ECE 3710

FALL 2015

FINAL REPORT

TEAM: FREE BOARD

Max Hansen
Dusty Argyle
Reuben Fishback
Thu Hoang

I. Abstract

The duration of our design and implementation of the final project was split into two parts over the course of the semester. The first half of the semester was spent completing labs that would aid us in the design of our project. The labs included the following: the design of our ALU; integrating the ALU with a Register File; integrating memory with the CPU Data Path; CPU control; and interfacing the CPU with a specified peripheral. The second half of the semester was spent on implementing our design with our peripheral and building our game. The integration of the entire system included the following: building the assembler; building a module for the accelerometer; interface the CPU with VGA; and building the game components.

II. Introduction

The main objective was to build a snowboarding game. The user would stand on a makeshift snowboard to control the snowboarder glyph on the PC. Our system included the basic hardware components: an ALU, Register File, and a peripheral. Our chosen peripheral was an accelerometer and the VGA. Given the specific hardware and our chosen peripherals, we were able to build a snowboarding game.

The following report is organized as follows: a brief overview of our game design; the design of the ALU; integration of the ALU and Register File; integration of the memory and CPU data path; the controls; integration of the CPU with VGA and accelerometer. An in-depth description of the software will be explained. A brief description on each team member's contribution to the project will also be mentioned.

III. Free Board

Our game is modeled similarly to the windows 95 computer game "Ski Free." Our game features a snowboarder cruising down a hill and his goal is to avoid any obstacles. The obstacles in this game is everything but the snow. The game awarded on a point-base scale where the longer you are able to go without hitting any obstacles, the more points you accumulate. The accelerometer is placed onto a makeshift snowboard with a cylindrical fulcrum to allow the user to pivot the board to simulate the snowboarder glyph from moving left and right on the screen. This is a game where you play with your feet and gauge how well you can balance on the snowboard just like you would in real life.

IV. System Organization

A. ALU

The basic design behind the ALU was a MUX interface. The MUX interface sends the data to the correct processing unit. This was done by using a case statement that passed in the op-code as an argument. The case statement was the input in the module. The appropriate operation being performed was dependent on the value of the op-code. The flags, such as for overflow and comparison, are are set during this operation. However, it was decided that the design should preserve the value of the flags from operation-to-operation, unless the flag was explicitly set during the current operation. For example, if a ‘compare’ value was operated as the first instruction, then the compare flag will stay active until another ‘compare’ is done. This is done by sending the flags to a register file outside of the ALU, then feeding the output of the register back into the ALU.

B. Integrating the ALU with the Register File

To verify that our ALU was working correctly with the Register File, we performed an exhaustive test. This method was done by putting the ALU and register file modules into a “main” module. This module has inputs for reset, a clock, and a 17-bit vector called *state*. The idea of this is it emulates the same thing as a binary instruction. The output for this module was the result from the ALU.

X	XXXX	XXXX	XXXX	XXXX
SelImm	Op	SelReg1	SelReg2	DestReg

Bits in *state* bit vector

This state was then fed into the *main* module and the output was verified. This was an initial test performed on the test bench to check that the correct value was being sent from the register file to the ALU. Additionally, the test checked that the ALU was producing the correct result. Initially, the instructions tested only included a few such ADDs and SUBs. Once it was confirmed that the program was working, the instructions were expanded into a small mock program with multiple instructions of all types.

ALU OPCODE Table:

A table of values associated with their respective op-codes used in this program

OPCODE Binary	**OPCODE Decimal**	**Instruction**
0000	0	NO-OP
0001	1	ADD
0010	2	ADDU
0011	3	ADDC
0100	4	ADDCU
0101	5	SUB

0110	6	COMP
0111	7	COMPU
1000	8	AND
1001	9	OR
1010	10	XOR
1011	11	NOT
1100	12	LSHIFT
1101	13	RSHIFT
1110	14	ALSHIFT
1111	15	ARSHIFT

Flag Bits Table:

Bit	**4**	**3**	**2**	**1**	**0**
Flag	Output is Negative	input1 == input2	overflow	input1 > input2	carry

Input/ Output Table:

A Table of inputs and outputs for ALU, Register File, and Number Display modules

Module	**Input**	**Output**
ALU	[15:0] input1	[4:0] outFlags
	[15:0] input2	[15:0] result
	[3:0] op	
	[4:0] inFlags	
RegisterFile	clk	[15:0] out2
	reset	[15:0] out1

	[15:0] data_in	
	[4:0] enable	
	[3:0] select1	
	[3:0] select2	
num_display	clk	[7:0] led
	[15:0] value	[3:0] state

C. Integrating Memory with CPU Data Path

The memory was responsible for holding information that was split into three different parts. The first part was to write and read data values during program execution. The second part held data for the VGA display. Finally, The third task was to hold the program instructions.

In our final implementation, we instantiated two memory modules. The first module was for the data read/write and VGA, while the second will only hold the program instructions. In the first instantiation we will reserve port B for the VGA instructions while port A will control all of the data reads/writes. The second instantiation will hold all of our program instructions and will be accessed by the PC counter.

Our data path is the state of our control wires and ALU control lines. The instruction that is being executed and the PS of the control will determine the state of the data path lines. Controlling the data path with the same PS controller will ensure that the processor stays in sync. The register file will be connected to the ALU and main data bus. Memory can output to a register or have data from the registers be saved for later.

D. Controls

The control module was our finite state machine. A new instruction is brought into the Next State (NS) logic block. At the positive edge of the clock the instruction is passed to the FSM decoder that will set the data path to reflect the instruction. Inputs to the NS logic block include the instruction output from memory, the present state, and the ALU flags. Based on these input signals the NS logic determines the next state. On the clock cycle the NS becomes the Present State (PS).

The first two states for all instructions will be the same followed by variations of an execute step that will add one additional state except for the load instruction that will have two states for the execute stage. First an instruction will be fetched followed by a decode state. The type of instruction will be determined in the decode state. For R type instructions the registers will be activated and the ALU control lines will be set. For a store the write enable will be turned on while the register will be deactivated. A branch will update the PC with an offset and a jump will replace the PC with a value stored in one of the registers. For all instructions the last state of the execute stage will turn the PC_{en} high in preparation for the next fetch state.

V. Integrated Peripherals

A. Accelerometer

1. Design

This portion reads from, and writes to, the accelerometer sensor. This is done by using a serial peripheral interface (SPI). This was done by breaking it down into three different modules: serial clock generator, SPI interface, and a controller. The serial clock generates several signals. The first, and most important one is the actual 500 kHz serial clock that gets sent to the accelerometer. However, it only sends the clock when transmitting or receiving data, otherwise it holds the signal high. The other signals are a pulse that tells the SPI interface to read data from the serial data input (SDI), or to write to the serial data output (SDO). The final signal that gets generated is a pulse that is used as a counter in the SPI interface. The SPI interface module write data to the sensor and reads from it. This is done but putting the output data into a shift register, then every time a write pulse from the clock generator is received, data is shifted to the output. The same thing is done to read the data, except the data is shifted in. The final module is the controller. This sets the transmission data and send it to the SPI interface. It also reads in the data when appropriate (when the axis data is transmitted). Finally, if a "transmission done" signal is received from the SPI controller, it enters an "idle" state. It delays for about 1 millisecond before reading the data back in. After the delay period finished, it will execute the next state (changes the transmission data).

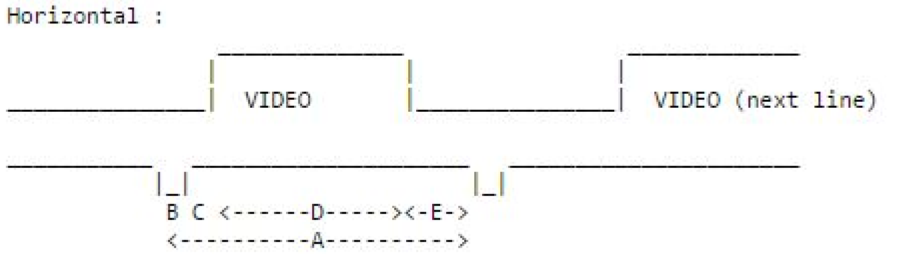
2. Debugging

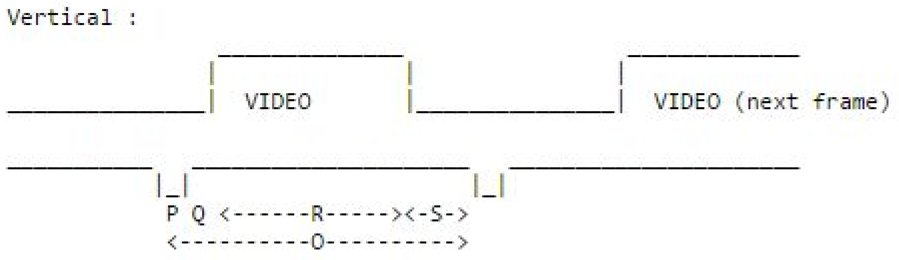
This module took a bit to debug. The most straightforward way to debug it without an oscilloscope was to hook the data lines to the eight LEDs on the Nexys 3 board. This was done in order to make sure that the signals were being read and sent at the appropriate times. This was all lining up and the timing seemed to be good, but it was still not receiving the correct data. This was testing by reading out the ID of the device, which is constant. However, the data that was being received was not correctly timed. When nothing was working, an oscilloscope was used to check the signals and confirm that they were being correctly transmitted. As it all turned out, it was because an extra two bits were being sent with the address. It should have been sent as two control bits, and six address bits. It was originally sent as 2 control bits and eight address bits.

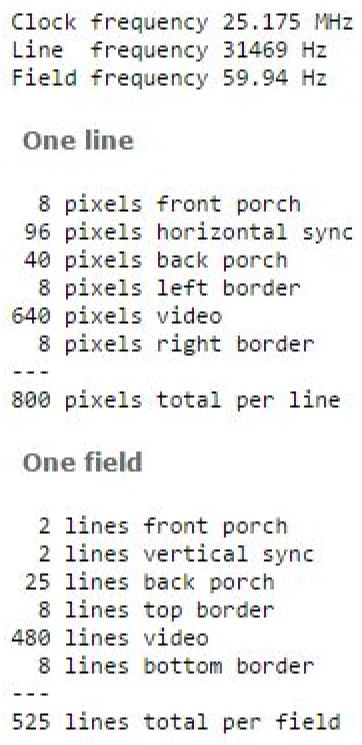
B. VGA

1. VGA Control Logic

VGA output is controlled by a horizontal sync and vertical sync pulse. These signals are active low and are based on the old tube based display. Tube displays used an electron beam that would rastar across the screen, the horizontal sync pulse moves the beam back across the screen while the vertical sync moves the beam from the bottom to the top. In our design we went with a 640X480 screen resolution with a refresh rate of about 60 Hz. A pixel clock will be used to know how long the beam is on each pixel, for our resolution and refresh rate we need a pixel clock of 25 MHz or ¼ of our on board clock. The duration of our front porch and back porch are as described in the table. Our VGA controller also kept track of our vertical pixel and horizontal pixel to assist in knowing what color to paint at that point.







```
Clock frequency 25.175 MHz
Line  frequency 31469 Hz
Field frequency 59.94 Hz
```

One line

```
  8 pixels front porch
 96 pixels horizontal sync
 40 pixels back porch
  8 pixels left border
640 pixels video
  8 pixels right border
---
800 pixels total per line
```

One field

```
  2 lines front porch
  2 lines vertical sync
 25 lines back porch
  8 lines top border
480 lines video
  8 lines bottom border
---
525 lines total per field
```

2. VGA Memory

With the display working we needed a way to paint our glyphs to the screen. We implemented a frame buffer and glyph library that will be described in a later section.

Memory was divided into four sections: instructions, data, frame buffer and the glyph library.

Purpose	Space Needed	Addresses
Instructions	2000 Lines	0-1999
Data	1000 Lines	2000-2999
Frame Buffer	4800 Lines	8192-14335
Glyph Library	2048 Lines	14336-16384

The frame buffer needed a minimum of 4800 lines because of the 80x60 resolution we were shooting for. Our glyph library was 2048 because we have four different images each with 16 individual glyphs. Each glyph was 8x8 pixels. Our memory addresses are 15 bits. The frame buffer and glyph library both needed an offset in order to be accessed. The frame buffer offset was 0b01 and our glyph library offset was 0b0111, that is why our frame buffer and glyph library start at memory location 8192 and 14336 respectively. The memory addresses became a concatenation of different values.

Frame Buffer Address

Offest		V-pixel[8:3]							H-Pixel[9-3]					
14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Glyph Libary Address

Offset				Data From Frame Buffer						V-Pixel[2:0]			H-Pixel[2:1]	
14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Our last horizontal pixel bit was not used in the address but it was used to mux between the first or second part of the data coming from the glyph library. Each chunk of data coming from memory is 16 bits and we used 8 bit color pallet. We were able to fit two pixels per memory location.

3. Frame Buffer

The frame buffer layout was sectioned into 8x8 pixel glyphs resulting in a screen size of 60x80 glyphs. Figure 1 below shows a visual representation of the frame buffer map depicting or introduction screen. The process was done in Excel where each cell shows a glyph code. The glyph code is highlighted in specific colors to help visually identify which glyph is being referenced. The frame buffer locations is then handled with software to correctly identify the position on the screen to draw the glyph. The frame buffer locations are also used to draw the map (or the course in the game) depicted in Figure 2.

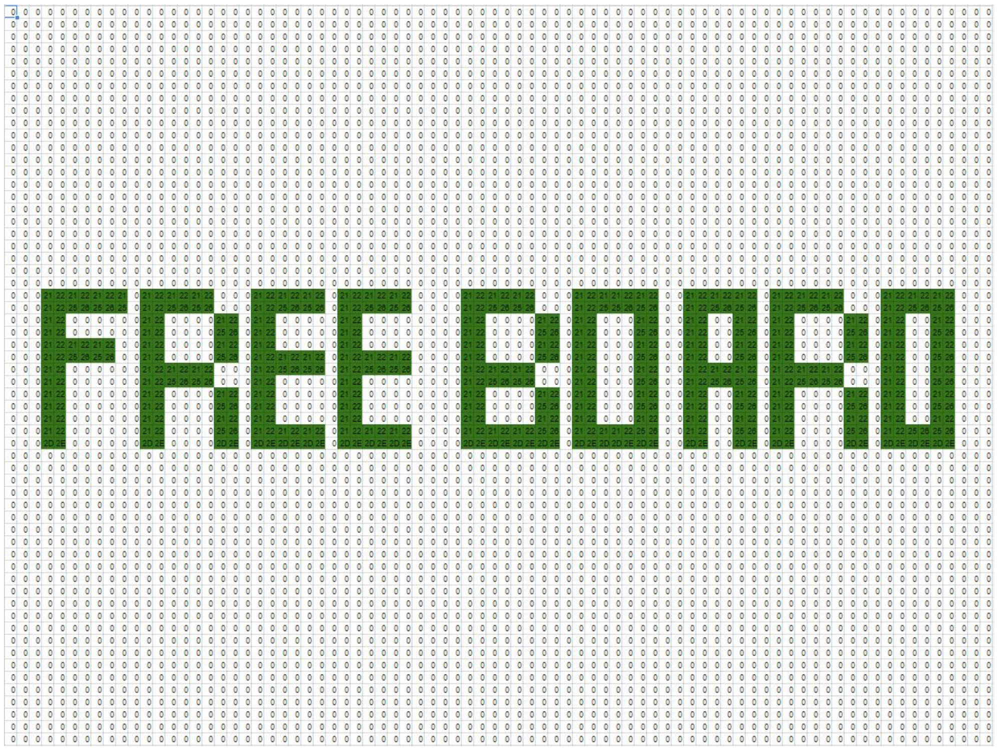


Fig 1. Intro Screen for Frame Buffer

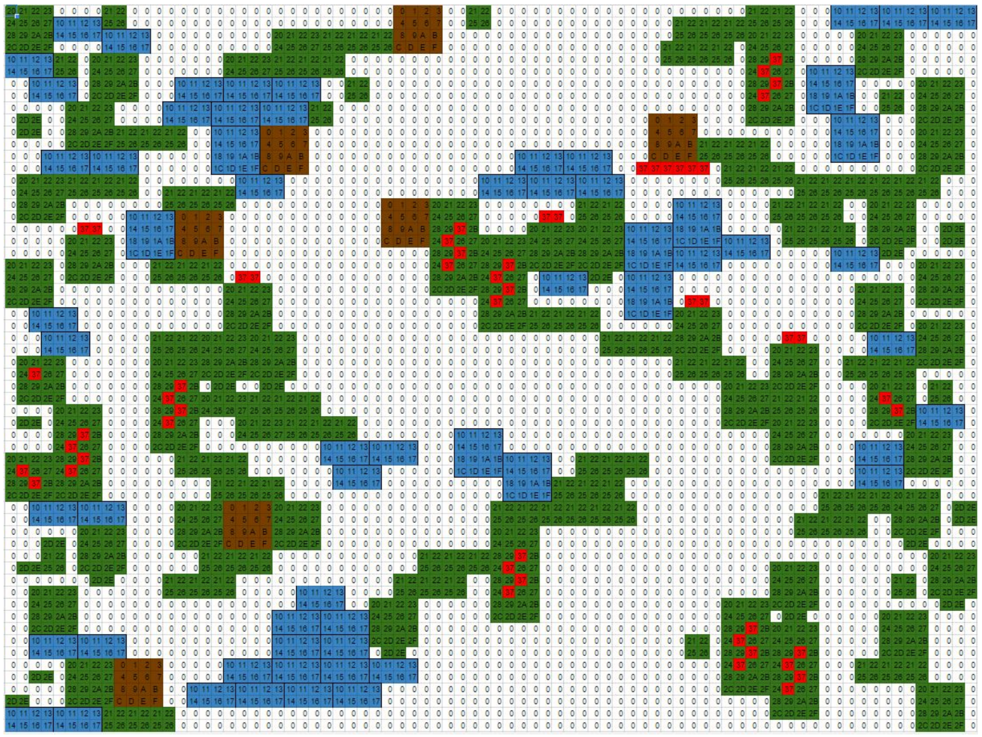


Fig 2. Glyph Mapping for Frame Buffe**r**

4. Glyphs

The frame buffer maps each location to a glyph code stored in the glyph library. Each glyph was an 8x8 pixel and each character glyph depicted in Figure 3 is a collection of glyphs. Each collection of glyphs was 4x4 glyphs resulting in a 32x32 pixel character glyph.

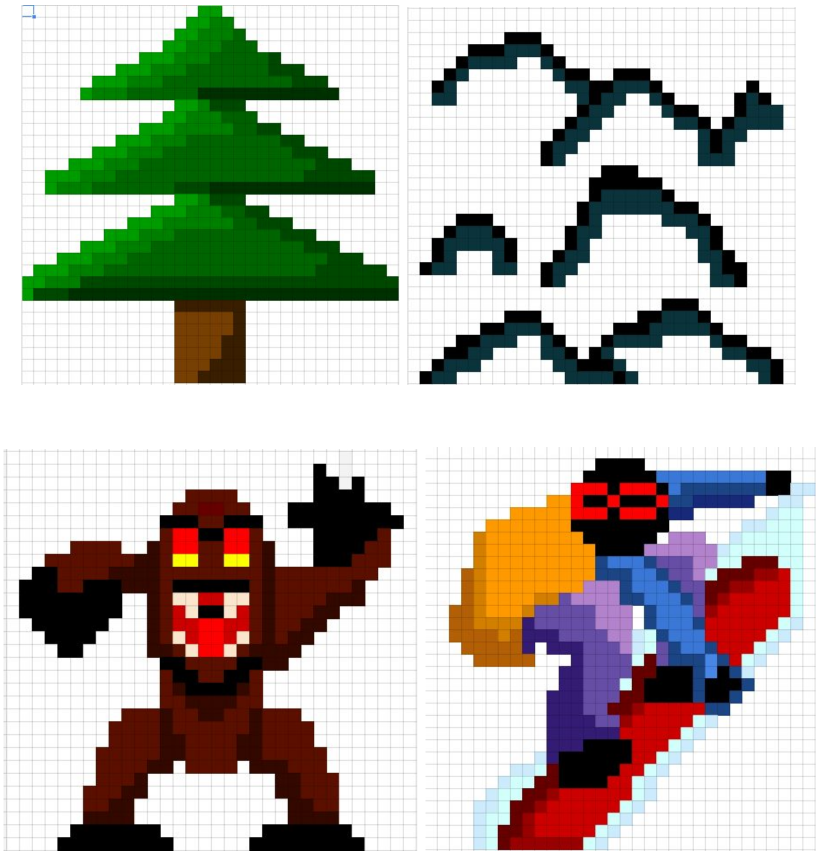


Fig 3. Character Glyphs

VI. Software

A. Assembler

1. Info

For the assembler, the decision was made to make a 2 panel GUI that would accept instructions on the left panel. It would then have a button that would initiate the compiling of the code. The application was written in C# using Visual Studio's WPF framework. The assembler ended up requiring that the output be hexadecimal. In order to satisfy this, a checkbox was created to determine if the output panel on the right would be in binary or hexadecimal.

2. **Debugging**

Some issues to consider are the compiler's ability and the CPU's interpretation. Since the CPU is more difficult to change, It is a lot easier to change the assembler's interpretation to match the current configuration of the CPU. It is also beneficial to work as a pair with the person in charge of testing and modifying the CPU.

B. Testing

1. **ALU operations**

This is very important that you check the compiled code with the ALU's operations. Understanding how the compares and branches really work is key in preventing bugs in your program.

2. **Frame-buffer Locations**

Key memory locations need to be tested to make sure that when you are storing glyph codes into memory locations, that they are actually where they should be. To test this, the seven segment display was used to insure proper values were being targeted and some visual confirmation in the frame buffer was used to positively identify the correct locations.

C. Frame Management

1. **Glyph Codes**

In order to manage the glyphs shown on the screen, codes identifying the glyph would be stored into the frame-buffer. The codes would start from 0 and go to 64. Each complete figure would be 16 glyphs. Debugging this was fairly simple, The only thing tricky about drawing the glyph is that the glyphs needed to map to the

2. **Frame-buffer Layout**

For the frame-buffer layout, a visible boundary in memory was used. There are 4800 visible locations on the screen. The frame-buffer was extended past this point to keep a rolling map for the game. The total frame-buffer is then 2 total frame buffers or 9600 memory locations. Debugging this aspect was done by writing known glyph codes into certain locations to verify key points in the process such as the start and end of the visible frame-buffer. The extended frame buffer was tested after implementing the infinite scroll capabilities.

3. **Infinite Scroll**

In order to make the screen scroll infinitely, the screen is recycled from the top of the visible frame-buffer to the top of the extended frame-buffer. Every 4 moves. A move just moves 1 glyph down on the full frame-buffer. This is executed by loading the glyph directly above or 1 full row back from the current location and storing it in the current location. This gives the appearance of movement down a hill.

4. **Debugging**

Debugging this issue required that I was certain that the key locations of the total frame-buffer were correct. Once this is done the scrolling was checked by placing the already created map into the invisible frame buffer. If you loop the move logic the map should pass through the visible frame buffer and display. The final debugging step is the infinite part of the scroll. This is done by loading in the top 4 rows of glyphs and storing them in the top 4 rows of glyphs in the extended frame-buffer. This is done this way so that the map is repeated with complete figures and not just broken glyphs throughout.

D. The Program Steps

1. **Setup**

The setup requires that appropriate values be stored and available. So initially, the key points like the top and bottom of the frame-buffer are saved into registers permanently. This is because it is hard to keep it around as an immediate value due to the immediate instruction constraints.

2. **Movement**

In the main executing part of the program, the movement is the first executing step. This is the scrolling appearance that was previously discussed in the section, C-3: Infinite Scroll. Later on in the writing of the program, it was decided that the infinite scroll should be split between the movement and the recycling of the pages.

3. **Removing The Previously Drawn character**

The previous character needed to be removed first in order to correctly draw the character again. This was done by using saved location of the character, moving down 1 row of glyphs to compensate for the movement, then storing 0 (the snow glyph) into all of the locations the character would be located at.

4. **Accelerometer**

The accelerometer was made simple enough to interpret that we would simply check for positive and negative values from it to move left and right. The accelerometer controller was made smart enough to write into a register and the software would simply read that register to get the accelerometer input. It was decided that on the left or right moves it would move 3 glyph positions. There was also a threshold established that would allow the player to go straight without the accelerometer being completely flat. The movement was added to the character position.

5. **Verify Position**
   This step is fairly self explanatory. All that is required is that the character position be checked to make sure that the character remains in the same row and in bounds. To do this, we set a left and right boundary calculated at the same row that the initial character position would be on. This was checked with these boundaries and set to the boundary exceeded boundary if the check failed.
6. **Drawing The Character With Collision Detection**
   First is checking the character's current position. The top left part of the character was the marked position. This was moved 6 rows of glyphs up so that the character did not sit on the bottom of the screen and the entire character would sit 2 rows of glyphs up from the bottom. So the entire character was drawn from that position. At each glyph that will be drawn, the position of the glyph needs to be loaded in and checked if it is not snow. If it is not snow, the score is reset to 0.
7. **Add to The Score And Iterate**
   In order to recycle the pages properly, every 4 moves (an entire model) of the infinite scroll, the 4 top most visible rows of the frame-buffer needed to be loaded and stored into the top most 4 rows of the extended frame-buffer. An iterator is required to make sure that 4 moves have been performed. This is the point where the iterator and the score both get 1 added to them.
8. **Recycling The Frame**
   When the iterator has reached 4, set it to zero and begin this step. Otherwise, skip this step and jump to step 1 to restart the program execution.

VII. Overall System Integration

A. State Machine

In order to for all of the modules to work together as a full processor, a state machine was used. A simple state machine block can be seen in Figure 4 below. This controls the various signals going to the different modules. For instance, it can tell the register when to write to a register. Our hardware integration was as simple as it could be. It consists of all of the modules being wired together. They were all designed to work together in order to make system integration go as smoothly as possible. Our finalized processor design can be seen in Figure 6 below in the Results section. The only module that has a unique functionality is the accelerometer. One of the outputs of the accelerometer module says whether or not new data is available. This signal is tied to the register file. When it goes high, the new data is written to register zero.

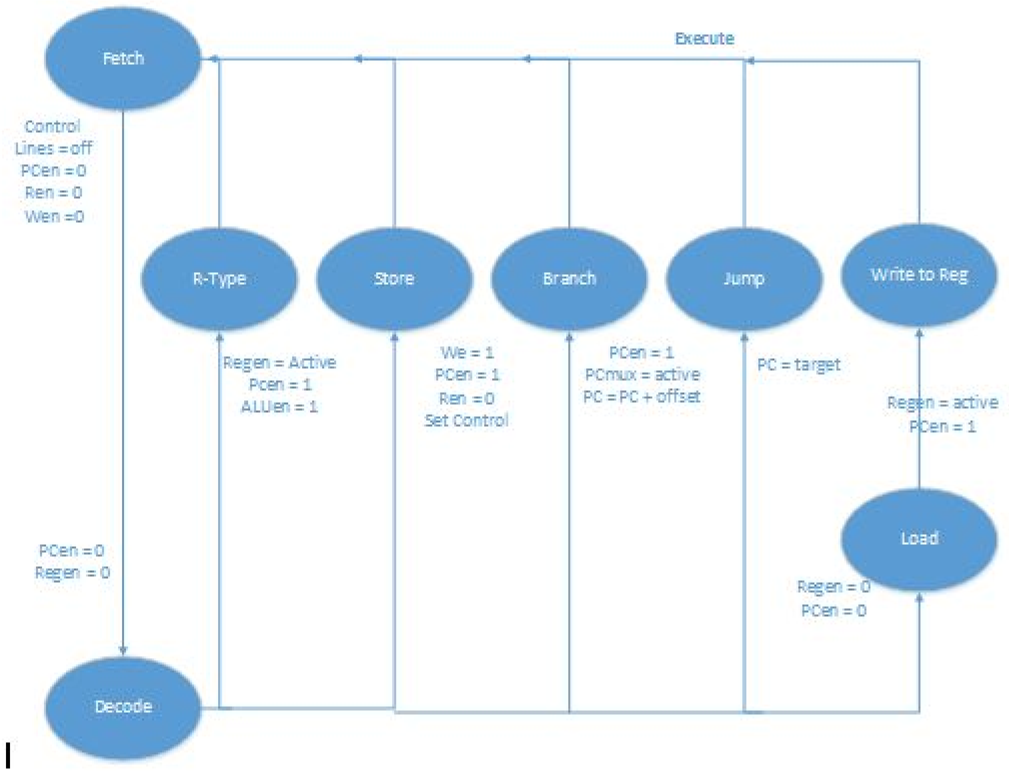


Fig 4. State Diagram

VIII. Team Members

A. Max Hansen

Max's contribution to the project included building and testing the ALU, Register File, Instruction Control, and the Accelerometer peripheral. He was able to integrate all the hardware together.

B. Dusty Argyle

Dusty's contribution to the project included building and testing the software and helped build the Memory with Reuben. He was able to create and build our game with his software design.

C. Reuben Fishback

Reuben's contribution to the project included building and testing the VGA peripheral and helped build the Memory with Dusty. He was able to integrate our hardware with the VGA and output visuals on the computer screen.

D. Thu Hoang

Thu's contribution to the project included creating the glyphs, glyph codes/library, frame buffer and helped Reuben with the VGA peripheral. She was able to come up with a system to help speed up the process of creating the glyph, glyph codes/library, and frame buffer.

IX. Results

We had some minor setbacks with getting our system integrated. For instance, or initial design shown in Figure 6 didn't work because our hardware was modeled after a combinational logic design. We finalized our design shown in Figure 6 with additional peripherals.

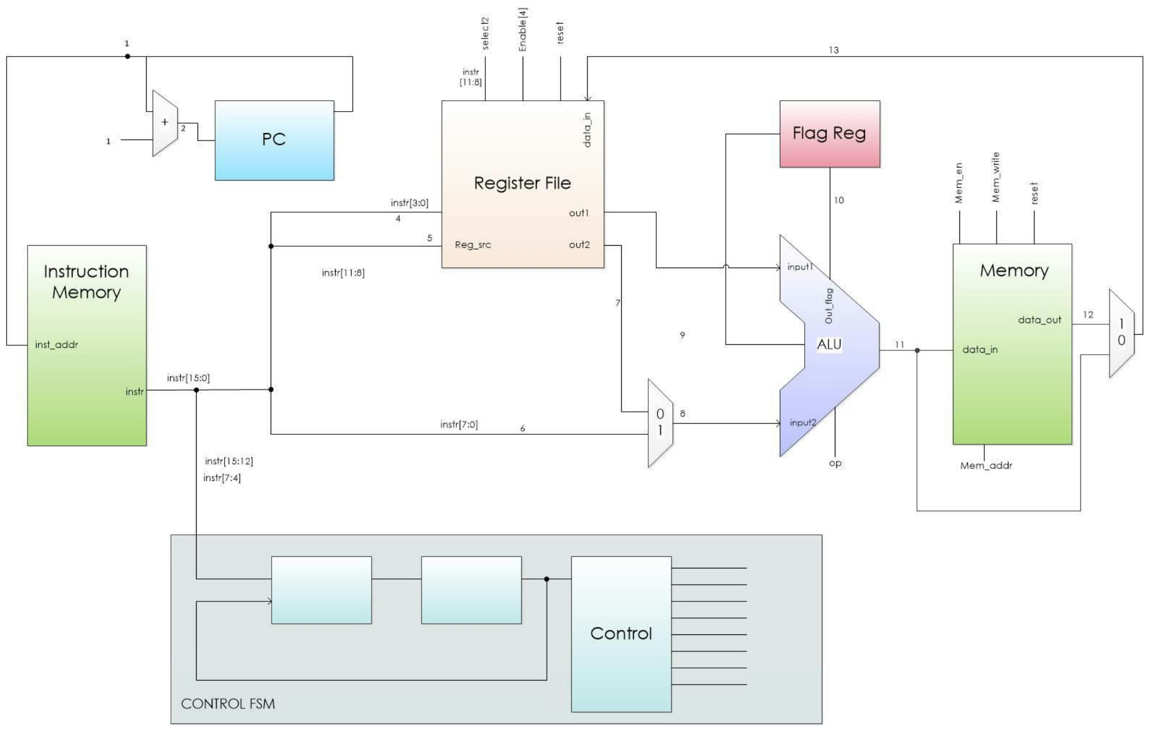


Fig.5 Previous Processor Design

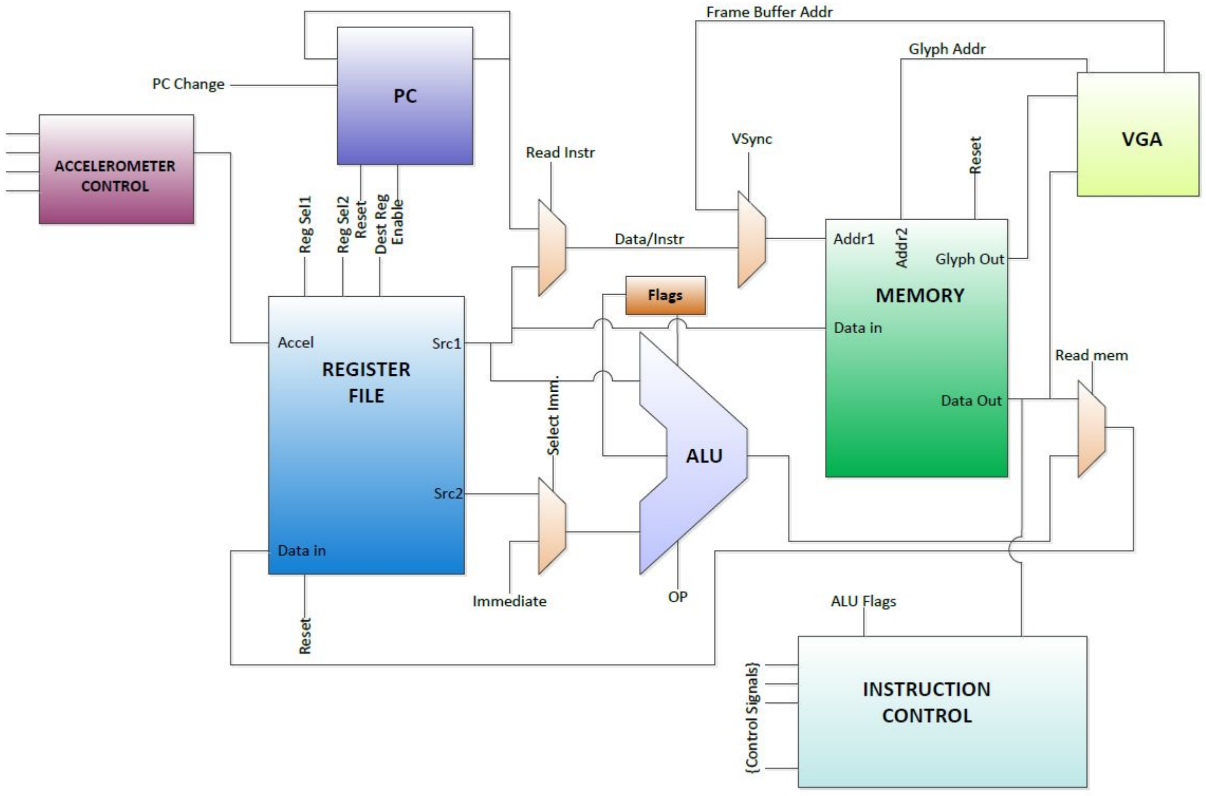


Fig 6. Finalized Processor Design

X. Conclusion

The hardware done as labs as a class included the following: the design of our ALU; integrating the ALU with a Register File; integrating memory with the CPU Data Path; and CPU control. From then on, our group worked on our individual project. We needed to complete a few more tasks before integrating our whole system together. first we needed to interface the CPU with our VGA and the Accelerometer. Once the peripheral hardware aspects have been added, the system could be tied together by software. The software was used to build and create our snowboarding game.