ECE 3710 Final Report:
Lunar Lander

Alex Hudson
Bailey Martin
Dawson Mildon
Nathan Sartnurak

University of Utah
College of Engineering
Electrical and Computer Engineering Department

ECE 3710 Final Report

ALEX HUDSON, University of Utah, Electrical and Computer Engineering, USA
BAILEY MARTIN, University of Utah, Electrical and Computer Engineering, USA
DAWSON MILDON, University of Utah, Electrical and Computer Engineering, USA
NATHAN SARTNURAK, University of Utah, Electrical and Computer Engineering, USA

Abstract-The control processing unit (CPU) was implemented in Verilog HDL. The CPU module performs arithmetic logic, controlling of read only memory (RAM), and executes instructions by reading the reduced instruction set architecture (RISC). The CPU architecture involves making connections between the arithmetic logic unit (ALU) and register files in Verilog HDL. The RAM was created in Verilog HDL using Quartus template files. The CPU Datapath and integration among the register file, ALU, and the RAM was implemented through the finite state machine (FSM), which read the RISC instructions. To create the Lunar Lander game, integrate the CPU RISC to read PS2 keyboard input to distinguish between 'w', 'a', 's', 'd', and space bar. To draw the game, use the video graphics array (VGA) to draw the background, platforms, and the ship's location. To create logic for the game, use Sam's assembler to covert the assemble code to the machine language.

Additional Key Words and Phrases: CPU, Signed extension, Unsigned extension, Registers, RAM, R-type instructions, Load instructions, Store instructions, Jump instructions, Store instructions, VGA, Velocity, Frame count, Acceleration

ACM Reference Format:
Alex Hudson, Bailey Martin, Dawson Mildon, and Nathan Sartnurak. 2021. ECE 3710 Final Report. 1, 1 (December 2021), 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The CPU is the core computing device on the motherboard. The CPU performs arithmetic logic, controlling of RAM, and executes instructions in a computer program. The CPU acts as the operation center for a computer, dictating how other components of the computer system should behave based upon the input from the user. Verilog HDL was used to design a CPU and synthesized onto field programmable gate array (FPGA) board. The created CPU in Verilog HDL reads the RISC, works with memory and the arithmetic logic unit, and handle inputs and outputs. Once the CPU was implemented in Verilog HDL, it was integrated with a PS2 keyboard and the video graphics array (VGA) to create Lunar Lander, a spacecraft game whose objective is to land on the surface of the moon.

The following section will first discuss the CPU Organization. The first subsection will go into depth about the ALU implementation. This subsection will include how the ALU handles signed and unsigned representation, immediate operations, sign extension, the complete implementation, and include a table of operations the ALU can perform. The following subsection will discuss the register file implementation. This subsection will include the register file design, how to read and write into registers, how to integrate the register file with the ALU, and

Authors' addresses: Alex Hudson, University of Utah, Electrical and Computer Engineering, USA, Alex.Hudson@utah.edu; Bailey Martin, University of Utah, Electrical and Computer Engineering, USA, u1073316@utah.edu; Dawson Mildon, University of Utah, Electrical and Computer Engineering, USA, u1255739@utah.edu; Nathan Sartnurak, University of Utah, Electrical and Computer Engineering, USA, u1089358@utah.edu.

XXXX-XXXX/2021/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

how to implement the RAM. The next subsection will discuss the CPU Datapath, instructions, and the finite state machine (FSM). This subsection includes how to handle R-type instructions, how the CPU handles load and store instructions, how the CPU handles jump and branch instructions, and how the program counter was implemented. The following section will be the CPU integration with PS2 keyboard input and the video graphics array (VGA), which will include the software aspect of Lunar Lander. The final sections will discuss the overall system integration, the lessons learned from creating this game, how each group member contributed, and the conclusion and future work of the game.

2 CPU AND ORGANIZATION

2.1 ALU IMPLEMENTATION

2.1.1 Signed and Unsigned. We took requested specifications in consideration before creating our ALU, quickly realizing that it should not differentiate between signed and unsigned numbers. All operations that need to consider the number as signed make use of the cast $signed() (i.e., AND, ARSH, CMP, SUB). Flags in ADD, CMP, and SUB are set for both L and N flags—the program should know if they are using signed or unsigned and check the flags accordingly. Otherwise, all other operations are the same regardless of the number being signed or unsigned.

2.1.2 Immediate Operations. All immediate can be handled by the operations we have created. We decided that the ALU will not directly load immediate, but a buffer outside of it in the CPU would. This buffer will likely be a MUX.

2.1.3 Sign Extension. It was decided that the CPU would handle sign extension and pass in a number to the ALU. This is to maintain separation of concerns, the ALU is purely for calculations.

2.1.4 Complete implementation. All operations are continuous to prevent internal latching — ALUs should not require "memory" of previous calculations to carry out a new one. It is controlled by a top-level module that instantiates all cases for a result, and within an always block there is a case statement for each opcode. When the opcode signal is found within the case statement it will assign input values accordingly.

2.1.5 ALU Operations Chart.

ALUOpcode	Functionality
ADD	This operation adds two register values (Rdest and Rsrc) together, and sets all flags.
SUB	This operation subtracts Rsrc from Rdest and sets all flags.
MUL	This operation multiplies two register values (Rdest and Rsrc) together
CMP	This operation compares Rdest and Rsrc setting the L, Z, and N flags. C and F are set to 1'bx to prevent latching.
AND	This operation bitwise ANDs Rdest and Rsrc.
OR	This operation bitwise ORs Rdest and Rsrc.
XOR	This operation bitwise XORs Rdest and Rsrc.
NOT	This operation gives the complement of its input.
LSH	This operation does a left shift of the bits of its input.
RSH	This operation does a right shift of the bits of its input.
ARSH	This operation does a right shift of the bits of its input and keeps its sign.

2.2 REGFILE IMPLEMENTATION

2.2.1 Register File Design. Each of the sixteen registers is essentially one D-Flip-Flop (DFF). A DFF latches a value on the positive edge of each clock cycle (unless reset is pressed in our case). If the registers is not written to, the data will not change during the clock cycle. In order to create a register file our group used a genvar and a generate-loop to instantiate 16 registers all running on the same clock and reset. A 4to16 decoder was created to only allow a single register to be written to at a time and a master enable was developed for the entire RegFile, both must be true.

2.2.2 Reading and Writing (into) Registers. Reading and writing to any given register is handled by our RegFile module. The RegFile module takes in an RdestRegLoc variable and an RsrcRegLoc variable. These variables describe the location of Rdest and Rsrc, respectively (via a mux), and then the value stored in the specified registers is retrieved. This is how the ALU knows what values to use for Rsrc and Rdest. When writing to a register, first the RegFile needs to be enabled. If the RegFile is disabled, writing to any register is disallowed, regardless of RdestRegLoc. Only RdestRegLoc is used because Rdest is the only register that is ever written to. The register is chosen by sending RdestRegLoc through a decoder. The output of this decoder is sent to all 16 registers and ANDed with the RegFile enable signal. When both signals are active, the register can be written to. Writing to a register is only permitted on the positive edge of the clock, and the value that is input to the RegFile (Load), is stored in the enabled register.

2.2.3 Register File and ALU Integration. The combination of the ALU and Regfile are integrated into one large Reg_Alu.v file. The layout of this module is similar to the one given in the lab handout. The Regfile outputs the value of Rdest directly into the ALU and the value of Rsrc into a Mux. The Mux selects either the immediate value or Rsrc and feeds that into the ALU for calculations. The ALU is then given an opcode and returns its output directly to the register file to be stored in Rdest on the next clock cycle. The Reg_Alu module includes a lot of parameters to adequately drive both the ALU and Regfile. This includes basic one bit control logic: Enable, Clock, Imm_s (control logic for the mux), Reset. These are needed to ensure writing correct values to the registers, at the correct time. Four-bit inputs include the location for Rdest and Rsrc, as well as the opcode for the Alu. Then a 16bit immediate needs to be included to be able to write into the registers. Reg_Alu then outputs the 16-bit value of Rdest, and the flags from the ALU.

2.3 RAM

2.3.1 RAM Block. Our RAM_block module was a Quartus template file which we changed to better fit our needs. We used a dual-access, single clock RAM block, knowing we might have to change to dual clock when we start work on the final project. We changed the DATA_WIDTH and ADDR_WIDTH variables to match the sizes we needed and added an initial block to initialize all RAM values to 0. Again, we know we will have to initialize to other values in future labs but initializing to 0 is enough to show that our RAM stores values. Test the RAM_block just consisted of us manually changing values in the test bench. The RAM_block test bench acted primarily as a proof-of-concept because we wrote a RAM module that more extensively tested each block of RAM.

2.3.2 RAM. The RAM file currently consists of two instantiations of the RAM_block module. In our design each address is ten bits wide, and each input sixteen bits wide. Then an always block set to *, where the most significant bit of addr_a and addr_b are checked to see if they are equal to zero. If so then the corresponding q_out (addr_a => q_a_out and addr_b => q_b_out) are set to q_a[0] or q_b[0].

2.4 CPU DATAPATH, INSTRUCTIONS, AND FSM

2.4.1 R-Type Instructions. R-Type instructions are used to make operations on data already in registers. In the FSM this type takes four states and therefore four clock cycles to complete. Looking at the FSM state s2 checks

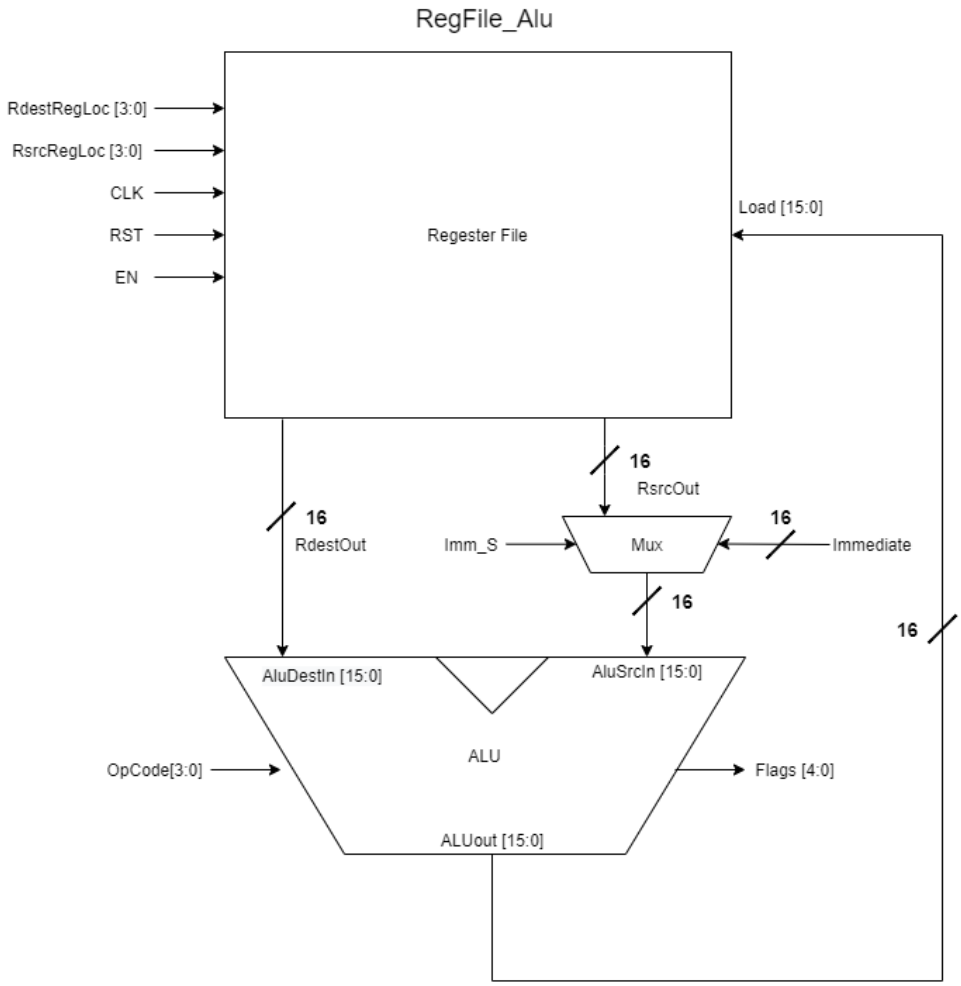


Fig. 1. Register File and ALU

for R-Type instructions; if saved instruction's four most significant bits equal all zeros (i.e. savedInstr[15:12] == 4'b0) then set next state to s3. State s3 obtains the stored locations for Rsrc (i.e. RsrcRegLoc = savedInstr[3:0]) and Rdest (i.e. i.e. RdestRegLoc = savedInstr[11:8]), and enables the regfile. It then also sets the operation code for the ALU by using the second to last four bits (i.e. savedInstr[7:4]) and setting ALUOpCode to the correct function to be carried out (i.e. if savedInstr[7:4] == 4'b0101, then ALUOpCode = ADD). Since these were the first instructions to be implemented there a few challenges in general. Firstly, the FSM had to be designed, it took time to figure out how to prevent internal latches and the timing to change states (both present and next state). Secondly, the CPU's datapath had to be created. At this point for R-Types it was fairly easy, but implementing load/store and jump/branch instructions involved adding more that module instantiations (such as MUXs). At the beginning during the work on R-type instructions the team struggled with how to put it on the board; creating instructions and getting the board to read them in.

2.4.2 Load/Store Instructions. Loads and Stores are the instructions that have the most states in the FSM, and, therefore, take the most clock cycles. The reason for this is, the CPU is required to read/write from RAM which takes a clock cycle and read/write from the Regfile which also takes a clock cycle. The way in which load and store instructions operate, is first we have a decode state that determines whether it is a load or store. This state is also responsible for setting the two register locations. The register that we have called Rsrc will always contain the address for the RAM. The register we have called Rdest, will contain the information we are saving to the RAM in the event of a store and will be the register we overwrite in the event of a load. To accommodate for loads and stores, we had to add a mux going into the Regfile. Previously, only the ALU was writing information into the Regfile, but now the RAM can as well, so we needed to add a mux to determine where the information was coming from. We also needed to add a mux going into the RAM address. Most of the time, the program counter will determine what the address is for the RAM, but in the event of a load/store, the CPU will be reading/writing information from a different address location—coming from the Rsrc register.

2.4.3 Jump/Branch Instructions. To facilitate the jumping conditions, we built a specific decoder module in the FSM. The ConditionDecoder module takes the condition code, saved flags, and outputs a 1 bit true or false. Saved flags is only updated when Rtype instructions are executed. This condition decoder can then be used for the three different jump states: Scond, Bcond, and Jcond. This was a particularly useful for testing because we could test the module specifically to make sure all the codes were correct instead of testing every condition in every jump argument. We had to update the CPU path to support the three new conditions. We decided to include the logic needed for jumps within the program counter. This simplified our FSM. The updated program counter takes in a 2-bit selector, R source out, and the signed extended immediate. This one addition created functionality for Jcond and Bcond jumps. We also updated the mux that connects to the load line for the registers. The mux takes in ALU out, mem out, and now immediate. We updated the selector to be 2 bits so it can select all three options. This also allows us to add move instructions without updating the data path. The jump states are directly transposed from the original instruction. They then point to the final state, which pulls the next instruction. The automatic pull was not a problem for Scond; Scond sets the immediate to 1 or 0 depending on the output from the condition decoder. Then load selector is set to accept the signed immediate and stores it in Rdest. Jcond and Bcond were slightly more difficult. Both instructions check the condition decoder output. If the output is true, they set the jump selector to the correct signal, else they set the program counter to stay on the same instruction. Because the program counter is enabled on the final state, if we jump, we jump to the location minus one. This is because the final state will always increment one time. We do believe that there should be a more elegant way to perform jumps, however we don't see any bugs with this solution, and it was simpler to implement. The biggest challenge we had for jumps was deciding how to update to the CPU data path. We had group discussions over separations of concerns, testing, and creating bigger mux's. We decided to bundle the logic for selecting an immediate or a specific address in the program counter. While this might not be the best for separation of concerns, it made it far easier to test our updates to the CPU data path. This actually became particularly useful after testing and realizing we were skipping instructions because of the additional jump in our FSM. Simply subtracting from the selected input line allowed us not to change any logic in the CPU or FSM.

2.4.4 Program Counter. The purpose of the program counter is to be able to increment the counter value on the rising edge of the clock or change the counter value to either the memory address value or add the immediate value depending on selector input. To do this, the program counter takes in 6 inputs: the clock, the reset, an enable for the program counter, a 2-bit selector, an immediate value, a memory address value, and outputs the current count of the program counter. The selector 2-bit input dictates how the counter will behave. If the selector input value is "00", then the counter will perform the default case which is incrementing the counter by 1. If the selector is "01," it will take the signed extension of the immediate value, add that value to current value of the program counter, and subtract 1. By signed extended the immediate value, the program counter will be able to

jump backwards. Finally, if the 2-bit selector is "10," the program counter will take the memory address value, subtract 1, and set that value as the new program counter. By having a selector that dictates the output of the program counter, this allows for self-checking testbenches and easier debugging of the program counter. This also makes the program counter easier to instantiate and implement in the CPU and the CPU FSM module

3 CPU INTEGRATION WITH PERIPHERALS

3.1 PS2 Keyboard Input

A PS2 port keyboard was chosen, because it will send a key code depending on if a key is pressed or not. When a key is pressed it sends a "make code" associated with that specific key on an interval from a clock on the PS2 port for as long as it continues to be pressed in the form < break code, make code>. Otherwise, it sends a "break code" indicating that a key is not being pressed; PS2 keyboard break codes are hexadecimal "F0". Waiting for these packets is obviously much easier to implement than other options such as USB keyboards. These packets come in a bitstream of eleven bits: the first bit being a signal that a packet is coming, then next eight bits is the packet, and the last two are to signal the end of the packet. These bit packets representing the keycodes are to be read in hexadecimal for PS2 port keyboards.

For this we found example code online for taking in a singular key code that ignores the first bit of the packet, captures the eight bits of information, and signals a boolean flag with the last to bits of the packet. The boolean flag signals an always block to reassign the outgoing data to the FPGA, updating it to the new key press. Only capturing a singular key code presents an obvious problem for the arrow make codes — they were a three part make code (i.e. up arrow= <E0, F0, 75>, left arrow = <E0, F0, 6B>, and right arrow = <E0, F0, 74>). Distinguishing which arrow key was being pressed would take until the third code and seemed difficult to differentiate between a break code and make code in this situation since "F0" was sent in between "E0" and the third hexadecimal code. Quickly, the decision was made to switch to WASD (minus S as there is only up, left-side, and right-side thrusters for our version of Lunar Lander) for the game, because these keys were only a two part make code (i.e. W = <F0, 1D>, A = <F0, 1C>, and D = <F0, 23>).

Clearly, for WASD capturing the first part of the make code would appear as a break code, so the code was changed to capture two bit packets and throw out the break code to get the keyboard input from the user. First, the code was changed to capture twenty-two bits — two packets, one as the "ignore" packet and the other as the "key code" packet; there are two separate flags for the two packages . Then there is an always block that signals on the positive edge of either of these flags. Inside this always the new packet is checked to see if it is a make code (key press) or a break code (no key press) and the output is assigned to the make code once it is determined that it is not a break code.

3.2 VGA

VGA output was written in a separate module to uphold separation of concerns. Using the set VGA values—front porch, back porch, display size, etc.—we are able to calculate v sync and h sync depending on what pixel is currently being drawn to. These screen draws from the top left, to the bottom right, and every pixel is drawn once every frame. Because the CPU is running on a 50MHz clock, a (phased lock loop) PLL is used to achieve the 25.175MHz clock necessary to draw to the screen. We then output this clock, as well as each pixel's red, green, and blue values to the FPGA.

For the purposes of this game, only one coordinate needs to change: the position of the ship. Every other object on the screen does not move. Because of this, each platform and invalid area has hard coded coordinates in the VGA module. These coordinates are then used to calculate collision detection in the assembly code. Every object also has a hard coded pixel color that cannot be changed by the CPU. This is to make the transfer of information between the rest of the CPU and the VGA module simpler. At this point in time, it would be possible to have the

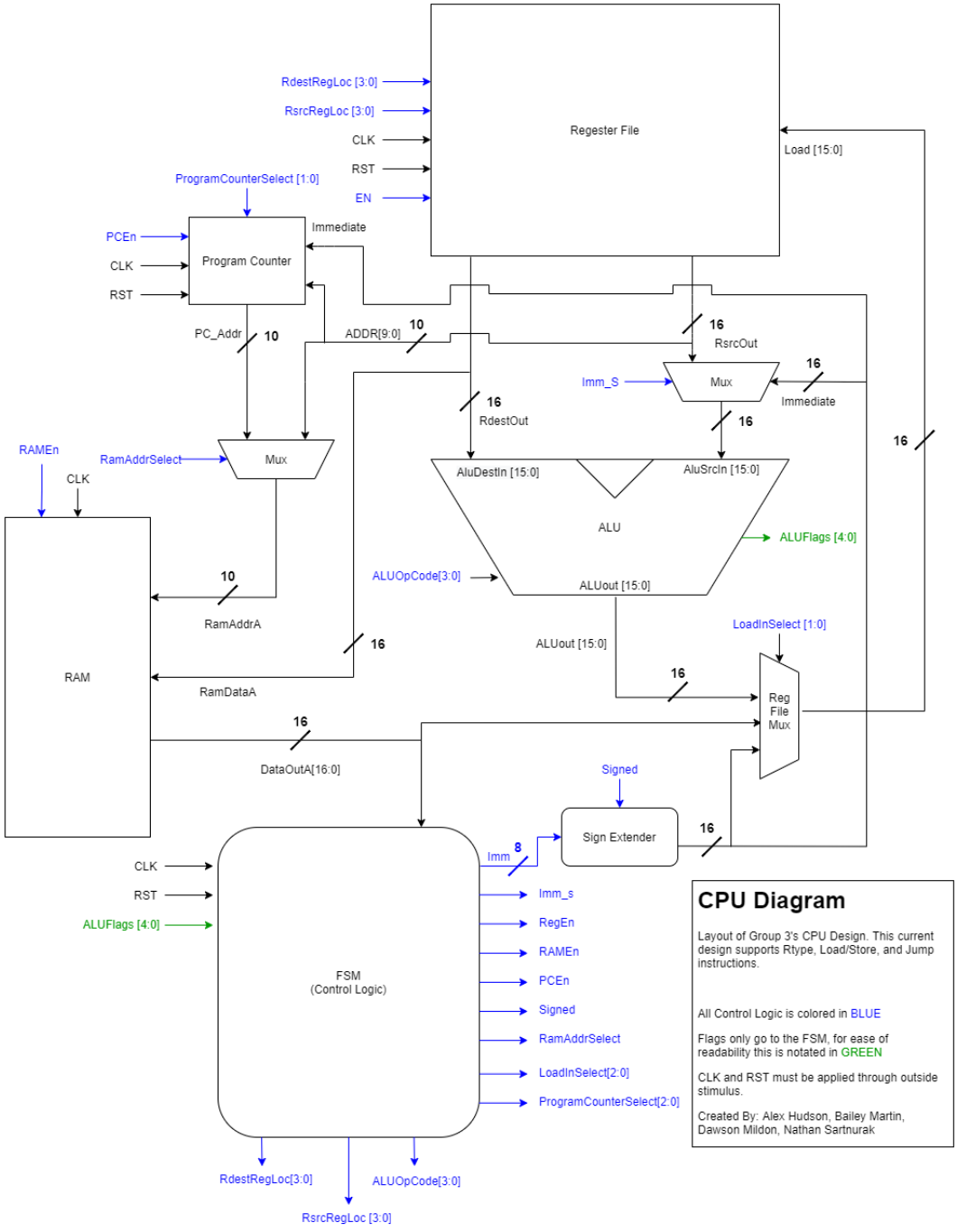


Fig. 2. CPU Diagram

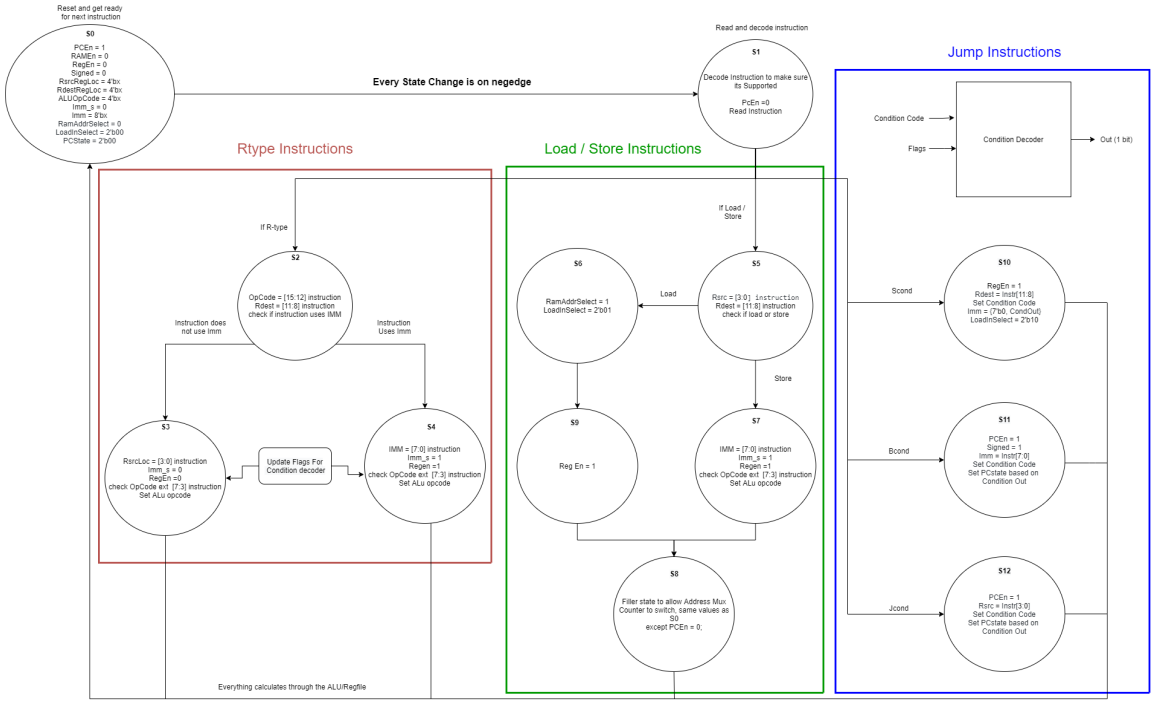


Fig. 3. CPU FSM

CPU pass in more coordinates and even pixel colors, but it would be more complex and unnecessary for game play.

At the end of every frame—i.e., the module is drawing to the bottom right corner of the screen—two registers housed in the VGA module read in the next coordinates of the ship. These two coordinates (one for x and one for y) detail the top left position of the ship. The ship's total size—15 by 15 pixels—and color is hard coded in the VGA module. This allows the top left coordinate to be passed in, and the VGA module can draw the rest of the ship in the right position from the given location.

4 SOFTWARE

The assembly followed a relatively simple path. We completed each portion listed in the figure below incrementally. First initiate all the values, then check keyboard input. The keyboard input was stored into a register and if that value was a known key code update the velocity accordingly. If it was space, the game would restart jumping to instruction 0. Then after velocity was updated, update position then draw. The draw instruction took a x and y pos from rdest and rsrc to draw the block on the screen. After position was updated, we then checked to see if the new position collided with anything. If the position collided with the left, top, or right side of the screen we would invert the velocity to give it a "bouncing" effect. Then we checked for collision along the bottom of the screen. Because the level was static, we could check each region. Our level had 5 regions in total, so we would see what region our x pos was in. Then depending on the region, we had specific y values to check against the y position. If the ship was in a specific region and below a specific Y value, we would set velocity and acceleration

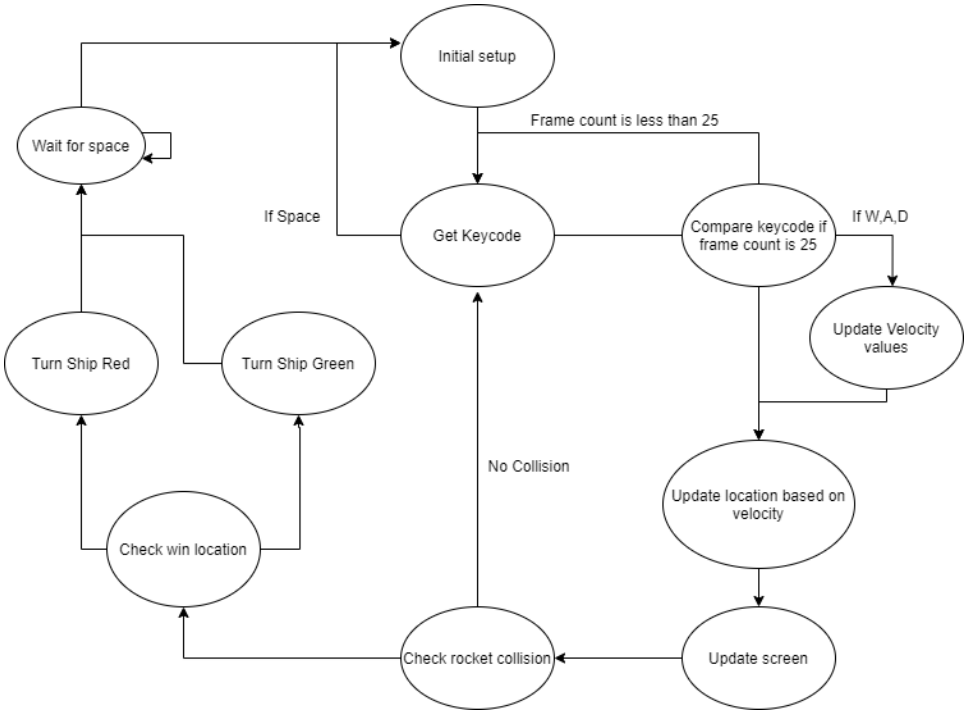


Fig. 4. Game Flow

to 0. Win detection was easy, if gravity was 0, we were in a win detection state. We would check if the ship was on a winning platform and update its color accordingly.

The assembly was probably one of the easier parts of the project. We ran into a couple issues, specifically with the given compiler not encoding an instruction correctly. Another issue was handling acceleration. At 60hz and the slowest velocity being 1 pixel per frame, the ship would cover large distances very quickly. We solved this in two ways, creating a velocity speed limit of 3 pixels per frame, and only detecting updates every 25 frames. This made game-play clunky, but we found it to be more fun and difficult. Despite these setbacks, it did feel like we were always making progress. Another reason the assembly was fairly easy is we offloaded a lot of work to hardware. The hardware draws the level and the ship, it also does the waiting to see when we can draw a frame. The assembly logic was pretty simple because of these abstractions. It was simple enough to only use the 16 registers and immediates. We did not even need to use load or store functions, and it was only about 130 instructions.

From lab 4 to our final project, we added 3 major new instructions. 2 specific instructions: KEY (save current keycode to rdest) and WAIT (waited until position rsrc and rdest could be drawn on the screen). The other instruction implemented was MOVI because we found this much easier than SUB %rdest %redest then ADDI $value %rdest. This made our code simpler and easier to debug. Overall, we are very happy with our how our assembly turned out, and we believe it produced a game that is very fun to play.

5 OVERALL SYSTEM INTEGRATION AND HOW DOES IT WORK

Our Lunar Lander game uses a PS2 port keyboard input with WASD (minus S as there is no downward thruster) keys to signal movement of the rocket ship. This keyboard press is given to an FPGA that is running the CPU via the CPU FSM instruction created to take in keyboard input. Within the assembly code the keyboard input is checked to see if it matches the four valid key inputs: W — moves rocket up, A — moves rocket to the left, D — moves rocket to the right, and the space bar — resets the game for another round. The VGA port is used to draw the output of the FPGA onto a monitor, it draws the landscape and the rocket's corresponding movements to the keyboard press. Assembly code tracks if the landing area of the rocket is valid as a win or not. If a win takes place the rocket is drawn green by the VGA and if a loss takes place it is drawn red.

6 LESSONS LEARNED

The very first lesson the team learned was that Verilog HDL is different than the C programming language. The first mistake the group made was thinking that modules behave like function calls in C. This is not the case because modules in Verilog HDL requires code runs concurrently. The next issue the team dealt with was implementing the VHDL design on the FPGA board. One of the biggest issues was that the code runs correctly in modelsim but does not behave correctly when it is loaded to the FPGA board. To fix this, the team checked to see that all inputs and outputs are correctly instantiated. For example, the team forgot to link a connection for a reset input to one of the modules. Latches were also being created since the always block did not handle all the possible cases. The final issue the team had was timing. The team learned that the FSM was not given clock cycles to correctly read and execute the RISC.

7 GROUP MEMBER RESPONSIBILITIES AND ACCOMPLISHMENTS

7.1 Alex Hudson

Alex was a major contributor to both the FSM and the Assembly in the final project. He worked closely with the instructions and understanding how they were executed. While the FSM and Assembly were his largest contributions, Alex also assisted with the PS2 keyboard reading module, Regfile design, simple ALU calculations, and implementing/debugging the jump instructions. Our group relied heavily on pair programming, because we felt it worked best for our skill set. This means that while Alex contributed heavily to the portions listed above, we concurred these challenges as a group.

7.2 Bailey Martin

Bailey was part of teams primarily focused on CPU design and software. She was also a massive contributor in the ALU, Register File, RAM, CPU datapath, and CPU FSM design. For each of these CPU components she worked with the team to make self-checking testbenches to guarantee accurate code. Within the ALU she was responsible for ADD, SUB, and CMP operations. In the RegFile she created multiple modules, most noticeably the four-to-sixteen register enable decoder. Along with our group members she implemented the CPU datapath and CPU FSM. In the final project Bailey was partnered with Alex on a team to handle the keyboard input and the assembly for the game. Bailey did research into PS2 port key codes and read extensively into how the PS2 protocol worked. In the keyboard code she largely designed the module to capture key presses and make sure it was the expected result to the FPGA. She and Nathan were the members of the team to first figure out how to use the assembler so the game could be written. She also wrote some of the assembly code for Lunar Lander. Bailey also handled putting lab reports together and delegating writing sections to members of the team.

7.3 Dawson Mildon

In the ALU, Dawson was primarily responsible for implementing the boolean logic instructions (i.e., AND, OR, XOR, NOT). In the Regfile, he instantiated the register module sixteen times using a generate loop and then helped write the logic to read information from the registers. The ram module was a default module in quartus that Dawson altered to fit with the rest of the CPU, including adding a parameter to pass in a file name to instantiate the ram. Dawson was also primarily responsible for the FSM. He wrote a lot of the logic and was the one most familiar with how the FSM functioned. Working with Nathan, he wrote the VGA module for the final project and then helped write portions of the assembly code.

7.4 Nathan Sartnurak

Nathan created self checking testbenches to check the group's code and that all the modules are functioning correctly. He integrate the program counter with the CPU, and responsible for synthesis and place and route onto the FPGA board. Nathan was in charge of the top level verilog HDL code. He taught everyone how to use modelsim to debug the verilog HDL code. This includes adding and highlighting signals to the waveforms, how to save the wave.do file, and how to recompile inside of modelsim without relaunching the modelsim from quartus prime. For the final project, he worked with Dawson to learn how to implement the VGA to correctly draw the space ship, the background, and the platforms for the game. He learned how to use and integrate Sam's assembler code to the CPU FSM. He integrated the PLL to generate exactly 25.175 MHz clock for the VGA. Finally, he worked with Bailey and Dawson to write collision and win/loss detection section in the assembly code.

8 CONCLUSIONS AND FUTURE WORK:

CPU and the data path to memory can be implemented using Verilog HDL. The first component is to design the ALU to successfully ADD, SUB, MUL, CMP, AND, OR, XOR, NOT, LSH, RSH, and ARSH. Create the register file design to be able to read and write to registers. Integrate the ALU along with the register file to ensure that RISC operations write to the correct registers, Develop the Ram by using Quartus's ram blocks. To create the RAM file, only instantiate a total of two RAM blocks. To finish implementing the CPU, create the CPU FSM to be able to read R-type instructions, load and store instructions, jump and branch instructions. Integrate the program counter to ensure that the jumps and branches are working correctly.

Once the CPU integration has finished, the next step is to create Lunar Lander. For the PS2 keyboard, create a RISC instruction to distinguish between 'w', 'a', 's', 'd', which is for movement of the space chip, and the space bar to reset the game. Once the PS2 keyboard can be correctly read on the FPGA board, use the VGA to be able to draw the game. The VGA module was first used to draw the background of the game, the ship, and the landing pads. The implement the logic of the game, first write the logic in assembly. Use Sam's assembler to convert the assembly logic code into machine language. Finally, load the machine language into the FPGA to play the game.

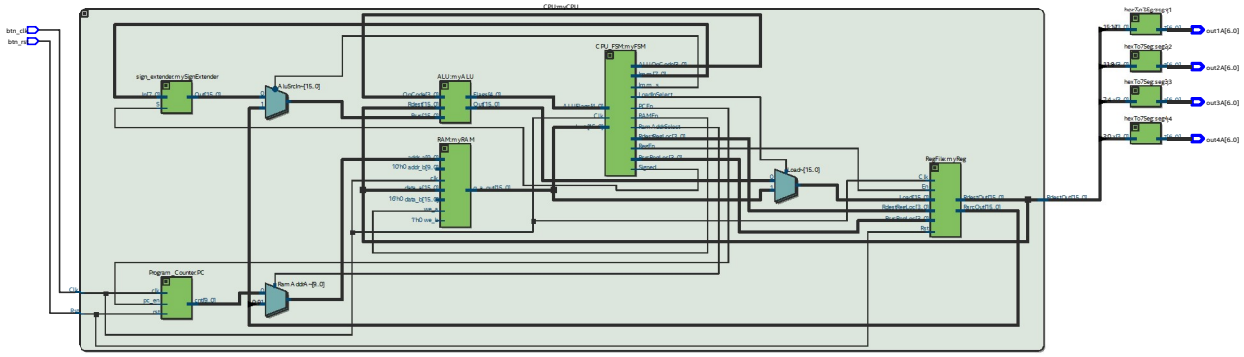
Even though the base game works, improvements for the game can still be made. The first improvement is to implement glyphs. Unfortunately, the team felt that implementing the logic of game is more important than drawing glyphs. However, if the team had more time, using the glyphs to draw the spaceship and a lookup table to store the location of the ship will be the next step to improving the game. After that, improve the movement of the ship. The glyphs code that was presented took a total of four frames and can be used to correctly display the correct frame of the spaceship based upon keyboard input. For example, as the PS2 keyboard reads an input of an 'a' the glyph should display a spaceship moving to left and a right thruster animation. This logic should be implemented for the remaining movement keyboards: 'a', 's', and 'd. Finally, the last way to improve the game would be create more logic to detect what velocity would be required to win the game. As of now, if the ship lands on the platform regardless of speed, the user wins the game. A better game would be to force the user to land the spaceship on the platform at a relatively low speed.

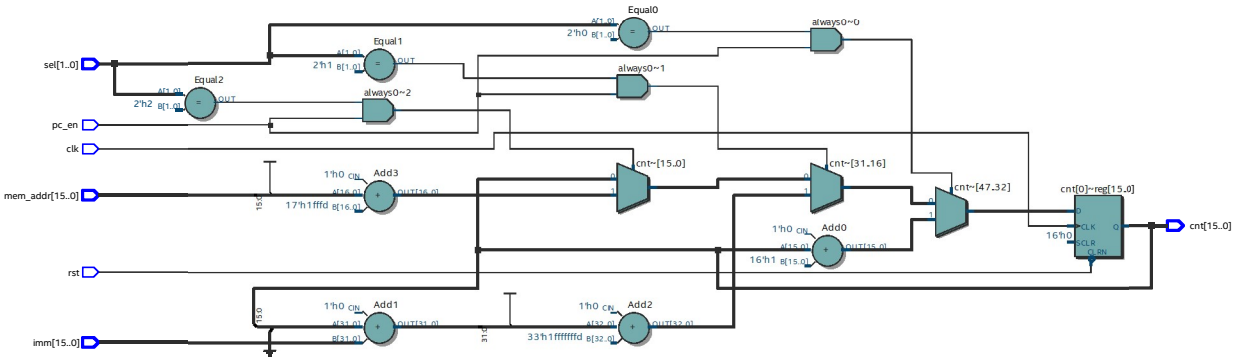
A ACKNOWLEDGMENTS

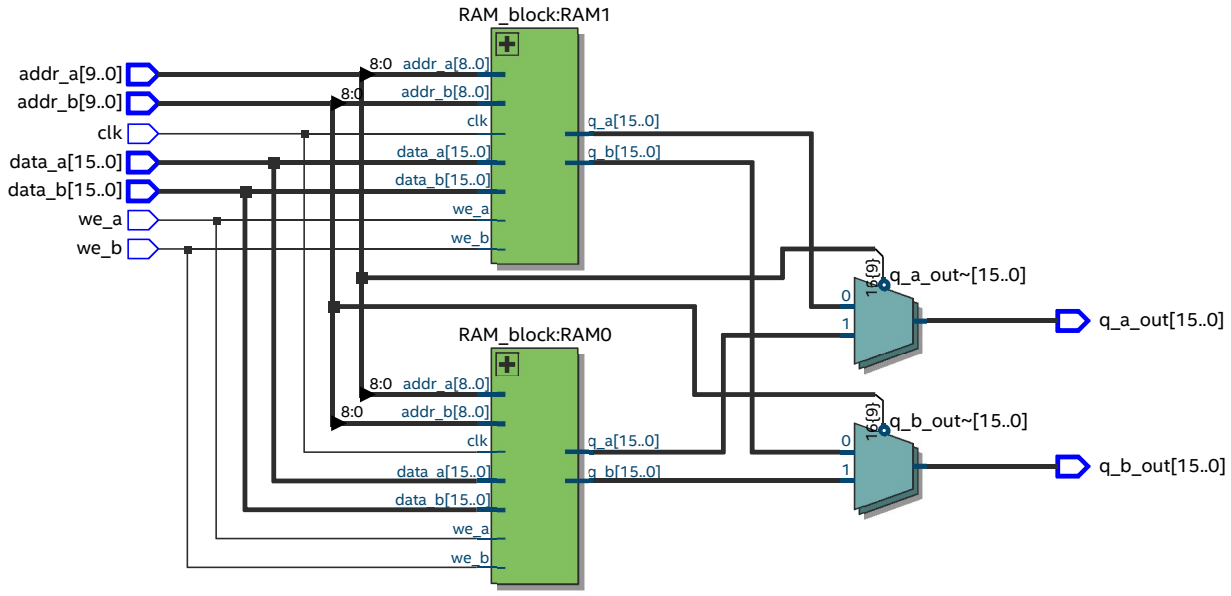
Thanks to Sam for being a wonderful TA this semester!

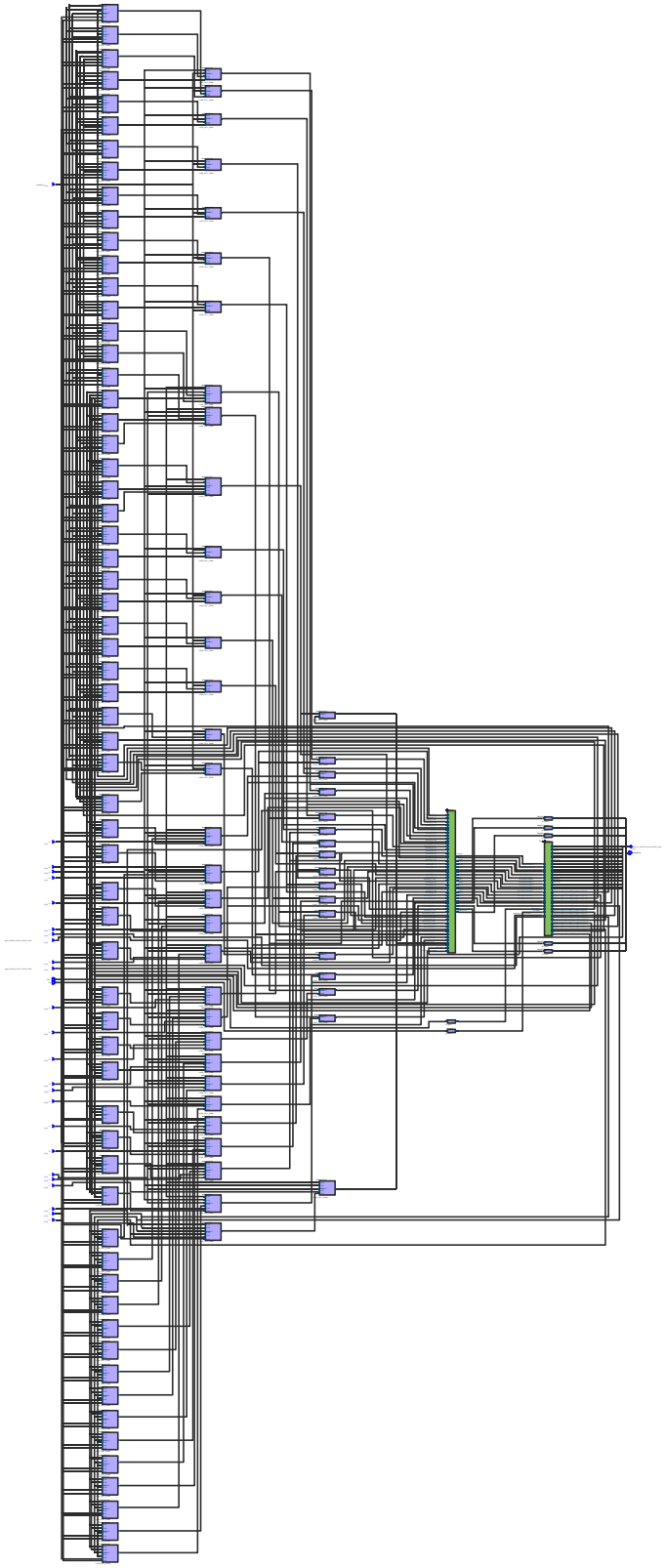
REFERENCES

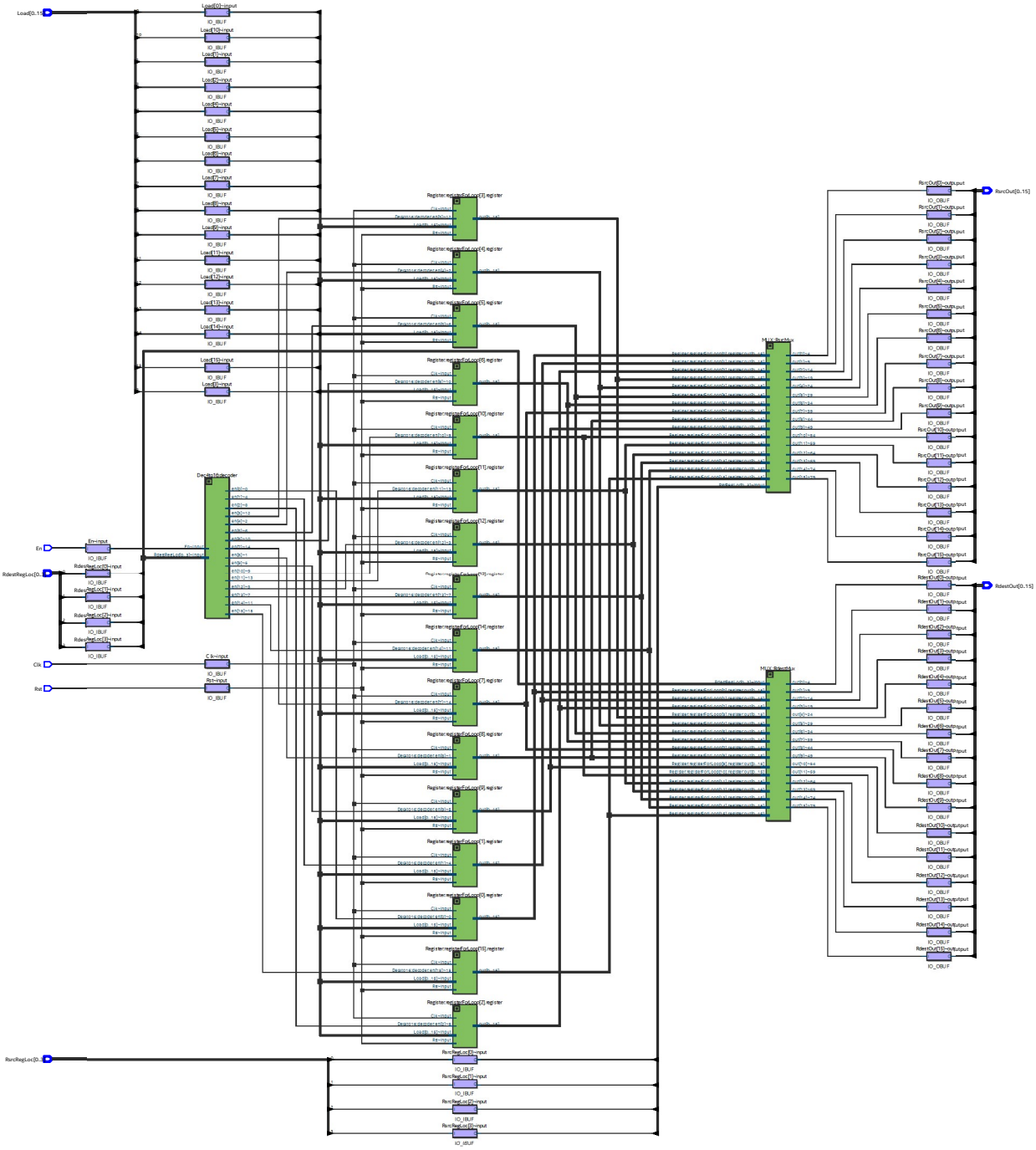
https://github.com/LimeSlice/vga_glyph
https://forum.digikey.com/t/ps-2-keyboard-interface-vhdl/12614

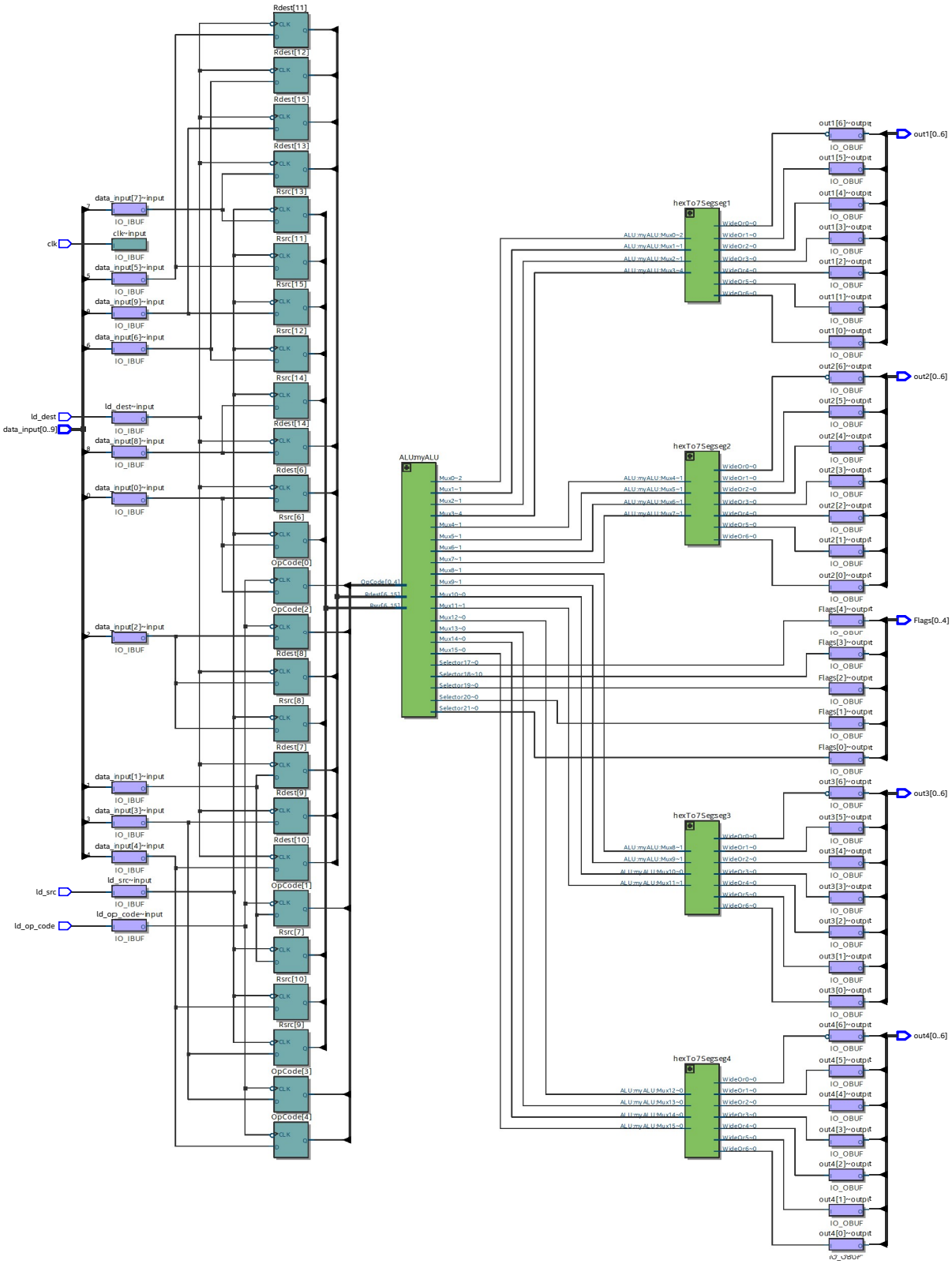



Equal0
Equal1
Equal2
always0~0
always0~1
always0~2
sel[1..0]
pc_en
clk
mem_addr[15..0]
rst
imm[15..0]
Add3
Add1
Add2
Add0
cnt~[15..0]
cnt~[31..16]
cnt~[47..32]
cnt[0]~reg[15..0]
cnt[15..0]


RAM_block:RAM1
addr_a[9..0]
addr_b[9..0]
clk
data_a[15..0]
data_b[15..0]
we_a
we_b
8:0 addr_a[8..0]
8:0 addr_b[8..0]
clk
data_a[15..0]
data_b[15..0]
we_a
we_b
q_a[15..0]
q_b[15..0]
RAM_block:RAM0
8:0 addr_a[8..0]
8:0 addr_b[8..0]
clk
data_a[15..0]
data_b[15..0]
we_a
we_b
q_a[15..0]
q_b[15..0]
16[9]
0
1
q_a_out~[15..0]
q_a_out[15..0]
16[9]
0
1
q_b_out~[15..0]
q_b_out[15..0]






Rdest[11]
Rdest[12]
Rdest[15]
Rdest[13]
Rsrc[13]
Rsrc[11]
Rsrc[15]
Rsrc[12]
Rsrc[14]
Rdest[14]
Rdest[6]
Rsrc[6]
OpCode[0]
OpCode[2]
Rdest[8]
Rsrc[8]
Rdest[7]
Rdest[9]
Rdest[10]
OpCode[1]
Rsrc[7]
Rsrc[10]
Rsrc[9]
OpCode[3]
OpCode[4]
data_input[7]~input
clk~input
data_input[5]~input
data_input[9]~input
data_input[6]~input
ld_dest~input
data_input[8]~input
data_input[0]~input
data_input[2]~input
data_input[1]~input
data_input[3]~input
data_input[4]~input
ld_src~input
ld_op_code~input
IO_IBUF
clk
ld_dest
data_input[0..9]
ld_src
ld_op_code
ALU:myALU
hexTo7Segseg1
hexTo7Segseg2
hexTo7Segseg3
hexTo7Segseg4
out1[6]~output
out1[5]~output
out1[4]~output
out1[3]~output
out1[2]~output
out1[1]~output
out1[0]~output
out1[0..6]
out2[6]~output
out2[5]~output
out2[4]~output
out2[3]~output
out2[2]~output
out2[1]~output
out2[0]~output
out2[0..6]
Flags[4]~output
Flags[3]~output
Flags[2]~output
Flags[1]~output
Flags[0]~output
Flags[0..4]
out3[6]~output
out3[5]~output
out3[4]~output
out3[3]~output
out3[2]~output
out3[1]~output
out3[0]~output
out3[0..6]
out4[6]~output
out4[5]~output
out4[4]~output
out4[3]~output
out4[2]~output
out4[1]~output
out4[0]~output
out4[0..6]
IO_OBUF