8-Bit Dreams and Linkers Schemes: An Odyssey Through the Creation of a MIPS Processor

3/14/2023

Authors:

- **Baibhav Barwal**
- **Janak Subedi**

Introduction:

The objective of this project is to design and implement an 8-bit (or larger) CPU architecture in Logism while applying knowledge of computer architecture. The project includes the creation of an assembler and linker that converts assembly code into hexadecimal machine language that runs on the CPU and can be loaded into the instruction memory of the CPU in Logisim. The project also requires the implementation of a single-cycle CPU with all components built in the lab, excluding the cache. The basic ALU is capable of addition, subtraction, and logical operations, support for both R-type and immediate instructions, a register file with a minimum of 8 registers, instruction memory, program counter, and data memory are also required. Additionally, a "kitchen sink" processor is required with various features, such as branch-if-equal, jump, support for SLT/SGT, carry look-ahead adder, hand-built multiplier circuit, support for function calls, pipelined architecture, cache, support for the stack, and a fancy assembler/linker.

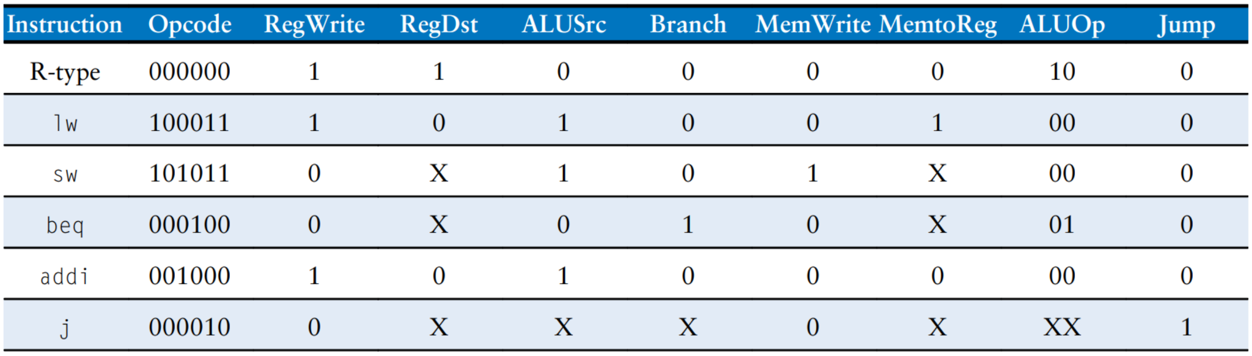
In this report, we will discuss the design decisions and implementation details of the CPU architecture, the assembler and linker, and the various features implemented. A grade sheet will be included, detailing each feature implemented successfully and the number of points requested for it. We will also provide proof that the basic assembly test code works with the implemented features and any additional test code used to test the features. Overall, this project is an opportunity to apply the knowledge of computer architecture to create a functional CPU architecture in Logism.

Data Representation

The following section will describe how the instruction word will be interpreted in the design we follow:

6-Bit OpCode	3-Bit ReadAddr1	3-Bit ReadAddr2	3-Bit WriteAddr	6-Bit Offset

The division of the 6-Bit OpCode follows the MIPS OpCode which is illustrated in the table below:



Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

Figure 1.1: the OpCode table demonstrates the 6-bit OpCode that carries out the instruction.

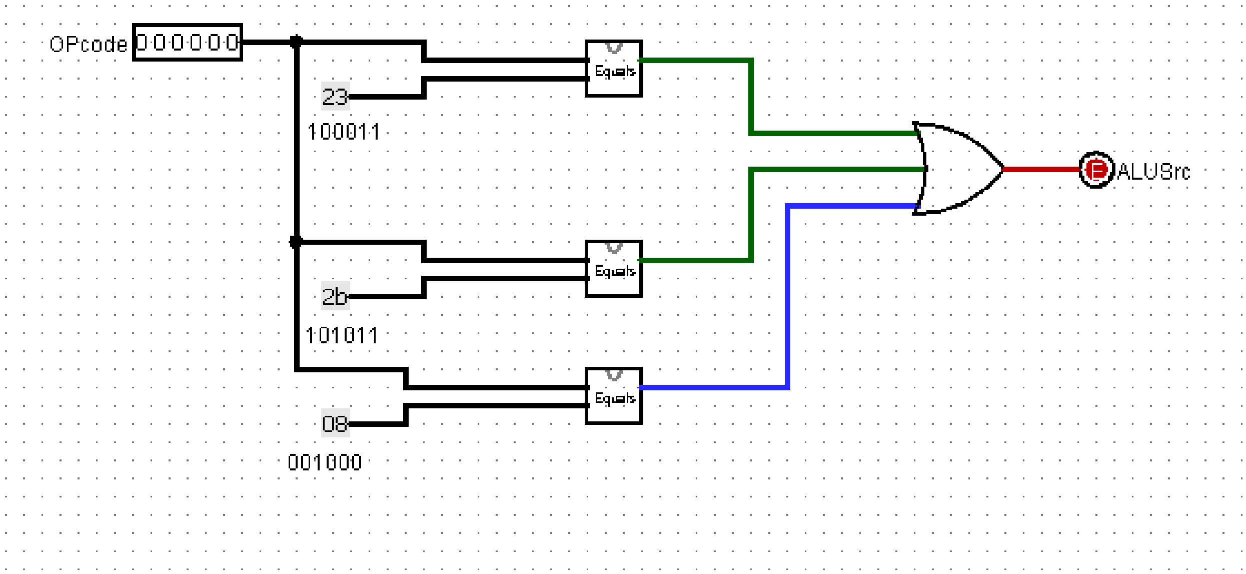
Steps for the development of the processor:

● Designing the basics of the Control Unit - Main Decoder

The Control Unit is a critical component of the processor, and it comprises two essential parts: the Main Decoder and the ALU Decoder. The Main Decoder generates eight outputs based on the six-bit input, which is illustrated in Figure (1.1). The subsequent section delves into the various approaches for designing the Main Decoder.

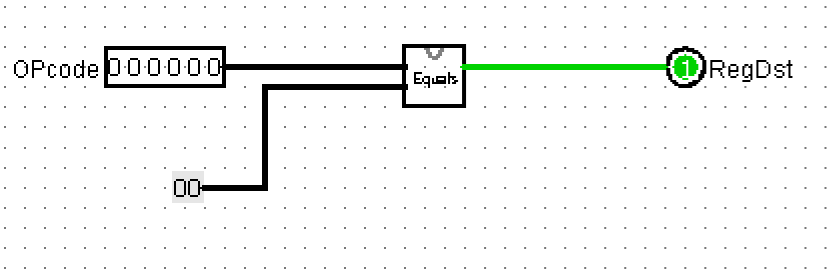
In this part, we start with the design and implementation of two fundamental circuits, namely Equals and NotEquals, which compare two 6-bit input values and determine if they are equal or not equal. The circuit for the Equals component is built using the XOR gate, while the NotEquals component utilizes the Equals component in conjunction with a NOT gate, which results in an inverse operation.

Building upon this foundation, we devise various circuits that leverage the aforementioned truth table to yield outputs such as RegWrite, RegDst, ALUSrc, Branch, MemWrite, etc. Our approach is grounded in the fundamental principles of truth tables, enabling us to create customized circuits tailored to our specific needs. For instance, the ALUSrc logic is demonstrated through a circuit that returns 1 if the OpCode is lw, sw, or addi; otherwise, it returns 0. Analogously, we construct a Boolean expression for each of the main decoders and subsequently employ the derived logic to yield the corresponding output.

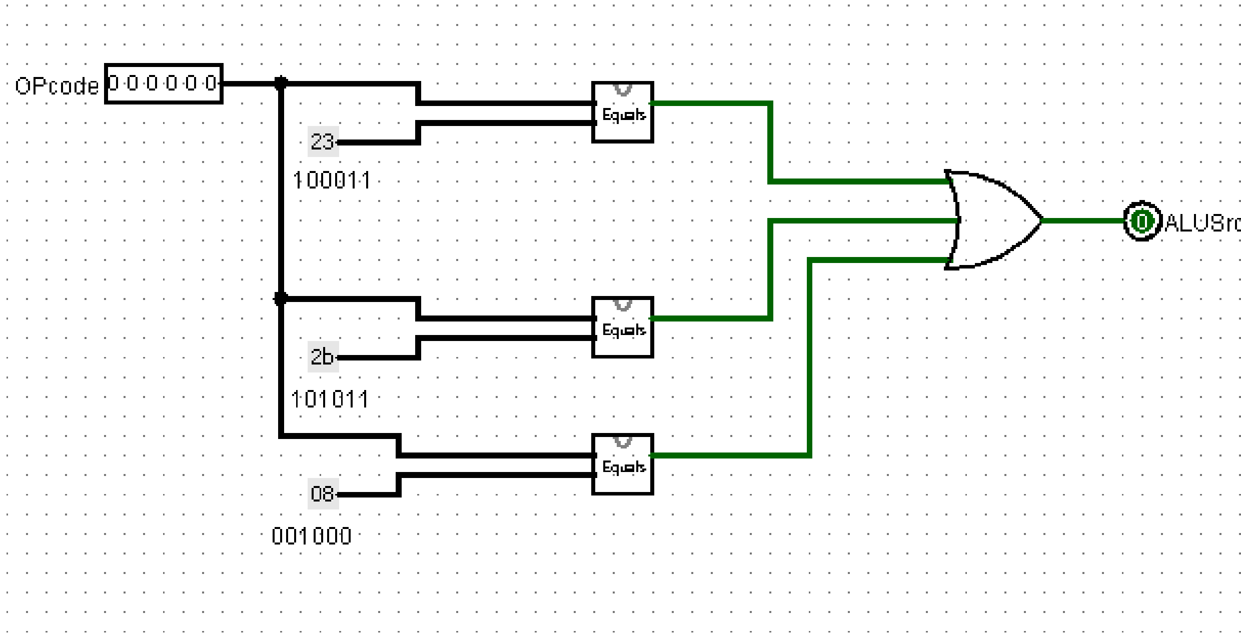


In this way, we design, the logical circuit for each of the following with their adjacent circuit in the following image.

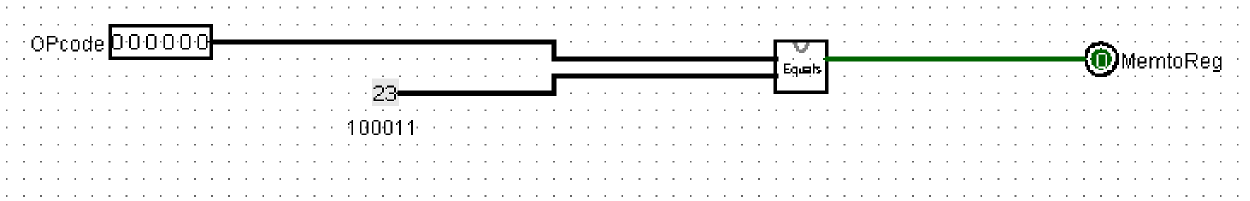
- RegDst



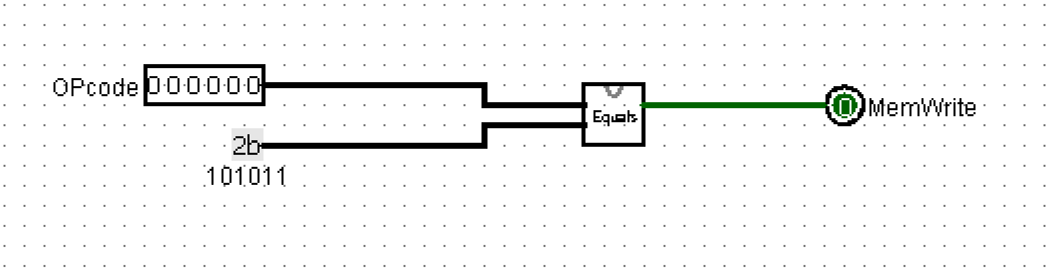
- ALUSrc



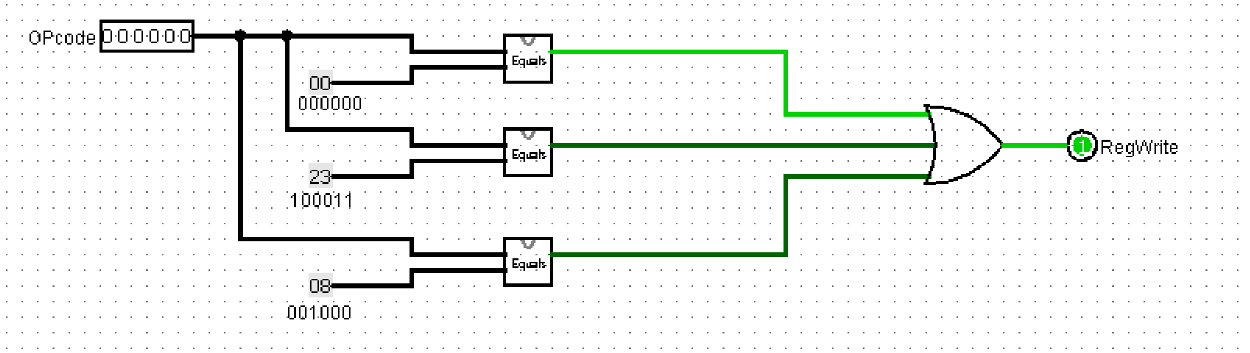
- MemtoReg



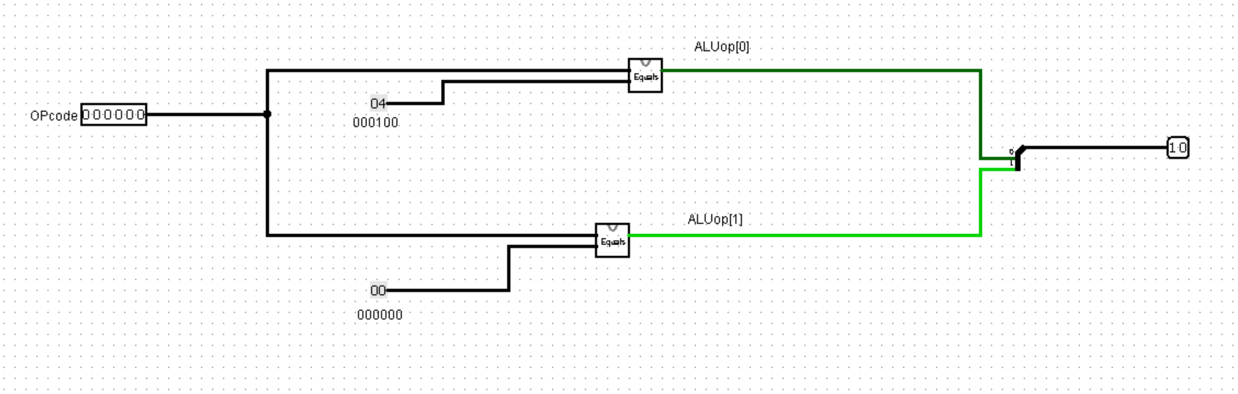
- MemWrite



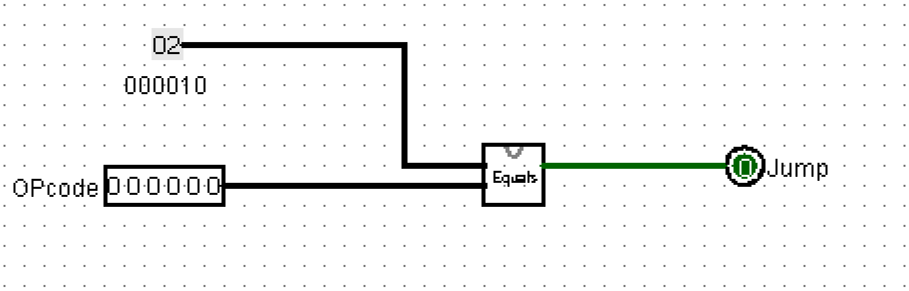
- RegWrite



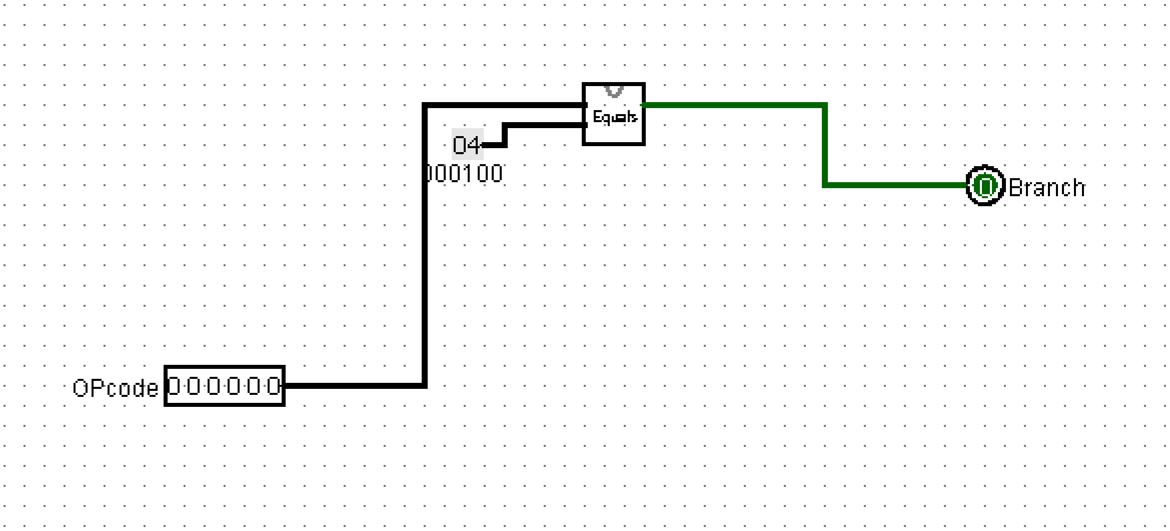
- ALUOp



- Jump



- Branch



- **Designing the Control Unit:**

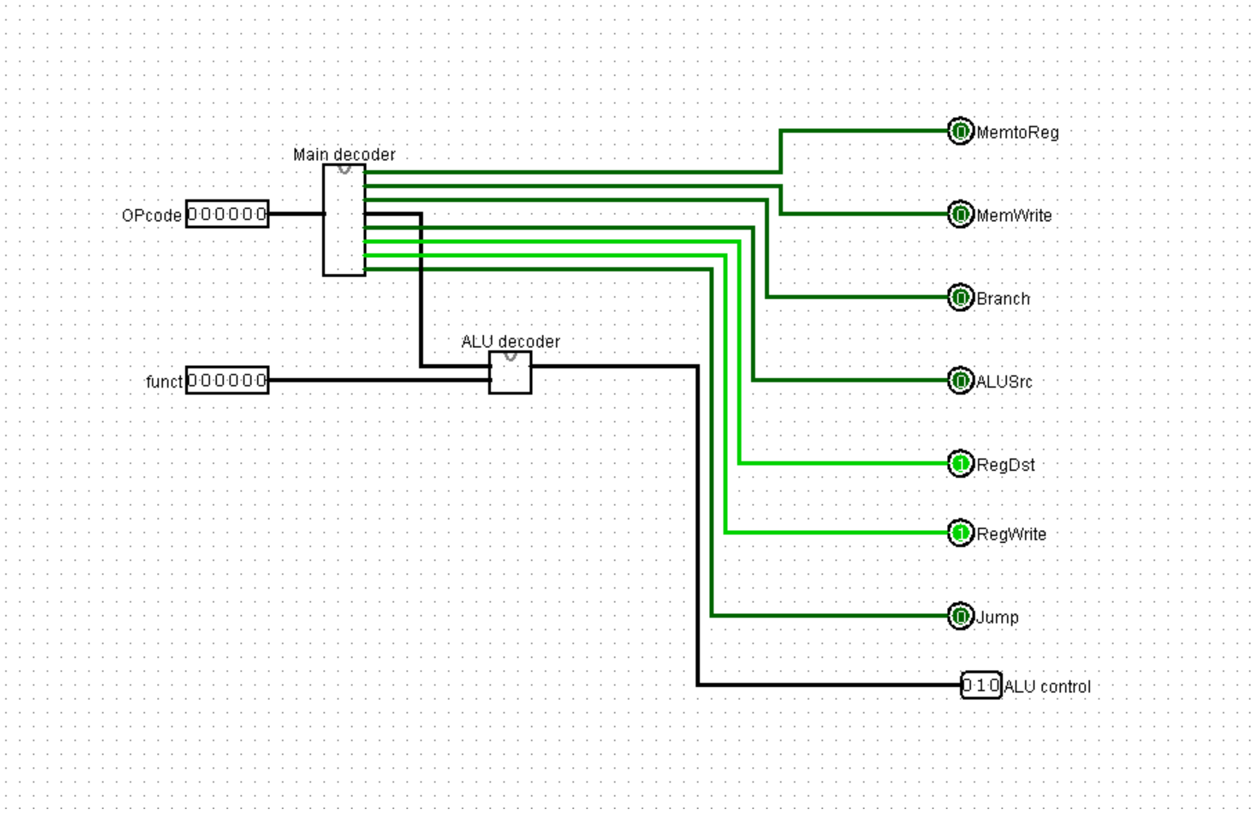


Figure 1.2: The circuit shown above is a control unit.

The circuit shown above is a control unit that uses two parts as described earlier in the paper.

- **Main Decoder**

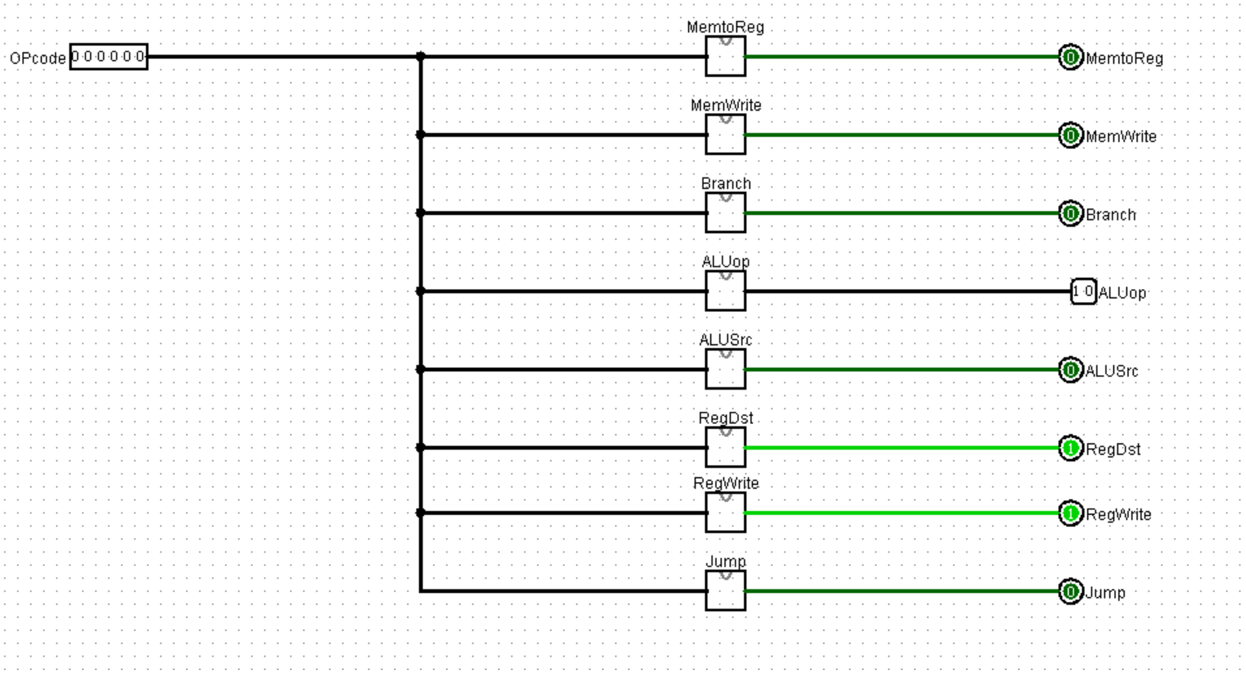


Figure 1.3: The figure above depicts a Main Decoder.

The Main Decoder is a critical component that receives a 6-bit input and processes it using a circuit to produce a specific output bit combination, as previously described. Each constituent element of the Main Decoder has been presented previously. This circuit is a key component of a more comprehensive control unit, as depicted below.

- **ALU Decoder**

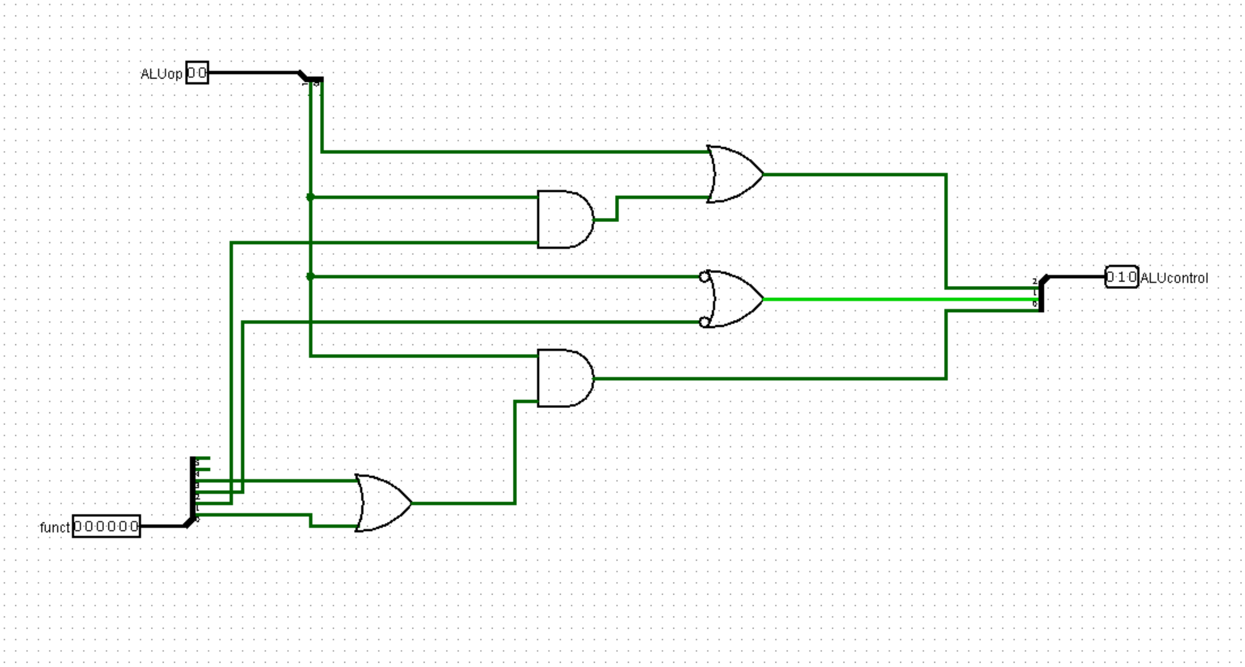


Figure 1.4: The figure above depicts the ALU decoder.

The ALU Decoder is a fundamental element of the control unit responsible for decoding the ALUOpcode obtained from the Main Decoder. In case the ALUOpcode value is 10, the funct field is also activated, which is the last 6-bit field from the least significant bit. This field plays a crucial role in determining the operation to be executed in the R-Type instruction.

In summary, the Main Decoder and ALU Decoder together constitute the larger Control Unit, which receives a 6-bit input starting from the most significant bit and uses the 6-bit field from the least significant bit if required to decide on the appropriate instruction to execute.

- **Designing the rest of the circuit:**

After completing the design of the Control Unit, we subsequently proceeded to implement the complete single-cycle MIPS processor using the schematic diagram provided below.

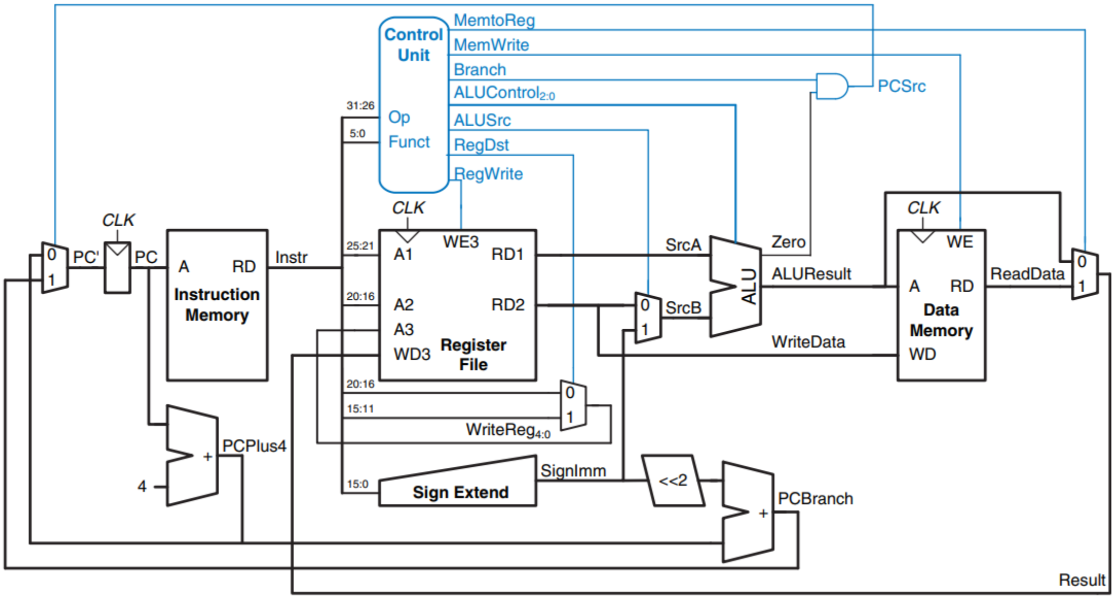


Figure 1.4: Complete single-cycle MIPS processor

Nonetheless, every section that follows will provide a comprehensive account of the design process for each of the individual components that we developed, ultimately leading to the overall design of the single-cycle processor.

- **Arithmetic Logic Unit(ALU)**

  The next component we designed is an ALU. The ALU takes in two 8-bit inputs since our processor is 8 bits capable. It has the following OpCode:

  - 000 - AND
  - 001 - OR
  - 010 - ADD
  - 110 - SUB
  - 111 - SLT

  The following picture depicts the ALU used in our project:

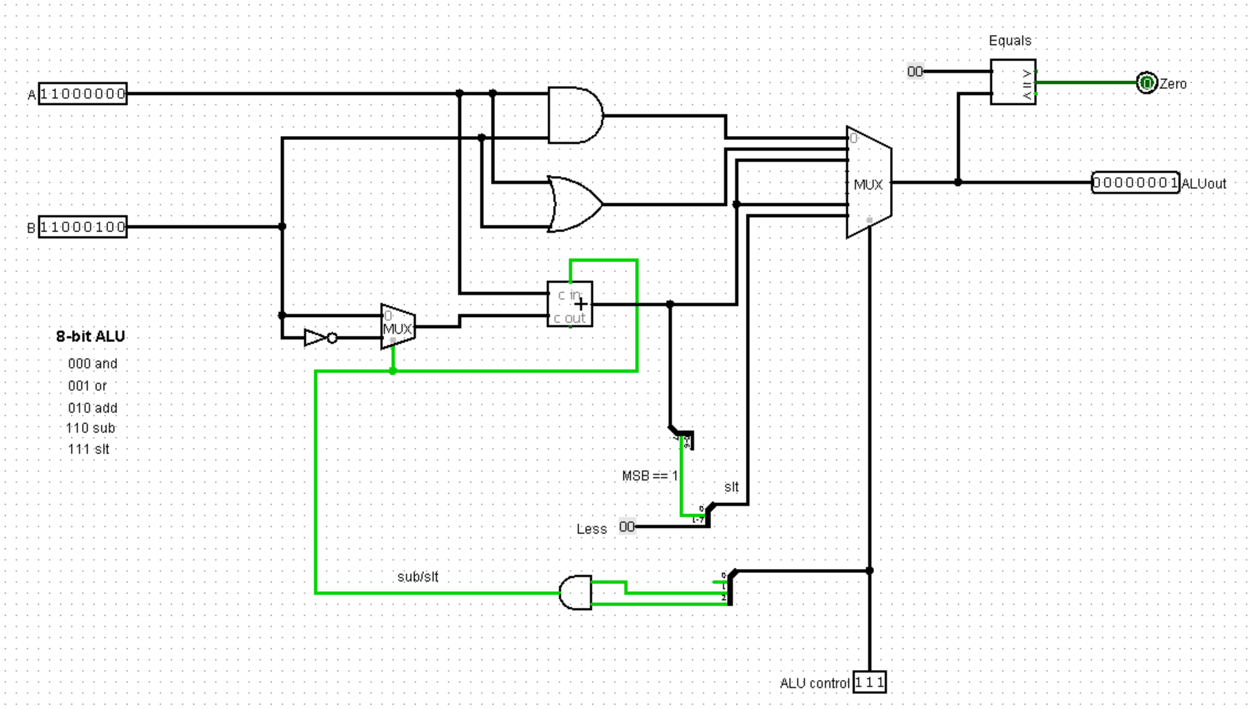


Figure 1.5: The 8-bit ALU

The above picture depicts the ALU which is scaled from the earlier Lab. A and B are the two inputs, each 8-bit long. The ALU is capable of conducting five operations:

- ❖ ADD
  - ➢ When the ALU control has an input of 000, the mux will select the output from the Adder. The output is then returned to the ALUOut
- ❖ SUB/SLT
  - ➢ When the ALU control has the input of 110 or 111, the data path will travel from the sub/slt pathway. Through this path, the subtraction

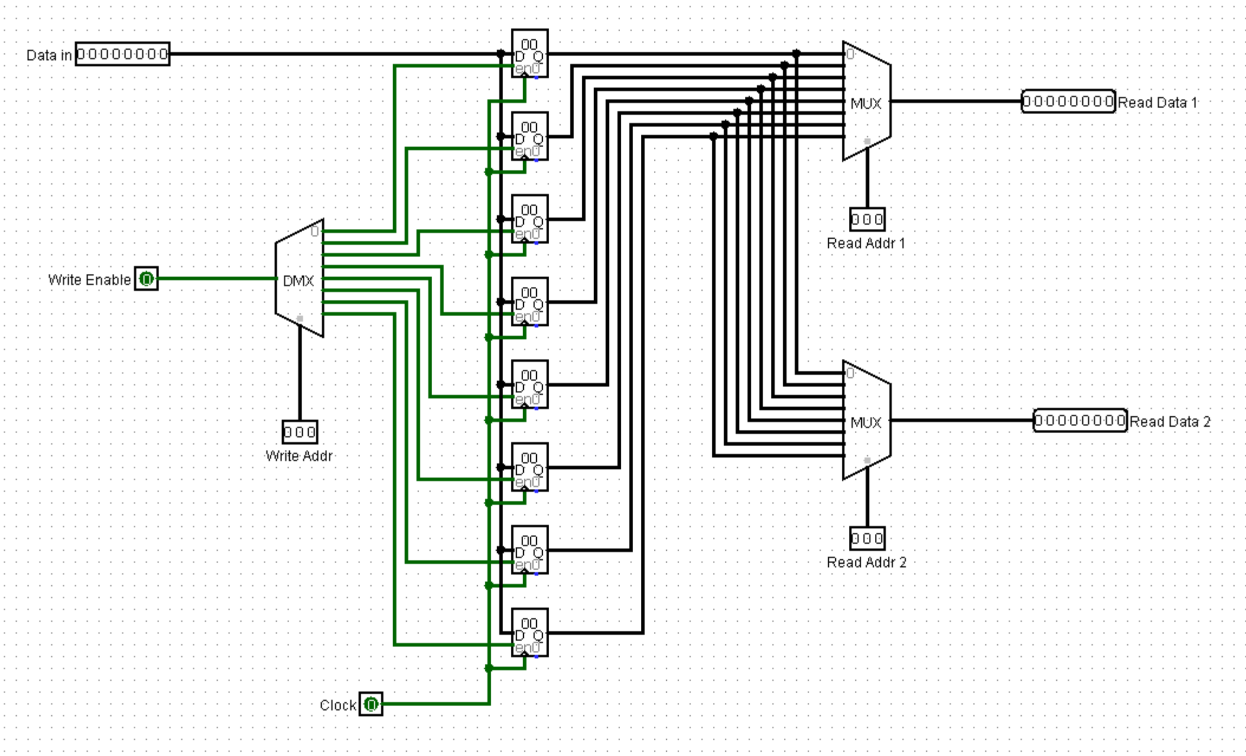
operation is mandated. Both the data are ready to be passed out at the MUX; however, the OpCode decides the data be returned.

-
    - **Overflow case:**
        - However, the overflow detection for the ALU hasn't been designed, and hence if either of the input bits or the resulting bits of the subtraction result in an overflow, then the logic will fail.
        - For example, if A = 0 and B = -128, then the subtraction will result in 128, which can't be stored by 8 bits due to which the ALU will fail to work
        -

- AND
    - When the ALU control has the input 010, the AND gate will be triggered which will send the data through the data path, ready to be outputted upon the instruction from the ALUOpCode
- OR
    - Similarly, with an ALU Control of 001, the OR gate will be triggered which will send the data through the data path accordingly, ready to be outputted upon the instruction from the ALUOpCode

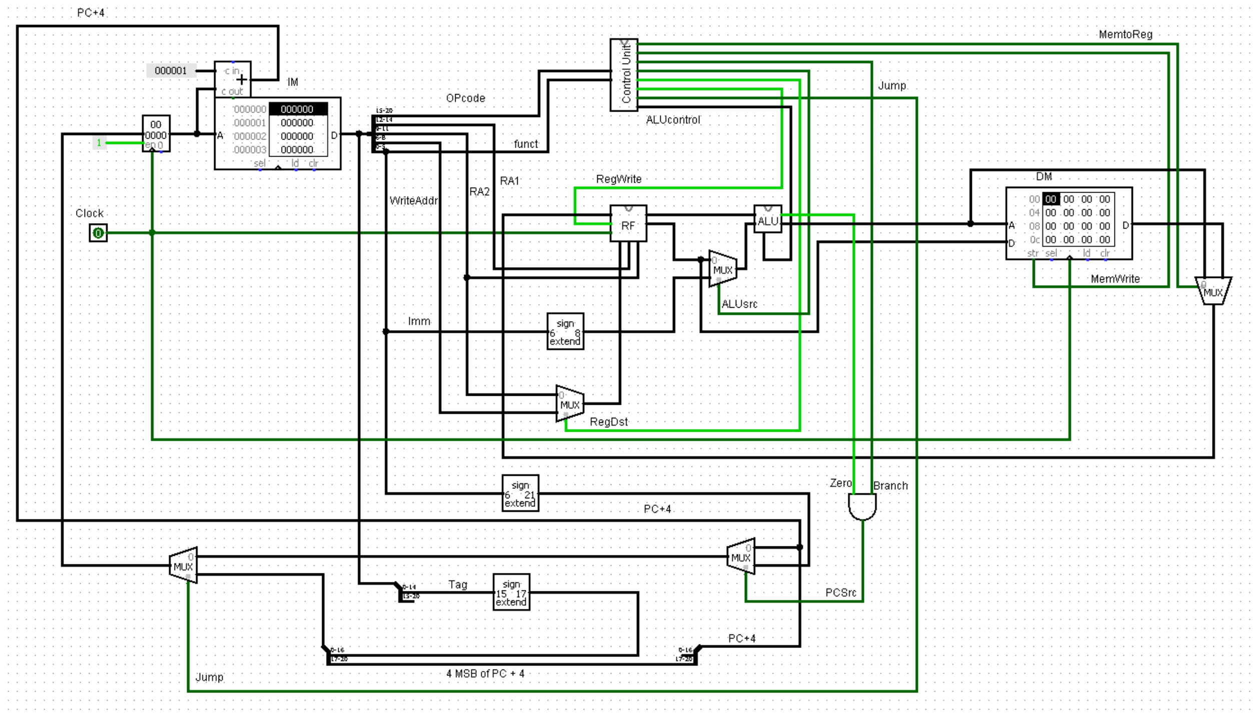
- **Register File:**

The figure below shows a register file



Instruction	OPcode	funct	ALU control
R-type AND	000000	100100	000 (AND)
R-type OR	000000	100101	001 (OR)
R-type ADD	000000	100000	010 (ADD)
R-type SUB	000000	100010	110 (SUB)
R-type Store Less Than(SLT)	000000	101010	111 (set less than)
LW	100011	OFFSET	010 (ADD)
SW	101011	OFFSET	010 (ADD)
beq	000100	LABEL	110(SUB)
ADDI	001000	VALUE	010 (ADD)
J	000010	OFFSET	N/A

Picture of the main circuit along with SLT, Branch, and Jump:



Tests:

The following table contains the instruction with the following equivalent hexadecimal code. It is used for testing the circuit which is 100% unit tested with every case possible. The screenshots of the test are in the google drive linked below: https://drive.google.com/drive/folders/1lN7PUwtvoF08i4Hu1fCQKVr7pOucl1f2?usp=share_link

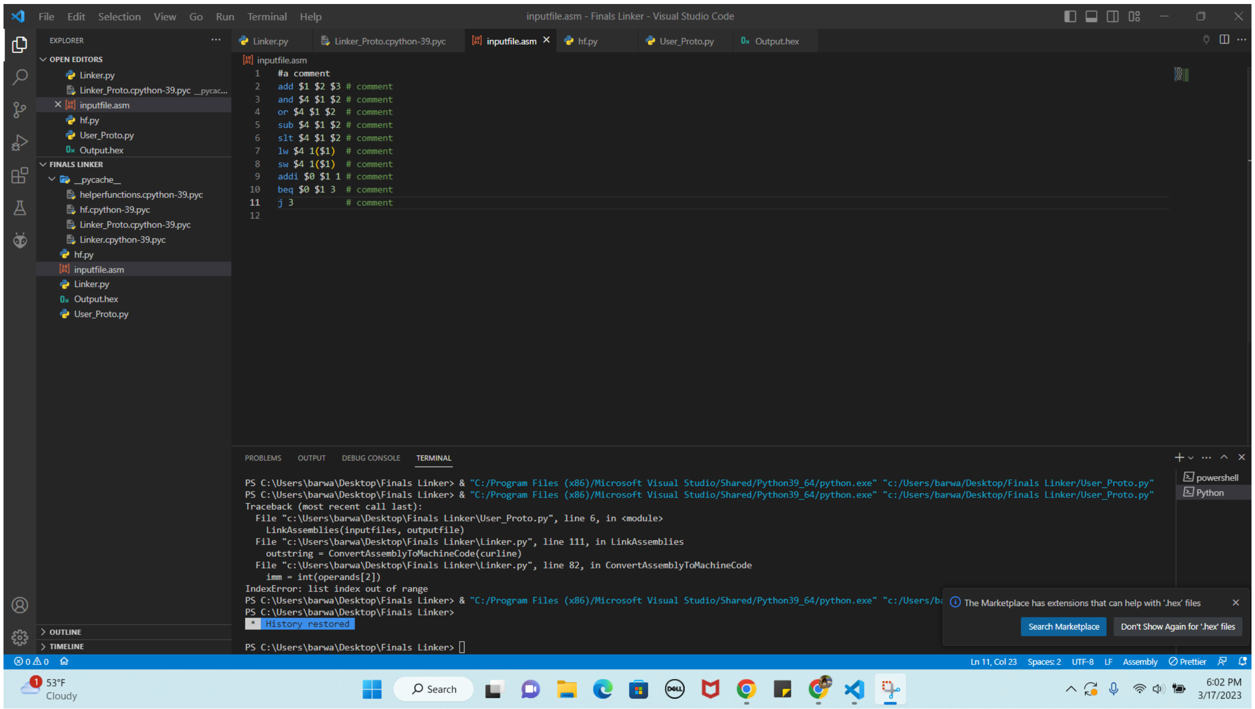
Please use the Union College account to open this google drive.

Test Code	Instruction	RA1	RA2	WA	Funct	EqHex
Test1	ADD(000000)	Reg2(010)	Reg3(011)	Reg1(001)	100000	2660
Test2	AND(000000)	Reg1(001)	Reg2(010)	Reg4(100)	100100	1524

Test3	OR(000000)	Reg1(001)	Reg2(010)	Reg4(100)	100101	1525
Test4	SUB(000000)	Reg1(001)	Reg2(010)	Reg4(100)	100010	1522
Test5	SLT(000000)	Reg1(001)	Reg2(010)	Reg4(100)	101010	152A
Test6	LW(100011)	X (000)	Reg1(001)	WA(100)	OFFSET(000001)	118301
Test7	SW(101011)	X(000)	Reg1(001)	WA(100)	OFFSET(000001)	158301
Test8	ADDI(001000)	Reg1(001)	WA(000)	XXX(010)	VALUE(000001)	41081
Test9	BEQ(000100)	Reg0(000)	Reg1(001)	000	LABEL(000011)	20203
Test10	Jump(000010)	XXX(000)	XXX(000)	XXX(000)	LABEL(000011)	10003

- **Use of Linker to convert the Assembly Code to Hexadecimal code:**

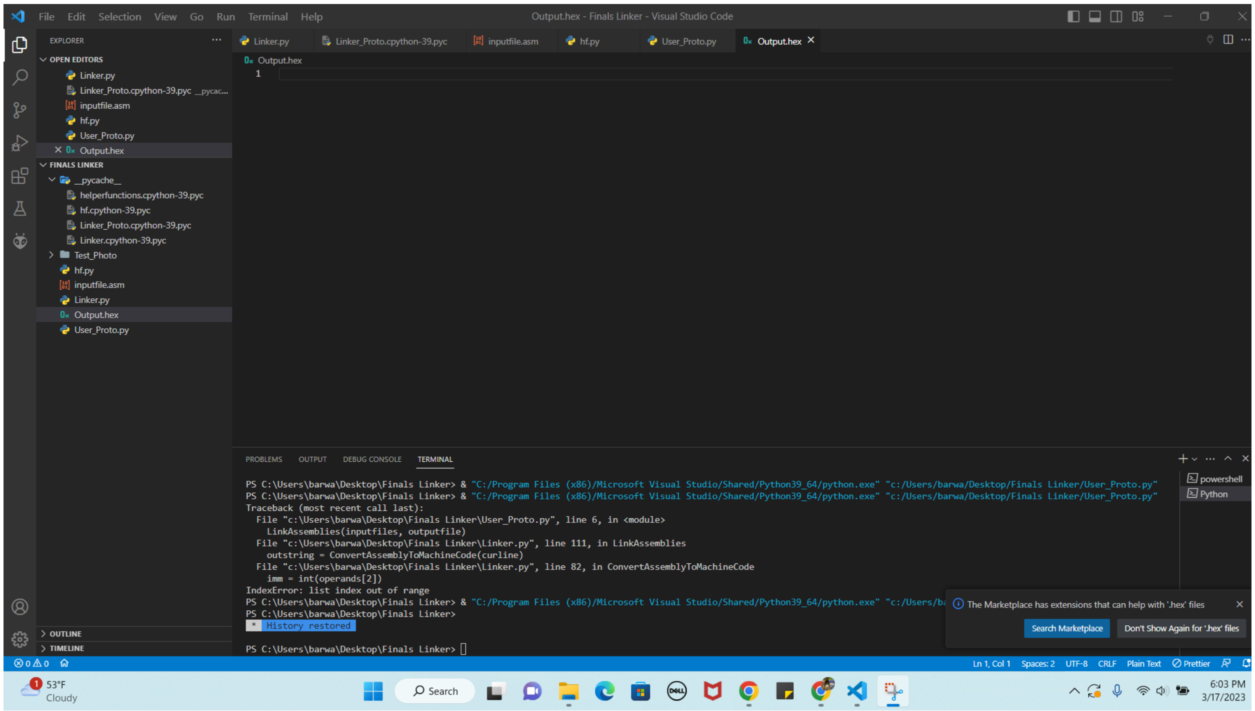
The picture sequence below shows an asm file in which the assembly code is written in one of them, which is then passed through the linker and returns the corresponding hexadecimal file. This is how the project connects the assembly code to the hexadecimalcode.



```
#a comment
add $1 $2 $3 # comment
and $4 $1 $2 # comment
or $4 $1 $2  # comment
sub $4 $1 $2 # comment
slt $4 $1 $2 # comment
lw $4 1($1)  # comment
sw $4 1($1)  # comment
addi $0 $1 1 # comment
beq $0 $1 3  # comment
j 3          # comment
```

```
PS C:\Users\barwa\Desktop\Finals Linker> & "C:/Program Files (x86)/Microsoft Visual Studio/Shared/Python39_64/python.exe" "c:/Users/barwa/Desktop/Finals Linker/User_Proto.py"
PS C:\Users\barwa\Desktop\Finals Linker> & "C:/Program Files (x86)/Microsoft Visual Studio/Shared/Python39_64/python.exe" "c:/Users/barwa/Desktop/Finals Linker/User_Proto.py"
Traceback (most recent call last):
  File "c:\Users\barwa\Desktop\Finals Linker\User_Proto.py", line 6, in <module>
    LinkAssemblies(inputfiles, outputfile)
  File "c:\Users\barwa\Desktop\Finals Linker\Linker.py", line 111, in LinkAssemblies
    outstring = ConvertAssemblyToMachineCode(curline)
  File "c:\Users\barwa\Desktop\Finals Linker\Linker.py", line 82, in ConvertAssemblyToMachineCode
    imm = int(operands[2])
IndexError: list index out of range
PS C:\Users\barwa\Desktop\Finals Linker> & "C:/Program Files (x86)/Microsoft Visual Studio/Shared/Python39_64/python.exe" "c:/Users/ba
PS C:\Users\barwa\Desktop\Finals Linker>
 *  History restored

PS C:\Users\barwa\Desktop\Finals Linker> 
```

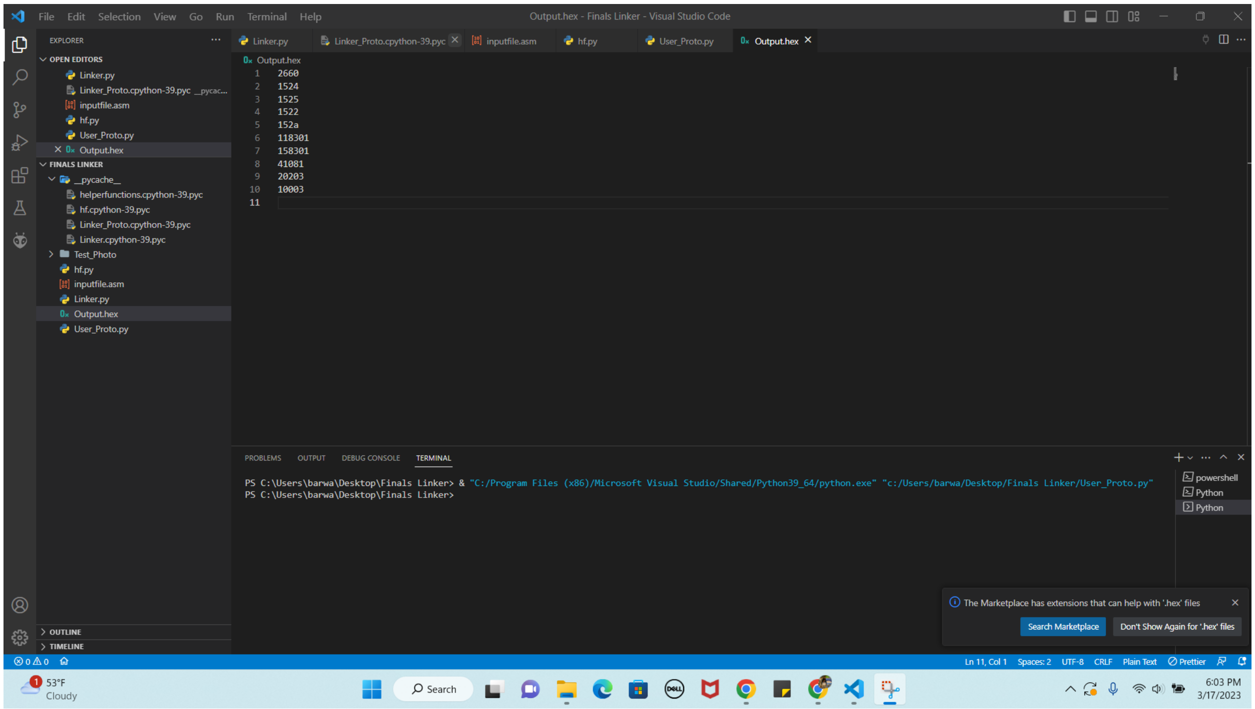


Output.hex

```
1
```

```
PS C:\Users\barwa\Desktop\Finals Linker> & "C:/Program Files (x86)/Microsoft Visual Studio/Shared/Python39_64/python.exe" "c:/Users/barwa/Desktop/Finals Linker/User_Proto.py"
PS C:\Users\barwa\Desktop\Finals Linker> & "C:/Program Files (x86)/Microsoft Visual Studio/Shared/Python39_64/python.exe" "c:/Users/barwa/Desktop/Finals Linker/User_Proto.py"
Traceback (most recent call last):
  File "c:\Users\barwa\Desktop\Finals Linker\User_Proto.py", line 6, in <module>
    LinkAssemblies(inputfiles, outputfile)
  File "c:\Users\barwa\Desktop\Finals Linker\Linker.py", line 111, in LinkAssemblies
    outstring = ConvertAssemblyToMachineCode(curline)
  File "c:\Users\barwa\Desktop\Finals Linker\Linker.py", line 82, in ConvertAssemblyToMachineCode
    imm = int(operands[2])
IndexError: list index out of range
PS C:\Users\barwa\Desktop\Finals Linker> & "C:/Program Files (x86)/Microsoft Visual Studio/Shared/Python39_64/python.exe" "c:/Users/ba
PS C:\Users\barwa\Desktop\Finals Linker>
 *  History restored

PS C:\Users\barwa\Desktop\Finals Linker> 
```



Output.hex

```
2660
1524
1525
1522
152a
118301
158301
41081
20203
10003

```

```
PS C:\Users\barwa\Desktop\Finals Linker> & "C:/Program Files (x86)/Microsoft Visual Studio/Shared/Python39_64/python.exe" "c:/Users/barwa/Desktop/Finals Linker/User_Proto.py"
PS C:\Users\barwa\Desktop\Finals Linker>
```

Points requested:

1. Basic processor: 70 points

2. BEQ: 10 points

3. Support SLT: 5 points

4. Jump: 10 points

5. Control Unit which tests for both J and Branch and is easily scalable for new features due to a reusable logic: 5 points

6. Partially implemented fancy linker: 5 points

7. Well research on MIPS architecture and the entire circuit follows it: 5 point

Total - 110 points