Knights of the Square Table Group documentation

Salwa Bukhari, Zahid Safi, Ryan Lukas, Esther Kim

CS/EE 3710 - Computer Design Lab Project Dec, 2019

1- Application:

The application of the design is an interesting part since it is the part where our project shine and narrow down the game instructions and how we should play it.

Our application is written in CR16.
We have it in a way that it starts by loading two memory location addresses from the keyboard, one for the current location and one for the destination piece.
After that, it keeps track of the two pieces inputted by the user to later on swap them.
It first swaps the two pieces then make sure the VGA redraw it.
For piece capture, we have it so that it will check the current piece's color, since we gave each piece a unique number, (see the table below), if the current piece was white for example, it will check if the destination piece color is black, it will overwrite it then draw a box in that location, otherwise jump to user input.
As soon as the user input the 4th key which is the last thing to determine where the piece should go, the piece will immediately move to the new location.

Below is the pseudo code of the application functionality.

A. Initialize/setup
- VGA will draw the 8x8 board.
- Store all possible glyphs in memory.
- Display glyphs at the start of the game.

B. User input
- Have the user select the chess piece;
- If (pawn) >> J pawn.
- If (bishop) >> J bishop.
- If (king) >> J king.
- If (rook) >> J rook.
- If (knight) >> J knight.
- If (queen) >> J queen.

C. Pawn
- Get user input, for the desired move.
- Have the VGA redraw the glyph at +1 and erase current position.
- J user input.

D. Bishop
- Get user input.
- Have the VGA redraw the diagonal square and erase the current.
- J user input.

E. King
- Get the user input.
- Have VGA redraw at any position +1 from current.
- J user input.

F. Rook
- Get user input.
- Have VGA redraw at any horizontal or vertical location from current.
- Erase the rook from the current.
- J user input.

G. Knight
- Get user input.
- Have the VGA redraw the knight at any “L” shaped location.
- Erase the glyph at current.
- J user input.

H. Queen
- Get the user input.
- Have the VGA redraw the queen at any horizontal or vertical or diagonal locations.
- Erase the current location.
- J user input.

Piece	Unique value
Black pawn	1
Black rook	2
Black knight	3
Black bishop	4
Black queen	5
Black king	6

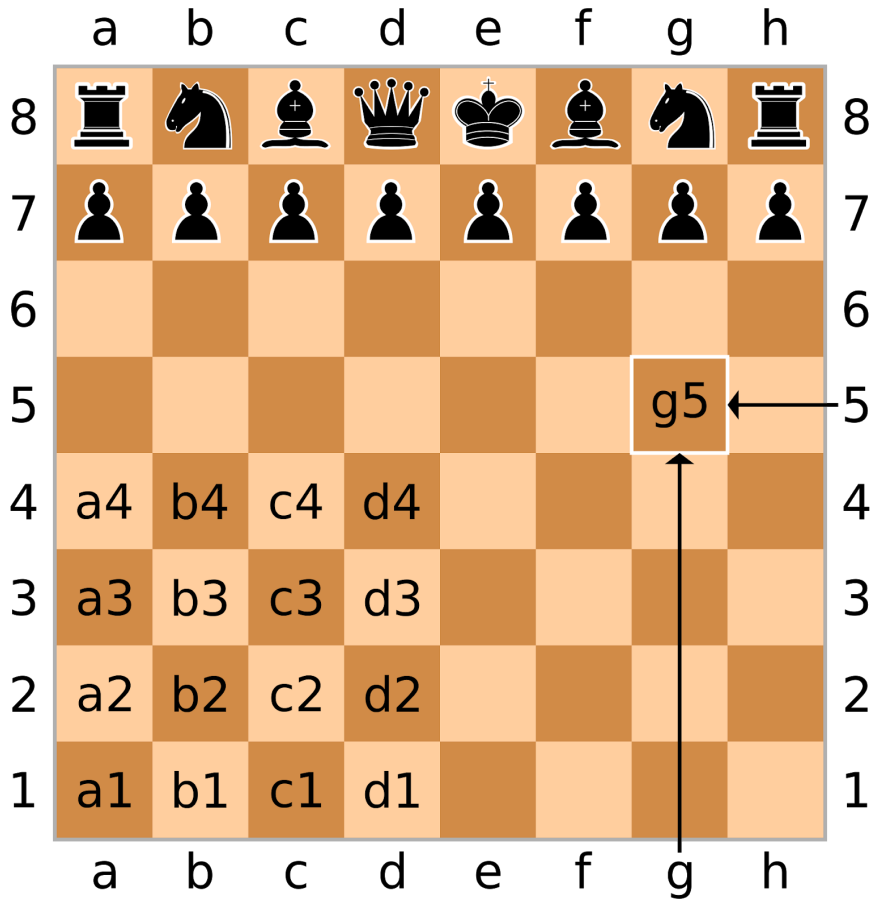
White pawn	129
White rook	130
White knight	131
White bishop	132
White queen	133
White king	134

Table. Unique values for the pieces.

Application's flowchart.

Application's user guide:

- For determining which piece you want to move, input the location of that piece, the letter followed by the number, after that input the remaining two keys for the designated destination you want to move the piece to.
- Note that after the fourth key pressed the piece will immediately move.
- After any movement, make sure to hit BACKSPACE to be able to clear what you have and then you can make another move.
- Below is a picture that clarifies the locations on the board.



2- Assembler:

The assembler is the translator from instructions in CR16 to the hex values machine language. We are writing the assembler code in C#.

A broad explanation for how the assembler works:

Initially we have our code use a streamReadear to read the file that has the instructions token by token and determine the type of instruction, register and the immediate values as well. After that according to the instructions we set our opcode and opcodeEx, to then convert it to binary and lastly to hex. And then write it to a text file in the hex form.

And to go more in depth about the assembler, as we have mentioned above the code will use the Stream reader class to read the file and it will read it line by line and during this process it will look for the labels in the lines and it will store those label and the address associated with it inside of a dictionary so that we can have easier access later, this will be the first pass on the assembler.

Next will be the second pass on the assembler in which this time we will read each instruction line and then split it up into tokens, separating the registers and the Operation labels. If reading the token, it comes across lets say ADD or SUB or any other operations, then it will check in a method that has multiple if statements for all the different operations and in it, once we find the correct one it will set the opcode, op code extensions, and condition codes as well. When it comes across a register, so for $15 for example, what we did is we made it so that we had a Boolean so that we knew if it was a Rdest or Rsrc, and then once knowing that information, we would remove the character “$” from the register and then

just convert the 15 to binary. But if we did come across an immediate value we would try to parse the value to an integer, and then check whether it was a positive or a negative number, if it was positive number then we would sign extend if needed the 8 bit value with zeros, and do the opposite for negative numbers in which we would sign extend with ones.

Once it has read through the instruction line and gathered up all the binary codes, for the opcode, opcode extensions, cond, Rdest, Rsrc and immediate. We will then have a set of If statements, that will concatenate all those correct binary numbers for the correct instruction based on the format, for example, ADD and ADDI have different instruction formats, so we would have to concatenate ADD like this: opcode + Rdest+opcodeExt+Rsrc. While the ADDI will be concatenated like so: opcode + Rdest+immediate. once we have put the binary values together, we then will convert the 16 bit binary number to an integer value, then convert that integer to hex using toString("X"). thus giving us the hex instruction and then we will store that within another data structure.

Lastly, once we have the all the instructions converted, we will then have another method that will call the Stream writer class in which it will write all the hex instructions to a text file, line by line so that the instruction memory in the processor can read it.

Assembler's user guide:

-To utlize our assembler, what you will need to do is install visual studios since have done it in c# and this can be the free version which is the community version.

-Once visual studio has been installed and all the initial setup is done, you will need to create a new project that is a console application and within this code you can copy and paste the AssemblerApplication code

-along with that class, you will add another class to the same project that will be a console library with .net standard, and within this file you can copy and paste the contents of the assembler.cs file

- now to use the assembler you just need to follow this file path of: source/repos/(name of your project)/(name of your console application)/bin/Debug and then within this folder is where you can store the instruction text files, and in doing so would allow the program to read the file and write to a text file that you can specify within the WriteHexInstruciton method

-to change what file to read, in the console application you will just need to change the name of the file within the new assembler and it will read the correct one

-for features, the assembler supports comments and commenting out code as well as it will throw exceptions for branches greater than 128 or less than -128.

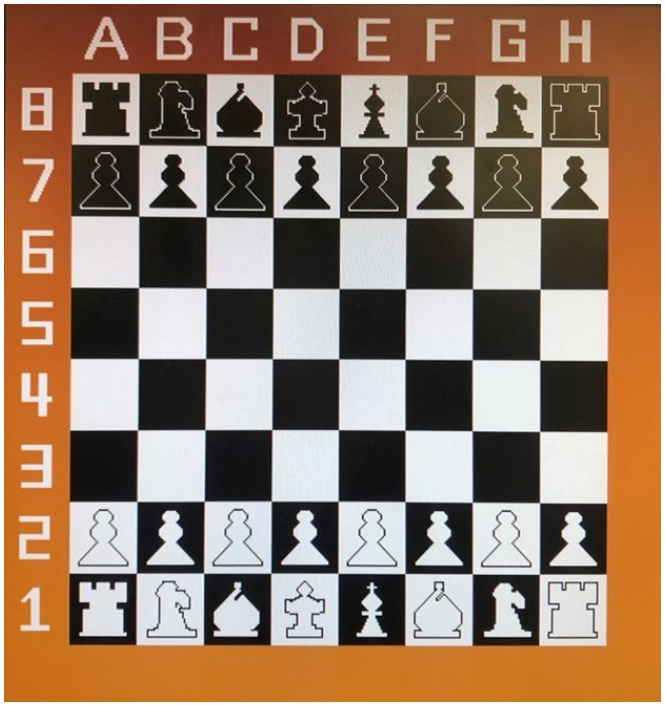
3- I/O:

The user will be allowed to use the keyboard as the input for our project, and the VGA will be displaying the 8x8 chess board. The keyboard was structured to take in a clock, data, and output its data through a LED. The arrow key on the keyboard was assigned a unique 8-bit scan codes, which then sent the code to the LED when pressed by the user. This provided information on what key was pressed through the LEDs. While building the keyboard as the users input, we had a problem in that the clock cycle was going too fast moving the LED too quickly. In order to resolve this problem, we included two

registers, one being the current data register and one for the previous data. This allows the previous data to hold the code and delay the LED movement.

For our design, in the main memory, all the pieces are being stored in the I/O memory as a default meaning that in the I/O memory we will be storing certain addresses for each of the piece which we have designated in the GLYPH ROM, for example, a four bit address of 4'd0 will be a black square with a white border and the 4'd1 will be a white square with a black border and the 4'd2 will be a white key with a black square and white border, with the borders being thing mainly because we will be on a black screen and it is just for aesthetic purposes. And the part the VGA controller has in this is that it will be controller the hcount and vcount like how we had it in lab 3 but it will go through the I/O memory and loop through each line and pull in address and then it will take in the address to the VGA controller and then the VGA controller will output the addr as a glyph select for the glyph rom, which if you take a look at the glyph rom module, we made it so that it is a 40x40 glyph. So the glyph select will take in the addr from the VGA controller letting the glyph rom know which piece should be displayed at the given time, and since the VGA goes from left to right, row by row, the glyph rom also takes in another input called row which will let the rom know which row it should be currently on and then on that row it will output a 40 bit value.

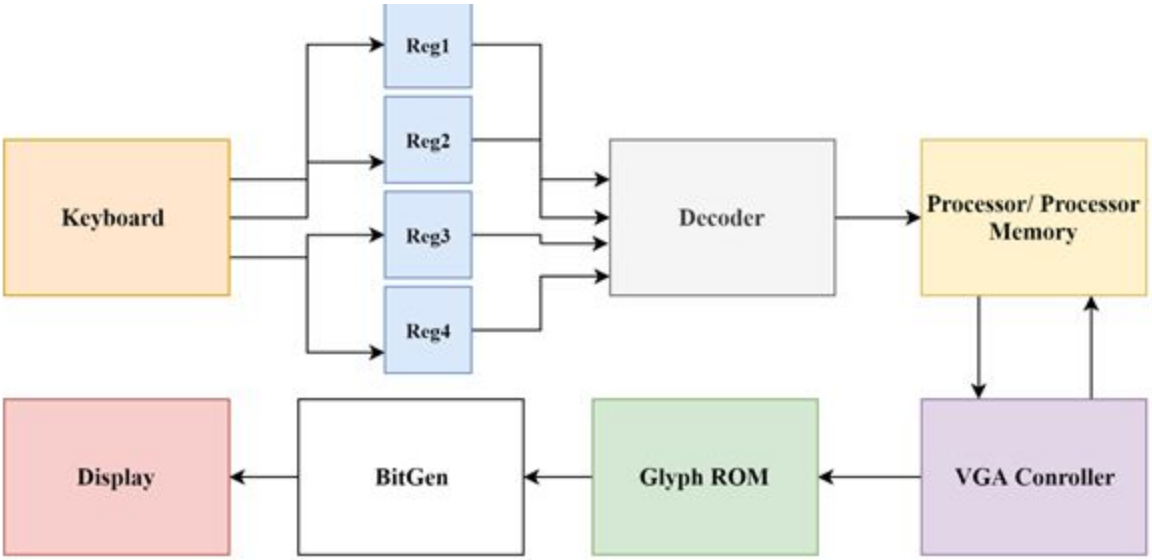
That 40-bit value will then be a combination of ones and zeros since we are only working with black and white. So that 40-bit value will then go the bit gen which inside it will take in the glyph data and it will read the 40-bit value, but having an internal loop caused problems with the display causing the images to move around and not be still, so to iterate through the 40-bit glyph data, what we tried to do was have it take in the hcount as an input so that it will be able to iterate through the data and since hcount is bit by bit and so is the bit gen, we figured that would be the best way to do so.
Then we had inside a bit gen an if state where if the value we come across is a 1 then we will display the RGB value to white and then if the value we come across in the glyph data is a 0 then we set the RGB value to white which in doing so allowed us to get the glyphs on the screen as such:



As you can see, we are able to display the glyphs in there correct format, and we did this by initializing the memory of the processor with the piece values and so like we explained before the VGA will simply read those values and with the hcount and vcount will display the glyphs in the correct locations, so for example the first 16 spots in memory will be the black pieces in order as you can see above and then for the last 16 spots it will be similar but it will be the black pieces and the glyph rom will be able to distinguish between which piece it is and what the square it is currently on. Along with the chess board we have also displayed a brown border in order to give it a more aesthetic feel to it and make it look more appealing and with the chess board border we have numbers from 1-8 on the left hand side and then letters A-H on the top side and this more so that the players will be able to look at the screen and make out the controls and move that they want to make to move a piece.

So, with the keyboard reading working we did have to make some tweaks to it for it to further work with the design that we are accomplishing. Since we are able to read one key at a time, our goal is to get the keyboard to read a keyboard input like: A2A3 which in turn would allow the pawn at A2 to move to spot A3. So in order to accomplish this we had to setup a counter within the Receiver and every time we received a key we would increment the count so that we would the number of keys we have and each time a key is pressed then they will be going into their designated register, so if A was pressed first then it would go into register 1 and then if 2 was pressed then it would go into reg2 and these are of course 8 bit registers that hold the hex values of the keys.

And within the keyboard we also added the ability to clear the registers when backspace is hit, so that would in turn clear out all the registers and set it back to zero. And then another key we added was once the enter key has been hit, it will send out all the keys that are in the registers and then it will output it to the decoder and within the decoder it will take in key combinations like A2A3, in there hex equivalent, and then it will read the key combinations then it will output the actual memory locations. So, for A2A3, it will output the current location of A2 as memory location 48 and then for A3 it will output it as 40. Then this will be sent to the processors memory so that the application will be able to pull from those memory locations and will use for example 48 and get the current piece in that spot and then get 40 and get the destination to move to and then it will store the value of the current in the destination, so the value at 48 will then be store at location 40 and in our current spot it will replace it with a zero which in doing so the VGA controller will take care of the rest and draw the pieces in there appropriate place and it will keep the pattern in order as well.



The I/O schematic.

4- Processor:

The processor is the main engine for the design, that takes care of the instructions and their functionality. Below is parts of the processor broken down in details.

For the very first step we had the ALU implemented in which we created an ALU control that took in the opcode and opcode extension and we would output our own opcode the ALU for the different instructions. We also created the regFile for all the registers and the register file which we can use for storage of values like ALU outputs, shift outputs, etc.

Then we added the data path to the processor, which for this we created our instruction register, Direct memory for the outputs, Pc counter, as well as the Muxes for the immediate and register values; muxes for the outputs of the shifter, ALU and Dmemory; and finally a mux for the input of what goes into the input of the regfile. We wired all these together with their control points to be set by the FSM depending on what instruction the FSM decodes.

For the FSM we drew out the diagram that has all the states and their signals. The FSM takes in the opcode, opcodeEx, and from reading the opcode extensions we will then be able to set the desired signals needed in order to perform the instruction required.

The process of building the FSM is to take into account the opcode and the opcodeEx, with having an initial state that sets all the signals to 0 except for the instructionEn is set to 1 so that the instruction register can store the instruction so that they can be used for decoding later. The next state will then take a look at the opcode which in some of the instructions when the opcode is 0000 then it is an Alu operation but doesn't have an immediate value, so all of those operations function the same the only difference is how the arithmetic is done and that is taken care of by the Alu itself. So once we know the opcode, the next state of the FSM will take care of the actual calculation of the processor and then in the

next state we will have it output so that it can be written into the Regfile in the next state after. Once the operation is done it will then go back to our initial state.

An important factor to take into consideration is if the opcode is not 0000 then we have to manually check the different opcodes, for the STOR, LOAD, Bcond, Jcond and JAL as they have to have different control points as they the branch and jump instructions use the PC counter but one displaces the value and the other one will change the pc counter to the value of the Rtarget. But the for the Alu operation the use immediate values, we decoded them but they end up going to the same set of states as they will be essentially functioning the same, only difference is we will enable the mux so that it can pick the value of the immediate instead of the register. And then the next states will take care of writing back to the reg files.

For the STOR and LOAD instruction we had it so for the Load, we had it so that it will take in the Raddress from the reg file, and then in the next state we will enable the memory and the regfile so that it can output from memory and then store it into the reg file. And then we will do the same for the STOR instruction but just the opposite as we will take the value from the reg file and then enable the memory by Dmemwrite in the next state so that we can write into it.

For Bcond, we just had the control point, displace, go into the PC counter and when it is enabled, we will add the displacement the current pc count so that we can get the address that we need to branch to. And then for the Jcond and JAL we will need to have the value of the A register, which contains the Rtarget, also go into the pc counter so that we can change the value of the count to the address needed, and this control point that will be enable is the jal control point.

We concluded that we will replace the tri-states to multiplexors, now we have the outputs of the memory, Alu and the shifter, go into a multiplexor and the select line from the FSM will decide which of the outputs is needed for the given instruction. So, using the multiplexor means that we will have a smaller number of control points, and we will just have to use different combinations for the select lines in order to get the desired outputs.

Another change we have is that we realized when we changed it to 32 that is unnecessary, therefore we changed it back to 16 bits because we can just sign extend the immediate value instead for the reg file, which what our earlier confusion was for and we felt the 16 bits was enough for our project and for what was needed.

Update : the processor works.

a) ISA:

For the ISA we are using the CR16 that was assigned to us and within this instruction set architecture, for our application what we ended up using the most were the Load and store operations as well as the some of the Jcond operations and this is mainly due to our application dealing with memory locations allowing us to move pieces by just swapping there locations in memory for them to be redrawn by the VGA. But nonetheless we did still implement the ALU operations along with Branches and Moves and shifts. And below you will be able to see that with each instructions they have a set of control points so when the given instruction is executed within the controller it will set these points so that data will flow in the correct way, so for the case of SUB, AregMux = 0 and BusMuxChooser = 10 and finalmux = 01 this will allow the processor to enable correct signals so that a subtract operation can be possible and for the processor to know it is subtract, is by the opcode and its extensions, from which the ALU control will

look to see what operation it has to enable. And this same process can be applied to all the instructions just with different control points.

instructions	Control points	OpCode
SUB	**AregMux = 0 busMuxChooser= 10 finalMux= 01**	**0000 1001**
SUBI	**AregMux = 1 busMuxChooser= 10 finalMux=01**	**1001**
ADD / ADDU	**AregMux = 0 busMuxChooser= 10 finalMux=01**	**0000 0101 / 0110**
ADDI/ ADDUI	**AregMux = 1 busMuxChooser= 10 finalMux=01**	**0101 / 0110**
MUL	**AregMux = 0 busMuxChooser= 10 finalMux=01**	**0000 1110**
MULI	**AregMux = 1 busMuxChooser= 10 finalMux=01**	**1110**
CMP	**AregMux = 0 busMuxChooser= 10 finalMux=01**	**0000 1011**
CMPI	**AregMux = 1 busMuxChooser= 10 finalMux=01**	**1011**
AND	**AregMux = 0 busMuxChooser= 10 finalMux=01**	**0000 0001**
ANDI	**AregMux = 1 busMuxChooser= 10 finalMux=01**	**0001**

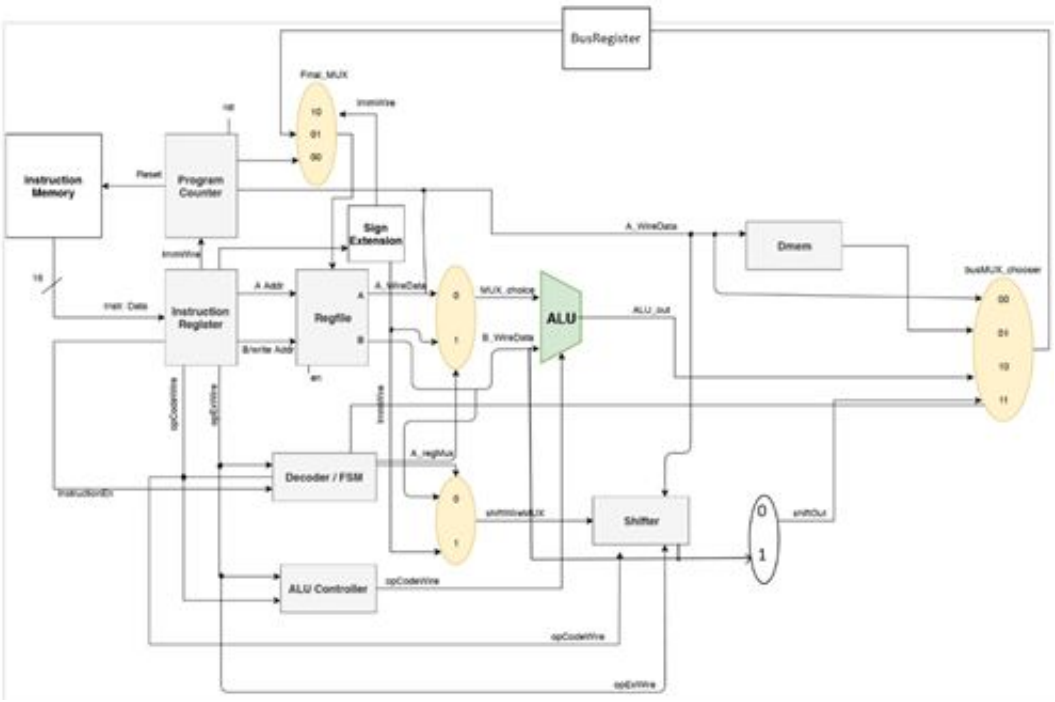
OR	**AregMux = 0 busMuxChooser= 10 finalMux=01**	**0000 0010**
ORI	**AregMux = 1 busMuxChooser= 10 finalMux=01**	**0010**
XOR	**AregMux = 0 busMuxChooser= 10 finalMux=01**	**0000 0011**
XORI	**AregMux = 1 busMuxChooser= 10 finalMux=01**	**0011**

STOR	**Dmemwrite = 1**	**0100**
LOAD	**changeValue =1 busMuxChooser=1 finalMux= 1 Regwrite = 1**	**0100**
JNE	**Replace = 1**	**0100/0001/1100**
JEQ	**Replace = 1**	**0100/ 000/1100**
JGT	**Replace = 1**	**0100/ 0110/1100**

JLT	**Replace =1**	**0100/ 1100/1100**

b) Data-path:

Our data path held all the different modules building the processor. In this, we had an instruction register. This register took the data from the main instruction register and help the values of the op-code, destination, immediate, etc. Then this was fed into the register file. This register had 16 different memory locations which held values to get manipulated or do calculations. Then we had an ALU and a controller that determined which operation would happen depending on the op-code. We also had a shifter within the data-path to be able to shift amounts if needed. Within the data-path, there was the Data memory where the final values will be stored. There was also 5 different muxes that chose which output we wanted. Lastly, the decoder was the FSM, controlling which control wires needed to be active high during each state. Below is a picture of the final data path.



A table of control points for the data path.

Control points	Functionality	When it will be set
InstrucitonEn	**Enables the instruction register**	**this signal will be enabled every time we are incrementing the program counter and reading the instructions from memory to store in the register.**
regwrite	**Enables the reg file so that it can take values**	**This signal will be set when ever something needs to be stored in the register or taken from the register**
Aregmux	**Select line that will make the mux decide between immediate or reg value**	**This select line will be enable if we use either an instruction with an immediate value or an**

		instruction that doesn't have one. For example: ADD or ADDI
AluTri	**Enables the tri state module that is connected to the output of the ALU**	**This signal will be enabled any time an ALU operation is needed to be completed that way the output can flow onto the bus to be stored.**
ShiftMux	**Select line that will make the mux decide between the an immediate or reg value**	**This select line will be enable if we use either a shift instruction with an immediate value or an instruction that doesn't have one. For example: LSH or LSHI**
shifttri	**Enables the tri state module that is connected to the output of the shifter**	**This signal will be enable any time a shift operation is needed to be performed/complemented and it will allow the output to go onto the bus to be stored.**
increment	**enable the PC counter to increment the count by 1 to go to the next line** **Every time it needs to get another set of instructions we will increment the PC counter by one.**	**it will be set in the controller/FSM and will be set every time we need to fetch a set of instructions**
DmemWrite	**is a signal that will enable the Dmem to write to and store values of our registers in there**	**It will be set every time we use the store operation.** **should only be enable for store operation**

DmemTri	**output of the Dmem to the regfile carried by bus**	**the tri will be for the load operation**
immjp	**it will set the pc count to the immediate address of a jump instruction when it is enabled**	**this signal will be set when we are using the jump instruction or for our case Jal**
jal	**will enable the tri state block for jal instructions so that we will be able to write to the pc counter**	**it is set when jal or Bcond is being performed**

c) Memory:

Our main memory held 256 different values. This was called Dmem which held the values of the final chest board. This memory was dual accessible but only the processor can write to it. It was dual accessible because the processor needed to access it and also the vga-controller. The vga-controller will request the value within the address and the memory will output that value back to the vga-controller.

d) Controller:

In this part, we are building the decoder that takes care of the instructions. It takes in the opcode, opcode extension, condition, clock, and reset. And it sets the signals in order to be written to the regfile. In the controller, we have a state for each instruction. The instruction mem reads a .dat file that has the hex operation for an instruction such as Addi, 4, $S1 >> which is adding the value 4 in the register $S1, and the assembler is converting to the instructions to the hex that will be written in .dat file being read by the instruction mem. That then is being passed to the controller to send the correct signal to the control lines to display the value to the register. In the memory map we need to be able to see the results in the specified register location.

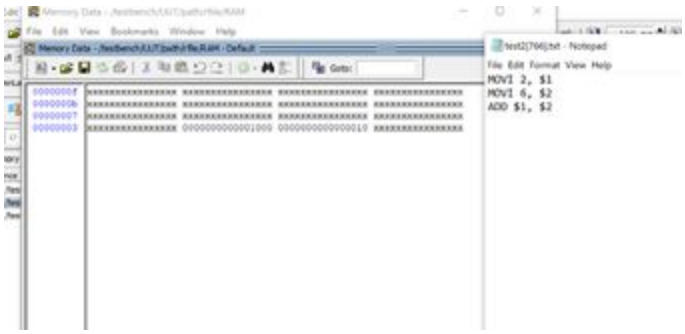
The structure for the controller is the following:

First step is the fetch, then we have the decode that determines what is the instruction.

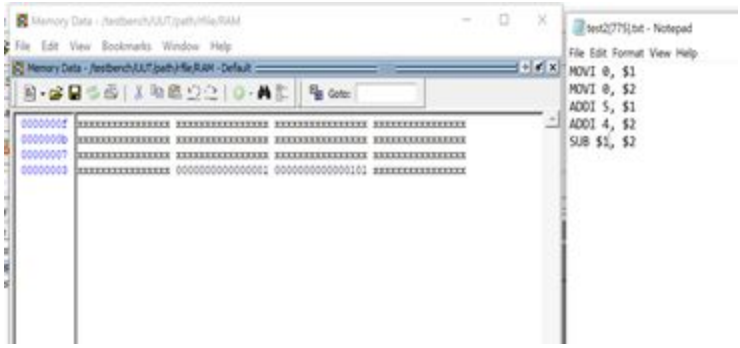
We have parameters for each of the instructions, and we are setting their functionality in each state using case statements inside the always block. We have two regs for current state and next state.

We had to add new intermediate state because after the value being outputted.

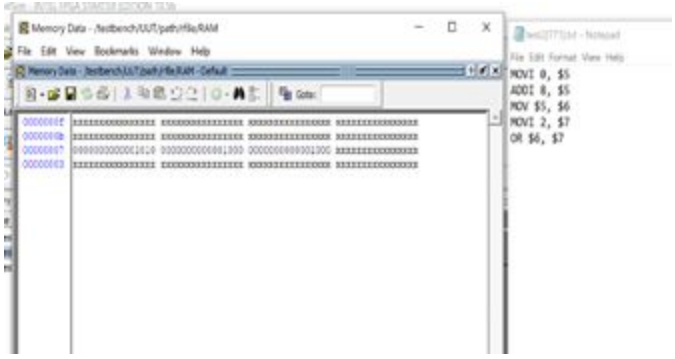
Below are some tests for instructions and their memory map.



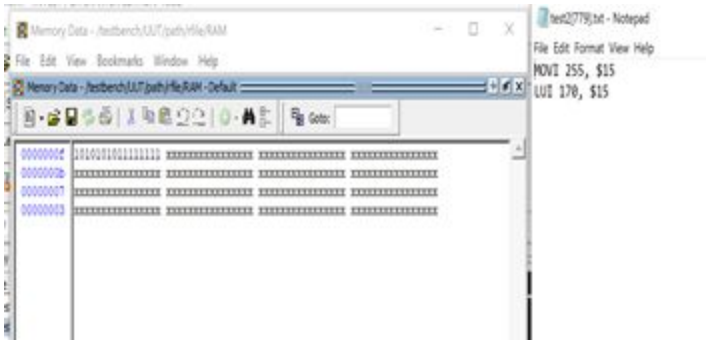
(Figure 4. Memory Map for simple ADD Instruction)



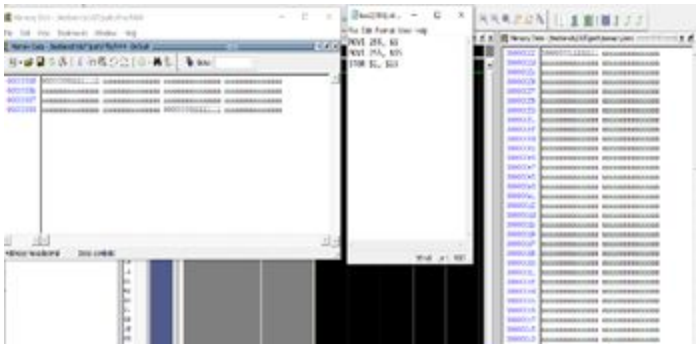
(Figure 5. Memory Map for simple SUB Instruction)



(Figure 6. Memory Map for OR Instruction)

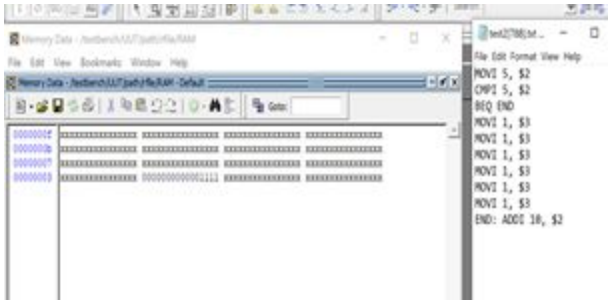


(Figure 7. Memory Map for LUI Instruction)



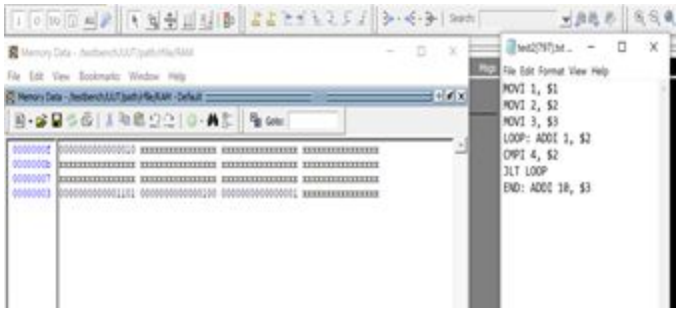
(Figure 8. Memory Map for STOR Instruction)

(Figure 9. Memory Map for AND Instruction)

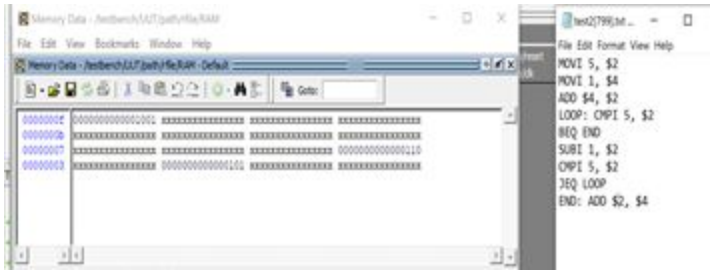


(Figure 10. Memory Map for BEQ Instruction)

(Figure 11. Memory Map for JEQ Instruction)



(Figure 12. Memory Map for JLT Instruction)



(Figure 13. Memory Map for J&Bco Instruction)

(Figure 14. Memory Map for JAL Instruction)