RISC-V (RV32IM) Assembler

Yazeed Mohammed Alzughaibi

Contents

Introduction:

One of the most effective ways of learning anything is implementation by using it in a real life example, that's why in order to get a deep understanding of the basics of RISC-V ISA (Instruction Set Architecture) assembly language we have to do this assignment. In this assignment we are required to create an assembler in Python that takes RISC-V assembly file and produce a machine code file. by the end of this assignment we will increase our knowledge in RISC-V ISA and Python programming, we will understand computer architecture even further than before as well as how to enhance the performance of the program, also we will be able to grasp and understand other assembly languages much faster than before. the report will include our problem statement where we state the general solution of our problem in the assignment which will give us a clear idea of the problem, after that we will mention the main objectives of this assignment then we will start with the details of our work, the program section is divided into four parts: Database, First pass, methods & functions and second pass. At the end we will have the discussion where we reflect on what was learned.

Background Information:

Before we start discussing the program, we must give a quick explanation of the RISC-V Instruction Set Architecture. RISC-V (reduced instruction set computer) is a computer optimized to work best with a specific instruction set. The main feature of RISC-V is that it has a large number of registers and a highly regular instruction pipeline which allows it to have a lower clock cycles per instruction (CPI), another feature is the load/store architecture which is an ISA that divided categories, memory access and ALU operations. [1]

Register is the fastest place to hold data in computer. In RISC-V there are 32 registers (x0-x31), each register has specific use. For example, register x1 is return address register, and registers x5-x7 & x28-31 is temporary registers, and registers x8-x9 & x18-x27 is saved registers. The figure down shows the number and usage of each register in the RISC-V: (Figure 1) [1]

Name	Register number	Usage
x0	0	The constant value 0
x1 (ra)	1	Return address (link register)
x2 (sp)	2	Stack pointer
x3 (gp)	3	Global pointer
x4 (tp)	4	Thread pointer
x5-x7	5–7	Temporaries
x8-x9	8–9	Saved
x10-x17	10–17	Arguments/results
x18-x27	18–27	Saved
x28-x31	28–31	Temporaries

Figure 1: RISC-V register conventions.

Memory is an important part in the RISC-V ISA to understand, it is divided into three main parts: reserved, text and data segments. The reserved is only for the CPU (central processing unit) and the OS (operating system), the text segment is for the whole code and it is indicated by the ".text" in the simulator, the data segment is also divided into three more sections: static where the global variables are identified, dynamic grows up towards the stack in an area called the heap. (Figure 2) [1]

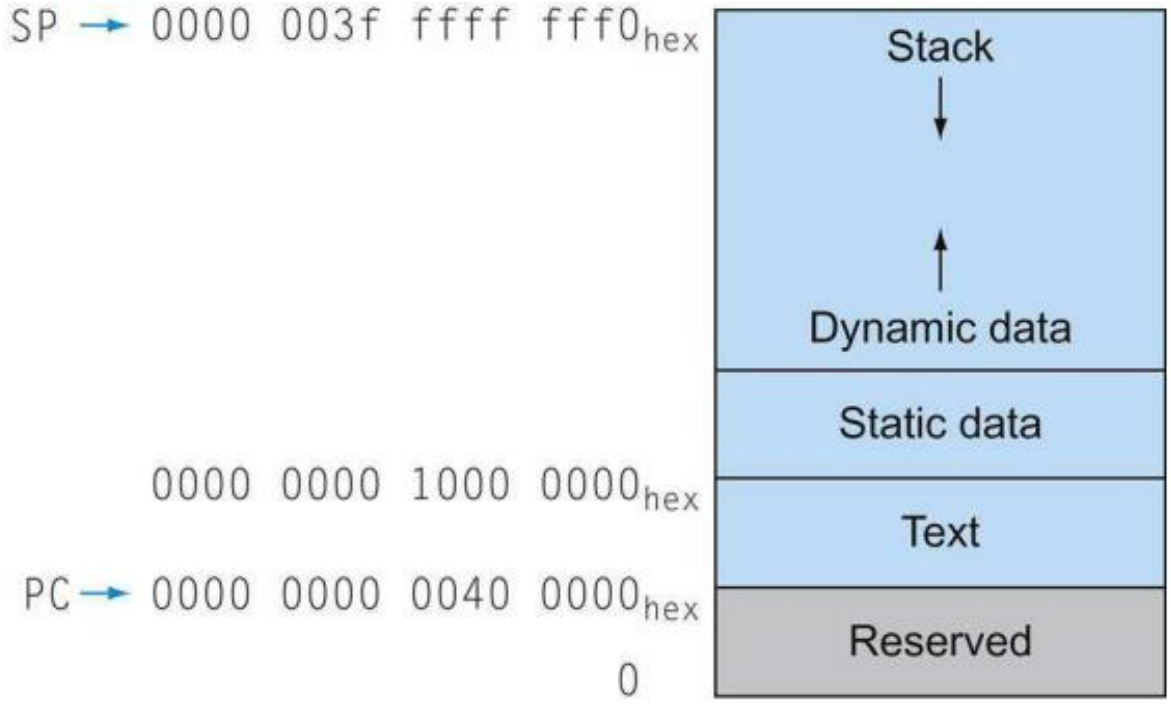

Figure 2: memory segments/allocations

-Problem Statement

Designing Python program to assemble RISC-V assembly code into machine code/binary code for the CPU to understand.

-Assignment Objectives

- Design and implement a RISC-V assembler.
- increase our level of python programming knowledge

The Program:

It is important to start the program by identifying the variables and building the database before anything, also knowing how to extract the data from the input file and prepare an output file to write the machine code on, another thing is knowing how to process the assembly lines and filtering out the unnecessary data which will be explained further.

Variables and Database

Starting with the first dictionary and probably the most important, "main_inst"; this dictionary contains every instruction that will be used corresponding with their instruction type (U, UJ, R, I…), this dictionary will be used later to check the type of the instruction and decide whether this instruction exists or not. (figure 3)

```
main_inst = {'lui': 'u', 'auipc': 'u', 'jal': 'uj', 'beq': 'sb',
        'bne': 'sb', 'blt': 'sb', 'bge': 'sb', 'bltu': 'sb', 'bgeu': 'sb',
        'lw': 'i1', 'lb': 'i1', 'lh': 'i1', 'lbu': 'i1', 'lhu': 'i1',
        'sb': 's', 'sh': 's', 'sw': 's', 'addi': 'i2', 'slti': 'i2',
        'sltiu': 'i2', 'xori': 'i2', 'ori': 'i2', 'andi': 'i2', 'slli': 'i3',
        'srli': 'i3', 'srai': 'i3', 'add': 'r','sub': 'r',  'sll': 'r', 'slt': 'r',
        'sltu': 'r', 'xor': 'r', 'srl': 'r', 'sra': 'r', 'or': 'r', 'and': 'r',
        'mul': 'r', 'mulh': 'r', 'mulhsu': 'r', 'mulhu': 'r', 'div': 'r',
        'divu': 'r', 'rem': 'r', 'remu': 'r', 'la' : 'ps' , 'mv' : 'ps',
        'ret' : 'ps','bgtz' : 'ps', 'blez' : 'ps', 'li' : 'ps', 'j' : 'ps',
        'call' : 'ps', 'ble': 'ps', 'beqz': 'ps', 'ecall': 'i4', 'ebreak': 'i4', 'jalr': 'i_5'
        }  # dictionary with all of our instructions with their types.
```

Figure 3: "main_inst" dictionary

After making one large dictionary for all the instruction types, we separate each type to store in their own list or dictionary as needed. For example, the I-type instructions has more than one format that is why we had to make separate list or dictionary for each format. These dictionaries will later be called for values like opcodes, funct3, funct7. (Figure 4)

```
#list or dictionary for every type/family
u = {'lui' : '0110111', 'auipc': '0010111'}#u instructions only has opcode
uj = {'jal': '1101111'} # uj instruction has opcodes
sb = {'opcode': '1100011', 'beq':'000', 'bne':'001',
      'blt': '100', 'bge': '101', 'bltu': '110', 'bgeu': '111'} # sb instructions
s = {'opcode': '0100011', 'sb': '000', 'sh': '001', 'sw': '010'} # s instructions

i_1 = {'opcode': '0000011', 'lb': '000', 'lh': '001', 'lw': '010',···
i_2 = {'opcode': '0010011', 'addi': '000', 'slti': '010', 'sltiu': '011',···
i_3 = (['opcode', '0010011'], ['slli', '001', '0000000']···

i_4 = (['ecall', '1110011', '00000', '000', '00000', '000000000000'],···

i_5 = ['jalr', '1100111', '000'] #i5

r = ( ['opcode', '0110011'], ['add', '000', '0000000'],···
```

Figure 4: dictionary for each type

To make the process of getting the binary value of a register easier, we made a dictionary called "reg_bin" consists of every shape of registers corresponding with their binary value in 5 digits. Just to make it a little easier later. (Figure 5)

```
reg_bin = {'zero': bin(0)[2:].zfill(5),
    'x0': bin(0)[2:].zfill(5),
    'ra': bin(1)[2:].zfill(5),
    'x1': bin(1)[2:].zfill(5),
    'sp': bin(2)[2:].zfill(5),
    'x2': bin(2)[2:].zfill(5),
    'gp': bin(3)[2:].zfill(5),
    'x3': bin(3)[2:].zfill(5),
    'tp': bin(4)[2:].zfill(5),
    'x4': bin(4)[2:].zfill(5),
```

figure 5: sample of the reg_bin dictionary, the [2:] is to take the whole number except the first 2 digits i.e. (0b00000 ---> 00000)

Unfortunately, we had to hardcode the labels addresses "label_address", even though we were able to extract the labels with their line numbers, we couldn't extract their addresses through the code successfully (figure 6).We had it increment for every instruction. We needed their addresses to calculate the jump or load addresses and many more instructions that require addresses.

```
method = {} #holds the {label name : line number}
labels = [] #storing labels
label_address = {'main' : 0x00400000, 'print' : 0x00400060,
'clear': 0x00400070, 'Loop': 0x00400074, 'copy': 0x00400088,
'CPLoop': 0x00400090, 'swap': 0x004000a8, 'sort': 0x004000bc,
'L4': 0x004000e8, 'L3': 0x00400108, 'L5': 0x00400110,
'L1': 0x00400118, 'L2': 0x0040012c, 'A1': 0x10010000,
'A2': 0x10010020, 'exitmsg': 0x10010040}
#unfortunatlly we had to hard code the addresses because we couldn't find a way to extract them
directives = [] #saving the directives with their line numbers {directive : line NO.}
pc = 1  #variable for program conter
pc_address = 0x00400000 #starting address for program counter
```

Figure 6: "labels_addresses"

After successfully building our database, we can start initiating important variables, list and dictionaries to be used and filled later in the program. The "method" dictionary holds the labels corresponding with their line number. The list of dictionaries "directives" holds in every index a dictionary with the directive corresponding with its line number. "pc" is the program counter. The "assembly_lines" list only contains the assembly code after it is been filtered (comments, labels…). Finally, "input_file" & "output file" with the input file carrying our assembly code and the output file contains the translated machine code (binary). We did initiate the addresses and memory part, but it didn't work out so well, we had issues separating the text from the data segments, the only part that worked and we actually needed to calculate the addresses later.(figure 7)

```
assembly_lines =[] #assembly lines/instructions e.g add x1, x2, x3
with open ("Bubble-sort.txt") as input_file: #assembly input file
    output_file = open("output.txt", "w")  # text output file
    output_bin = open('output.bin', 'w')   #binary output file
    #memory part
    static = [] #store static global variables such as (A1, A2)
    stack = [] #stack is the work space for the code
    stack_address = 0x7fffeffc
    dynamic = [] #dynamic is the heap
    dynamic_address = 0x10040000
    global_pointer = 0x10008000
    data = [static, dynamic, stack] #all of the data segment stored here
    data_address = 0x10010000
    text = [] #all of the instructions are stored here
    text_address = 0x00400000
    reserved = [] #reserved for the CPU + OS
    memory = [reserved, text, data]
    assembly_with_lineNo = {}
    #reg_adr = {}
    lineNo = 0
```

Figure 7: memory part and some variables

First Pass (filtering the assembly file)

Once we have completed the Database for our program which will help us start. In this part of the program the first pass is going to filter the assembly file "input file" by removing the comments (#), tabs (\t), directives (.text, .data, …), empty lines etc...

Before starting the first pass, we create a variable for the program counter "pc" and set it to 1 (pc = 1) and create an empty array to save the filtered assembly lines in "assembly_lines". Then we open the input file (Bubble-Sort.txt) for reading we used with open to be able to run from the "CMD", the output files (output.txt) and (output.bin) for writing. After that, we create lists/dictionaries for the memory part to store static global variables, stack, dynamic (heap) etc...

The third step is to create a loop that read the input file line by line and filter it using .find to find and .replace to replace, or store and remove unneeded data such as comments, tabs, empty lines, labels, and directives. In addition to that we save the filter line in assembly_lines array after removing the spaces from both sides of the line. In the end of the loop (for loop) we raise the "pc" by 1 (pc = pc + 1). (figure 8)

The last step in the first pass is to reset PC (PC = 0) and create empty variables for each of the arguments, instruction, opcode, and Type. (figure 8)

Third & last steps in the first pass (for more detail on the For Loop/Third Step see the python code).

```
for line in input_file: #reading the input file line by line
    label_index = 0
    if line.find('\t') == 0: #finding tabs in the beggining of every line and removing them
        x = line.index('\t')
        line = line[x + 1:]
    if line.find('\t') >= 0: #replacing tabs in the middle of the lines to regular spaces
        line = line.replace('\t', ' ')
    if line.find('\n') >= 0: #finding and removing new line indecation
        line = line.replace('\n', ' ')
    for word in assembly_lines:
        if word == '':
            assembly_lines.remove(word)

    if line.find(' ') == 0: #checks if the first element in the line is a space if it is it removes it
        y = line.index(' ')
        line = line[y + 1:]
    if line.find('#') >= 0: #finds comments and skips them
        line = line [:line.index('#')]
    if ' ' in line: #removes spacing from both sides of the line
        line = line.strip(' ')



    if line.find(':') > 0: #extracting labels
        label_index = line.index(':')
        method.update({line[:label_index]:pc})
        labels.append(line[:label_index])
        line = line.replace(line[:label_index + 1], '') #removing the labels from the lines


    if line.find('.') >= 0: #extracting directives···


    if ' ' in line: #removes spacing from both sides of the line
        line = line.strip(' ')
    assembly_lines.append(line)
    if line != '':
        assembly_with_lineNo.update({pc : line})
        lineNo += 1

    pc = pc + 1
```

Figure 8: first pass, some if statements were compressed for the picture.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI	U-type
imm[31:12]					rd	0010111	AUIPC	
imm[20|10:1|11|19:12]					rd	1101111	JAL	UJ-type
imm[11:0]			rs1	000	rd	1100111	JALR	I5-type
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ	SB-type
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE	
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT	
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE	
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU	
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU	
imm[11:0]			rs1	000	rd	0000011	LB	I1-type
imm[11:0]			rs1	001	rd	0000011	LH	
imm[11:0]			rs1	010	rd	0000011	LW	
imm[11:0]			rs1	100	rd	0000011	LBU	
imm[11:0]			rs1	101	rd	0000011	LHU	
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB	S-type
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH	
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW	
imm[11:0]			rs1	000	rd	0010011	ADDI	I2-type
imm[11:0]			rs1	010	rd	0010011	SLTI	
imm[11:0]			rs1	011	rd	0010011	SLTIU	
imm[11:0]			rs1	100	rd	0010011	XORI	
imm[11:0]			rs1	110	rd	0010011	ORI	
imm[11:0]			rs1	111	rd	0010011	ANDI	
0000000		shamt	rs1	001	rd	0010011	SLLI	I3-type
0000000		shamt	rs1	101	rd	0010011	SRLI	
0100000		shamt	rs1	101	rd	0010011	SRAI	
0000000		rs2	rs1	000	rd	0110011	ADD	R-type
0100000		rs2	rs1	000	rd	0110011	SUB	
0000000		rs2	rs1	001	rd	0110011	SLL	
0000000		rs2	rs1	010	rd	0110011	SLT	
0000000		rs2	rs1	011	rd	0110011	SLTU	
0000000		rs2	rs1	100	rd	0110011	XOR	
0000000		rs2	rs1	101	rd	0110011	SRL	
0100000		rs2	rs1	101	rd	0110011	SRA	
0000000		rs2	rs1	110	rd	0110011	OR	
0000000		rs2	rs1	111	rd	0110011	AND	
fm	pred	succ	rs1	000	rd	0001111	FENCE	
000000000000			00000	000	00000	1110011	ECALL	I4-type
000000000001			00000	000	00000	1110011	EBREAK	

Figure9: table of each type from google classroom site edited by us

Methods/Functions (function for each type)

After successfully filtering our assembly lines from comments, directives and labels. Separating our assembly lines in the second pass knowing which is the instruction and which are the operands. We can start translating our assembly code into machine/binary code for the CPU to understand. The methods/functions will be explained by the order they are written in. [2]

Before we start explaining methods, some abbreviations should be noted;

Table 1: operand abbreviations and their meanings.

abbreviations	Meaning
Opcode	Operation Code for the instruction
Rd	Return Destination Operand
Rs1	First Register Source Operand
Rs2	Second Register Source Operand
Imm	Immediate Value (Addresses & values)
Funct3	Additional Opcode field (3 Bits)
Funct7	Additional Opcode field (7 Bits)
shamt	Shift Amount

Starting with the "i1_i2_function()", the reason why we put "i1" and "i2" types together in one function is that they have the same format (figure 9). Since we have the operands ("args") separated from the "instruction" all we must do is find which operand is which, simply since we know how certain types syntax look like. If its i1 it will look like this, ("instruction" "rd", "imm"("rs1")). If its i2 (("instruction" "rd", "rs1", "imm")). They all have the same "rd" location/index and we can locate it by finding a space (" ") in the "args" with <args.find(" ")> and using that index to store "rd". Now i1 & i2 differ in the last two operands and to decide we take the rest of the line after the index of (" ") and use the <.find> to find a parenthesis ("("), if there is one we find the index of the opening and the closing parenthesis < rest_of_line.index("(")> <rest_of_line.index(")")> and whatever is in between them will be the value of "rs1" and what ever is behind all of that will be the offset "imm". If its not i1 it'll defiantly be i2, same way as before we find a space in the rest of the line, whatever is before the space will be "rs1" and the rest will be "imm". Now after separating every operand in their own variables, any register will be turned to binary code by using the "reg_bin" dictionary by simply using the register as a key and it'll output its value in binary <rd = reg_bin[rd]>. For the "imm" value sometimes it might be a negative value, we just check and store the binary value of that, anf if it's not negative we use this line <imm = bin(int(imm))[2:].zfill(12)>, basically it turns it to binary code and remove the first two elements([2:]) which are (0b00000000) and use ".zfill" to fill the left with zeros with the desired amount of digits. "funct3" and "opcode" are what's left, simply just use an if statement to check the type of instruction and get the "funct3" and "opcode" from their separate dictionary. Finally print with the order needed and write on output file as well. (figure 10)

```
def i1_i2_function():
    rd_index = args.find(" ") #find the space in the line indicating rd
    rd = args [0:rd_index]
    rest_of_line = args [rd_index + 1:]
    if rest_of_line.find("(") >= 1: #incase it is i1 type
        beg = rest_of_line.index("(")
        end = rest_of_line.index(")")
        rs1 = rest_of_line[beg+1 :end]
        imm = rest_of_line[:beg]
    else: #it will be i2
        space = rest_of_line.index(" ")
        rs1 = rest_of_line[:space]
        imm = rest_of_line[space + 1 :]
    rd = reg_bin[rd]
    imm = int(imm)
    if imm < 0:
        x = str(imm)
        a = x.strip('-') # 11 in binary
        x = int(x)
        b = (x ^ 4095) + 1
        imm = bin(b)[3:]
    else:
        imm = bin(int(imm))[2:].zfill(12)
    rs1 = reg_bin[rs1]
    funct3 = ''
    opcode = ''
    if Type == 'i1':
        opcode = i_1['opcode']
        funct3 = i_1[instruction]
    elif Type == 'i2':
        opcode = i_2['opcode']
        funct3 = i_2[instruction]
    global pc_address
    a = 0x4
    pc_address += a
    print(imm, rs1, funct3, rd, opcode, sep="" )
    output_file.writelines(imm + rs1 + funct3 + rd + opcode +  '\n')
    output_bin.writelines(imm + rs1 + funct3 + rd + opcode + '\n')
```

Figure10: i1_i2_fuction().

Next will be “i3_r_function()”, the reason why I put these two together is same as before; they might have different opcodes and what not, but they still have the same format(figure 9). We first start by checking the type of instruction with an if statement <if Type == ‘r’> if yes than go through the “r”(tuple) list with a for loop and find the instruction index to later be able to get the “opcode”, “funct3” & “funct7”. If the previous if statement is no than do the same with “i3”(tuple). Separate the operands “args” with the “.split” like such <args.split()> to split up the operands into a list

(['x1', 'x2', 'x3'])and then basically “rd”, “rs1” & “rs2” will be in indices 0, 1 & 2 respectively. Same as before turning registers into binary code using the “reg_bin” dictionary. Next check of type of the instruction to get the “opcode”, “funct3” and “funct7”. Lastly print in order needed and write on output file as well. (Figure 11)

```
def i3_r_function():
    inst_index=0
    if Type == 'r': #check type of instruction to know location in their tuple
        for a in r:
            if a[0] == instruction:
                break
            inst_index+=1

    else: #for i3
        for a in i_3:
            if a[0] == instruction:
                break
            inst_index+=1
    arguments = args.split() #e.g ['x1', 'x2', 'x3']
    rd = arguments[0]
    rs1 = arguments[1]
    rs2 = arguments[2]
    rd = str(reg_bin[rd])
    rs1 = str(reg_bin[rs1])
    rs2 = str(reg_bin[rs2])
    func3 = ''
    func7 = ''
    #we bring the funct3 & funct7 numbers from the dictinaries/sets
    if main_inst[instruction] == 'r':
        opcode = r[0][1]
        func3 = r[inst_index][1]
        func7 = r[inst_index][2]
    else:
        opcode = i_3[0][1]
        func3 = i_3[inst_index][1]
        func7 = i_3[inst_index][2]
    global pc_address
    a = 0x4
    pc_address += a
    print(func7 , rs2 , rs1 , func3 , rd , opcode , sep="")
    output_file.writelines(func7 + rs2 + rs1 + func3 + rd + opcode + "\n")
    output_bin.writelines(func7 + rs2 + rs1 + func3 + rd + opcode + "\n")
```

Figure 11: i3_r_function().

Afterwards, “s_function()”; same as the “i1_i2_function()”, we find “rs2” by finding the first space. Then take the rest of the line and find the parenthesis (“(“) indicating “rs1” between the parentheses and whatever is before the opening parenthesis will be “imm”. Change the operands to binary same as before. Get the “funct3” & “opcode” from the “s” dictionary using the instruction as key. Print in the order needed and write on output file as well. (Figure 12)

```
def s_function():
    rs2_index = args.find(' ') #first element will be rs1
    rs2 = args[0:rs2_index]
    rest_of_line = args[rs2_index + 1:]
    if rest_of_line.find('(') > 0: #check for parethasis for rs2
        beg = rest_of_line.index('(')
        end = rest_of_line.index(')')
        rs1 = rest_of_line[beg+1:end]
        imm = rest_of_line[:beg] #whatever is before the parethasis is imm
    #turn to binary
    imm = bin(int(imm))[2:].zfill(12)
    rs1 = reg_bin[rs1]
    rs2 = reg_bin[rs2]
    funct3 = s[instruction]
    opcode = s['opcode']
    global pc_address
    a = 0x4
    pc_address += a
    print(imm[0:7], rs2, rs1, funct3, imm[7:], opcode, sep='')
    output_file.writelines(imm[0:7] + rs2 + rs1 + funct3 + imm[7:] + opcode + '\n')
    output_bin.writelines(imm[0:7] + rs2 + rs1 + funct3 + imm[7:] + opcode + '\n')
```

Figure 12: s_function():

In the "sb_function()" we start right away by splitting the operands "args" (['x1', 'x2', 'x3']), simply first two elements are "rs1" & "rs2" and the last one should be the "imm" but not right away, the third element is a label but we can't simply use it since its still a string value and we need to get its address from it. We do that by calling the "label_address" dictionary with the label in hand to get its address, then we use a formula, address = (pc_address - label_address[target]) / -2, (target is the label). This is the address in decimal all we need to do is turn it into binary but first we need to check if its negative or not and if it is we get its 2's complement. After that just change the register values to binary like before, get the "funct3" and "opcode" from the "sb" dictionary and lastly print the output and write on the output file. (Figure 13)

```
def sb_function():
    arguments = args.split() # ['x1', 'x2', 'x3']
    rs1 = arguments[0]
    rs2 = arguments[1]
    target = arguments[2]
    global pc_address
    address = (pc_address- label_address[target]) / -2
    imm = int(address)
    if imm < 0:
        x = str(imm)
        a = x.strip('-')
        x = int(x)
        b = (x ^ 4095) + 1
        imm = bin(b)[3:]
    else:
        imm = bin(imm)[2:].zfill(12)
    rs1 = reg_bin[rs1]
    rs2 = reg_bin[rs2]
    funct3 = sb[instruction]
    opcode = sb['opcode']
    a = 0x4
    pc_address += a
    print(imm[0], imm[2:8], rs2, rs1, funct3, imm[8:12], imm[1], opcode, sep='')
    output_file.writelines(imm[0]+ imm[2:8]+ rs2+ rs1+ funct3+ imm[8:12]+ imm[1]+ opcode+'\n')
    output_bin.writelines(imm[0]+ imm[2:8]+ rs2+ rs1+ funct3+ imm[8:12]+ imm[1]+ opcode+'\n')
```

Figure13: sb_function().

With the "u_function()" nothing is new, just our two operands "rd" and "imm", get the opcode and print the binary value of each. (Figure 14)

```
def u_function():
    rd_index = args.find(' ')
    rd = args[0:rd_index]
    imm = args[rd_index+1:]
    imm = int(imm, 16) #hexadecimal to decimal
    rd = reg_bin[rd]
    imm = bin(int(imm))[2:].zfill(32)
    opcode = u[instruction]
    global pc_address
    a = 0x4
    pc_address += a
    print(imm, rd, opcode, sep='')
    output_file.writelines(imm[0:19] + rd + opcode + '\n')
    output_bin.writelines(imm[0:19] + rd + opcode + '\n')
```

Figure 14:u_function():

Last official type function is “uj_function()”, just like before: splitting the operands, taking the label (target), calculating the jump address and outputting the final machine code. (Figure 15)

```
def uj_function():
    imm = ''
    rd = ''
    separated = args.split()
    rd = separated[0]
    rd = reg_bin[rd]
    if instruction == 'jal' :
        target = separated[1]
        for x in method:
            if x == target:
                imm = (4 * method[x]) - (4 * pc)
        if imm < 0:
            imm = bin(int(imm))[3:].zfill(21)
            imm [0] = '1'
        else:
            imm = bin(int(imm))[2:].zfill(21)
        opcode = uj [0] [1]
        global pc_address
        a = 0x4
        pc_address += a
        print(imm[0], imm[10:20], imm[9], imm[1:9], rd, opcode, sep='')
        output_file.writelines(imm[0]+ imm[10:20]+ imm[9]+ imm[1:9]+ rd+ opcode+ '\n')
        output_bin.writelines(imm[0]+ imm[10:20]+ imm[9]+ imm[1:9]+ rd+ opcode+ '\n')
```

Figure 15: uj_function().

getting into the hardest part of the assignment, the pseudo code instructions. Pseudo instructions are a normal instruction, they use one or more basic instructions. We had a certain amount of pseudo instructions that we had to add. We used the “RISC-V Formal Reference 2019” and “RARS 1.4 (help)” to see what each pseudo instruction is made up of. (figure 16)

```
def pseudo_instruction():
    if instruction == 'la': #la a, (label) = auipc a, (label)  +  addi a, a, (label address)···
    elif instruction == 'mv': #mv a, b = add a, x0, b···
    elif instruction == 'ret': #ret = jalr x0, x1, 0x00000000···
    elif instruction == 'bgtz': # bgtz a, (label) = blt x0, a, (label address)···
    elif instruction == 'blez': #blez a, (label) = bge x0, a, (label address)···
    elif instruction == 'li': #li a, (immediat value) = addi a, (immediat value), (immediat value address)
    elif instruction == 'j': # j (label) = jal x0, (label address)···
    elif instruction == 'call': # call (label) = auipc x6, 0x00000000 + jalr x1, x6, (label adress)···
    elif instruction == 'ble': # ble a, b, (label) = bge a, b, (label address)    ···
    elif instruction == 'beqz': # beqz a, (label) = beq a, x0, (label address)···
```

Figure 16: pseudo_instructions().

As you can see in figure 17, every pseudo instruction has a short explanation Infront of it. Basically, we just check which pseudo instruction it is and do what we did before to decode it and

understand it more to translate it. Next figure will show one of the pseudo instructions translation process. (figure 17)

```
if instruction == 'la': #la a, (label) = auipc a, (label)  +  addi a, a, (label address)
    #first translate into auipc
    seperated = args.split()
    rd = seperated[0]
    rd = reg_bin[rd]
    target = seperated[1] #to get immidiat value
    imm = int('fc10', 16) # for some reason the value "fc10" is a constant here
    imm = bin(imm)[2:].zfill(20)
    opcode = u['auipc']
    global pc_address
    address = hex(pc_address- label_address[target])[7:]
    a = 0x4
    pc_address += a
    print(imm, rd, opcode, sep='')
    output_file.writelines(imm + rd + opcode + '\n')
    output_bin.writelines(imm + rd + opcode + '\n')
    #second translate into addi
    opcode = i_2['opcode']
    rs1 = rd
    imm = int(address, 16)
    if imm < 0: # 2's complement
        x = str(imm)
        a = x.strip('-')
        x = int(x)
        b = (x ^ 4095) + 1
        imm = bin(b)[3:]
    else:
        imm = bin(int(imm))[2:].zfill(12)
    funct3 = i_2['addi']
    a = 0x4
    pc_address += a
    print(imm, rs1, funct3, rd, opcode, sep="")
    output_file.writelines(imm + rs1 + funct3 + rd + opcode + '\n')
    output_bin.writelines(imm + rs1 + funct3 + rd + opcode + '\n')
```

Figure 97: la instruction process

the last function to mention is the "memory_update()" function, we tried to implement it but it didn't work, it probably would've worked if we were better with python programming but this will only make us learn more about. (Figure 18)

```
def memoryupdate(): #faild attempt at memory updating
    text_line = 0
    data_line = 0
    for i in range(len(directives)):
        if directives[i].keys() == '.text':
            text_line = directives[i].values()
        elif directives[i].keys() == '.data':
            data_line = directives[i].values()

    if pc > text_line and pc < data_line:
        text.append(line)
    else:
        data.append(line)
```

Figure 18: memory_update() failed attempt

-Second Pass:

What the second for loop in the program does is extracting the instructions and operands from the assembly line and store them in different variables. The loop starts by reading the "assembly_lines" list line by line.

Before doing anything two special instructions need to be check for first, "ecall" and "ret" and processing them. (Figure 19)

```
if line == 'ecall':
    instruction = line
    i4_i5_function()
    continue
elif line == 'ret':
    instruction = line
    pseudo_instruction()
    continue
```

Figure 19: processing "ecall" and "ret"

we remove every comma "," from the file by replacing it with a space, then store it in "wo_comma" variable, the next thing is looking for the index of the operation, to find it the program will look for any space and determine the index from the beginning to the space as in <instruction_index = wo_comma.find(' ')> and after <instruction = wo_comma[0:instruction_index]> now the instruction variable contains the operation for each line in the file. Next step is looking for the arguments of each operation in a similar way and will be stored in a variable as follows: (Figure 20)

```
args_index = wo_comma[instruction_index + 1:]  # the operands for the operations
```

Figure 20:args_index

now that we have extracted the operation and we can point it's argument, the program checks if the instruction (from the operation name) can be found in the "main_inst" dictionary, if it does it will save the type of the operation, if it doesn't the program will terminate. With the arguments there was a problem because it will store every word after the operation but that means even the comments will be stored and we don't have a use for it, so a for loop was created to read each word and remove the comments. (Figure 21)

```
for word in args_index.split():
    # print(word)
     if  word.find('#'):
     break
    args = args + word + " "
```

Figure 21: check for comments #

The final if statements check for the instruction type and execute the method that correlates to it.

Results & Findings:

checking our results and findings helps with knowing the correctness of our work. By looking through the output file and comparing it to the file provided to us by our instructor, we can check it for mistakes. Some instructions did not want to work at all such as, "j" and "call" (to some extent). So, the correctness is not 100%, but we tried our best. The method that was used to compare the output file is a tool in "Notepad++" called "Compare2.0", it simply compares the files line by line and shows where the errors were. (Figure 22)

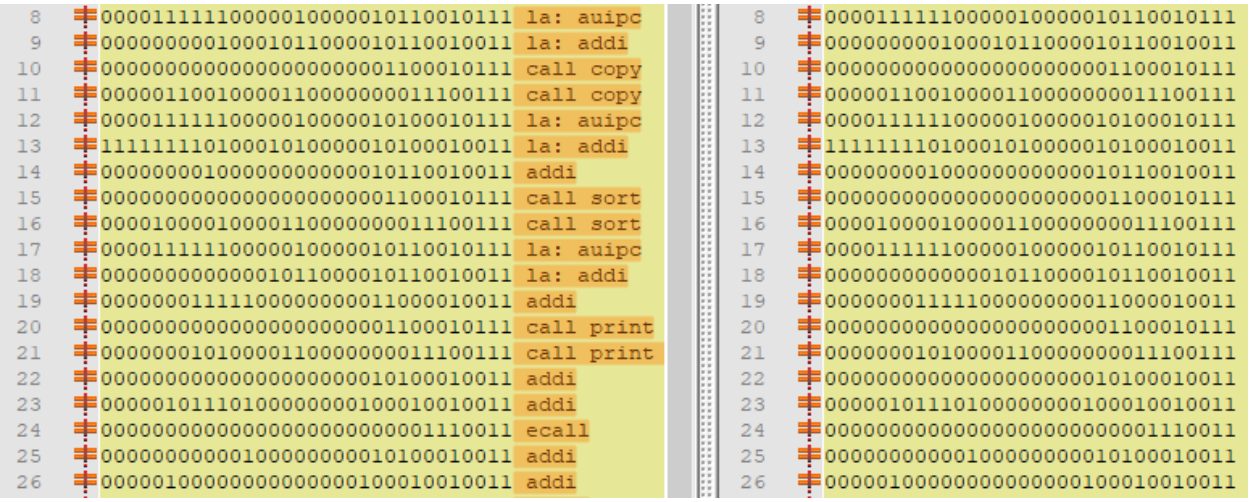

Figure 22: Compare2.0 in Notepad++

Discussion:

Reading the materials of the RISC-V ISA alone did not really leave a strong mark in understanding the concepts compared to doing this assignment, it really helped in understanding the machine side more. We learned to use some built-in functions to manipulate data we did not know before, another important understanding is RISC-V instructions, their different formats, how they're translated into machine code, also how the memory stores in the RISC-V. The process in making the assembler was not easy at all, it was a long process of trial and error, testing and debugging to make sure we get it right and some things could have been better, as for the correctness of the result it was not 100% done correctly, some were hardcoded, We did not get everything perfect but overall it was a valuable experience. Using this python program now we can easily translate any RISC-V assembly code into machine by just having it in a text file

References:

[1] -Computer Organization and Design RISC-V edition

[2] -RISC-V Formal reference 2019