on the bus is
  + If BusWriteEnable is 1 (sw), send the “KeyboardRead” signal to the keyboard to get the next character. The value in DataFromBus is ignored.
  + If WriteEnable is 0 (lw): put the value coming from the keyboard status (0 or 1) on the data bus.
* Else If the Address on the bus is
  + If write enable is 0 (lw): put the value in the keyboard data signal on the data bus.
* Else: Do nothing

From the perspective of your CPU/Assembly, to test your circuit, the assembly code protocol to access the keyboard looks like this:

* If a lw on address 0x3FFFF10 / 0xFFFFFF10 / -240 gives you 0: no new keypress
* Else:
  + do a lw on address 0x3FFFF14 / 0xFFFFFF14 / -236 to get the value of the keypress
  + do a sw on address 0x3FFFF10 / 0xFFFFFF10 / -240 to increment to the next keypress (if there is one).

## RGBController

Your RGB MonitorController should have the same connections to the bus as memory, as well as another series of inputs to connect to the RGBmonitor in Logisim. The monitor itself should be in the main circuit of your project, and the controller should be a self contained box connected to the bus. The monitor controller is allocated addresses .

The monitorController needs internal registers because sending pixels requires multiple instructions (and clock cycles).

It will have inputs:

From bus:

* AddressFromBus (26 bits)
* DataFromBus (32 bits)
* WriteEnableFromBus (1 bit)

and have outputs:

To RGB Monitor:

* RGBx (8 bit)
* RGBy (8 bit)
* RBGColor (24 bit)
* RGBWrite (1 bit)

You need three internal registers:

1 internal register to store the x coordinates of the pixel to write,

1 internal register to store the y coordinates of the pixel to write,

1 internal register to store the RGB color of the pixel.

These registers should be able to be written to by the bus, and also send their values as output to the RGB monitor (with appropriate bit extenders).

In summary, your RGBcontroller does:

* If the Address on the bus is 0x3FFFF20
  + If write enable is 1 (sw), store the value on the bus in the internal X coordinate register.
* If the Address on the bus is 0x3FFFF24
  + If write enable is 1 (sw), store the value on the bus in the internal Y coordinate register.
* If the Address on the bus is 0x3FFFF28
  + If write enable is 1 (sw), store the value on the bus in the internal color register.
* If the Address on the bus is 0x3FFFF2C
  + If write enable is 1 (sw), send the write enable signal to the monitor for one cycle. The value on the bus is ignored.
* Otherwise do nothing

And to draw a pixel, your assembly program can:

* Write (sw) the x coordinate of a new pixel to display to 0x3FFFFF20 / 0xFFFFFF20 / -224
* Write (sw) the y coordinate of a new pixel to display to 0x3FFFFF24 / 0xFFFFFF24 / -220
* Write (sw) the RGB color of a new pixel to display to 0x3FFFFF28 / 0xFFFFFF28 / -216
* Write (sw) anything to 0x3FFFFF2C / 0xFFFFFF2C / -212 to draw the pixel

# Testing

You should test each of your terminal / keyboard / monitor controllers by writing a small assembly program that tests each one separately, using lw and sw to the appropriate addresses to interact with each device.

I will provide a few test cases. It is up to you to write smaller tests as part of the debugging and testing process.

# Challenges

**Challenge Requirements for Project Checkpoint 4:**

Teams must complete at least 9 stars' worth of challenges for this checkpoint.

**Available Challenges for Project Checkpoint 4:**

You can earn a maximum of 49 stars on this assignment.

Across checkpoint 4 and checkpoint 5, you can earn 5 stars *each* for attaching sound, joysticks, LED panels, or any other widget from the Input/Output menu of Logisim to the bus with an appropriate controller and appropriate assembly device driver. In part 4, you write the controller to attach the device (3 stars), and in part 5, you write a device driver in assembly to communicate with that device (2 stars). You can earn a maximum of 40 stars this way across the two checkpoints (24 stars here and 16 on checkpoint 5).

You can earn up to 25 stars for adding an on-chip cache. If you choose to attempt this, you should submit *two* versions of your .circ file: one with the cache and one without. Build a subcircuit for your cache and incorporate it in your CPU circuit. Your CPU should check your cache first for any memory address it needs. If it is already in the cache, get the data from the cache, not from main memory. You may implement direct-mapped, fully-associative, or partially-associative caching. You may choose either write-through or write-back, but your cache must use *some* strategy to ensure that main memory is (eventually) updated.

# What to Submit

* Project4.circ
* Your test cases as .asm files.
* (If you decide to go for the caching stars) Project4Cache.circ

# Checkpoint Interview

As always, be sure to schedule an interview to show off your circuit and testcases. Getting credit for the checkpoint requires showing me (a) that it works, and (b) that you understand how it works.