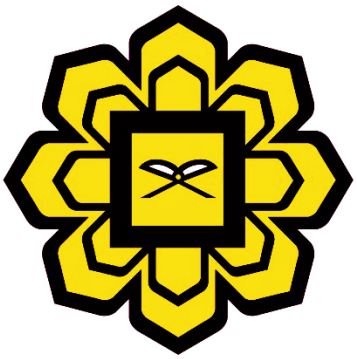


الجامعة الإسلامية العالمية ماليزيا
INTERNATIONAL ISLAMIC UNIVERSITY MALAYSIA
يونيرسيتي إسلام انتارابغسا ملسيا

CSC 3402 –Computer Architecture & Assembly Language

Semester 1 2017/18
Section: [2]

Project Report

Project Specification (Part 1) – Encoder
Project Specification (Part 2) – Calculator
Project Name: "Tiny Calculator"

Group Members

S M Raju – **1416463**
Md. Shariful Islam – **1412813**
Ali Mohammad Tarif – **1418735**
Mohammad Mostafizur Rahman – **1333513**

Supervised by

DR. HAFIZAH BINTI MANSOR

Submission Date: 23/12/2017

Table of Contents:

Part-1

Introduction

An assembly language, often abbreviated asm is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the language and the architecture's machine code instructions. Each assembly language is specific to computer architecture. In contrast, most high-level Programming languages are generally portable across multiple architectures but require interpreting or compiling. Assembly language may also be called symbolic machine code. Assembly language is converted into executable machine code by a utility program referred to as an assembler. The conversion process is referred to as assembly, or assembling the source code. Assembly time is the computational step where an assembler is run. Assembly language uses a mnemonic to represent each low-level machine instruction or opcode, typically also each architectural register, flag, etc.[1]

We used Python Programming Language to develop the encoder for converting asm file into hex file. In this assembler, all the registers (32 registers) and 3 types of instructions (R, I & J) are included. When user write the valid instruction, it will show Function type, Formatted binary, Hexadecimal(Hex) number and Binary number. The process will continue until user type "exit".

General Instructions

The encoder can run the following subset of the MIPS instruction set in the format described.

- add $rd $rs $rt [immediate value can also be given instead of $rs & $rt]
- sub $rd $rs $rt [immediate value can also be given instead of $rs & $rt]
- and $rd $rs $rt [immediate value can also be given instead of $rs & $rt]
- or $rd $rs $rt [immediate value can also be given instead of $rs & $rt]
- slt $rd $rs $rt [immediate value can also be given instead of $rs & $rt]
- move $rd $rs
- li $rd immediate_value
- lw $reg var_name

[1] https://en.wikipedia.org/wiki/Assembly_language

- sw $reg var_name
- beq $rs $rt label
- bne $rs $rt label
- j label

And many more instructions set can run on our encoder.

General Overview

The program works as follows: -

- Read instructions from input file
- Convert instructions into opcodes
- Load the converted instructions into the instruction memory
- Execute: -
  - Start from the first instruction
  - Encode and perform necessary operations
  - Proceed sequentially unless jump/beq encountered.

Screenshot of encoder program

```
                    WELCOME TO THE MIPS ENCODER!
                    Type MIPS code below to see it in binary and hex form
                    Syntax: If using hex, use the '0x' label
                    Type 'exit' to exit
--------------------------------------------------------------------------------

Type MIPS code here: add $t1 $t2 $t3

Function type: R-Type
Instruction form: opcode|  rs |  rt |  rd |shamt| funct
Formatted binary: 000000|01010|01011|01001|00000|100000
Binary:           00000001010010110100100000100000
Hex:              0x014b4820
--------------------------------------------------------------------------------

Type MIPS code here: lw $t1 100 $s2

Function type: I-Type
Instruction form: opcode|  rs |  rt |   immediate
Formatted binary: 100011|10010|01001|0000000001100100
Binary:           10001110010010010000000001100100
Hex:              0x8e490064
--------------------------------------------------------------------------------

Type MIPS code here: exit


Process finished with exit code 0
```

Technical Documentation

Instruction Memory

The instruction memory is implemented as a structure holding the opcodes for each instruction, which is then used to perform the appropriate functions on execution.

Data Memory

The data memory is implemented as a structure too. It holds the name of the variable and the corresponding value.

Labels (for jump and beq)

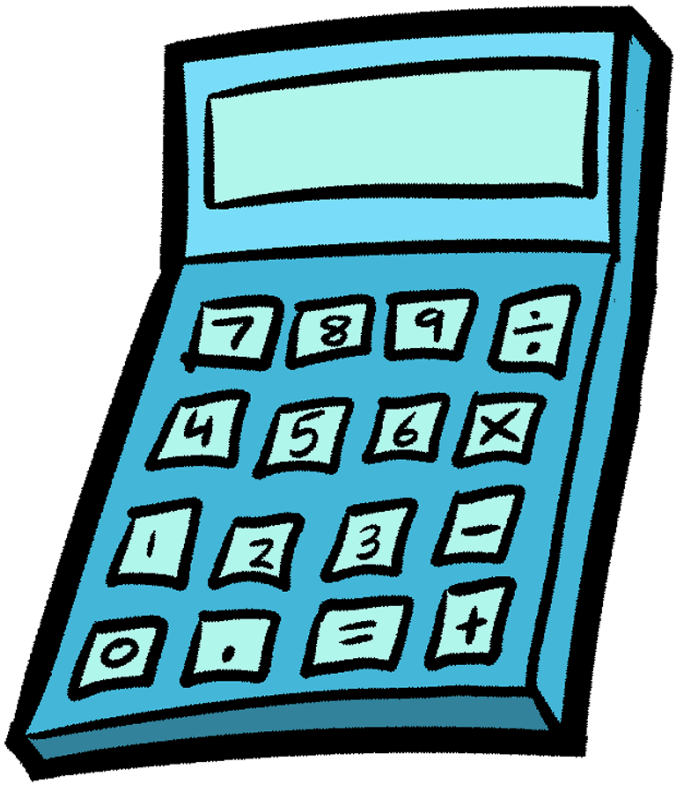
The labels are stored in an array of structure of a string and and integer. The integer holds the instruction number the label points to. To make this a 'one-pass' assembler instead of a 'two-pass'; if a label is encountered in a jump statement (or beq statement), then the label is stored in the array with the instruction number variable set to -1. Once the label is found, the value storing the instruction number is appropriately set.

Conclusion

The MIPS encoding system identifies three major classes: R-type, I-type, and J-type. We develop the encoder to convert MIPS instruction set into hex file also binary form and number. However, the encoder performs well when the valid statement input by the user.

Part 2

Tiny Calculator



Why “Tiny Calculator”?

- We named it Tiny Calculator because it can operate only Addition (+), Subtraction (-), Multiplication (*) and Division (/) with the remainder (%).

Introduction

Many operations require one or more operands in order to form a complete instruction and most assembler can take expressions of numbers and named constants as well as registers and labels as operands, freeing the programmer from tedious repetitive calculations. Depending on architecture, these elements are also being combined for specific instructions or addressing mode using offsets or other data as well as fixed addresses. Many assemblers offer additional mechanisms to facilitate program development to control the assembly process, and to aid debugging.

Background

In this project we used some syntax and they are shortly described below:

li $v0,4

la $a0, newline

- Above code is used for printing newline. Also entering number, welcome message by changing newline the code.

li $v0,5

- For getting input from the user.

move $s0, $v0

- For moving number from user out of v0 to s0.

move $t1, $v0

- For moving number from user out of v0 to t1.

li $v0,4

la $a0,sum

- For printing sum string.

```
li $v0,1
move $a0,$t0
```

- Printing user first entered number.

```
li $v0,4
la $a0,comma
```

- Printing comma (,) between two entered number.

```
li $v0,1
move $a0,$t1
```

- Printing user second entered number.

```
add $t2,$t0,$t1
```

- For performing addition.

```
li $v0,1
move $a0,$t2
```

- For printing user sum of numbers.

```
li $v0,4
la $a0,difference
```

- For printing difference string.

```
neg $t3,$t1
add $t2,$t0,$t3
li $v0,1
move $a0,$t2
```

- Performing subtraction and printing result.

```
li $v0,4
la $a0,product
```

- To print the multiplication string.

```
mul $t2,$t0,$t1
li $v0,1
move $a0,$t2
```

- Performing multiplication from two number and print.

```
li $v0,4
la $a0,quotient
```

- Print the quotient string.

```
 div $t0,$t1
mflo $t6
mfhi $t7
```

- Perform division and print.

```
li $v0,4
la $a0,remainder
```

- Print the remainder string.

```
li $v0, 10
```

- Exit from the sequence.

```
j loop
```

- Jump to the loop and continue the process.

Project Description

At the very first of our program it will show the options to be calculated.

```
                    Welcome to Tiny Calculator
                            1 - Addition
                            2 - Subtraction
                            3 - Multiplication
                            4 - Division
                            0 - Exit
        Choose an option:
```

If the input is valid then it will go to the next step and if it is not, then it will show invalid number and exiting from the program.

```
                    Welcome to Tiny Calculator
                            1 - Addition
                            2 - Subtraction
                            3 - Multiplication
                            4 - Division
                            0 - Exit
        Choose an option: 5
        Invalid number, exiting...
-- program is finished running --
```

If the input is valid then the next step it will do the operations that user wants to operate.

- If user input 1 it will take input from user and will do the addition of two entered numbers. And the result is distributed in Decimal.

- If user input 2 it will take input from user and will do the subtraction of two entered numbers. And the result is distributed in Decimal.

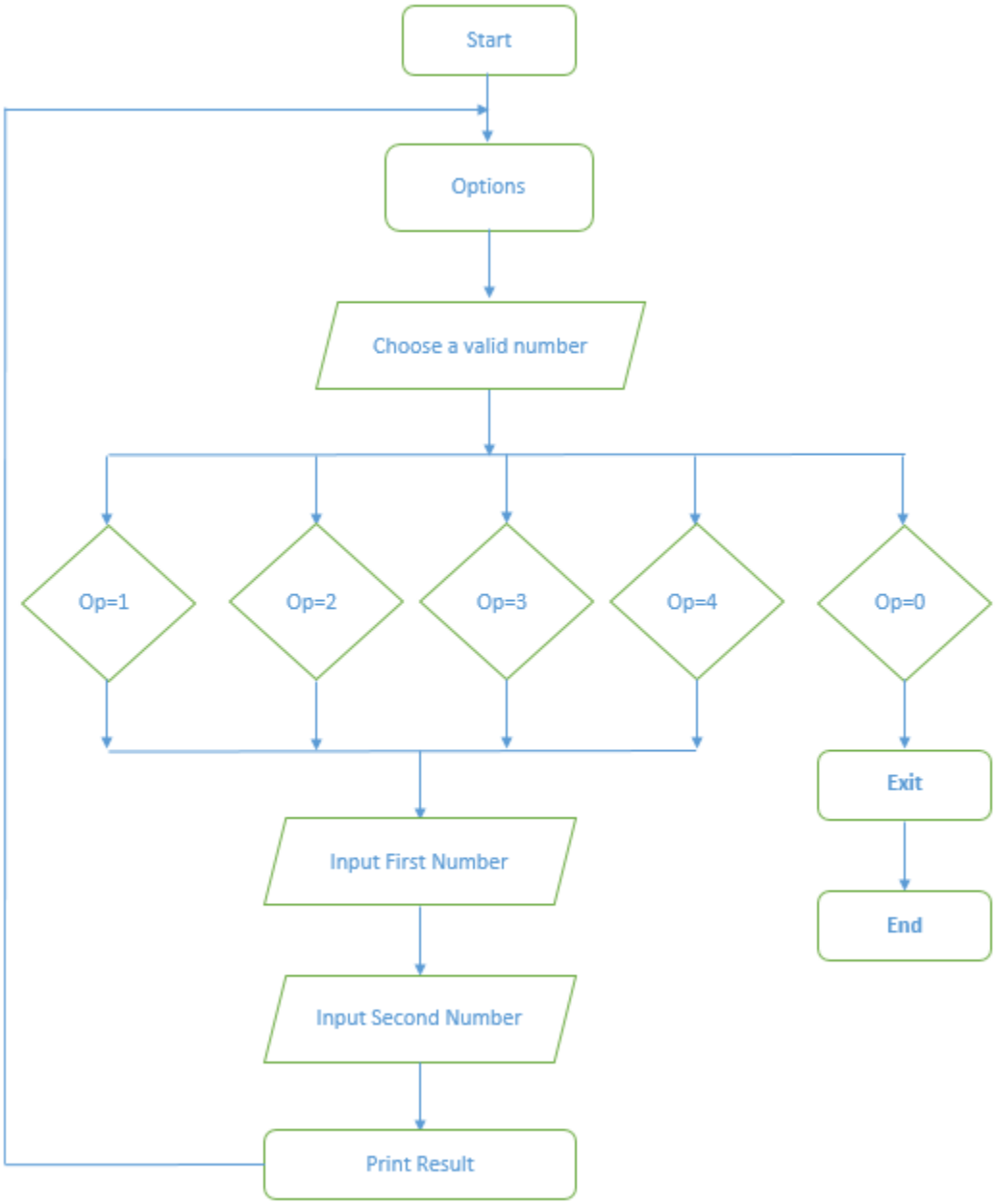
- If user input 3 it will take input from user and will do the multiplication of two entered numbers. And the result is distributed in Decimal.

- If user input 4 it will take input from user and will do the division of two entered numbers. In this case, it will also show the remainder of the numbers after divided. And the result is distributed in Decimal.

After every operation it will take your opinion that either you want to exit, or you want further calculation. If you want to exit, then it will simply exit by entering 0. Otherwise it will start from the beginning of the program.

That was the description of our project.

Flowchart


Start
Options
Choose a valid number
Op=1
Op=2
Op=3
Op=4
Op=0
Exit
End
Input First Number
Input Second Number
Print Result

Experimental Result

Here is the snapshot of Addition, Subtraction, Multiplication and Division accordingly.

```
                                Welcome to Tiny Calculator
                                        1 - Addition
                                        2 - Subtraction
                                        3 - Multiplication
                                        4 - Division
                                        0 - Exit
Choose an option: 1
Please enter first number: 2
Please enter second number: 5
The addition of 2 , 5 is = 7


                                Welcome to Tiny Calculator
                                        1 - Addition
                                        2 - Subtraction
                                        3 - Multiplication
                                        4 - Division
                                        0 - Exit
Choose an option: 2
Please enter first number: 4
Please enter second number: 7
The substraction of 4 , 7 is = -3

                                Welcome to Tiny Calculator
                                        1 - Addition
                                        2 - Subtraction
                                        3 - Multiplication
                                        4 - Division
                                        0 - Exit
Choose an option: 3
Please enter first number: 4
Please enter second number: 6
The multiplication of 4 , 6 is = 24

                                Welcome to Tiny Calculator
                                        1 - Addition
                                        2 - Subtraction
                                        3 - Multiplication
                                        4 - Division
                                        0 - Exit
Choose an option: 4
Please enter first number: 4
Please enter second number: 4
The quotient of 4 , 4 is = 1
And the remainder is: 0
```

Advantage

- It can perform Addition.
- It can perform Subtraction.
- It can perform Division with the remainder.
- It can perform Multiplication.
- It can distribute the result into Decimal.
- It can calculate any integer digit number.
- It can check inputs whether it is valid or invalid.
- It can calculate with 4 functions until user input 0 to exit.
- It can calculate signed value. (e.g. +2 and -2 =0)

Disadvantage

- It can't divide 0.
- It can't produce binary and hexadecimal result.
- After inputting invalid number of options, it will exit program.

Conclusion

Assembly language still taught in most computer science and electronic engineering programs. Although few programmers today regularly work with assembly language as a tool, the underlying concepts remain very important.

Our calculator can calculate with big values. Despite having some limitations, we can get the concept of more perfect programs with this.

Appendix

Part-1(source code)

Instruction_encoder.py

```
# converts the instruction part of a line of MIPS code
# param 'instr' is the instruction given
# returns an arrya in the form [function type, opcode, function number]
def instr_encode(instr):
    if instr == "add":
        func_type = "r"
        opcode = 0
        funct = 0x20

    elif instr == "addi":
        func_type = "i"
        opcode = 0x8
        funct = None

    elif instr == "addiu":
        func_type = "i"
        opcode = 0x9
        funct = None

    elif instr == "addu":
        func_type = "r"
        opcode = 0
        funct = 0x21

    elif instr == "and":
        func_type = "r"
        opcode = 0
        funct = 0x24

    elif instr == "andi":
        func_type = "i"
        opcode = 0xc
        funct = None

    elif instr == "beq":
        func_type = "i"
        opcode = 0x4
        funct = None

    elif instr == "bne":
        func_type = "i"
        opcode = 0x5
        funct = None

    elif instr == "j":
        func_type = "j"
        opcode = 0x2
        funct = None

    elif instr == "jal":
        func_type = "j"
        opcode = 0x3
        funct = None
```

```
    elif instr == "jr":
        func_type = "r"
        opcode = 0
        funct = 0x8

    elif instr == "lbu":
        func_type = "i"
        opcode = 0x24
        funct = None

    elif instr == "lhu":
        func_type = "i"
        opcode = 0x25
        funct = None

    elif instr == "ll":
        func_type = "i"
        opcode = 0x30
        funct = None

    elif instr == "lui":
        func_type = "i"
        opcode = 0xf
        funct = None

    elif instr == "lw":
        func_type = "i"
        opcode = 0x23
        funct = None

    elif instr == "nor":
        func_type = "r"
        opcode = 0
        funct = 0x27

    elif instr == "or":
        func_type = "r"
        opcode = 0
        funct = 0x25

    elif instr == "ori":
        func_type = "i"
        opcode = 0xd
        funct = None

    elif instr == "slt":
        func_type = "r"
        opcode = 0
        funct = 0x2a

    elif instr == "slti":
        func_type = "i"
        opcode = 0xa
        funct = None

    elif instr == "sltiu":
        func_type = "i"
        opcode = 0xb
        funct = None

    elif instr == "sltu":
        func_type = "r"
        opcode = 0
```

```
        funct = 0x2b

    elif instr == "sll":
        func_type = "r"
        opcode = 0
        funct = 0x00

    elif instr == "srl":
        func_type = "r"
        opcode = 0
        funct = 0x02

    elif instr == "sb":
        func_type = "i"
        opcode = 0x28
        funct = None

    elif instr == "sc":
        func_type = "i"
        opcode = 0x38
        funct = None

    elif instr == "sh":
        func_type = "i"
        opcode = 0x29
        funct = None

    elif instr == "sw":
        func_type = "i"
        opcode = 0x2b
        funct = None

    elif instr == "sub":
        func_type = "r"
        opcode = 0
        funct = 0x22

    elif instr == "subu":
        func_type = "r"
        opcode = 0
        funct = 0x23

    else:
        func_type = None
        opcode = None
        funct = None

    return [func_type, opcode, funct]
```

Register_encoder.py

```
# Array used to contain register numeric values
registers = {
    "$zero": 0,
    "$at": 1,
    "$v0": 2,
    "$v1": 3,
    "$a0": 4,
    "$a1": 5,
    "$a2": 6,
```

```
    "$a3": 7,
    "$t0": 8,
    "$t1": 9,
    "$t2": 10,
    "$t3": 11,
    "$t4": 12,
    "$t5": 13,
    "$t6": 14,
    "$t7": 15,
    "$s0": 16,
    "$s1": 17,
    "$s2": 18,
    "$s3": 19,
    "$s4": 20,
    "$s5": 21,
    "$s6": 22,
    "$s7": 23,
    "$t8": 24,
    "$t9": 25,
    "$k0": 26,
    "$k1": 27,
    "$gp": 28,
    "$sp": 29,
    "$fp": 30,
    "$ra": 31
}


# Given the function type, reg_decode will output an array containing the
# numeric values of the registers and immediates in MIPS code.
# param 'func_type' is the function type of the MIPS code
# param 'instr' is the instruction given in the MIPS code
# param 'regs' is an array containing the registers used in the MIPS code
# returns an array in the form [rs, rt, rd, shamt] for r-type functions
# returns an array in the form [rs, rt, immediate] for i-type functions
def reg_encode(func_type, instr, regs):
    # execution for r-type functions
    if func_type == "r":

        # special case for MIPS shifts
        if (instr == "sll") or (instr == "srl"):
            try:
                # return[rs,        rt,               rd,             shamt]
                return [0, registers[regs[1]], registers[regs[0]], int(regs[2])]
            except:
                return None

                # special case for MIPS jr
        if (instr == "jr"):
            try:
                # return[        rs,        rt, rd,shamt]
                return [registers[regs[0]], 0, 0, 0]
            except:
                return None

                # standard r-type MIPS instructions
        try:
            # return[      rs,                rt,               rd,          shamt]
            return [registers[regs[1]], registers[regs[2]], registers[regs[0]], 0]
        except:
            return None
```

```
    # execution for i-type functions
    elif func_type == "i":

        # special case for lui
        if (instr == "lui"):
            try:
                if len(regs[1]) > 1 and regs[1][1] == "x":
                    imm = int(regs[1], base=16)
                else:
                    imm = int(regs[1])

                # return[rs,       rt        ,  immediate  ]
                return [0, registers[regs[0]], imm]
            except:
                return None


                # special case for lw, sb, sw, ll sc
        if (instr == "lw") or (instr == "sb") or (instr == "sw") or (instr == "ll") or
(instr == "sc"):
            try:
                if len(regs[1]) > 1 and regs[1][1] == "x":
                    imm = int(regs[1], base=16)
                else:
                    imm = int(regs[1])

                # return[       rs,                rt        ,  immediate  ]
                return [registers[regs[2]], registers[regs[0]], imm]
            except:
                return None

        # standard i-type MIPS instructions
        try:
            if len(regs[2]) > 1 and regs[2][1] == "x":
                imm = immnt(regs[2], base=16)
            else:
                imm = int(regs[2])

            # return[        rs                rt             immediate ]
            return [registers[regs[1]], registers[regs[0]], imm]
        except:
            return None


            # execution for j-type functions
    elif func_type == "j":
        try:
            if len(regs[0]) > 1 and regs[0][1] == "x":
                imm = int(regs[0], base=16)
            else:
                imm = int(regs[0])

                # return [ immediate ]
            return [imm]
        except:
            return None

    else:
        return None
```

Mips_encoder.py

```
# Converts MIPS instructions into binary and hex
import sys
from instruction_encoder import instr_encode  # converts the instruction part of a
line of MIPS code
from register_encoder import reg_encode  # converts the register and immediate parts
of the MIPS code


# the main conversion function
def convert(code):
    code = code.replace("(", " ")
    code = code.replace(")", "")
    code = code.replace(",", " ")
    code = code.replace("  ", " ")
    args = code.split(" ")
    instruction = args[0]

    if instruction == "exit":
        sys.exit()

    codes = instr_encode(instruction)
    func_type = codes[0]
    reg_values = reg_encode(func_type, instruction, args[1:])  # get the numeric
values of the registers

    # the following if statement below prints an error if needed
    if reg_values == None:
        print("Not a valid MIPS statement")
        return

    # execution for r-type functions
    if func_type == "r":
        opcode = '{0:06b}'.format(codes[1])
        rs = '{0:05b}'.format(reg_values[0])
        rt = '{0:05b}'.format(reg_values[1])
        rd = '{0:05b}'.format(reg_values[2])
        shamt = '{0:05b}'.format(reg_values[3])
        funct = '{0:06b}'.format(codes[2])
        print("Function type: R-Type")
        print("Instruction form: opcode|  rs |  rt |  rd |shamt| funct")
        print("Formatted binary: " + opcode + "|" + rs + "|" + rt + "|" + rd + "|" +
shamt + "|" + funct)
        binary = opcode + rs + rt + rd + shamt + funct
        print("Binary:           " + binary)
        hex_string = '{0:08x}'.format(int(binary, base=2))
        print("Hex:              0x" + hex_string)

    # execution for i-type functions
    elif func_type == "i":
        opcode = '{0:06b}'.format(codes[1])
        rs = '{0:05b}'.format(reg_values[0])
        rt = '{0:05b}'.format(reg_values[1])
        imm = '{0:016b}'.format(reg_values[2])
        print("Function type: I-Type")
        print("Instruction form: opcode|  rs |  rt |   immediate      ")
        print("Formatted binary: " + opcode + "|" + rs + "|" + rt + "|" + imm)
        binary = opcode + rs + rt + imm
        print("Binary:           " + binary)
        hex_string = '{0:08x}'.format(int(binary, base=2))
        print("Hex:              0x" + hex_string)
```

```
    # execution for j-type functions
    elif func_type == "j":
        opcode = '{0:06b}'.format(codes[1])
        imm = '{0:026b}'.format(reg_values[0])
        print("Function type: J-Type")
        print("Instruction form: opcode|          immediate           ")
        print("Formatted binary: " + opcode + "|" + imm)
        binary = opcode + imm
        print("Binary:           " + binary)
        hex_string = '{0:08x}'.format(int(binary, base=2))
        print("Hex:              0x" + hex_string)

    else:
        print("Not a valid MIPS statement")
        return

    return


# main

print("\n\t\t\t\t\t\tWELCOME TO THE MIPS ENCODER!")
print("\t\t\t\t\t\tType MIPS code below to see it in binary and hex form")
print("\t\t\t\t\t\tSyntax: If using hex, use the '0x' label")
print("\t\t\t\t\t\tType 'exit' to exit")
print("-------------------------------------------------------------------------------
-")
while True:
    mips = input("\nType MIPS code here: ")
    print()
    convert(mips)
    print("-------------------------------------------------------------------------------
-----")
```

Part-2(source code)

.data

welcome: .asciiz "\t\t\t\t\tWelcome to Tiny Calculator \n \t\t\t\t\t\t1 - Addition \n \t\t\t\t\t\t2 - Subtraction \n \t\t\t\t\t\t3 - Multiplication \n \t\t\t\t\t\t4 - Division \n \t\t\t\t\t\t0 - Exit "

choose: .asciiz "\tChoose an option: "

enterNumber1: .asciiz "\tPlease enter first number: "

enterNumber2: .asciiz "\tPlease enter second number: "

newline: .asciiz "\n"

sum: .asciiz "\tThe addition of "

comma: .asciiz " , "

is: .asciiz " is = "

```
difference: .asciiz "\tThe subtraction of "
product: .asciiz "\tThe multiplication of "
quotient: .asciiz "\tThe quotient of "
remainder: .asciiz "\tAnd the remainder is: "
error: .asciiz "\tInvalid number, exiting... "
finishing: .asciiz "\t\t\t\t\tThank You For Using Tiny Calculator. \n"

.text
.globl main
main:

loop:
		#printing newline
	li $v0,4
	la $a0,newline
	syscall
		#printing welcome
	li $v0,4
	la $a0,welcome
	syscall
		#printing newline
	li $v0,4
	la $a0,newline
	syscall
		#printing option to choose
	li $v0,4
```

```
        la $a0,choose
        syscall
                #get int from user
        li $v0,5
        syscall
                #move number from user out of v0 to s0
        move $s0, $v0
                #branches
        beq $s0,0,exit
        beq $s0, 1, addition
        beq $s0, 2, subtraction
        beq $s0, 3, multiplication
        beq $s0, 4, division
        j errorExit

addition:
                #printing Enter Number
        li $v0,4
        la $a0,enterNumber1
        syscall
                #get int from user
        li $v0,5
        syscall
                #move number from user out of v0 to temp0
        move $t0, $v0
                #printing Enter Number
```

```
li $v0,4
la $a0,enterNumber2
syscall
	#get int from user
li $v0,5
syscall
	#move number from user out of v0 to temp1
move $t1, $v0
	#printing Sum String
li $v0,4
la $a0,sum
syscall
	#print user number1
li $v0,1
move $a0,$t0
syscall
	#printing comma
li $v0,4
la $a0,comma
syscall
	#print user number2
li $v0,1
move $a0,$t1
syscall
	#printing is
li $v0,4
```

```
        la $a0,is
        syscall
                #perform Addition
        add $t2,$t0,$t1
                #print user sum of numbers
        li $v0,1
        move $a0,$t2
        syscall
                #printing newline
        li $v0,4
        la $a0,newline
        syscall
        j loop


subtraction:
                #printing Enter Number
        li $v0,4
        la $a0,enterNumber1
        syscall
                #get int from user
        li $v0,5
        syscall
                #move number from user out of v0 to temp0
        move $t0, $v0
                #printing Enter Number
        li $v0,4
```

```
la $a0,enterNumber2
syscall
	#get int from user
li $v0,5
syscall
	#move number from user out of v0 to temp1
move $t1, $v0
	#printing Difference String
li $v0,4
la $a0,difference
syscall
	#print user number1
li $v0,1
move $a0,$t0
syscall
	#printing comma
li $v0,4
la $a0,comma
syscall
	#print user number2
li $v0,1
move $a0,$t1
syscall
	#printing is
li $v0,4
la $a0,is
```

```
        syscall
                #perform subtraction and print
        neg $t3,$t1
        add $t2,$t0,$t3
        li $v0,1
        move $a0,$t2
        syscall
                #printing newline
        li $v0,4
        la $a0,newline
        syscall
        j loop

multiplication:
                #printing Enter Number
        li $v0,4
        la $a0,enterNumber1
        syscall
                #get int from user
        li $v0,5
        syscall
                #move number from user out of v0 to temp0
        move $t0, $v0
                #printing Enter Number
        li $v0,4
        la $a0,enterNumber2
```

```
syscall
        #get int from user
li $v0,5
syscall
        #move number from user out of v0 to temp1
move $t1, $v0
        #printing Product String
li $v0,4
la $a0,product
syscall
        #print user number1
li $v0,1
move $a0,$t0
syscall
        #printing comma
li $v0,4
la $a0,comma
syscall
        #print user number2
li $v0,1
move $a0,$t1
syscall
        #printing is
li $v0,4
la $a0,is
syscall
```

```
		#perform multiplication and print
	mul $t2,$t0,$t1
	li $v0,1
	move $a0,$t2
	syscall
		#printing newline
	li $v0,4
	la $a0,newline
	syscall
	j loop

division:
		#printing Enter Number
	li $v0,4
	la $a0,enterNumber1
	syscall
		#get int from user
	li $v0,5
	syscall
		#move number from user out of v0 to temp0
	move $t0, $v0
		#printing Enter Number
	li $v0,4
	la $a0,enterNumber2
	syscall
```

```
        #get int from user
li $v0,5
syscall
        #move number from user out of v0 to temp1
move $t1, $v0
        #printing Quotient String
li $v0,4
la $a0,quotient
syscall
        #print user number1
li $v0,1
move $a0,$t0
syscall
        #printing comma
li $v0,4
la $a0,comma
syscall
        #print user number2
li $v0,1
move $a0,$t1
syscall
        #printing is
li $v0,4
la $a0,is
syscall
```

```
        #perform division and print
div $t0,$t1
mflo $t6
mfhi $t7
        #printing lo
li $v0,1
move $a0,$t6
syscall
        #printing newline
li $v0,4
la $a0,newline
syscall
        #printing Remainder String
li $v0,4
la $a0,remainder
syscall
        #printing hi
li $v0,1
move $a0,$t7
syscall
        #printing newline
li $v0,4
la $a0,newline
syscall
j loop
```

```
exit:
        #finishing
    li $v0,4
    la $a0,finishing
    syscall
        #exit sequence
    li $v0, 10
    syscall


errorExit:
        #printing error message
    li $v0,4
    la $a0,error
    syscall
        #exit sequence
    li $v0, 10
    syscall
```

Thank you