Controlling a Robotic Arm using a DE1-SoC FPGA Board

Jack Gladowsky
Department of Computer Engineering
Northeastern University
Gladowsky.j@northeastern.edu

***Abstract* - This paper presents a way to control a 5 degree of freedom robotic arm using an FPGA development board. The arm is made up of five servo motors that are controlled using pulse-width modulation (PWM). We built a circuit in Quartus Prime that let the user set the robot arms initial and final positions using switches and buttons. The user then flips the last switch which tells the robot to go back and forth between the two positions until told to stop. The goal of this paper is to program a robotic arm that can sit next to an assembly line and pick up and place down different components automatically. In this paper, we simulated the robotic arm picking up and placing down a plastic water bottle.**

Keywords—Pulse-width Modulation, FPGA, Robot-Arm, Quartus

I. Introduction

As the world becomes more and more automated, robotic arms are being found in more and more places. Robot arms are controlled by different types of motors, with some examples being stepper motors and a servo motors. The robotic arm that was used in this paper was controlled by servo motors. We controlled the servos using PWM. PWM is a technique that changes the period of an electronic pulse, and the modulation part is switching the pulse state on and off. Doing this means that's the motor is being turned on and off rapidly and because the servo can't handle the quick changes, the motor results in being turned on for a certain length of time based on how quick the pulse state changes.

In this paper, we designed the controller for the robotic arm using Quartus Schematic 18.1 and then uploaded it onto a DE1-SoC FPGA development board. The DE1-SoC has a FPGA processor and an ARM processor, which means we can do both FPGA and embedded development using Linux all on the same board. The Quartus software lets us program the FPGA processor on the board using different logic blocks.

The arm we used during this project was a 5 degree of freedom arm, that was built by a previous class. We were provided with a simple circuit that when connected to the robot arm, would make the arm stand up at 90 degrees. We used this circuit as the basis for controlling and loading the positions of the arm. The user was able to change which part of the arm they were controlling by flipping a switch, and they could move that position using two buttons. When the user had the arm at the correct position to pick up a bottle, the user then flipped up another switch which saves the arms position. The user then moves the arm to where it should drop the bottle, and the user flips another switch to save that position. Finally, the user flips the last switch which makes the robot go back and forth between the two saved positions, repeatedly picking up and placing down water bottles.

II. Algorithm Implementation

A. Pulse-Width Modulation

PWM is the basis of the entire project. To move the arm, we must first understand how PWM works. The main way to control PWM is by controlling the duty cycle. A duty cycle is the ratio of how often the pulse state changes, and the period of the pulse. If the state changes many times within the period, this results in the motor moving in a certain direction. If the duty cycle is high, the robot should move clockwise, and vice versa when it is low. To implement this using Quartus, we have a modulus counter that goes up to 3,500. The 25-bit output from this counter goes into a comparator that gives out a signal when the counter is 1. Since the clock on the board is a 50 MHz clock, this means that a one will be output every .00007 seconds. The output of the comparator then feeds into another modulus counter clock. This counter goes up to 75,000, which is the value that we found makes the robot arm stand straight up. There is another counter that counts to 1 million repeatedly. These two counters feed into a comparator which outputs when the value of the million counter is less than the 75,000 counter. Fig. 1 shows a flow chart of the entire PWM process implemented using Quartus

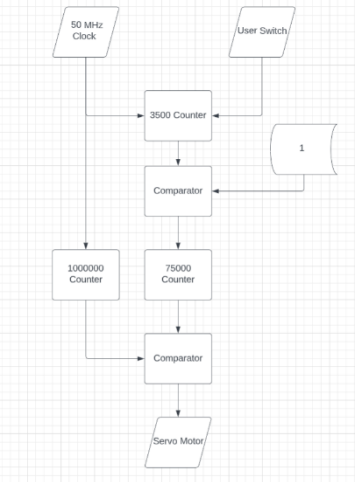


Figure 1. PWM Control in Quartus

B. *Robot Arm Input*

Once the robot successfully stood straight up when the board was turned on and programmed with the circuit, we had to figure out how to control each servo on its own to set the initial and final positions of the arm. The implementation that we decided on was to have only two buttons that would control the direction of each arm. We then had five different switches that corresponded to each of the servo motors. When a servos switch was on, the user was able to move the arm only at the specified servos. We implemented the previous logic by having a D-Flip-Flop for each button. The clock input for the D-Flip-Flop was a comparator that had a counter input that only counted from 0 to 1. This counter was only enabled when a specific servos switch was flipped to on. The comparator then sent out a signal whenever the counter was at 1. This circuit was then replicated five times, one for each servo. When a specific servos switch was flipped on, the counter would enable which means that the user could then move that servo using the switches. Fig. 2 shows the circuit that is used for each servo.

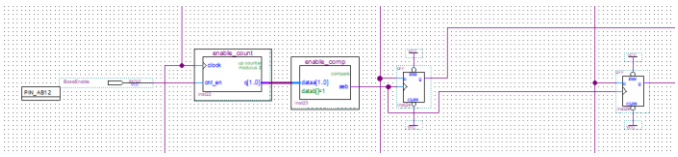
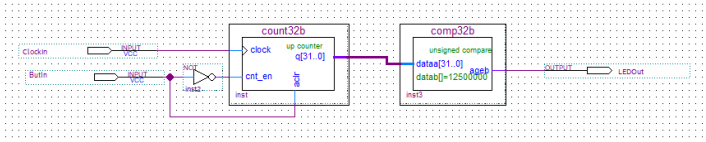


Figure 2. Servo Control Circuit

C. *Debounce-Pulse Module*

Now that we were able to handle the inputs from switches and buttons, we had to implement this into the PWM controller from figure 1. We did this by adding a enable to the two counters in the PWM controller. The outputs from the 2 flip-flops fed into a logic block that represented a debounce-p module. This module would read the input and put out a pulse signal because buttons are too unreliable to send a steady signal. Fig. 3 shows the debounce p circuit in Quartus. Figure 3. Debounce-P Circuit

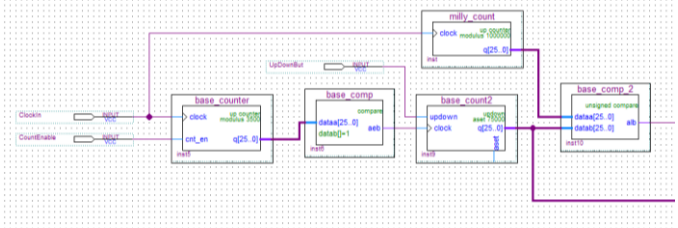
Figure 3. Debounce-P Circuit



D. *Servo Control Module with User Input*

Each pulse module goes into an XOR gate which will only output when one of the buttons are pressed. The XOR gate then feeds into the count enable in the PWM controller. Now when a servos switch is up, the user can move that servo using the two buttons, and the movement will be smooth due to the debounce p. Fig. 4 is the updated control module that now has a count enable.

Figure 4. User Input Servo Control Circuit



E. *Storing Arm Positions*

Now that the user can move the robot arm along its 5 degrees of freedom, we had to figure out a way to save and load the initial and final positions. We came up with a 26-bit counter that has an asynchronous load and count enable input. These two inputs were connected to the initial and final position save switches. When one of the switches was flipped on, the current position of the arm would be saved into that counter. The enable count input was connected to a ground so that way the counter would never count. We had two counters for every servo motor, one for the initial and one for the final positions. We got the data to be stored from the PWM controller. We saved the output of the 75000 counters into the saved position counters. Fig. 5 is a representation of how we stored data and loaded it into the arm.

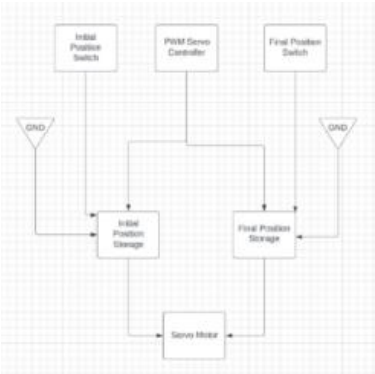


Figure 5. Position Store Counter

F. *Position Control Multiplexer*

Once the two positions were successfully stored, we had to implement a way to load the positions to the robot automatically. We finally found a way that was simple to implement, and easily changeable. To come up with the solution, we had to think about each servo individually. More specifically, what should every servo position be while the robot is moving between positions. Using that question, we figured that the transition between positions could be done in 8 steps. First the arm starts straight up. Second it turns to face the bottle. Third it lowers the arm to the bottle. Fourth it closes the grip around the bottle. Fifth it raises the arm up a bit. Sixth it turns the arm to the position it wants to drop the bottle at. Seventh it lowers the arm. Eighth it releases the grip on the bottle, setting it down on the table. Once we had these eight steps, we were able to figure out what each servos position should be at every step in the algorithm. We used an 8 to 1 multiplexer block to accomplish this in Quartus. Fig. 6 is a picture of one of the five multiplexers used. The input at the bottom is used for the buffer counter which is explained in the next paragraph.

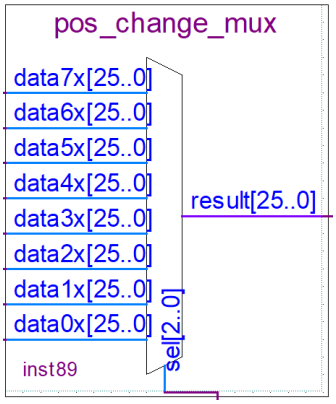


Figure 6. Position Load MUX

G. *Buffer Counter*

The way we selected which position was currently being used was by setting up a buffer counter. A buffer counter is two counters and a comparator. The first counter counts to 45 million repeatedly. The output of that is fed into a comparator which only outputs a signal when the counter is at 1. The comparator will output a 1 every 0.9 seconds. This output then goes into a counter that goes from 0 to 7 repeatedly. This buffer counter will count from 0 to 7 in 7.2 seconds which means that a cycle of the robot arm to pick up and place down a bottle takes 7.2 seconds. A diagram of the buffer counter is shown in Fig. 7.

H. *Final Servo Controller*

We now have a way to store and load the data in the proper order. To connect everything together and make it run smoothly, we took the output of the mux and the output of the PWM controller into a two to one mux. This is connected to the 9th switch which is the start switch. While this switch is down, the user can move the arm freely using the controls. When the switch is up, the robot arm will move between the initial and final positions until the switch is flipped down again. The output of the MUX goes into a comparator that checks to see when this number is lower than a million counter. The output of the comparator then goes to the servo motor. Fig. 8 shows the implementation of the final controller in Quartus.

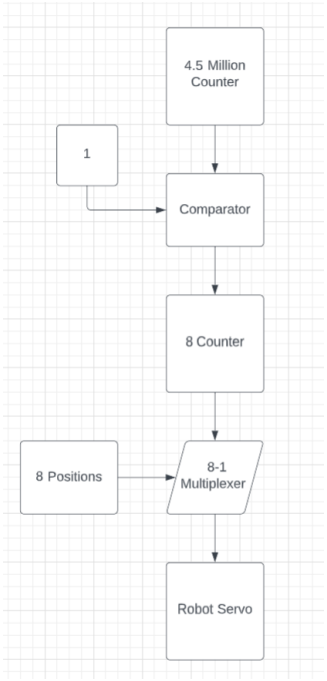


Figure 7. Buffer Counter Chart

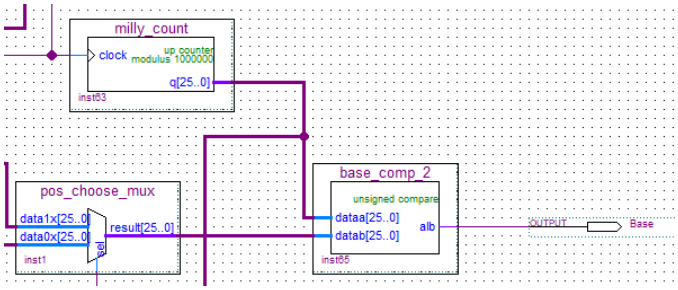


Figure 8. Movement Controller Circuit

III. Results

We conducted tests of the program on a robot arm with 5 degrees of freedom. The arm consisted of 5 different servos, and we referred to each arm as a part of a human arm. Figure 9 refers to the robotic arm and labels each servo with the name used in red.

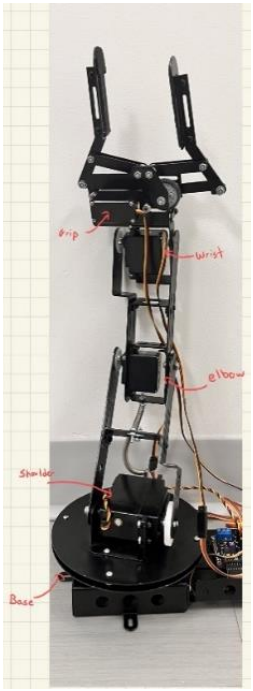


Figure 9. Robot Arm with Servos Labelled

A. *User Interaction*

The user turns on the robot using a switch on its base, and then they can control the robot using the switches and buttons on the DE1-SoC. Using the implementation in Fig. 3, the user can manually control the robot using the switches and buttons. The first five switches are connected to a servo on the robot starting from the base on the right and going to the grip fifth from the right. The user can then use the first two buttons to turn the servo. The user would move the robot to the initial position, then flip the 7th switch which saves that position. Then they would move the robot to the final position and flip the 8th switch. Once the two positions are saved, the user can flip switch seven and the robot arm will automatically go between the two positions, picking up and placing down the bottle until flip 9 is switches off.

B. *Algorithim Limitations*

Throughout the design process of the circuit, we ran into a few limitations and problems. The first big limitation to our implementation is by using a multiplexer to control the movement of the arm between positions. When testing first started for this implementation, we found that the arm would go back to straight up 2 times during the process. The first time is when the arm grips the bottle and then raises straight up so it can rotate towards the second position. The second time is when the arm releases the bottle and then raises straight up to pick up another bottle. The problem with the arm raising straight up is that when it lowers the arm, the arm basically goes into a freefall and then stops at the position it's supposed to be at. Since the arm is moving so quickly, it over rotates past its stopping point and then snaps back up to the stopping point. When it over rotates, the arm hits the table very hard which will either knock the bottle out of the way or knock the bottle out of the arm's grip. To fix this, we figured that we could have a sort of buffer position in between the arm going from straight up to straight down. The arm would stop about 2/3 the way to the stopping position, and then it would proceed to the stopping position. Fig. 10 shows a diagram of the positions and values that make up the multiplexer's inputs.

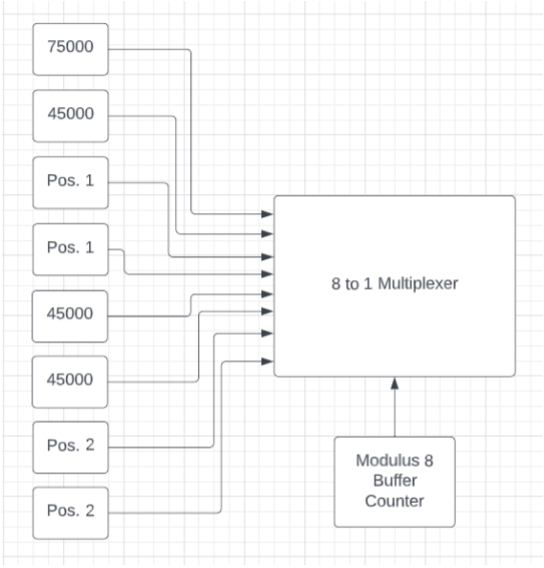


Figure 10. Position Input Diagram

C. *Constant Blocks*

We used constant blocks to create the buffer positions. These are blocks that just output a certain number in binary constantly. Fig. 11 shows a constant block with a value of fifty thousand, and the output is 26-bits wide.

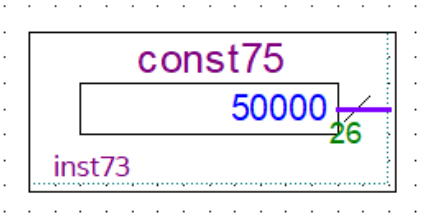


Figure 11. Constant Block in Quartus

D. *Control Switching Logic*

Another problem that we ran into was how to control the robot arm manually, and then make it move automatically. Our solution to this was to use a 2 to 1 multiplexer. We took in the output from the PWM motor controller and the output of the 8 to 1 multiplexer. When switch 9 is off, the multiplexer outputs the first input, and when it's flipped on it outputs the second input. As switch 9 is hooked up to the count enable for the buffer counter also, these two things combined tell the robot arm to move between the two positions. Fig. 12 shows the multiplexer that is used to change between user and automatically controlled.

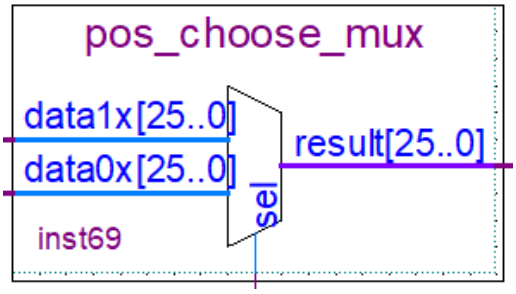


Figure 12. Position Chooser MUX

E. Algorithm Improvements

While implementing the fix for the arm slapping down on the table, we found that using constant blocks to set positions for the arm was very intuitive and improved the ability to understand what the circuit is doing. Because we knew the dimensions of the water bottle that would be on the "assembly line", we were able to set a lot of the servos to constant positions during the process. Along with the constants used in the fix, we also added constants for the elbow and wrist throughout the process. We did this because during testing we found that if the wrist and elbow positions were moved, the bottom of the water bottle would not always be parallel to the top of the table. If we kept the arm straight and just bended it at the shoulder, the bottle would stay parallel meaning a better chance for the arm to place the bottle down without it tipping over. The only positions that were being loaded from the saved positions were the initial and final base and arm height positions.

Another improvement to the program was outputting the buffer counter to a seven-segment display. This helped with debugging the code during testing as we could see which position the robot arm was currently at. By seeing what position, the arm was at, we could tell when a certain position had to be adjusted. We were able to change the values of the constant blocks to adjust the positions of the arm at each servo.

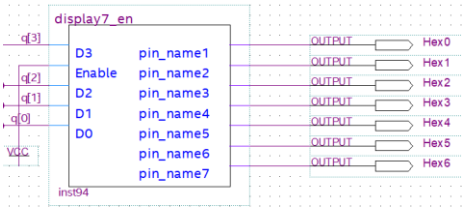


Figure 13. Seven-Segment Decoder Block

We were able to use a seven-segment decoder that we had built in a previous lab to display the output from the counter. Fig. 13 shows the seven-segment decoder block. The outputs of the multiplexer correspond to a certain segment of the display.

IV. Conclusion

This paper presented our implementation of a controller for a robotic arm. We implemented our circuit using the FPGA chip in the DE1-SoC board. We used Quartus Schematic 18.1 to design our circuit, and then used the same program to upload it to the board. We can simplify the circuit down to a few main portions. First the servo controller which is shown in Fig. 1. Then how the user can move the arm in Fig. 2. Then we implemented a way to store and load different positions into the arm which is shown in Fig. 6 and Fig. 10.

Using our implementation, we were successfully able to move our robot arm using the buttons and switches on the board. We were also able to store and load the initial and final positions that the robot arm would be moving to when picking up and placing down the bottle.

Acknowledgements

We would like to thank Professor Marpaung for helping us learn all the material to complete this paper. We also want to thank the TAs for the embedded design class for helping answer questions that We had during the paper.

References

[1] Prof. Julius Marpaung, "*Quartus Schematic Introduction*", Northeastern University, Spring 2021.

[2] Terasic, "DE1 – SoC User Manual", January 28, 2019.

[3] I. E. E. E. Journal, "Conference Letter Template."