ES 215 COMPUTER ORGANIZATION AND ARCHITECTURE PROJECT REPORT

ASSEMBLER AND DISASSEMBLER

April 22, 2022

Tanvi Chaudhari 20110043

chaudhari.tanvi@iitgn.ac.in

Hetvi Patel 20110076

hetvi.patel@iitgn.ac.in

Lipika Rajpal 20110102

lipika.rajpal@iitgn.ac.in

Samiksha Kamble 20110182

samiksha.kamble@iitgn.ac.in

Indian Institute of Technology, Gandhinagar

1. Abstract

Assembler and disassembler principles have greatly eased human-computer interaction. In fact, it has aided people in gaining control over the tasks that computers undertake. There is a growing demand for software that can be understood by both people and robots. The purpose of this study is to present an organized approach to MIPS 32 assembler and disassembler, as well as its relevance and implementation. This project report includes an overview of the MIPS 32 architecture, assembler and disassembler, idea behind the approach towards programming a set of assembler and disassembler compatible with a wide range of instructions.

2. Introduction

The project aims at the development of an assembler and disassembler that supports the MIPS-32 architecture. MIPS stands for Microprocessor without Interlocked Pipeline Stages. MIPS architecture is based on the fixed-length, regular encoding format of the instruction set. It is a family of reduced instruction set computer(RISC) instruction set architectures. Developed by John Hennessey in the 80s, MIPS architecture is known for its simplicity yet compatibility with various high-level languages and operations.

The MIPS 32 architecture is based on a fixed-length encoding of the instructions. The 32 instructions are written in a predefined regular order. It used a fixed set of registers for carrying out the load/store operations. It has 32 registers available to the processor. The instructions of this architecture are divided into three

categories - I-type, R-type, and J-type.

The following figure shows the instruction format of all the three classes:

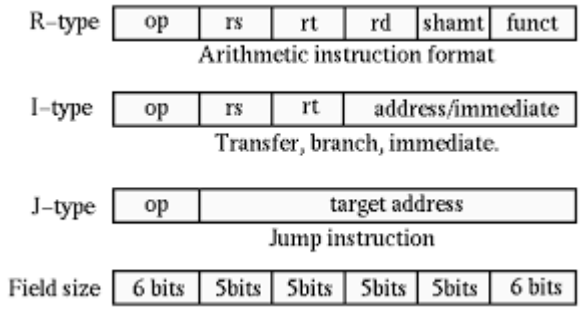


Fig.1:https://www.google.com/imgres?imgurl=https%3A%2F%2Fi.stack.imgur.com%2F5rgyM.gif&imgrefurl=https%3A%2F%2

The compiler converts the high-level code into the assembly language. The assembly language is human readable and consists of mnemonic codes for instructions, symbols to denote various registers, etc. However, the machine does not understand this language. An assembler converts the assembly language into the machine-readable format or the machine language. A disassembler does the opposite job. It converts the machine codes to corresponding assembly codes.

- The goal of this project

  To make an assembler that converts MIPS 32 assembly language into machine code and develop a disassembler that converts the machine language code into MIPS 32 assembly language code. Deployment of the assembler and the disassembler to a Streamlit-based web app.

- The rationale of the project

  Being a bunch of computer science students in the second year, we have a keen interest in the nitty-gritty of what happens at the backend of our

computer system. We believe that a sound understanding of the MIPS architecture will prepare us to understand more complex architectures. The best way to understand a concept is applying it practically. This project caters to all our interests. Hence, we were compelled to take up the MIPS 32 assembler-disassembler project as a meaningful conclusion of the ES215 course.

• The importance of the project

As MIPS 32 has powerful features and is highly performance efficient architecture, it plays a very crucial role in various electronics and has important features like SIMD (Single Instruction Multiple Data) and virtualization. The MIPS architecture's advantages include its high-performance caches' flexibility and memory management techniques. Hence it is necessary to create a way in which MIPS 32 assembly language can be converted into machine language and vice versa. The assembler helps us to make assembly language executable by helping us write human - understandable code to easily feed to machines.

• Your main contributions to the project

We created a MIPS 32 assembler and disassembler using python, in which python will take assembly code from the user and convert it into machine language understandable by the computers. We used the following libraries streamlit, pyparsing, bitstring

We distributed the work as follows:

Assembler - Lipika, Hetvi

Disassembler - Samiksha, Tanvi

Report and Presentation - Lipika, Hetvi, Samiksha, Tanvi

GUI - Lipika

3. Literature Review

• How, when the problem was raised by whom, what event, etc.

As the technology was growing rapidly back in 1947 it is believed assembly language was started. Kathleen Booth was working on the ARC2 at Birkbeck, University of London when she first created the assembly language. In late 1948 during a bootstrap it was first seen that there was an electronic delay storage automatic calculator that contained an assembler. It was made to convert assembly language software into machine language, code, and instructions that a computer can understand. David Wheeler is regarded as the inventor of assembler. In their 1951 book The Preparation of Programs for an Electronic Digital Computer, Wilkes, Wheeler, and Gill coined the word "assembler," which they defined as "a programme that assembles another programme consisting of numerous portions into a single programme."

• Any attempts have been proposed to address the problems

- Assembly languages were quite popularly implemented over a wide range of computers. It was because they relieved the programmers from complex tasks. However, their use was drastically reduced in the 1980s due to the avail and introduction of high-level languages such as C, C++, and Python.
- Many programs and versions of Assembler-Disassembler have been

written using high-level languages. This trend was completely substituted with the initiation of the Burroughs MCP in 1961. (It was written in ESPOL). Additionally, several commercial programs and applications were written using assembly languages.

• Pros and Cons of Assembler and Disassembler (each existing solution)

1. Prons - Assembler and disassembler allow reduction of complexity for ease. They require less memory usage if both programs are efficient. It should run faster and require less instructions to get the outputs. All these factors are critical for companies and jobs.
2. Cons - Writing instructions for each function and opcode takes too much time and effort. They are a large number of functions, opcodes, and registers. Therefore, a computer takes more space and memory to run complex and lengthy code.

• How does your proposed solution fit within here, i.e., is your solution improving an existing solution, just another solution, or revolutionizing the way of thinking?

- Our proposed solution is improvising many existing solutions as:
    1. The number of instructions our assembler and disassembler are handling is quite large.
    2. Some of the existing solutions do not provide more than one input format for J-type instructions. For example, opcode+26 bits is the general input format used. Our solution supports two types of input formats : opcode+26 bits and opcode + label(of any other

instruction line).

3. Functions like sll, srl are not supported by many common solutions as they require some optimization in the specification of R-format. Usually, common solutions only support three register-input formats for R-type instructions. However, our solution is compatible with ‘reg, reg, imm’ format of sll and srl instructions.

4. Our project idea

The project aims at developing the assembler and the disassembler that are compatible with the MIPS 32 architecture instruction set. Our project can handle all the instructions mentioned in the following data card(Fig. number). This is an exemplary data card provided to us by the professor. The specification of all the 32 registers is also done according to information provided during the lectures. (see Fig number)

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

Fig 2. Source: Assignment 2 posted on the classroom

MIPS Reference data card

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

FIGURE 2.14 MIPS register conventions. Register 1, called $at, is reserved for the assembler (see Section 2.12), and registers 26–27, called $k0–$k1, are reserved for the operating system. This information is also found in Column 2 of the MIPS Reference Data Card at the front of this book.

Fig. 3. Souce: Assignment 2 on classroom

The assembler and disassembler have been deployed on a web application. Here, the user can find well-specified areas and instructions on where and how to give the inputs. The output of the assembler and the disassembler is also displayed on the web application interface. The detailed implementation of each stage of the project is mentioned in the following section.

5. Project Implementation

- Programming languages:

The assembler and the disassembler have been coded in Python. A small patch of HTML is used to set the web page's background.

- Libraries used:

1. Pyparsing is a python module used to parse the input (in string format) according to the format specified by the programmer. We have used this module to read the assembler's input (in human-readable assembly code) and identify different components of the instructions. The library enables

us to identify alphabets, numbers, special characters, and more in the input string. Hence, by treating each code line of the input file, we can identify different components of the instructions like mnemonics, registers, immediate values, etc.

```
line_format = Combine(Optional(Word(alphas)) + Optional(Word(nums)))    #this will be used for translating the load, store and branch instructions.
label_format_j = Word(alphas)
imm_format_j = Word(nums)
remove_comma = Suppress(',')  #we are ignoring the commas in the instructions of the assembly code
identify_reg = oneOf(all_regs)    #identify_reg will contain the valid regs mentioned in the input file
num = Combine(Optional('-') + Word(nums))    #for parsing the "-x" string as negative integer -x. If the number is positive then also it will be store
label_add = label_format_j.setResultsName('label_add')
imm_add = imm_format_j.setResultsName('imm_add')
address_for_branch = line_format.setResultsName("address_for_branch") + Suppress(":")     #if the branch of a loop is in the form of a label, eg. For:
rs = identify_reg.setResultsName('rs')    #from now any valid input reg from the all_regs array will be denoted as rs
rt = identify_reg.setResultsName('rt')    #from now any valid input reg from the all_regs array will be denoted as rs
rd = identify_reg.setResultsName('rd')    #from now any valid input reg from the all_regs array will be denoted as rs
imm = num.setResultsName('imm')           #any input string containg numbers will be taken as imm
addr = line_format.setResultsName("address")     #any input string parsed according to the line_format will be denoted as address

R_format = (oneOf(R1).setResultsName('instruction') + White() + rd + remove_comma + rt + remove_comma + num.setResultsName('shamt')) ^\
           (oneOf(R2).setResultsName('instruction') + White() + rs)^\
           (oneOf(R3).setResultsName('instruction') + White() + rd + remove_comma + rs + remove_comma + rt)
               #This contains the possible instruction formats of R-type instructions. Any input valid ins of the above options will be denoted as R
```

Fig. 4

The above figure (Fig. 4) is a snippet of our code representing our application of the Pyparsing library in the project.

2. BitsArray: It is a python library that we have used to convert a string representing a decimal to a corresponding binary string. BitsArray is a sequential data structure that stores the booleans as distinct elements. We have converted these booleans' values to strings.

3. Streamlit: It is a python module for creating and deploying a web

application. This is at the core of our GUI. Streamlit connects the GUI with the backend python program.

- GUI and the i/o operations:
  The GUI shows two input panels for the assembler. One panel asks for the address of the input file in .txt format. The second panel asks the user to specify the initial address of the first instruction in hexadecimal.
  These inputs are fetched by the backend and used by the assembler to create an output file that contains the machine code translations of the input instructions. The contents of this .txt file are printed on the GUI.

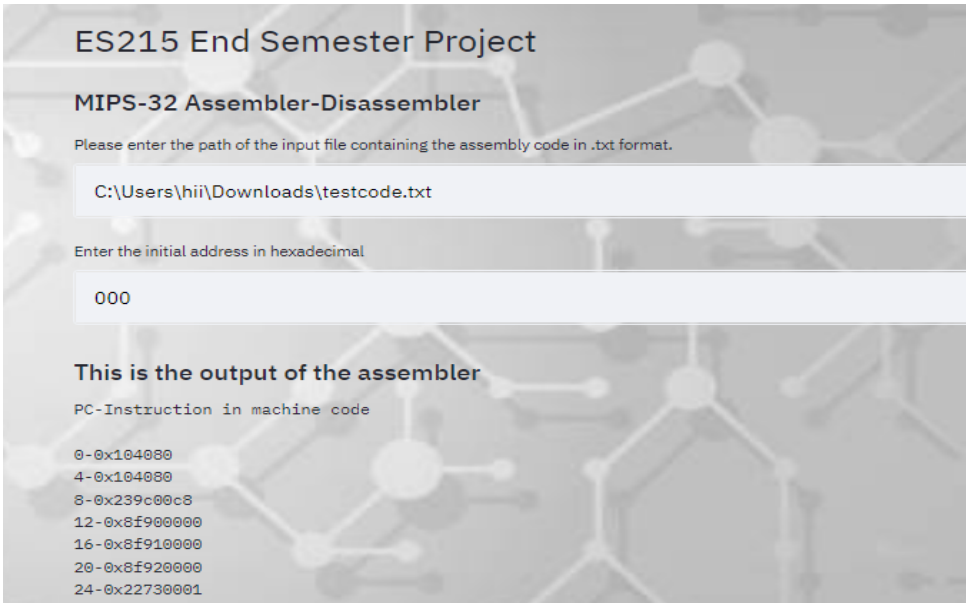


Fig. 5

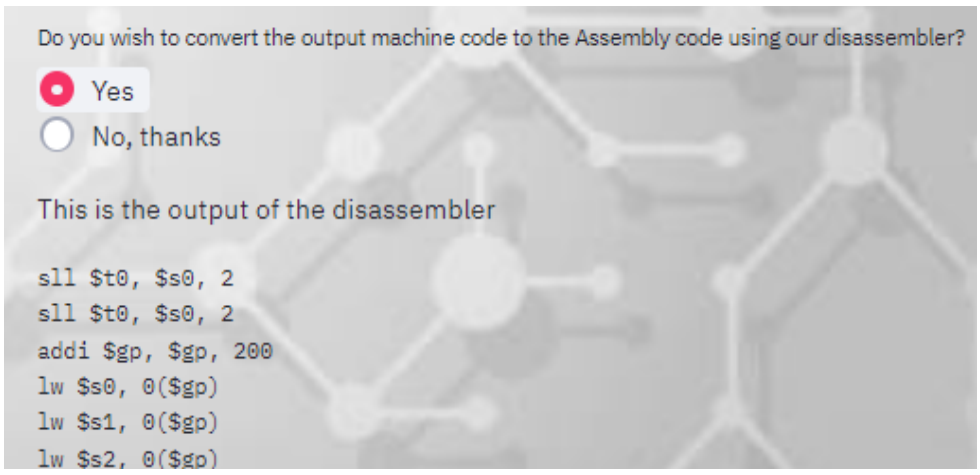


Fig. 6. A snippet of program asking user if he wants to disassemble the code

- A brief description of the code:
  The code file uploaded on Github is well commented and self-explanatory.

The Assembler:

All the instructions in the specified data card are represented by their mnemonics. These mnemonic codes work as keys to hold all the information, namely the opcode, instruction class, format of the input registers, and function codes(in R-type instruction) in a python dictionary. Similarly, we have created a dictionary specifying all the 32-registers.

```
instructions['lw']    = {'format': 'I', 'opcode': '23', 'num_operands': 0}
instructions['sw']    = {'format': 'I', 'opcode': '2b', 'num_operands': 0}
instructions['lh']    = {'format': 'I', 'opcode': '21', 'num_operands': 0}
instructions['lhu']   = {'format': 'I', 'opcode': '25', 'num_operands': 0}
instructions['sh']    = {'format': 'I', 'opcode': '29', 'num_operands': 0}
instructions['lb']    = {'format': 'I', 'opcode': '20', 'num_operands': 0}
instructions['lbu']   = {'format': 'I', 'opcode': '24', 'num_operands': 0}
instructions['sb']    = {'format': 'I', 'opcode': '28', 'num_operands': 0}
instructions['ll']    = {'format': 'I', 'opcode': '30', 'num_operands': 1}
instructions['sc']    = {'format': 'I', 'opcode': '38', 'num_operands': 0}
instructions['lui']   = {'format': 'I', 'opcode': 'f', 'num_operands': 4}
```

Fig 7: A snippet of our code displaying the specification of the instructions' dictionary

Using the pyparsing module, we have defined all the possible formats of all the instructions in the data card. All the instructions are iterated from the input file and then stored in a list named Mem. This imitates the memory of a computer. At last, all the instructions stored in the memory are translated according to the translation rules specified in the code and written in the output file.

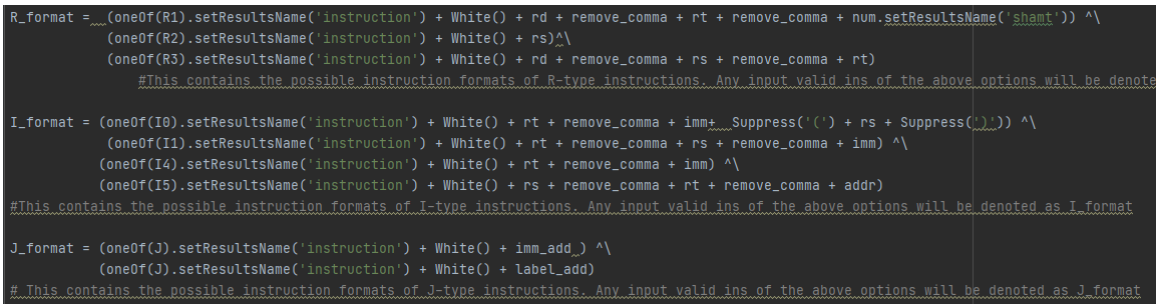


Fig 8: The specification of the format of R, I and J type instructions

The Disassembler:

Each feature of the instruction, namely, opcode, register name, and function codes (for R-type instructions), are defined according to the instructions dictionary made in the code for the assembler.

Then for each line in the Assembly code, we converted the assembly code to its corresponding MIPS code as explained using comments in the code. The addresses are sorted in the ascending order, and a hash table is made, which stores the addresses as keys, and the labels are values.

The assembly code with labels is printed with the label before the MIPS code, and the ones without labels are printed directly.

6. Testing and Experiments

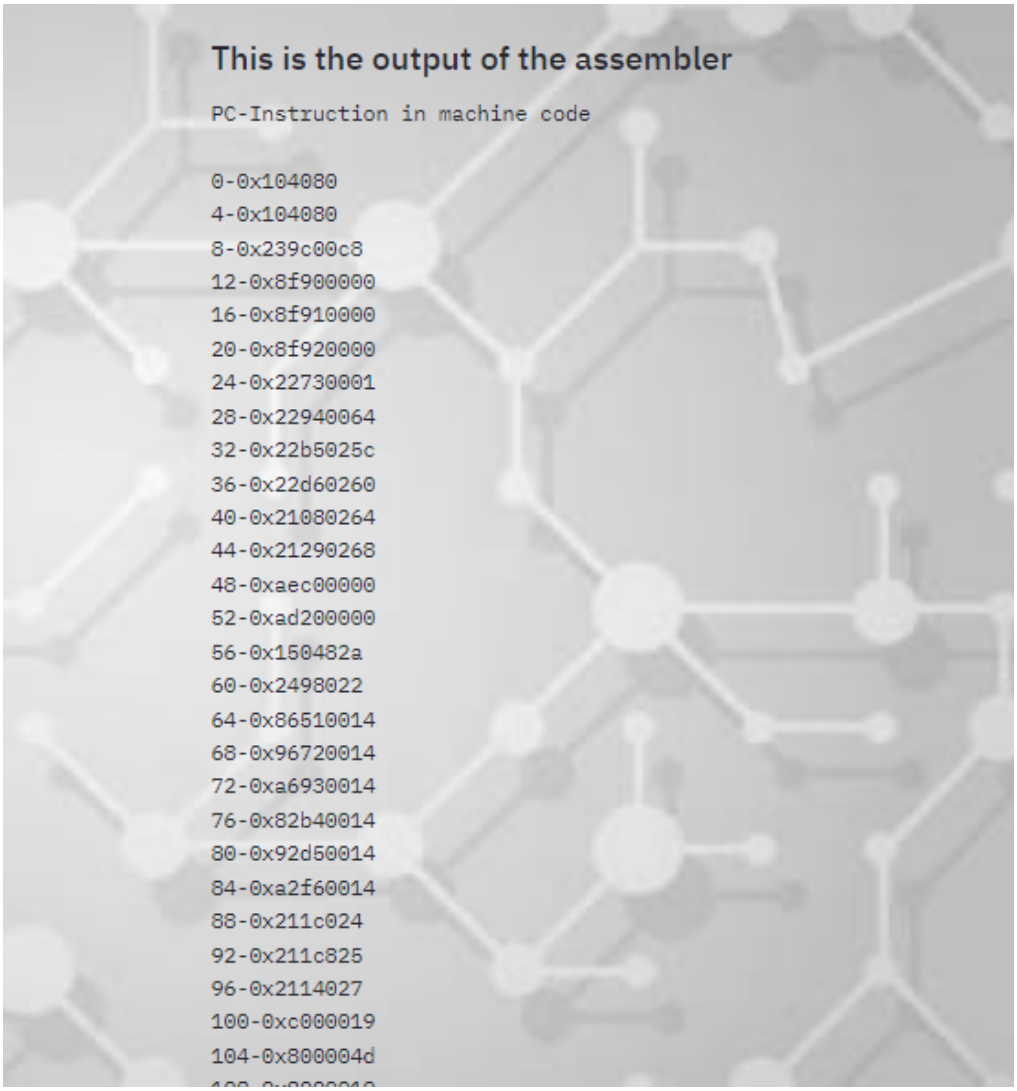
We used the MIPS code written for the Assignment 2 for testing and experimenting with the assembler and disassembler. Both the assembler and the disassembler were previously giving errors. We analyzed the errors and tried different methods to make the code efficient. To run the code on a broad spectrum of instructions used, we included all the instructions in the MIPS reference card in the testcode. We faced issues with some instructions like the sll, ll, etc. We added more features and made changes so that the code runs error-free for any kind of instruction. Thus, the codes for the assembler and the

disassembler were complete and could run for all types of instructions effectively.

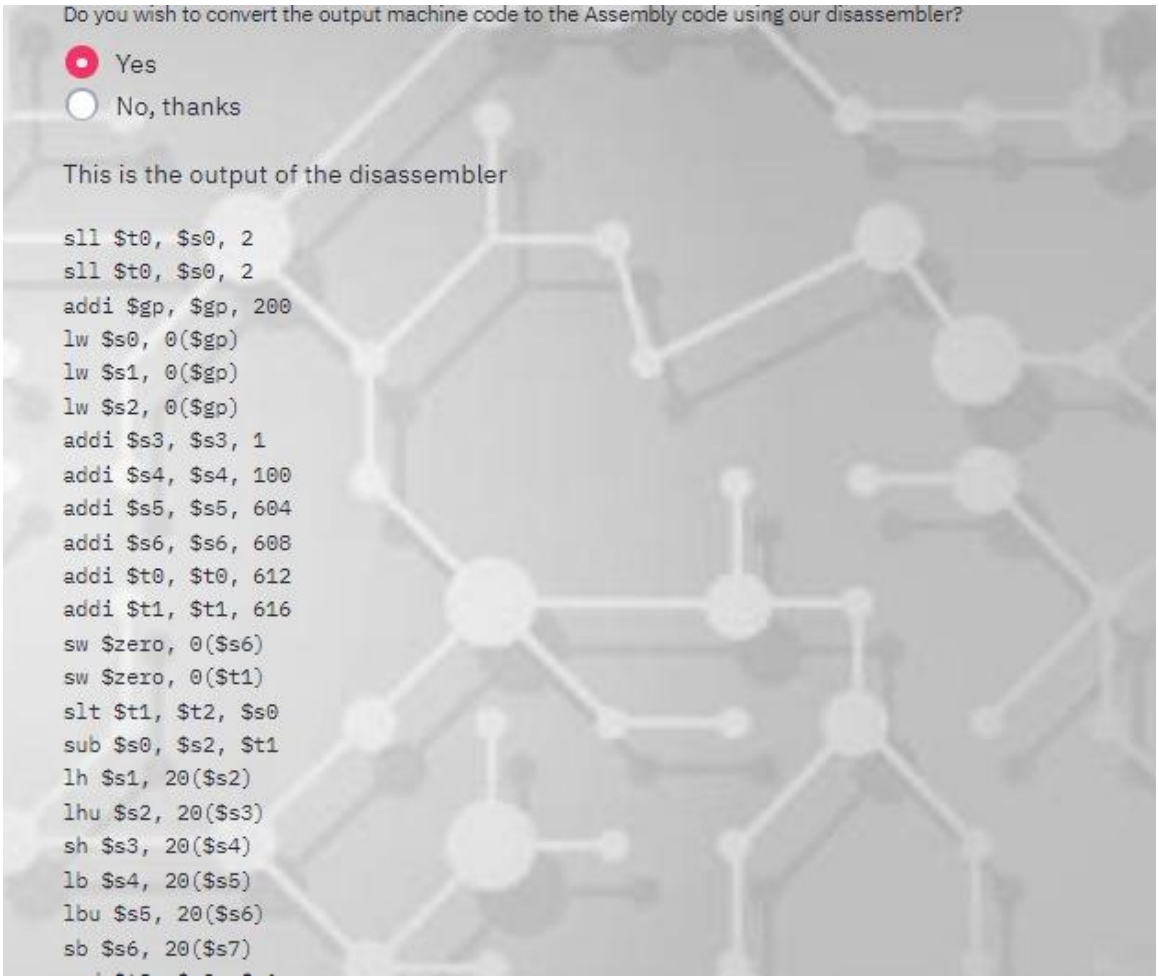
7. Data Analysis

The “testcode.txt” file is the combined file containing all the testcodes. We have run this file and arrived at the following results:

Output of the assembler code:



Output of the disassembler code:



Test code used is:

```
sll $t0, $s0, 2
sll $t0, $s0, 2
addi $gp, $gp, 200       # storing the address of first element of array in general purpose register
lw $s0, 0 ($gp)             # initializing a variable for max i.e. statement signifying int min=0
lw $s1, 0 ($gp)             # initializing a variable for min i.e. statement signifying int max=0
lw $s2, 0 ($gp)             # initializing a variable for max i.e. statement signifying int avg=0
addi $s3, $s3, 1            # initializing a variable i=1 to iterate in the loop
addi $s4, $s4, 100        # initializing a variable to store the number of times loop will run
addi $s5, $s5, 604       #storing the min variable at address 604 after the address of array
addi $s6, $s6, 608       #storing the min index variable at address 608 after the address of array
addi $t0, $t0, 612         #storing the max variable at address 612 after the address of array
addi $t1, $t1, 616         #storing the max index variable at address 616 after the address of array
sw $0, 0 ($s6)
sw $0, 0 ($t1)
slt $t1, $t2, $s0
sub $s0, $s2, $t1
lh $s1, 20($s2)
lhu $s2, 20($s3)
sh $s3, 20($s4)
lb $s4, 20($s5)
lbu $s5, 20($s6)
sb $s6, 20($s7)
and $t8, $s0, $s1
or $t9, $s0, $s1
nor $t0, $s0, $s1
jal 100
j min
j 100
```

As it is clear from the results shown above, all lines in the test code are matching with the lines that the assembler has printed, with the exception of labels in the test code (for, min) are printed as L1, L2, etc. since the labels are printed corresponding to the addresses in the address_table.

8. Conclusion

Our project includes a MIPS 32 assembler and disassembler. We were successfully able to execute the program for the distinct types of instructions mentioned in the test code. The project caters to a wide range of instructions and can handle various input formats smoothly.

The Git link containing the assembler-disassembler code and the test file: https://github.com/Lipika-Rajpal/ES215_endsem_Project/tree/main

Presentation Slides:

https://www.canva.com/design/DAE-zs8sbeU/lROeRz1_Ql46VW0z4DGYBQ/view?utm_content=DAE-zs8sbeU&utm_campaign=designshare&utm_medium=link&utm_source=publishsharelink

References

- Priya Pedamkar, “What is Assembly Language?” [Online] Accessed 26 April 2022 https://www.educba.com/what-is-assembly-language/
- MIPS32 Architecture [Online] Accessed 26 April 2022 https://www.mips.com/products/architectures/mips32-2/#:~:text=The%20MIPS32%20architecture%20provides%20seamless,support%20for%20past%20ISA%20versions
- Assembly language - Wikipedia [Online] Accessed 26 April 2022 https://en.wikipedia.org/wiki/Assembly_language#:~:text=The%20term%20%22assembler%22%20is%20generally,sections%20into%20a%20single%20program%22.
- Assembler - Technopedia [Online] Accessed 26 April 2022 https://www.techopedia.com/definition/3971/assembler
- https://github.com/A-Hemdan/mips-assembler-disassembler/blob/master/assembler.py
- https://en.wikipedia.org/wiki/MIPS_architecture
- https://www.google.com/search?q=pyparsing&oq=Pyparsing&aqs=chrome.0.69i59i512j0i512j69i59j0i512l3j69i60l2.1808j0j7&sourceid=chrome&ie=UTF-8
- https://pypi.org/project/bitarray/