CS 215: Computer Organization and Architecture

Assembler And Disassembler

Final Report

25.04.2022

Members:

Madhav Kanda 20110104

Medhansh Singh 20110111

Saatvik Rao 20110175

Sahil Agrawal 20110178

Abstract

Our project consists of two parts. The first part involves implementing a MIPS assembler that takes assembly code as input and produces the corresponding machine code as output. The second part consists of implementing a MIPS disassembler by producing the MIPS assembly code corresponding to the machine code given as input by the user. This machine code need not be the previously assembled code.

- ❖ Assembler: We have written a program to translate assembly language code to machine language code. The resulting machine code is in the same format as the input for our disassembler program, i.e., the output is a set of lines, each containing 32 bits of single machine instruction.

- ❖ Disassembler: We have written a program that takes in strings representing a machine code and writes out the corresponding assembly language instructions.

Introduction

a. **Goal of the project**

- ➢ Dive into more profound concepts of computer architecture and organization, discover and develop new insights.
- ➢ Our goal was to strengthen our conceptual understanding of the course, and its application in the outside world rather than just having bookish knowledge.
- ➢ The goal of this project was to make such an application which is not very complex and a normal MIPS programmer could easily understand and run it according to his/her convenience.

b. **Rationale**

- ➢ We all know that computers only understand 0's and 1's. Thus, it became necessary for us to design a tool which could convert the human input instructions into binary, which the computer could understand and perform our desired operations easily and efficiently. This task is done by an Assembler. Hence, it holds a huge importance in any language whether it be a high level language or an assembly language like MIPS-32

- ➢ Because disassembled code lacks programmer comments and annotations, reading it is more difficult than reading original source code. Hence, we need a disassembler which could perform operations that are the inverse of those performed by an assembler and translate it into a human readable format.

c. **Importance of the project**

- ➢ This project is of very high importance in the field of Instruction Set Architectures like MIPS.
- ➢ As discussed in the Rationale part, assembler is a tool which reads source code written in assembly language and produces executable machine code. Disassembler is just the opposite. Why are programmers taking so much headache in building these tools? There is a reason behind this.
- ➢ Here comes the importance of this project. Assemblers provide very important inside information about the code which is used by the linkers and is quite crucial for the debuggers to find any type of errors in the code. Obviously, disassembler is important because a human cannot understand the type of instruction by just seeing a series of 0’s and 1’s in front of him. He needs something which could translate that into a ‘readable’ instruction.

d. **Main contributions**

- ➢ We were a team of 4 great friends, so we had a mutual understanding, and our project journey was smooth. As our project consisted of 2 parts i.e the assembler and the disassembler. We divided our group of 4 into 2 subgroups.
- ➢ Medhansh and Saatvik worked on the Assembler code. Madhav and Sahil worked together on the Disassembler code. All of us have understood the whole logic behind both the programs.
- ➢ Sahil and Madhav worked on the project report whereas Saatvik and Medhansh took care of the presentation slides.
- ➢ Lastly, we would like to thank Prof. Sameer Kulkarni for providing us an opportunity to work in teams so that we could understand what

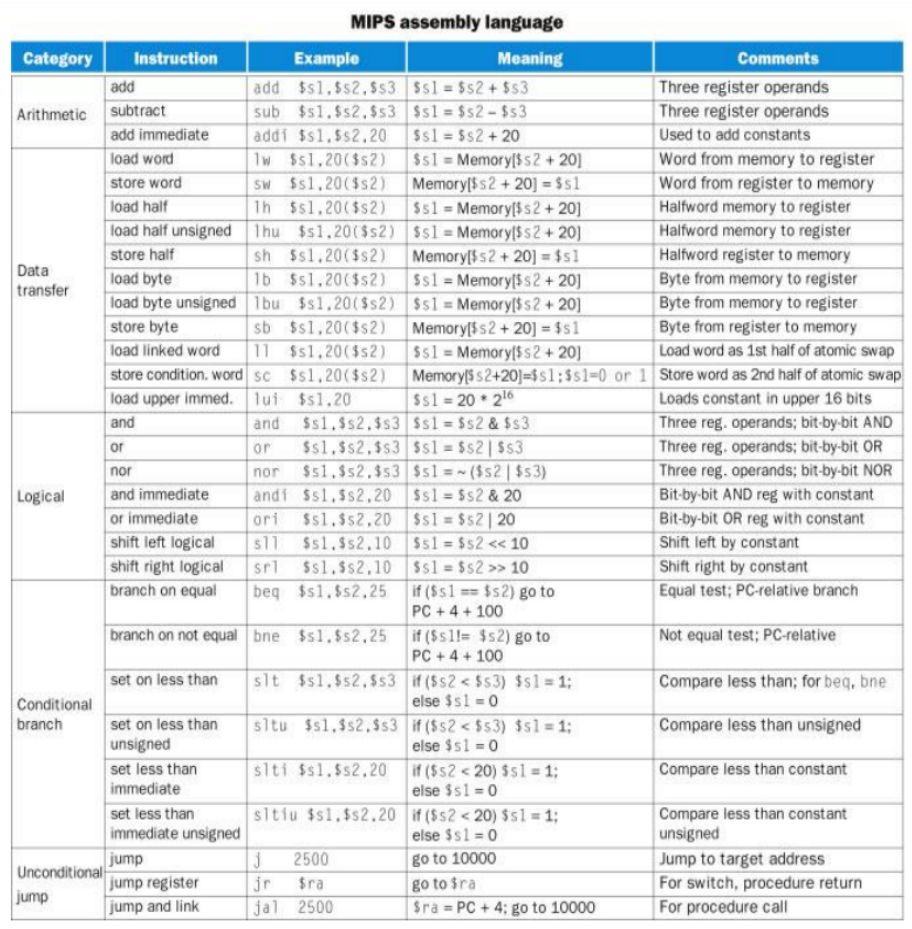
team management is.

Literature review

We are working on the MIPS-32 instruction set architecture. Therefore, we needed a brief document which could tell us about all the instructions which are available in the MIPS-32 assembly.

We have taken into account some famous MIPS instructions such as add, addi,lw, sw, sub, sll, srl, nor, and, or, j, jr, jal, beq, bne, etc.

We referred to the MIPS data card for knowing the syntax of all the above mentioned instructions.



MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Three register operands
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 – \$s3	Three register operands
	add immediate	addi \$s1,\$s2,20	\$s1 = \$s2 + 20	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Word from memory to register
	store word	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Word from register to memory
	load half	lh \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	store half	sh \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc \$s1,20(\$s2)	Memory[\$s2+20]=\$s1;\$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui \$s1,20	\$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	\$s1 = \$s2 | \$s3	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	\$s1 = ~ (\$s2 | \$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	\$s1 = \$s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	\$s1 = \$s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq \$s1,\$s2,25	if (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if (\$s1!= \$s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	For procedure call

We have also looked at various resources (mentioned in the references section) about the MIPS instruction format. To code the whole assembler and disassembler, it was very important to understand the whole instruction format (opcode, rs, rt, rd, shamt, funct) for each and every instruction.

We also referred to the register naming conventions of MIPS from the course textbook (referenced below).

While researching for our project, we found out that there were many assemblers available in the market, but the number of disassemblers were quite less.

A few disassemblers which are used, for eg. The one provided by linux is quite difficult to run as we need to remember too many linux commands and then execute them in the right order. But in our case, both the assemblers and disassemblers are very easy to use and understand.

Project Idea

Detailed specifications of what our project is all about:

a. **Architectural aspects**: We decided to work on the **MIPS Architecture** based on the **RISC** (Reduced Instruction Set Computing) instruction set architecture. We worked on the instruction set having 32-bit size (address space), configured to big endianness. We are handling all the 3 types of MIPS instructions **R, I, and J**. Each instruction set starts with a 6-bit opcode.

   We chose MIPS as the architecture to work upon because it has a fixed encoding system and supports backward compatibility.

b. **ISA specifications:**

   Following are the data instructions on which we are working:

   Arithmetic - *add*, *sub*, *addi*
   Logical - *and, or, nor, srl, sll*
   Conditional branch - *beq, bne, slt*

Jump - *j, jal, jr*
Data transfer - *lw, lhw, lb, sw, shw, sb*

Instruction encoding: Fixed length (MIPS-32)

Final Project Idea: To design and code a solution which could convert the assembly language (MIPS-32 instruction) into the machine code and vice versa.

We take the assembly code input in the form of a .txt file. After getting the machine code output, we ask the user if they want to convert the machine code back to assembly code. If the user says 'N', then we terminate the program, and if the user says 'Y', we give the machine code output in the MachineCode.txt file.

The solution to the project idea:

- ➢ The goal was to make an assembler and disassembler of MIPS 32. So, we made both the tools as separate programs.
- ➢ The code for the assembler and disassembler has been written on the basis of our knowledge of MIPS-32 instructions, 3 types of instructions and their instruction format, and register files.

Pseudo Code explaining the overall idea of the assembler

```
# First of all, we have defined some basic converting functions

functions : bin_to_hex, hex_to_bin, dec_to_bin, bin_to_dec

# 3 dictionaries: key-value pair = (operation, function code)
R-type: {}
J-type: {}
I-type: {}

registers: {}

input = instructions

instruction_set = []
```

```
instruction_set.append(instructions)

result = []

for inst in instruction_set:
    if isnt is R-type:
        execute R block:
            result.append(machine_code)

    else if inst in I-type:
        execute I block:
            result.append(machine_code)

    else if inst in J-type:
        execute J block:
            result.append(machine_code)

write(result)
```

Pseudo Code explaining the overall idea of the disassembler

```
#defined some basic functions

functions: dec_to_bin28, dec_to_bin32, bin_to_dec

function_dict = {(func_code, {operation, operation_format})}
registers = {value, reg}
op_codes = {opcode, op_type}

open (machine_code.txt)

initial_address = 4194304
program counter = 4194304
instruction_list = []
address_store = []
```

```
address_dict = []

for inst in machine_code.txt:
    if op_type is R:
        calculate (rs, rt, rd, funct, shamt)
        write instruction

    else if op_type is J:
        calculate (address of label)
        address_store.append(address)
        write instruction

    else if op_type is I:
        calculate (rs, rt, imm)
        calculate(address)
        address_store.append(address)
        write instruction

instruction_list.append(instruction)

sort (address_store)

for address in address_store:
    address_dict = L1, L2 ....

inst_num = 0
open(assembled_code.txt)

for inst in instruction_list:
    compute(inst)
    write (inst in assembled_code.txt)
```

Further details including code snippets and images have been provided in the next section.

Implementation

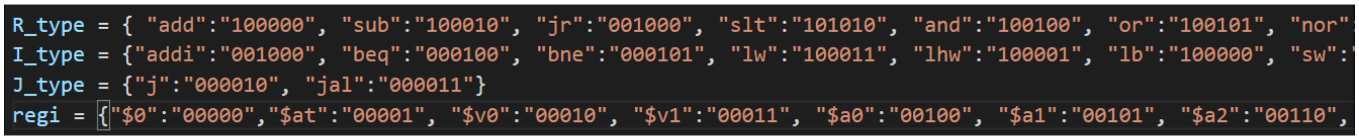
Link to the code: [Github]()

a. **Design Methodology:** We have implemented the assembler and disassembler program in python. We have used python as it provides us with many in-built functions which are way too easy to use. In short, python makes your life easier in terms of code.

   To implement the Assembler,

   First of all we have defined the function for various conversion that we will be needing such as conversion from decimal number to binary, and binary to hexadecimal as we are outputting the machine code as an hexadecimal number.

   We have stored the list of instructions that we will be using, in a dictionary. We have divided the instructions into 3 categories: R type, I type and J type instructions and their corresponding instructions have been stored in the dictionary. We have also made a dictionary for all the 32 registers (key corresponding to the register and value corresponding to the 5 digit binary representation of it).



```
R_type = { "add":"100000", "sub":"100010", "jr":"001000", "slt":"101010", "and":"100100", "or":"100101", "nor"
I_type = {"addi":"001000", "beq":"000100", "bne":"000101", "lw":"100011", "lhw":"100001", "lb":"100000", "sw":
J_type = {"j":"000010", "jal":"000011"}
regi = {"$0":"00000","$at":"00001", "$v0":"00010", "$v1":"00011", "$a0":"00100", "$a1":"00101", "$a2":"00110",
```

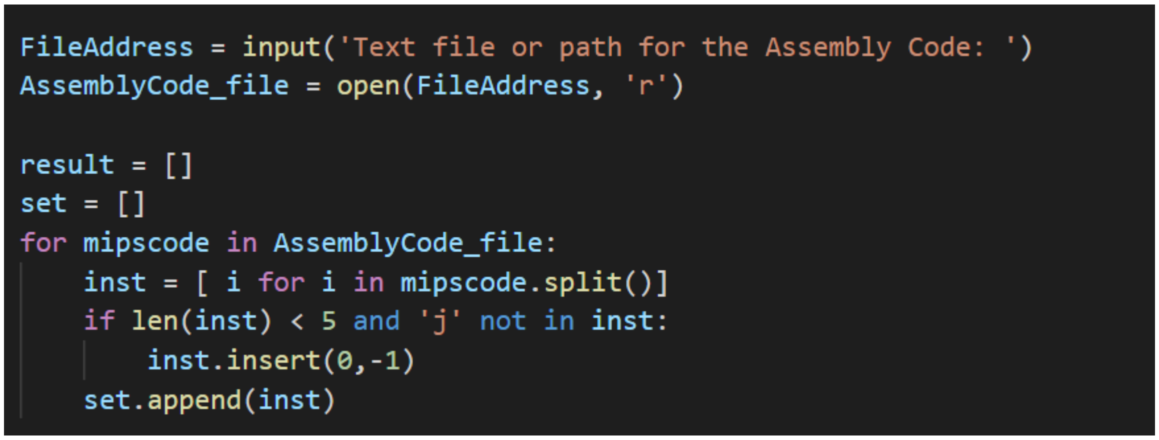
   Eg. for R type, we have add, sub, jr, slt, and, or, nor, srl, sll.

   Eg. for J type, we have j, jal

   Eg. for I type, we have addi, beq, bne, sw, lw, lhw, lb, sw, shw, sb

Then the input is taken from a text file that must be stored in the same directory. Following that arrays are created for the result and instruction set (called set).

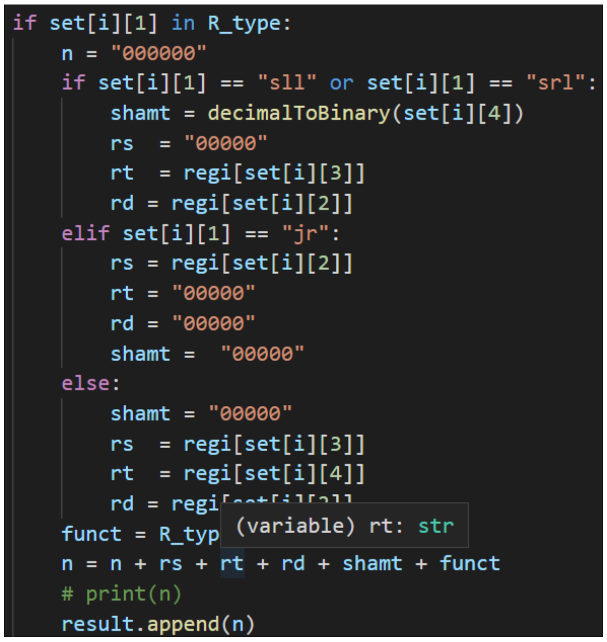
The input is taken such that every line from the file corresponds to one instruction and every instruction is split into elements and stored in the form of an instruction array(called inst). This array (inst) is then stored inside a set array making a 2d array.



```
FileAddress = input('Text file or path for the Assembly Code: ')
AssemblyCode_file = open(FileAddress, 'r')

result = []
set = []
for mipscode in AssemblyCode_file:
    inst = [ i for i in mipscode.split()]
    if len(inst) < 5 and 'j' not in inst:
        inst.insert(0,-1)
    set.append(inst)
```

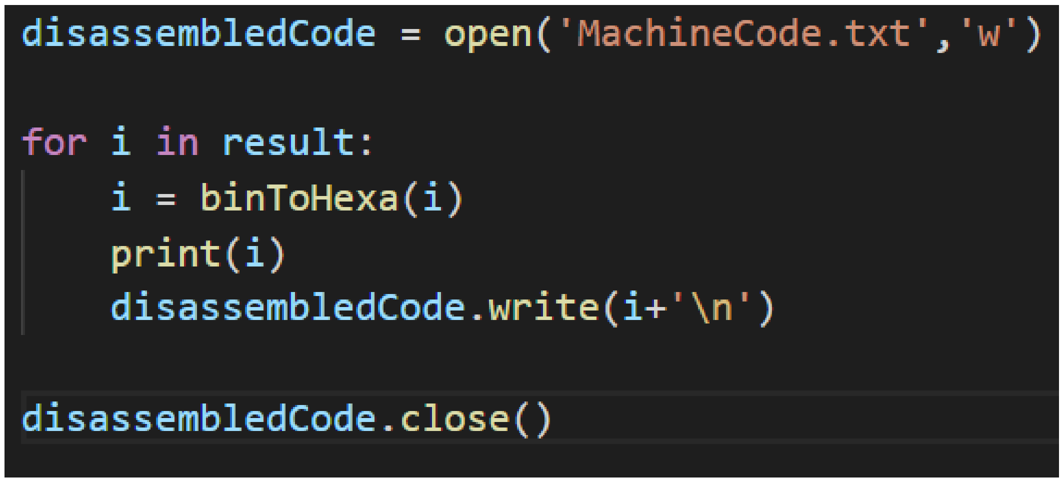
Then the set is traversed through for instructions and each instruction is checked whether it is of R-type, I-type or J-type.

If an instruction corresponding to R type is given, the block corresponding to R type instruction will be executed. All the three blocks are responsible for changing their corresponding type of assembly instruction into machine code.



```
if set[i][1] in R_type:
    n = "000000"
    if set[i][1] == "sll" or set[i][1] == "srl":
        shamt = decimalToBinary(set[i][4])
        rs  = "00000"
        rt  = regi[set[i][3]]
        rd = regi[set[i][2]]
    elif set[i][1] == "jr":
        rs = regi[set[i][2]]
        rt = "00000"
        rd = "00000"
        shamt =  "00000"
    else:
        shamt = "00000"
        rs  = regi[set[i][3]]
        rt  = regi[set[i][4]]
        rd = regi
    funct = R_typ   (variable) rt: str
    n = n + rs + rt + rd + shamt + funct
    # print(n)
    result.append(n)
```

The machine code corresponding to each instruction is stored in the result array and further the machine code are written into the output text file as well as displayed on the terminal.

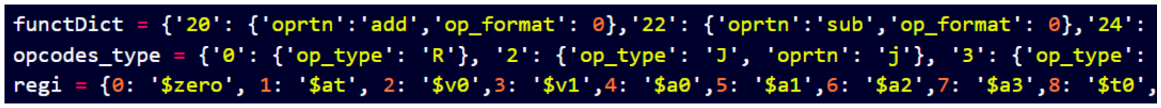


```
disassembledCode = open('MachineCode.txt','w')

for i in result:
    i = binToHexa(i)
    print(i)
    disassembledCode.write(i+'\n')

disassembledCode.close()
```

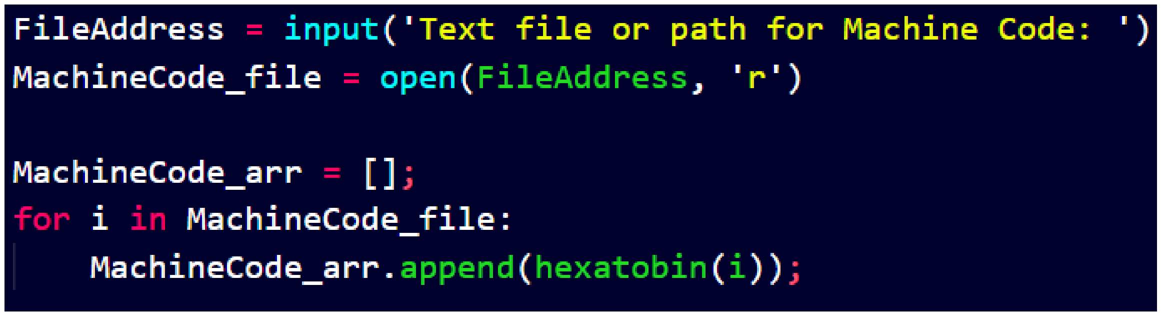
To implement the disassembler,

First of all, we have defined some basic conversion functions such as the hexadecimal to binary , binary to hexadecimal, decimal to binary28, decimal to binary32, binary to decimal as we take the machine code as input in the form of hexadecimal numbers and we want to output the mips instructions , which will need the conversion of these hexadecimal machine codes to binary.

We have made 3 dictionaries, one with the func code, operation and op_format, one having the opcode and the optype (R, I or J), one with the registers as done in the assembler code.



```
functDict = {'20': {'oprtn':'add','op_format': 0},'22': {'oprtn':'sub','op_format': 0},'24':
opcodes_type = {'0': {'op_type': 'R'}, '2': {'op_type': 'J', 'oprtn': 'j'}, '3': {'op_type':
regi = {0: '$zero', 1: '$at', 2: '$v0',3: '$v1',4: '$a0',5: '$a1',6: '$a2',7: '$a3',8: '$t0',
```



```
FileAddress = input('Text file or path for Machine Code: ')
MachineCode_file = open(FileAddress, 'r')

MachineCode_arr = [];
for i in MachineCode_file:
    MachineCode_arr.append(hexatobin(i));
```

As the input, we take the machine code (hexadecimal) from the MachineCode.txt file. We converted each hex instruction into binary form and then appended it into the MachineCode_arr.

We take each element of the array, and through its opcode, we determine which type of instruction it is (R, J, I).

For each type, we follow the op_format and then append the instruction into the set array. We only have to handle the labels for the I and the J types, we do that by storing the addresses of all the instruction sets that need labels to storeAddr array.

We sort the storAddr array just to maintain a generalized way of allotting the labels (L1, L2, L3 and so on).

For each instruction in the set array, we check if we need a label for the instruction. We handle both the cases and at each interaction, we store the instruction in the printArr array and write it into the assembly.txt file.

Finally, we output the printArr array.

b. **Challenges faced**:

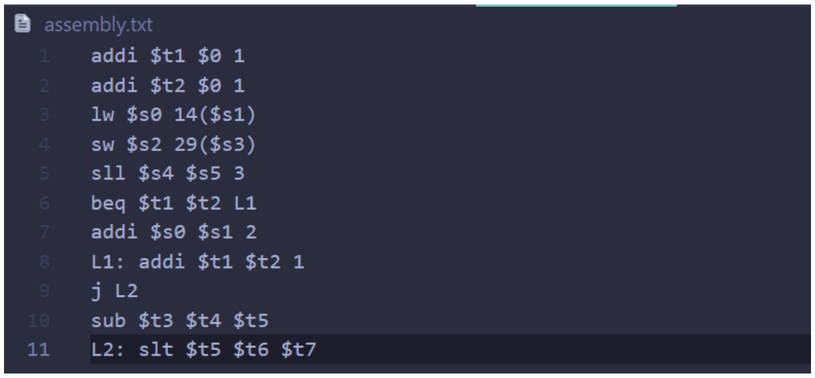
- ➢ The first challenge was faced while writing the program of the assembler. The logic behind the code was very important. As we had taken the MIPS-32 Assembly language, it contained a lot of instructions. We worked on 20 instructions (add, addi, lw, sw, j, jal, beq, slt, and, nor, etc). Understanding each instructions' instruction format, dividing them into R, I and J categories based on their op code was a difficult task.

➢ It was difficult to implement the beq,bne instructions as we had to completely implement a different logic for it as we had to search for the relevant address corresponding to it.

Testing and Experiments

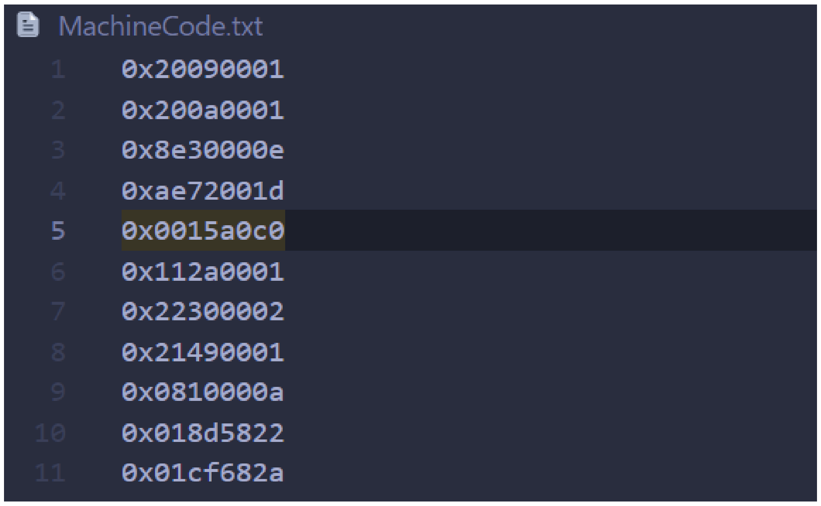
Assembler:

➢ We will be inputting certain sets of MIPS instructions as shown below to see the output of the Assembler:



```
assembly.txt
addi $t1 $0 1
addi $t2 $0 1
lw $s0 14($s1)
sw $s2 29($s3)
sll $s4 $s5 3
beq $t1 $t2 L1
addi $s0 $s1 2
L1: addi $t1 $t2 1
j L2
sub $t3 $t4 $t5
L2: slt $t5 $t6 $t7
```

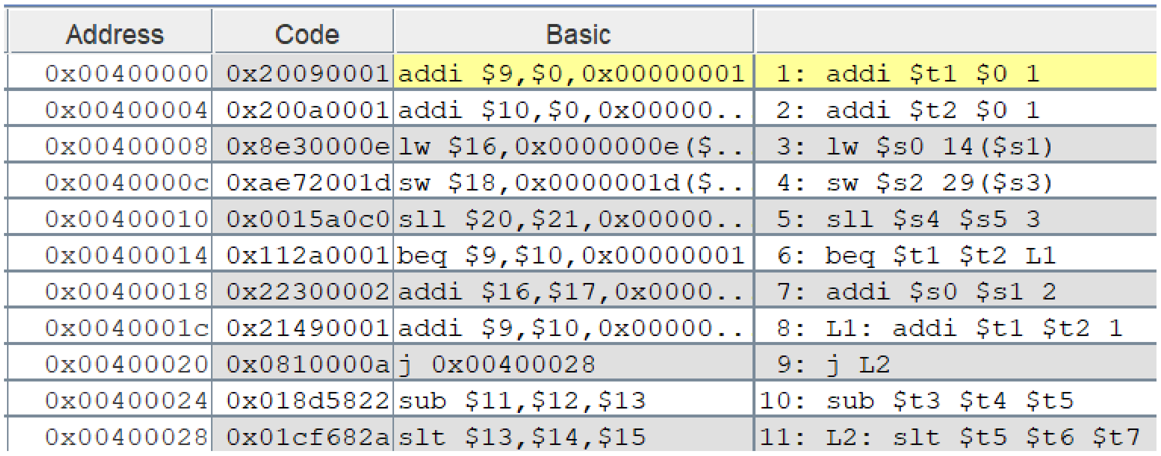
➢ The output that we get from the implemented assembler gives the following output:



MachineCode.txt

```
0x20090001
0x200a0001
0x8e30000e
0xae72001d
0x0015a0c0
0x112a0001
0x22300002
0x21490001
0x0810000a
0x018d5822
0x01cf682a
```

➢ To verify the results obtained from the implemented assembler we verify it using the MARS simulator using the same instructions as given in the implemented assembler:



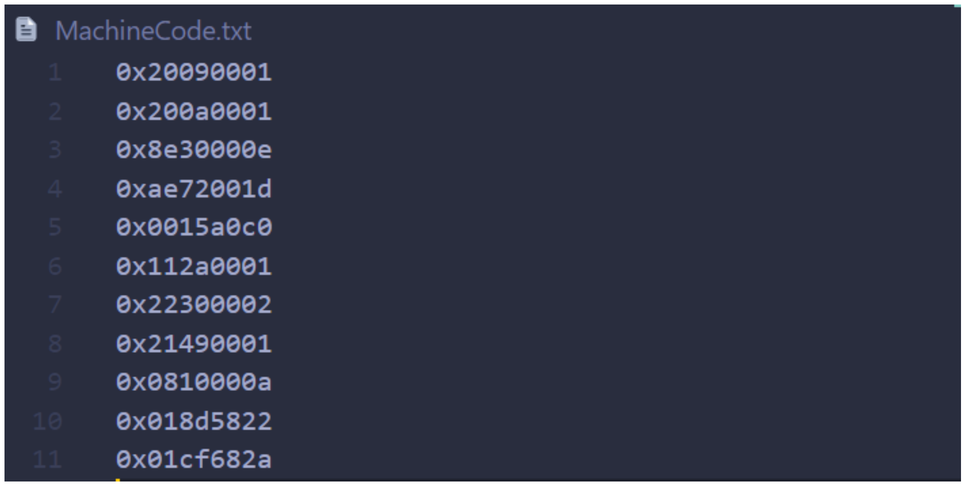
Address	Code	Basic	
0x00400000	0x20090001	addi $9,$0,0x00000001	1: addi $t1 $0 1
0x00400004	0x200a0001	addi $10,$0,0x00000..	2: addi $t2 $0 1
0x00400008	0x8e30000e	lw $16,0x0000000e($..	3: lw $s0 14($s1)
0x0040000c	0xae72001d	sw $18,0x0000001d($..	4: sw $s2 29($s3)
0x00400010	0x0015a0c0	sll $20,$21,0x00000..	5: sll $s4 $s5 3
0x00400014	0x112a0001	beq $9,$10,0x00000001	6: beq $t1 $t2 L1
0x00400018	0x22300002	addi $16,$17,0x0000..	7: addi $s0 $s1 2
0x0040001c	0x21490001	addi $9,$10,0x00000..	8: L1: addi $t1 $t2 1
0x00400020	0x0810000a	j 0x00400028	9: j L2
0x00400024	0x018d5822	sub $11,$12,$13	10: sub $t3 $t4 $t5
0x00400028	0x01cf682a	slt $13,$14,$15	11: L2: slt $t5 $t6 $t7

➢ The results obtained from MIPS and the implemented assembler are same. Thus, the assembler correctly outputs the machine code.

Disassembler:

➢ To verify the correctness of the implemented Disassembler we will be using the output received from the MARS simulator using the above mentioned

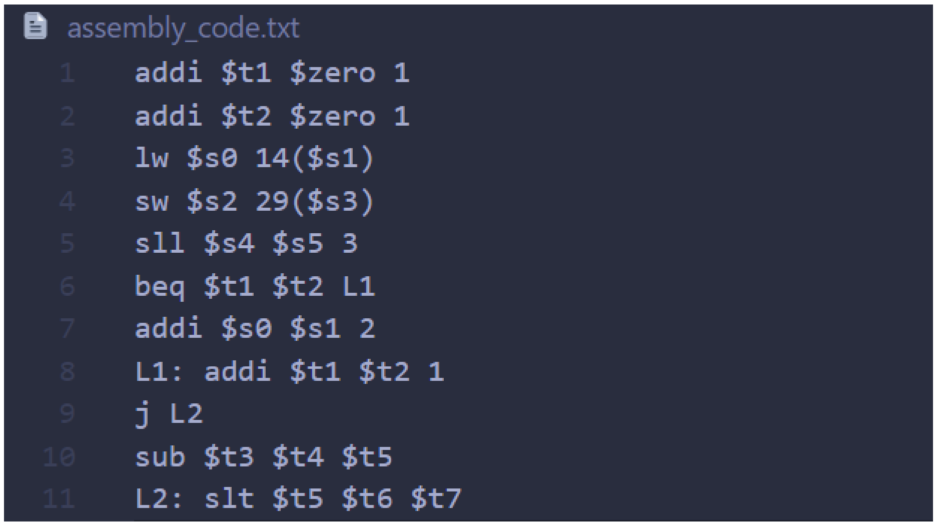
MIPS instructions. Thus the input given to the Disassembler is:



MachineCode.txt

```
0x20090001
0x200a0001
0x8e30000e
0xae72001d
0x0015a0c0
0x112a0001
0x22300002
0x21490001
0x0810000a
0x018d5822
0x01cf682a
```

➢ The results obtained from the disassembler is as shown below. The mips code received as the output is same as the one that we gave as the input in the MARS simulator.



assembly_code.txt

```
addi $t1 $zero 1
addi $t2 $zero 1
lw $s0 14($s1)
sw $s2 29($s3)
sll $s4 $s5 3
beq $t1 $t2 L1
addi $s0 $s1 2
L1: addi $t1 $t2 1
j L2
sub $t3 $t4 $t5
L2: slt $t5 $t6 $t7
```

Data Analysis

- ➢ Since we are dealing with either MIPS instructions or machine code, hence we don't have any numerical data to perform any computation or do the data analysis.

Conclusion

This project helped us strengthen our conceptual understanding of the course and its application in the outside world rather than just bookish knowledge.

We implemented the assembler which takes the MIPS-32 instructions as input in a .txt file and outputs the machine code in a separate .txt file. We also implemented a disassembler which takes the input as the machine_code.txt file and writes the set of instructions in the assembly.txt file.

We as a team are quite happy to work on such an interesting project. Everyone tried their best to contribute their time and knowledge in this project. We must say that the concepts which were taught prior to the mid semester examination were put to a great test through this project.

If we would have had more time to work on this project, we could have implemented more instructions from the MIPS data card.

Future Scopes

There are a lot of future prospects to this project.

- ➢ We could have added comment detection in our assembler. Like, if anybody adds any comment in the code, then our assembler should detect that ok, this is a comment, and I don't need to convert it into the machine code.
- ➢ To make the project more presentable, we can add a graphical user interface, which could showcase our project in a more presentable manner.

References

[1] MIPS architecture - Wikipedia. “MIPS Architecture - Wikipedia.” en.wikipedia.org, January 1, 1985. https://en.wikipedia.org/wiki/MIPS_architecture.

[2] David A Patterson and John Hennessey, Computer Organization and Design 5 th Edition, Morgan Kaufman Publishers, 2017.

[3] William Stallings, Computer Organisation and Architecture, 10 th Edition, Pearson Edition, 2020.