Implementing the TRON Lightcycle Game on an FPGA

Brian Burton, Michael Linnebach, Brady Smith, Benjamin Wadsworth

Implementing the TRON Lightcycle Game on an FPGA

Brian Burton, Michael Linnebach, Brady Smith, Benjamin Wadsworth

Abstract

Using an Altera FPGA, we built a two player Tron game. In the game, a player, drawn as a line showing everywhere they have been, tries to avoid the trails produced by both players. To accomplish this, a CPU was built with memory and arithmetic capability, and interfaced with a custom built VGA and PS/2 driver to provide input and output. Software was produced to accomplish this, and the end result accomplishes all of the goals of the project.

I. Introduction

WHEN first deciding on our ECE 3710 final project, we wanted to build a robot that would draw a picture on a piece of paper using Bluetooth for input and VGA to display the state of the machine. We planned to create a SCARA robot using stepper motors, controlled by an Arduino. This plan however didn't last long as we began working as we learned that the project would have been infeasible given the amount of time we had to complete it. Our course then shifted toward creating a fully functional 2 player Tron game that would be able to detect collisions to determine a winner for the game. Then, each time a player won a game they would get a "point" and their total score would be displayed on the seven segment display of the FPGA. We also decided to implement the ability for a player to "jump" a short distance approximately every 8 seconds. This would be used to jump over a line if a player gets trapped. For our fully integrated system, we used our CPU, the VGA output from the FPGA board, and a PS/2 keyboard. This report details the construction of each part of the system, from the individual parts of the CPU, to the software, and the VGA and PS/2 drivers that make the game able to interact with the player.

II. The Arithmetic Logic Unit

The ALU was the first unit we designed. Using the instruction set outlined in the class, our ALU is able to do logical operations, addition and subtraction, and bit shifting, and does so while outputting the various flags required.

A. Implementation

We started our implementation by implementing all of the basic operations, as well as NOT. We decided from the outset that because the operation codes are the same for register and immediate versions of these commands, albeit in different positions, that we would have our ALU take that operation code and a register select line from the CPU control. The ALU would then perform the various operations the same way, switching to the immediate as needed. In this process, the choice was made for the ALU to not care whether a number was signed or unsigned numbers, allowing the programmer the ability to choose what the data is at runtime. However, we realized that some immediates would need to be signed to support some instructions operations. Therefore, we had the ALU perform sign-extension (yielding an effective 7 bit immediate) on operations in which sign could be important, such as add, subtract, and compare. For other operations including the logical operations, the ALU extends with zeros. This gave us enough of the ability we needed when using immediates, and allowed us to load in different values if required.

For easier implementation, we grouped the instructions by function. All the add instructions perform the exact same logic under the hood, as well as all the subtracts and compare. For flags, our design required that every flag possible be set in every instance. We supported 5 flags. For all operations, we supported

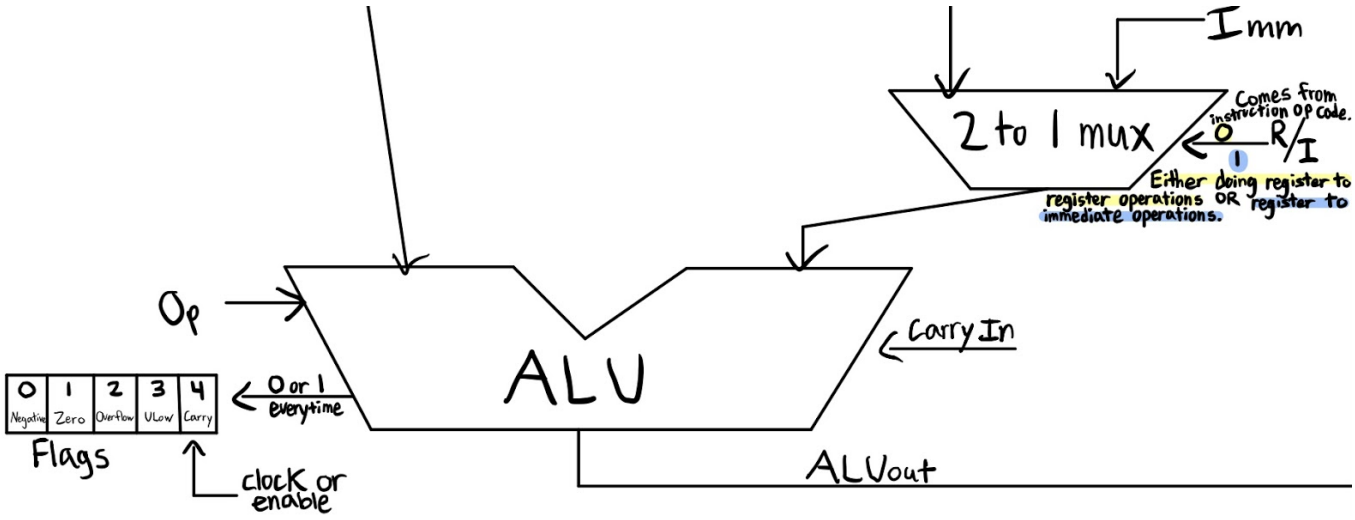

Fig. 1. ALU Block Diagram

Zero, and for Add, Subtract, and Compare, we supported Negative, Overflow, and Carry; we supported Lower (an unsigned compare) for subtraction and compare only. We did our best to support as many flags per operation as possible, allowing for easier programming and the ambivalence to signed-ness. Using conditions, the correct flags can be used when performing jumps. Additionally, although the result and flags would always be generated, they would be able to be controlled whether they should be written later.

Our ALU does contain a few instructions that do not fit the traditional design. Because of constraints, register to register moves, as well as loads from immediate into the upper and lower portions of a register are all passed through the ALU. Shifts also work differently because of their unique encoding, and they were by far the most difficult part of the ALU. Because the shift instructions work differently than the rest of the ALU, the instruction code is passed through the immediate, as well as the register or immediate for shifting by. In the beginning, we believed that Verilog would support twos complement signed shifts, but we realized in testing that this was not the case, therefore we used a sign bit to determine the direction of the shift with a four bit immediate. When we developed the assembler, we determined that we would support both left and right versions of the instructions through this sign bit, even though no change to the ALU was made.

B. Testing

To test our ALU design we tested it both in simulation and on the FPGA board. To test it in simulation we developed test benches which contained test cases for all instructions that examined both random and edge case instructions could be performed by the CPU. Then, for each test case the output of the ALU was examined and determined to be working correctly. Then, for on board testing we followed a similar route in which a module was developed that created specific test cases for the ALU and the flags produced were shown on the LEDs and the result was displayed on the seven segment display. Then, every time a button on the FPGA was pressed (which served as the “clock”) it would move on to the next test case until there were no more test cases to be run and displayed. While doing this, we did determine a bug to the way the Low flag worked, and changed it accordingly. After this we were able to confidently conclude that the ALU was working as intended.

III. The Register File

Another essential and basic part of a CPU are the register files. We decided to create sixteen individual 16-bit registers stored in a regbank. Our registers are essentially just D-Flip-Flops that have an enable signal coming in. This means that each of our sixteen registers have their own enable signal, thus allowing

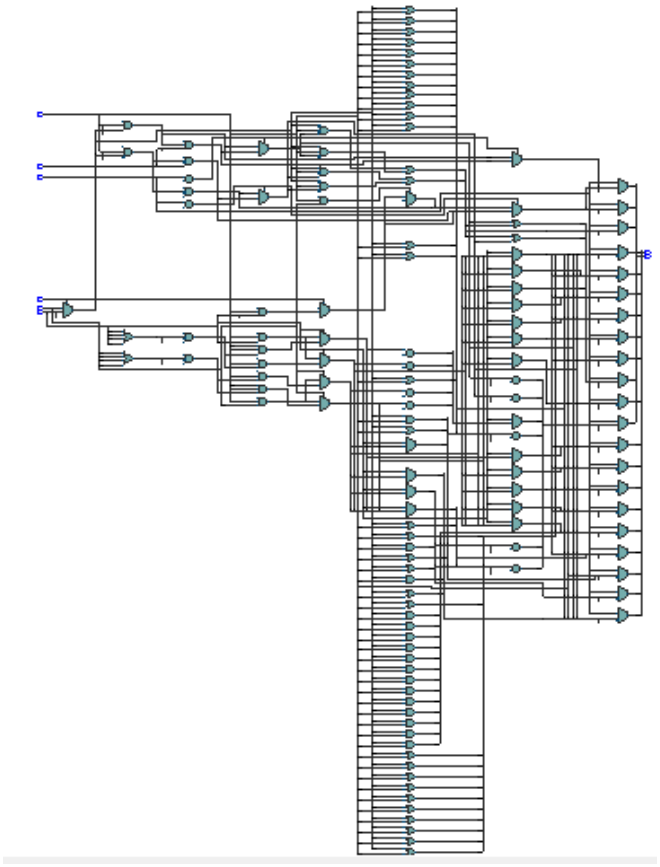

Fig. 2. ALU RTL Schematic

us to control when values are stored into each and every register. The dest and src field in our instruction encodings is four bits, so we designed a four to sixteen bit decoder to help integrate our ALU and Register Files once put together. Furthermore, we made a separate flags 5 bit register to hold the flag values.

A. Implementation

In order to connect our ALU to our Register files, we sent the ALU output to each register through a mux which would enable us to write other values in the future. The registers with an enable signal would then write the value into the register on their clock. In order to send information back to the ALU, we used two sixteen to one multiplexers. These multiplexers determine which register has the source and the destination operand values are for the ALU to operate on. Flags are also still set for the special arithmetic situations and for some instructions such as addu the enable signal for the flags register is not set to true so the flag values produced are not committed. The block diagram shown below served as the design basis for our register and ALU datapath.

B. Testing

For our register testing we once again designed a tester to be displayed on the board as well as a simulation test bench. Both followed a similar format except for this time we knew the ALU was working so we didn't have to test all instructions. Instead one of the test benches was designed such that it performed the Fibonacci sequence making each register be equal to the two before it while another tester made it so that each register was equal to the left shift of the previous register. After all values were stored you could then loop through each register to determine that it was storing values correctly. Once again for the on board testing a similar tester to the ALU was designed such that it performed the next

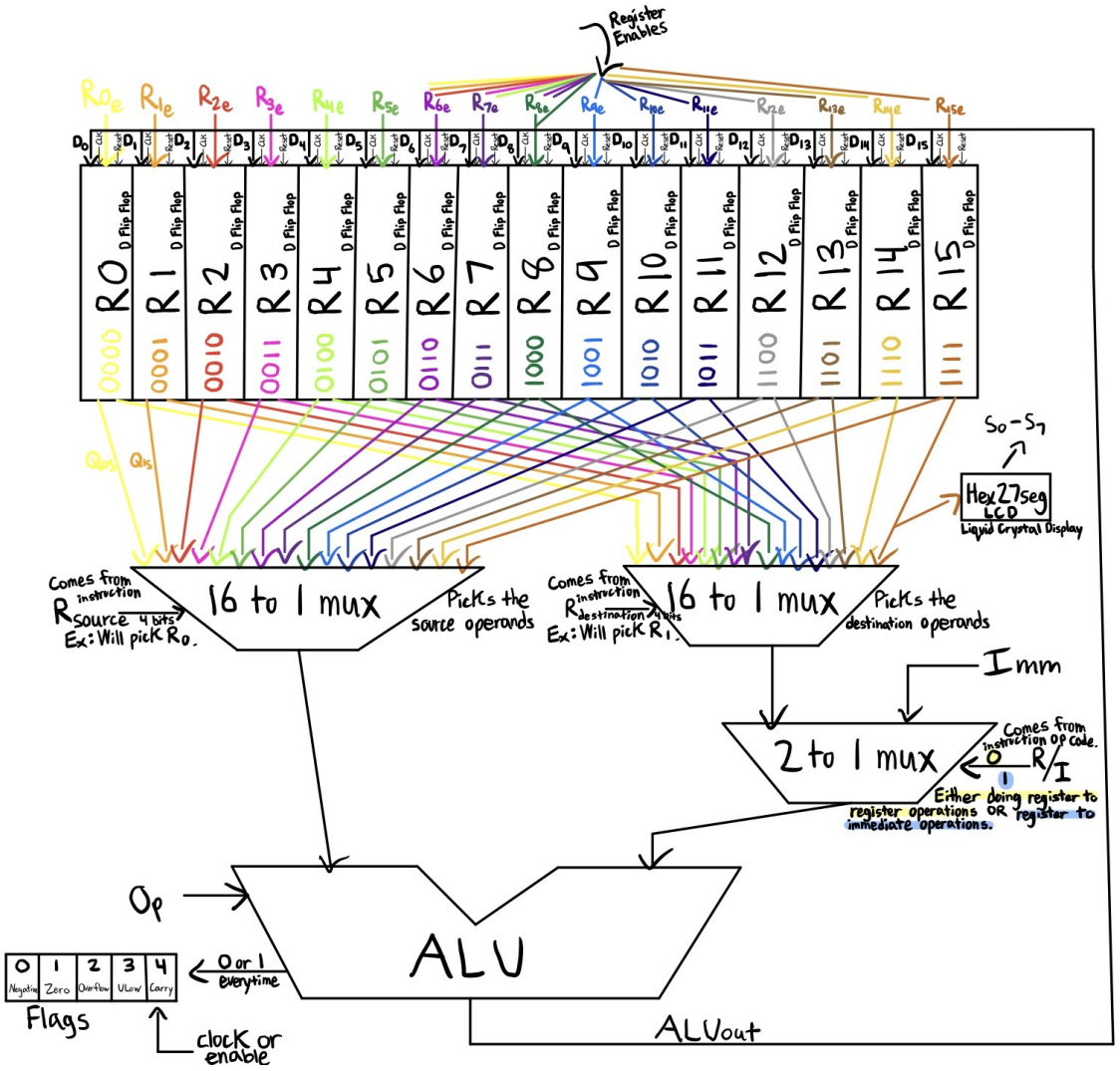

Fig. 3. Register File Block Diagram

step of the Fibonacci sequence every time the button was pressed and at the end all register contents were examined. After this we were able to confidently say that the registers were indeed working as intended.

IV. Memory through Block Ram

In conjunction with our ALU and register files, a Central Processing Unit or CPU needs a memory component in order to store data. We utilized the Quartus true dual port ram memory with a single clock in order to implement our ram on the FPGA M10K block RAM components.

A. Implementation

Initially, We synthesized two blocks of bRAM which operate on the posedge of the clock. This will allow us to read a text file into memory that contains encoded machine code instructions and thus will help facilitate the ability for our design to run a program on its own. This machine code will ultimately hold the software component of our final project. In addition, our memory element has the ability to read and write values simultaneously with the two ports. This will be essential for storing the values computed by our ALU and accessing the memory to be used for the VGA at the same time.

Upon Synthesis of the design it was found that it took up a total of 16,689 ALMs out of a possible 32,070. This is a big jump from previous assignments and therefore is slightly concerning. It appears that

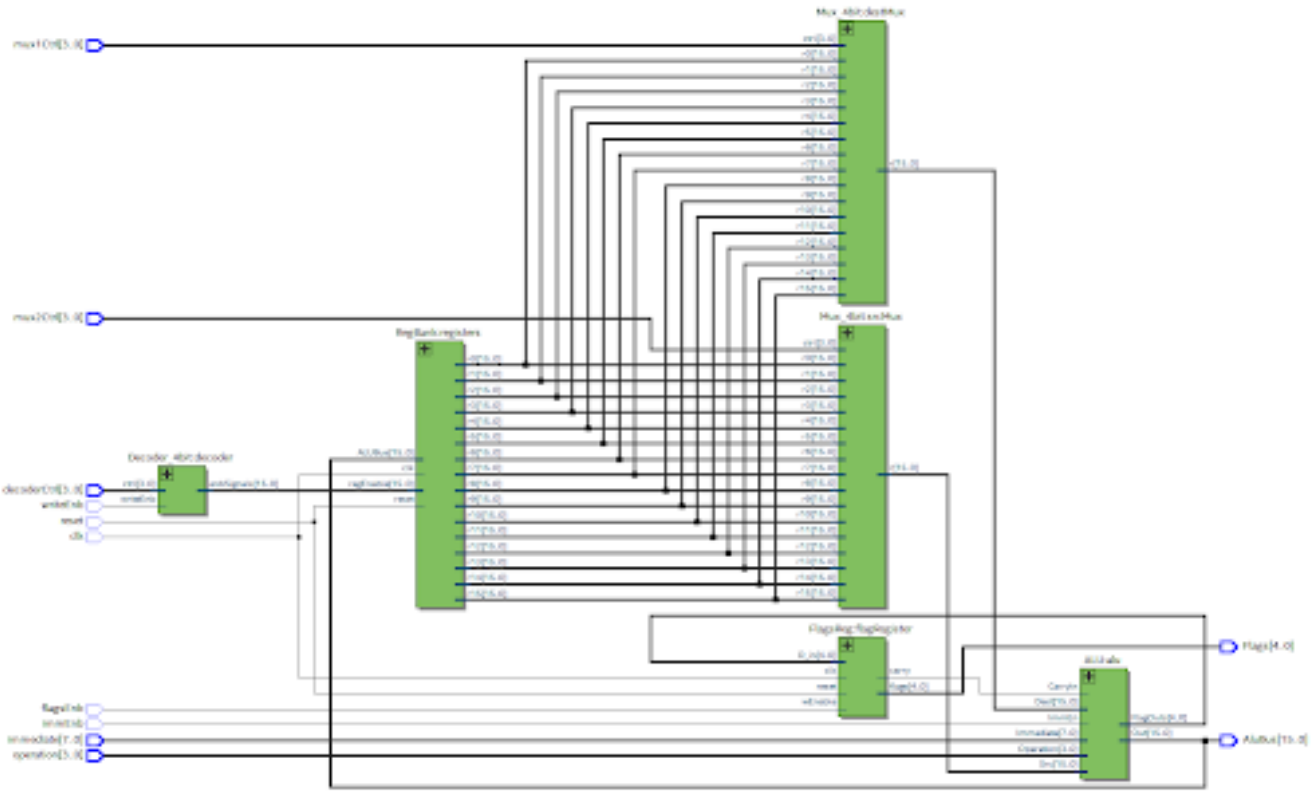

Fig. 4. Register File RTL Schematic

this is because the design is trying to compile without using the block memory on the board. After careful analysis, it was discovered that the ram was attempting to ”read during write” which is not supported. After changing this, we were able to synthesize the ram correctly. For the program, we increased the number of blocks to include 65536 words, which is the maximum that our 16 bit addresses could use. We also decided to trigger it on the negative edge to properly synchronize with the state machine, as explained below.

B. Testing

Once again both a simulation test bench and an on board tester were designed to ensure that the bRAM was working as intended. For both they stepped through a sequence of steps to load and store multiple different values in memory using either port for memory accesses. From this we were able to determine that values were indeed being loaded and stored in memory correctly. Also we created a sample hex file for the bRAM to store initially and were able to examine the original memory contents to determine that the initial values were being read in correctly as well. We were able to get this working both in software as well as on the board.

V. CPU Control and Integration

Our next task was properly integrating all of the above elements to create a working CPU. In conjunction with our ALU, register files, and memory, a control element which manages the instructions and the control signals is needed to create a full CPU. To fulfill this requirement, we built the CPU Control, which includes a control FSM, instruction register, instruction decoder, condition checker, and the program counter. This became the main element of our newly revised datapath, which can be seen diagrammed in general form below.

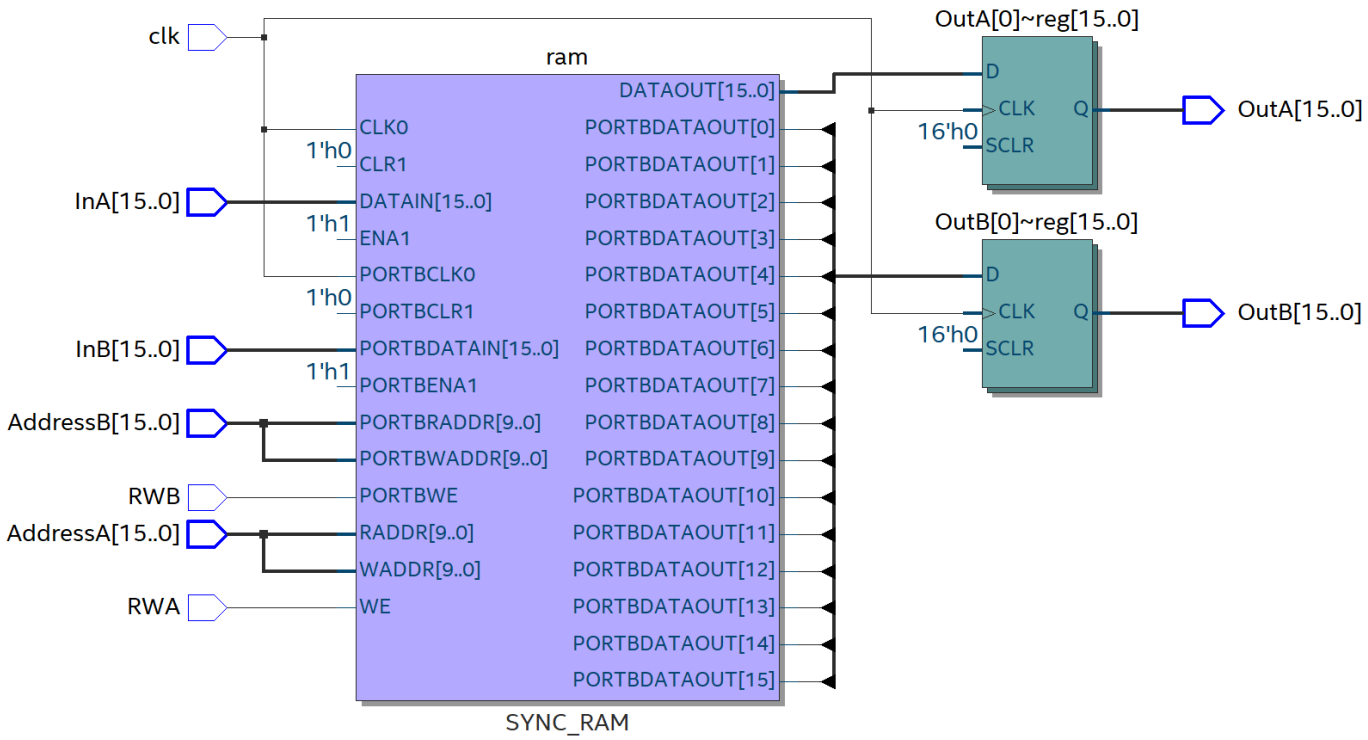

Fig. 5. Memory RTL Schematic Using Block RAM

A. *Implementation*

We initially began by implementing a program counter and instruction decoder, and tried to work with these in a single-cycle CPU design. This ultimately failed because of its inability to support load and store, which take multiple cycles, as well as clock synchronization issues for the RAM, causing improper instructions to be executed. Therefore, a state machine was created to divide the work into manageable parts so that all the instructions could be executed. For ease of control signal routing between these components, we eventually integrated all of these modules into a monolithic CPU control, which combined the program counter, combinational instruction decoder, and state machine into one module. Each module is detailed below.

In general, a program counter keeps track of the location of the instruction that is being executed, meaning that it can be used to look up the proper instruction on each cycle. In normal operation, the counter is incremented by one for each trigger, leading to a linear execution. Our counter also performs the jumps and branches required by the program. In a situation where the program counter would normally update, the instruction decoder can signal that an absolute or relative jump may happen, and the condition checker signals whether the conditions for doing so have been met. Assuming the condition is met and an absolute jump is requested (for jump and jump and link), the program counter replaces its stored address with the provided address. In like manner, if the condition is met and a relative jump is requested (for branches), the program counter adds the provided offset to its internal count. To add additional functionality, the program counter can choose between getting this offset from a register, or sign extending an 8 bit immediate. This expands our branch possibilities with the ability to calculate the correct amount to branch, useful for performing an inline jump table.

The majority of the control signals in our design are determined by an instruction decoder. When hooked up to the instruction register, the decoder creates all of the control signals needed, such as the ALU operation, register selection and input control, the condition codes and jump enables, and even the memory addresses and read/write flags for load and store. A specific part of the instruction decoder called the condition checker combined the flags with the input condition codes to determine if conditions were

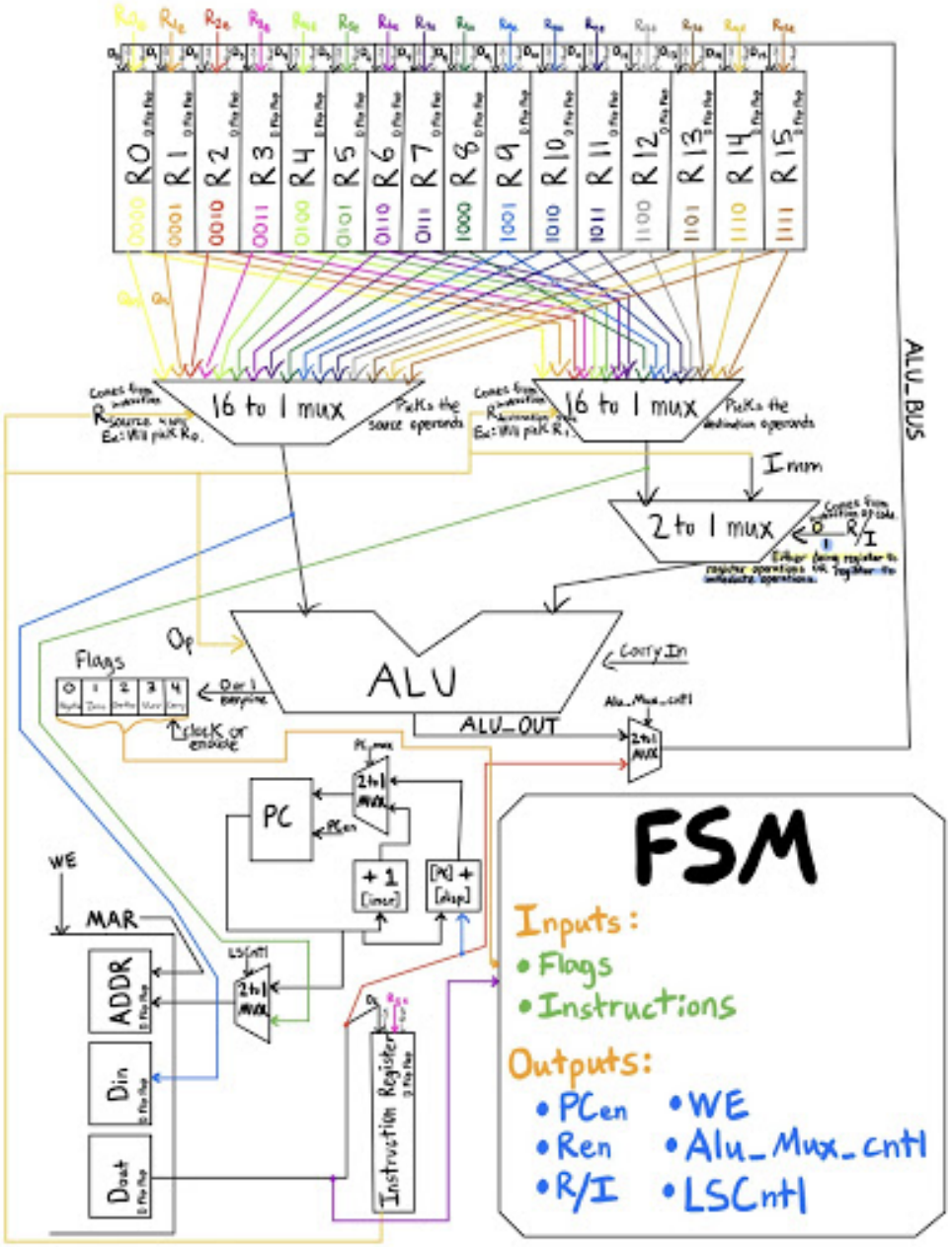

Fig. 6. General Block Diagram of the CPU

met. Because it is implemented entirely combinationally, the decoder always produces every signal, even if they will not be used, requiring that writebacks to all the registers also have control signals so that writebacks do not happen when they shouldn't, for instance in a branch instruction. The only signals that the instruction decoder does not handle are those which are timing related, specifically the multiplexer which controls what source the memory should use as its address, the signal edge which causes a write on registers and the program counter, and the instruction register controls.

The last major part built was the state machine to control the timing of all the actions. Initially, we used 3 states to complete most instructions and 5 states to perform read and writes to memory, but after bug fixing, we found that our concerns about setup and hold times for the memory were unfounded. We also found that the ALU instructions could finish in a decode state, so we reduced our total number of states to four, with state transitions happening on the positive edge of the clock. As seen in the state graph below, all instructions, including the instruction occurring after a reset, begin in the FETCH state. This state

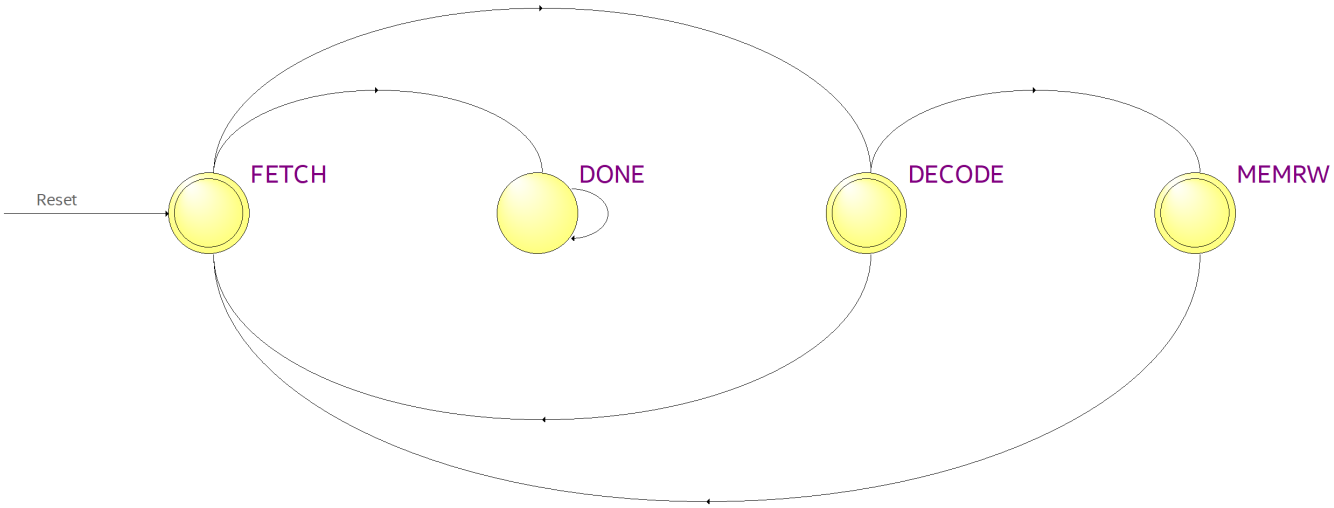

Fig. 7. State Transition Diagram for the CPU Control

ensures that the memory is set up to read data from the address on the program counter, such that at the negative edge of the clock, the instruction is retrieved and available on the change of state. If the program counter is at the end of the memory, the next state is DONE, which is a state to wait for a reset, created for compatibility with programs that terminate. Otherwise, the instruction taken from memory is stored in the instruction register, and the machine moves to the DECODE state, where all the control signals are determined by the instruction decoder and the alu and program counter perform their calculations as needed. On the positive edge at the end of the decode state, the program counter is updated, and the type of instruction is read. For read and write instructions, the state is changed to MEMRW to perform the memory interaction. For all others, a clock is delivered to the registers to allow any data that should be written, and the machine is advanced to the FETCH state. In MEMRW, the address to the memory and read/write state are determined by the instruction, which allows the memory to perform the proper operation at the negative edge. At the completion of this cycle, the registers are updated if a word was loaded from memory, and the machine progresses to the FETCH state. A diagram of our CPU control state machine is shown below.

In addition to the ALU instructions, the CPU control allows us to support Jumps on conditions, branches on conditions from both registers and immediates, loads, and stores. We also added in support for an instruction called LDKEY, which allows an exposed register in a peripheral module to be loaded. This allows us to load the keyboard state into the register for our software.

B. Testing

To test the CPU properly, we created two main programs which tested the integration of the ALU and the other instructions respectfully. In addition to getting these to run on the board with a new datapath, we additionally wrote a very simple testbench to drive the datapath. Although at various times different signals were exposed and printed, it proved extremely difficult to debug the system based on printing alone, and therefore many control signals were watched as waveforms over both programs to determine if they were functioning properly. Several issues were found as we did this, including improper memory timing, incorrect condition codes, and a few instructions that were implemented later into the ALU which malfunctioned in some scenarios.

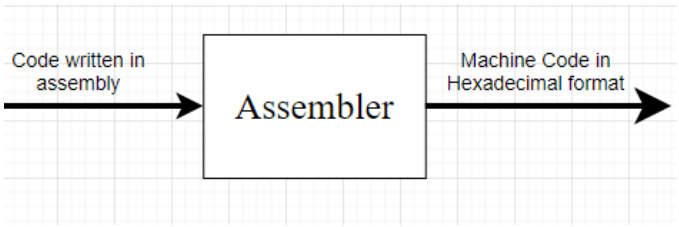

Fig. 8. Overview of the Assembler

VI. Assembler

In order to convert our game software code to a format that was compatible with our CPU, we had to write an assembler. This assembler is fairly basic, it takes a text file containing the assembly instructions, one per line, and converts them first into their binary equivalents using the encoding described in the CR16 instruction set architecture. These binary values are then converted into their hexadecimal equivalents. These converted instructions are then output to a hex file that can be read into our memory. An overview of what our assembler does is shown in Fig. 8.

VII. Software Design

In order to properly run the Tron game, our software was built around a central matrix which represented every position that could be attained in the game. With positions starting from what would be the top left hand corner of the game board, each position was represented by two bits in memory, with 8 positions per 16-bit word. Using this, we could track the past positions of players and check for collisions properly, as well as communicate with the VGA module through memory. From the perspective of the software, a 00 represented an empty square, and a player number was written depending on if a player visited that square. Player positions and directions were kept as reserved registers, with the position being encoded linearly, allowing us to use the upper bits as the word position in the matrix and the bottom three as identifiers for which position in the word. The other reserved register for each player held their jump cooldown, jump active flag, and direction information, allowing for the position to be accurately updated.

A. Implementation

Our program began with a menu loop. For ease of programming the display adapter, we stored simple menus in the memory in the same matrix format, and stored a word in memory to determine which state the game was in for the adapter. The software then checked to see if the spacebar was pressed on the keyboard, which would stop the loop. The game would then set the display flag such that the game would display, and set up the game by clearing the entire game matrix, resetting all the controlled register values, and writing the proper starting positions in memory.

The game loop was then performed until the game ended. To start, the keyboard was read. For each player a counter is stored in the top 8 bits of their direction register to determine if it is a valid time for them to jump. If it is any number, it is decremented, but a value of zero indicates that a jump is possible, so we check whether the player's jump key is pressed. By masking the bit for 'Q' or 'Enter' from the loaded keyboard key state, we can determine if the player wants to jump. On a jump, the counter is loaded to the max value, and we load another 3 bit counter located in bits 6-4 of the direction register to show the jump length remaining. During all other cycles, this counter is also decremented if the value is above 0. With these two counters, you are able to jump for 7 positions approximately every 8 seconds. After this has been completed for both players, we update the player's directions. Their current direction will be stored in bits 1-0 of their direction register, where 00 means up, 01 means left, 10 means right, and 11 means down. Using a loop the software moves through the keys attached to each player in the keyboard

state register, determines if that key is pressed, and updates the direction if a key is pressed, ignoring any direction that would be invalid. If no valid keys are pressed, the direction remains the same.

Using the direction found above, the proper position update was applied. The game allows you to wraparound from one side of the screen to the other, so that also needed to be checked. Because the positions were linear, a wraparound from one row to the next was automatic, as we considered the single row difference to be acceptable considering that additional encoding would be required to detect it. However, the position is checked whether it is negative (for a wraparound from the top to the bottom) or above the maximum position (for a wraparound from the bottom to the top), and the proper offset was applied in either case. The next task for the software was updating the matrix. If the jump active timer was not zero, this was skipped, meaning the position was updated without checking for collisions. Otherwise, the word and offset for the matrix were calculated, and the player's number was loaded into the memory. If the position loaded was clear, the game continued, but if the position was filled, then the display flag and score was updated to reflect the winner, and the menu loop was redisplayed. Using a 50 MHz clock, the game loop completes near instantaneously, and therefore, the game loop concludes with a large busy loop to perform it approximately 30 times per second.

B. Testing

To test our software code we would first add it to the memory hex file and then run the game using the FPGA. If the game ran as intended that meant that the software was working properly. Then, if issues were encountered we would use the same code we used for testing the CPU Datapath to try and step through instructions one by one to try and debug the software and figure out which line of code was causing it to behave improperly.

VIII. Displaying the Game with VGA

Finally, we got to turn our attention to creating the game Tron. Thankfully our teaching assistant Kris Wolff was able to give us a fantastic starting place. He provided us with a detailed description on how VGA works, as well as an example of VGA code, found here [1]. This example provided us with more than half of the code that we would need to implement our Tron VGA code. Unfortunately for us however, that didn't mean that we were without issues in our initial implementation.

A. Implementation

Within our VGA Verilog code, there are three main components. The Controller, the Address Generator, and the Bit Generator. The Controller uses the board clock, hcount, vcount, hsync, vsync, and bright in order to determine the necessary values to draw onto the screen. Hcount and vcount tell the address generator the x and y position of the pixel currently being drawn on the screen, and hsync, vsync, and bright all tell the bit generator when it is possible to draw to the screen. Our Address Generator first checks the memory address that contains a flag. This flag determines which screen template to draw on the screen. If the flag's two least significant bits are 00, this tells the VGA to draw the game, 01 tells the VGA to display that player one wins, and finally 10 tells the VGA to display that player two wins. It then creates and outputs the address to pull from memory based off the offset determined by the flag. The Bit Generator is synchronized with our constant variables hcount and vcount in order to determine where on the screen to draw. It works by accepting sixteen bits of data from our bRAM, breaking the bits up into eight two bit values, then drawing on the screen, in 4x4 pixel squares, based off of those two bits. If the two bit value is 00, then it displays the background, 01 it draws player one, 10 shows player two, and finally 11 is reserved for our winning screens.

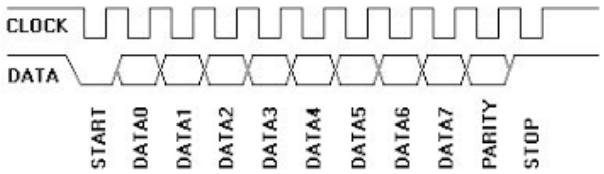

Fig. 9. The PS/2 protocol, from [2]

B. Testing

This VGA module was first tested by giving memory a file that contained test values for the pixels to ensure that a static image could reliably be drawn to the screen using values retrieved from memory. Once the VGA module passed this test, we moved on to integrating it with the actual game software to ensure that the module was reading the pixel values from the correct memory offset address. Finally, the VGA module was tested by running the full game with it to ensure that it would produce the correct movements and lines from the constantly updating matrix.

IX. Gathering Input with a PS/2 Keyboard

A. Implementation

The next essential element that we needed for our game was the PS/2 keyboard input. It was designed so that the PS/2 would listen to which keys were pressed down. A module was created that would take in the system clock, PS/2 clock, and PS/2 data wire. Using these we could then read the bits sent from the PS/2 keyboard and use it to determine which keys are pressed down and which keys have been released. When the PS/2 keyboard detects that a key is pressed it sends 11 bits on the negative edge of the PS/2 clock (which is explicitly controlled to only drop low during a send). The first bit is a low bit, the next 8 bits are the specific code sent for the associated key which are sent in least significant to most significant order, the 10th bit is a parity bit with odd parity used for error checking (it is 1 if an even number of 1 bits were in the 8-bit code and 0 otherwise), then the final bit is always a high bit (1) [2]. Fig. 9 demonstrates the sending protocol.

Now, assuming the bits were received correctly by checking the parity, first, and final bits you could use the code received to determine which key was pressed/released. Specifically if the A key is pressed 1C is sent and when it is released F0 followed by 1C is sent (F01C). Then, for the Up Arrow when it is pressed E0 followed by 75 (E075) is sent and when it is released E0 followed by F0 followed by 75 (E0F075) is sent. [3] Specifically, the code associated with a key being pressed is called its make code and the code associated with the key being released is called its break code.

The module to detect the input from the PS/2 keyboard specifically listens to the make or break codes from 16 specific keys that are used to control our game. These keys are: A, S, D, W, Left Arrow, Down Arrow, Right Arrow, Up Arrow, J, K, L, I, Q, Enter, U, and Spacebar. The module marks which keys are pressed by having a 16 bit register output which contains the “state”, documenting which keys are pressed down at that moment in time. Each key has a specific bit associated with it in the output register (W is bit 0 and Spacebar is bit 15) and when that key is pressed, that key’s make code gets sent, and its bit goes high (1). When that key is released, that key’s break code gets sent, and its bit goes low. This effectively marks all keys pressed down at all times and it is then the job of the software to determine what exactly that key being pressed down means for the game. Originally, we attempted to only use the negedge of the PS/2 clock to try and detect these codes but after some problems with the consistency of detection, especially for the codes that took more than one byte (8-bits), we used the system clock to manually perform the edge detection. From our research, this manual edge detection protects against a noisy PS2 clock signal, therefore making the design more robust and reliable. There were multiple resources available on the internet that we were able to use to help integrate the system clock into our

design. [4], [5] Once the system clock was integrated through testing we determined that the input was being detected exactly as intended.

The software does not detect if J, K, L, I, or U have been pressed because these would all be associated with a third player which was never added. We decided against adding a third player because adding it would make our game a forced three player game (since there would be no way to choose between two players or three players) which is a really odd number of players to have for the game. Therefore, we felt like the game was better as a two player game as opposed to a three player game but the inputs are still detected in the keyboard module in case we ever decided to add a third player.

B. Testing

For testing the PS2 input reader we used two main approaches. Firstly we tested it in simulation using a test bench in which the test bench would send the correct bits for an A key press and a Down Arrow key press. Then, using print statements in the test bench we could examine that the resulting keysPressed register output was correct. Then, the other thing we did to test is we created a separate datapath module for the keyboard input reader and set it up such that the keys Pressed register is tied to certain LEDs that light up when a certain key is pressed and when that key is released they go off again. This was tested on board using an actual PS2 connection allowing us to conclude that everything was working properly. As a note these testing files were designed when we were only listening to the direction commands for player 1 and 2 and the space bar. Then, we didn't perform as extensive testing when adding in the possible player 3 inputs and the jump key inputs since they followed the same logic in the code that the other key detection methods did meaning we were fairly certain that they would work when they were added. Therefore, the testing for these additional keys to listen to came by running the entire module and if q and enter were detected correctly and caused the game to do something that meant that their presses and releases were being detected correctly.

X. Menu Bitmaps

The other essential part of the memory used for our game is the menu bitmaps. These are drawn the same way the game matrix is drawn meaning that each menu consists of a bit mapped matrix to make it display correctly. In order to design this bitmap a full scale representation was designed using excel and filling the boxes that would be filled on the game screen. From this by hand we went through and assigned values such that when the VGA reads the memory matrix it would display the menu screen properly. These matrix bitmaps were stored @C000 and @D000 in the memory hex file for the player 1 win screen and the player 2 win screen respectively. These elements are represented in the memory hex file below these labels. Then, the memory file also contains an @B000 label with a 0001 underneath it to cause the game to start on a menu screen. Since this is where the flag that tells the VGA what do draw is stored and 1 means draw the player 1 winning screen.

XI. Final Resource Utilization

As with any other piece of hardware there are only so many resources you have at your disposal and the FPGA is no different. In the end our final complete design including all software, memory, the full CPU, the PS/2 input reading, and the VGA module used 839 of the 32,070 ALMs (3%), 420 total registers, 61 out of 457 available pins (13%), 2,097,152 out of the 4,065,280 block memory bits (52%). Then, the router interconnect usage once again only used 3% of the devices available resources. As for timing constraints the final compilation reported that there was 6000 ns of routing delay which is 3.4% of the available device routing delay and this delay is not a problem since everything works as intended when run on the board. Overall, none of these ended up being very close to the FPGA limits and therefore we are left with ample room to expand our game if desired. The resource usage summary generated by Quartus is shown in Fig. 11. We also checked that the 50 MHz clock we gave the board would satisfy

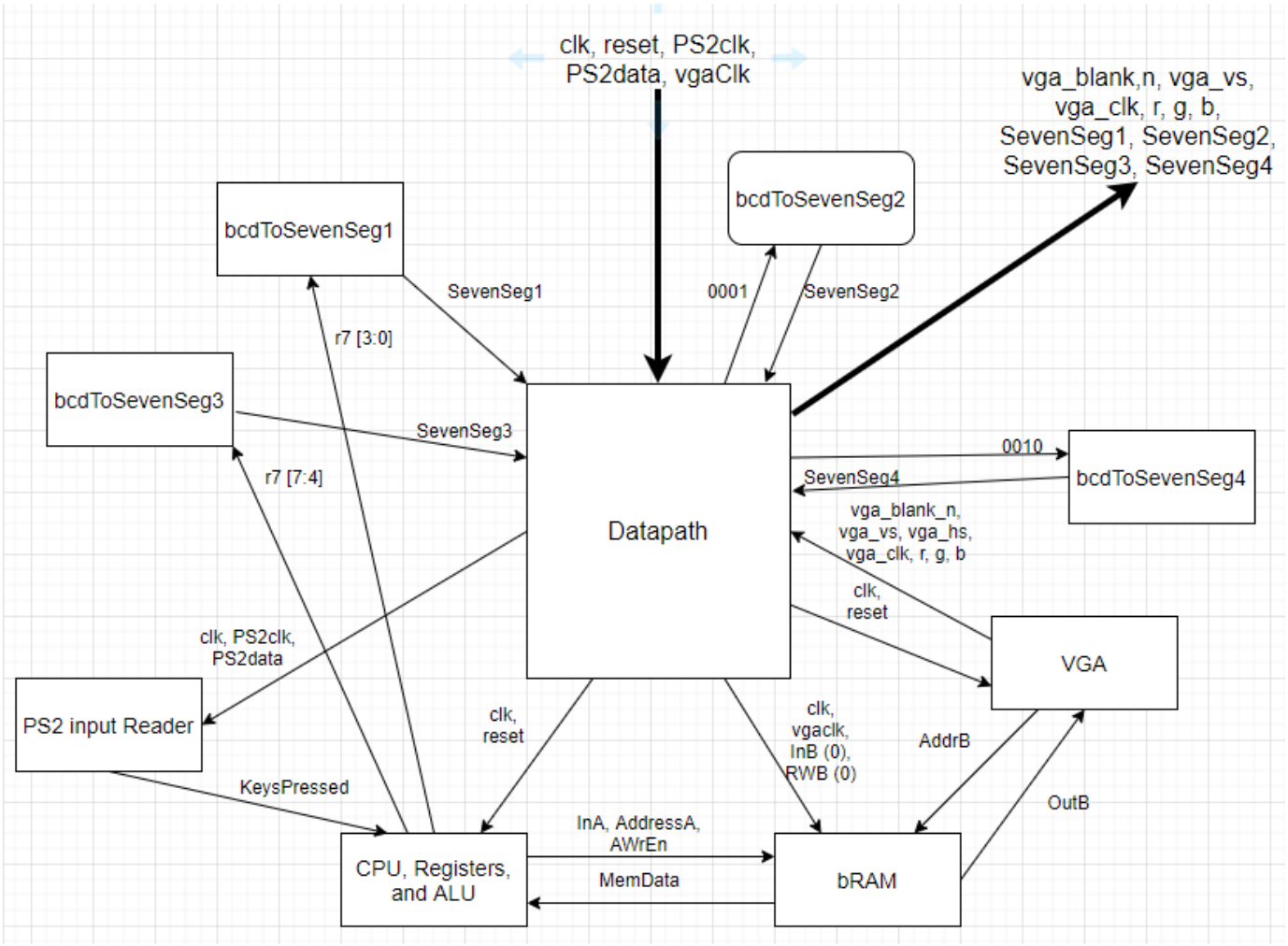

Fig. 10. Fully integrated Datapath

timing requirements. The timing analyzer in quartus reports that our design will work with up to a 69.2 MHz clock, which means that our design should be stable with the onboard clock. Our full integrated block diagram is shown in Fig. 10. The datapath is the central module and the thicker arrows are its input and outputs.

XII. Contributions

Overall, the work was split up fairly evenly between each of our team members. With the ALU, Regfile, bRAM, and CPU Control, we rotated who was implementing the actual module and who was testing it. After that, in order to complete the final project on time, we split up the work. With Michael Linnebach and Benjamin Wadsworth heading up the development of the game software, and Brady Smith and Brian Burton working on the VGA implementation. Michael also implemented the keyboard interface. After realizing that the VGA was going to take more time than we had to figure out, and after Michael and Benjamin finished the software code, we all started working on the VGA together to get it finished in time, which proved successful.

XIII. Lessons Learned

There are a number of lessons that we gained from this project. First and foremost is that teamwork makes the dream work. There is no way that we could have finished this project successfully without the

Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	ECE3710_Project
Top-level Entity Name	CPUControlDatapath
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	839 / 32,070 (3 %)
Total registers	420
Total pins	61 / 457 (13 %)
Total virtual pins	0
Total block memory bits	2,097,152 / 4,065,280 (52 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Fig. 11. Resources report for the completed CPU with peripherals

help of everyone in our group especially with the time constraints put upon us. We were also reminded of the importance of testing the code that we wrote thoroughly. You can never have enough test cases.

Arguably, our biggest hurdle was getting the VGA code to behave how we wanted it to. We ran into quite a number of issues. Our main issues were getting the address generator for the VGA to be synchronized with our bRAM module so that the correct addresses were sent to bRAM at the correct time, as well as the data from bRAM was being received by the bit generator at the correct time. Once we got our whole team working on this issue however, the issue was found and corrected. Also, at one point it mentioned that we ran out of LABs while compiling the new VGA code, which was very strange as our code absolutely should not have been using that many LABs. Thankfully with the help of Professor Kalla and our teaching assistant Kris Wolff, we were able to find the bug and get it fixed.

Another fairly large hurdle we had to overcome was a weird integration issue. We learned that with the current ISA if you set FS and FC to be associated with condition code 8 and 9 and the be based off of the overflow flag it caused the VGA to draw weird red lines on the screen. We suspect this may be due to how wires were assigned and some delays in the system when quartus compiled the design. If you change the ISA to make FS and FC be associated with 12 and 13 instead and have LT and GE associated with 8 and 9 then it fixed the issue so this is what we did to fix the error.

XIV. Conclusion/Future Work

Our redeveloped video game of Tron was a success. We are very proud of the way that it turned out and the work that we were able to achieve. There were many bumps in the road that we encountered, but we were able to find a way to overcome them by helping one another out when needed. Although the game is not perfect, the parts of the game that we were able to complete are good. The player lines are distinguished from the background, well defined, and bright. Their movement is silky smooth and clear. It is vibrant and exciting when a player wins the game. Our CPU has enough memory, is able to access the memory quickly, compute the needed values, and runs smoothly.

If we were allotted additional time, we could have made quite a few more improvements to our game. The first thing that we would have done is included the iconic Tron grid lines in our background. We also would have included a scoreboard at the bottom of the screen that would keep track of the number

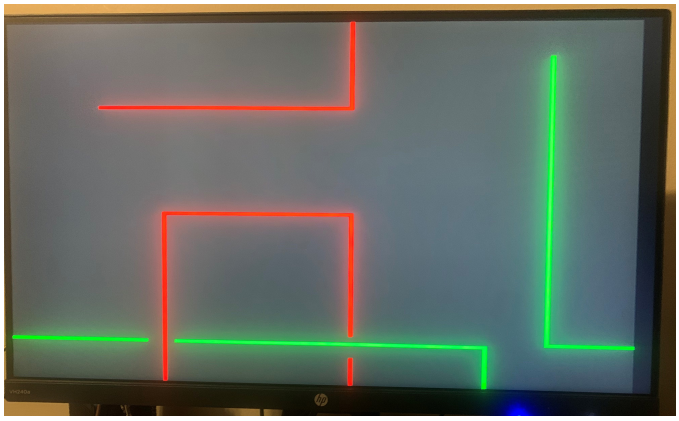

Fig. 12. A picture of the Game while Running

of wins for each player instead of just displaying the scores on the seven segment display. We also would have liked to have created a game menu of sorts and maybe added in the ability to choose a 2, 3, or 4 person game. Then, another thing we would have added is to have a dedicated start screen as well instead of having the game start on the player 1 win menu. However, since our full focus was not on creating a video game in the early stages, but rather focused on a robotic artist, we are extremely thrilled with the outcome and what we were able to get working. Fig 12 shows the game screen for an in progress game for which jumps have been used.

REFERENCES

[1] K. Wolff, "LimeSlice/vga_glyph," GitHub, 2020. [Online]. Available: https://github.com/LimeSlice/vga_glyph. [Accessed: 15-Nov-2020].
[2] A. Chapweske, " PS/2 Mouse/Keyboard Protocol," The PS/2 Mouse/Keyboard Protocol, 1999. [Online]. Available: http://www.burtonsys.com/ps2_chapweske.htm. [Accessed: 22-Nov-2020].
[3] " PS2 Keyboard Scan Codes," PS2 Keyboard Scan Codes — Online Documentation for Altium Products, 2017. [Online]. Available: https://techdocs.altium.com/display/FPGA/PS2 Keyboard Scan Codes. [Accessed: 22-Nov-2020].
[4] LBEbooks, "Lesson 110 - Example 75: PS2 Keyboard Interface," YouTube, 21-Nov-2012. [Online]. Available: https://www.youtube.com/watch?v=EtJBqvk1ZZw. [Accessed: 22-Nov-2020].
[5] Montvydas and Instructables, "PS2 Keyboard for FPGA," Instructables, 04-Oct-2017. [Online]. Available: https://www.instructables.com/PS2-Keyboard-for-FPGA/. [Accessed: 22-Nov-2020].