Humza Chouhdry, Harrison York
ECE 350
April 24, 2024

ECE350 Final Project: Jabber Robot Arm

Introduction:

Jabber is a five-servo robot arm that can be controlled using two joysticks and switches on the Nexys A7 FPGA board. The arm has four axes of rotation, plus a claw grip that can be used to pick small objects up. The joysticks enable manual control of the arm, while a custom movement mode is triggered when a switch on the FPGA is on. Custom movements are defined in assembly, specifying duty cycles that correspond to exact positions of the arm and repeating in a loop until the switch is turned off.

Hardware:

We utilized Duke's 3D-printing resources to create the body of the robot arm and the joystick housing. Most of the parts were found on Thingiverse (Build Some Stuff). The base of the robot was modified and the joystick housing was custom designed using Fusion 360.

Modifications to the original design include using two Lego gears for rotating the gripper up and down. The original plan was to use three differently sized gears, but there was not enough material to anchor these gears, and the Lego gears were a much more reliable solution, although they take up more space.

The prints consistently had less tolerance than needed for fastening the parts together with screws, so we often drilled holes to use the available screws on campus.

Five servos were used in the robot arm: one 20 kg load servo, three MG996R servos, and one MG90S servo. The 20 kg servo was used for the first segment of the arm, analogous to a shoulder, because it could resist and provide the most torque. The mini MG90S was used in the gripper, and the other three servos were used to articulate the arm as the elbow and wrist analogs (Fig. 1).

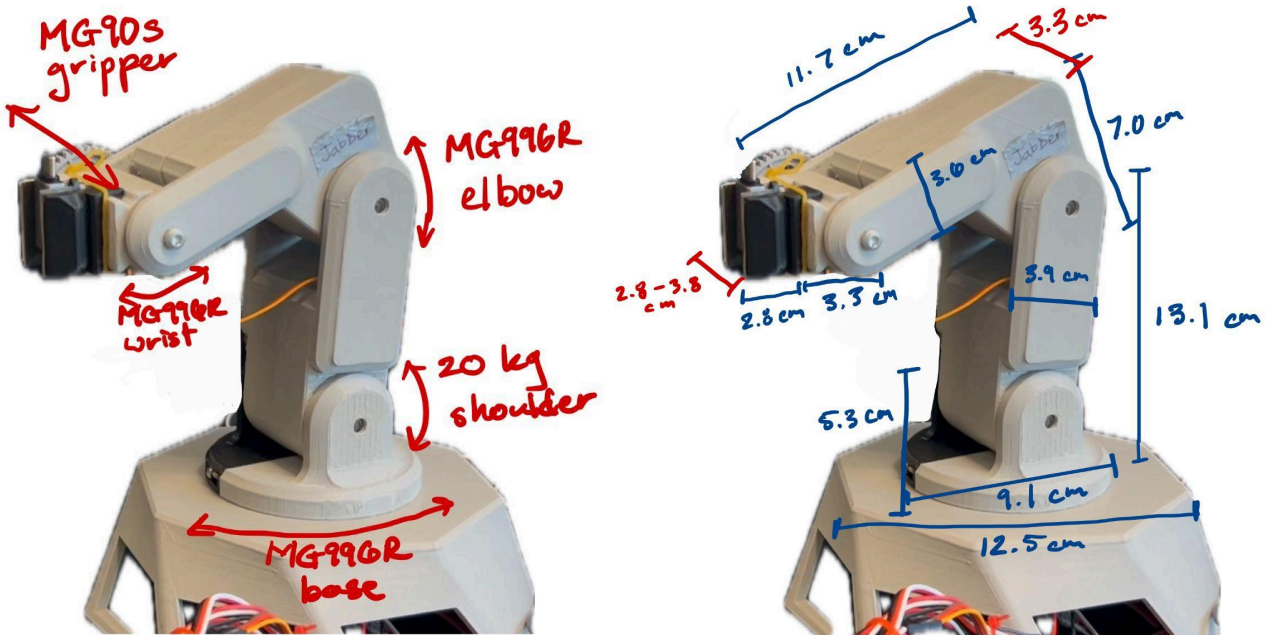

Figure 1: Diagram of Jabber

We used two level shifters to bring PWM signals from 3.3 V (FPGA output) to 5 V (servo input). All of the servos took 5 V as power, which we supplied using a wall adapter.

The two arcade joysticks had four outputs (up, down, left, right) each, and after some experimentation and research we found that the outputs are active low. They also operated at 5 volts, meaning we had to shift the voltage down to 3.3 volts to use it as an input to the FPGA.

The level shifters we used to bring the servo PWMs up to 5 V were only one way, so we decided to make simple voltage dividers. These consisted of three 10 kOhm resistors in series so that each resistor would experience ⅓ of the total voltage drop. We connected the FPGA input to the node in between the first and second resistor, meaning it experienced ⅔ of the total 5 V when the signal was high (Fig. 2). This means that only 3.3 V would be sent to the FPGA.

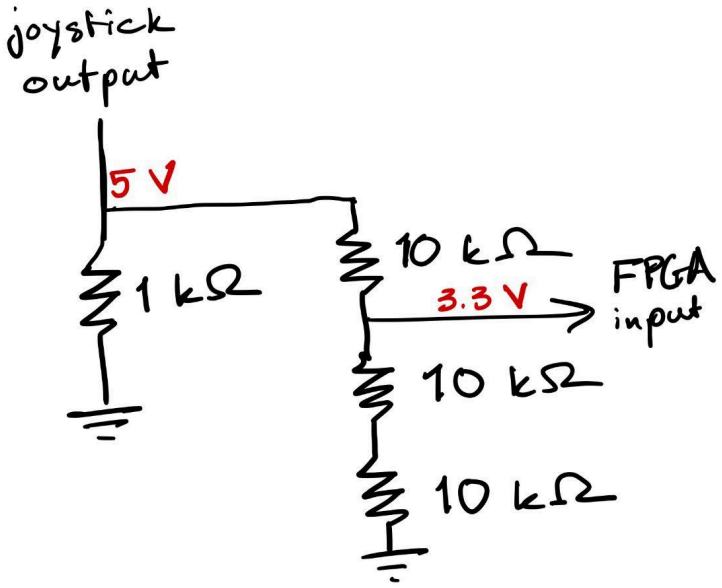

Figure 2: Voltage Divider Circuit

This circuit was repeated for each output and soldered onto a protoboard (Fig. 3). There are also pull-up resistors attached to each input. The pull-up network causes the output to be high when the respective level switch is open (no current, so resistor has zero voltage drop and output is full five volts) and output to be low when the level switch is pressed (all voltage drop is over resistor). This output is then wired as the input to the voltage divider.

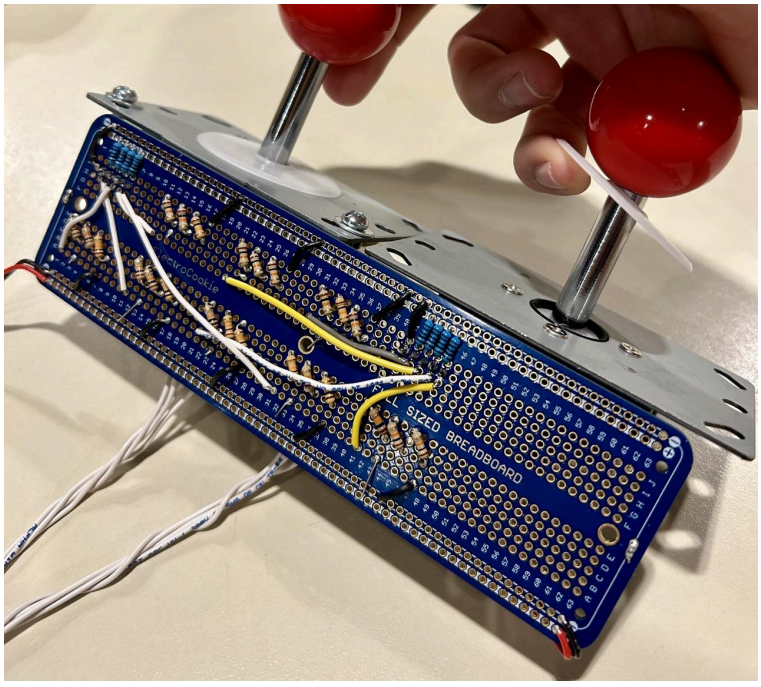

Figure 3: Joystick Protoboard with Soldered Components

Inputs and Outputs:

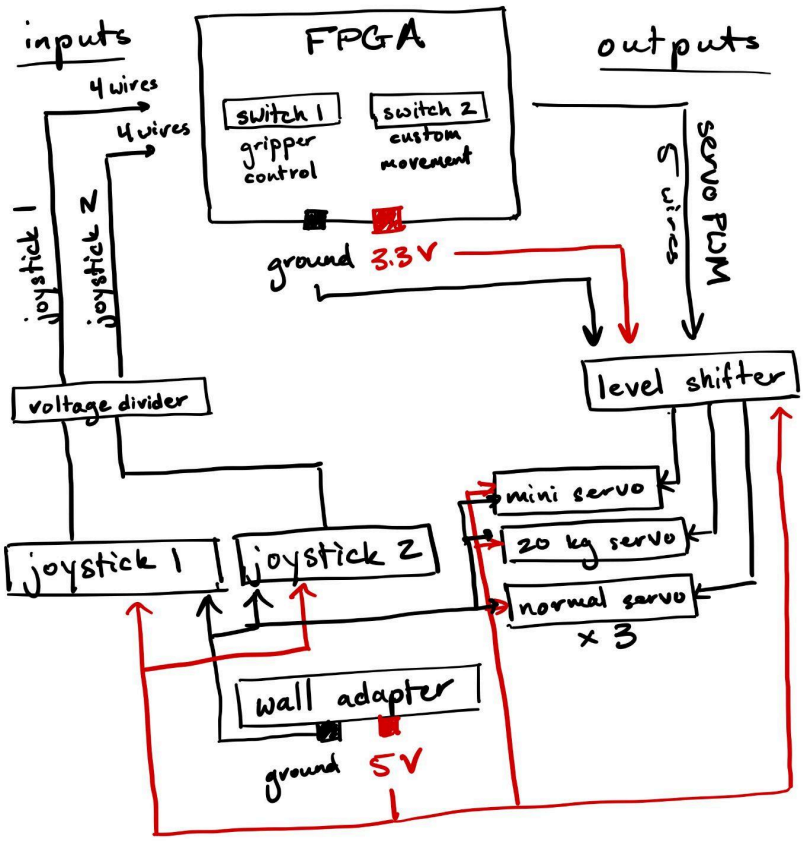

Figure 4: Diagram of inputs and outputs relative to the FPGA

Each joystick has four output signals corresponding to the up, down, left and right directions on the joystick. The switches used in the joysticks are digital and active-low, so they output 0 when the switch is pressed. We can also read two directions at once, for example up and right or down and left, if the joystick is pressed into a corner.

We set registers 7 and 8 to read the output of the joysticks into the bottom four bits of each register (right, left, down up – RLDU, from most to least significant bit). If the joystick is resting, the register will read 1111, while a reading of 1101 means that the joystick is moved down.

There are two different PWM signal types used in our project. The first is used by the 20 kg servo and the three “normal” servos (MG996R). The pulse-width modulation period is 20 µs, with duty cycles ranging from 0.5 µs (0) to 2.5 µs (99). The mini servo also has a PWM period of 20 µs, but with duty cycles from 1 µs (0) to 2 µs (99).

In order to generate these two kinds of PWM signals, we modified the PWM serializer from lab with our specifications and created distinct modules as needed. These modules read in the values of registers 1, 2, 3, 4, and 5 as duty cycles from which to create PWM signals, so the same duty cycle PWM will always be output unless it is specifically changed by an instruction.

The two switches are also set to read directly into two registers, changing the least significant bit in each. We later read this bit to determine whether to open or close the gripper or switch from manual to preset control.

The joysticks and servos all take 5 V from the adapter, and the PWM signals are boosted by level shifters from 3.3 to 5 V. The level shifters also need a 3.3 V signal from the FPGA in order to properly raise the signals. Our voltage dividers ensure that nothing above 3.3 V is ever input into the FPGA.

Changes to processor:

The processor formed an integral part of our project, it was largely unaltered except for adding support for a custom instruction.

Delay instruction:

The Delay instruction is a JI type instruction with an opcode of 11111. Its 27 bit immediate value is used to determine how many clock cycles to “wait”.

```
70  ////////////////////////////////////// Handling Delay  //////////////////////////////////////
71  reg [63:0] current; // Create register to count clock cycles
72  reg [63:0] count_till; // Create register to hold which clock cycle to count till
73
74  /* Math:
75  Counts to 5000 every 1 second. Counts to 5 every 1ms*/
76
77  always @(negedge clock or posedge reset) begin //make sure to insert 5 nops
78      if (reset) begin
79          current = 0;
80          count_till = 0;
81      end else begin
82          current = current + 1;
83          if (q_imem[31:27] == 5'b11111) begin
84              count_till = current + 100000 * q_imem[26:0] + 1;  // Update count_till based on q_imem value
85          end
86      end
87  end
88
89      ////////////////////////////////////////////////////////////////////////////////
```

Figure 5: Delay Counter Implemented in Processor

Two new 64 bit-registers are created in the processor, *current* and *count_till*. The *current* register increments every clock cycle and keeps track of the number of clock cycles passed since the last reset. These registers have a capacity to store up to 2^64

clock cycles. With a clock of 100MHz, these registers can accurately update up to 51 million hours, which is more than enough for the small delays this is intended for. Whenever a delay instruction is fetched, the *count_till* register is updated by the equation:

$$Count\ till\ =\ current\ +\ 100000\ *\ qimem[26:0]\ +\ 1 \qquad (1)$$

The scaling of immediate value by 100,000 is to effectively create a delay resolution in which a value of 1 in immediate creates a 1 millisecond delay. The 1 added is to account for the fact that the delay will begin once the instruction reaches the FD latch which will be in the next cycle.

```
123
124     ////////////////////////////////////// Handling Delay  //////////////////////////////////////
125     reg stall_for_delay;
126
127     always @(posedge clock) begin
128         if (current <= count_till && FD_IR[31:27] == 5'b11111) begin
129             stall_for_delay = 1'b1;
130         end else begin
131             stall_for_delay = 1'b0;
132         end
133     end
134
135     ////////////////////////////////////////////////////////////////////////////////
136
```

Figure 6: Stall signal based on delay counter

When the delay instruction is fetched from the FD_latch, we wait till *current* catches up to *count_till*. While we wait for that to happen, the flag *stall_for_delay* turns high which disables all latches in the processor from propagating instructions. Once *current* reaches *count_till,* we turn the flag off and the processor operations occur as normal. This delay instruction requires 5 NOPS before every instance to ensure no important operations are stuck in the pipeline while we wait for the delay to expire.

Challenges:

Button issue:

The issue that consumed the most of our time was button bounce. Upon getting the manual control for Jabber working using the joysticks, we wanted to integrate control for the mini-servo as well as an input to let the arm know to default to pre-programmed custom movements. For our purposes, we decided to use the buttons on the FPGA, simply due to the ease of the setup and the accessibility. But upon integrating them with our code as well as a few additional branches to reflect the new conditions, our manual movements started failing. Duty cycles would randomly jump to 0 to 99 or the other way around.

Most of our debugging for this however was focused on software as we hypothesized that we had introduced an incorrect branch or jump somewhere. After quite a few iterations of re-checking the code as well as comparing it with our base code with working manual control, we isolated the integration of buttons as the only possible change left that might have been causing the issue. We pivoted from using buttons on the FPGA to using the switches and this immediately resolved the issue for the exact same code. We hypothesize that the issue was button bounce or unstable continuity.

Gears:

Another major issue was related to the fourth (wrist) servo and the gear system designed to rotate the gripper mechanism up and down by this servo. The original plan was to use three gears in series, but we found that the first and third gears were at different heights and so the middle gear would easily pop out of the mechanism. The screw for the middle gear also did not have enough material to hold on to, and this similarly caused the gear to slip.

One idea we implemented to try and solve the problem was to use a pulley. Our gear kit included two pulleys, and we attached them with a rubber band, but we found the band too elastic to provide enough tension to turn the gripper.

When we were working in the Pod to repair a part that was stripped by a servo, we noticed a container full of Lego gears. We ended up finding two gears that fit together surprisingly well, and although they were much larger than the original gears, they worked nicely (Fig. 7).

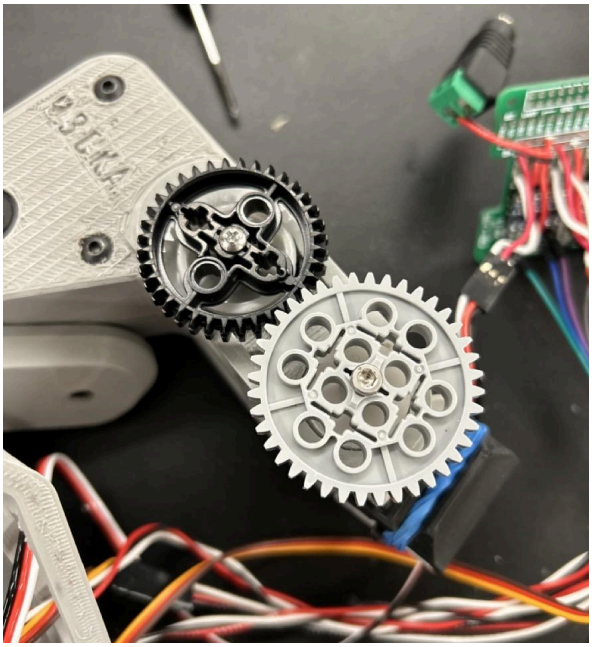

Figure 7: Lego Gear Solution

Testing:

We used several methods of testing depending on the stage of completion of the project. For example, after soldering the joysticks and voltage dividers onto a protoboard, we tested continuity and measured outputs using a power supply and the oscilloscope. PWM signals were also tested on the oscilloscope to confirm functionality.

When we suspected that the first mini servo had been fried, we tested a duplicate and made sure that our signals could control the servo before dismantling the arm and replacing the motor.

During the most grueling stage of our project – integration – we used version control to isolate changes and identify issues as they arose in our software. This was a slow journey, especially due to Vivado's workflow processes, but it ultimately resulted in success with our robot meeting our criteria for a minimum viable product.

Testing helped us grow more familiar with the laboratory tools and use them to our advantage. At first they are somewhat intimidating, but after learning how they can be useful they definitely made our lives easier, and we can apply these skills to future projects.

Assembly programs:

Our assembly has two main components. The first component controls the manual movements from the joysticks while the second component has instructions for the custom or automatic movement of the arm. Common to both components is a few key registers, these being register 1 - register 9.

$$Reg1 \sim Reg5 \;=\; DutyCycles\ Servo1 \sim Servo5 \tag{2}$$

$$Reg6[0] \;=\; FPGA\ Switch\ 2 \tag{3}$$

$$Reg7[3:0] \;=\; Directional\ values\ from\ Joystick\ 1 \tag{4}$$

$Reg7[0]$ = !UP
$Reg7[1]$ = !DOWN
$Reg7[2]$ = !LEFT
$Reg7[3]$ = !RIGHT

$$Reg8[3:0] \;=\; Directional\ values\ from\ Joystick\ 2 \tag{5}$$

$Reg8[0]$ = !UP
$Reg8[1]$ = !DOWN
$Reg8[2]$ = !LEFT
$Reg8[3]$ = !RIGHT

$$Reg9[0] \;=\; FPGA\ Switch\ 1 \tag{6}$$

This mapping of certain registers to hardware components was done in Verilog as described above.

Manual movement:

Manual movement of the arm depends on received joystick values that are decoded to increment or decrement values in registers that correspond to the duty cycles of a certain servo. Initially, all values are isolated using a series of shift left and shift right operations that isolate whether a certain joystick value is on or not

```
sll $11, $7, 30
sra $11, $11, 31
```

Figure 8: Shift Instructions to Isolate a Single Joystick Output

Figure 8 isolates the second bit in register 7 (joystick 1), if after these operations the result in register 11 is a -1, that would mean that joystick 1 → Down is not on whereas if it was a 0, that would mean that joystick 1 → Down is on. This is done for all 4 directional inputs from each joystick. Once all values have been decoded into registers, we then begin incrementing or decrementing duty cycles as well as checking for boundary conditions.

```
68   check_1:
69   bne $10, $0, check_2 #if not equal to 0, go to check_2
70   blt $20, $1, check_2 #if 98 < duty cycle, then go to check_2
71   addi $1, $1, 1
72
73   check_2:
74   bne $11, $0, check_3 #if not equal to 0, go to check_3
75   blt $1, $21, check_3 #if duty cycle < 1, go to check 3
76   sub $1, $1, $21 #reg 21 = 1
```

Figure 9: Control for Servo 1

Figure 9 shows the implementation of conditional logic used to update Servo1. If reg10 (joystick 1 → UP) has a value of 0, and if the duty cycle is below a certain threshold (99), then we would increment the duty cycle of Servo1. In contrast, if either of the two conditions are not met, we jump to check_2 to see if reg11 (joystick 1 → DOWN) is on, and if the duty cycle is above a certain threshold (0). If these last two conditions are met, we would decrement the duty cycle of Servo1. This string of conditional logic is repeated for all the other servos with the mini-servo as an exception. The mini-servo toggles between a duty cycle of 0 and 98 based on if Switch 1 (LSB of reg 9) is on.

New joystick values are latched every 20ms using the delay instruction and this cycle continues until Switch 2 is turned on, in which case, we switch to custom movements.

Custom movement:

The custom movement is an arrangement of appropriately timed delay instructions and duty cycle assertions. Our current custom movement corresponds to Jabber moving to a specified position near the ground, closing its claws to grab something, picking it up, rotating by a certain angle before opening its claws to drop the object. The way this custom movement was accomplished was after successfully testing for manual control, we moved Jabber to certain positions, disconnected servo outputs and measured, using an oscilloscope, values for duty cycle corresponding to the current position of each servo. We did this for three positions (Fig. 10):

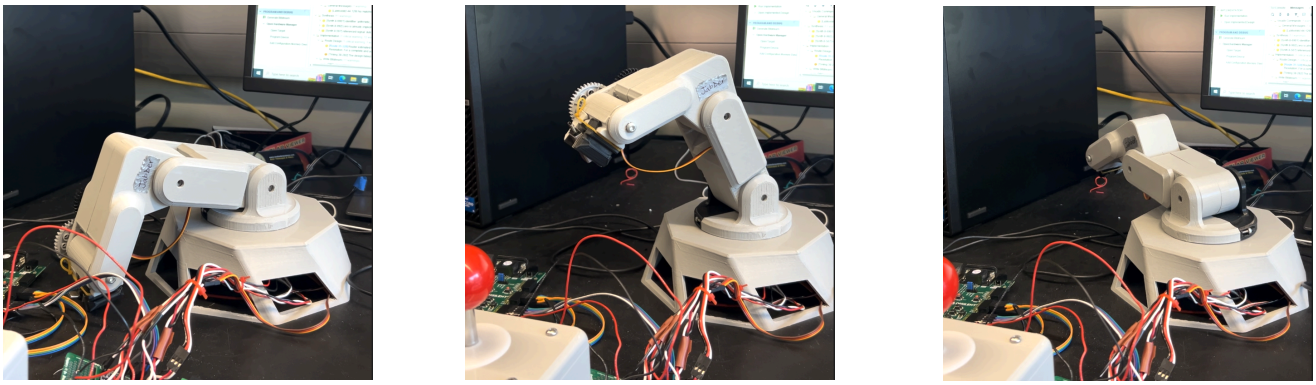

Figure 10: Successive Preset Movements

After noting these specific values, we added some intermediate duty cycle values accompanied with delays to ensure smoother, less sudden movement. In code, this meant adding immediate duty cycle values to each servo register and delaying by an appropriate time to ensure the servo had time to move (and in some cases stay) at the correct position before moving onto the next position.

```
147    delay 10 #delay of 10ms
148    addi $5, $0, 85 #open claw
149    delay 100 #delay of 100ms
150    addi $4, $0, 62 #move downward to correct position
151    addi $3, $0, 52
152    addi $2, $0, 53
153    addi $1, $0, 57
154    delay 5000 #delay of 5000ms
155    addi $5, $0, 0 #close claw
156    delay 1000 #delay of 1000ms
157    addi $1, $0, 38 #make it upright
```

Figure 11: Custom Movement

Figure 11 is an example of part of the movement (with nops in between removed for ease of readability). This custom movement will loop, effectively simulating a sort of assembly

robot, while the switch remains on. Turning the switch off reverts Jabber back to manual control.

Improvements:

Replacing mini servo:

The evening before our presentation day, disaster struck as we observed (by sight, sound, and smell) our new mini servo (which was already painstakingly replaced once before) struggling to open the gripper against too strong a rubber band. Immediately, we unplugged the power and removed the gripper, but upon testing we discovered that one of the internal gears had been damaged and the servo was now unreliable.

Because of how little time we had, we decided to forgo the process of dismantling the bot in order to replace the servo. Fortunately, it managed to perform as intended during our presentation, however we would still replace it to increase reliability of the gripper.

Easier way to create custom movements:

The current process of creating custom movements is effective, but slow. We did this by using manual control to move the arm into the desired position, then disconnected the servo input wires and instead sent the PWM signals to the oscilloscope. We could then read the duty cycle percentages, which corresponded to the numbers that we needed to store in registers 1 through 5 at each position to replicate the movement.

This took a lot of time because of the transition from manual movement to reading with the oscilloscope, as well as the math to convert the duty cycle percentage to a number from 0 to 99 that would be stored in the registers. We used the same equation as in the PWM serializer module to do this, just in reverse.

A better way to do this would be to have a “record” mode. This would involve another switch control that stored the PWM information in memory that would later be read to repeat the movements. This would be intensive on the assembly side, but would not need changes in Verilog. The record mode would trigger a loop that moves through the RAM and writes duty cycle values in groups of five corresponding to the five servos. This would be done at some fraction of a second (utilizing our custom delay instruction) to ensure smooth, continuous movement.

Once recording is turned off and the custom movement switch is enabled, we could loop through the values and use them as duty cycles accordingly.

Conclusion:

Jabber integrated hardware and software in a way that challenged and rewarded us throughout the project. From the start, we knew that we wanted a project that would build our knowledge in robotics and hardware, and in that we were successful.

Despite various obstacles along the way, we were able to deliver our minimum viable product – manual control of the arm – as well as added functionality in the custom movement mode. Our processor was integral to the project, from reading joystick inputs to providing delays so that manual control is possible.

Jabber was an engaging project that made use of our processor and taught us many practical and technical skills. It serves as a fitting end to our learning in 350 and a foundation for future projects at Duke.