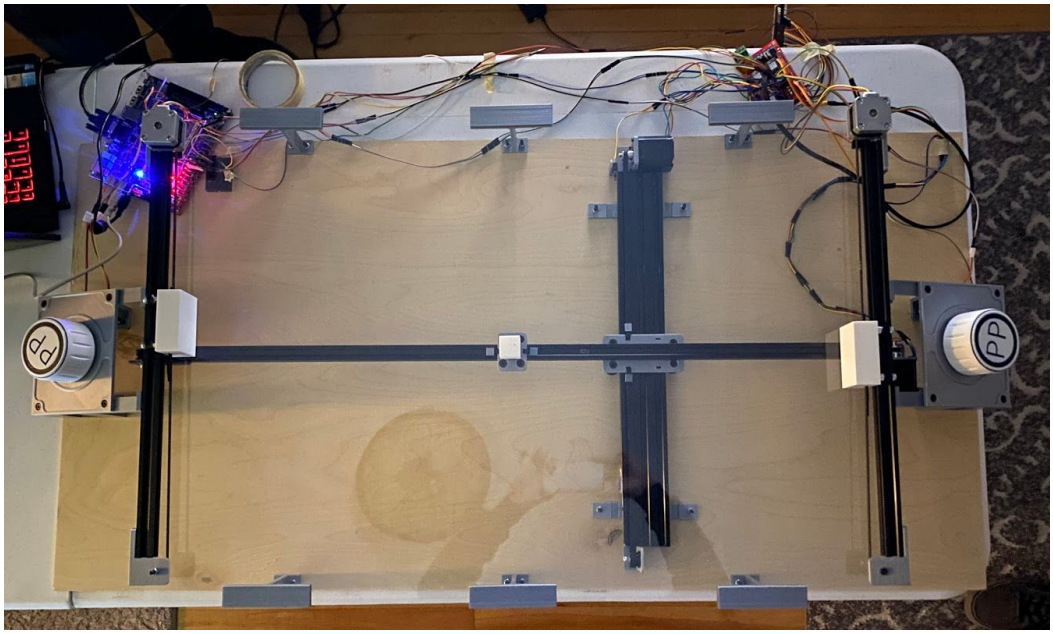
Report for

ECE 3710

Final Project Report



TA:

Kristopher Wolff

Prepared by

Ian Lavin, Colin Pollard, McKay Mower, Luke Majors

Date: December 14, 2020

Abstract

This report describes an electromagnetic Physical Pong game that was built as the final project from group four in ECE 3710. It encompasses several stages of development. If followed correctly, it can be replicated and built from the ground up. The purpose of building this project was to gain experience in different ways: building a CPU and creating instructions to design something out of them, working on a project that simulates a real life job with a hard deadline, and practice verbal presentation skills to present to those who are interested in learning about it or replicating it. The design of Physical Pong included a mechanical board with moving parts that acted as a visual representation of the game. This representation was controlled by game logic that was designed from the ground up. The game logic was then translated from machine instructions that were designed throughout the semester into binary which was stored in memory. The designed CPU was then able to compute the correct logic and transmit data to the arduino. After this overall design section, things that were learned throughout the process of creating this project will be discussed in depth which will involve any future changes that may be made, as well as hopefully giving guidance to those who may decide to replicate it, and finally followed up by a conclusion.

Introduction

The goal of this project was to construct a fully functional physical pong game. The idea of the project was to take the classic pong video game and convert it to a mechanical game. It makes use of an Arduino, an FPGA board, 3D printed parts, and core game logic. First, a CPU was designed to run the game. This CPU is based off of the CR-16 architecture and uses opcodes that are shown in Appendix A. To control the game, rotary encoders are used as inputs to the CPU, and UART serial communication is used to send serialized messages to an Arduino controlling stepper motors. The mechanical design of the board consists of a magnetic ball piece that slides on a glass panel and two player paddles. The structure is held together by other 3D-printed parts that make it lightweight and stable. Lastly, the game was programmed in assembly, which was then processed by the assembler to translate every instruction into binary. These instructions were then stored into memory and ran on the CPU. The details of these different modules are described in the following sections.

CPU Design

The central processing unit design for this project was modeled after the CR16 architecture — a 16-bit reduced instruction set computing architecture. The CPU was designed and programmed using Verilog code, and uploaded onto a Cyclone V field-programmable gate array. The CPU was designed incrementally by combining individual modules into one overall design. The main components of the CPU are described in the following paragraphs.

The arithmetic-logic unit was one of the first modules designed in this project. It takes in two 16-bit inputs, A and B, and computes a 16-bit output, C, based on the opcode. The ALU supports the full list of opcodes shown in Appendix A. The ALU module also sets the five-bit flags module that is used by other parts of the CPU. Next, the 16-bit regfile was implemented. This consists of 16 registers that are used to store values in the CPU. The regfile connects into the inputs of the ALU. The next module designed was a memory access interface. This module makes use of the built in block ram on the FPGA. Specifically, the module was dual BRAM which allows for two separate read and write ports. The memory access module is also used to store instructions that run on the CPU. The memory is initialized by reading a text file containing newline separated binary instructions and storing them into memory.

The next step was to design a program counter. The program counter module was used to keep track of the current instruction in memory. After each instruction completed, the PC either incremented to the next address in memory, or in the case of a branch instruction moved to the requested jump address. The output of the program counter is tied directly to the address input of the memory module. The instruction that is read from memory is passed directly into an instruction register before it is passed on to the instruction decoder module. The instruction decoder was designed to split a 16-bit instruction into an 8-bit opcode, a 4-bit source register value, and a 4-bit destination register value for R-type instructions, or into a 4-bit opcode, 4-bit destination register value and an 8-bit immediate value for I-type instructions.

The next step was to design a finite state machine to control the processor. The CPU FSM is one of the most important components in the design. The finite state machine sets all of the control wires used throughout the CPU. This helps control the program counter, reading from and writing to memory and the regfile, and selecting which values should be passed

through various multiplexors. The FSM is what allows all of the individual modules in the CPU to function as one.

The next step was to implement new modules specific to the needs of this project. This included a module for UART serial communication, a module for reading rotary encoder values, and two modules for a VGA display. These modules were designed and tested individually, then they were integrated into the CPU. These modules are described in further detail below. The overall RTL schematic for the CPU design is shown in Appendix B.

IO Interfacing

UART Communication

To allow the CPU to be able to communicate with external devices a communication method had to be selected and implemented. For this project the common protocol known as universal asynchronous receiver transmitter or UART was chosen. This communication method is asynchronous meaning that it allows for data to be transmitted independent of the clock cycle of the device. It works through a few different steps. For this system since data is only required to travel in one direction from the cpu to the arduino the primary focus was on the design of the transmitter within verilog. As for the receiver the arduino's built in serial receiving port was utilized to allow for simple one way communication.

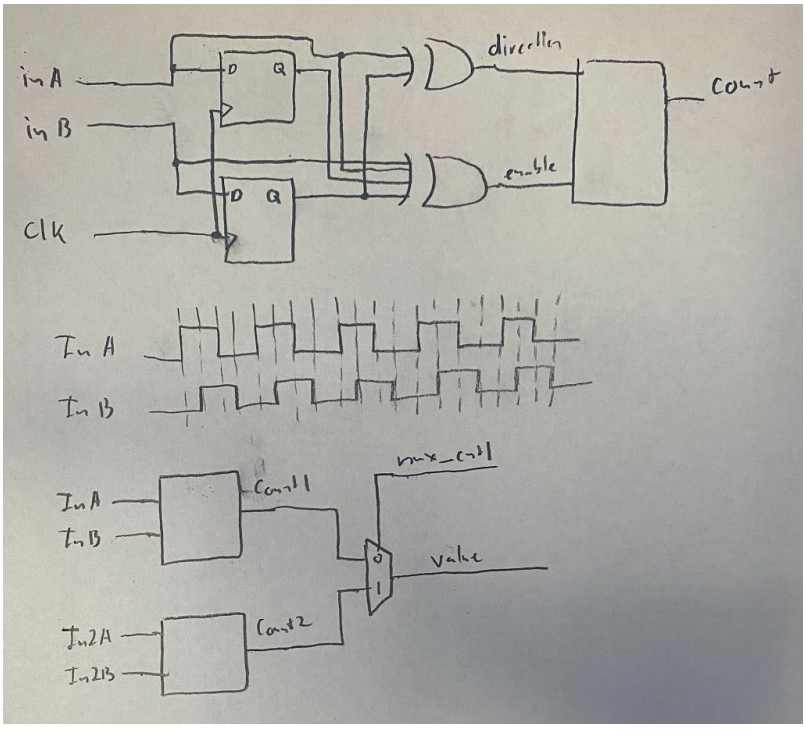
When designing the transmitting module it had to be considered how data was set up to be received on the arduino. The default mode of serial communication on arduino consists of one start bit, followed by eight bits of data to form one byte, followed by one stop bit and no parity. Additionally both the receiver and the transmitter have to agree on a bits per second rate or baud rate. For this system it was decided to use the standard baud rate of 115200. This baud rate was recreated in the verilog module by dividing the input clock signal by the baud rate to calculate the number of clock cycles each bit should last. In addition to the clock input the verilog module also has an eight bit data input and one bit activation signal. The clock is also used to constantly sample the data wire to check for new data. The one bit serial output of the module is driven high to one by default. When the activation signal is flipped to high the module begins going through its transmitting procedure. First the serial line is driven to low for one bit length. At the same time whatever value is on the incoming data wire is stored into a temporary eight bit register. After the start bit has completed the module goes through the data loaded into the temporary register and transmits each one by one for the agreed upon bit length. Once all eight bits have been sent the module sends one final stop bit by driving the output back to high for at least one bit length in order to prepare for future transmissions.

In order to use the transmitter with the pong game, a transmit instruction was integrated into the main CPU data path. The transmit instruction reads data from a register and transmits that data to an Arduino using the UART serial protocol described above. The Arduino accepts incoming serial data at a much slower rate than the CPU on the FPGA is capable of sending. To support this, the transmitter module in the CPU has to wait a certain number of clock cycles, which was calculated by dividing the CPU clock speed by the desired baud rate. At first there were issues in the CPU where another transmit instruction would begin execution before the previous transmission was completed. This resulted in lost data. To account for this, a control signal was added as an input to the FSM that was high if data was being actively transmitted. If

a new transmit instruction is received while a transmission is still in progress, the CPU FSM enters a sort of NOP state where the program counter does not update and execution pauses until the current transmission is completed.

Rotary Encoders

Another important aspect of the hardware interfacing for this project involved the use of rotary encoders to receive data from the players about how they wanted to move their paddle. These encoders had two primary functions. Their first function was to determine which way they were currently being rotated by the player. They achieved this by having two different wires sending 90 degree out of phase signals. The difference in phase between these two signals was read by a verilog module and used to determine the position change. This verilog module will be elaborated upon further in the next paragraph. The other function the rotary encoders served was as a button. When connected to power the encoder button would send a high signal by default. If the encoder was pressed down upon however the signal would be driven low to signal a button press. A basic module was developed that took in both rotary encoder button signals and ANDed them together. This module would only produce a one if both buttons were pressed at the same time and was used within the game to allow both players to signify they were ready to play.



With the encoders now properly sending signals to the FPGA the next step was to develop a module capable of decoding and utilizing those signals. This was achieved by utilizing a few d flip flop modules connected to two xor gates. The flip flop modules were connected to the CPU clock which was faster than the clock of the encoder signals. This was important because the clock needed to be faster than the incoming signals in order to “oversample”.

These two flip flops were also connected to input wires A and B coming from the encoder and each generated a signal that was delayed by one clock cycle from the incoming signal. Finally the undelayed A signal and the delayed B signal were connected to the first xor gate while both the delayed and undelayed A and B signals were attached to the second xor gate. The first xor gate then produced output count direction that was one if the encoders were turned positive and zero if they turned negative. The second xor gate meanwhile was positive if the two signal wires were offset and changing and zero if they were equivalent. This allowed it to serve as an enable wire and detect if the encoder was moving or not. Finally these two inputs were fed into a register that would activate with the enable wire and increase if the direction was positive and decrease if it was zero. This allowed the register to keep a running count of the change in position of the encoder. For the final step two of these modules were instantiated so each encoder would have its own module. The two count register outputs were then fed into a mux that allowed selection between which encoder value was desired. This mux was controlled by the FSM which also controlled a mux within the ALU datapath allowing external encoder data to be read into a register. When the FSM detected a custom designed encoder read instruction it would select the proper encoder value to pull from, enable external data to be fed into the datapath, and select and enable an appropriate register to write into. Finally after all of this was done the FSM would send a reset signal to the encoder that was read from that reset the encoder count back to zero. This was important because it allowed the count value within the encoder module to only record the change in position from the last time it was sampled as opposed to holding the absolute position value. The code was designed in such a way that it only needed the encoder to record the change in their position so that it could be used to compute the overall paddle position.

When implementing the encoders into the CPU data path, some errors were encountered. The encoder module reads in a count from the encoder as an 8-bit value. Originally, this value was not sign extended before being stored into a 16-bit register. This caused undefined behavior when the encoder values were used in a larger program. The error was very subtle because the lower eight bits in the register always had the correct value, but when used with compare and branch instructions, the undefined upper eight bits caused unexpected behavior. Another minor error with the encoders had to do with the wiring. Sometimes noise from high current motor wires causes improper encoder values to be read by the FPGA. This was resolved by separating the wires for the rotary encoders from the motor wires.

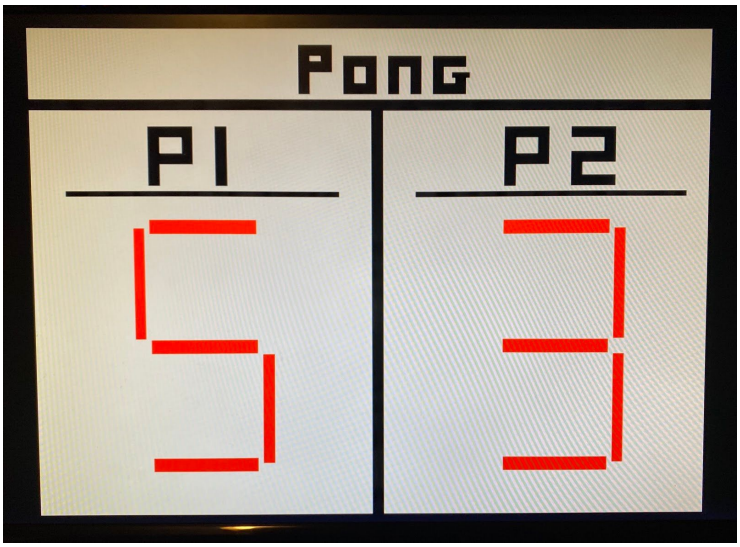
VGA

Another component of the design was a scoreboard displayed on a VGA monitor. This involved adding two modules to the CPU data path. One module controls the VGA display. The other module, known as the bit generation module, determines the colors of the pixels that are displayed. In this implementation, there were no glyphs stored in memory. Simple rectangles were drawn by using if statements to check a region defined in an X and Y coordinate grid. Each X and Y coordinate represents a single pixel on the screen which has a 640x480 resolution. To make displaying the score easier, the VGA bit generation module was designed to act as two seven segment displays. The module takes a seven-bit input for each score, and

each bit corresponds to a segment of the score on the display. The code snippet below shows how a single segment is drawn.

```
//Top edge of P1s score (Segment A)
else if (x_pos >= 115 && x_pos < 210 && y_pos >= 187 && y_pos < 200) begin
   if(p1[0])
      r = 8'd255;
   else begin
      r = 8'd210;
      b = 8'd210;
      g = 8'd210;
   end
end
```

In the CPU data path, the scoreboard is implemented by connecting two registers to the VGA bit generation module each through a BCD to seven segment converter. These registers always store the players' scores in the game, and are updated in the game code. This implementation allows the scores to update instantly on the screen after a player scores. The complete score board display is shown in the figure below.



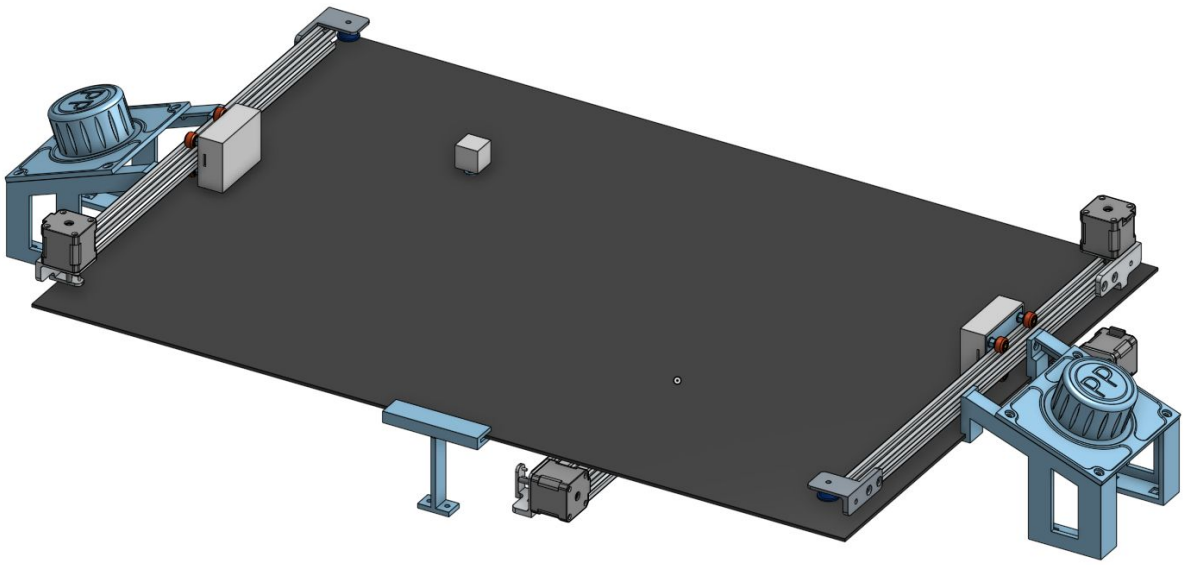
Buttons, Switches and Seven Seg Displays

One final hardware interfacing method that was used were buttons and switches built into the FPGA board. There were a few different functions each of them served. One button on the board was directly connected to the reset input on the FSM of the CPU. By pressing this button the game would be hard reset and clear the players score. Another useful input on the board were the 10 switches located under the seven segment displays. When the game logic was being considered one question was how to control both the game difficulty as well as the initial starting direction of the ball as it was served. Due to the complexity of developing a random number generator, a simple solution was considered of using the switches to allow the players to input their desired speed and starting conditions. The leftmost five switches were used to generate a five bit value for initializing the dx value while the rightmost five switches could be used to initialize the dy value. Before each serve of the game the players could flip the

switches and change the difficulty for that round. After implementing this however it was realized that it was becoming a hassle to have to mentally convert the speed desired into an equivalent 5 bit binary representation. So to make this simpler the seven segment displays on the board were used to display what the current initialization value was set to. Using the previously developed bcd to seven segment display modules the bcd input was hardcoded to the current switch value while the outputs were wired to the left and right center seven segment displays. Thus as the switches were flipped the actual value being input was now shown on the display in hex which made it much simpler to quickly set the desired speed and difficulty level.

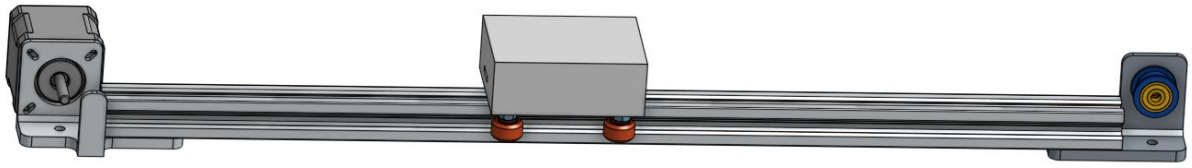
Mechanical Design

The physical pong board was designed in its entirety in 3D modeling software. For this project, Onshape was utilized to create all of the subparts, and to model physical parts such as motors, glass panels, rotary encoders, and switches. From these subparts, subassemblies were created that model parts like a paddle gantry, which would then be put together into larger assemblies until the full design was reached. This process flow allowed for minimal clearance and dimension mistakes once manufacturing took place. All pieces were custom designed from scratch, and manufactured on a Prusa i3 Mk3 3D printer from PLA filament. In total, around 1.5 pounds of plastic were used. Shown below are some of the assemblies created.

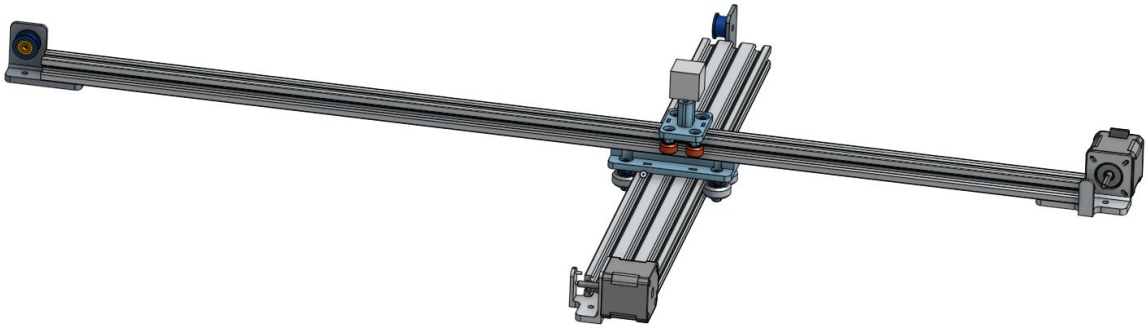


The moving parts are based on Openbuild's v-rail system. These are a collection of aluminum extrusions (similar in shape to 2020 t-slot), with complementary plates and wheels that allow for smooth movement along the edges of the rails. For this application, the wheel spacing patterns were recorded from standard v-rail gantry plates, and used to create custom paddles and ball movement plates. Each paddle assembly consists of a nema-17 stepper motor with limit switch, a v-rail extrusion, a moving paddle, and an idler. From these parts, a gt2 belt is anchored on one end of the paddle, routed around the motor pulley, around the idler, and then

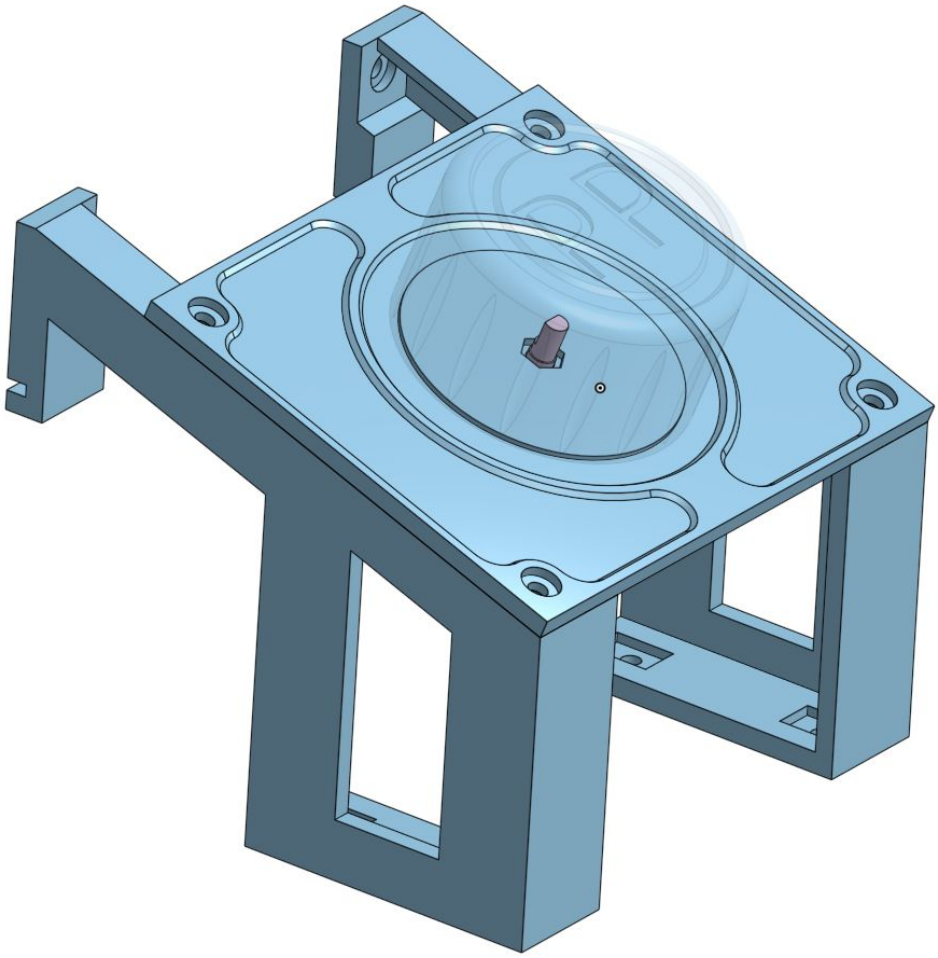
to the other end of the paddle to be anchored. Belt tensioning is handled by a built-in belt clip in the paddle.



Ball movement is handled through a two-axis gantry system. Y movement is handled by a triple-wide v-rail with a wide gantry plate that mounts the X rail. The positioning of the Y movement is such that the center of mass of the assembly is over the plate, hence the offset appearance. The x movement is similar to a paddle assembly, but instead of a visible part moving, the gantry moves two oppositely polarized magnets. The visible ball has an identical magnet pattern flush on the bottom. This allows a piece of 0.125” glass or acrylic to be mounted between the ball movement system and the ball while maintaining movement.



To mount the rotary encoders, a plate was designed that matches the outside thread of the encoder, and has a countersunk nut pocket. The plate mounts to two brackets that screw into the back of each paddle assembly using t-nuts - rectangular nuts that fit into the extrusion and allow sliding mounts. Knobs were designed for each encoder that have a d-shaft extrusion hole to capture the encoder, and embossed edges for better grip.



Finally, the glass sheet is supported by two types of components. As seen above, the encoder brackets have a slot for the glass below the mount point to the paddle assembly. These two points on either side of the board are not enough to prevent the glass from sagging and interfering with the ball adhesion, so additional T-shaped mounts were designed. Six were created in total, and three are mounted on each side of the board. All parts were then mounted to a base board made of wood using M5 bolts.

Software

Assembler

An important piece of this project was the assembler that translates written assembly code into binary machine instructions that are loaded into memory and run on the CPU. The assembler was programmed in Python because it offers a lot of easy to use functions for parsing text files. The assembler simply takes in a text file containing assembly code instructions and outputs another file containing newline separated binary instructions. The assembler supports all of the instructions included in the instruction opcode table in Appendix A. The assembler can also handle comments preceded by the '#' character, blank lines, and jump labels preceded by a period character. To make the game code easier to develop, custom variable names were mapped to certain registers. For example, the x-position of the pong ball in

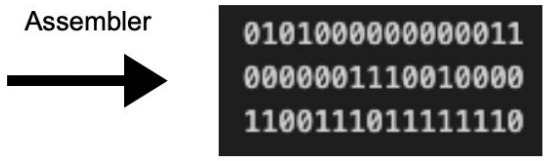
the game is always stored in the R7 register. The assembler was updated so that the name ‘X’ now refers to the R7 register value. An example of a simple program is shown below. This program shows the required format of the assembly code, and the result of the assembler translation.

```
#This is an example assembly program
.Start  #This is a jump label

#I-type instruction: OP $Imm, Rdest
ADDI $3, R0

#R-type Instruction: OP Rsrc, Rdest
SUB R0, R3  #R3 = R3 - R0

#Jump/Branch type instruction: OP Label
JMP Start
```



Game Logic Design

To create the game logic, the first step was researching different variations of pong online to either replicate that code or to get a feel of what the game logic needed to be. What was found online was not really suitable for this project since it was written in html, java or some other IDE. These types of languages involve drawing in a graphics panel of some sort, which was something that differed quite a bit from what the goal was. In the end, it was decided to ditch the idea of implementing something online, and the code for this project was written from scratch.

To start, the first challenge was to think about what kind of logic pong has. There are three main things in the original pong game: a paddle for player 1, a paddle for player 2, and the ball that moves between the paddles during the game. Being a simple game there is not much to it. First, it was decided how the game needed to begin. This involved toying with ideas on who the ball was going to be directed towards, and where the ball will start on the actual beginning of the game and during the reset stage after someone scores. Where the ball will start is an easy answer: just move it back to the middle of the board, which is what was done. To decide who the ball goes to was a different question. One of the reach goals was to implement a random number generator to decide who the ball was going to head towards at the very start of the game. When either player 1 or player 2 scored, the ball would just head to whoever did not score to keep it simple. Unfortunately, this group was not able to create a random number generator in the allotted time but that may be something that could help make this project better.

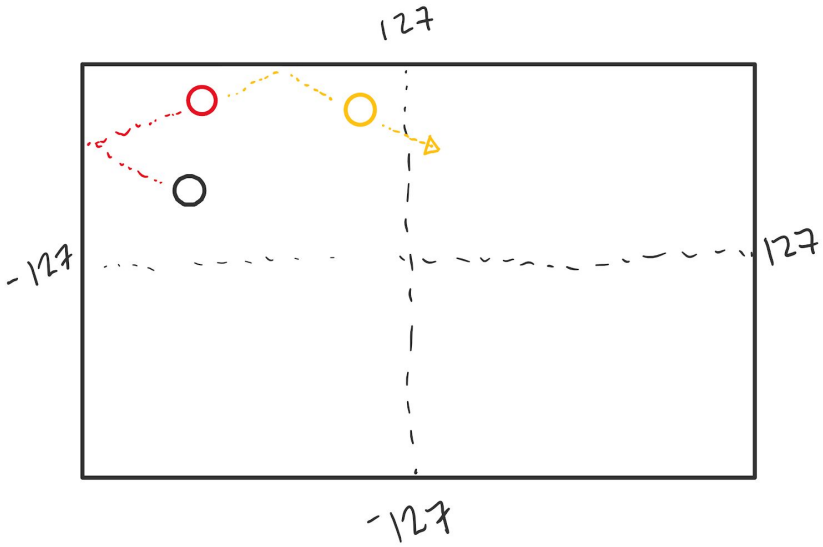
The ball starts in the middle during the start stage of the game, as well as during the reset stage when either player scores. Currently, the ball just heads towards one player or the other since a random number generator was not yet created. To expand upon what was needed

in the reset stage, it was also decided that whoever scored gained a point so during this reset stage the player who scored is also awarded a point. The last thing that was needed during this stage, was to know when to start again. A waiting loop was implemented that took input from both the rotary encoders (they can be pressed down) and once both rotary encoders are pressed, the game goes from the reset stage to a running stage. This means the reset stage only involves moving the ball back to the center, incrementing the score of who scored by one point, and then waiting for the game to start again once both players are ready. This kept the reset stage simple which is both good to change it if needed, and good for the players on the other end.

For the main game loop there were several things to consider: paddle movement, paddle collision with the walls (done in Colin's part), ball movement, and ball detection with both the paddles and the walls. Incrementally adding code to see if it would work on the physical board was crucial during this part of development. The whole game board was represented by an 8 bit by 8 bit game space, so the ball was made to move from -127 to 127 in both the X and the Y directions. As far as the paddles go, they can only move in one direction so they can move from -127 to 127 in only that direction (it is arbitrary but paddles are technically the Y axis).

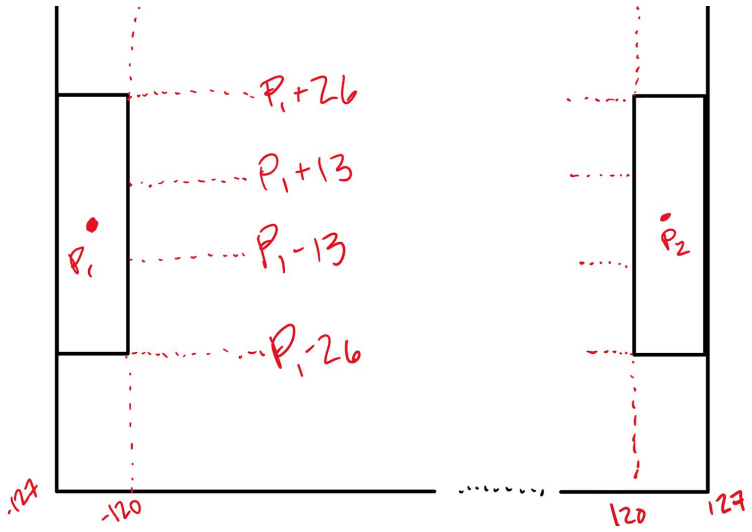
To represent ball movement a variable was stored in a register for the velocity in the x direction called "ballDx" and a variable was stored in a register for the velocity in the y direction called "ballDy". The x and y position of the ball stored in a register as well. Since the board was 2 feet by 3 feet physically, and the gamespace is an 8 bit by 8 bit virtual space, an arbitrary direction was scaled so the ball was able to move constantly throughout the game. In the main loop of the game logic, the velocity of the ball in the x direction was added to the x position, and the same for y. This was done in assembly code, so there were a few manipulations of registers and then a simple ADD instruction to add the velocity to the position. This happened every iteration of the game loop to constantly move the ball in the physical board.

To detect ball and wall collisions was simple. Referring to the diagram below, all that is needed to do is check the x position and the y position of the ball. There are four separate cases to check for: if x is greater than 127, if x is less than -127, if y is greater than 127 or if y is less than -127. (Note: to detect collisions with paddles, these values change slightly, but the logic is technically the same). These checks are done using our CMP/CMPI opcodes that were created, then branch to another instruction if these 'if' statements trigger. If x is greater than 127 or less than -127, simply negate dx (velocity of the ball in the x direction), if y is greater than 127 or less than -127, then negate Dy (velocity of the ball in the y direction). To negate either of these values, they are moved to a temporary register (i.e. MOV dx, R1), the velocity is set to 0 (i.e. XOR dx, dx), and finally the respective value is set by subtracting and storing back into Dx/Dy (i.e. SUB R1, dx). This allows the code to be kept clean and simple, and it does exactly what is required.



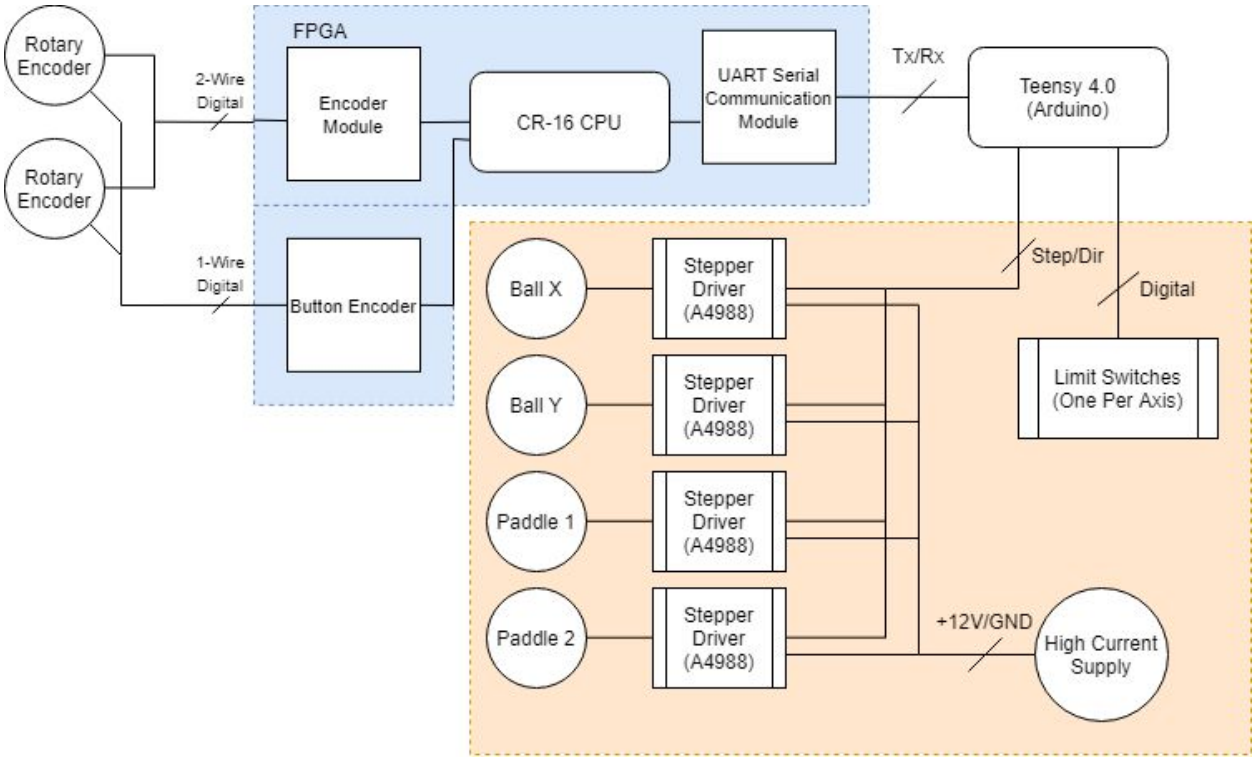
Paddle positions were kept as absolute values. That is, referring to the diagram below (the diagram with the ball omitted to keep the diagram clean), the dot in the middle of each paddle is their exact position. If the board size was 2 feet (24 inches) by 3 feet (36 inches), a simple ratio was created to map the size of it to a virtual space. Doing some math to figure out the amount of offset needed for paddle collision detection, it was determined that the length of the paddle is about 7 units in the virtually mapped space, and the width of the paddle was about 40. From here it was decided to break up paddle detection into three different scenarios depending on where the ball hit the paddle. The first check was whether or not the y value of the ball was greater than p1/p2 position + 26 or less than p1/p2 position -26. If it is outside of the paddle range, then the x value of the ball is checked. Referring to the diagram below, if x is less than or equal to -120, p2 would score, if x is greater than or equal to 120, p1 would score. Now, if the ball's y value is inside the paddle range (i.e. less than p1/p2 + 26 and greater than p1/p2 - 26) the x value is checked. If x is greater than or equal to 120 or less than or equal to -120, then the ball simply bounces off the paddle.

To keep the game interesting, the ball was made to bounce off the paddle in different ways depending on where it hit the paddle. Assuming the ball is inside of the range of the paddle, there are three spots to check where the ball hit: in the middle of the paddle, on the rightmost side of the paddle, or on the leftmost side of the paddle (each of the bounces simply negates the Dx value of the ball the same way as discussed earlier). If the ball hits on the left side, the y velocity of the ball is slowed down, and if the ball hits on the right side, the y velocity is sped up. This allowed for variation depending on where the ball hit. The ball could never be below 0 speed, and could never move faster than 16 units in the x direction and 11 units in the y direction during any iteration of the game loop. Once this was completed, the logic was done and the game was able to be played.



Overall System Integration

This section provides a general overview of the project and how the different components explained above connect. The primary input is read from the two rotary encoders which are read in by the encoder module on the FPGA. The encoder module ties into the CR-16 CPU where the input is handled by the game software stored in memory. Also connected to the CPU is the UART serial communication module that is used to transmit data, specifically ball and paddle positions, to the Teensy Arduino. The Arduino then connects to and controls four stepper drivers. The stepper drivers then move their corresponding stepper motors to the desired positions. The overall block diagram of how the project is connected is shown below.



Testing and Verification

A large portion of time was allocated to the testing and debugging of this project. With the integration of game logic software, Verilog code, and a mechanical system, testing was a bit of a challenge. The testing of this project could be broken into three groups. These include testing of Verilog models and the CPU, testing hardware components and I/O interfacing, and testing the software used to run the pong game. Each category is described in further details below.

First, it was important to test the Verilog code that made up the CPU. As stated above, the CPU was designed incrementally. During this process, each module had to be tested individually before it could be integrated into the overall design. To accomplish this, test benches were written in Verilog and tested in ModelSim. The next step was to test the overall CPU design as each component was added. For this testing, the same same test bench was reused and overwritten to adapt to the updated CPU. When testing individual modules, print statements were useful to easily verify a module's behavior. However, when testing the integrated design, waveforms were used instead of print statements. The waveforms made it possible to view values of internal components of the CPU that were not direct inputs or outputs of the top level module. This was very useful for checking values stored in individual registers, verifying the state of the FSM, and checking any relevant control wires. Towards the end of the design, the CPU was tested by writing very simple programs in assembly. The programs were assembled, stored in the memory initialization file, and ran in the test bench to verify the behavior of the CPU.

Another important category was the testing and debugging of hardware and I/O interfacing. Testing these components involved a lot of small adjustments and observations. For example, testing the UART transmitter involved uploaded simple programs with expected behavior and observing the result on the Arduino serial monitor. When testing the rotary encoders, the most effective method was reading in the encoder values to the FPGA, transmitting them to the Arduino, and reading the values as the encoders were rotated. The hardware debugging also entailed verifying many wire connections. The mechanical system also required a substantial amount of testing. Numerous small adjustments were made until the system was running smoothly.

Next, testing the game software was required to ensure the game performed as expected. An important part of this was testing the assembler that translates the written code into bytes that are stored in memory. This was tested and verified with every instruction used in the game code. As described in the game logic section, code was incrementally tested and implemented in the project. Starting off with the ball movement, we thoroughly tested different velocities of the ball and hand picked speeds that were not jittery (where game logic was updated too fast for the arduino to transmit) as well as speeds that were not too slow (where game logic had too much busy looping each iteration). After ball movement, we moved towards getting the ball to bounce off walls using logic that was discussed in the game logic section. Once we were able to bounce off walls, we used the same logic with slight modifications to get the ball to bounce off the paddles. Incrementally testing the code allowed us to make sure one thing at a time was working which allowed us to slowly build the project from the ground up.

Conclusion and Future Work

The physical pong demonstration was a great opportunity to practice engineering skills in CPU design, hardware interfacing, mechanical design, and assembly coding. A CR-16 CPU implementation required concrete knowledge of single-cycle computer design, and IO interfacing required intuition strong enough to modify the design for new functionality while maintaining function. This was also an exercise in precision and testing, as any bugs in the ALU could be discovered in final programming if testing was not thorough enough. While the final project has a fair amount of extra work to be done before it could be a commercially viable product, it is an impressive and robust demonstration of what is possible in this amount of time.

In future revisions, there are a few key areas to improve upon. The most important of which is closed-loop collision detection. Currently, a position is calculated in the game logic based purely on CPU calculations. This ball position is capable of moving much faster than what the physical motor configuration can support. This creates a situation where the ball is “lagging” behind where the game logic places it. The most obvious symptom of this is when a ball collision happens in the game logic that looks out of place with the current position of the ball and paddle. For example, if a player moves the paddle very quickly at the last second before contact with a paddle, the game logic (controlling the collision detection) will move the paddle instantly while the physical paddle lags behind. The virtual paddle could be beyond the ball, while the physical paddle is directly on the ball, and the collision would still fail in the game logic despite it looking visually correct. This, and many other synchronization issues, can be fixed by calculating collision logic with current physical positions - instead of commanded virtual positions. One way to implement this would be to create a serial receiver in the CPU datapath, which would receive the most current physical positions of the ball and paddles, these values could then be used for collision detection and other game attributes. Another area to improve upon is wiring and mechanical stability. Transportation of the current demonstration requires two people to move, and ideally integration with a more sturdy case like a coffee table would be a more suitable final product. Wiring - particularly on moving parts - could use improvement with cable guides, and guide chains for these moving linkages.

Appendix A

Instruction	Op Code	Comments
ADD	00000101	
ADDI	0101XXXX	
ADDU	00000110	
ADDUI	0110XXXX	
ADDC	00000111	
ADDCI	0111XXXX	
ADDCU	00000100	Custom #1
ADDCUI	00001000	Custom #2
SUB	00001001	
SUBI	1001xxxx	
CMP	00001011	
CMPI	1011xxxx	
CMPU/I	00001100	Custom #3
AND	00000001	
OR	00000010	
XOR	00000011	
NOT	00001111	Custom #4
LSH	10000100	
LSHI	1000000s	s=sign, (5 bit immediate, one of which is in the op code -15 to 15)
RSH	01001111	Custom #5
RSHI	10000101	Custom #6
ALSH	10000111	Custom #7
ARSH	10001000	Custom #8

NOP/WAIT	00000000	
MOV	00001101	Not implemented yet
LOAD	01000000	Not implemented yet
STOR	01000100	Not implemented yet
ENC1	10001100	Custom #9
ENC2	10001101	Custom #10
TRANSMIT	10001111	Custom #11
Bcond	1100xxxx	
LOADSWITCHL	01001010	Technically TBIT
LOADSWITCHR	01001110	Technically TBITI
READSTART	01000001	Technically LPR

Appendix B