Seth Journal

Week 13

This week was spent debugging why our PIO was not working and testing our code for interchange on the final PCB for the front panel with Brian.

Entry 4

Friday night

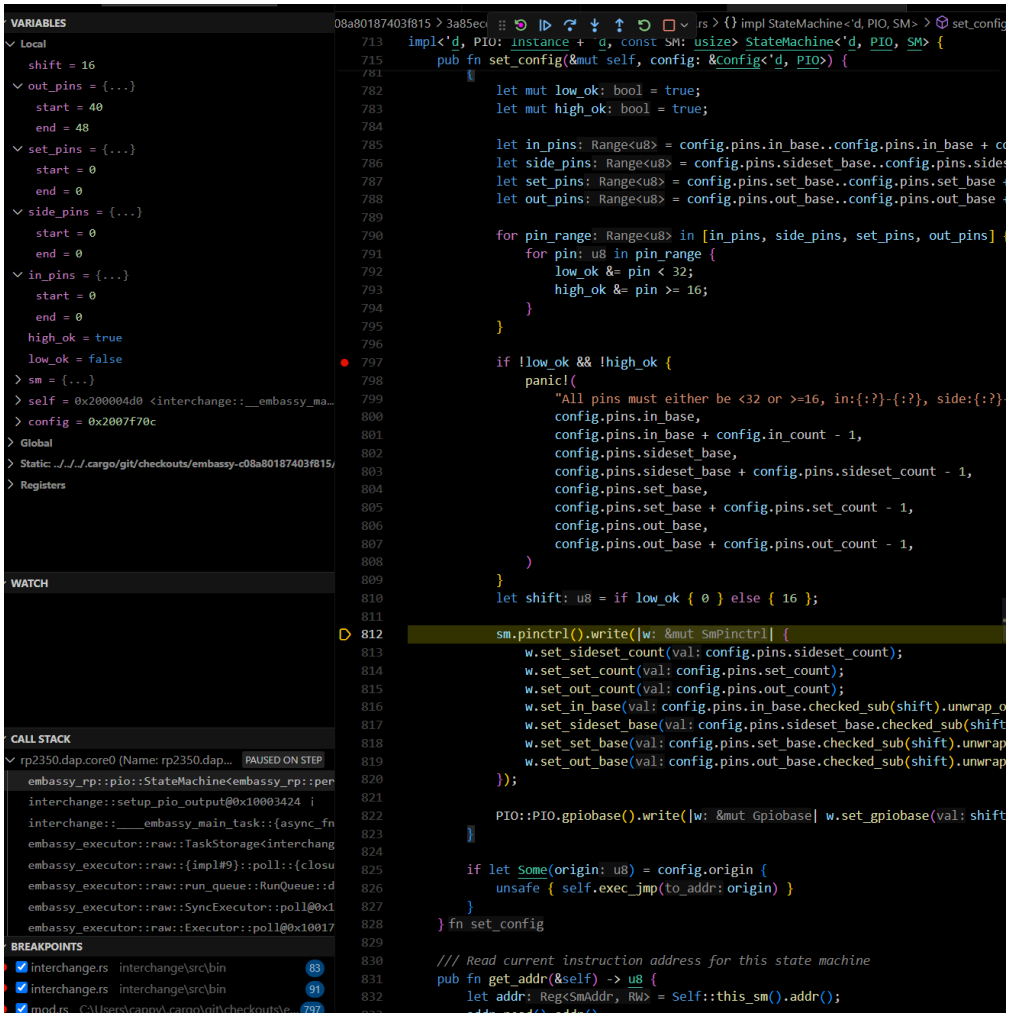
3 hours

10pm-1am

During the time I was away Brian setup VSCode debugging via GDB and OpenOCD so that we could figure out root case by tracing through to verify that GPIOBASE was actually set.

He did this by using the AArch32 bare-metal target Toolchain instead of the one that comes with Rust that probe-rs/openocd uses. This gave us the debugging notation for the ELF file and that we needed in order for debugging to work.

Here is a screenshot of the VSCode debugging working from Brian's computer:



We did this and noticed that GIPOBASE was getting set for the higher pins, unlike what we originally found. This was some erratic behavior. There was no issue it turned out in any of our packages.

We weren't sure what was the issue. I then tried to program our prototype front panel PCB with our PIO program instead of the dev boards since those seemed to be unreliable.

The program then worked for the upper pins and the LEDs lit up on the front panel!

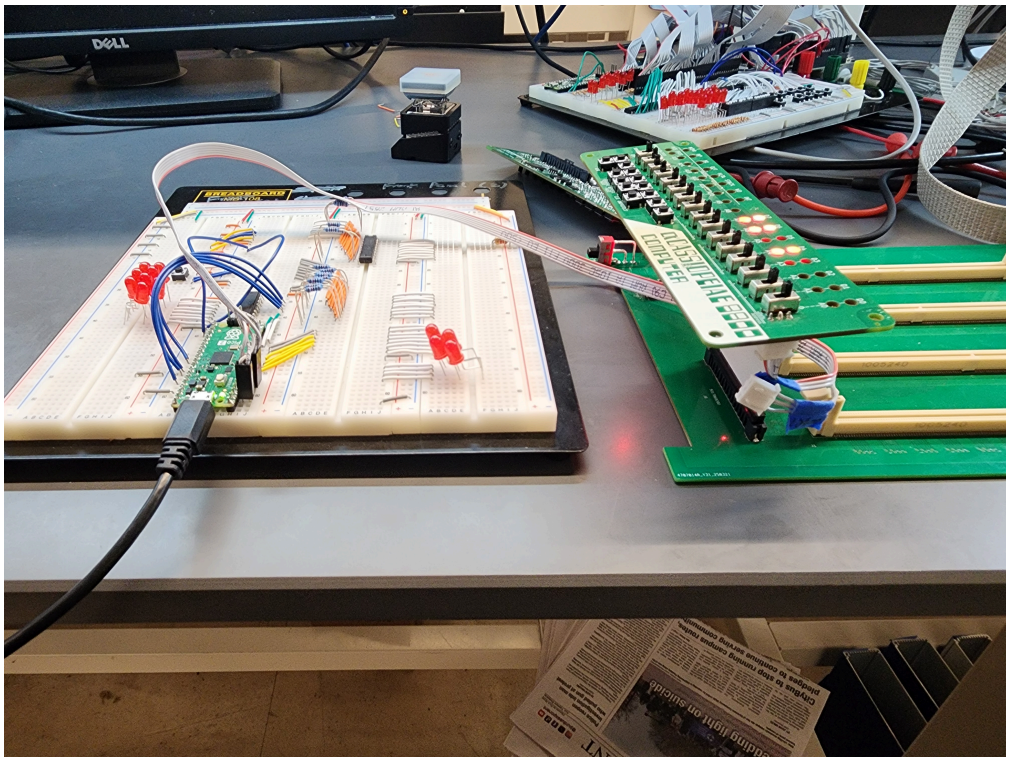
Now that we knew the LED outputs over we then needed to verify that the I2C + IO Expanders also worked on the prototype PCBs. We need to change the pins that the microcontroller knew the IO expanders were on since they were different than our dev boards.

We then used our original program which read from I2C and then output that data on the FIFO for the state machine the PIO program was running from. This would demonstrate that flipping switches would then correspond to changed LEDs on the front panel.

```rust
loop {
    let mut addr0: [u8; 1] = [0];
    let mut addr1: [u8; 1] = [0];
    let mut data: [u8; 1] = [0];
    let mut tmp: [u8; 1] = [0];

    // Read from port B, top IC (data)
    i2c.write_read(address: ADDR1, write: &[GPIOA], read: &mut data).await.unwrap();


    // // Read from port B, top IC (lower addr)
    i2c.write_read(address: ADDR1, write: &[GPIOB], read: &mut addr0).await.unwrap();
    // // Read from port A, bottom IC (top addr)
    i2c.write_read(address: ADDR, write: &[GPIOA], read: &mut addr1).await.unwrap();
    i2c.write_read(address: ADDR, write: &[GPIOB], read: &mut tmp).await.unwrap();

    info!("data = {:02x}  {:02x}  {:02x}  {:02x}", data[0], addr0[0], addr1[0],  tmp[0]);

    // Send the data to the PIO FIFO - this will be output to pins 40-47
    sm0.tx().wait_push(data[0] as u32).await;

    // Optional: Add delay between updates
    Timer::after_millis(50).await;
}
```

And it did!

As you can see on the front panel bellow, 2 of the 8 switches farthest from the front of the image are flipped down while the rest are up. And likewise 6 of the 8 expected LEDs are lit up. When we flipped the other switches we saw that their corresponding LEDs switched as well. This is the expected behavior!

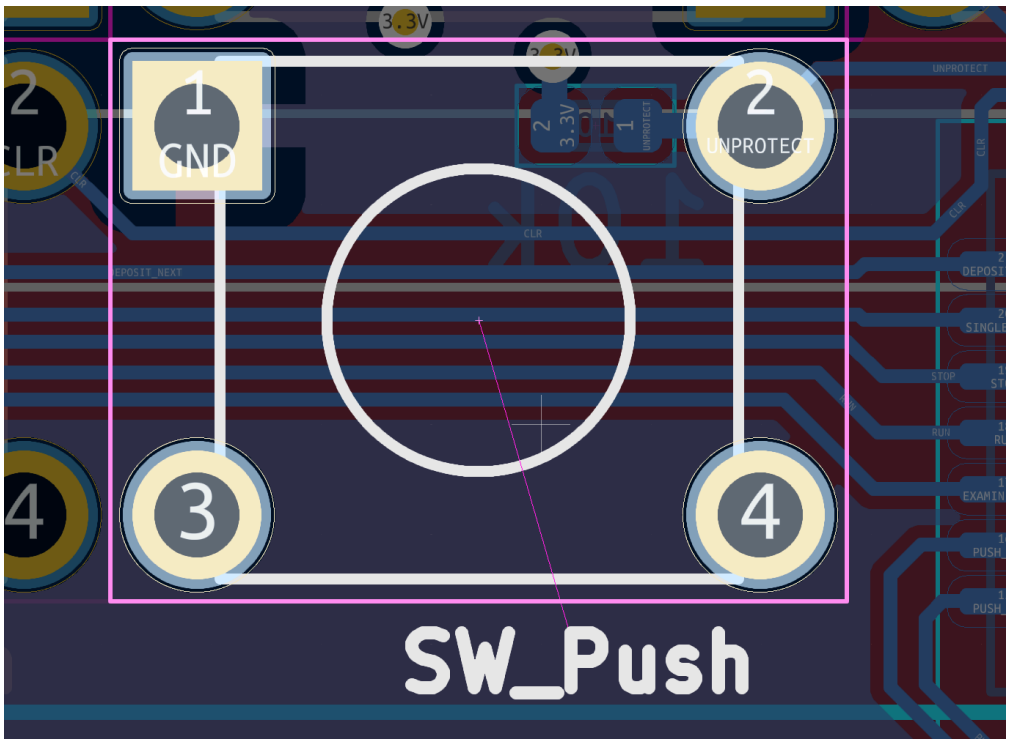


We then tested if the buttons worked as well.

We noticed that they actually did not work. When we pressed a button, no output on our terminal changed that indicated the signals coming from I2C.

When we looked at the datasheet for the button, we actually noticed that the notation in kicad for what was one side of the switch was different than the datasheet.

Below you can see that the schematic infers that when the button is pushed, pins 2 and 4 are linked with 1 and 3. But what actually happens is the button is split horizontally and pins 1 and 2 are connect to 3 and 4 when the button is pressed.



This was a annoying sign because it meant that all we were getting outputs that were just tied to ground. We realized we needed to desolder all the buttons and somehow rework them to allow for the button to actually work. We decided to later cut off pin 1 to remove it from being tied to ground.

Next steps are to actually complete the full interchange logic so that the buttons and switches work as function and bus signals are sent to control the CPU and memory from the front panel.

We also need to rework the buttons so that we can actually use them. They will need to be desoldered and legs cut off and then resoldered.

Entry 3

4/9 Weds night

6pm-8pm

2 hours

When I came back for tonight, Brian had determined that there was a line in the `rp235x-pac` Rust crate that was not setting the GPIOBASE correctly.

I needed to apply a patch to our Cargo patch to fix this.

`rp235x-pac` patch `src/inner/pio0/gpiobase.rs`

```
  impl W {
      #[doc = "Bit 4"]
      #[inline(always)]
      #[must_use]
      pub fn gpiobase(&mut self) -> GPIOBASE_W<GPIOBASE_SPEC> {
-         GPIOBASE_W::new(self, 4)
+         GPIOBASE_W::new(self, 16)
      }
  }
```

Through a google search I learned there are multiple ways to do this. You can:

1. Download the repository of the package, apply the patch, and repoint your Cargo.toml to it instead of downloading it from crates-io.
2. Use a crate like cargo-patch.

I decided to go with the first solution since it seemed easier and we needed to apply the patch to all of our packages in our Cargo workspace so i thought this would be easier.

I downloaded the repo and made the change and added

```
[patch.crates-io]
rp235x-pac= { path = "../rp235x-pac" }
```

to the `Cargo.toml` in our interchange package since that was the package we were using.

I did this and rebuilt the project. I didn't notice anything from the terminal saying that it applied the patch or anything which concerned me.

When I the programmed the project to the microcontroller with pin 47 as the output for PIO, I did not see it light up or any signal on the oscilloscope.

I was concerned the patch was not being applied. I wanted to verify that the dependency rp235x-pac was actually a dependdency since it wasn't a root-level dependency I saw in our Cargo.toml. I asked chatgpt how i could get a printout of the dependency tree with cargo and it said I should run `cargo tree`. I ran that but still did not see `rp235x-pac`.

Brian came back at this time and he also seemed to be mistaken that `rp235x-pac` was causing the problem. We decided we wanted to step through the the functions this time using a debugger instead of just searching through the code at what might be the cause for GPIOBASE not being set.

Brian and I tried GDB first, however, when we tried setting a breakpoint at a function name, it was unable to find it. We guessed that embassy or Rust mixed up or did not provide notations for the embassy functions in the ELF file cause of state machines or something so we needed to find another solution.

Brian and I decided that VSCode debugging and fixing the debugger would be the best way forward so Brian got to work at getting the debugging figured out and the VSCode launch file setup and I went home.

Entry 2

4/9 Weds morning

10am-12am

2 hours

Today we tried to get the PIO (programmable IO) working to output to the LEDs on the dev board . We need this so that we can drive our parallel bus with PIO instead of bit banging, which is much slower.

I had tried this before on the dev board but was failing when trying to get pin 47 or 46 to trigger. I then tried a lower level pin because I thought maybe that might be the issue as some elements of the microcontroller have different operation for the larger QFN package. Pin 30 worked.

We then tested pin 46 and 47 again to make sure that the this wasn't a one off fluke. The lights on the board barely lit up for pin 30 so we verified with an oscilloscope to see if any signal could be observed. We did not see any.

We then needed to debug why some pins were working but others were not. Brian setup OpenOCD to observe the registers of the microcontroller. I installed telnet on my computer as OpenOCD requires it. We were able to observe that there was no PIO code in the registers where PIO code was supposed to be. But this was the case for both when the program specified pin 30 as the output pin and pins 46 and 47, so we didn't think this was the issue.

Brian and I then searched the datasheet for the RP2350 to see if there were any other configuration registers that controlled PIO related output and found that a GPIOBASE register exists:

PIO: GPIOBASE Register

Offset: 0x168

Table 1016. GPIOBASE Register

Bits	Description	Type	Reset
31:5	Reserved.	-	-
4	Relocate GPIO 0 (from PIO's point of view) in the system GPIO numbering, to access more than 32 GPIOs from PIO. Only the values 0 and 16 are supported (only bit 4 is writable).	RW	0x0
3:0	Reserved.	-	-

We then check that this was enabled in memory for the program that drove pins 46 and 47 but did not see this.

We then needed to search through the embassy HAL for the RP2350 to figure out if GPIOBASE was set and where.

Brian got to work on this and I went home. Next steps are to continue to debugging so that we can drive higher level GIPO pins with PIO.

Entry 1

4/7 Monday night

10-11pm

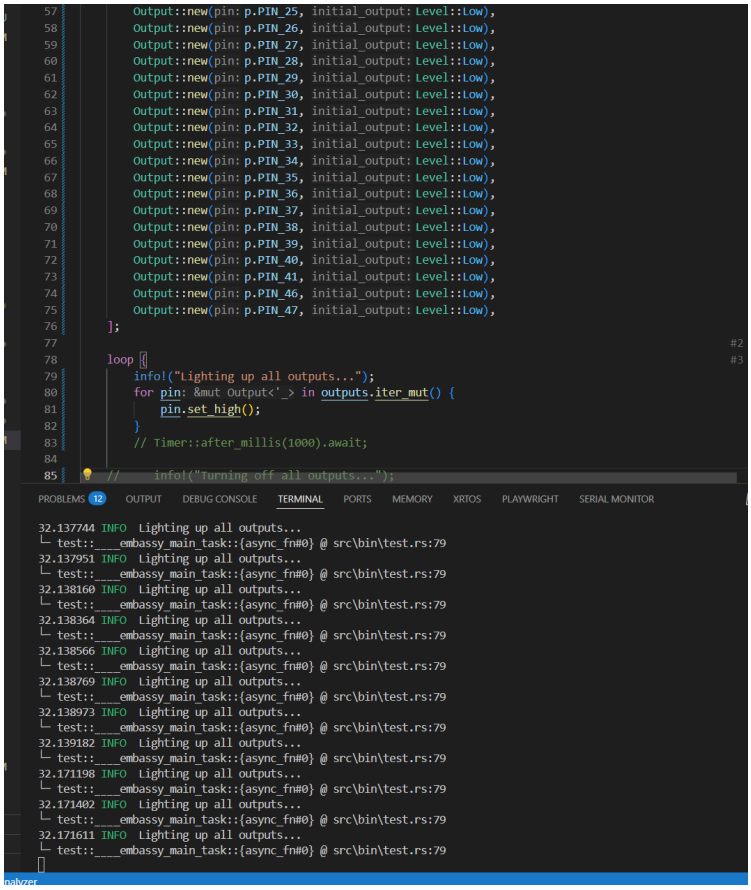
1 hour

Today I tested that the front panel and cards were able to be programmed with Brian.

We first hooked up the card and front panel to the backplane and connected the card from a Pico acting as a debug probe to the card over SWD.

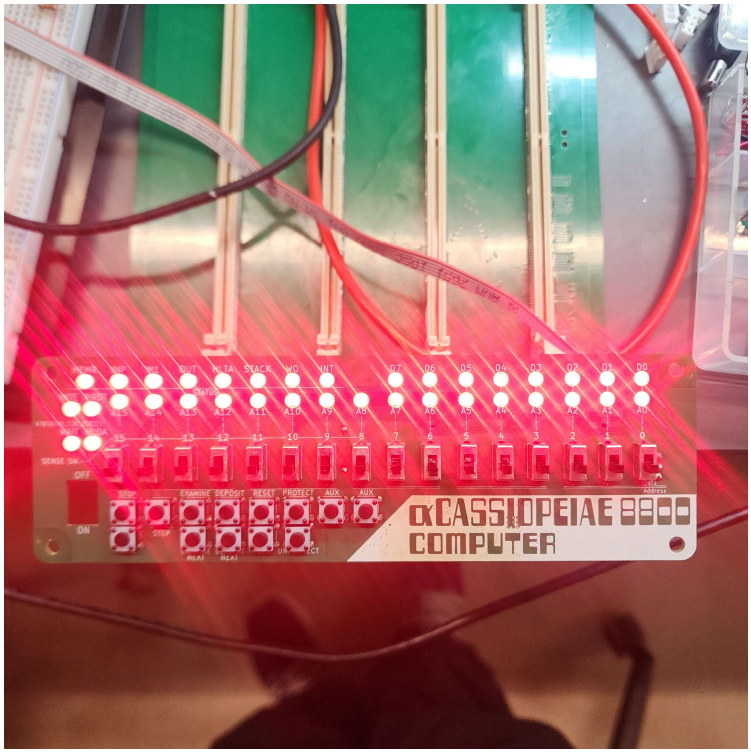
We did this after Brian last week figured out that the SWD programming was not working because the flash chip was improperly wired. Brian rewired it and we were able to reprogram it again.

We flashed a program that turned on all of the LEDs to the card



Almost all of the LEDs lit up on the front panel. That let us know that the connection between the card and the front panel for all the bus lines was not reliable. Our soldering for either the DIMM slot, or the ribbon cable connector may not be correct.

I then programmed the front panel with the same program and all of the LEDs did lit up this time:



This let us know that all the routings between the microcontroller on the front panel were at least correct.

Next steps are to fix the soldering between the card and the front panel so that the card can drive all the LEDs on the bus properly.

Week 12

Entry 1

Wednesday

10-11am

1 hour

Today I registered my team for the spark challenge and design expo. I also did a more research on how to output to parallel pins with PIO using the embassy Rust package.

Week 11

Entry 3

3/28 Friday

10-12

2 hours

For this time I focused on drafting the code for the rest of the front panel functionality.

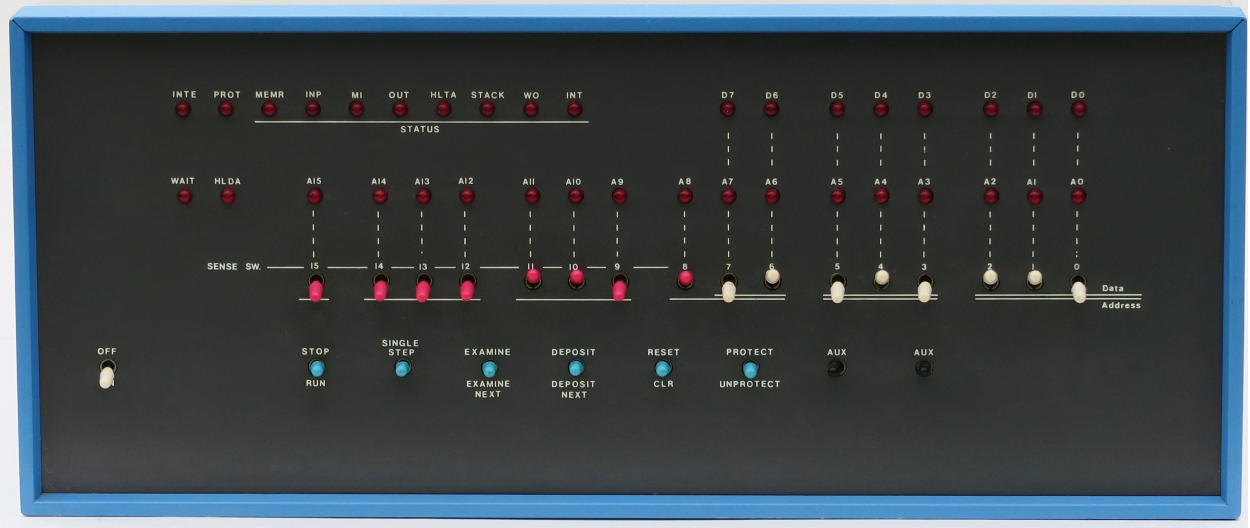
I needed to write code to output to the LEDs by writing to the ADDR and Data lines on the bus.

```
// Read from port A, top IC (data)
i2c.write_read(ADDR, &[GPIOA], &mut data).await.unwrap();
info!("data = {:02x}", data[0]);

// Read from port B, top IC (lower addr)
i2c.write_read(ADDR, &[GPIOB], &mut addr0).await.unwrap();
info!("addr0 = {:02x}", addr0[0]);

// Read from port A, bottom IC (top addr)
i2c.write_read(ADDR1, &[GPIOA], &mut addr1).await.unwrap();
info!("addr1 = {:02x}", addr1[0]);
```

I need to do the button functionality and the different operations associated with them like single step, examine, and deposit.

I need to tie the switches to such programming logic.

Here is the button logic I came up with using the button reference of course below, chatgpt to remind me of each button functionality, and our bus layout table



GPIO #	Type	Name	Description	Ownership		
4	Control	CLOCK	bus clock	Panel		
5	Control	~PRESET	reset	Panel		
6	UI	PANEL	front panel data control	Panel		
7	UI	PROT	memory protection	Panel		
8	Status	SM1	first machine cycle of instruct	CPU		
9	Status	SHLTA	halt	CPU		
10	Status	SSTACK	stack operation	CPU		
11	DMA	~PHOLD	hold bus control			
12	R/W	PDBIN	data bus in	CPU		
13	R/W	SWO	write-out	CPU		
14	R/W	PWAIT	mem/io	CPU		
15	I/O	PRDY	io ready			
16	I/O	SOUT	output IO cycle	CPU		
17	I/O	SINP	input io cycle	CPU		
18	Memory	XRDY	memory ready	RAM		
19	Memory	MWRT	memory write strobe	CPU		
20	Memory	SMEMR	memory read strobe	CPU		
21	Interrupt	~PINT	interrupt request			
22	Interrupt	SINTA	interrupt acknowledge	CPU		
23	Interrupt	PINTE	interrupt enable	CPU		
24-39	Address	A{15:0}				
40-47	Data	D{7:0}	tri-state	CPU	RAM	Panel

```
if (reset) {
   nRst = 0;
   addr = 0;
}
else {
   nRst = 1;
   if (clr) {
      addr = 0;
      data = hiz;
      clock = clock;
   }
   else if (stop) {
      clock = clock;
   }
   else if (run) {
      clock = !clock;
   }
   else if (single_step) {
      single_step = 1;
   }
   else if (examine) {
      // read addr pins
      i2c.write_read(ADDR, &[GPIOB], &mut addr0).await.unwrap();
      i2c.write_read(ADDR1, &[GPIOA], &mut addr1).await.unwrap();

      // flog loan word instruction over the bus
      panel = 1;
      // send addr
      panel = 0;
   }
```

```
    else if (examine_next) {
        // flog loan word with next addr
        panel = 1;
        panel = 0;
    }
    else if (deposit) {
        panel = 1;
        // read data
        i2c.write_read(ADDR, &[GPIOA], &mut data).await.unwrap();
        // send data sw (addr)
        panel = 0;
    }
    else if (deposit_next) {
        panel = 1;
        // read data
        i2c.write_read(ADDR, &[GPIOA], &mut data).await.unwrap();
        // send data sw addr + 1
        panel = 0;
    }
}
```

I clarified with brian what the HALT and PANEL signals do on the bus. HALT is just a status flag for when the cpu reaches a halt instr. on the front panel we also trigger the halt led when that happens. PANEL is a flag asserted by the panel to indicate to the cpu and mem it wants to take control of the data lines. That way the panel can issue instructions to write to memory.

The next step is to turn this from psuedocode into actual rust code and write the PIO setup and runtime tasks to read and write to respective pins on the bus. I'll also need to figure out how to do reads and writes from the front panel too.

Entry 2

3/26 Wednesday

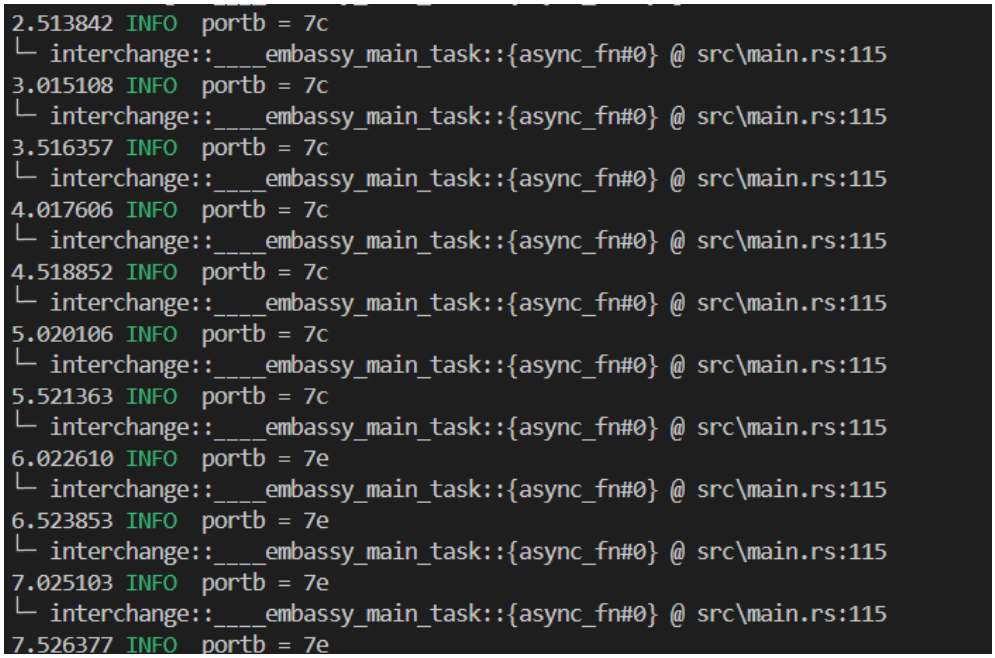
10:00-11:00

1 hour

Today I need to fix why the messages printing to console were not always the input state of the switches.

I chatgpted this and it told me that the MCP likely cycles through using a register pointer all the registers and prints the values of those instead of just the input data to the IO expander. I would need to issue a write over I2C requesting that I wanted just the data and then it would give that back to me for each iteration.

That's when I realized that I actually needed the `write_read()` function as was used in the original I2C example. This function writes to the bus first and then reads a response. I replaced the `blocking_read()` with this line: `i2c.write_read(ADDR, &[GPIOB], &mut portb).await.unwrap();` and it worked!



As you can see, it prints out in hex whether each switch is on or off and halfway here I flip one of the switches so the output turns from 7c to 7e.

I had some flashing issues before programming it because of a known bug so this took about 20 minutes to figure out how to get consistent flashing and running to occur. I figured out that hitting reset on the IC after you program it once and it doesn't run and then reprogramming it, causes it to successfully run the next time about 80% of the time. If that doesn't work after consecutive tries, unplugging and reconnecting the entire system to power over usb.

My next step was to continue working on the front panel code and

Entry 1

3/25 Tuesday

11:30-5:30

6 hours

Today I wanted to get a working bus implementation working

I pulled out the bus prototyping board and I realized I needed debug LEDs. I could add them to the bus board, however we already had the front panel breadboard done, so I thought, oh i could just code that right now since the front panel code needs to be done at some point too.

I looked at the board and pulled up the datasheets for the 2 boards

https://www.ti.com/lit/ds/symlink/tlc59211.pdf?ts=1742960394353&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FTLC59211

https://ww1.microchip.com/downloads/aemDocuments/documents/APID/ProductDocuments/DataSheets/MCP23017-Data-Sheet-DS20001952.pdf

I verified all the pin connections and learned the power, data, and addrs pins on the io expanders

I noticed some of the ICs nathan hadn't connected the power yet. I added the wiring for those

I then added the power and ground wires from the pico to the board rails

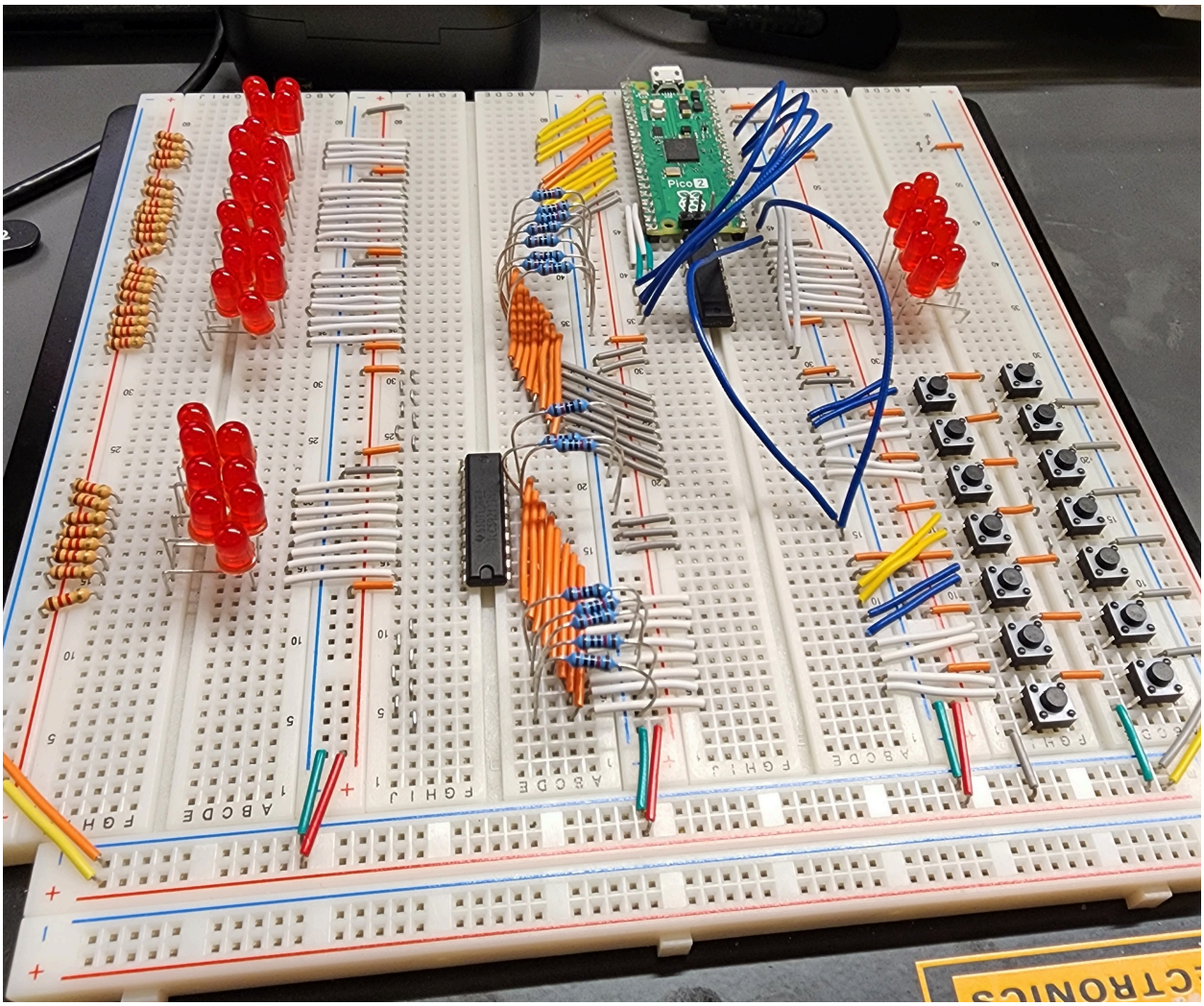
I then power the pico up over USB and the board LEDs lit up.

I noticed the switch LEDs were still shining

I asked brian why and he noted there weren't pull down resistors and that the switch inputs leading to the LEDs were in a floating state.

I added 10k pullup resistors (the ones in the middle below) and the weird LED behavior went away.

And here is a partially disassembled view of the final setup of the front panel breadboard:

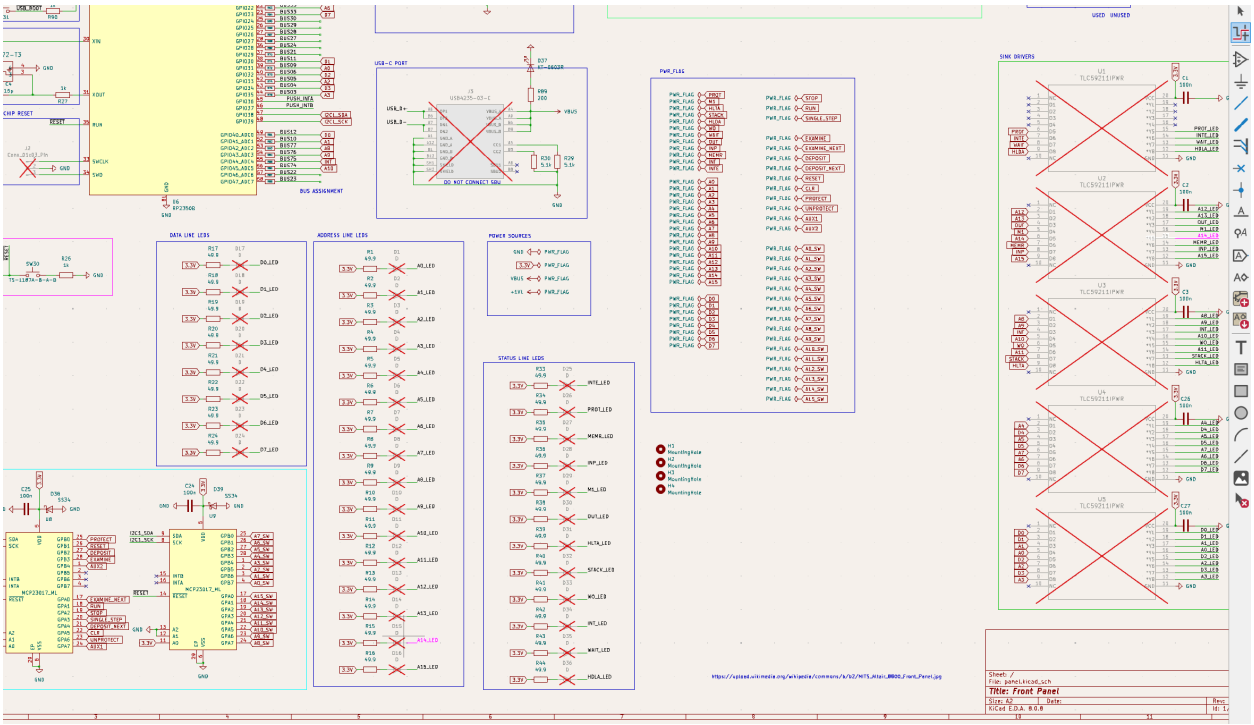


However, I realized that the LEDs in this configuration are directly conected to the switch inputs and are not individual inputs like they are on our production board. Additionally, getting this breadboard to be able to connect with our other dual-stamp bus breadboard so I could use the LEDs as bus debugging LEDs was going to basically be impossible because I had already used up all the GPIO pins on the raspi pico. I could have used our third stamp which had more pins to replace the pico and then bridged that over, however that would take a lot of work rewiring.

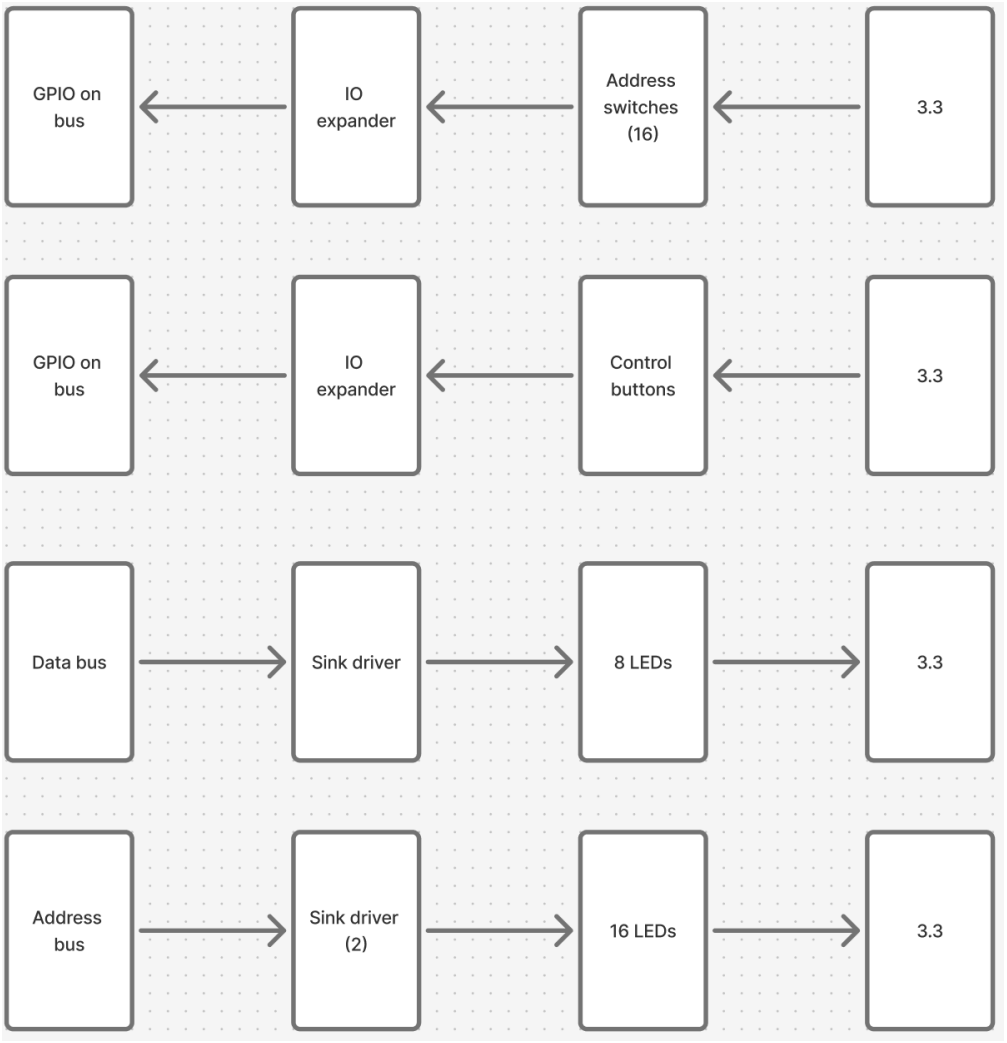
Instead I opted for a more concise solution where we would move all the switches, buttons, and LEDs of the front panel, to just the main bus breadboard and have one of the two stamps act as the microcontroller for the front panel, and the other act as both the cpu and the memory. This would consolidate all of our testing down to one breadboard and be much simpler to wire than adding a third stamp.

First I investigated how I was going to route this. I pulled up the schematic for the current front panel and noted all of the connections again using the highlight nets tool.

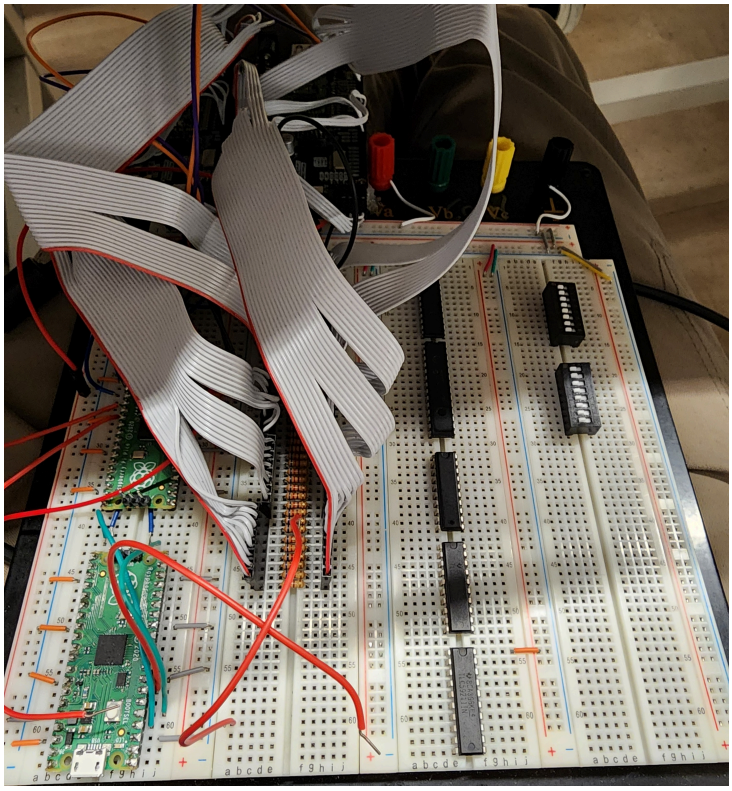
Here is one such connections (highlighted pink):



I then made a schematic in Figjam to plan the the ordering and all of the stages, just to verify they would all fit on the breadboard:



Now that I knew I had enough pins to put all the IO Expanders and sink drivers on one row, and the switches, buttons, and LEDs on another, I placed them there:



I then began routing the rest of the board.

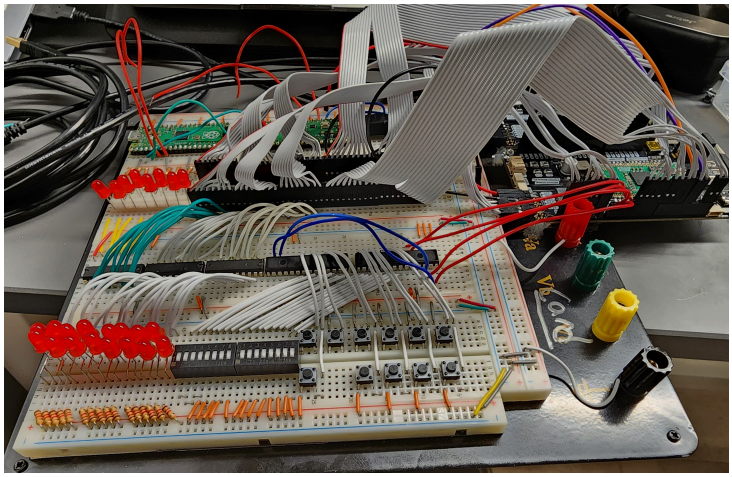
I figured out all 24 LEDs weren't going to fit in the second column cause the switch arrays each took up 2 extra pins, so I moved the data LEDs to next to the bus on the second row.

I then placed the address LEDs at the bottom in the 4th row, moved the switches down next to them, and then placed the buttons. This had enough room so it worked perfectly.

I then began wiring up the LEDs to the sink drivers, the sink drivers to the bus, the buttons and switches to the IO expanders, and the IO expanders to the bus. We are using 2 interrupt pins on one of the io expanders, so I added those to directly to the stamp board because we needed to reserve enough pins for the other simulated bus signals.

Finally I routed the power and checked all the wiring.

Here is the final wiring:



I plugged it in and it appeared nothing was smoking!

I didn't notice any LEDs were on but this was expected since none of the GPIO pins were being driven.

I wanted to get the LEDs working so first I programmed a blinky program to light up the LED on pin 47. It worked. Pin 47 linked to the data LED on the far upper left of the board as shown above. It blinked, however, it was very dim. I wasn't sure why it was very dim vs. the first board where the LEDs lit up more strongly. Maybe more current drained on the bus pins?

Anyways, the LEDs/bus GPIO outputs were confirmed to be working, so I then moved onto confirming the switches and buttons worked as inputs.

I needed to figure out how I2C worked again and what messages were transferred on our MCP23017.

I used chatgpt for this to give me a quick overview of I2C again and I looked at the IC datasheet to learn that the MCP broadcasts the values of its registers over I2C.

I then needed an I2C example from our HAL library, embassy, and I found one. Luckily, the example actually was personally made for the MCP23017 and I could use the code immediately. I realized the original code used read_write() functions and I thought these weren't necessary for basic reads, so I replaced them with blocking_read().

I also realized from the datasheet that the MCP doesn't have pull down resistors. only pull up. So I needed to reroute all switches and buttons to ground instead of 3.3 which I had them before.

Here is the first iteration of testing code with these changes:

```
use mcp23017::*;

info!("init mcp23017 config for IxpandO");
// init - a inputs, b inputs
i2c.write(ADDR, &[IODIRA, 0xff]).await.unwrap(); // all inputs
i2c.write(ADDR, &[IODIRB, 0xff]).await.unwrap(); // all inputs
i2c.write(ADDR, &[GPPUA, 0xff]).await.unwrap(); // pull up inputs
i2c.write(ADDR, &[GPPUB, 0xff]).await.unwrap(); // pull up inputs

loop {
    let mut porta = [0];
    let mut portb = [0];

    // Read from port A (buttons)
    //i2c.blocking_read(ADDR, &mut porta).unwrap();
    //info!("porta = {:02x}", porta[0]);

    // Read from port B (switches)
    i2c.blocking_read(ADDR, &mut portb).unwrap();
    info!("portb = {:02x}", portb[0]);

    Timer::after_millis(500).await;
}
```

I then built and flashed this program to the stamp and it began to run.

I noticed though that as it was running, it wasn't always outputting the value of one of the switch arrays after the portb line. it only did it like every 7 meesages like here:

```
2.012589 INFO  portb = 7c
└─ interchange::____embassy_main_task::{async_fn#0} @ src\main.rs:115
```

I had run out of time for the day so I saved fixing this issue for tomorrow.

Week 9

Entry 2

3/13 Thursday

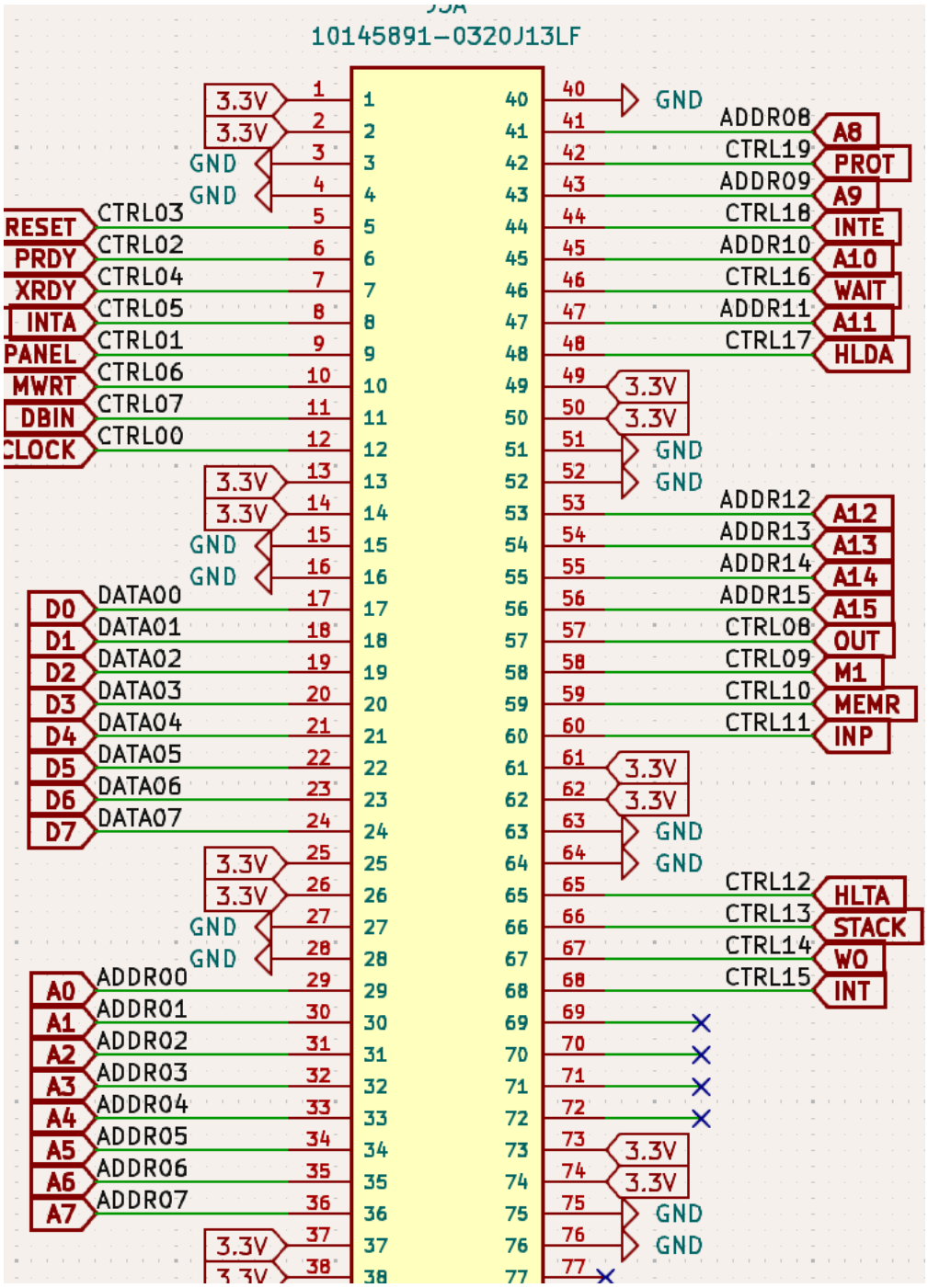
10pm-1:30am

3.5 hours

Today I needed to figure out if we were going to keep the Brian designated/non-essential pin order for the cards/dimm and route more difficult, or maybe switch their positioning up based on what pin order the card is in.

As you can see, the pin order for the front panel is not organized very well. The data and address pins are not next to each other. This was to make routing on the front panel easier

Meanwhile the pin functionality order was organized for the cards:



So first, I needed to learn if the PIO pins do need to be next to each other to send data over the bus. I looked up how to send parallel bus signals using PIO since I couldn't find any Embassy examples that did that. I found this Raspi forum post showing how to do it. They just needed to add the line

```
pio_sm_set_consecutive_pindirs(pio, sm, 0, 8, true);
```

This gave me some indication that you can only setup output pins that are consecutive.

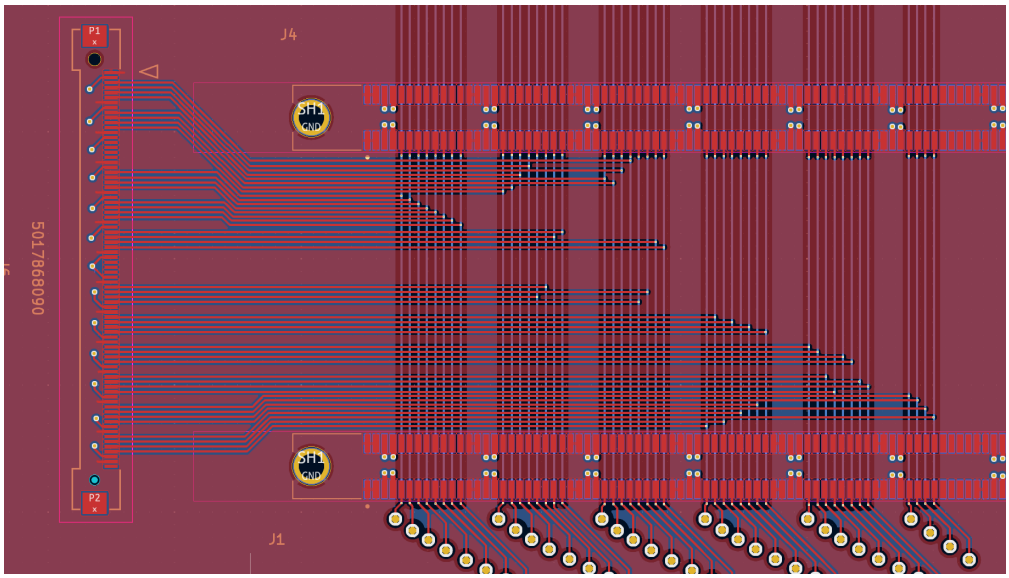
Then I confirmed that setting consecutive pins was the only way to set out pins for the Embassy rust package (the one we're using) and this was correct:

```
pub fn set_out_pins(&mut self, pins: &[&Pin<'d, PIO>])                    Source
```

Sets the range of pins affected by OUT instructions. The range must be consecutive. Out pins must configured as outputs using `StateMachine::set_pin_dirs` to be effective.

So if the pins need to be next to each other for the card and moving them around in SW would be bad, then it would make sense to keep their ordering and route them criss cross to the corresponding backplane/ffc connector pins.

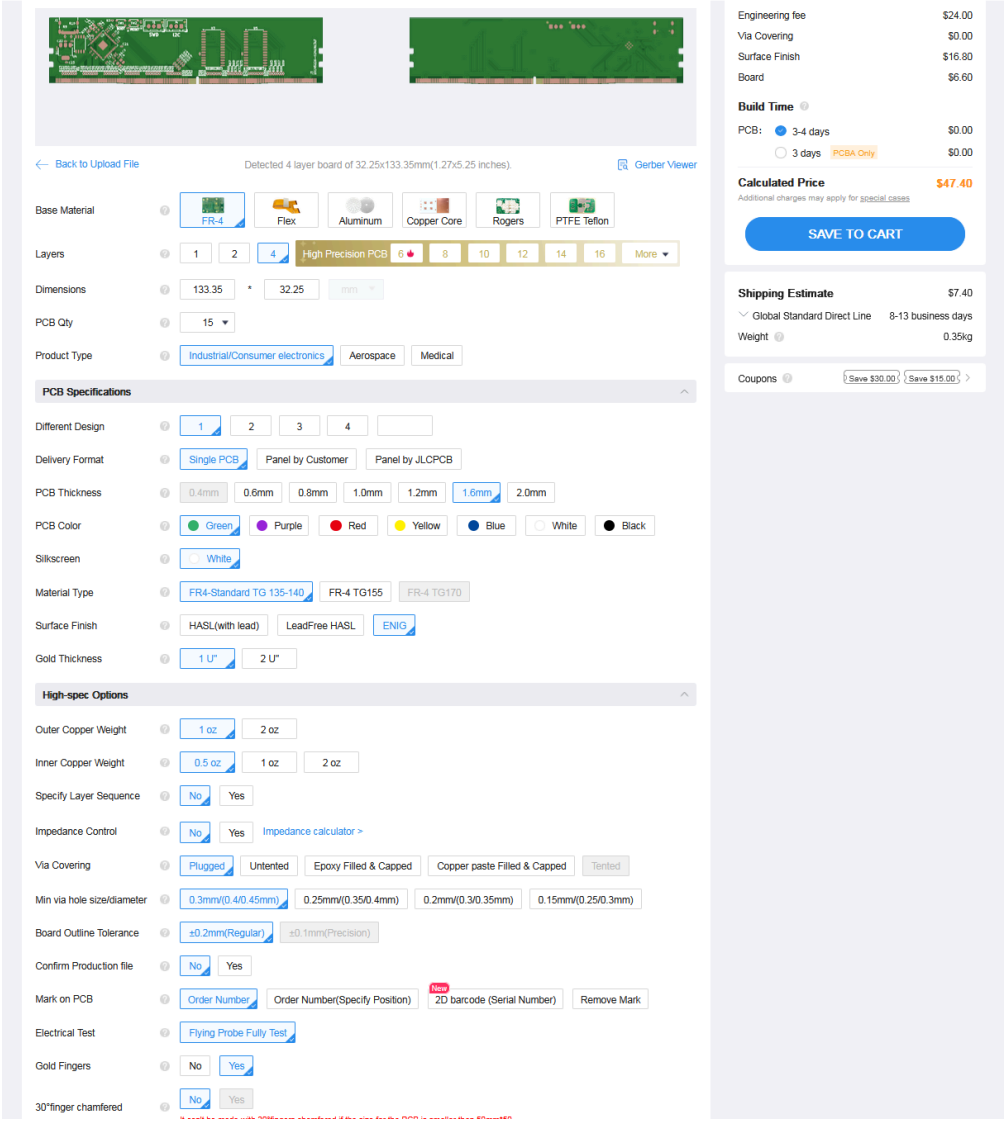
Nathan took over backplane routing from here so he did the criss-cross layout of the traces based on their assigned functionality:



Since Brian was cleaning up the front panel, the card was done, and Nathan was working on the back, I decided to get started researching how much our PCB order was going to cost.

I exported our card PCB as gerber files and went to JLC PCB's website.

Our PCB was processed successfully and here is the configuration chosen:

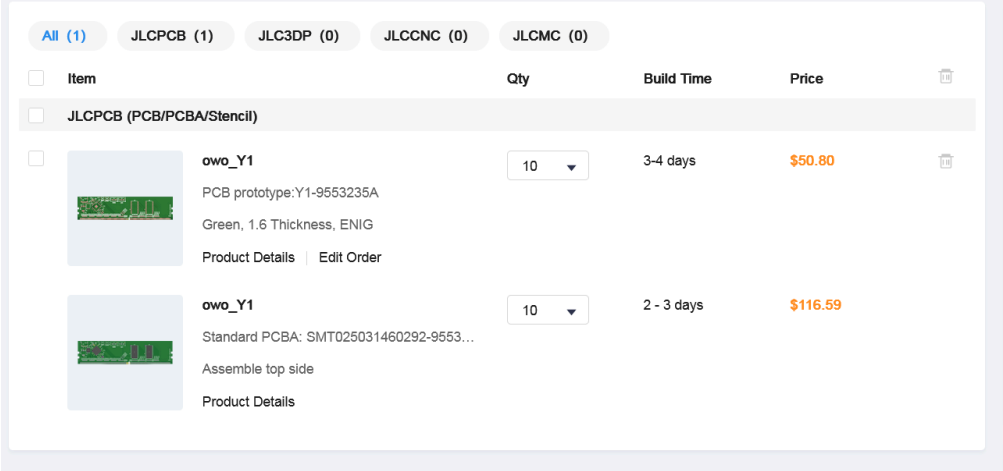


We need about 15 cards 3 cards x 4 machines. And we also needed Gold fingers for our edge DIMM connectors. Finally we need ENIG surface finish since only that is comptible with gold fingers. This came out to be $47.

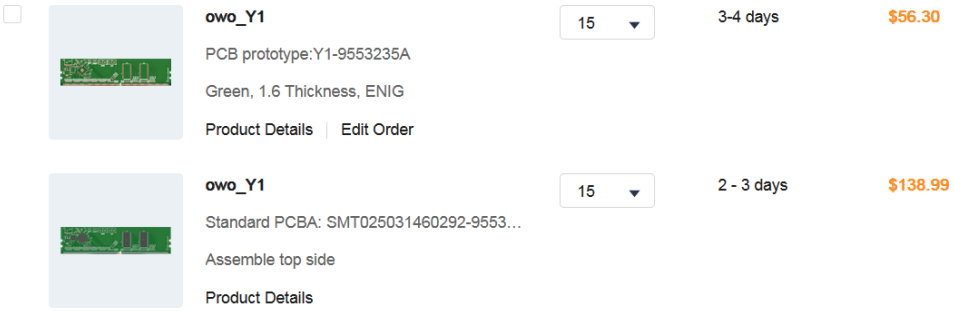
Then we needed to have JLC do assembly as well because that's the only way we can get our microcontroller on the board. I clicked on the assembly tab and it requested a BOM and placement file. I exported the BOM from kicad from the fabrication outputs but I was unsure how to get the placements. I thought location from fabrication output would be good, however, it failed to upload.

I then googled how to get this placements file and i came across the JLC PCB kicad extension Fabrication Toolkit. This makes it easy for you to export all the files needed for JLC so I downloaded it and used it and the placement file it generated worked. I then saw all the parts that JLC had and proceeded and got my final assembly cost.

For 10 cards:



For 15:



As you can see, there's not much difference when you scale. The assembly price was quite a lot.

If we were to get 10 cards, that would be around $170 and our enclosures which also are a large cost that we already know are $39 each, would come out to $160 for 4. That is $330 spent with only $280 left for the 5 front panel cards and the 5 back panel. It maybe shoudl be enought? The front panel has a lot of components though so I'm a bit concerned. I will wait and see how much those are calculated as since I am going to bed now. Brian and Nathan will calculate them in the morning before the order them. This gives us a good idea now though on what we can and can't order and where we could cut costs.

As for me, my next steps is to get going on the software over break. I have brought home our development breadboards so I can work on sw over break. I will need to complete tram over break to stay on schedule. I'd also like to do cycle (CPU) and interconnect (front panel) since then I won't have to do work on the week after break. I have a good understanding of PIO and how output works now that I know how the consecutive pin output assigning works for a parallel bus so I am confident i can get a bus implementation with state machines working quickly.

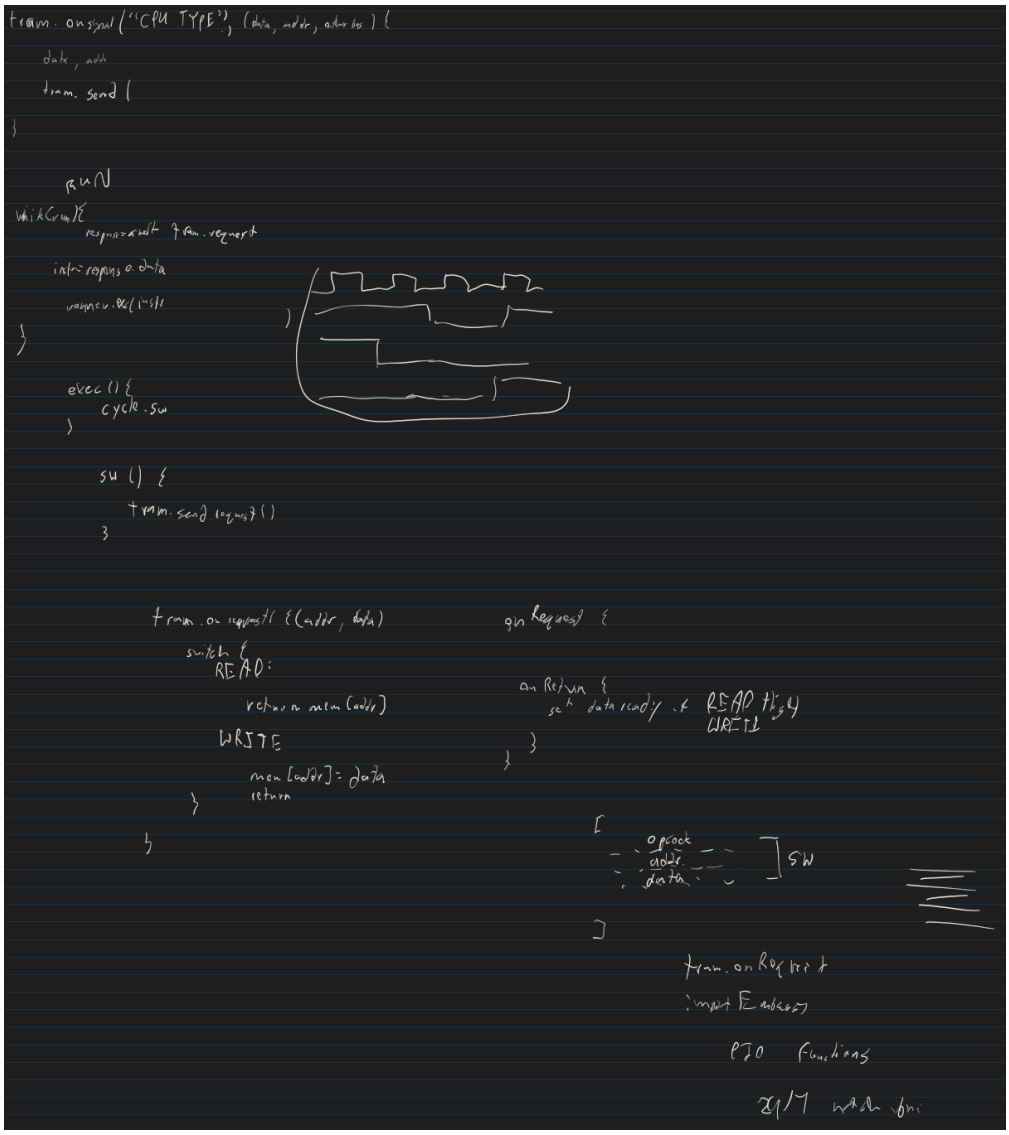
Entry 1

3/12 Wednesday

8pm-2am

6 hours

Today I first talked with Caleb about how our software stack is going to work.

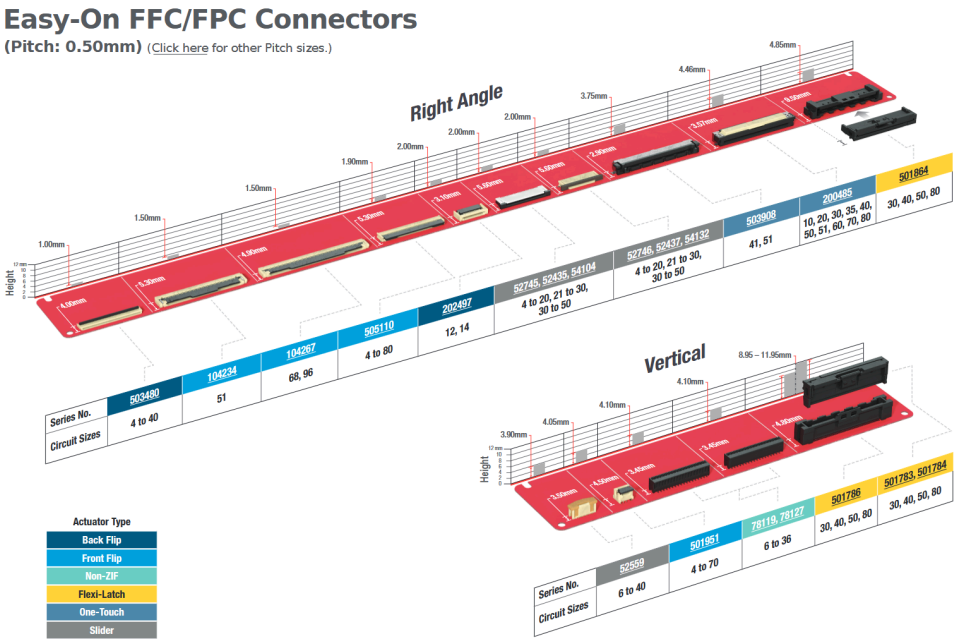


I made the diagram above to explain that retreiving instructions from memory takes actually 3 clock cycles and that tram would handle doing memory reads and the CPU firmware would not worry about that. That si why tram is its own package. As you can see, I also wrote some psudocode on how we could use async/await, callback functions and event listeners to create onRequest and sendRequest functions for tram and then exec intructions for exec. This helped me better understand the software stack and I copied this psuedocode in a new frm over to our tram repo and got started on it.

Next I went back to working on our backplane since I realized that we need to add the connector that will link the front panel with the back panel. The connector will not be a DIMM connector. I knew it was going to be a ribbon cable. I learned Nathan had previously chosen and FFC connector for the front panel, so I looked at the datasheet for that to get a better idea of how it works. I also googled/chatgpted what the difference between an FFC and FPC connector/cable is since the datasheet uses both terms. FPC is like a printed circuit in the cable so we only care for FFC type ribbon cables.

I also inquired Nathan why he chose this connector since it is quite large. I learned that we need an 80 pin connector for all our bus signals and Molex only makes 80 pin connetors that are in this size. We also needed a vertical FFC connector and Molex only makes them in this formfactor too.

I found this Molex product guide and it shows all their connector types



As you can see, the larest is the only vertical one that supports that many pins. and has a nice clip.

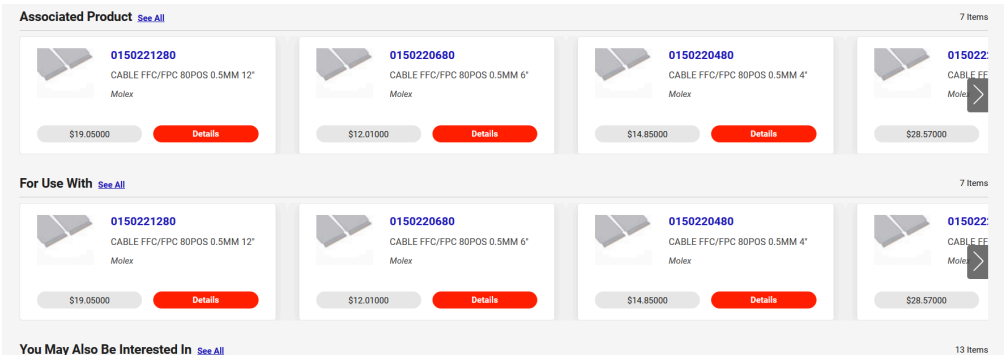
Now that the connector was confirmed, I needed to find a compatible cable for it. I noticed on the datasheet it said the connector was compatible with a certain type of jacket. I didn't know what a jacket was so I looked that up and it is a plastic housing that protects cables that goes on the outside of an FFC cable. Here were the Jackets that were compatible with our connector:

Use with Part(s)

Description	Part Number
0.50mm Pitch FFC-to-Board Plug Jacket, 80 Circuits	5017838009
0.50mm Pitch FFC-to-Board Plug Jacket Cover, 80 Circuits	5017848009

I chose the 783 varient and found its part on digikey. Then I just needed the cable.

I tried using Molex's website to find compatible cables, however, they didn't have a filter for finding cables based on compatible connectors so I was very lost. Until, I went to digikey's website! When I pulled up the Digikey page for our connector, there was a compatible parts section at the bottom that showed which cables were compatible:

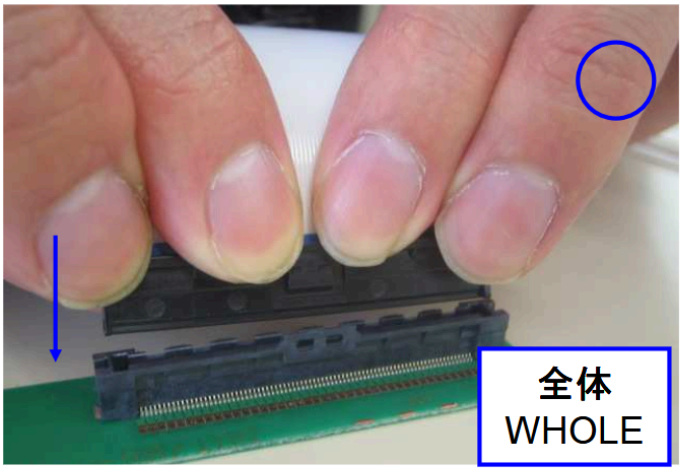


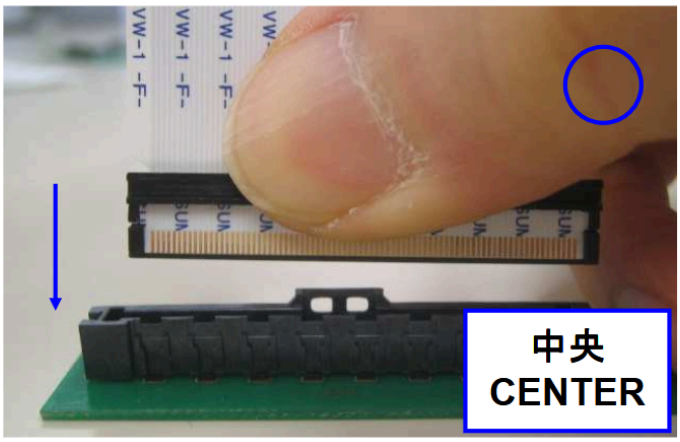
Here were all the varients! In the future, I now know I should probably search for parts using distributor websites since they are going to be the ones I'll be buying the product from in the end and they have better filters as well. I just didn't know they sold these as I struggled to find them in the first place.

The cables were all different sizes. We only need a small ribbon cable. Unfortunately the 2 in ones were sold out so I selected the 4 inch one.

Next I need to figure out what the cable looked like and how it was inserted, just to make sure how it was going to work with the jacket and such. I found the product specification from Molex for the connector and inside of it, it shows the jacket and how the cable is oriented inside of the jacket:

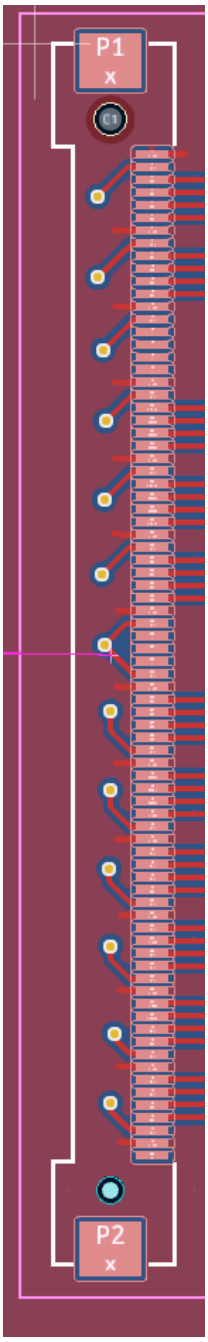
When mating the connector, hold the JACKET AND JACKET COVER together at either a whole or the center portion of them to insert.



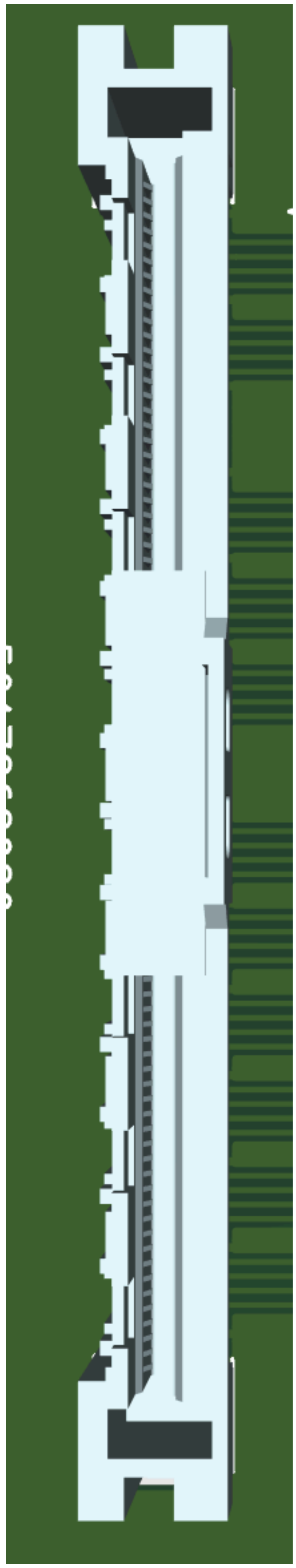


With this knowledge, I now know that we need to orient our connectors for the frontplane and backplane to match which way the ribbon will be pointing so that the connectors are on the right side. The cable only has connectors on one side so this is why it is important.

I placed the connector in our PCB layout for routing later:



I also learned from brian how to import the 3D model for it from digikey because it originally wasn't showing up in the 3D view for the backplane. Now it does:



Ok, now that the FFC connector is sorted, I also realized that a power switch was missing from the backplane. This needs to be a vertical power switch that will show through a cutout on the front panel. To fin this part, I simply looked for in-stock vertical 2 way switches on Digikey. Here is the cheapest option I found and it fits our needs:

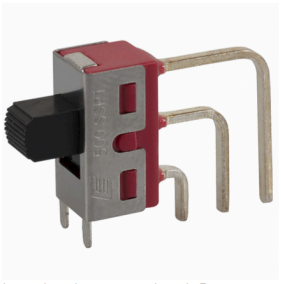
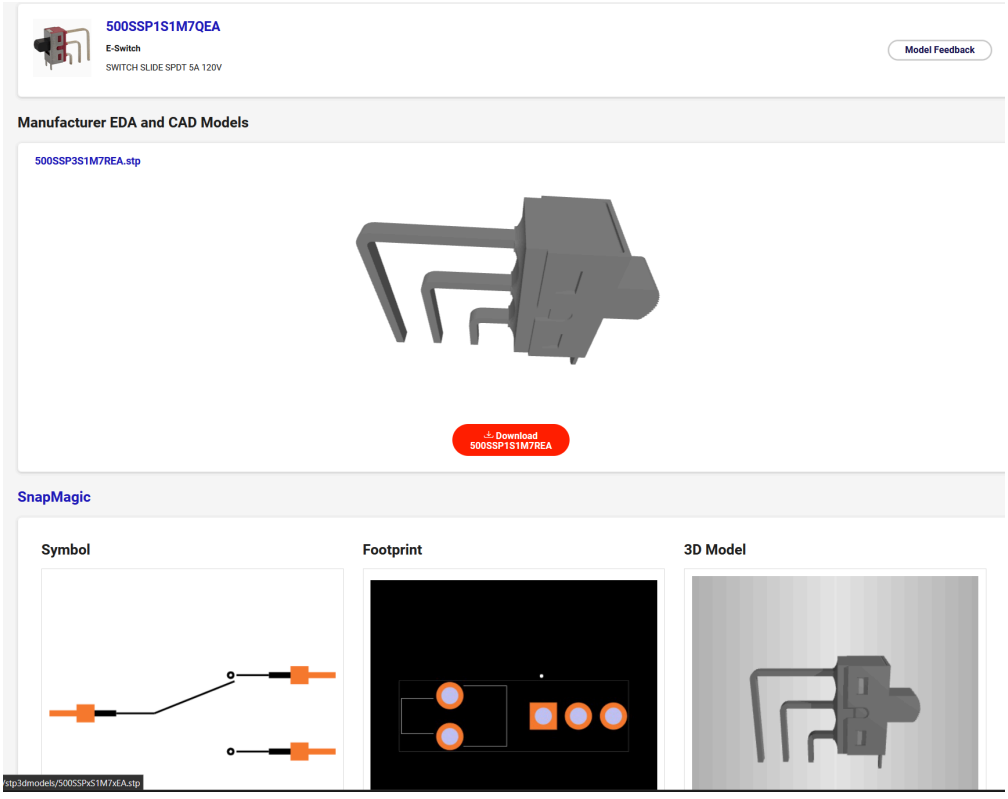


Image shown is a representation only. Exact specifications should be obtained from the product data sheet.

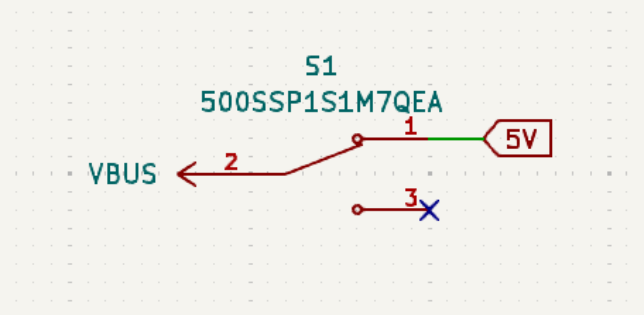
500SSP1S1M7QEA

DigiKey Part Number	EG2480-ND
Manufacturer	E-Switch
Manufacturer Product Number	500SSP1S1M7QEA
Description	SWITCH SLIDE SPDT 5A 120V
Manufacturer Standard Lead Time	11 Weeks
Customer Reference	
Detailed Description	Slide Switch SPDT Through Hole, Right Angle, Vertical
Datasheet	Datasheet
EDA/CAD Models	500SSP1S1M7QEA Models

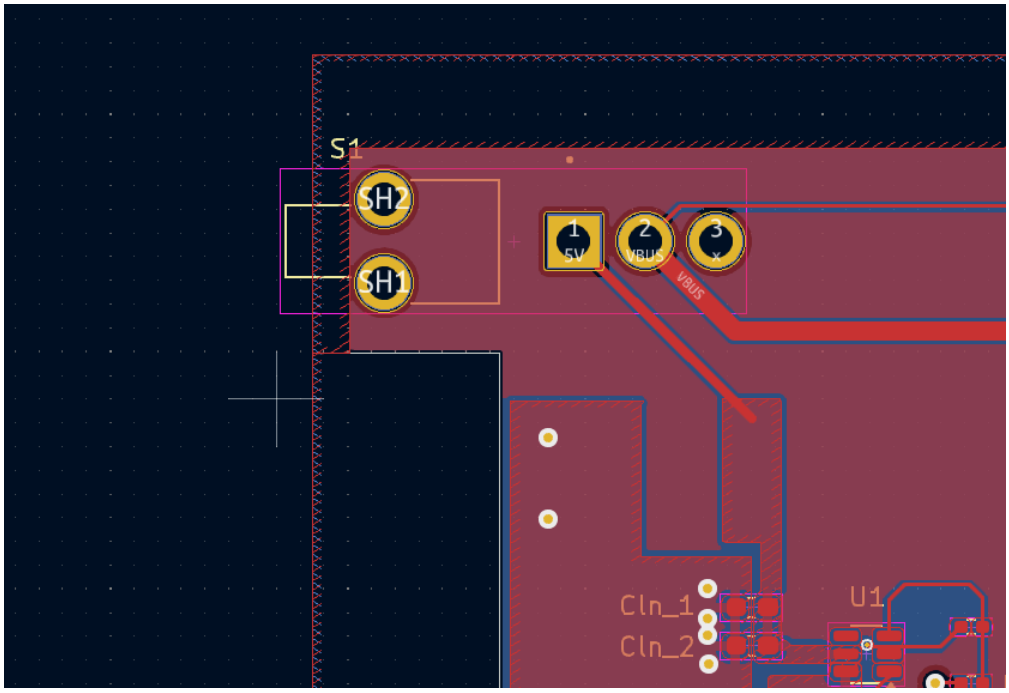
I needed the 3D model for this and footprint so I learned how to get to digikey's footprints page and import it into kicad:



I then added it to the schematic:



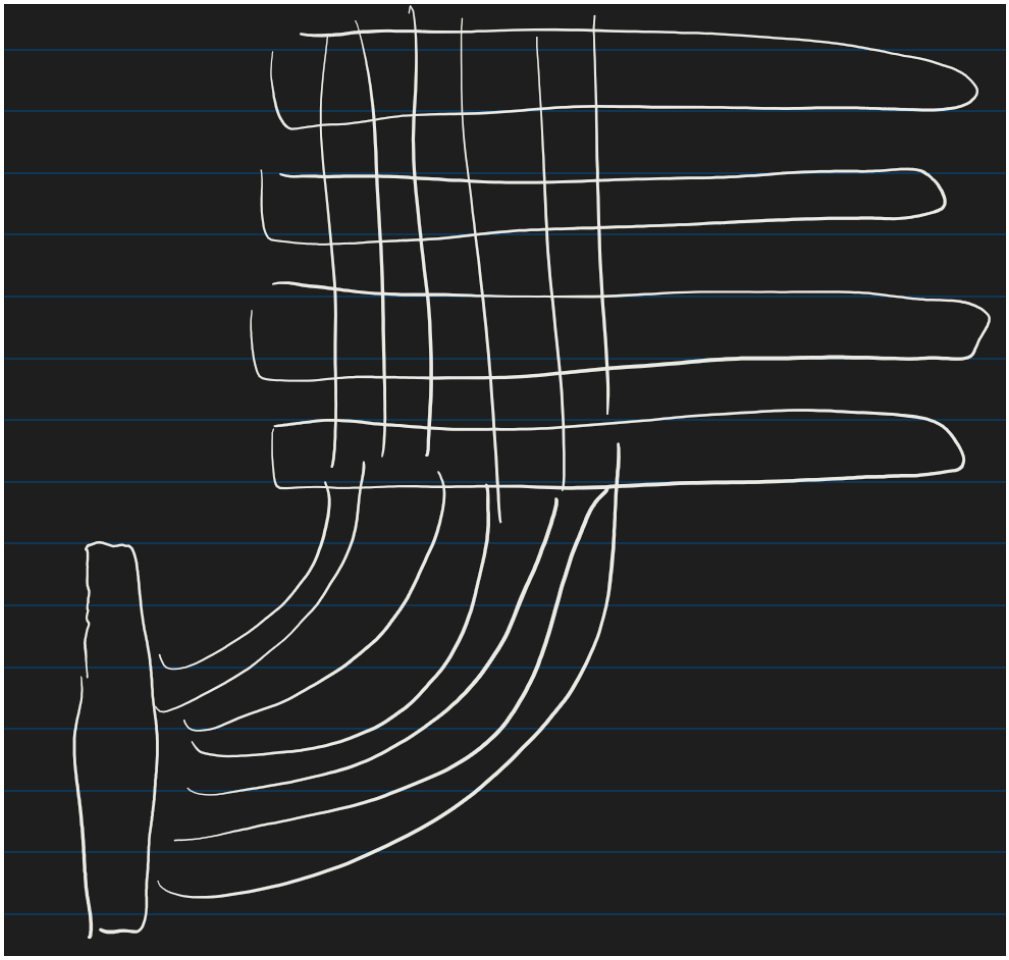
And inserted it into the layout where it needs to go on the left side of our backplane:



The final thing to do was to route the FFC connector to the DIMM slots. Only thing was that Brian was still working on the pin assignments for the front panel and which order they were going to be for the front panel connector, so I waited a bit.

Brian finished, however I realized that the pin order does matter for the front panel but for the cards/DIMMS it does not. We can change the pin functionality in sofwtare.

So I thought, maybe it would be easy then to route them like this:



But I wasn't so sure. Because maybe the card pin order *does matter* because the PIO pins need to be next to each other. I wasn't sure because I hadn't done my own programming in PIO yet so I waited for next morning to figure out what to do about this.

Week 8

Entry 1

3/4 Tuesday

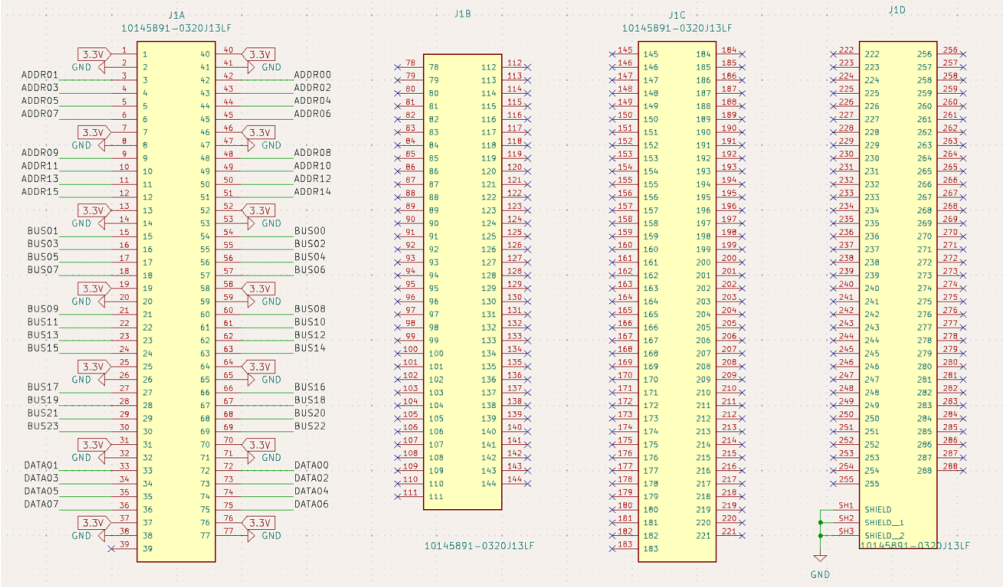
10:30pm-10am

11.5 hours

Today we needed to finish all the pcb layouts and insert them into our presentation that was for the next day.

I realized that we had not started on our backplan PCB schematic and layout so I took to getting that done while my teammates worked on the other PCBs and slides.

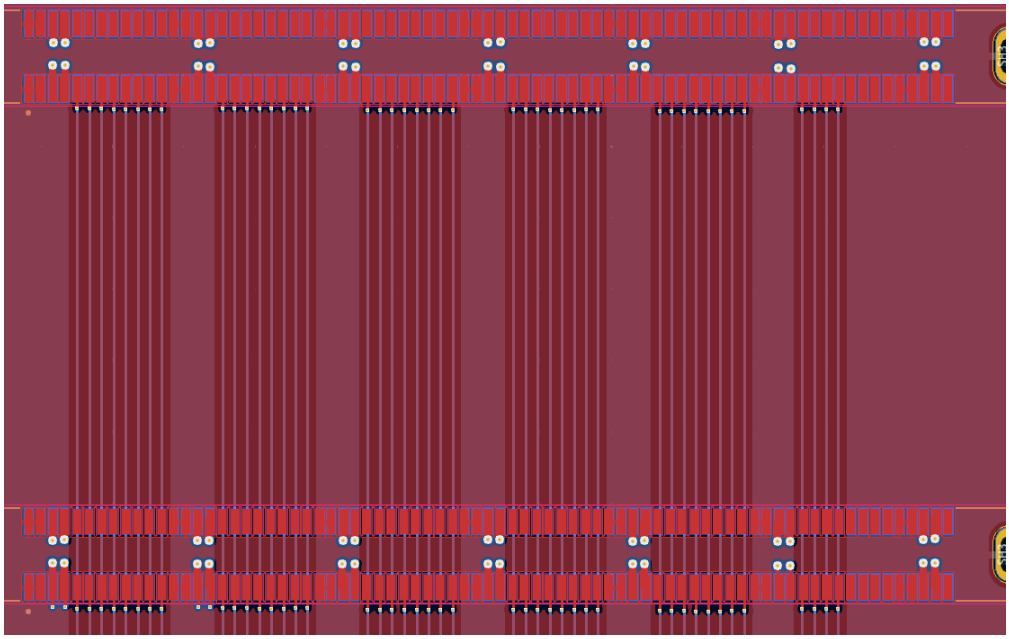
First I needed to figure out how I was going to do the schematic for the dimm slots. Here is what I assigned them nets.



I then copied this 4 times. We chose SMD Dim slots as they can be more easily routed for a bus than through hole ones as I learned from an earlier layout experiment.

Then I went to PCB layout for the bus.

I routed them in the below fashion where bus signals are viaed to the bottom layer and all share a bottom copper strip. Power and ground which are after every block of signal pins, get zone-linked either for the front layer (for power) or through vias to the back layer (for ground).

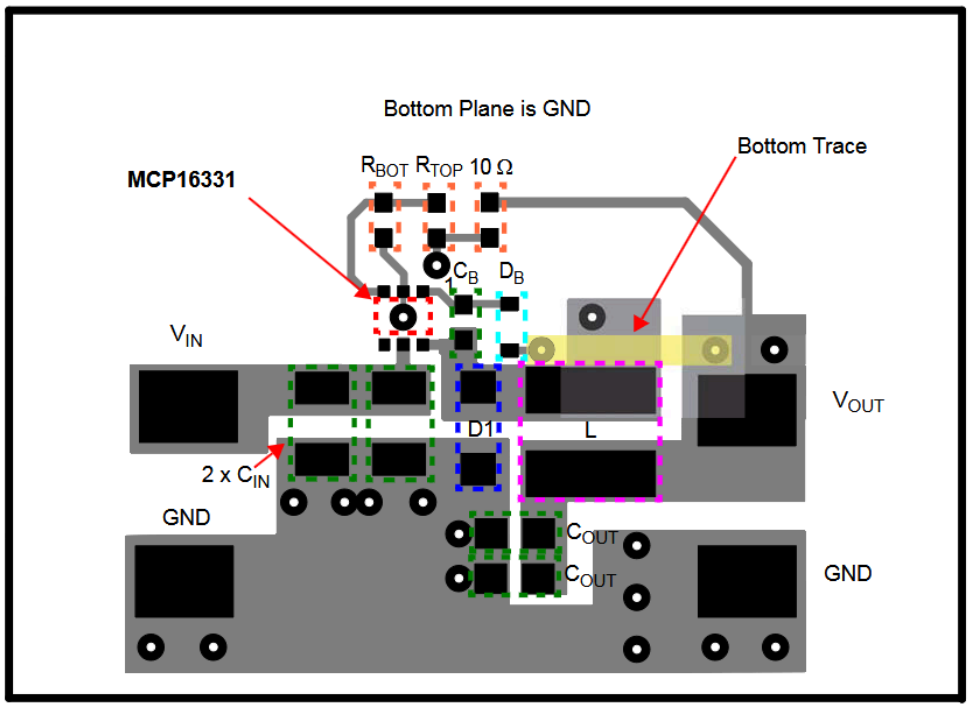


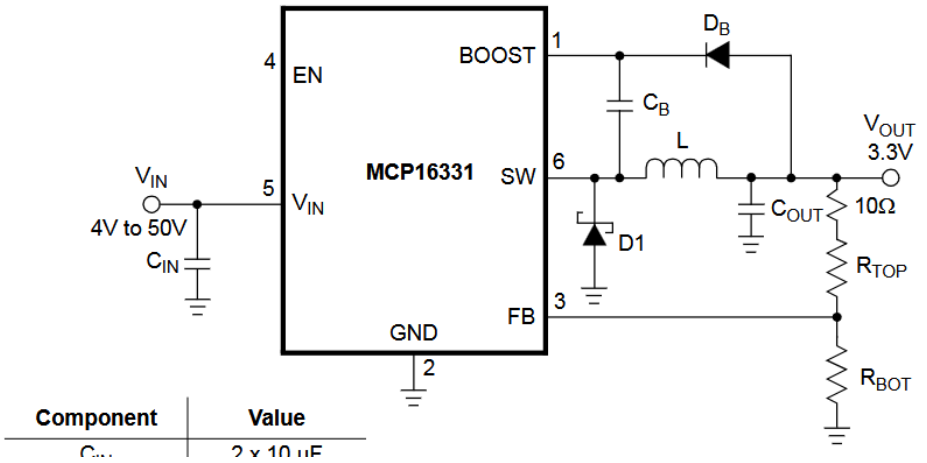
As I did this, Brian showed me how to use KiCad. I learned all the key shortcuts, how to import new footprints and models, how to set a custom grid, and how to make zones.

After laying out the DIMM slots, I proceeded to research the buck converter and recommended layout to place on the backplane.

I found the docs here and I followed the schematic and layout given on page 22 for the 500 mA design.

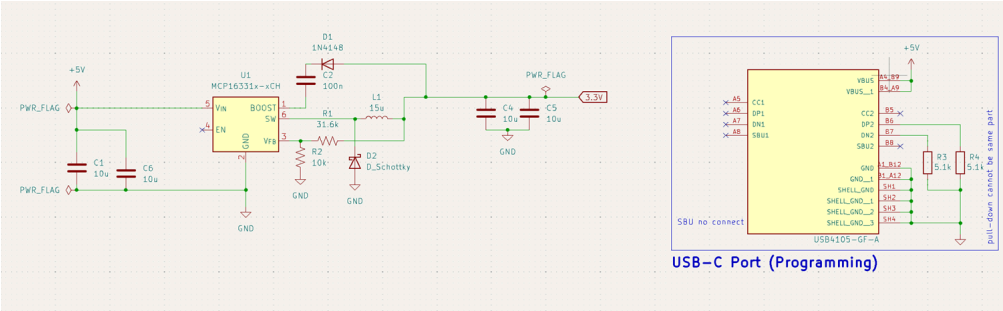
Here was the reference:

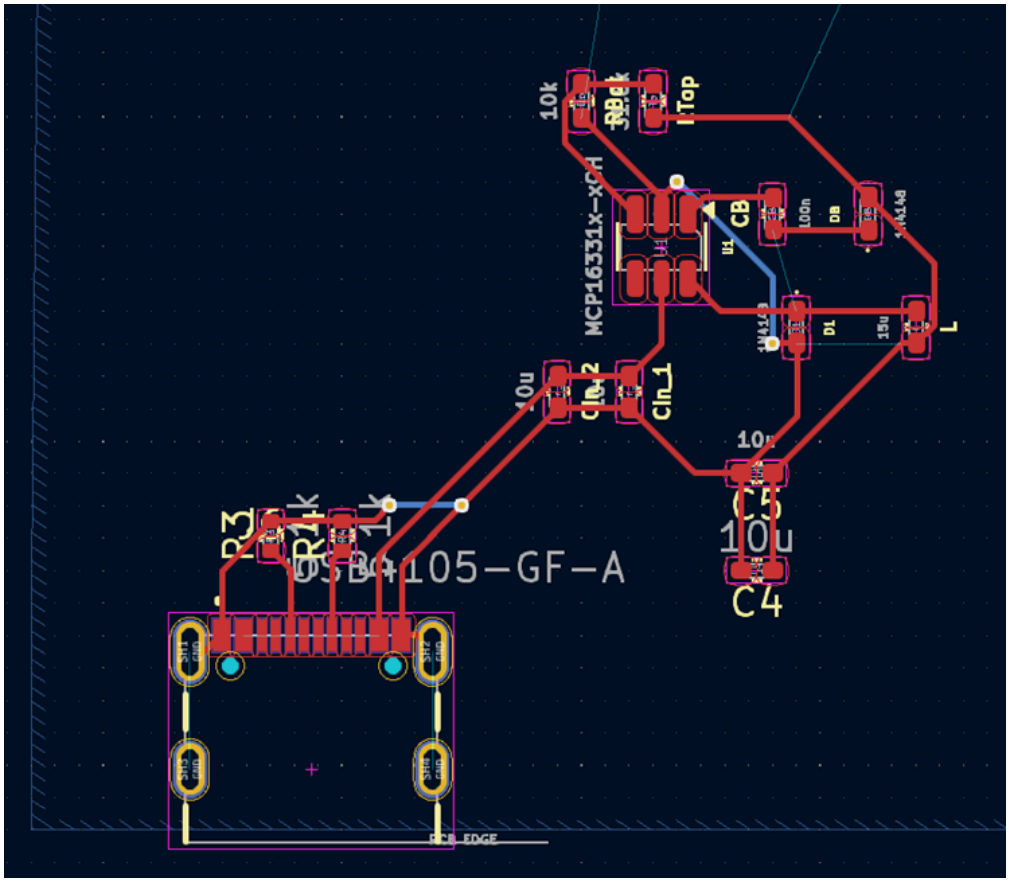




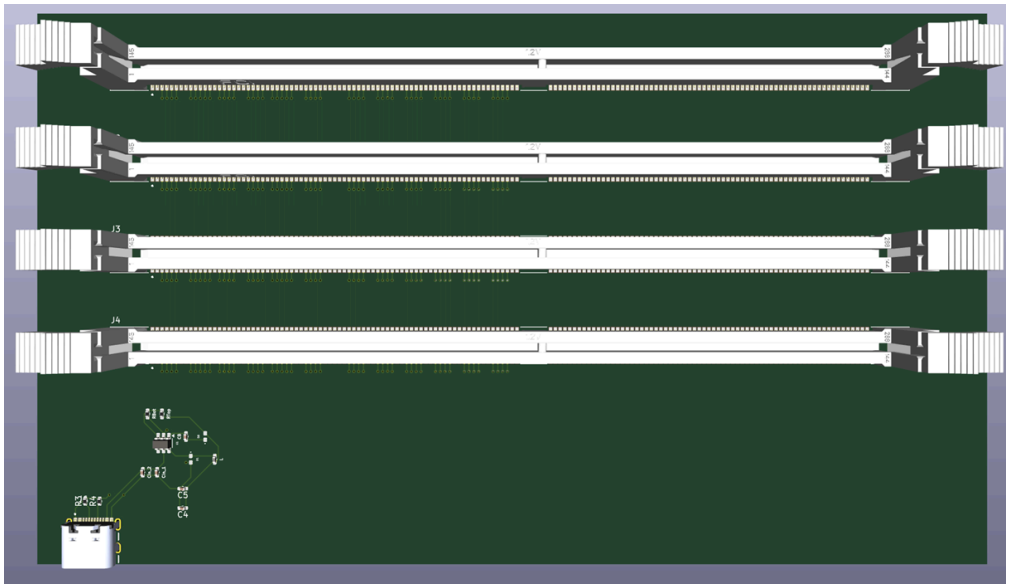
Component	Value
C...	2 x 10 uF

And here was my implementation with the USB C port as well



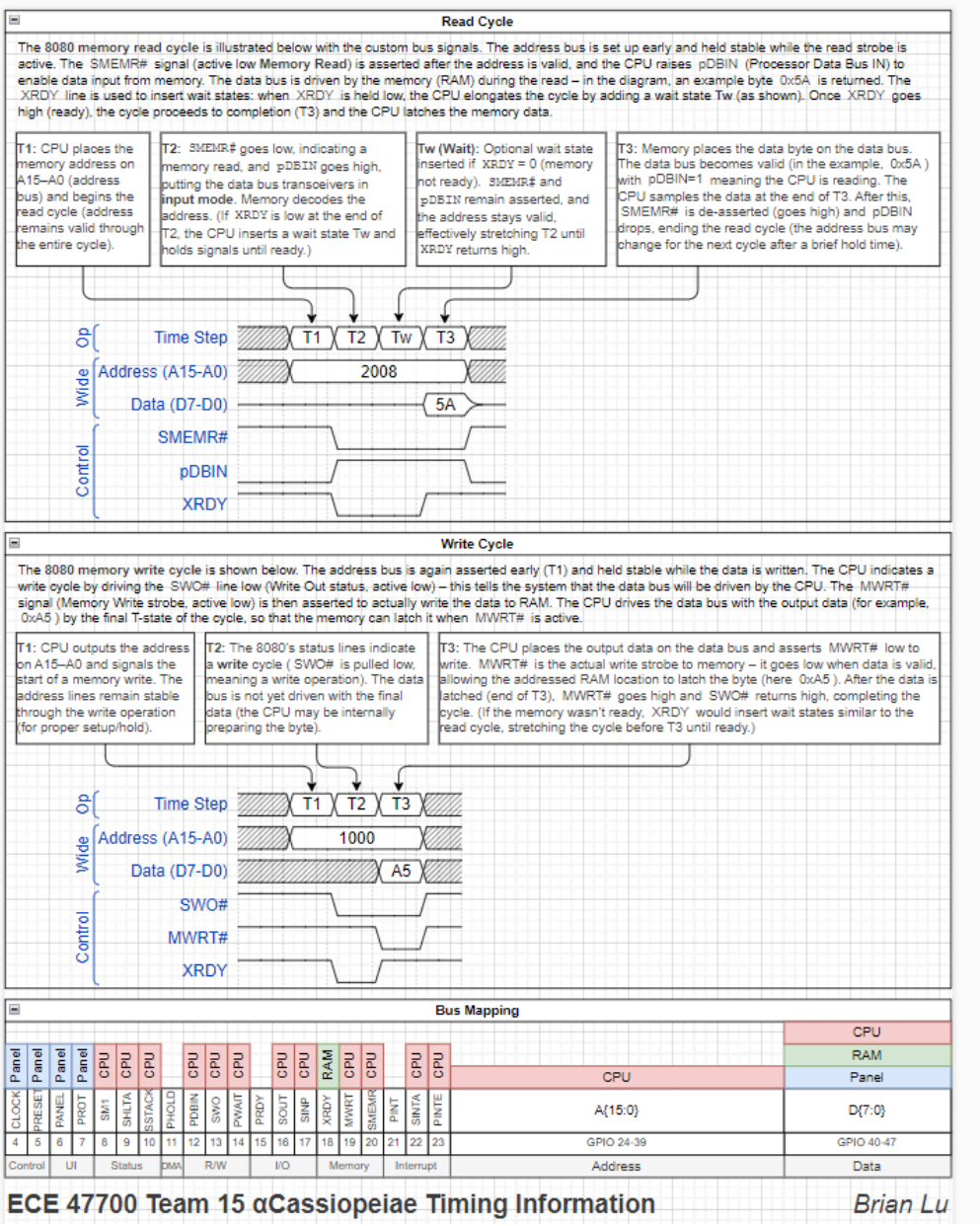


I then verified everything looked good in the 3D model



Now that the backplane first prototype was done, I switched back to researching software since we needed a bit more for our slides

Brian made a document highlighting how our bus read and write transactions are going to work so i reviewed that:



This gave me a better idea of what bus signals are relevant to memory operations

I then added this info to our presentation and also listed all the memory communication transactions that would need to be made between the CPU, memory, and front panel.

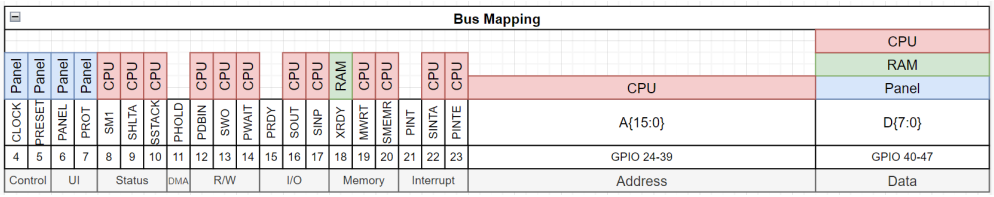
As part of realizing the front panel needs to communicate over the bus as well, I talked and learned from brian that whenever data is written to memory from the panel input, the panel itself needs to send over 3 cycles to complete a full write machine cycle. This is why our front panel needs a microprocessor and why it's not always a reflection of the bus and the switches alone cannot handle writing to memory.

After I learned this, I was curious how the deposit (write) and examine (read) process worked so I rewatched this video explaining how the altair 8800 is programmed again. This got me familiar with the examine and examine next switches and how to operate the whole thing so I now know how to better implement the software. I documented my findings about how a full bus transaction works on our slide like this:

Clock always running
Single step just triggers a flip flop to have the pc attach to the main clock.
Examine sends a jmp instruction (jmp, upper, lower)

1. Flip On switch (deliver power to all PCBs, microcontrollers start, set front panel = stop, CPU pc = 0)
2. Program first instruction
    1. First byte
        1. You're already in data mode. Flip the switches to change the data line. Lights do not change
        2. Hit deposit. Data lights should change to switch values.
    2. Second byte
        1. Flip switches to change data line
        2. Hit deposit NEXT. Address lights flip to mem address 1, data line reflects switches.
    3. X byte …
3. Examine byte
    1. Flip switches to address you want to examine
    2. Hit Examine. Data lights should change.
4. Running
    1. Hit Stop, Reset (goes to mem location 0, reset does not clear memory)
    2. Hit single step. Will execute machine cycle

I was still curious about all the bus signals we had and I wanted to make a better version of this diagram

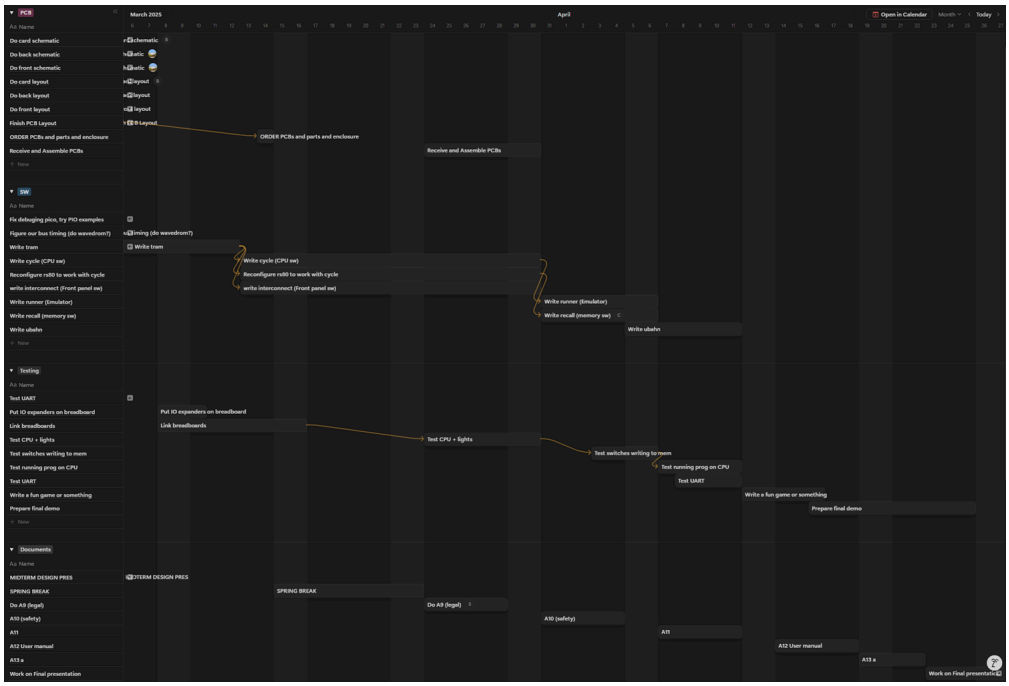


that was vertically oriented and also included the description and category so I made this:

GPIO #	Type	Name	Description	Ownership		
4	Control	CLOCK	bus clock	Panel		
5	Control	~PRESET	reset	Panel		
6	UI	PANEL	front panel data control	Panel		
7	UI	PROT	memory protection	Panel		
8	Status	SM1	first machine cycle of instruct	CPU		
9	Status	SHLTA	halt	CPU		
10	Status	SSTACK	stack operation	CPU		
11	DMA	~PHOLD	hold bus control			
12	R/W	PDBIN	data bus in	CPU		
13	R/W	SWO	write-out	CPU		
14	R/W	PWAIT	mem/io	CPU		
15	I/O	PRDY	io ready			
16	I/O	SOUT	output IO cycle	CPU		
17	I/O	SINP	input io cycle	CPU		
18	Memory	XRDY	memory ready	RAM		
19	Memory	MWRT	memory write strobe	CPU		
20	Memory	SMEMR	memory read strobe	CPU		
21	Interrupt	~PINT	interrupt request			
22	Interrupt	SINTA	interrupt acknowledge	CPU		
23	Interrupt	PINTE	interrupt enable	CPU		
24-39	Address	A{15:0}				
40-47	Data	D{7:0}	tri-state	CPU	RAM	Panel

and added it to the presentation.

The final part of the presentatoin that needed to be completed was our timeline. I already had setup our project timeline in Notion, I just needed to add more events to it and adjust it for the end of the semester. I figured out that Spring break will cut off an entire week however, we can still get all our software down if we complete our packages around 1 a week and we will have 2 week of room at the end in case.



Next steps will be to receive presentation feedback and continue finishing our PCB designs for ordering next week.

Week 7

Entry 5

2/28 friday

10:30-1pm

2.5 hours

Reading the pio spec in the rp2350 manual. Talking to caleb about how rpa vs rp b pins are specified in the hal project.

Then I continued to try to understand the pio async example program. I learned about all the types of directives in pio assembly code like `wrap` and `wrap_target` and what delay does.

I also learned that side stepping is how you can control multiple pins with a single state machine.

Our microcontroller consists of:

- 3x PIO Blocks
- 4x state machines per block
- 5x GPIO pins per state machine

So we can control about 5 pins per instruction. But then the manual also notes we can shift 30 bits in and out over pins so I think we can do a combination of this. We need to think about this so that we can determine how many state machines we need and what state machines will be controlling what pins on our bus

Here is a good diagram of the layout of the state machines:

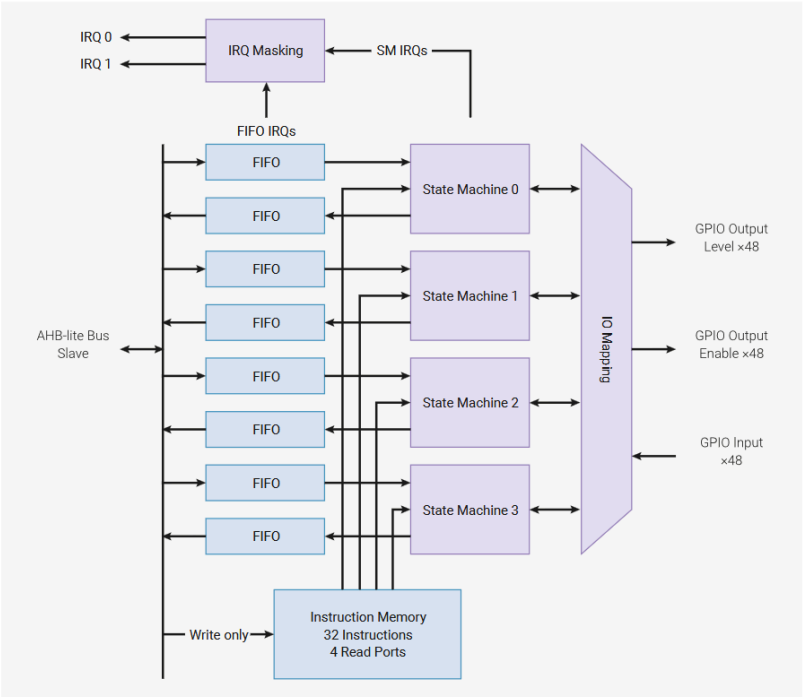


Figure 43. PIO block-level diagram. There are three PIO blocks, each containing four state machines. The four state machines simultaneously execute programs from shared instruction memory. FIFO data queues buffer data transferred between PIO and the system. GPIO mapping logic allows each state machine to observe and manipulate up to 32 GPIOs.

Next steps are to write my own pio program for the bus using the programming knowledge I've learned about the PIO standard.

Entry 4

2/27 Thursday

9:00-11am

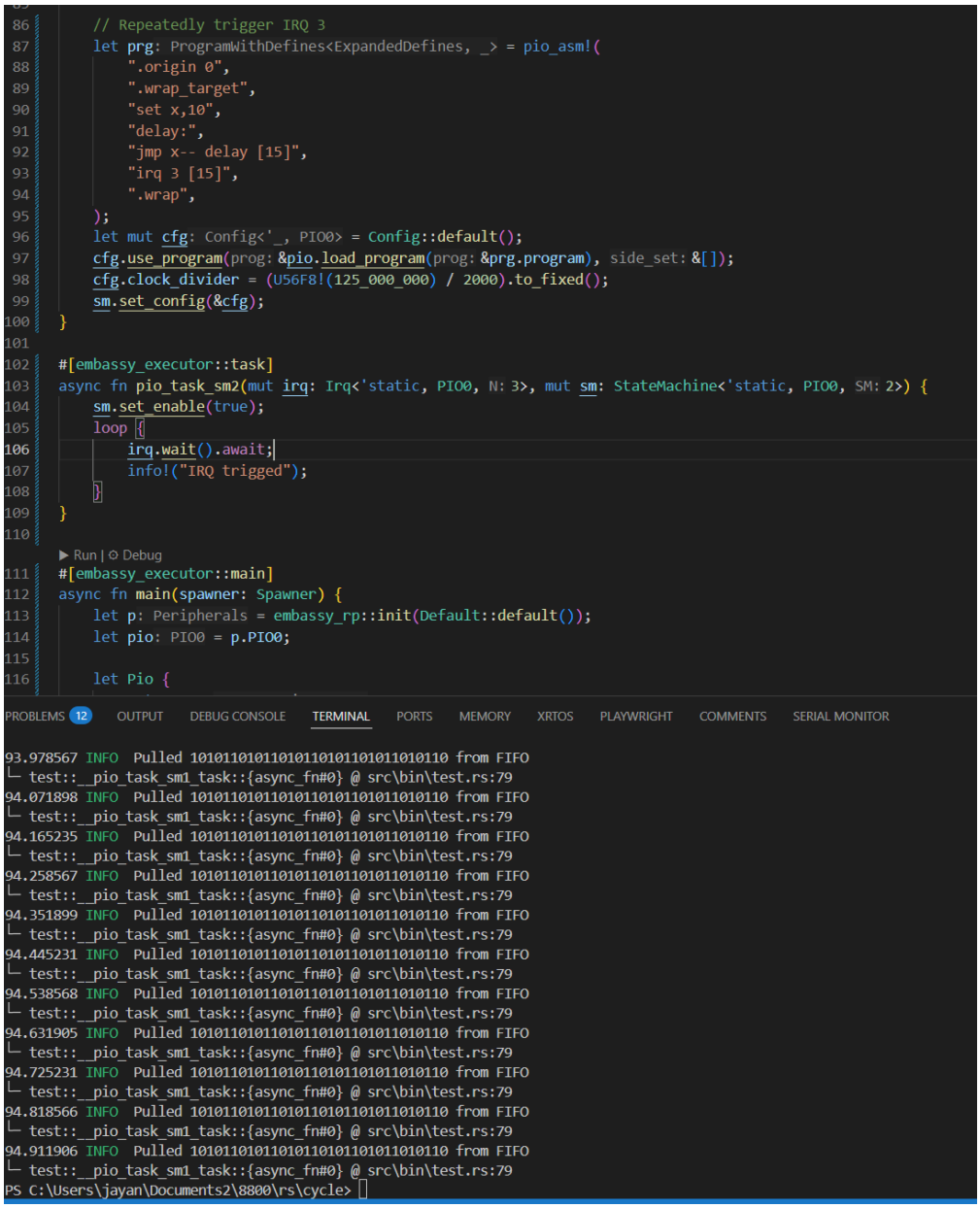
2 hour

Ok had debugging problems where embassy doesn't want to consistently program.

Got that resolved and helped caleb get onboarded with rust and probe-rs so he could begin programming the rp2350.

I had brian try debugging like I did and he also ran into the issue of the lines not highlighting and the breakpoints not triggering. We were thinking it's probably a rust compiler thing that the breakpoints are not working as expected. We could look into compile flags to maybe fix this, however, we already have uart and defmt (a rust logger for embedded devices) setup and that should be good.

Right now i am reading on pio and trying the async pio example from the embassy examples repo. The example will help me figure out how to configure pio and how it works. I'm also using copilot and the rp2350 datasheet to understand the code and capabilities of each PIO state machine.



Testing the PIO standard with the async embassy example. Terminal shows it is working on the microcontroller

Next steps are to continue reading about PIO until I fully understand the standard.

Entry 3

2/25 Tuesday

11:30pm-5am

5.5 hours

Okay so brian brought another pico and plugged it into the board. First i verified we could program both boards with both picos. From this, I learned one of the picos did not have picoprobe on it, so I quicly flashed that uf2. Then I verified it could flash a program using probe-rs like pio-dma and it did successfully.

I then worked on getting UART over the debug probe to work. I wanted to do this so that we could have one pico send signals over the bus and the other pico confirm that it recieved those signals by sending a message of UART.

I first looked up the pico debug probe wiring again in the pico getting started manual and found what RX and TX pins it expected. I then looked through the rp2350 datasheet to find which pins were available for UART from the GPIO functions table. I decided on pins 42 and 43 as the lower ones were closest to the pico they had to connect to on our breadboard so it was easy to wire.

After wiring, I opened the UART program on rp-rs-hal examples repo changed the GPIO pins to be used for UART, and tried programming. When it was building it ran into the error that it didn't recognize pins 42 and 43. I looked at the internal code for the package and realized that GPIO pins 42 and 43 were not listed as UART pins. I then realized that the HAL only supported the rp2350-a (60 pin varient) and not the rp2350-b (80 pin varient) that we were using that has the higher pin numbers. This Github issue confirmed this lack of support. I could add the pins myself but i didn't want to waste time. I remembered embassy was another rp HAL we could use to program our microcontroller with rust. I check the pins support and it did support the rp2350b model. We were in luck! I setup our tram project to use the embassy dependencies and added the uart example and programmed the board with it. I ran into a pin error again but I just needed to change one of the embassy-rp package features from "rp235xa" to "rp235xb".

It looked like it flashed, but when I opened the serial monitor, I wasn't seeing anything on my computer. I then tried adding breakpoints, a probe-rs launch.json file to my vscode workspace, and debugging, however, the breakpoints were not being hit. I think the debug support for our microcontroller isn't fully there yet since it was just added recently. This issue and PR might clarify. we may need to install gdb, or use openocd for a debugger instead.

To make sure the uart is working, I'm going to add led flashes when it is sending messages. I looked at the blinky embassy example to do this in my own code.

I eventually got serial working by switching up some wires and saw it on the console so no LEDs were needed!

```rust
//! This example shows how to use UART (Universal asynchronous receiver-transmitter) in the
//!
//! No specific hardware is specified in this example. Only output on pin 0 is tested.
//! The Raspberry Pi Debug Probe (https://www.raspberrypi.com/products/debug-probe/) could
//! with its UART port.

#![no_std]
#![no_main]

use embassy_executor::Spawner;
use embassy_rp::uart;
use {defmt_rtt as _, panic_probe as _};

#[embassy_executor::main]
async fn main(_spawner: Spawner) {
    let p = embassy_rp::init(Default::default());
    let config = uart::Config::default();
    let mut uart = uart::Uart::new_blocking(p.UART1, p.PIN_42, p.PIN_43, config);
    uart.blocking_write("Hello World!\r\n".as_bytes()).unwrap();

    loop {
        uart.blocking_write("hello there!\r\n".as_bytes()).unwrap();
        cortex_m::asm::delay(1_000_000);
    }
}
```

The embassy-rs UART example that successfully printed Hello World on our serial monitor. It is much less code than the rp-rs package UART example.

Our next step is get PIO working instead of UART as that is the communication interface we will use to build our bus.

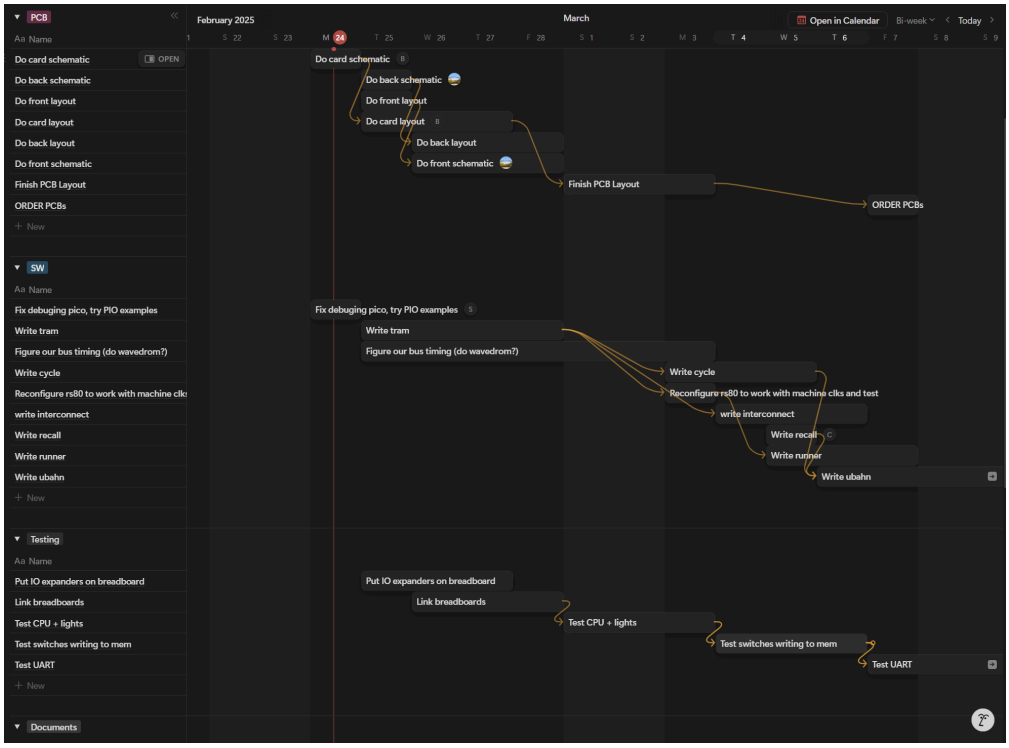
Entry 2

2/24, Monday

9:30am-12:30am

3 hours

First I created a notion timeline view in order to track all of our tasks that are remaining in the semester to complete our project and remain on time. Prior to this we all had an “idea” in our head what needed to get done and when and Brian was “confident” that we could get the project done in time. This didn’t sit well with me, so i created a timeline and list of all tasks we need to do. Here it is:



Here is the link to our todo: 17 Calendar

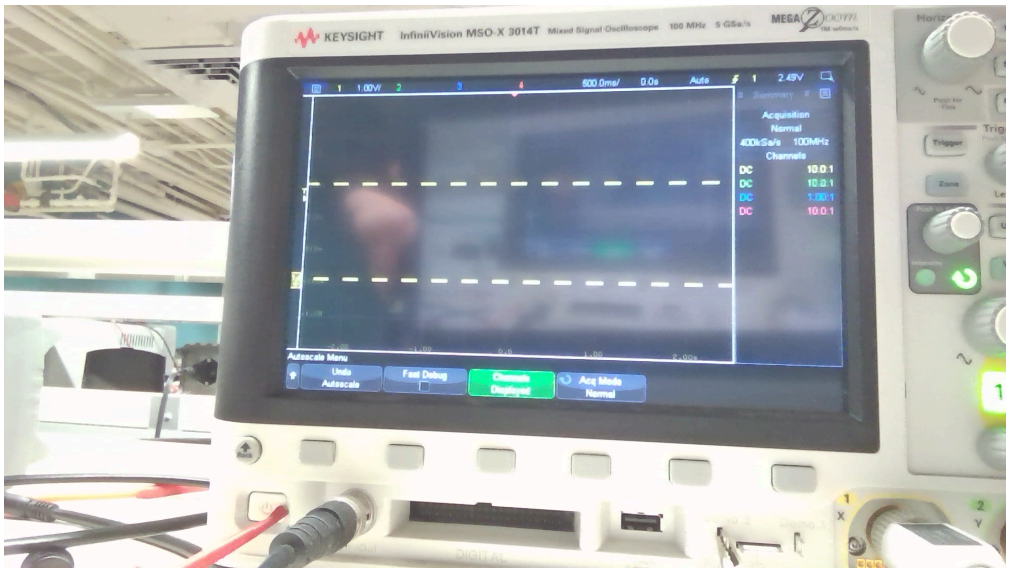
After the timeline I got started on researching the bus signals

In the meantime I explained to caleb how our sw packages are going to work together and how to flash the rp2350. I also created the interchange package (package for our front board).

Now i’m switching over to fixing the flashing for one of our microcontrollers. it wasn’t working last time i checked. it could connect but not flash.

Ok got it to flash again. Also sometimes the device is locked so i need to hit reset before programming it.

Here is PIO based blinky working! so we know the pio works and how we can use it to send signals using Rust



Verified that the PIO blinky works the same on both of our boards.

Next I verified that the PIO R/W blinky works. Then I'm going to transform this program to run on one stamp ouputting and recieving on another to simulate a bus.

Next step though is to test UART as I think that will be an easier cross-communication example to setup than a full transactional PIO bus.

Entry 1

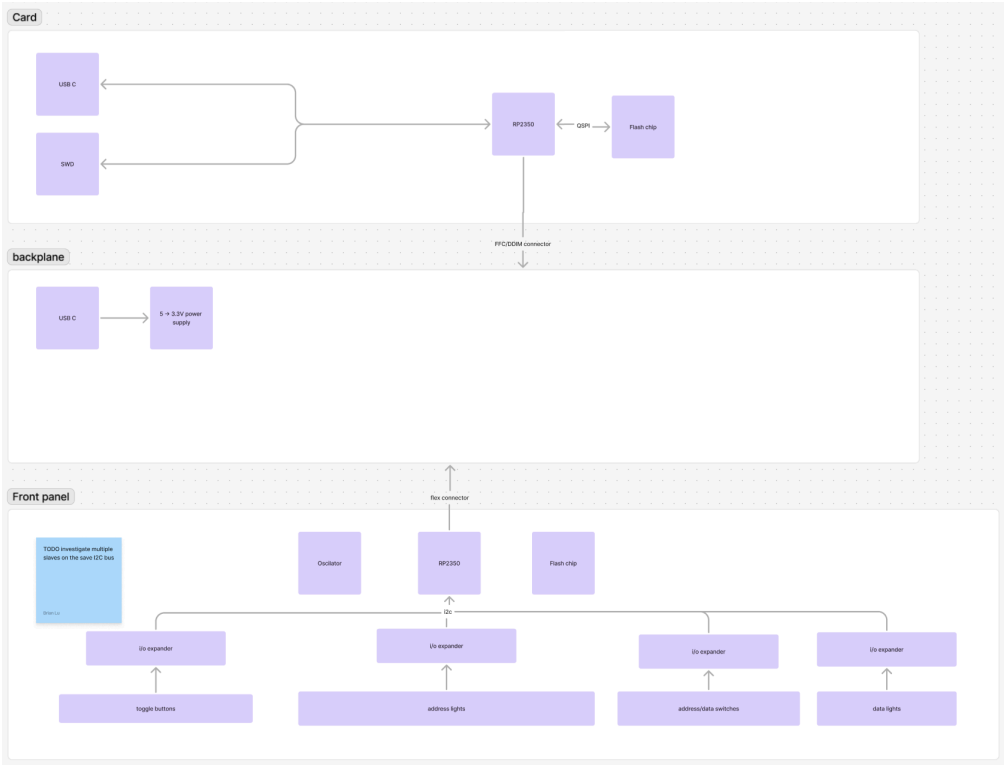
2/22, Sunday

12:00 am

2 hours

Today I talked with brian about our general software and hardware stack since I was writing A8.

We clarified how we were going to do the software layout and exactly connect the packages we made to link the bus with the card-specific packages. While doing this I got clarification on the hardware we are using and decided to make a Figjam diagram showing our layout from a top down level as this would help everyone in the team know the components or our project and interconnects:



Link: https://www.figma.com/board/mlreLoKpSHAxyEdUPpXdqf/CASS?node-id=0-1&t=44ofTralHgVXIb8m-1

I then added this to our website for reference as well.

I finished drafting A8 with making a testing plan with Brian. We are going to focus on making and testing tram, our bus package first, and then all other cards later. E2E tests are the most important for us.

Week 6

Entry 4

2/21, Friday

3 hours

trying to understand microcontrollers and gpio again. I don't remember anything from 362 😭.

I used this resource to do so: https://embeddedartistry.com/blog/2018/06/04/demystifying-microcontroller-gpio-settings/

Push/pull:

- Inputs params
  - Pull down: default input 0 and connected to VDD
  - Pull down: default input 1 and connected to GND
- Output params
  - Push/pull: pin drives high or low
  - Open drain. Normally pulls low but usually you pull high so the line is always high and when a signal writes to the bus like with i2c (pulling it down), all the devices on the line pull down.

So relearned i2c so can get an idea how s100 bus works. https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/

—

I learned from brian that our addr line always asserted by CPU, so we don't need pull up reisistors.

dedicated functions like i2c and uart exist for gpio pins because they are high speed and can't be done in sw like bit banging. our bus is probably going to be lower speed then. pio can help too of course

—

Learned about the ahb crossbar from ChatGPT. allows multiple peripherals to talk to cores at the same time.

Also asked chatgpt about mutexes and hardware spinlocks. Our microcontroller has 2 cpus, and uses these cross-cpu communication channels to manage access to shared resources. We likely won't need these for our project as our entire program can likely run on one CPU, but it's good to learn this and know what it is capable of. This microcontroller has a lot more features than the STMF0 we used in 362.

—

Reviewed how AHB busses work because I want to know how multi-master buses work inside of the microcontroller and how they compare to busses we implement outside of the microcontroller.

This made me review how we did in 337. I referenced the AMBA guide: https://www.eecs.umich.edu/courses/eecs373/readings/ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf

Then I realized PIO is a thing and that we aren't going to be bit banging like I had planned to do to test out a basic bus connection between the two microcontrollers. So to learn about PIO, I watched this video which introduced PIO:

https://www.youtube.com/watch?v=yYnQYF_Xa8g

I learned all of the basic PIO configuration options for the GPIO and the mini instruction set the PIO blocks have as well.

Finally, I went back to reading the rp2350 datasheet and began reading about all the GPIO functions available on each pin and the registers that configured them.

This will help me understand how to configure the gpio pins for the actual bus (liek pull down, open drain, etc)

Next steps are to continue reading through the rest of the microcontroller datasheet, and read and implement the PIO examples so that I can get the basic bus working.

Entry 3

2/19 weds

1 hour

10am

Today I needed to figure out why the probe-rs debugger session was not highlighting lines when I added breakpoints and it stopped on them.

I retried debugging restarting vscode but still didn't work. I then tried debugging with python and saw it works fine. idk why i doesn't doesn't work with my c or rust projects. I leave it as it is. I'm still able to know what line it is by clicking on the sidebar so I should be fine.

The highlights were to help me debug faster and see the registers. Next steps are to implement the code for the bus and use debugging to make sure signals are being asserted and recieved.

Entry 2

2/19 weds

3 hours

I worked with brian to get the pico debugger working.

We tried doing debugging using OpenOCD vs probe-rs which we were trying before and the pico could not find the device still.

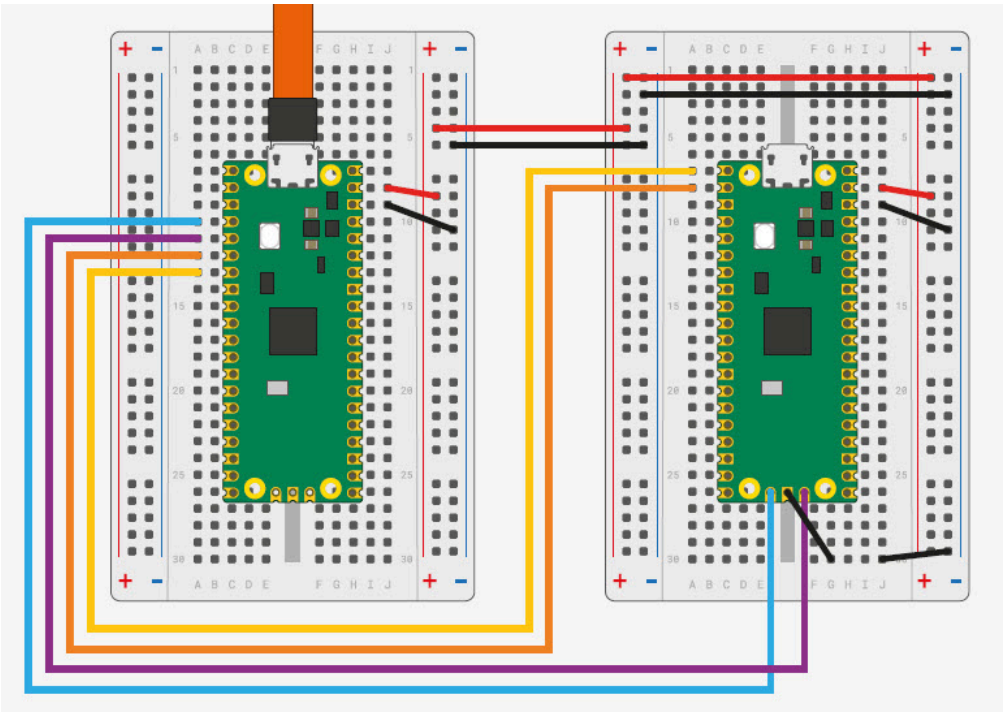
I tried connecting the pico to the stamp by connecting the swd clk and io and gnd pins from both, however the pico could not find the device.

We then tried monitoring the pico signals on an ossiloscope so that we knew it was sending data. That's when I realized we might not have the right pins being used.

I then looked up how to use the pico as a debugger and I found this guide:

https://mcuoneclipse.com/2022/09/17/picoprobe-using-the-raspberry-pi-pico-as-debug-probe/

I saw we need to wire the pico this way to act as a debugger



After rerouting the wires it worked for a bit but then stopped. We needed to readjust the contact between the jumper wires and the board so we soldered the jumper cables to the stamp. This made sure the contact was good. We tested the contact was secure using the multimeter conduction test.

Below is the completed setup

Next steps are to jumper all of the pins together to setup an actual bus between the microcontrollers and then write and test the bus code.

Entry 1

2/17 mon

3 hours

Stamp boards got finished this weekend so I want to start programming and using them instead of the pico. I first explored the stamp schematic to see what pins and stuff were available. There was usb device/host.

I then plugged in the stamp into my computer. I noticed the default things available were a circuitpy thing. I saw circuitpy is a python library you can use to program stuff and the stamp xl carrier board is a supported board for the library.

I just wanted to flash a simple blinky program and use one of the LEDs on the board. I saw one LED was available on the Stamp XL board however this just seems to be an indicator LED and not one attached to a GPIO. Then I looked at the pins for the carrier boarda and saw an LED was available over GPIO 3. So I loaded up my rust RP2350 blinky project and changed the LED pin to GPIO 3 and flashed it holding the boot button. It worked!

Now I'm looking at how to use one carrier + stamp as a debugger, and the other as the board to be flashed. or i could use the pico we have as a debugger. I might do that instead since we want to use each board as ones with programs in the future.

ChatGPT pointed me to using the raspi debug firmware which is usually flashed to the raspi debug probe: https://www.raspberrypi.com/documentation/microcontrollers/debug-probe.html

I saw how the debug probe has UART for serial debugging like printf and then swd for actual debugging breakpoints and stuff with gdb. I also saw that openocd is what it recommends as a debugger, however I'm going to use probe-rs. I recalled probe-rs didn't support the rp2350 chips yet, however I checked again and the latest release does!

I got probe-rs up and running and it recognizes the debugger loaded on the pico! However, when I tried to flash the program from my computer over the debugger to our microcontroller, it said it couldn't find the board. So I need to fix this later so that we can flash over SWD.

This debugging capability will allow us to flash programs faster than usb and thus develop faster.

Next steps are to fix this connection issue.

Week 5

Entry 5

Date: Februrary 14

Start Time: 1:30pm

Duration: 1 hour

Today instead of getting embassy to work, I went back to researching on how to implement the emulator because I now get how to implement the CPU. You can make the CPU a struct with methods like step(instruction) that will parse the instruction, and modify the registers/execute memory transactions based on the instruction.

Now that I knew this, I then looked through the Intel 8080 datasheet again to look at how the opcodes for each instruciton are formated. As I read, I realized that the opcodes were actually the same for multiple different instructions and that only register values of a different arrangement of bits differentiated them. 8080i is not like RISC-V where the opcode is the entire instruction. This means that I will need to write a parser that will take the opcode and the register values and determine what instruction to execute based on the opcode and the register values.

After understanding this, I then looked at the rs80 emulator again to see how he implemented parsing the instructions. He had a custom txt file format that stored all the instructions and their format and what rust code to execute for them. This txt file was then ingested by a separate package from the emulator that was responsible for code genning all of Rust code to handle the instructions. As he notes in his documentation, there are "256 possible 8-bit opcodes. Instead, the input file contains 59 instruction definitions which are processed and specialized." This is a great approach and probably what we should do for our project. In fact, how he implemented the emulator is exactly what we should do for our project so I then freaked out a bit how we could possibly make our emulator different.

After talking with Brian, Brian reminded me that the rs80 emulator did not handle memory wait, waiting for actual memory to respond and that in our emulator we would need to do that.

Therefore, I came up with my next steps which will be:

- I think I'm just going to get the Emulator programming on the Pi
- Run and hook up the RS80 simulator to our embedded code
- Simulate bus signals on the GPIO pins
- Hook it up to the pins and LEDs on the board
- Then use the IO expander we have to expand the signals like we will do on our final board.
- Modify the rs80 code to handle memory waiting stages so that is is machine-cycle based and not instruction-cycle based.

And that's our software development plan!

Entry 4

Date: Februrary 12

Start Time: 12:30pm

Duration: 1 hours

As a continuation of this time on the 12th, I asked ChatGPT a variety of questions about RTOSes and how interrupts/allocating CPU time between proceses work to better understand how process switching works in operating systems. This wasn't exactly relavant to our project, however it gave me a better perspective on how interrupts work and when they are executed so I knew their differnce with classic polling loops. For this reason I think we will do interrupts to step through/run/stop
the CPU.

Thy questions/answers:

does px4 autopilot use an rtos when it runs on a pixhawk? yes nuxtt os

what's happens between when a person clicks a button on a keyboard and the program recieves it on linux? what syscalls or whatever are involved? x11/wayland, can use poll, read syscalls. poll freezes
execution of a program. can spawn multiple threads to wait for things in
an os.
can you manipulate cpu time, the scheduling method, etc. in linux? yes cgroups
can you cahnge scheduling type in windows? kinda ya you can set priority
can i see the list of processes and their ids in linux? ya

what does %cpu usage mean in task manager technically? isn't the cpuu techincally running at 100% most of the time beause it's running monitoring loops? it's the percent of time the cpu is active with user
requests. most programs just wait for interrupts triggered from syscalls
to resume execution. they don't need to run 24/7. only if you make a
while (1) loop will it run 24/7.
what's the diff between proceses, subprocesses, and ids in windows task manager cpu. ids mean all threads
does rust have a callback queue/event loop like js? no it just spawns threaded runtimes
when you write a driver for linux do you interact with syscalls?
are syscalls the only interface/api to a kernal? no you use kernal
space interfaces. kernal interfadce links hardware to os. syscalls link

os to program. open() syscall will call the open function defined in the
driver. same with read() write().

END Q/A

Finally I learned about memory ICs and broswed digikey on how to find them. I later learned Brian had already selected one, however this gave me the opprotunity to understand what was available out there, the sizes of memory ICs, and the different protocols they used.

My next steps will be to try to get the embassy crate to work with our Pico repository so that we can program our microcontroller with Rust using that.

Entry 3

Date: Februrary 12

Start Time: 10am

Duration: 2.5 hours

Today because I now shifted my focus from coding to emulator to coding for the microcontroller, I decided to create a new package in our main Rust repository dedicated to just programming the Pico. I did this instead of just starting on a Rust repo because the Pico was just our development board and was not going to have the same layout/interface/board abstraction layer as our final board design. I then added the cargo dependencies from the example RP2350 example repository that I used previously to first progam the RP2350 with Rust. I ran into some missing linker files which I added from the original repo to mine, however even after adding those I ran into other linker errors that did not relate to missing files. This is the one I got stuck on:

rust-lld: error: undefined symbol: _critical_section_1_0_release

I think it got stuck because the rp235x-hal package is not correctly built for the version that is published on Cargo.

I was unable to fix this so instead of trying to further debug how to get the rp235x-hal package to work, I researched again if any other Raspberry Pi microcontroller HAL cargo crates existed. I came across this Rust on the RP2350 blog post again and in addition to mentioning the rp-rs project that publishes the rp235x-hal, they also mentioned another project called Embassy.

I looked at Embassy's website and realized it had the unique property of async/await which I was familiar with Javascript. I used ChatGPT to help me understand how async/await worked in Rust vs how it did in Javascript. I learned that in Rust, async/await is compiled to a bunch of state machines and the order of execution/waiting hands idle time over to following instructions. I also learned the difference between concurrent and cooperative running. I also learned what rust runtimes like tokio do. Finally, I learned what RTOSes do and how they differ from regular operating systems. I did this because the Embassy project noted that it was a faster alternative to traditional Real Time Operating Systems at the cost of maybe more code. RTOSes are more lightweight than traditional operating systems and directly use interrupts to handle events and poll for new event priorities maybe every 5 or so insturctions vs time-based CPU slicing which traditional OSes do.

This was all of interest to our project because I wanted to know if async/await or RTOS would be a better/easier implentation for our emulator/microcontroller over standard interrupts/polling loop. I determined async/await would be okay and that we could proceed with embassy.

Entry 2

Date: Februrary 10

Start Time: 10am

Duration: 2.5 hours

Today I focused on beginning writing the emulator because I thought that would be easier than starting directly on the embedded interface code. Because I have not coded in Rust a awhile, I needed to review data structures so that I

knew how to implement the data structures to hold the registers for the emulated CPU and methods to perform operatoins on the CPU. I began going through the beginning of the Rust Book again. I recalled how structs, enums, mutability, ownership, and implements work. These functions of the language should be enough for me to be able to program the emulator.

I also realized when beginning to write the emulator that we would need separate code for our memory card that was to also operate on a clock cycle and send signals from our bus on the GPIO pins over SPI to a memory module on our memory board. Because of this I started another Rust package in our workspace for the memory and drafted memory get/recieve methods.

Midway through drafting the architecture, I got confused how our Rust package for the emulator was going to react to clock signals. I thought implemting the code needed for our microcontroller from the top down (GPIO signals first, emulator second) would help clear this up.

Now that I switched to thinking about how the emulator would handle clock signals, I was researching whether we should use just an infinite loop to poll for external clock signals or create interrupt functions. This led me to do more research on their differences and review what actually happens during an interrupt routine as we learned in 362.

I ran out of the time for the day so I decided to leave this research of interrupts vs loop for the next day.

Entry 1

Date: Feb 8

Start Time: 9:30

Duration: 1 hour

Today I focused on debugging why the pico was not flashing with example Rust-based code.

I was trying to flash it using the elf2uf2-rs tool, however, I found a Github issue that said the elf2uf2-rs tool cannot flash the RP2350 yet and that I should use picotool instead. So I downloaded picotool and changed what tool was being used to flash in the `.cargo/config.toml` file and ran `cargo run --target thumbv8m.main-none-eabihf` and it worked!

What's annoying is that you have to unplug and replugin the USB every time and hold the BOOTSEL button when flashing in order to get it into flashing mode for UF2. However, once it flashes, the program runs immedeately.

Now that we can program the microcontroller with Rust, we can begin writing the Rust code to listen to bus signals on the GPIO pins as well as begin integrating it with basic emulator code to say enter
instructions with switches and trigger a clock signal with the push of a button. That will be my next step.

Week 4

Entry 3

Date: Februrary 7th

Start Time: 9:30am

Duration: 4 hours

First I updated our PSDRs and functional description based on weekly feedback.

Then I got to trying to figure out how to flash the pico with our Rust project code. I compiled the Pico code sucessfully using cargo and tried to flash using `cargo run` but that failed. I saw it was trying to use probe-rs which I thought I had installed, but it looks like I did not because the command was not available. I then instaleld it and tried running it again but it said no probes could be found. I then realized probe-rs is only meant to work with a dedicated/separate debugger and I would need a dedicated debugger like another Pico or ST-Link in order to use it to flash our Pico.

So because I did not have another debugger on hand, I found in the flasshing methods section of the repo that you can flash in other ways like UF2.

UF2 allows you to just upload a UF2 file to the pico over its USB file system which is triggered when you hold down the boot select. I was able to do that and it programmed sucessfully.

A debugger will be useful in the future, so I've decided I'll bring my st-link from 270 and hook it up to the SWD lines on the pico to try debugging it.

Finally I needed to work on A5 and write the justification for why we chose the RP2350 microcontroller. As part of this process, I realized I didn't have a good grasp on Program IO so I read this article on how it works and how it can be programmed. I then consulted the RP2350 datasheet to further learn how PIO is supposed to be used.

Entry 2

Date: Februrary 5th

Start Time: 10am

Duration: 3 hours

First I worked with Nathan on getting the raspi pico that we had on hand programmed.

First I tried following the raspi pico getting started manual and using the VSCode extension, but I was unable to get the toolchain to download on my computer. So then I had nathan do it and he was able to get it to download and flash.

Next I got started on figuring out how to setup an embedded Rust project and program our Pico with it.

I first created a rust package for "cycle", our emulator, which I wanted to run first. When creating this, I had to learn about cargo and the difference between workspaces and packages. I decided to make our emulator just a package in our monorepo since it may link with other things in the future.

Finally I needed to figure out how to use the repos in the rp-rs org to program embedded rust for our microcontroller. I came across this starter project template https://github.com/rp-rs/rp2040-project-template

I also needed to learn the terms of the other types of repos in the github org like HAL, which is hardware abstraction layer, and PAC which is Peripheral Access Crate. These are the handy APIs/interfaces for manipulating our microcontroller with rust without having to change registers and stuff.

Entry 1

Date: Februrary 3rd

Start Time: 10:15am

Duration: 1 hour

First I watch how a microcontroller starts video to learn how the boot process of the program we flash is set up and how the Rust based flasher we are using will do it as well.

Next I did a bit of reading of the embedded rust book, learning about how embedded rust code typically looks and what data structures/coventions we can expect.

Next I got into a bit of a read about posix/cygwin cause I noticed embedded rust code can run on a variety of environments like bare metal vs. hosted. We will probably only need bare metal for our project.

Finally I looked back into how platformio works since I was wondering if it supported Rust-based microcontroller development. I needed an easy way to get started with the pico we had lying around and I knew platformio was mostly C programming so I anticipated it was not available yet and I was correct. So we're going to have to stick with rp-rs.

Week 3

Entry 4

Date: January 31st

Start Time: 9:30am

Duration: 3 hours

Today I needed to figure out how to convert the Intel hex to binary. ChatGPT pointed me to just writing a simple Rust program to this which I tried.

I realized that I need to place it in a specific part of the repository like as a binary to make the rust compiler happy because the folder I was putting it into was a cargo project and all Rust files in a project either need to be a library or a binary.

I then did ChatGPT's second suggestion of using the `srec_cat` command line utility to compile the hex to binary. I was confused how to get this CLI tool until I learned more how Debian packages work since
I'm working on Debian. I took a little bit of a sidetrack into learning about Debian packages.

I learned that packages can have the same name in different repositories (debian, debian-updates, debian-security) and the package manager will chosse the package with the highest version when upgrading

I also learned about supported architectures for debian and that the official ones, powerpc, arm varients, x86 64 and 32, ibmz all the packages are supported.
So all the packages must be compiled to run on those architectures and the maintainers of distros handle upgarding packages, not the package maintaners.

And then you can have multiple versions of a package on a system and apps on a system can point to different versions of a package they need since they are dynamically linked. And then finally, Rust packages try to all be statically linked (all in the binary) but debian maintainers don't like that so they try to break up the packages.

Although this was a bit sidetracked, this helped me learn about the packaging ecosystem and how Rust works with it. It helped me tie my knowledge about dynamically linked libraries learned in OS right now to my work of how we are going to program and link our programs on our microcontroller.

I then got the actual package name for `srec_cat`, Srecord, and tried converting the hex, but didn't have luck. It seemed like that the assembler was not happy with my output and the assembler was not happy what I loaded into memory and tried to step over the isntructions. I wasn't fully confident why it wasn't stepping correctly and also wondered how the processor knows it's on an opcode vs value vs register/address. I need to save that for next time.

Since I now have have a good idea of how the entire repository works, I diverted my attention to getting our own repository setup. However, I realized that I didn't know really how to start and how our emulator was going to interface with the microcontroller itself and not just a human user through a terminal like the emulator in the rs80 project.

This led me beginning my research into embedded Rust. First I looked through the respositories that Brian had linked that are part of the rp-rs organization that has developed tools to program RP-series microcontrollers with Rust. However, I realized I didn't know exactly what most of the repositories did and that I need more of a getting started guide.

I then looked up some embedded Rust resources. I came across this How to Do Embedded Development with Rust which I began watching. Also began watching this How a Microcontroller starts video because I want to get an idea how the Rust framework programs the microcontroller and how memory is laid out on a Microcontroller. Finally I found this Rust on RP2350 - Raspberry Pi blog post which I will save for later.

The progress today has opened my research up to jumping into embedded Rust and getting started on our software development. I'm excieted to learn how to write type-safe code for our microcontroller and how to interface with the hardware. Next steps are to finish getting through those above resources and begin coding!

Date: January 29th

Entry 3

Start Time: 9:30am

Duration: 2.5 hours

I wasn't able to find the registers available in the original 8080 manual for computations so I found this additional cheat sheet. We could even make our own cheat sheet that is a bit more understandable like this RISC-V one so that future people can more easily program for our emulator.

After understanding all instructions available for the 8080, I then went back to investigating how to run the rs80 8080 emulator Rust repository. I found that these compiled binaries were available:

mon: a debugger that allows you to read and write to registers and load a file into memory and jump to address and then exec

asm: assembler that take 8080 instructions and turns them into
8080 ones doesn't seem to work well though rn with MOV A, M just as text in an input .asm file. Uses the CP/M ASM compiler. The emulator actually spawns the original CP/M compiler on the emulator itself to compile the assembly code you provide on the host system and then spit it back out. I found the manual for the CP/M assembler here which gave me a better idea of its output and how to use it.

disasm: Just a disassembler

main.rs: A main runtime for the emulator for which you are supposed to provide an OS image like CP/M to execute.

From this research I determined that we will need an assembler like CP/M in order to convert our 8080 code to binary.

I also realized that the assembler only spits out Intel Hex and that the intel hex cannot just be inserted into memory using the mon debugger. Intel Hex != binary. Only part of the intel hex is the binary we need, and we need to convert the hex to binary using some tool.

Finally I learned about the tests folder that contains various test .COM (executable 8080 binary) images that are used to verify the emulator properly exeecutes all available 8080 instructions. We should incorporate such tests into our emuator as well.

I will save the rest of converting the intel hex to 8080 for friday. This progress today got me closer to my goal to running the rs80 repo.

Entry 2

Date: January 28th

Start Time: 7:15pm

Duration: 0.75 hours

During this time I attempted to get the rs80 8080 emulator Rust repository created by cbiffle on GitHub running. I di this so that I can get an idea of how we can structure our own implementation of a repository and how to construct a Rust project since I don't have really any experience
using Rust.

To understand how to run the repository, I utilized GitHub Copilot to look at the repository for me and give me an idea of what its contents, produced binaries, and available commands were.

I then realized I didn't have a good foundation of what the 8080 instruction set looked like so I didn't know how I could write an assembly program to run on the rs80 emulator.

I then began reading the 8080 ISA datasheet. I just read through the first few pages and during this time I determined how the two different clocks of the 8080 work, what cycles and states are, and read through the instruction set. I

then learned that opcodes are 8 bits which take up the entire processor bus (8 bits) so it takes multiple states to execute one instruction, contrary to say RISC-V. Learning this helped me understand how I would write our own emulator to process instructions and that we don't need to emulate the entire functionality of the 8080, just be able to process the instructions according to external bus signals.

Next steps will be to continue reading the manual since I didn't get through all the instructions available.

Entry 1

Date: January 27th

Start Time: 9:30am

Duration: 2.5 hours

I read through more of the RP2350 datasheet.

I specifically read through the GPIO pins available and their functions. This was to determine what pins would be available to function as our S100 bus.

I then investigated the bus layout (2.1) and processor layout. This was to get an idea of how the microcontroller interfaced with its pins. Additionally how data was controlled between the RISC-V and Arm cores available and how we could use them if we wanted to choose one or the other.

Terms that I found in the document and used ChatGPT to learn about included the Always-On-Timer and Serial Wire Debug Input/Output. Reasearching these terms helped me get a wholistic view of everything that was available with the microcontroller and how we could use it for our project.

I think I am well-versed in the overall capabilities of the microcontroller we are using. I'm now going to move onto researching how to code the emulator I and Caleb will be responsible for implementing since that is a more urgent task.

Week 2

Entry 2

Date: January 24th

Start Time: 9:30am

Duration: 1.5 hours

I discussed with brian how our computer would load programs

I learned that the front panel is basically useless because it doesn't allow for easy program input while the computer is running and is mostly for just programming the bootstrapper for the microsoft basic OS you were going to load into memory. Then all programming was done via the os on a monitor connected over the uart. could be teletype, etc.

So our computer is not that useful without uart and although it is a stretch goal it is one I want to achieve.

We would want to connect our computer over uart and open a serial console to interface with it.

I also determined with the team that we are going full rust and not programming the 8080 with C. Based on this rust implementation of the 8080 it's not that much to establish the data structures to get the state machines for the processor. I was concerned that because rust has a high learning curve and Caleb who was working on the software with me might struggle, however we believe we have enough time to learn rust and program the 8080 emulator in the semester.

Entry 1

Date: January 22th

Start Time: 9:30am

Duration: 2.5 hours

Today I went through the RP2350 datasheet today (1.5 hours). Learned about how programmable i/o works using ChatGPT (what we're going to use for our bus) and why we're choosing it for our microcontroller vs. others (dicussed with brian).

I attended project discussion/group meeting (0.5 hours)

Looked through the code of the Rust 8080 emulator repository we were looking at that was similar to what we wanted to implement. Determined that we could use this as a test suite to check our hardware if we don't finish our own emulator before.

Week 1

Entry 2

Date: January 17th

Start Time: 9:30am

Duration: 2 hours

I wrote the functional description on the a1 doc as well as drafted the budget

I made and reviewed project milestones with the team

I uploaded bio to website and functional description

Entry 1

Date: January 15th

Start Time: 8:00am

Duration: 3.5 hours

Before lab I learned about the s100 bus today, researched the 8080 instruction set, and watch some altair 8080 programming videos

Then in lab we learned about the lab resources and introduced our team

I then discussed with brian and team what case we want (extruded aluminum) where the pcb slides in on one of the ledges. What box we choose determines our PCB size

Then we discussed what project management software. microsoft planner, notion, github issues?

I got the Github org setup with the name Alpha Cassiopeiae