Robotic Arm

Northeastern University
Boston, MA 02115
December 6th, 2018

Prepared By:

Fatema Janahi (janahi.f@husky.neu.edu)

Julia Gangemi (gangemi.j@husky.neu.edu)

Abstract:

For this project, both logic design and C++ code were used in order to control the robotic arm. Prior to the project, simulink was used to create a design which converts angles to PWM signals. In the first two parts, this was expanded by creating logic designs that change speed to pulses and then angles and speed together into a PWM signal. The third part then involved the upload of this design to the Zedboard using the workflow advisor in order to verify its functionality. The fourth part involved combining 5 of the final designs to each represent a different servo, then uploading this to the board as well. The fifth part built off this and the robot arm was programmed to pick up and toss an object using C++. The final part used the Wiimote as input to control the robot arm, combining both logic design and C++. All logic designs are included in the report, and all code and videos are attached.

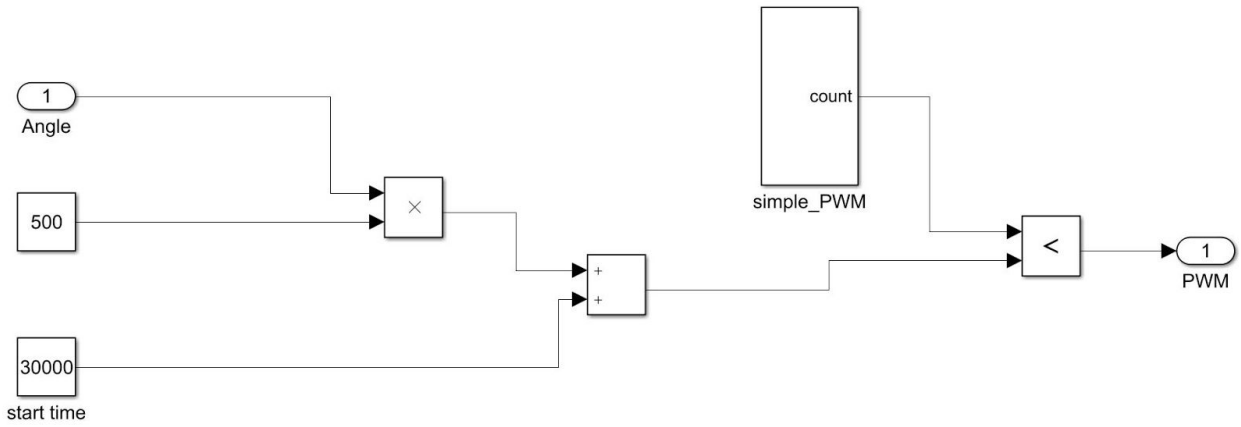
Introduction

This project served to combine everything we have worked with in Embedded Design in the Fall 2018 semester into one project. We combined hardware and software in one cohesive design in order to control the robotic arm. Simulink is used to build logic gates to turn angles and speed into PWM signals to control the robot through the FPGA. Then, the wiimote is connected by bluetooth and serves as the input for the arm. MobaXTerm is used to write C++ code that reads the button input from the Wiimote, which is connected to the Zedboard using the ARM processor, in order to move the robotic arm correspondingly.

Prior to the Project

Prior to starting the project, a design was created in Simulink that converts an angle to a PWM signal. To begin, the PWM signal generator from a class lab was used to build off of. The angle was taken as an input and then multiplied, using a product block, to a constant block with the value 500, which represents the number of cycles per degree. Then this value is added to a constant block with the value 30000 because that is the start pulse (the pulse for zero degrees). This sum is compared to the simple_PWM generator from a previous lab. If the simple PWM was less than this value, the output connected to the operator is true. That way, the output is true until the angle given is greater than or equal to the other PWM generator, than the output is false, creating a PWM that represents the given angle. All of these values were later scaled down for simulation purposes.

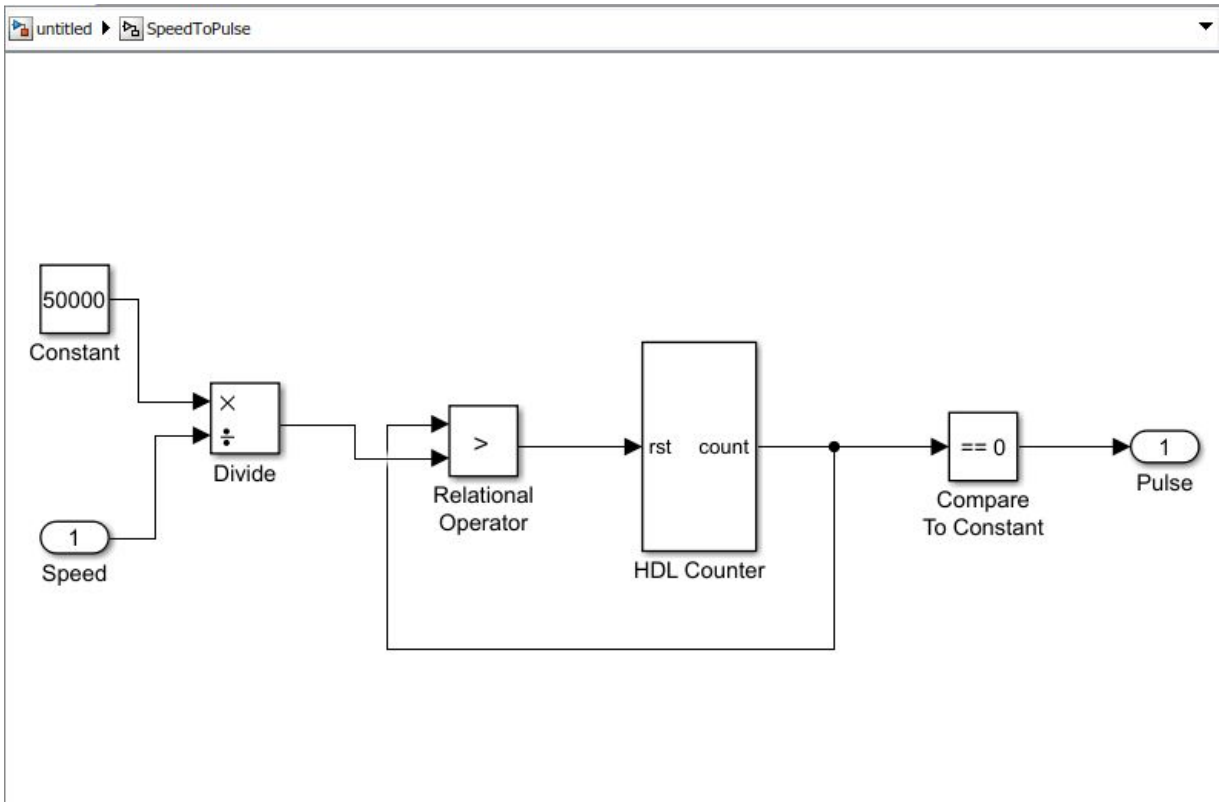
Figure 0.1 Design for AngleToPWM



Part 1

The first part of this project required building a logic design that changes speed, given as an 8 bit integer, to pulse in order for the robotic arm to understand the values. The speed is represented as an 8 bit unsigned input block, so that different values for speed can be send as input. A constant block with a value of 50000 (the number of cycles per minute for the simulation) is then divided by the speed and the quotient is sent to a relational operator, along with the value of a free running counter. If the free running counter is less then the value of the quotient, the relational operator sends 0 to the counter's reset port, thus the counter continues to count. If the counter's value is greater than the value of the quotient, then the relational operator sends a 1 to the reset port and the counter resets to 0. A compare to 0 block is then connected to the output of the counter, and an output block is connected to the output of this comparator. This means that every time the counter is reset, the comparator will output a 1. The higher the speed, the lower the compare to value will be, meaning the counter will reset more times, producing more pulses in a given period. The design can be found in Figure 1.1.

Figure 1.1 Logic Design for SpeedToPulse



The output was then replaced with a scope to better understand the behavior, and a constant block was used for the speed input. If the speed is lower, there will be less pulses as it will take longer to reset. If the speed is higher, there will be a higher pulse as the counter will reset more frequently. This behavior is tested twice, using 5 (Figure 1.2) and 20 (Figure 1.3) as inputs for speed to see how the scope behaves, and indeed a higher speed resulted in more pulses.

Figure 1.2 Test of Logic Design for SpeedToPulse with Speed 5

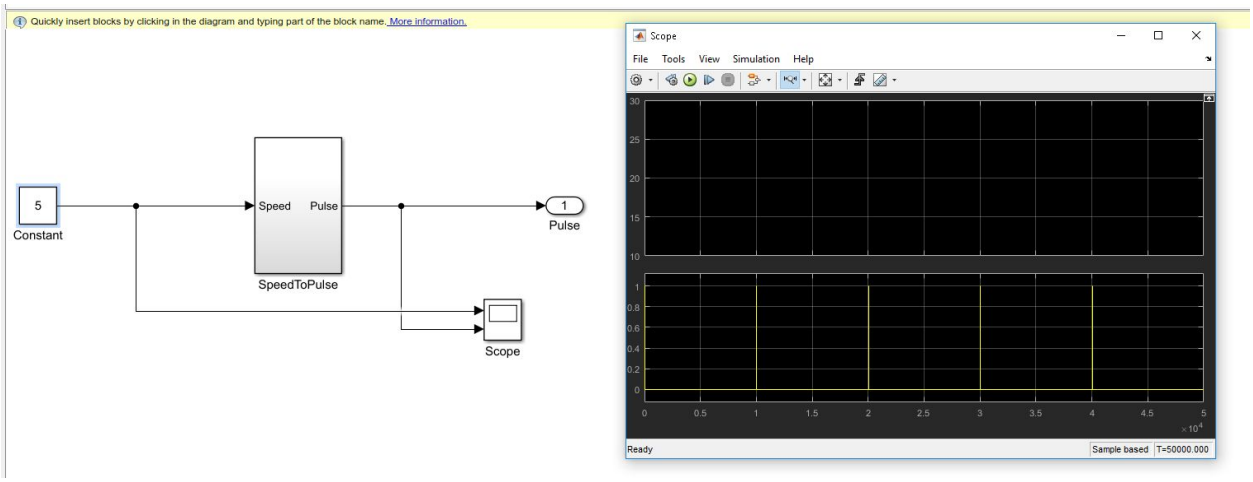
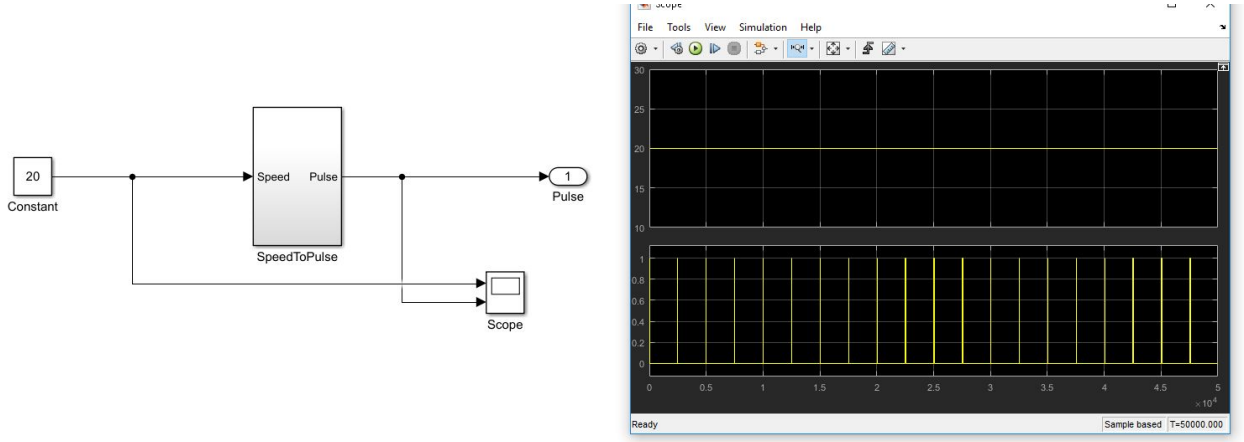


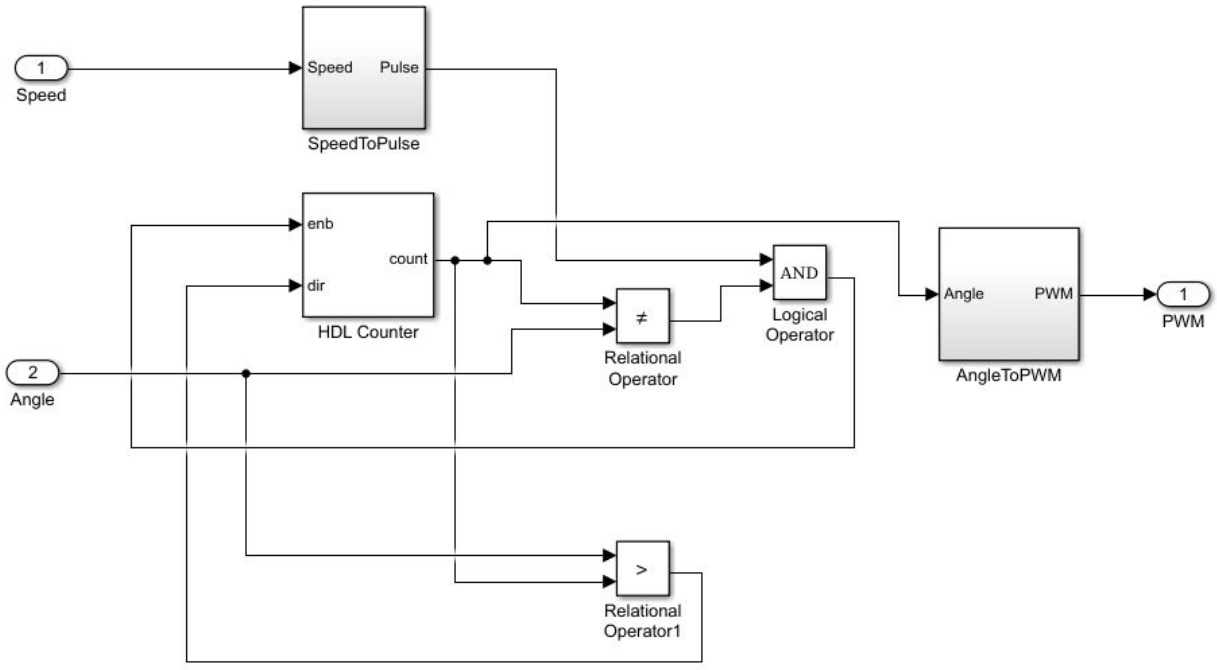
Figure 1.3 Test of Logic Design for SpeedToPulse with Speed 20



Part 2

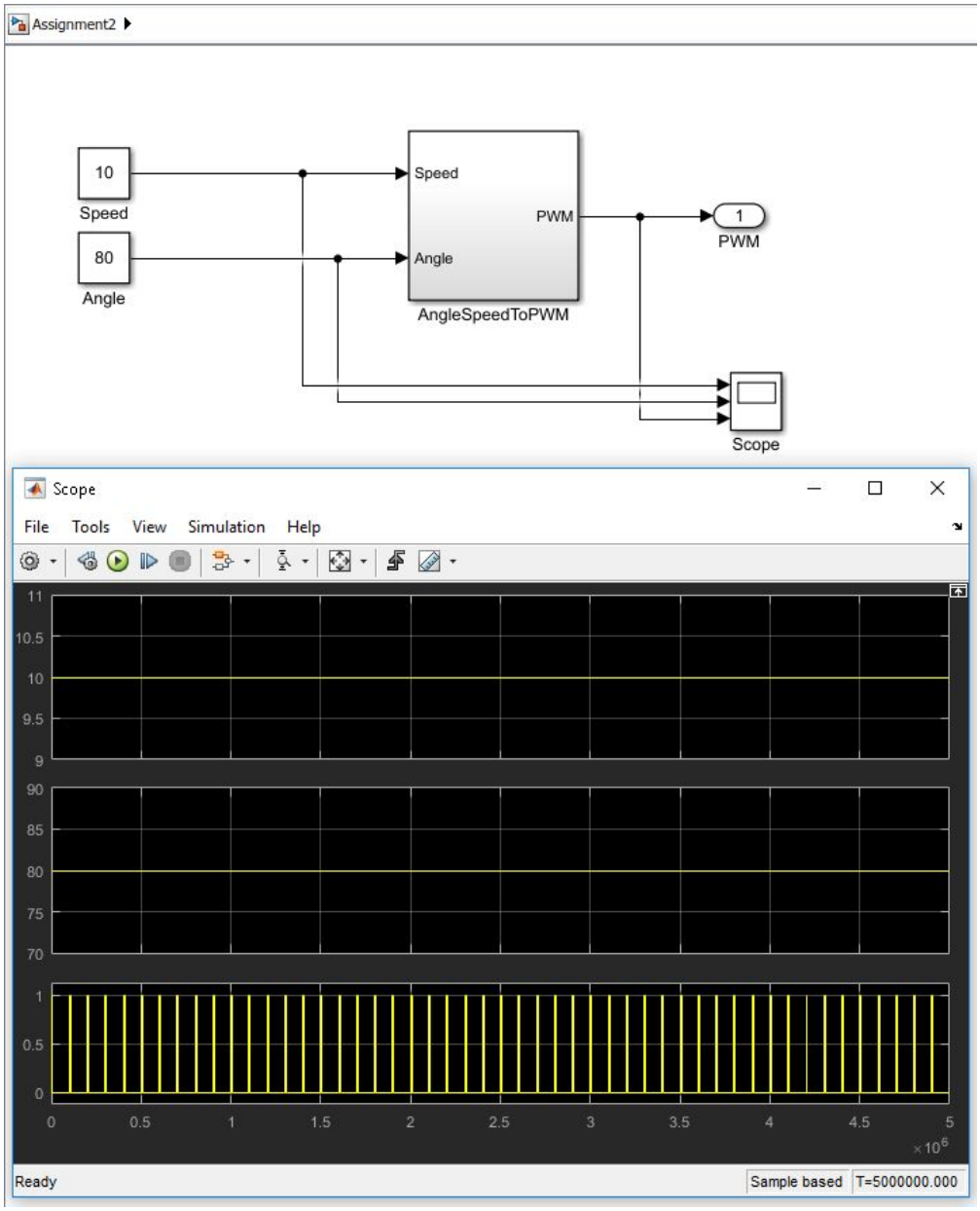
The next part involved creating another logic design that converts an angle into a PWM signal at a given speed. First, an HDL counter was used to hold the current angle the robot is at, and this was initially set to 90. The counter was also given both enable and direction input ports. The current output of the counter, 90, was connected to a relational operator that compares it to the angle input. If the angle input is greater than the count, a 1 is sent to the direction input, increasing the count. If it is smaller than the count, a 0 is sent to the direction input, decreasing the count. As for the enable input, its value was determined by comparing the count to the angle input using a "not equal to" comparator, so that it would be 1 until the counter reaches the inputted angle. However, this was not directly sent to the enable input, bur rather was sent to an AND gate along with the output from the SpeedToPulse block created in part 1. This way, while the input angle is not equal to the counter value AND the speed is 1, the counter is enabled, such that the count changes at the given speed, meaning the angle changes at the given speed. Furthermore, the counter output is connected to the AngleToPWM block created prior to starting the project, so that the the angle is converted to a PWM signal.

Figure 2.1. AngleSpeedToPWM Circuit Design



The AngleSpeedToPWM circuit was then connected to two constant blocks as the inputs and a scope connected to the output to assess its' behavior. The simulation was run for 1 second of virtual time with the speed set to 10 degrees per second and the final angle set to 80 degrees.

Figure 2.2 AngleSpeedToPWM with Angle as 80 and Speed at 10 with Scope



The first and last pulse of the scope were then zoomed in on to study the behavior of the pulse. The first pulse began at 0 and was on until 7450 where it dropped down to zero. The last pulse began at 4,900,000 and stayed on until 4,907,000, which was a difference of 7000. This makes sense because the start value is 90 degrees, which converted to cycles is 75,000 degrees, and since everything was scaled down by 10 it makes sense that the pulse would be on for 7450 cycles. Then, since the given angle is 80 degrees, the last pulse should represent 80, which it does because 80 degrees converted to cycles is 70,000, and everything is scaled down by 10, so 7000 makes sense for the width of the last pulse.

Part 3

Part 3 involved uploading the design from part 2 onto the ZedBoard. First, the values were scaled back up so that they match the frequency of the board (50 MHz). The constant block values were changed to a speed of 15 and an angle of 180, and the output was mapped to the port responsible of controlling the gripper, JA1[4]. Once the upload was complete, the gripper moved from 90 degrees to 180 at a speed of 15 degrees/second, verifying that the logic circuit created was correct.

Part 4

Part four involved creating a PWM Generator that controls each servo (Base, Bicep, Elbow, Wrist, and Gripper). This was done by creating a subsystem of five AngleSpeedToPWM blocks (Figure 4.1), such that the PWMGenerator takes in 10 8-bit inputs (a speed and angle for each 5 servos), and outputs five boolean values, one for each servo (Figure 4.2). Using the Advanced eXtensible Interface (AXI4), the inputs were configured to memory locations which the ZedBoard can access. Thus, C++ programs can be used to modify the values in the memory locations, in turn sending this data to the PWM Generator. Using the HDL Workflow Advisor, each input was set to a specific memory location, which can be seen in Figure 4.3.

Figure 4.1. PWMGenerator Circuit Subsystem

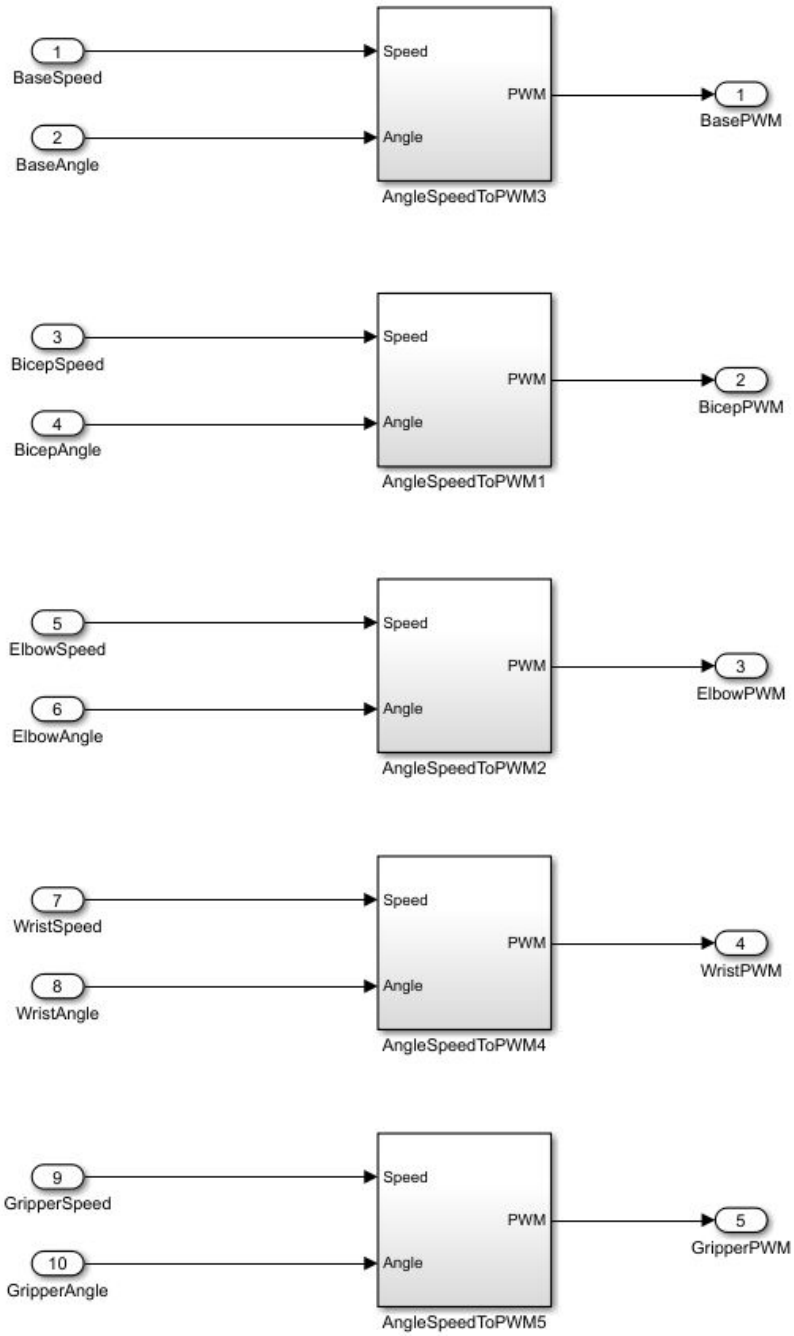


Figure 4.2. PWMGenerator Circuit

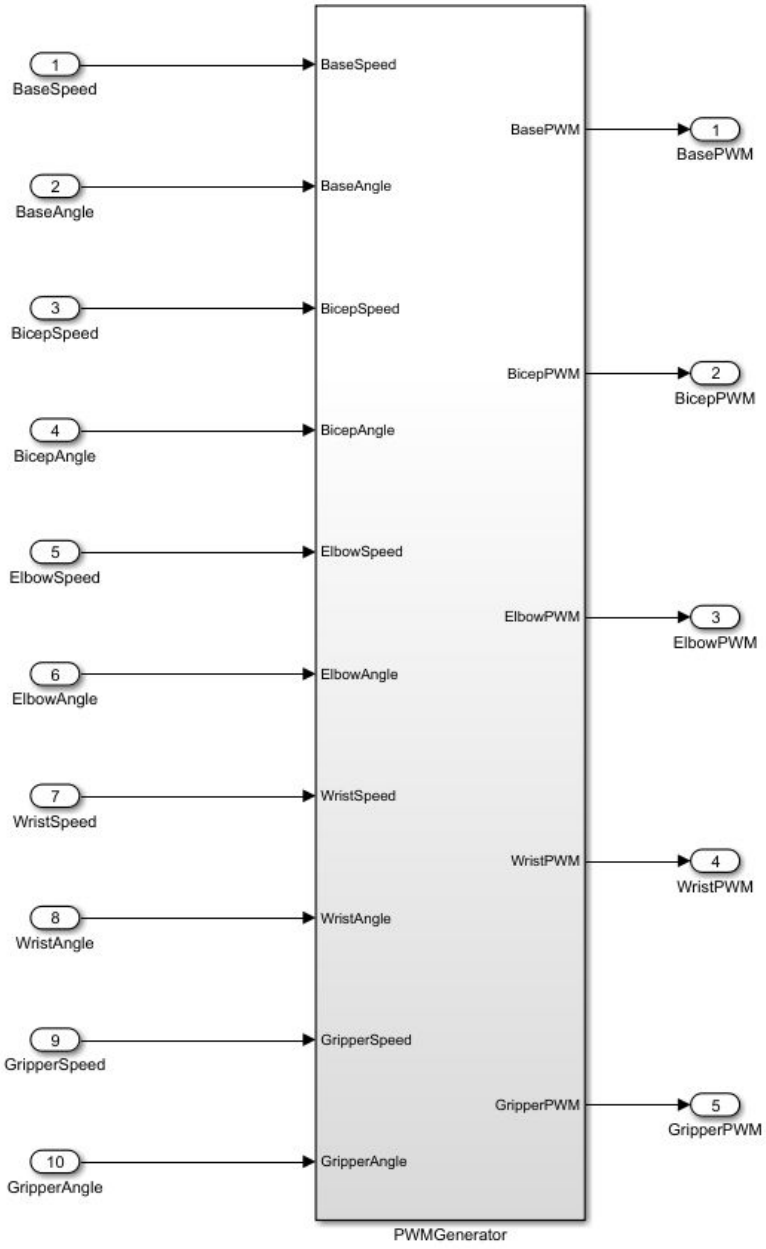
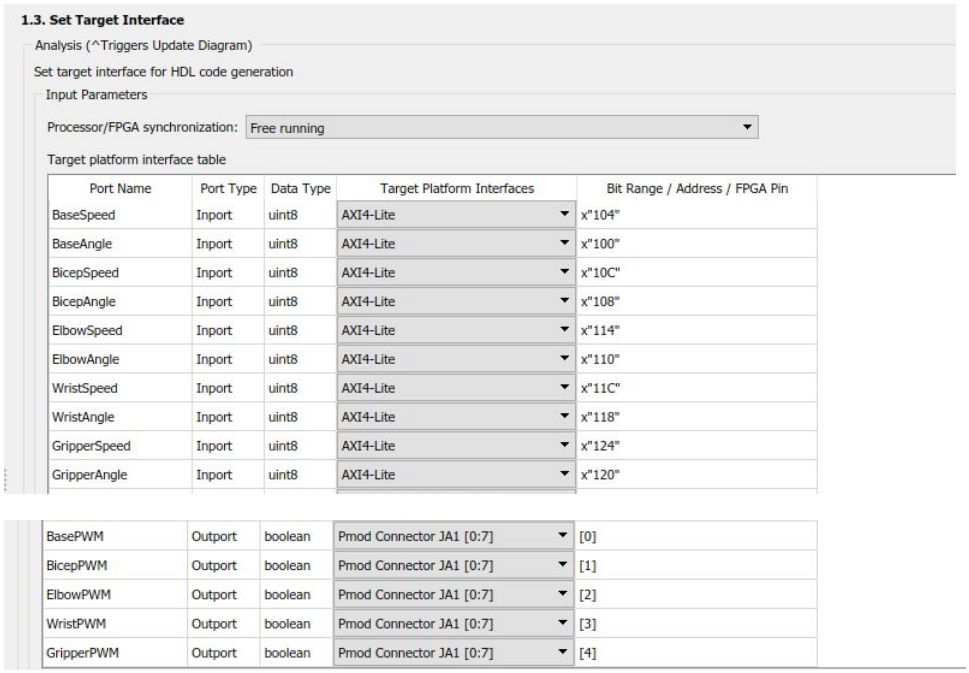


Figure 4.3. Input/Output Mapping to Memory Addresses and Servo Motor Ports



Part 5

Part 5 involved programming the robot through C++ to grab an object and throw it. Since the speed and angles of each servo were mapped to specific memory locations in the previous part, all that is needed is a program to modify these values according to how the robot should move. A header and cpp file for the RoboticArm movements were provided to assist in this part.

The header file included the address of the GPIO, the length of the memory-mapped IO window, and the offsets of each servo's angle and speed values, so that the program can access and write to those locations. In addition, a class RoboticArm is defined, with a private integer `fd` (to store the file descriptor of the memory mapped IO), a private char pointer `pBase` which points to the mapped address, and a private function RegisterWrite, which takes an offset and a value as parameters. Using the offset, the function can write or read from the memory locations. The public access modifier contains the constructor, destructor, and function MoveServo, which takes as parameters three integers, one for the servo id (0-4), one for the angle, and one for the speed.

The cpp file defines the RoboticArm constructor, destructor, and the contents of the RegisterWrite and MoveServo functions. The constructor initializes the mapped memory area and sets the servo positions to 90 degrees at a speed of 180 degrees per second using the MoveServo function. This is done in a for loop, since the servo id numbers range from 0 to 4. The deconstructor unmaps the physical memory and closes the mapped memory area. The RegisterWrite function sets the pointer that points to `pBase + offset` to equal the value parameter, thus sending different offset values allows writing the value to different servos. How this offset is determined is in the function MoveServo, which first checks whether the servo id is between 0 to 4 using an if statement, and exits the program if its not. If it is, it uses the `RegisterWrite()` function and sends the angle along with an offset of `(base_angle_offset + id * 8)`, as this equation expresses the offsets of the servo angles. It sends the speed with an offset of `(base_angle_offset + (id * 8) + 4)`, as this equation expresses the speed offsets of each servo.

In the main file, the RoboticArm header and unistd.h library were included. In main, an object of class RoboticArm called robotic_arm was created. For each servo from 0 to 4, the MoveServo function was called to set the initial conditions so that the arm is oriented towards the object it is to pick up (an eraser), and so that the arm is not too high or too low, and so that the gripper is wide open at 180 degrees. The values were determined through trial and error and can be found in the cpp files attached. The sleep command was used so that the next commands do not run before the robot is done moving. The bicep and elbow were then lowered so that the gripper is around the eraser, then the gripper was closed. The throwing movement was done by moving the bicep by 90 degrees and the elbow 40 degrees in the same direction, at a speed of 200 degrees per second. At the same time, the gripper was opened from at a slower speed of 75 degrees per second in order to let go of the eraser while the arm was moving. The functionality of the robot can be viewed from the attached video.

Part 6

For the sixth part, the robot's arm was now controlled using the Wiimote instead of manually plugging in values to the C++ program. Code from lab 4 was used in order to connect the Wiimote and read the buttons pressed. First, the shell script written in lab 4 to connect the Wiimote by bluetooth was used in order to connect the Wiimote. Then, the files written in lab 4: WiimoteBtns.cpp, WiimoteBtns.h, WiimoteAccel.cpp, WiimoteAccel.h, Zedboard.cpp, and

Zedboard.h, were all downloaded into the project folder along with the RoboticArm.cpp and RoboticArm.h. The header files were all included in the main file in order to access all of the functions in these classes. Although all of these files were included, the WiimoteAccel and Zedboard classes were not utilized in the final product, only WiimoteBtns was used. For further explanation of the code in these files, refer to lab 4.

The main file was then written to control the motion of the robot arm using input from the Wiimote. First, an object of RoboticArm was created in order to access functions of this class and communicate with the robot. Next, an object of WiimoteBtns was created in order to read the button values as input. The `Listen`() function from the WiimoteBtns class is then called in order to read the button inputs. This void function was modified from the previous lab, as before it just listened to the buttons, then printed the data. Now the function is an integer function that returns the value of the button code. In an infinite while loop, the `Listen`() function is called and the returned value is saved in an integer called `Bcode`. Then, 5 if statements are implemented to see which of the 5 buttons are pressed, up, down, left, right, or A, based on the value returned by `Bcode`, as they each control a different servo. The corresponding servo, based on the button being pressed, is then saved into an integer named `servo`.

After these if statements, 2 more if statements are used to see if the + or - button was pressed. If the + button was pressed, the integer representing the angle (originally set to 90) was increased by 10, and if the - button was pressed, the angle was decreased by 10. After the angle was changed, an if statement was called to see if the angle was out of the range 0-180, and if it was, the angle was truncated to remain in this range. Finally, the MoveServo() function from the RoboticArm class was called sending the number servo, the angle, and a speed of 45 as parameters, in order to move the correct servo to the desired position. A makefile was then used to compile each of these files into one executable file. The behavior of the robot can be verified by the attached video.

Conclusion

Many different components were all used in the implementation of this program. Using Simulink to build logic designs made it simple to convert angles and speeds to PWM signals. It also made it more convenient to map the inputs to corresponding memory addresses, so that they can be implemented through C++ code, and to map the outputs to corresponding servo motors on the arm. Additionally, by modifying the code written in previous labs, the Wiimote was used as an input to move the robot arm correspondingly. The skills used in this project combined everything learned during previous labs and will also be beneficial going forward.