FPGA CONTROLLED MAZE SOLVING ROBOTIC ARM

Thomas Edmonds

10622544

BEng Robotics

School of Engineering, Computing and Mathematics

University of Plymouth

May 2024

Abstract

This project examined the idea of using an FPGA as the main controller for a robotic arm. The idea behind this is to prototype a self-contained arm controller that can quickly respond to its environment with minimal outside control or input.

This project used Q learning implemented on an FPGA to facilitate fast processing of input data and used testbenches to simulate and test the HDL.

Practical testing was unfortunately limited to the simpler HDL for manually moving the arm with no automatic control.

The arm proved unsuitable and caused significant difficulties and delays and it is recommended that others hoping to do something similar use different simpler hardware, such as a remote-controlled car/buggy or a maze made of LEDs to demonstrate the project's principles before attempting to implement the project on an arm.

This Project successfully proves that Q learning could theoretically be implemented on an FPGA and used to solve mazes and control a robotic arm, but falls short of proving it practically due to the FPGAs resource limitations.

This project's findings suggest that this idea might be worth pursuing but only by HDL experts who are able to optimise their work to a greater extent and or those who have the budget for an FPGA with more resources.

Acknowledgements

I would like to thank both the original project supervisor Dr Nicholas Outram and his replacement Dr Paul Davey for the invaluable guidance they offered throughout this project. I would also like to thank Dr Ian Howard for his assistance on the topics of kinematics and machine learning and also for writing the original coursework that most of the machine learning elements were based on.

I would also like to thank the University of Plymouth technical staff in particular, James Rogers for his support with the robotic arm and 3D design/printing, John Welsh for his help finding suitable isolators, George Seymour for attempting to repair the original robotic arm, Martin Simpson for support with Quartus and Andrew Norris for general technical support.

Finally, I would like to thank arnaudeveloper on GitHub as I used their seven-segment display Verilog module to save time, the author of this article https://projectf.io/posts/fixed-point-numbers-in-verilog/ as it was very helpful when trying to get the Q Learning working, the creators of this https://chummersone.github.io/qformat.html#converter fixed-point calculator for saving me a lot of time when working with fixed point numbers.

Contents

Glossary

FPGA: Field Programmable Gate Array

EMF: Electromotive force

PCB: Printed Circuit Board

GPIO: General Purpose Input Output

DOF: Degrees of Freedom

1 Introduction

Robotic arms are often used in industry to complete repetitive tasks such as assembling components or packing items into boxes, but what if the box is off centre or the component is upside down? With typical dumb robotic arms, the component would be welded upside down and the item would be put next to the box or in the wrong place.

This project hopes to provide a solution to these problems by creating a prototype robotic arm controller that can react and adapt to its surroundings on the fly with little to no external control. As a proof of concept, the arm should be able to use the colour data from a photo to map a maze, Q-Learning to solve it and inverse kinematics to move through it.

In the context of this project having more states in a maze of the same size would increase the resolution, a more advanced version of this project would run a second Q-Learning algorithm using the same number of states within the destination state to fine tune the position of the end effector.

An FPGA is used for three main reasons. Firstly, they are significantly faster than microcontrollers particularly for complex mathematics due to their parallel processing capabilities. Secondly, they are far more flexible, where additional peripherals or microcontrollers may have been required to solve new problems or meet requirements as they appeared, an FPGA can complete most tasks on its own. Lastly, while this project could have been completed easier and faster using a microcontroller the results wouldn't have been as good/useful as the controller would have been slower to adapt/respond to its environment.

2 Project management & Planning

The majority of this project was to be done during the 4-month period from the 22nd of January to the 2nd of May, with some initial planning done during October/November.

This project can be broken down into six main tasks:

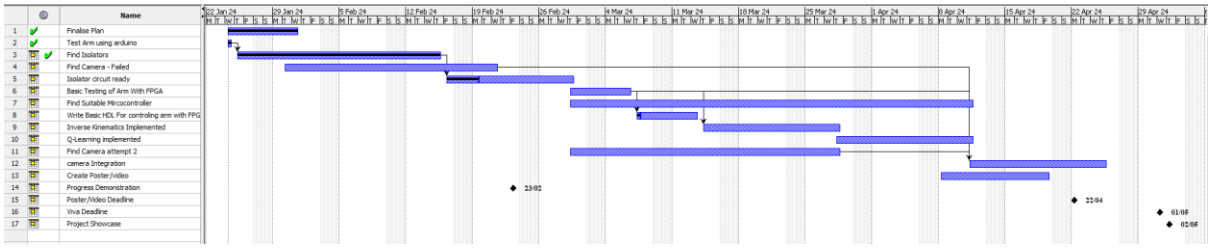
1. Setting up and testing the robotic arm,
2. Simple control for the arm using the FPGA,
3. Calculating inverse kinematics and sending that data to the arm,
4. Setting up Q-learning on the FPGA,
5. Setting up the camera and sending image data from the microcontroller to the FPGA,
6. Combining all of the above work.

During the initial planning period a SainSmart robotic arm was selected from the project store as there wasn't enough budget to buy a better one and wasn't enough time to build a new one. Unfortunately the SainSmart wiki where most of the documentation for the arm was found is currently inaccessible and doesn't seem to be archived on the Wayback Machine. As a result the relevant files are available in the GitHub repository for this project [1]

A DEO-CV Cyclone V development board was also booked out of stores for the FPGA as it was the most powerful available and buying one of equivalent or greater power/speed would have taken the project significantly over budget. [2]

It was decided that as the camera and microcontroller were not going to be used until the final stages of the project and were mostly an extension of the original proposal, that they would be acquired closer to the end of the project when a clearer picture of requirements could be found.

As part of preparing for the project this Gannt chart was created to help with project management, unfortunately due to numerous unforeseen delays and extenuating circumstances this original schedule couldn't be followed.



Figure[1]: A Gannt chart for planning the project.

A risk assessment was also completed in preparation for the project. Thankfully this project was relatively low risk with no real standout precautions that needed to be taken. [X]

3 Initial arm build

Once project work fully started in January the first step was to test the arm's servos and find out what needs to be done to control it with the FPGA, by controlling it with the included Arduino.

The first step was downloading example code and the user guide from the SainSmart wiki. Unfortunately as previously stated, as of the time of writing the SainSmart wiki is no longer available. As such these files have been included in the project GitHub repository [1].

Unfortunately, it was immediately clear that there was a problem with the original design which was confirmed with the help of the lab technicians. The power supply board included with the arm was not capable of protecting the Arduino from the back-emf generated by the noisy servo motors and as a result two Arduino boards were overloaded before this issue was found. Because of this fault an isolator circuit was required, especially because while replacing the cheap Arduino was easy, replacing the FPGA would have been expensive and probably taken too long. Between finding the required isolators, completing work for other modules, and getting them delivered it took almost a whole month to get these isolators with them arriving on the 22nd of February.

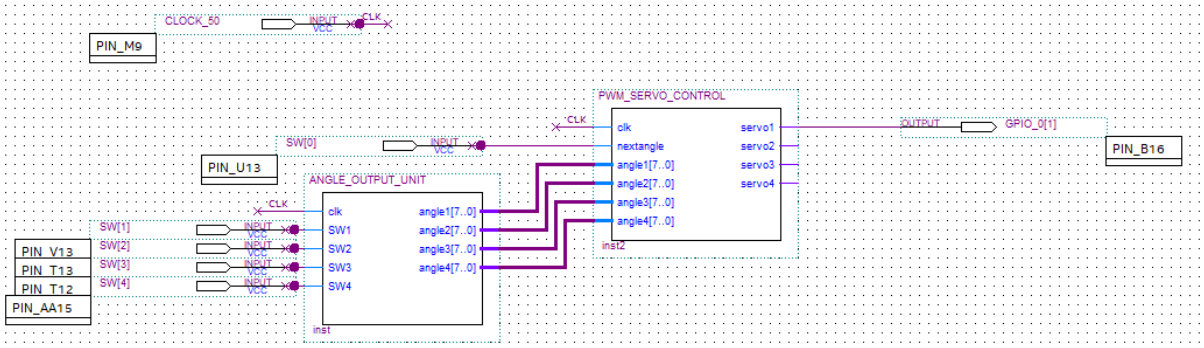
As a result of the power supply problem described above it was never possible to test the arm with the provided example code for more than a few seconds and as a result all testing had to be done using the FPGA.

3.1 Basic Testing HDL

To test the arm and isolation circuit some simple HDL was written using Quartus as can be seen in the in version 0.0.1 of the GitHub repository. This code simply outputs different PWM signals to the output GPIO pin depending on which switches are active. It works by using a simple timer to enable the output when the value state_1 is greater than the maximum value, minus the angle multiplied by 515 which is the number of microseconds per degree of movement, divided by the period of the clock where the maximum value is the 20ms period of the PWM signal converted to microseconds and divided by the period of the clock.

The input angle value was determined by the position of switches 1-4 on the FPGA, each angle was associated with a combination of switches for example 0 degrees was 0000 and 180 degrees was 1111. This was done rather than giving each angle its own switch because if each angle was given its own switch and none of the others were checked, then if multiple switches were active only the first angle in the else if statement would have been sent.

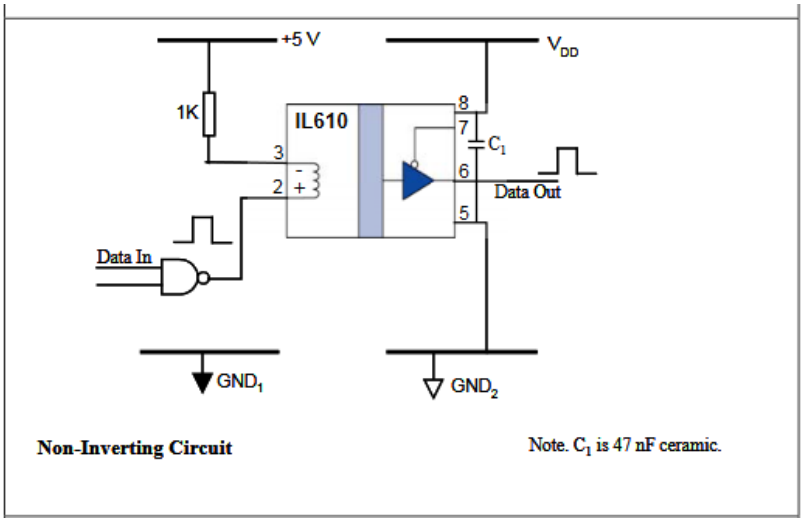
The schematic diagram for this HDL can be seen in figure[2] below.



Figure[2]:schematic diagram for simple testing HDL.

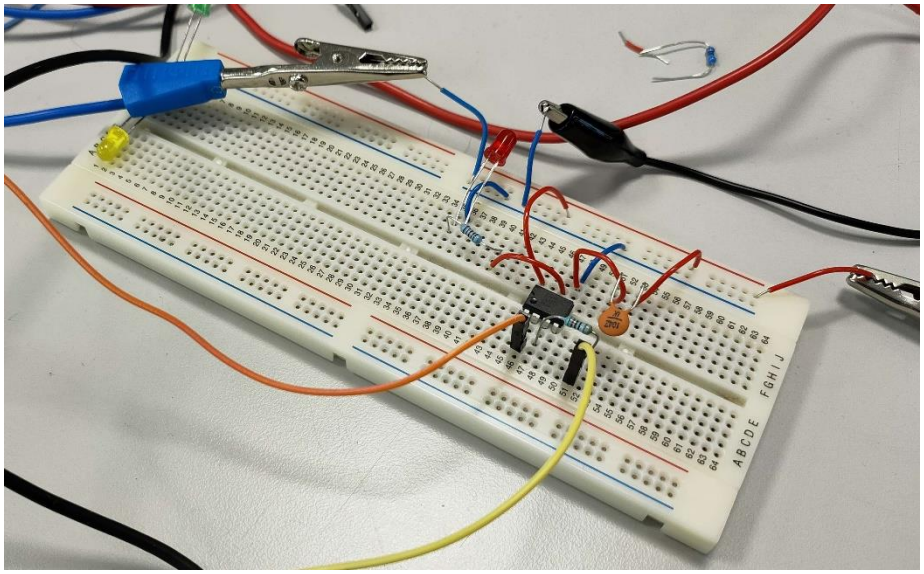
3.2 Isolator circuit design/build/testing

Magnetic isolators were chosen for their low cost and the circuit seen in figure [3] below was built.



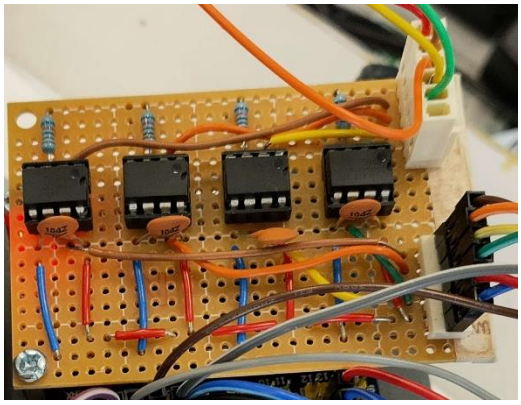
Figure[3]: Non-Inverting circuit diagram (IL600 Series Datasheet [3])

Initially a test circuit was built on a breadboard to confirm that the circuit would work as intended. This circuit was tested, first by using a simplified version of the testing HDL written earlier where a switch on the FPGA was connected to an LED, and then using the full testing code descirbed in section 3.1 with an osciliscope as seen in the videos found in the refrences [4, 5]



Figure[4]: breadboard with prototype Isolator circuit on it

Once this simple circuit had been tested a full circuit with four isolators for the four key servomotors was soldered on a perfboard. Perfboard was used here rather than a PCB as getting a PCB made would have taken a lot of time. Planning/testing this isolator circuit had already used up a lot of time, meaning none of the other project work could be tested until it was completed on the 6th of march.

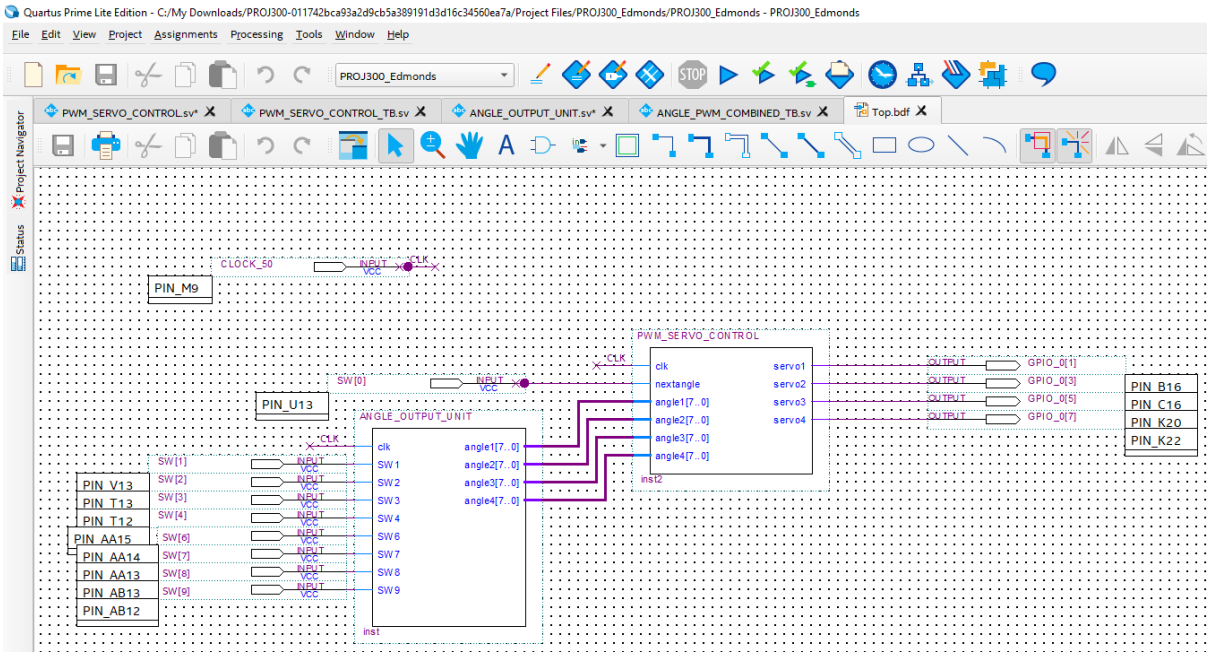


Figure[5]: Full Isolation circuit fully assembled and attached.

Unfortunately when testing this circuit with the FPGA it no longer worked, this fault was noticed when the isolator circuit was built on the 7th of march. As a result much of the lab time was spent trying to fix it. Initially each soldered wire was connectivity tested using a multimeter however no faults were found. So over the next week finishing on the 12th the HDL code which had been modified to add four separate outputs was tested using testbenches in preperation for the lab session on the 13th.

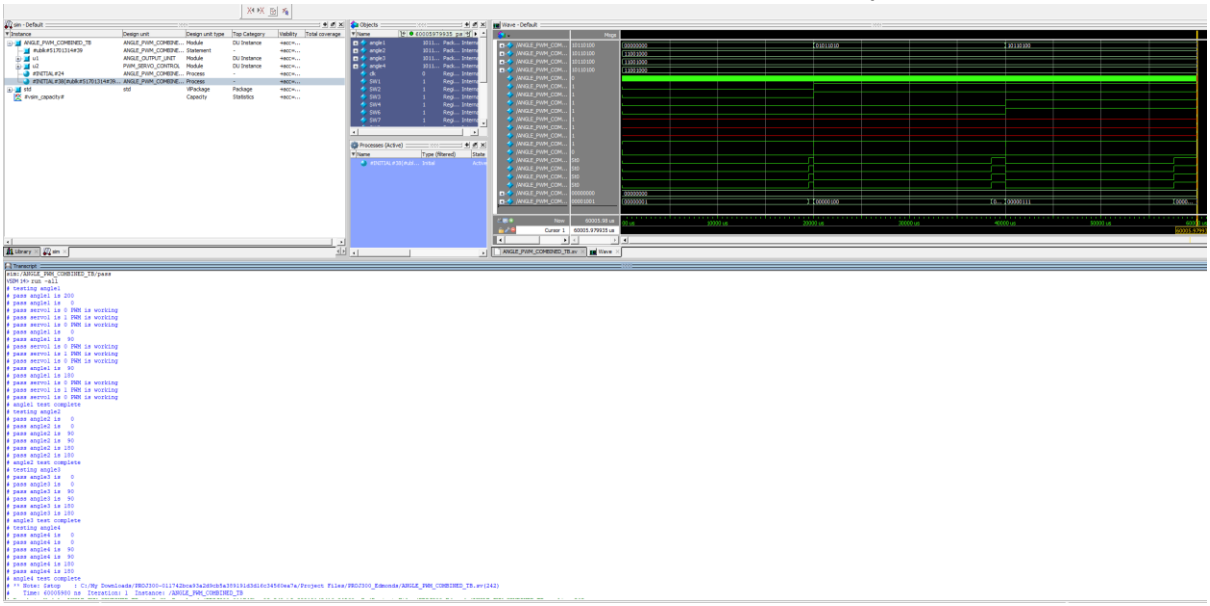
3.3 Modified HDL, Testbenches & Further Testing

As mentioned at the end of section 3.2 a modifed version of the testing HDL was created; as before this modified code can be found in version 0.0.4 of the GitHub repository [1]. This modified version of the HDL used three extra values which were compared to the same timer as before rather than creating three new timers. This modified HDL also used four extra switches to determine which servo should be moved.



Figure[6]: Schematic diagram of modified testing HDL

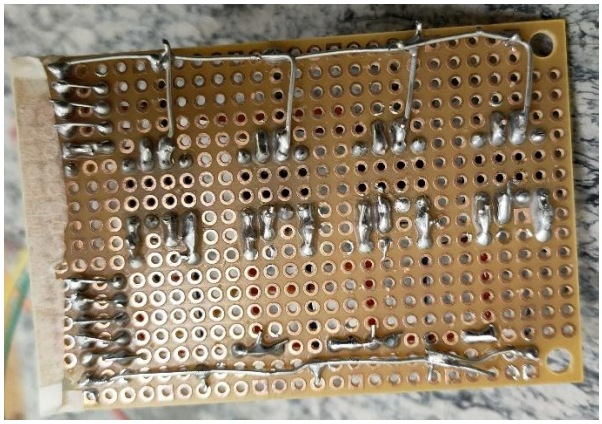
Once it became clear that there was something wrong with the modified modules two testbenches were written to test the individual modules and then a single complete module was created by combining the HDL from them. This testbench ANGLE_PWM_COMBINED_TB initially tests each motor to confirm that the correct PWM signal is being output, it then tests each angle to make sure the correct output signal is produced for each combination of switches, as can be seen in the waveform screenshot figure [7] below. This result meant that the fault had to be either in the wires connecting the FPGA to the isolator circuit, on the isolator circuit itself or on the physical FPGA hardware.



Figure[7]: Output of testbench ANGLE_PWM_COMBINED_TB

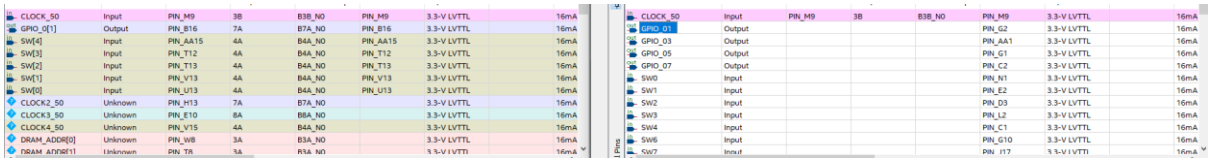
During the lab session on the 13th the connecting wires were connectivity tested to remove that possibility and then to make sure that there was no problem with the FPGA hardware

the isolator circuit was tested using the signal generator on an oscilliscope. This revealed that three of the isolator circuits actually worked with the first one having the output wire connected to the output enable pin as seen in figure[8], this mistake was quickly fixed by resoldering the wire. Unfortunately this meant that the only remaining potential causes of the problem were either the FPGA hardware or Quartus not synthesising properly.



Figure[8]: Underside of isolator board showing incorrectly soldered wire

After many hours of testing the problem was identified as Quartus not correctly assigning pins, this can be seen in figure[9] where the left side of the image shows the previous working versions pin assignments, and the right side shows the new version's pin assignments. To try and fix this the project supervisor Nicholas Outram was asked for help but as he was out of office at the time he could only provide limited support. Following his advice a separate LED pin was defined and tested this pin was assigned correctly and the LED worked, to get further support a meeting was booked the next day to discuss the issue further.



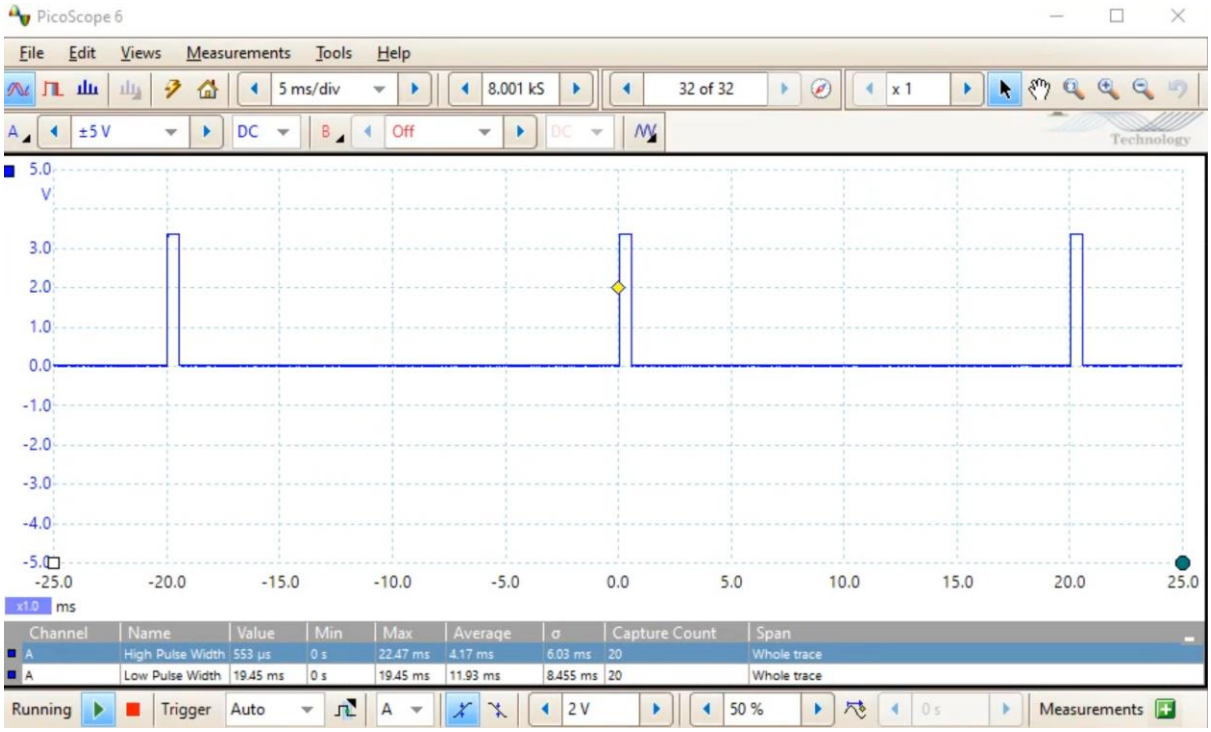
Figure[9]: Pin assignments for previous version on the left & new version on the right

During the meeting Nick was unable to identify any issues with the project files that could have been causing this problem putting it down to most likely being caused by the changing of versions between the lab where Quartus 20.1.1 is used and home where Quartus 23.1 is used, as a result he was only able to offer the solution of manually assigning each pin which is a work intensive task. Fortunately, shortly after the meeting while preparing to start manually assigning pins it was noticed that the compilation log had warnings about an incomplete GPIO bus, as a result of noticing this the missing pins were defined and on the next compilation they were assigned perfectly as can be seen in figure[10].

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	er Analog Settings	GXB/VCCT_GXB V
CLOCK_50	Input	PIN_M9	3B	B3B_N0	PIN_M9	3.3-V LVTTL		16mA (default)				
GPIO_0[7]	Output	PIN_K22	5B	B5B_N0	PIN_K22	3.3-V LVTTL		16mA (default)	1 (default)			
GPIO_0[6]	Output	PIN_K21	5B	B5B_N0	PIN_K21	3.3-V LVTTL		16mA (default)	1 (default)			
GPIO_0[5]	Output	PIN_K20	7A	B7A_N0	PIN_K20	3.3-V LVTTL		16mA (default)	1 (default)			
GPIO_0[4]	Output	PIN_D17	7A	B7A_N0	PIN_D17	3.3-V LVTTL		16mA (default)	1 (default)			
GPIO_0[3]	Output	PIN_C16	7A	B7A_N0	PIN_C16	3.3-V LVTTL		16mA (default)	1 (default)			
GPIO_0[2]	Output	PIN_M16	5B	B5B_N0	PIN_M16	3.3-V LVTTL		16mA (default)	1 (default)			
GPIO_0[1]	Output	PIN_B16	7A	B7A_N0	PIN_B16	3.3-V LVTTL		16mA (default)	1 (default)			
LEDR[9]	Output	PIN_L1	2A	B2A_N0	PIN_L1	2.5 V		12mA (default)	1 (default)			
SW[9]	Input	PIN_AB12	4A	B4A_N0	PIN_AB12	3.3-V LVTTL		16mA (default)				
SW[8]	Input	PIN_AB13	4A	B4A_N0	PIN_AB13	3.3-V LVTTL		16mA (default)				
SW[7]	Input	PIN_AA13	4A	B4A_N0	PIN_AA13	3.3-V LVTTL		16mA (default)				

Figure[10]:Correct pin assignments after the problem was fixed

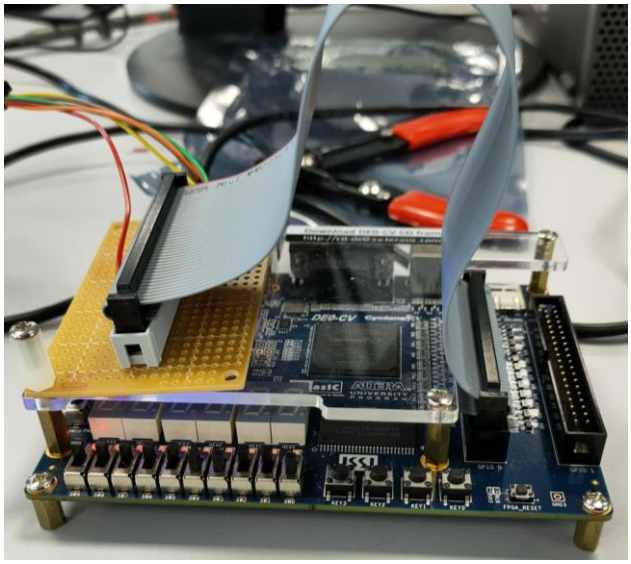
This was then checked using a Pico Scope and this testing proved that the fix had worked as seen in figure [11]. The FPGA was then connected to the isolation circuit and the arm worked near perfectly.



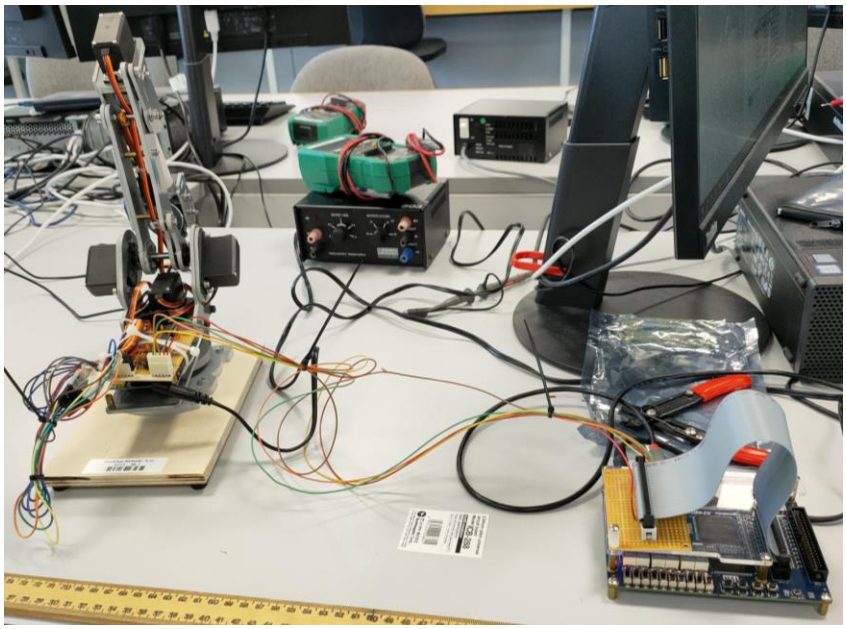
Figure[11]: Pico Scope output showing PWM signal

3.4 Initial Build Final Touches

While testing the now working circuit it was noted that when the arm moved it dragged the isolator circuit with it, sometimes even dragging it over the FPGA causing a short. Luckily this never caused any major damage, but it was identified as a significant problem and as such a GPIO header was soldered to perfboard which could then be attached to the FPGA using spacers and a ribbon cable as seen in Figure [12]. This was then connected, using long wires with a lot of slack to the isolation circuit which had been attached to the arm's power supply board again using spacers as seen in Figure[13]. This allowed the arm to move freely without dragging around the FPGA or isolator circuit. It was also one of the last pieces of project work completed before the original project supervisor left the university.



Figure[12]: Perfboard and ribbon cable attached to FPGA with spacers.

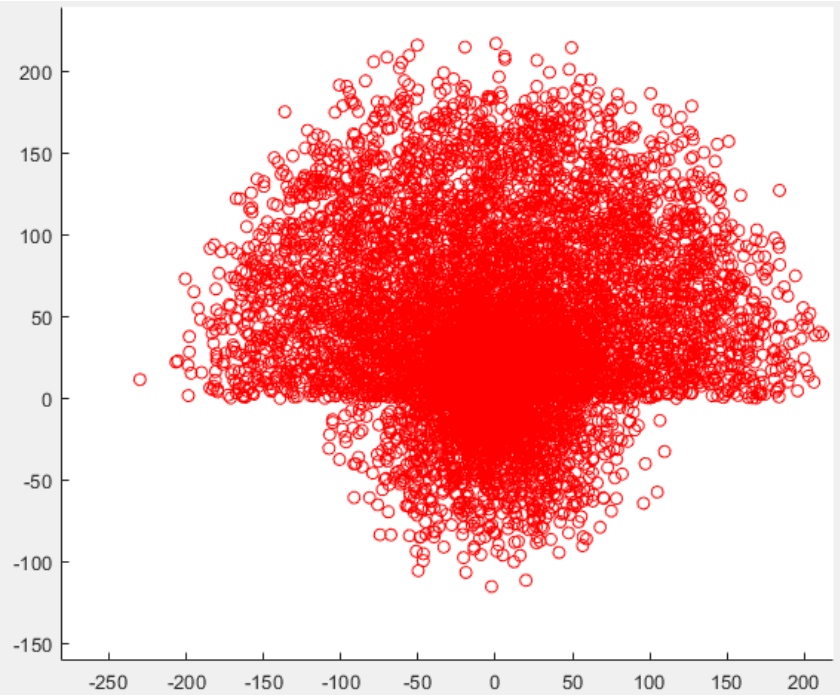


Figure[13]: Arm and FPGA connected with new boards and wire with slack

4 Machine learning

The initial plan for finding the angle values required to control the arm and send the end effector to specific states was to use machine learning to estimate the inverse kinematics values. This plan was based on the coursework for the machine learning module ROCO351 where a simulated method for doing this was used to calculate the inverse kinematics for a simpler 2DOF arm.

The initial concept was fairly simple and with some help from Ian Howard the forward kinematics for the 6DOF arm were quickly found this time using a method based on the coursework for ROCO224 . This work can be seen in the MATLAB files Forward_Kinematics_Test_main, Forward_Kinematics_test and TVec on the GitHub repository. These modules create the kinematic matrices and multiply them to find the end effector location, for any number of randomly chosen angle values. When plotted on a scatter diagram these end effector location values seem to show a fairly accurate depiction of the arm's reach as seen in figure[14] below.



Figure[14]: Estimated end effector location scatter diagram.

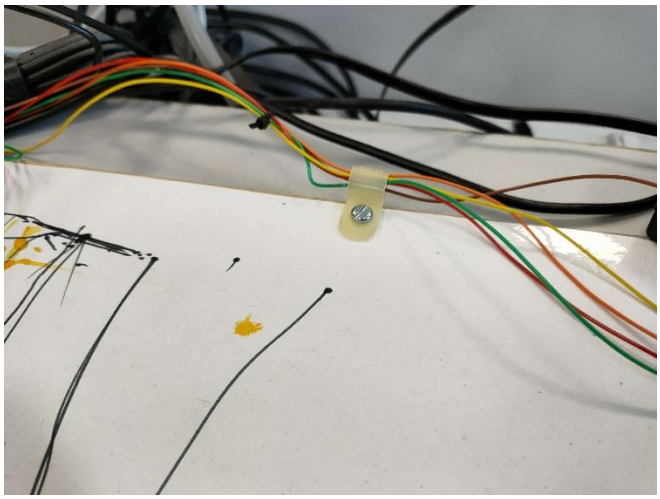
Unfortunately adapting the existing algorithm for a 6DOF arm and three-dimensional movement proved extremely challenging and quite time consuming. As a result after almost a week of struggling, with less than a month left on the project and a failed attempt to create a new algorithm from scratch this part of the project was scrapped.

5 Late arm build

Once the attempts at machine learning were finished it was noticed that unless the project was on a base board the ink from the pen could soak through the paper and mark the desk and also that the required angle values could potentially change if the arm was knocked, or the maze was in the wrong place.

Three pieces of MDF were found and holes were drilled for the FPGA, arm and for screw down wire clamps as seen in Figure[15]. Larger holes were then drilled on the back so the nuts could sit flush, and the board would sit level. This was important as if the board had wobbled then the arm's movement could have caused greater inaccuracy in the end effectors position.

Finishing the board took roughly two days due to issues getting the right size drill bit and finished on the 24th.



Figure[15]: Wire clamp attached to board keeping wires out of the way.

5.1 Pen Attachment 3D Design & Print

Next a pen mount was 3D designed and printed to attach the pen to the end effector. This design had two iterations, first a pen sized tube with screw holes was designed to be bolted onto the servo horn of the end effector as seen in Figure[16]. This first design had flaws however, as the tolerance on the diameter of the tube was too tight and the pen barely fit. This also caused the problem of not fitting other size pens, when the first pen was found to be too stiff its replacement did not fit in the holder. Additionally, the bolt holes on the base were also too close to the tube and neither the head of the bolt or a nut would fit in the space without significant filing.

Figure[16]: Original 3D printed pen mount.

As a result of these issues a replacement was designed and printed, this time rather than using a tight tube to hold the pen a loose a semi-circular platform was placed below the bolt holes leaving plenty of space to avoid the same issues suffered by the first design. With this new design the pen was instead cable tied to the platform making it easier to attach and also to replace with pens of different sizes. This can be seen in Figure[17] where it is attached to the arm's end effector.

Figure[17]: Final pen mount attached to the end effector.

6 Manual Control HDL

Instead of using inverse kinematics, a set of HDL modules were created that allowed the servos to be manually controlled using buttons on the FPGA with the current angle output on the seven-segment display. This HDL was written between the 25th and the 27th of April. It can be found in the GitHub repository[1].

First there needed to be a way to see what the current angle value of each servo was so that the angle values for each state could be written down, unfortunately at this point there was less than a week until project showcase and with a lot of work that needed to be done, the HDL for the seven segment display was sourced from GitHub (arnaudeveloper, 2018) [6] This code simply uses case statements to assign the correct binary value to each output and form specific letters/numbers by lighting up the correct segments on the display. The pins for the seven-segment display and push buttons were also defined on the top-level schematic diagram and were assigned correctly on compilation thanks to the lessons learned in section 3.3.

Next KEY_INPUT a module to register button presses was written. This module was based on and used many of the same principles as the ANGLE_OUTPUT_UNIT module, the main difference being that rather than using pre set angle values it starts with angle values of zero and increments or decrements them with each button press. Thankfully according to the DEO CV user manual [2], the push buttons are already debounced using a Schmitt trigger circuit which simplifies the module. The push buttons are high by default and pushed low when pressed, so the HDL checks on the positive edge of the clock which button is pressed and then uses a set of else if statements to check which motor's angle value should be changed, if the new angle value would be over 180 or under 0 and if the current real angle is equal to the previous temporary value. Once one set of these checks have been passed a temporary holding value for the respective angle gets set to the current value ±3, this new value is then passed to a second always block.

The second always block checks on each positive clock edge if the push buttons are all unpressed, if they are it checks which servo should be moved and that the temporary value is different to the old one, if so It is assigned otherwise the previous value is held. The check to see if the temporary value is different to the current value is not really necessary, but it was one of many attempts to stop the angle value from instantly hitting the maximum or minimum value as soon as the button is pressed. It was kept as it shouldn't cause problems and there wasn't much time left once this was working, so it wasn't worth optimising at the risk of breaking the project.

The final module for manual control ANGLE_DECODE takes the angle value and breaks it down into its individual digits, again it uses a case statement to determine which motor is being moved based on the active switch. The angle values for the current angle are divided into their individual digits, by calculating the modulo of the angle for the hundreds and then taking the modulo of the angle divided by 10 and 100 for the tens and ones respectively. The modulo command is very resource intensive on FPGA's and so an alternative method should be considered, one that has been used before would be to use a very long case statement that checks if each number is within a range of numbers for each digit. This method could potentially be used to optimise the design however it is very time consuming to write and it's

very easy to make mistakes during the process, which is why in this situation where time was limited modulo was used instead.

7 Q-Learning

The core of this project was implementing Q-Learning on the FPGA.

Q-Learning approximates the optimal policy Q, Once the Q values for each action in each state have been found then the optimal action for each state can be found.

The Q-learning work for this project was based on work completed as part of the ROCO351 coursework, so the method definitely works.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure[18]: Sutton and Barto Q-Learning Pseudo-code (Sutton and Barto, 2018)

The pseudo-code for Q-Learning (Sutton and Barto, 2018) is shown in figure[18] from this a rough outline of the HDL modules can be produced;

1, Initialise Q: modules INIT_Q and BLOCKED_STATES,
2, Loop for each episode: module Q_LEARN_V2,
3, Choose A from S: module MAXQ_REWARD,
4, The rest: modules MAXQ_REWARD, NEW_Q and Q_TRIAL.

1, INIT_Q uses a for loop to assign 148 individual 32bit numbers to the actions of each state, so the first row of four numbers is assigned to the action values of state 0 and the next four are assigned to the action values of state 1. This is done to initialise the Q values as generating actual random numbers on an FPGA is both complicated and not truly random, as a result it was determined to be an unnecessary use of limited time that can be implemented later. The numbers were generated with a MATLAB script RandQ, four extra numbers with a value of 0 were then added to the start as state 0 is unused. It assigns the numbers using a modified piece of code generated by ChatGPT, this was an experimental use of ChatGPT to speed up workflow, ChatGPT was also used to convert the numbers to the correct format, these two uses of ChatGPT turned quite a tedious but non complex task into one of the quicker modules to create.
BLOCKED_STATES takes the values assigned by INIT_Q and replaces the Q values for actions that lead to blocked states with 0. This is done by creating an array that contains the

numerical value of each blocked state, a for loop then goes through each value in the blocked array and checks it against a case statement, if the value is in the maze then it replaces the action values that lead to that state, otherwise all Q values remain the same. As of writing this report it also changes the Q values of actions that lead outside of the maze to zero an oversight that was only just noticed, with this change the checks in Q_TRIAL should be completely redundant.

2, Q_LEARN_V2 was the first attempt at writing module connections and wires in HDL rather than using a schematic diagram, this was done because a multidimensional bus was required and Quartus struggles to synthesise multidimensional busses from schematic diagrams. This module combines all of the Q-Learning modules and uses a generate statement to create a loop from MAXQ_REWARD, NEW_Q and Q_TRIAL.

3, MAXQ_REWARD takes the Q values for the current states and uses if statements to determine the highest, it then outputs the action associated with the highest value. This module also outputs the reward, but only for a specific final state due to time constraints.

4, NEW_Q takes the current Q values and applies the equation shown in figure[19].

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$$

Figure[19]: Pseudocode for new Q

This is done simply by applying the required mathematical actions in order using intermediary variables. Any multiplications have 64bit intermediary values as when multiplying fixed-point numbers the output is double the size, (Will Green,2020[6]) these numbers are then trimmed by taking 16 bits off each end. If left to be trimmed automatically then the value is changed completely. The reward isn't in fixed point format, so it is shifted left 16 bits to convert it. If min_Q is negative then it is inverted to make it positive, it is then multiplied by the learn rate before being inverted back to a negative number. This is done as negative fixed-point numbers don't seem to multiply properly. Finally, the new Q value is made equal to the current one and the current actions Q value is updated.

Q_TRIAL takes the current state and the action and using a case statement determines the next state which is assigned if it passes a complicated if statement which determines if the next state according to that action would result in the next state being either a blocked state or outside the maze.

Most of these modules also have an input and output done value, these values are used to let the next module know that the previous one has finished. This stopped each module from running until the output values of the previous module were correct as once the final module finished it's values would be used for exploitation and if the modules were allowed to run immediately then the final modules could simply try to use an empty set of Q values to run.

7.1 Q Exploitation

Exploitation of the Q values to move the arm along the maze's optimum path is carried out using two modules ACTION_EXPLOIT and Q_TRIAL_EXPLOIT these modules are almost identical to the modules MAXQ_REWARD and Q_TRIAL respectively. The main difference is that ACTION_EXPLOIT doesn't output the reward and Q_TRIAL_EXPLOIT starts and restarts a timer based on if the arm is ready for its next instruction. These modules were tested using Q_LEARN_TB as seen in figure [20]. While it isn't entirely clear from looking at figure [20] the maze state is changing based on the Q values. Unfortunately, because the Q learning never randomly selects the next state and the Q learning doesn't really go through enough iterations, it doesn't work completely and it repeats instead of being able to reach the final state.

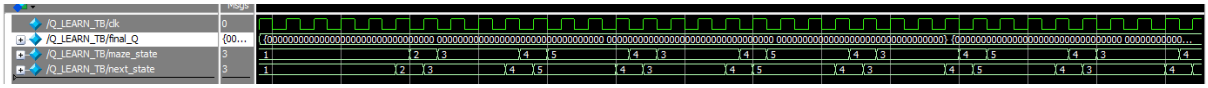


Figure [20]: shows the output of the Q Exploitation modules

7.2 Analysis

Unfortunately, until this report was being written the Q learning never worked properly, the new Q was correctly calculated when provided testing values and the initial/blocked Q values were correctly generated, but beyond that things started to go wrong. Testing using the testbench Q_LEARN_TB showed that after the first few loops the modules stop working. Fortunately, because the problem of not setting the Q values of actions that lead outside the maze to 0 was noticed while writing this report it does now work.

No greedy selection algorithm was implemented, a mistake that went unnoticed until project showcase day by which point it was too late to fix. As a result of this mistake the modules can get stuck taking the same actions, in some cases looping back and forth between states introducing the risk that a path could get blocked off if the loop ends going in the wrong direction.

As a result, if there was enough time implementing greedy selection would have been the next step.

The Q Learning wasn't heavily tested and doesn't have a very clear testbench as it was only fixed very late but the testbench output in figure [21] shows that final_Q is output, meaning the Q learning modules do at the very least work enough to reach the final episode and produce an output.

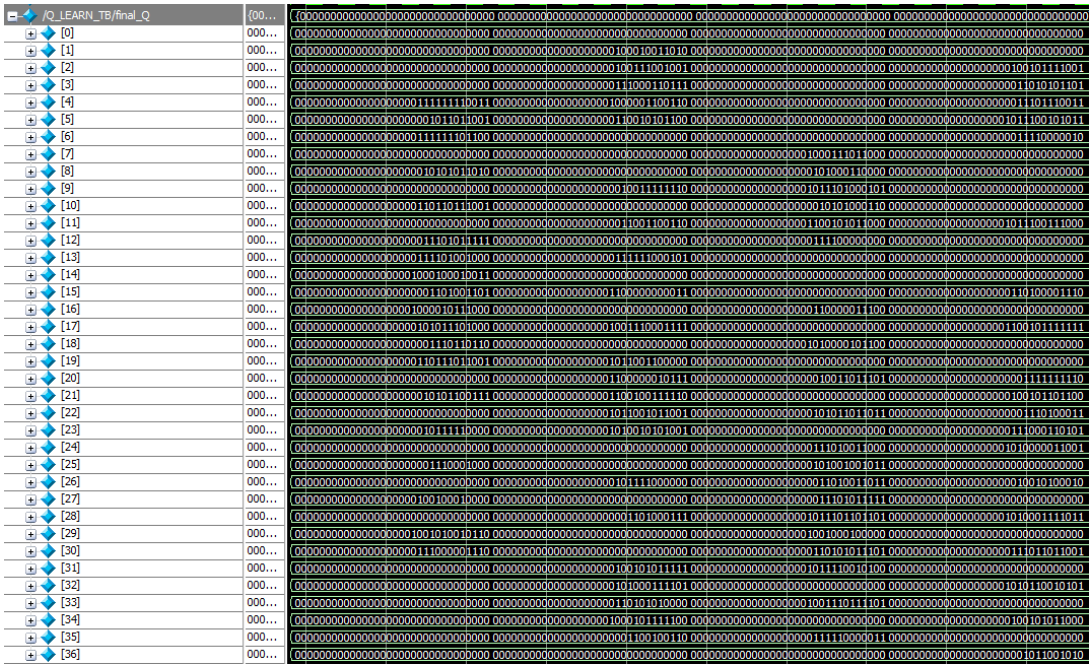


Figure [21]: This shows the final_Q output of the Q Learning TB module

8 Faulty Arm & Replacement

Throughout the course of this project there were numerous issues with the arm beyond the initial Arduino overloading problem.

Initially the PWM duty cycles found for the servo motors were incorrect and stated that 544us was 0 degrees and 2400 was 180 degrees. As a result, throughout most of the project the motors were trying to go over the limit when set to 0 or 180 degrees. This made them noisy even when they weren't moving and most likely caused some damage. Towards the end of the project while trying to replace a motor with a stripped thread this was noticed and corrected.

The motor controlling the angle of the end effector and the motor responsible for extending the top half of the arm were both faulty, the motor controlling the top extension seemed to completely break towards the end of the project, and the motor controlling the angle of the end effector had a stripped thread meaning it was loose and it's zero point shifted after only the slightest knock. Several hours over multiple days were spent trying to fix the arm by replacing the motors with the support of technical staff and the replacement end effector motor worked near perfectly, unfortunately the replacement top extension motor used a completely different PWM scale and no other alternatives could be found on short notice. In the end it was easier to simply replace the arm as another one was available which only had a faulty Arduino and seeing as this project didn't need the Arduino it worked perfectly.

Another issue was that the pen had too firm a tip, it wouldn't flex enough which caused friction and made the arm struggle to move. So it was replaced with a brush pen, these pen's were almost too flexible and sometimes struggled to leave a mark on the page but caused almost no friction.

9 Accuracy & Repeatability

While the automated control didn't work the manual control and preprogramed state movements worked fairly accurately and consistently.

The manual control showed that the motors had a dead band of around 4 degrees and wouldn't move unless the signal sent to them had changed by that much or more. This probably led to a small amount of inaccuracy, but only seemed to cause problems for the manual control as the preprogramed movements almost always moved by larger amounts. With additional time this probably could have been compensated for and with unlimited time and a proper budget more accurate motors and a better arm design would most likely have been used.

Testing the preprogrammed movements showed that the arm had an inaccuracy on the horizontal axis of roughly 1cm most likely caused by the rotational motor having a small amount of give resulting in it being able to move when at rest. It is suspected that all of the motors have a similar problem but that because those motors move the arm vertically the arm always falls back to the same place. This again most likely could have been accounted for with additional time, either by replacing the arm with a better one or by adding some form of control system to correct for error.

Overall, when using preprogramed coordinates the end effector was always within a reasonable margin of error and it was always clear which state it was in. These tests can be seen in the videos linked in the references [8,9,10].

The Q Learning didn't work until shortly before the deadline and it was too late to produce any meaningful data from it. Additionally, even if the Q Learning did work early enough for testing it's unlikely it could have been tested on hardware as in its current form the project requires 41622 logic registers while the FPGA being used only has 18480 meaning it requires 225% of the available registers.

10 Conclusion

The Isolation circuit worked perfectly, but it wasn't originally intended to be part of the project and getting it working to be able to start properly working on the actual project took far too long.

The initial HDL for controlling the servos also worked perfectly once the issue of using the wrong duty cycle was resolved. This combined with the near perfect manual control modules and the preprogramed states mean that the arm can be controlled by the FPGA and should qualify for the first stated deliverable.

The robotic arm and its faults created many unnecessary additional tasks that consumed far too much time, such as the isolation circuit mentioned above. Similarly, the inverse kinematics were a complete failure and overall added too much unnecessary complexity to the project by adding multiple additional required skills and areas of research that were completely different to those required by the rest of the project. If the project were to be repeated, some alternative to this should be found.

The Q Learning mostly worked, there were a couple of missing features such as greedy action selection which would have improved it but as a result of the recent fixes it does work. NEW_Q works near perfectly, typically outputting values with an error of less than ± 0.01. However, ultimately it was irrelevant that it worked as it required far too many logic registers to work on the hardware available and so it can't even be tested properly. Realistically this project required greater support from someone with HDL and FPGA experience than was available for most of it, as with the support of someone more experienced with HDL it would have been easier to spot simple mistakes and may have been possible to optimise the design to a point that it would compile.

Despite taking efforts to minimise risk while writing the project proposal there were too many unexpected risks. Including the arm being faulty, being unable to solve the inverse kinematics and how time-consuming writing and debugging HDL is.

Overall, this project serves as a good proof of concept and reasonable groundwork, but the original aims and deliverables were far too ambitious for the time frame. Had there been no extenuating circumstances and if the arm had encountered fewer issues it is possible all the work could have been completed on time.

Recommendations

If this project or a similar one was to be repeated then the robotic arm should be removed, the most suitable replacement would probably be a remote controlled car or buggy running on a larger maze on the floor. This could be controlled with more traditional methods with the FPGA only sending state numbers to a microcontroller. using this method you wouldn't need to worry about the inverse kinematics and there would be no need to build a base board. Also a remote controlled car/buggy would most likely be easier to repair or replace than the robotic arm. This would mean pretty much the only thing you would need to work on is getting the Q-Learning to work on the FPGA, drastically reducing the amount of work to an amount that could be reasonably done by a single person in the time frame.

Similarly, ideally at least one other person with HDL/FPGA experience would have been available to peer review each module, or even better to work with on two-person coding. Realistically a project that required this many different skills and had so many varied tasks should be done either as a group or over a significantly longer period of time.

An FPGA with far more resources is realistically required unless significant optimisation can be performed as the current design with too few iterations for proper machine learning is already using far more than what is available.

References

[1] Project GitHub repository https://github.com/Edmonds-Git/PROJ300

[2] DEO-CV User manual
https://www2.pcs.usp.br/~labdig/material/DE0_CV_User_Manual.pdf

[3] Isolator Datasheet https://www.nve.com/Downloads/il600.pdf

[4] isolator LED Test: https://youtu.be/WXU1UwsgJGA

[5] isolator PWM Test: https://youtu.be/epZFnG3XhxA

[6] 7 Segment HDL source https://github.com/arnaudeveloper/Display_7seg

[7] Fixed Point Numbers in Verilog, Will Green https://projectf.io/posts/fixed-point-numbers-in-verilog/

[8] Manual Pre-Set Values Test: https://youtu.be/xRO3ya57xBg

[9] Manual Button Control Test: https://youtu.be/PJylTEoCMLQ

[10] Error Measurement: https://youtu.be/OjvJlNO9JCc