ES 215 COMPUTER ORGANIZATION AND ARCHITECTURE
PROJECT REPORT

# ASSEMBLER AND DISASSEMBLER

April 26, 2022


**TEAM - THE QUAD CORE**

Bhavini Korthi 20110039
bhavini.korthi@iitgn.ac.in

Siva Sai Bommisetty 20110041
bommisetty.siva@iitgn.ac.in

S Sri Manish Goud 20110174
srimanishgoud_s@iitgn.ac.in

Voorugonda Rajesh 20110231
voorugonda.rajesh@iitgn.ac.in

Indian Institute of Technology, Gandhinagar


Github link for source code: https://github.com/BhaviniKorthi/QuadCore

## 1. Abstract

The Assembler and Disassembler project deals with the conversion of MIPS32 Assembly language code to the machine code and vice versa. MIPS stands for Microprocessor without Interlocked Pipeline Stages. We implemented MIPS assembler and disassembler in two python programs. The assembler program takes MIPS assembly language as input and converts it to machine code (both binary and hex code). The disassembler program takes either hex code or machine code as input and converts it to MIPS assembly code.
The assembler and disassembler are both designed for the following data sheet.
MIPS Datacard

## 2. Introduction

- Our project aims to build Assembler and Disassembler programs in high level language (Python 3.7 or higher) which converts the MIPS32 Assembly language code into Machine code (both binary and hexadecimal) and Machine code into the Assembly language code respectively.
- The project has many advantages, one of which is to get familiarize with the low-level languages, and multiple instructions of the MIPS32 assembly language. The other trivial advantage is that while designing assembler, and disassembler, we can explore the features of high-level language (Python).
- The Assembler and Disassembler acts as a handy learning tool for learning MIPS32 architecture.
- Our project is a **new prototype** of Assembler and Disassembler which is entirely based on our own ideas.
  For Assembler: accepting both the commonly used syntaxes - (add $s1 $s2 $s3 and add $s1, $s2, $s3), and it takes care of any unnecessary white spaces and blank lines in the input, etc.
  For Disassembler: Our model of disassembler accepts both hexadecimal and binary forms of input, and in the output assembly code of the disassembler every instruction is labeled with a unique name as label x, where x-1 denotes the offset of the instruction from the base instruction, etc
  Let us discuss more in the later part of the report.

## 3. Literature Review

- There were misconceptions about the offset values, which was resolved by approaching the instructor and referring to the MIPS manual[2]. We used the following MIPS data card for this project. https://drive.google.com/file/d/12D9Cc7gNEaCQx-kvbDDpZUh9A4gsRk1v/view?usp=sharing

- There were difficulties in implementing the symbol table, detecting the labels in the input, and some basic things about Python language syntax and inbuilt functions. We referred to Geeks for Geeks for the same[3].
- Misconceptions regarding MIPS instructions' syntax were dealt with by referring to Minnesota University's official site[4].

## 4. Project idea

The project contains two main parts: Assembler and Disassembler for the MIPS32 architecture. The Assembler is supposed to convert the given input MIPS assembly code into the corresponding machine language and display the same. Similarly, the disassembler converts the given machine code to the corresponding assembly code and displays it on the screen.

The Assembler and Disassembler in our project are designed using Python language.

Assembler:

The Assembler that we have designed is a two pass assembler. The first pass is used to run over each and every line in the given text file. During the first pass, all the blank lines are removed and a symbol table is created. The symbol table contains all the labels and their corresponding addresses. After organizing the input, the second pass is done which includes the identification of the instruction format, instruction type and finally the operations based on the opcode values and function values.

Disassembler:

Incase of the disassembler, both the passes are implemented. All the instructions of the binary code are assigned a unique label name. These label names are used in the code to represent offset incase of conditional branches and address incase of jump instructions.

**5. Project Implementation**

Since the project contains both Assembler and disassembler we will discuss their implementations individually.

**ASSEMBLER**:

The Assembler that we have designed is a two pass assembler.

It takes the input from the **text file** that we provide during the beginning of its execution.

**Pass 1**:

- In our project, Pass 1 is done for the purpose of building the **symbol table** and allocating the address for each instruction.
- During the first pass, we will be going through each line of the input code and detect all the labels. When a new label is detected it is added to the symbol (label) table along with its address. We also make an array that holds the corresponding address as we move through the input code.
- Also, the first pass is used for the purpose of removing all the unnecessary white spaces and blank lines.

Suppose consider the following input code:

```
add $s1, $s2, $s3
j done
addi $t1 $t2 5
lw $t1 100($s1)
done : addi $s1 $s2 100
abc: sw $s1 150($s2)
```

Here, in Pass 1, we will be going through all the instructions and detect all the unnecessary black lines and white spaces. Then we will be recording all the instructions into an array, and their corresponding addresses into another array

The detection of a label is done using the sign, ':'. When we detect this sign, we separate the instruction from the label and record the label in the symbol table. The symbol table is in the form of a dictionary, where the labels are the keys and the addresses are the corresponding key values.

For the above code (with 1000 as the base address) the following is the result after Pass 1.

| Array representing the address | Array containing the detected instructions |
|---|---|
| 1000 | add $s1, $s2, $s3 |
| 1004 | j done |
| 1008 | addi $t1 $t2 5 |
| 1012 | lw $t1 100($s1) |
| 1016 | addi $s1 $s2 100 |
| 1020 | sw $s1 150($s2) |

**Label table**:

| Label | Address |
|---|---|
| done | 1004 |
| abc | 1020 |

**<u>Pass 2</u>**:

In the second Pass, we will be going through each line of the instruction that we have appended in the instruction array.

In our project, we have divided the MIPS instructions into three formats, I, J, and K, which are further divided into different types depending on their structure.

Following are all the classifications that we have made to perform pass 2:

**R-type:**

- Type 1: **oper rs, rd, rt**
- Type 2: **oper rs, rd, sa**
- Type 3: **oper rs** (oper- operation, sa - shift amount)

**I-type:**

- Type 1: **oper rs, rt, imm**
- Type 2: **oper rs, rt, label**
- Type 3: **oper rt, imm(rs)**
- Type 4: **oper rt, imm**

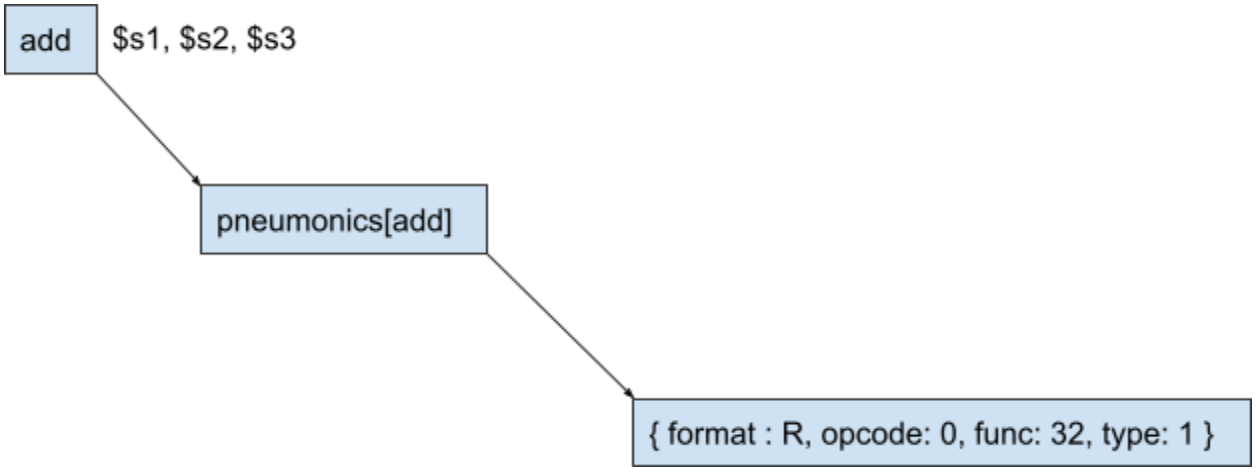
**J-type:**

- Type 1: **oper label**

The above classifications are implemented in the code using a dictionary called '**pnemonics'.** In this dictionary, we store all the corresponding numeric values for each part of the instruction like the opcode, register numbers, function values, shift amount, etc. depending on the instruction type and format.

We will be accessing the required numeric values from the dictionary as follows:

For add $s1 $s2 $s3:




Using this dictionary, the format and type of the instruction is identified. Hence, we then frame a structure for the binary code for that specific type of instruction using the value of '**type**' in the above dictionary. We will be accessing the values for each registers through a dictionary named registers. A clear idea of this process can be obtained if we observe the code.

**Jump instructions**:

When we detect a jump instruction or the instructions which involve labels (bne, beq, etc.), we will be accessing the addresses of the labels from the symbol table, and construct the binary code accordingly. Please refer to the code to get a better understanding of this process.

Finally, all the binary codes for each line that are stored in an array are displayed on the output console.

**DISASSEMBLER**:

Similar to the Assembler, the input code for the Disassembler is taken from the mentioned input code. For the disassembler, we will be using two passes.

**Pass 1**:

In this pass, we assign labels for each instruction of the input code. Also, an array containing the addresses for each line is made. For the disassembler, there must not be any kind of blank lines or unnecessary spaces, and hence, we would not be detecting the blank lines in this case.

For example, for a set of three instructions, the following will be the result after Pass 1:

| Address array | array of Instructions | Array for lables |
|---|---|---|
| 1000 | 0000001010110100 | Label-1 |
| 1004 | 0000101011101011 | Label-2 |
| 1008 | 0001011101010001 | Label-3 |

**Pass 2**:

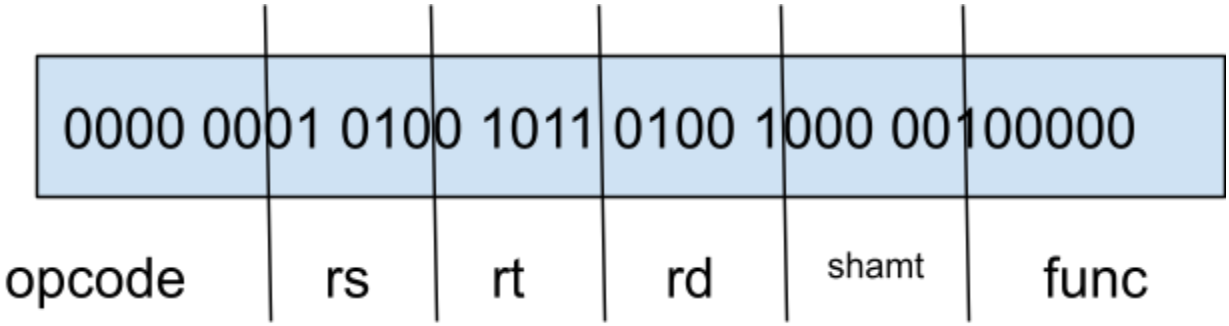
In this pass, we will be converting the binary code into the corresponding MIPS instructions. For this, the MIPS instructions are divided into different formats and types exactly similar to the classification that we have made in case of assembler.

A dictionary called reg_rev is used in which the register values are stored as the keys and the registers are stored as the values of the corresponding register value keys.

Among the different instruction formats, R type is the only format in which we couldn't identify the operation with the opcode itself. All the R-type instructions had opcode as zero and needed a value called function value in order to identify the operation. So, a dictionary R_fun is defined

to store the functions values and the type required to identify a particular operation.

In case of J and I formats, opcode alone is enough to identify the operation. The dictionaries J_op and I_op are defined to store the opcodes and types to identify the operations belonging to J and I formats.




Based on the above division, the given binary code is classified into opcode, rs, rt, rd, sa, shamt, function value, offset, and address. The classification differs with the format and type. So, the classification follows the order format → type → operation.

**Difficulties faced and their solutions:**

- In the beginning, there was no idea regarding the implementation of the symbol table. After referring to a few online sources we came up with an idea of using dictionaries for its implementation.
- The major difficulty is the offset and address limitations. Depending upon the base address and the length of the code only a certain range of offset and address values are allowed. Inorder to make the algorithm adhere to this range, we have come up with a few techniques for displaying errors.

**Special features of our project:**

- In case of assembler the instruction that corresponds to a particular label can be written either immediately after the colon of the label or in the next line. Both kinds of formats are accepted.
- The indentation for the instructions does not give any kind of error in case of the assembler
- In case of disassembler the input machine code can be given in both forms: binary and hexadecimal form.

- For the jump instructions in disassembler, arbitrary labels are displayed in the output console instead of numerical offset and address values.

**6. Testing and Experiments**

**i. For Assembler:**

To check the optimality and working of Assembler, we have inputted all the instructions given in the MIPS data card. We did not get the expected output (correct machine code) at the first go. We have had to reexamine, rewrite our code several times. In some cases, we input the wrong assembly code. After many unsuccessful attempts, we made sure our Assembler worked for any combinations of instructions. We used CPUlator, an online MIPS simulator to check if the output machine code is correct. They came to be almost the same. There were small differences like change in order, which is based on CPUlator design. The following are the results.

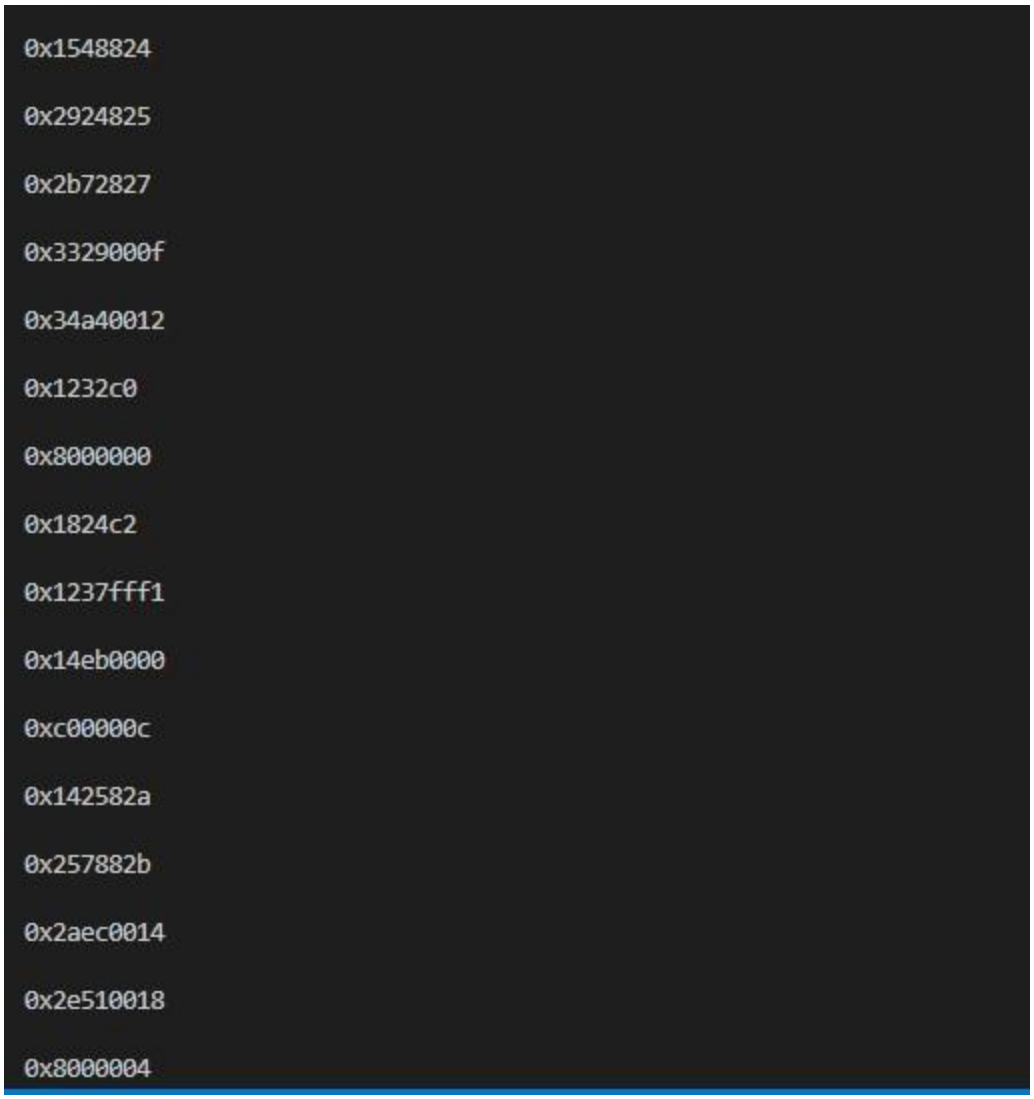
**CPUlator Results:**

```
00000000  02538820  1   add     $s1, $s2, $s3
00000004  02538822  2   sub     $s1, $s2, $s3
00000008  22510014  3   addi    $s1, $s2, 20    ; 0x14
0000000c  8e560010  4   lw      $s6, 16($s2)
                    5   done2:
                    done2:
00000010  aef10014  6   sw      $s1, 20($s7)
00000014  84190014  7   lh      $t9, 20($zero)
00000018  96710014  8   lhu     $s1, 20($s3)
0000001c  a655000c  9   sh      $s5, 12($s2)
00000020  82f10014  10  lb      $s1, 20($s7)
                    11  done: lbu $s1, 4($t9)
                    done:
00000024  93310004      lbu     $s1, 4($t9)
00000028  a1290018  12  sb      $t1, 24($t1)
0000002c  c1130014  13  ll      $s3, 20($t0)
                    14  done1: sc $s1, 16($t3)
                    done1:
00000030  03e00008  15  jr      $ra
00000034  e1710010      sc      $s1, 16($t3)
00000038  3c110014  16  lui     $s1, 20 ; 0x14
0000003c  01548824  17  and     $s1, $t2, $s4
00000040  02924825  18  or      $t1, $s4, $s2
00000044  02b72827  19  nor     $a1, $s5, $s7
00000048  3329000f  20  andi    $t1, $t9, 15    ; 0xf
```

```
00000044  02b72827  19  nor     $a1, $s5, $s7
00000048  3329000f  20  andi    $t1, $t9, 15    ; 0xf
0000004c  34a40012  21  ori     $a0, $a1, 18    ; 0x12
                    22  sll $a2, $s2, 11
                    23  j 0
00000050  08000000      j       0x0
00000054  001232c0      sll     $a2, $s2, 11
                    24  srl $a0, $t8, 19
                    25  beq $s1,$s7, done
00000058  1237fff2      beq     $s1, $s7, 0x24  (0x24: done)
0000005c  001824c2      srl     $a0, $t8, 19
                    26  bne $a3, $t3, 0
00000060  14ebffe7      bne     $a3, $t3, 0x0
00000064  00000000      nop
                    27  jal done1
00000068  0c00000c      jal     0x30  (0x30: done1)
```

**Assembler Results:**

```
Hexadecimal code

0x2538820

0x2538822

0x22510014

0x8e560010

0xaef10014

0x84190014

0x96710014

0xa655000c

0x82f10014

0x93310004

0xa1290018

0xc1130014

0xe1710010

0x3e00008

0x3c110014
```




**ii. For Disassembler:**

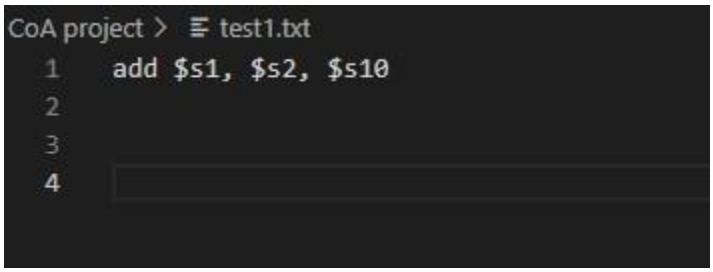
To check the optimality and working of the disassembler, we have had to iterate for many test cases, and finally we achieved the correct results. Below are the results when the output of the Assembler is given as input for the Disassembler. The input of the Assembler and the output of the Disassembler came to be exactly the same.

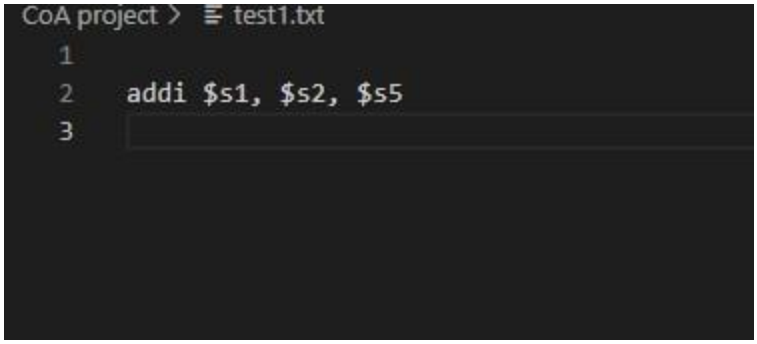
```
Lable-0 : add $s1, $s2, $s3
Lable-1 : sub $s1, $s2, $s3
Lable-2 : addi $s1, $s2, 20
Lable-3 : lw $s6, 16($s2)
Lable-4 : sw $s1, 20($s7)
Lable-5 : lh $t9, 20($0)
Lable-6 : lhu $s1, $s3, 20
Lable-7 : sh $s5, 12($s2)
Lable-8 : lb $s1, 20($s7)
Lable-9 : lbu $s1, 4($t9)
Lable-10 : sb $t1, 24($t1)
Lable-11 : ll $s3, 20($t0)
Lable-12 : sc $s1, 16($t3)
Lable-13 : jr $ra
Lable-14 : lui $s1, 20
Lable-15 : and $s1, $t2, $s4
Lable-16 : or $t1, $s4, $s2
Lable-17 : nor $a1, $s5, $s7
Lable-18 : andi $t1, $t9, 15
Lable-19 : ori $a0, $a1, 18
Lable-20 : sll $a2, $s2, 11
Lable-21 : j Lable-0
Lable-22 : srl $a0, $t8, 19
Lable-23 : beq $s1, $s7, Lable-9
Lable-24 : bne $a3, $t3, Lable-25
Lable-25 : jal Lable-12
Lable-26 : slt $t3, $t2, $v0
Lable-27 : sltu $s1, $s2, $s7
Lable-28 : slti $t4, $s7, 20
Lable-29 : sltiu $s1, $s2, 24
Lable-30 : j Lable-4
```

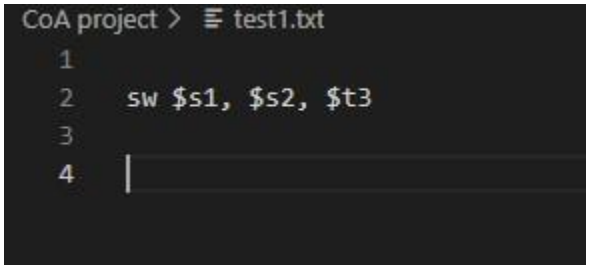
Thus, we have made sure our Assembler and Disassembler work for the correct inputs. If Assembler and Diassembler are working efficiently, then they should output an error for the wrong inputs.

So, let us try for the wrong inputs.

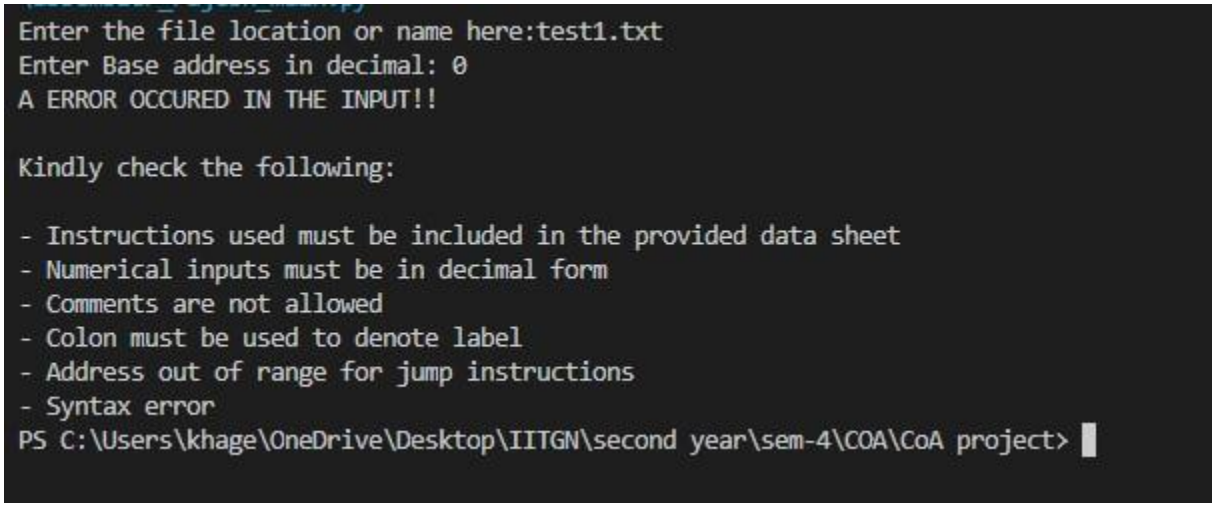
**iii. Wrong Inputs for Assembler:**



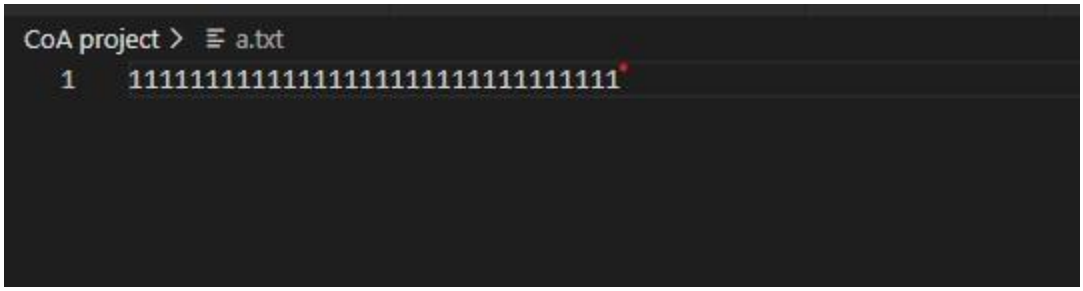


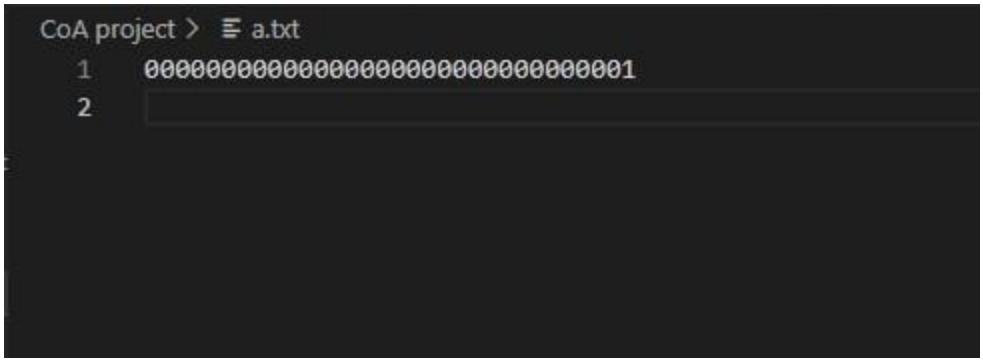



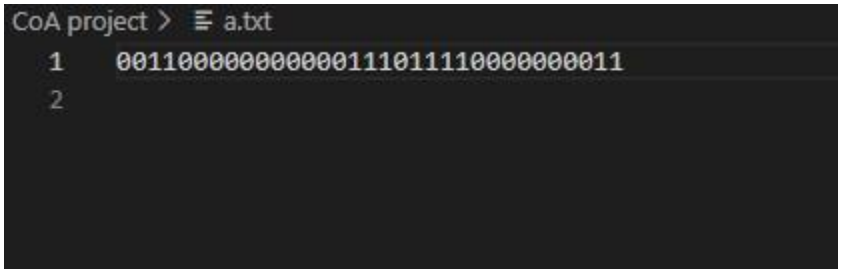

For the above three assembly codes, the result shows error for all. The following is the result.



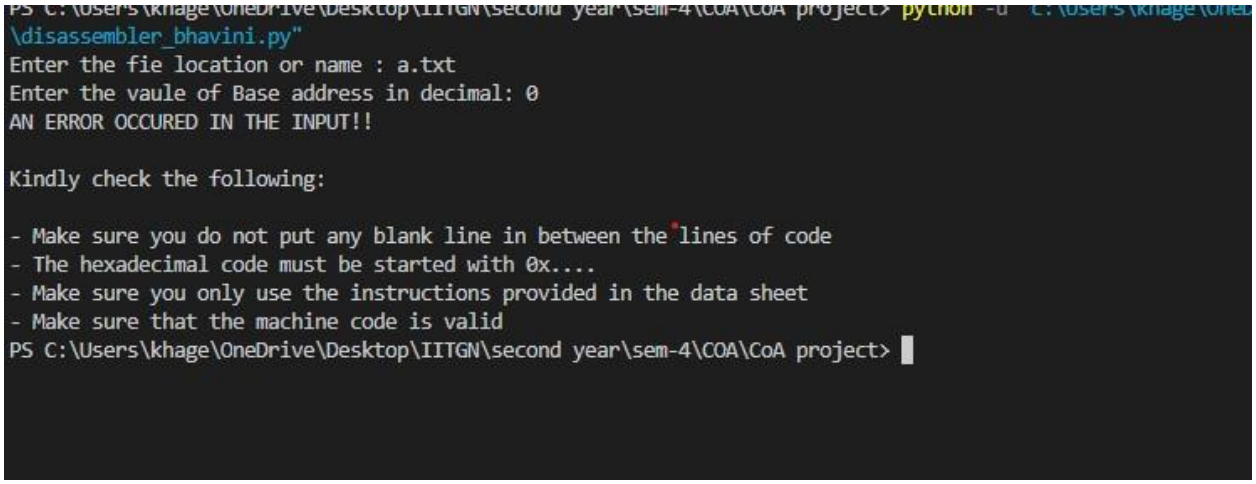

**iv. Wrong inputs for Diassembler:**






For the above three machine code inputs, the result is the same error for all. The following is the result.




Based on the several test cases, and trying out different combinations of the MIPS data card instructions, we claim our Assembler and Diasssembler should be correct. Our Assembler and Disassembler output the error for the wrong inputs. So, from the results of various testings and experiments, we can say Assembler and Disassembler worked efficiently, and will work for any other inputs as specified in the readme file attached.

## 7. Conclusion

We have now implemented the MIPS32 assembler and disassembler. The main idea which we used in this process is parsing through the input code, twice in both the cases. For assembler, in the first pass we create symbol tables and remove the black lines and in the second pass we use those to convert the assembly code to the machine code and for the disassembler, in the first pass we assign labels to each line of the input code and in the second pass we use those labels to convert the machine code to the assembly code.

**Future aspects:**

- The current project focuses on limited MIPS instructions specified in the data card. However, we can extend this to include all the instructions in the MIPS32 architecture.
- Further, we can work on optimizing the code to get faster results.
- When a fallacious input assembly code is provided, the improvised project should be able to deal with the errors, and output the information regarding the error line, why the error occurred, and correction guidelines, etc.

**Work distribution:**

As there are two parts in the project (Assembler and Disassembler) we divided the work equally, that is, we splitted our team into two groups where one group worked on assembler and the other worked on disassembler. However, we discussed with each other if any doubts were raised while working. Later we explained each group's work to the other group.

The second group worked on preparing the slides for presentation, collecting the codes for testing and experimenting. Video presentation and its editing is done by the group 1 members. However, we worked collectively on debugging the codes and report writing.

Group1: Bhavini Korthi, Voorugonda Rajesh
Group2: S Sri Manish Goud, Siva Sai Bommisetty
Individual Contributions:

Bhavini Korthi - Assembler, Video presentation and editing, report writing, debugging
Voorugonda Rajesh - Assembler, Video presentation and editing, report writing, debugging
S Sri Manish Goud - Disassembler, presentation slides, preparing codes for testing, report writing, debugging.
Siva Sai Bommisetty - Disassembler, presentation slides, preparing codes for testing, report writing, debugging.

**References:**


[1] David A. Patterson, John L. Hennessy, *Computer Organization and Design, Edition 5, The Hardware/Software Interface*. Morgan Kaufmann, 2014.

[2] MIPS Technologies, Inc, "MIPS32™ Architecture For Programmers Volume I: Introduction to the MIPS32™ Architecture" [Online]. Available: https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol1.pdf
MIPS Technologies, Inc, "MIPS32™ Architecture For Programmers Volume II: Introduction to the MIPS32™ Architecture" [Online]. Available: https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol2.pdf
MIPS Technologies, Inc, "MIPS32™ Architecture For Programmers Volume III: Introduction to the MIPS32™ Architecture" [Online]. Available: https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol3.pdf

[3] [Online]. Available: https://www.geeksforgeeks.org/

[4] Minnesota University's official site, [Online]. Available: https://www.d.umn.edu/~gshute/mips/itype.xhtml
Minnesota University's official site, [Online]. Available: https://www.d.umn.edu/~gshute/mips/rtype.xhtml
Minnesota University's official site, [Online]. Available: https://www.d.umn.edu/~gshute/mips/jtype.xhtml