Technical Report

Ayush Botke and Ishan Vyas - Myoelectric Prosthetic Hand

TABLE OF CONTENTS

Project Design and Specifications

Summary

Our project uses the CPU to complete signal processing on electrical impulses from the muscle and drive the result of that signal to a prosthetic hand. Due to imperfections in the analog component of the electrode signal processing, the connection between the sEMG signals and the machine code did not work. Absent that technical failure, our project is able to successfully amplify muscle signals and separately analyze, process, and interpret a simulated sEMG signal from a function generator. Shown below in Figure 1 is the full project setup with all components included.

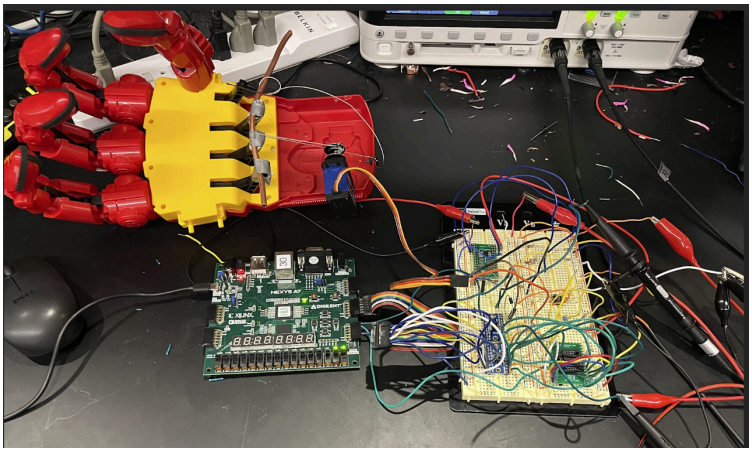

Figure 1: Full Project Setup

Front-End Electrical Design

A diagram of the front-end and back-end electrical components is shown below in Figure 2. The top half of the diagram shows the sEMG signal processing that TA Vincent Chen virtually demonstrated during our presentation. The electrodes are placed onto the **back of the wrist (ground) and the forearm (signal)** and inputted into the differential amplifier.

We used a **MAX4199 differential amplifier**, chosen specifically due to its high common mode rejection ratio (CMRR) and low offset voltage. This is important because our signal is small and therefore needs to be amplified in a later gain stage. The differential amplifier is powered from +2.5V to -2.5V to catch waveforms that go below zero. Furthermore, the differential amplifier is set to a gain of 10x, amplifying a typical sEMG signal into one observable by the oscilloscope.

The result of the differential amplifier is put into an **LM741 operational amplifier with a gain set to 100**. For the op-amp to function, the rails must be set to +15V and -15V, forcing us to stick to the power supply for our project.

At this point in the circuit, we have an amplified signal centered around 0V. In order to feed in a signal to our ADC without clipping, we need a DC offset. We attempted to use a **level shifting circuit** with a capacitor in series and a resistor connected to our desired DC offset (2.5V). However, this is where our project failed. The capacitive coupling had defects where the capacitor would discharge at random sampling intervals, setting the signal value to 0 and hampering our ability to read a clean sample from the ADC. Further work ought to be placed into either a more efficient level shifting circuit or a functional alternative. A potential solution would be an op-amp.

Regardless, an AC signal is then sent to a **TI DAC0808**. This AC to DC converter takes the AC signal and represents it in an 8-bit unsigned integer representation. The result is sent into a **bidirectional level converter** that converts the 5V binary

representation of the AC signal into a 3.3V binary representation. This value is inputted into the FPGA via registers and used for DSP and calculations.

Back-End Electrical Design

On the other side of the FPGA, dubbed the “back-end”, are the **servo** and **mechanical hand components**. We used a continuous revolution servo to provide adequate torque for the clenching and unclenching of the mechanical hand. This was powered with 5V and received a PWM signal from the FPGA. The mechanical hand used was a toy that we assembled for the purposes of this project. We used quick-dry glue to fasten a custom-bent metal bar to the hinges of the hand. We then attached a pulley to the servo and wrapped this wire around the metal bar. Thus, rotating the servo will result in lateral movement of the hinges and contract the hand.

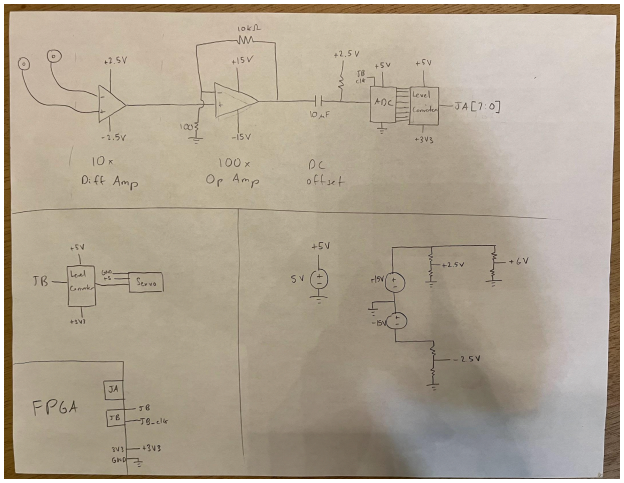

Figure 2: Circuitry Diagram

Architectural Design

The design of the CPU, including checkpoint 4 and earlier, is not included in this report. Instead, we will discuss the unique changes we made to the CPU, including direct I/O register mapping and software algorithm designs.

The CPU interfaced with the electrical components via register-mapped input/output. Figure 3 shows a table of the registers that we used, including registers $r1, $r8, and $r26 that have significant interaction with the electrical components. **Register 1 was wired directly to the ADC**. The ADC was clocked at 500kHz, giving a new ADC value every 2 us. However, we read the ADC at 50kHz, latching the value from register 1 every time a measurement was made. Reading the ADC at 50kHz as opposed to 500kHz was a compromise between having enough samples per period and having enough time between samples to execute the lines of code that we needed. Though register 8 was not directly tied to any electrical hardware, **we used the value in**

register 8 to determine when to read from the hardware, ensuring we don't read the same value multiple times from the ADC. Finally, **register 26 was used to create the PWM signal**. We would put the value we derived from the software into r26, and the value would be passed into a FSM within the processor that outputted a duty cycle. That duty cycle was then used to serialize a PWM signal that was wired to the input of the servo.

	A	B	C	D
	Registers			
1	mod 0	mod 1	mod 2	mod 3
2	0	ADC Out		store rest
3	store active	BTNL	BTNR	
4	ADC Ready	Count	Sum	toAdd
5	Avg	USED FOR CALCULATION	USED FOR CALCULATION	USED FOR CALCULATION
6	USED FOR CALCULATION	USED FOR CALCULATION	USED FOR CALCULATION	USED FOR CALCULATION
7	USED FOR CALCULATION	USED FOR CALCULATION	CONSTANTS	CONSTANTS
8	TESTING	TESTING	PWM Duty cycle signal	
9				

Figure 3: Table of Registers, r0 through r31 (Left/Right then Top/Down)

Software Design

Algorithm overview: Calculate the amplitude of the input signal and determine whether the user is clenched or unclenched.

The algorithm can be found in main.s and follows these steps:

1. The constants are defined in registers, including the offset and sample size
2. Upon BTNR pressed , the "active" situation is sampled, setting a baseline for when the hand is contracted
3. Upon BTNL pressed, the "rest" situation is sampled, setting a baseline for when the hand is restored to normal condition
4. Main loop begins; 2500 data points from the ADC are sampled, ensuring ADC is ready for each sample.
5. Distance between each data point and our offset is calculated and accumulated over 2500 samples
6. Current state set based on whether sampled amplitude is closer to rest or active. State added to lowest bit of r26 via sll and addi instructions.
7. Main loop ends; repeats back to top until reset or terminated.

(*) A note on the sampling algorithm: rather than taking the average of the function, which would always be the DC offset for every signal, we take the total variation, where we find the absolute value of (DC offset - measured value). This metric is better at quantifying amplitude changes while simultaneously being simple to implement in software.

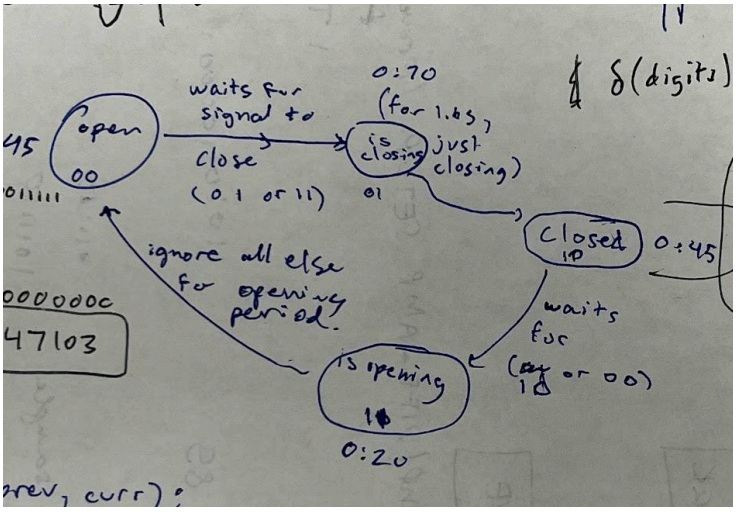

Figure 4: FSM diagram to determine when to output to the servo

FSM:
The output of our algorithm is then inputted into an FSM that we implemented behaviorally (see Figure 4). This FSM determines what PWM signal to output to the CR servo.

Inputs and Outputs

Table of inputs and outputs to the FPGA

Inputs	Outputs
Amplified electrode signal in digital form, simulated using waveform generator	PWM signal with variable duty cycle based on input signal

Processor Modifications

We did not include any custom hardware instructions. However, we did use a combination of both of our processors for the final project. Specifically, we used Ishan's CPU components, but we used Ayush's register file. Shown below in Figure 5a and 5b are snippets of the two register files we had.

We used Ayush's register file for two reasons. First, Ishan's register file takes a ton of time to compile and test due to the 1024 bit long bus. Second, Ayush's register

file gave us precise control over each specific register, making the register-mapped I/O far simpler to implement.

```
28    // makes each individual register. Output is 32x32 bus array stored as 1024x1 bus array.
29    wire [1023:0] reg_out;
30    wire [31:0] reg_in_enable;
31    genvar regn;
32    genvar tri_state;
33    // make 0th register
34    and register_enable_and0(reg_in_enable[0], ctrl_writeEnable, RDbus[0]);
35    register regfile0(.out(reg_out[(31):(0)]), .in(data_writeReg), .clk(clock), .en(1'b0), .clr(ctrl_reset));
36    // before, above was the following: .en(reg_in_enable[0])
37    generate
38        for (tri_state=0;tri_state<32;tri_state = tri_state + 1) begin: tri_state_buffers0
39            assign data_readRegA[tri_state] = RS1bus[0] ? 1'b0 : 1'bZ;
40            assign data_readRegB[tri_state] = RS2bus[0] ? 1'b0 : 1'bZ;
41        end
42    endgenerate
43    generate
44        for (regn=1; regn<32; regn = regn + 1) begin : registers
45            and register_enable_ands(reg_in_enable[regn], ctrl_writeEnable, RDbus[regn]);
46            register regfiles(.out(reg_out[(32*regn+31):(32*regn)]),
47                .in(data_writeReg), .clk(clock), .en(reg_in_enable[regn]), .clr(ctrl_reset));
48            for (tri_state=0;tri_state<32;tri_state = tri_state + 1) begin: tri_state_buffers
49                assign data_readRegA[tri_state] = RS1bus[regn] ? reg_out[32*regn+tri_state] : 1'bZ;
50                assign data_readRegB[tri_state] = RS2bus[regn] ? reg_out[32*regn+tri_state] : 1'bZ;
51            end
52        end
53    endgenerate
```

Figure 5a: Code sample of Ishan's Register File

```
20      register ZERO(qReg0, 32'b0, clock, 1'b0, 1'b0);
21      register REGISTER1(qReg1, data_writeReg, clock, write_slct[1], ctrl_reset);
22      register REGISTER2(qReg2, data_writeReg, clock, write_slct[2], ctrl_reset);
23      register REGISTER3(qReg3, data_writeReg, clock, write_slct[3], ctrl_reset);
24      register REGISTER4(qReg4, data_writeReg, clock, write_slct[4], ctrl_reset);
25      register REGISTER5(qReg5, data_writeReg, clock, write_slct[5], ctrl_reset);
26      register REGISTER6(qReg6, data_writeReg, clock, write_slct[6], ctrl_reset);
27      register REGISTER7(qReg7, data_writeReg, clock, write_slct[7], ctrl_reset);
28      register REGISTER8(qReg8, data_writeReg, clock, write_slct[8], ctrl_reset);
29      register REGISTER9(qReg9, data_writeReg, clock, write_slct[9], ctrl_reset);
30      register REGISTER10(qReg10, data_writeReg, clock, write_slct[10], ctrl_reset);
31      register REGISTER11(qReg11, data_writeReg, clock, write_slct[11], ctrl_reset);
32      register REGISTER12(qReg12, data_writeReg, clock, write_slct[12], ctrl_reset);
33      register REGISTER13(qReg13, data_writeReg, clock, write_slct[13], ctrl_reset);
34      register REGISTER14(qReg14, data_writeReg, clock, write_slct[14], ctrl_reset);
35      register REGISTER15(qReg15, data_writeReg, clock, write_slct[15], ctrl_reset);
36      register REGISTER16(qReg16, data_writeReg, clock, write_slct[16], ctrl_reset);
37      register REGISTER17(qReg17, data_writeReg, clock, write_slct[17], ctrl_reset);
38      register REGISTER18(qReg18, data_writeReg, clock, write_slct[18], ctrl_reset);
39      register REGISTER19(qReg19, data_writeReg, clock, write_slct[19], ctrl_reset);
40      register REGISTER20(qReg20, data_writeReg, clock, write_slct[20], ctrl_reset);
41      register REGISTER21(qReg21, data_writeReg, clock, write_slct[21], ctrl_reset);
42      register REGISTER22(qReg22, data_writeReg, clock, write_slct[22], ctrl_reset);
43      register REGISTER23(qReg23, data_writeReg, clock, write_slct[23], ctrl_reset);
44      register REGISTER24(qReg24, data_writeReg, clock, write_slct[24], ctrl_reset);
45      register REGISTER25(qReg25, data_writeReg, clock, write_slct[25], ctrl_reset);
46      register REGISTER26(qReg26, data_writeReg, clock, write_slct[26], ctrl_reset);
47      register REGISTER27(qReg27, data_writeReg, clock, write_slct[27], ctrl_reset);
48      register REGISTER28(qReg28, data_writeReg, clock, write_slct[28], ctrl_reset);
49      register REGISTER29(qReg29, data_writeReg, clock, write_slct[29], ctrl_reset);
50      register REGISTER30(qReg30, data_writeReg, clock, write_slct[30], ctrl_reset);
51      register REGISTER31(qReg31, data_writeReg, clock, write_slct[31], ctrl_reset);
```

Figure 5b: Code sample of Ayush's Register File

Challenges We Faced

We faced many challenges during this process, and though not every challenge was fully resolved, the experience was nonetheless extremely educational.

ADC: One of the most baffling issues we faced was with the ADC. From the datasheet, we gathered that the pins 23:25 control which address the ADC read from; see Figure 6. As per the datasheet again, by grounding ADDR A, B, C, we should expect the address to read from pin 26: IN0. However, our testing revealed otherwise. Astonishingly, despite grounding ADDR A, B, C, we saw the signal read from IN7. This issue took a long time to resolve, given that we were unclear whether the issue was from the input, the address selection, the bit reading, the oscilloscope itself, or a variety of other possible explanations for the oddities we saw.

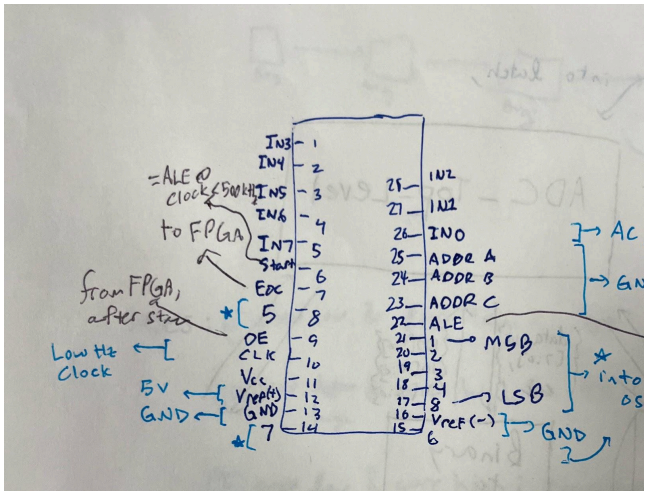

Figure 6: Pin Layout of ADC0808, with annotations

MIPS Algorithm: Testing MIPS is difficult enough (ref: ECE250 office hours during MIPS assignment), but testing on the FPGA was a unique challenge itself. Since we were doing an electrical project with no screen, there were no print statements; the debugging tool that we utilized the most was the on-board LEDs. Unlike GTKWave, we could only test 16 bits at a time, and could not change those 16 without resynthesizing. Furthermore, in the algorithm itself, we had to account for imperfections. Those imperfections include variations in the measured average (even with a clean input signal), transients, and less-than-perfect ADC readings. Not only did we have to discover these edge cases, we also had to design a resilient program that included them.

Other challenges: Beyond those two challenges, we had many other challenges that forced us to rethink our original plan. Those include:

- Designing the PWM for a CR servo instead of a closed-loop servo. Originally to get better torque, but algorithmically more challenging.
- Level shifting circuit; adding a DC offset to the signal before entering the ADC posed a challenge.
- Back-current from the servo was corrupting the PWM signal that was supposed to control the servo. Upping the voltage and adding a diode was ultimately a sufficient fix.

Circuit Diagrams

See Figure 2 in Back-End Electrical Design. The diagram itself was created after the signal was processed as designed, but the logic behind each step was from our initial plan. Each major component served a specific role, including:

- Calculating the electrode voltage difference
- Amplifying the signal
- Converting to digital
- Preparing for input into / output out of FPGA

Test Plan - Software

The goal of our test plan was to verify the functionality of our CPU, register-mapped I/O, and MIPS program. We created separate MIPS programs and utilized on-board LEDs to test the following functionalities:

- Basic Servo Control using buttons
- Latching ADC values using buttons (successively)
- Displaying a total value for the latched ADC value
- Ensuring the functionality of our total variation (ie absolute value) function
- The entire calibration step

Overview of Assembly Program

Full assembly code submitted in separate assignment (main.s). For explanation of the algorithm, please see Software Design

Potential Improvements

Many future improvements could be implemented; here are some of the improvements we would focus on if we had more time:

- More Robust Signal Processing / Level shifting Problem: Ensuring that connecting the op-amp to the ADC does not cause the voltage to drop to ground during sampling, allowing the muscle impulses to fully control the prosthetic hand
- Soldering and Wire Management: Due to the iterative and ever-changing nature of our project, we were constantly replacing wires and changing schematics. Once we fix the problem above, we would focus our efforts on ensuring the wire connections were clean and would be able to pass the ‘pinky test’
- Less ‘DIY’ Hand: The plastic used for the hand in the final project was flimsier than we realized, and bent when the servo clasped. We would want to pick sturdier materials like metals that could withstand those forces better.

Extra Pictures

In addition to the figures throughout the report, we wanted to include the following:

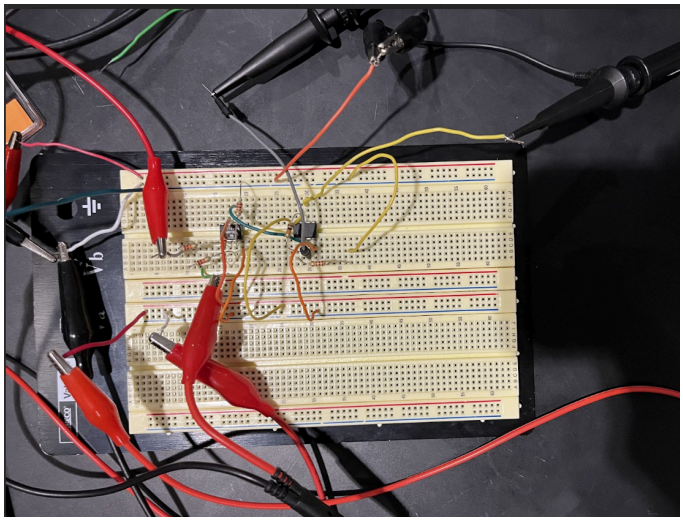

Figure 7: Picture of the first circuit we saved - proof of concept of op-amp functionality.

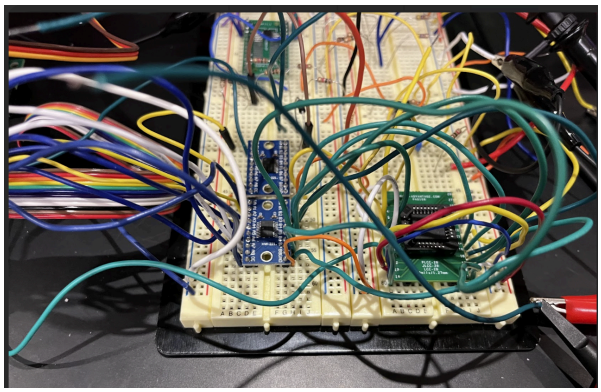

Figure 8: Close up of our final circuit