Robotic Arm Technical Overview

Milo Chung and Aiden Crowe

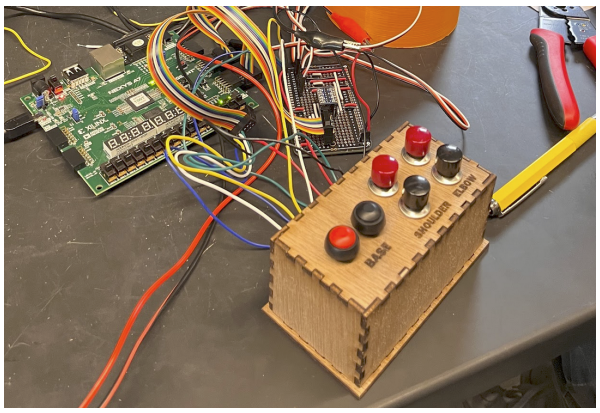
Project Design and Specifications

The goal of this project was to create a system for controlling a robot arm using our custom processor and assembly code. Although at the outset, we did not have a specific task in mind for the robot arm to achieve, we knew we wanted to manipulate or interact with some kind of real-world object. This goal could have manifested in the form of sorting objects with a gripper, creating multi-dimensional arrays of sensing data, or even participating in some kind of game with a human user. In the end, we decided to create a drawing robot, with which a user would be able to draw in marker via joint-based movement inputs. Due to the flexibility of the robot arm, this choice meant that we could essentially use the robot as a 2D plotter on a flat horizontal surface, or draw on more complex surfaces, such as a vertical surface, curves, or essentially anything that fit within the robot's work envelope.

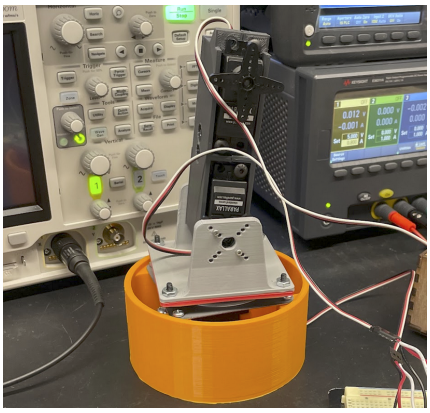
With this general goal in mind, we knew that this project would be a fairly complex digital system, both mechanically and electrically. However, in order to increase the amount of complexity on the computing side of the project, we added in the specification that the robot would have to be able to replay the user's button inputs to recreate their drawing. The addition of this task made the project significantly more complex and was meant to show the power of the processor that we had created and improved upon for this project. Thus, we had our cumulative goal of both creating a robotic arm that would be able to accept user inputs to draw, and subsequently being able to play back those same user inputs to recreate their drawing.

Input and Output

User inputs for this project consisted of a set of six buttons inside a case that was distinct from the FPGA, wiring board, and robot. Additionally, we used two buttons onboard the FPGA to accomplish the task of replaying and resetting the sequence of instructions given to the robot. In total, this meant eight methods of user input, with their locations grouped according to their function. Each button is robust against initial state errors, meaning that if the button is pressed when the FPGA is programmed or powered on, the logic controlling that button doesn't become inverted. Additionally, each of the 6 control board inputs is tracked for the replay and reset functions, with changes in state being tracked in memory.



Control Board with Buttons



Servo Outputs on Robot Arm

Initially, all of these inputs were meant to be read in from a keyboard attached via the PS2 interface on the FPGA, but we ran into several issues while working on this system. First and foremost was the difficulty in reading the “key released” code from the incoming bitstream. While we could successfully detect key presses from the keyboard, the data that was supposed to be sent upon a key being released was not received by our FPGA. This issue was far from limiting, as it would have been solvable with some time and effort, however when troubleshooting we also took a moment to step back and discuss why we were using a keyboard in the first place. The inputs would be less clear and there was significant confusion about which keys on the keyboard to use, especially since we would need to support multiple motors moving at once. Due to all of these factors, we opted for the control panel with six buttons, giving us a simple set of inputs and plenty of room for users to press multiple buttons at once.

There are only three functional outputs in this system, each of the three servo motors on the robot arm, each of which is controlled via pulse-width modulation (PWM). These are standard servos, meaning that they are limited to a range of angles with precise control over position. Inside of our wrapper module, we pulled values from three different registers to generate the required waveforms to move the servos to their respective positions. These positions in the registers were directly changed by the button inputs on the control board, incrementing or decrementing each motor position based on which button was being pressed. The combination of these three servos gave the robot three degrees of freedom, each with a range of 180 degrees of rotation. As our rotation range was limited, we mechanically optimized each range to give the robot maximal reach and angular flexibility in our desired work area.

```
assign regOUT[31:0] = 32'b0;
assign regOUT[63:32] = {31'b0, btn};  //      fpga top button
assign regOUT[95:64] = {31'b0, btn_2}; //     fpga bottom button
assign regOUT[127:96] = {31'b0, in_1}; //3      inputs 1-6
assign regOUT[159:128] = {31'b0, in_2}; //4
assign regOUT[191:160] = {31'b0, in_3}; //5
assign regOUT[223:192] = {31'b0, in_4};; //6
assign regOUT[255:224] = {31'b0, in_5};; //7
assign regOUT[287:256] = {31'b0, in_6};; //8
assign signal_1 = regOUT[319:288]; //9        servo outputs
assign signal_2 = regOUT[351:320]; //10
assign signal_3 = regOUT[383:352];  //11
assign led = regOUT[384];
```

Modified Registers

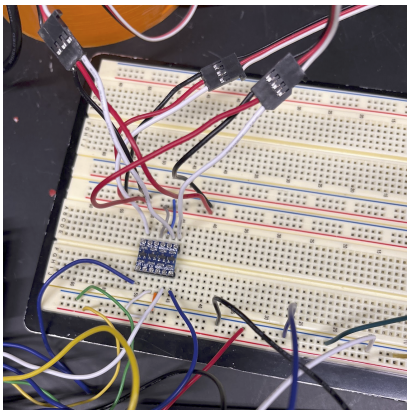
```
//Servo_1
wire[31:0] desired_angle_1;
wire [30:0] limit;
reg [30:0] counter = 0;
reg on_off = 1;
reg servo_signal = 1;

assign limit = on_off ? (75000+((desired_angle_1)*56)): 1000000 - (75000+((desired_angle_1)*56));

always @(posedge fpga_clock) begin
    if (counter < limit)
        counter <= counter + 1;
    else begin
        counter <= 0;
        servo_signal <= ~servo_signal;
        on_off <= ~on_off;
    end
end

assign srvo_1 = servo_signal;
```

PWM Signal Generation

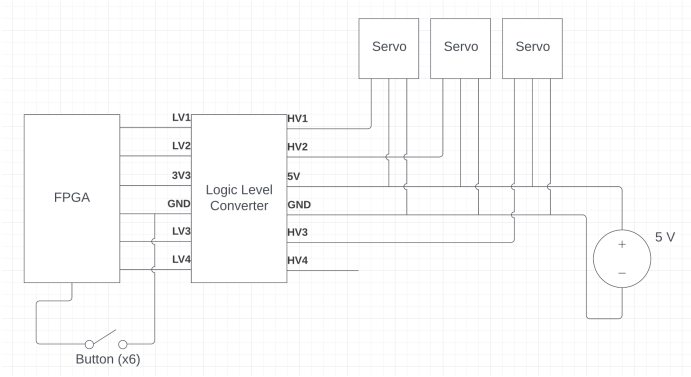
As an output for testing, we had the LEDs on the FPGA connected to various buttons and keyboard input bits throughout the development process as well. This method of debugging output was how we knew that the keyboard input wouldn't work and made the decision to move forward without that aspect of user input. However, we removed the majority of the LED functionality by the conclusion of the project.

External Circuitry

Our external circuitry on this project was not very complex, consisting of PWM outputs for each servo motor, a logic level shifter to amplify the motor control signals from the FPGA, and several ports for digital inputs and outputs. Our initial testing setup is shown below.

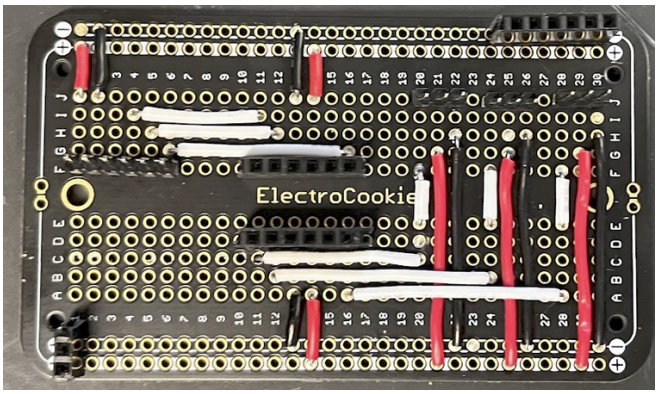


Initial Breadboard Circuit

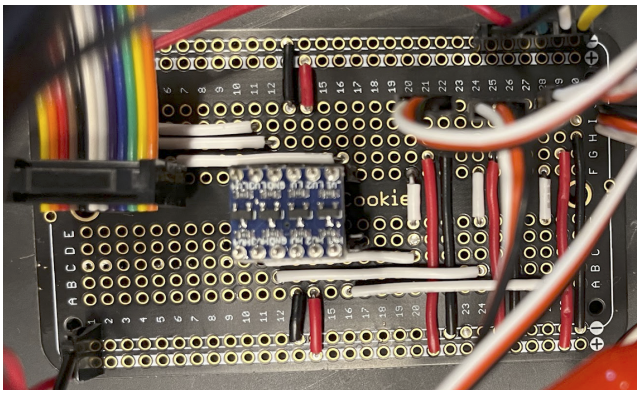


Electrical Schematic

The logic level shifter was wired on one side to the 3.3V digital outputs, 3.3V power source, and ground from the FPGA, and to the motors and a 5V power source on the other side. The buttons were all wired to the FPGA ground, acting as pull-down switches for the digital inputs on one of the FPGA ports. Below, all of those same features can be seen on our soldered protoboard, with the power inputs, FPGA connections, and PWM cable connections converted to pin headers. We used pin headers to make assembly and disassembly easier, as well as to simplify the wiring process, allowing us to use ribbon cables instead of loose wires. On the board below, the 3.3V rail is at the top of each image, and the 5V rail is at the bottom.



Protoboard without connections



Protoboard with components and connections

Assembly Code

We wrote out assembly code in two stages. First, we wrote a simple program that allowed our arm to be controlled with push buttons. After testing and debugging that code, we expanded the program so that it could remember and replay the movements executed by its most recent operator.

The basic control code was structured as one big loop. During each loop iteration, we would cycle through each input value, and update the corresponding output accordingly. Each input output pair had one logic block. If the input was a one, we would increment the servo angle register then use the wait command to pause for a fraction of a second. If the input was a zero, we would branch over that code and continue onto the next logic block.

Implementing the record and replay features proved to be far more complex. To achieve the correct movement timing in our replay, we created a loop counter. Each time an input signal changed from a one to a zero, we would record the current loop number in RAM. Since we only recorded the times when the inputs changed, we could store long sequences of movements with very little memory. To organize the on/off markers for our six input/output pairs, we assigned each pair its own section of memory.

Our code was divided into two loops, the first recorded a user's movements while the second replayed those movements. The record portion of the code had six registers which contained the expected state of our six inputs—either a one or a zero. Before updating the servo angle register for each of our servos, we would check if the current input differed from the expected input. This difference signified an input change and triggered a separate branch that recorded the current loop counter value.

At any time a user could press one of the FPGA buttons to transition from the record loop to the replay loop. This transition reset the loop counter, and changed a number of register values. In the replay loop, each servo was assigned a separate register that acted like an input value. These stand-in input registers were toggled when the loop counter reached one of the values stored in RAM. Our code also allowed the user to jump back to the first loop to record a new sequence of movedes. Upon jumping back, all RAM values would be set back to zero.

```
#--------record-------
nop
nop
bne $3, $16, base_a_record   #jump to record section of value saved is different than current button value
nop
nop
base_a_return:             #return from record section and execut normal procedure
#-------normal---------
nop
nop
bne $3, $15, base_a
nop
nop
addi $9, $9, 1
nop
nop
wait 10000
nop
nop
base_a:
```

Code From ‘Record Loop’

```
base_a_record:
nop
nop
sw $13, 0($22)       #stor current time in given button's memory section
nop
nop
addi $22, $22, 1      #increment memory adress for next value
add $16, $0, $3      #change current value to match with button state
nop
nop
j base_a_return      #return to normal procedure
```

Code to Save Input Sequence to Memory

```
#base_a
nop
nop
bne $13, $16, base_a_start   #check if we have reached the point in counter where we want to change from off to on or visa versa
nop
nop
bne $22, $0, base_a_not_o  #check whether to change from one to zero or zero to one
nop
nop
addi $22, $0, 1
addi $12, $12, 1
nop
nop
lw $16, 0($12)
nop
nop
j base_a_start
nop
nop
base_a_not_o:
nop
nop
addi $22, $0, 0
addi $12, $12, 1
nop
nop
lw $16, 0($12)
nop
nop
#-------------------------
base_a_start:
nop
nop
bne $22, $15, base_az
nop
nop
addi $9, $9, 1
nop
nop
wait 10000
nop
nop
base_az:
```

Section of ‘Playback’ code

Major Challenges

Our biggest setback was our inability to get the keyboard system working. As discussed above, we spent a substantial amount of time working on this issue before deciding to pivot to push buttons instead. Although this was the most frustrating obstacle, we had a number of smaller setbacks as well. One difficulty came from the conversion from degrees to PWM. We generated our PWM signal using a clock cycle counter. Each time the counter flipped, we adjusted the counter limit to create a valid signal. Initially our conversion from degrees to counter limit values was flawed. No matter the degree input, the output signal always caused the servo to rotate 90 degrees. Since we had only tested this capability with a goal of 90 degree rotation, we did not realize the conversion was causing the problem for some time.

Another persistent issue arose when debugging the assembly code. We used six different registers for each input, and ended up using most of the other registers to hold various values. Despite rigorous commenting, it became difficult to keep track of which registers corresponded to which input output pair. We had one mix up where we used a different order of execution for two different sets of registers. Finding this bug required heavily documenting our code and analyzing the erroneous behavior of our robot, a time consuming process. This could have been avoided if we had put more thought into writing efficient, organized code before jumping into the programming.

Testing

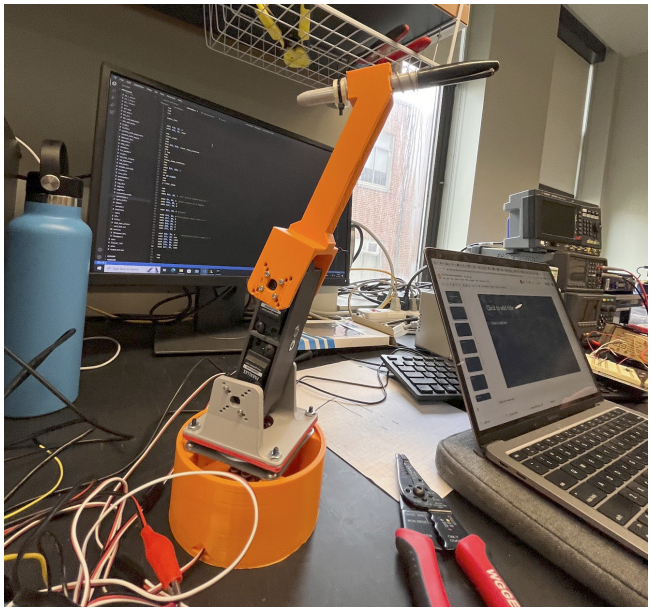
We iteratively tested our project as we were designing it to make sure the individual components were functional. There was not enough time to formally test our final integrated product, however we did sufficient test runs to be confident that our robot worked. We repeatedly recorded and replayed sequences of robot movements without reprogramming the FPGA between each trial. In each of these tests the robot was successful. To test the accuracy of the replay function, we attached a pen to the end of the arm and used it to draw on a piece of paper. When the robot replayed the drawing, it mimicked the exact shape and line thickness from the original drawing, giving us further confidence in the arm's accuracy and the ability of our code to run successfully.

Future Improvements

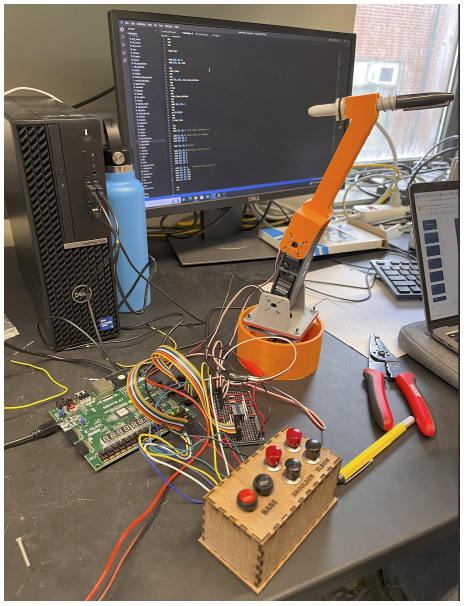
In this project, we felt that we executed everything well, as there were very few technical or functionality issues by the end of our development. The main shortcomings were mechanical in nature, with the base of the robot arm coming off of the floor when the arm was lowered excessively and the spring-loaded sharpie pressure relief mechanism required too much force to function properly. However, both of these are simple fixes that impacted the functionality of the arm very little. The only area of improvement on the assembly code side was the potential to combine reused code snippets into functions, but everything worked as expected with how the current program was written, meaning that this issue had no impact on the functionality of the arm.

Despite the fact that there was little to improve on the arm, there are many ways that this project could be expanded beyond its current state. For instance, we could include a feature to export previously recorded sequences, as currently they are saved to memory and subsequently overwritten. Additionally, we could make the attachment at the end of the arm that is modular, supporting other writing instruments or expanding functionality to include other types of grippers. For instance, we could put a mechanical claw on the end and use the arm as a pick and place for electrical components or to assemble a second robot arm. Alternatively, we could add more degrees of freedom to the arm by putting a rotation axis perpendicular or parallel to the elbow joint or mounting the system on a rail to add a linear axis of motion.

Additional Pictures of Project



Robot Arm



Full System