

ES 215

COMPUTER ORGANIZATION AND ARCHITECTURE

PROJECT REPORT

ASSEMBLER

Hima Sagar Potnuru

21110158

himapotnuru@iitgn.ac.in

Keerthi Ramineni

21110176

keerthiramineni@iitgn.ac.in

Sudharshan Kumar Bhardwaj Suvvari

21110218

sudharshansuvvari@iitgn.ac.in



Indian Institute of Technology, Gandhinagar

Github source: https://github.com/himasagarpotnuru/Assembler_SDLX

Abstract:

In this project, a two-pass assembler for an SDLX processor has been built. An assembler is a software that converts assembly language code into processor-executable machine code. It also enables the users to monitor the conversion with our implementation from assembly to machine code. The assembler software receives the SDLX assembly language and converts it to machine code which is in binary format. In this project, we used Python language to build the entire program. Python is a high-level language that is pleasant and easily understandable to humans.

Introduction:

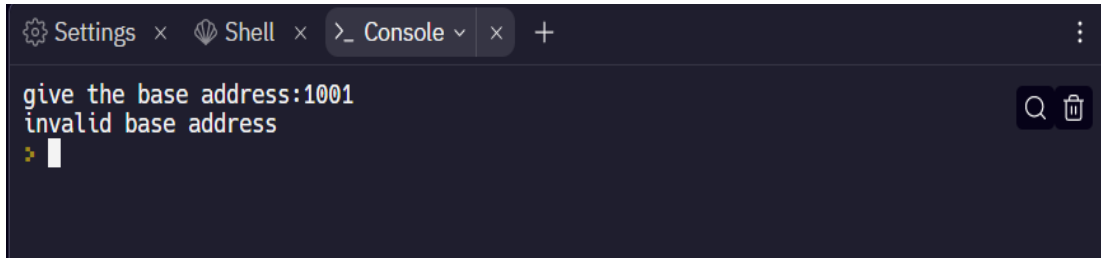
Our project aims to develop an assembler that converts assembly language code to machine code for SDLX processors using Python (3.7 or higher). An assembly language is a human-friendly version of machine code that consists of text which is easy to follow, unlike machine code that consists of binary code to represent instructions.

This project allows us to get an understanding and familiarize ourselves with the underlying concepts and multiple instruction sets of assembly code and concepts of SDLX processors along with instruction formats. We tried to implement all the concepts of SDLX processors and assembly language code considering both text files and single command lines to the assembler as input.

We designed and developed a complete Assembler with the knowledge we gained through this semester's coursework and included the different possibilities or cases in the instruction and input formats.

The assembler we have implemented consists of 2 passes,

- We run through each command line in the first pass, storing the addresses of the instructions as well as the instructions themselves. We also removed the blank lines at the ends of the instruction lines by using the `strip()` function.
- In the second pass, we go through every instruction we've stored, combine every component, and convert it to binary format.
- We have made sure to include a reporting feature in our program so that if there is ever an error, it will be flagged. For instance, an error will be reported if the base address is 1001, which is not divisible by 4.

A screenshot of a terminal window with a dark background. The window has tabs at the top labeled 'Settings', 'Shell', and 'Console'. The 'Console' tab is active. Inside the console, the text 'give the base address:1001' is on the first line, and 'invalid base address' is on the second line. A yellow prompt character is visible on the third line. On the right side of the console, there are icons for search and trash.

Project Idea:

The project's main objective is to implement a two-pass Assembler that takes the assembly language as input and generates a file copy of data and machine instruction (SDLX processor instruction format, i.e., sequence of 32 bits) that will be later loaded into a computer in preparation for execution.

We implemented the project in Python programming language. We further encoded our assembler, considering the possibility of the programmer's mistakes. The assembler may be able to catch some of these errors. The effectiveness of assembler error signals significantly impacts how simple assembly language programming is. We implemented it to provide information about the machine code that aids the programmer in debugging runtime errors.

It runs through the given text file or the command lines and stores them into a string named “command_line” after removing the unnecessary spaces. A system table is created to store the label and their corresponding address through a dictionary in Python named “label ” for data or branch or jump targets by implementing the assembler in the two-pass organization. There wouldn't be any problem in handling the labels that come after branch functions with the labels because we stored the address previously. After organizing the input as mentioned above, based on the opcode values, functioning instruction formats, instruction types, and operand bits are finalized to output a sequence of 32-bit words.

Implementation of our project :

We designed the assembler in a two-pass organization. This mode of organization serves both assembler functions and the usual nature of the assembly language code. While the memory address for a label cannot be determined until its definition has been read, the jump or branch target's address must be known to generate code for those operations. As a result, the input is processed twice by the assembler. The assembler gives labels memory addresses during the first pass, and the machine code is generated during the second pass.

Each opcode is denoted with a number, and an opcode is a necessary operation we must carry out between the registers. As an illustration, 1 denotes the act of adding. If the addition operation is requested via input when we are attempting to transform assembly language code into machine language code, we must include 1 in the output. As a result, we came up with a dictionary of codes, each with a name, a number corresponding to it, and the operation it describes; this same thing is applied to the registers.

```
opcodes = {
    'add': 1, 'sub': 2, 'mul': 3, 'or': 4, 'xor': 5, 'and': 6, 'sll': 7, 'sla': 8,
    'sra': 9, 'srl': 10, 'ror': 11, 'rol': 12, 'slt': 13, 'sgt': 14, 'sle': 15,
    'sge': 16, 'ugt': 17, 'ult': 18, 'uge': 19, 'ule': 20,
    # till here instruction codes goes in R type triadic

    'addi': 21, 'subi': 22, 'ori': 23, 'andi': 24, 'xori': 25, 'slli': 26, 'srli': 27,
    'srai': 28, 'slti': 29, 'sgti': 30, 'slei': 31, 'sgei': 32, 'ugti': 33, 'ulti': 34,
    'ugei': 35, 'ulei': 36, 'lhi': 37, 'lh': 38, 'sw': 39, 'sb': 40, 'sh': 41,
    # from 21 to 41 instruction code goes in RI type triadic

    'bnez': 42, 'beqz': 43, 'jr': 44, 'jalr': 45,
    # These branch instruction goes in R type dyadic

    'j': 46, 'jal': 47
    # These jump instruction goes in J type
}
```

```
Registers = {
    '$r0': 0, '$r1': 1, '$r2': 2, '$r3': 3, '$r4': 4, '$r5': 5, '$r6': 6, '$r7': 7,
    '$r8': 8, '$r9': 9, '$r10': 10, '$r11': 11, '$r12': 12, '$r13': 13, '$r14': 14,
    '$r15': 15, '$r16': 16, '$r17': 17, '$r18': 18, '$r19': 19, '$r20': 20, '$r21': 21,
    '$r22': 22, '$r23': 23, '$r24': 24, '$r25': 25, '$r26': 26, '$r27': 27, '$r28': 28,
    '$r29': 29, '$r30': 30, '$r31': 31
}
```

Initially, the assembler asks the programmer to provide the base address equivalent to the (program counter * 4); as we know, the address to store memory is a 4-byte alignment. Thus the valid addresses for the instructions are 0, 4, 8, 12... all of which is a multiple of 4. Hence, therefore, the assembler reports an error to the programmer if it isn't a valid address.

It takes the input of two different types from the text file or simple single-lined command lines. we provide at the beginning of its execution through try and except blocks in Python. It stores the command lines in a dictionary, "lines" and initiates PC(\$/4) or the address (\$) with the help of a string, "address" and an empty dictionary "label" to store the labels and their corresponding address value.

Pass 1:

1. It initiates with removing blank spaces within or unnecessary blank lines between the command lines of our assembly language code to free the programmers from indentation or unintentional blank space errors.
2. We initially iterate over the lines using the nav (navigator) variable set to zero using a while loop (Test expression: $\text{nav} < n$). This includes the empty strings of the assembly code. Thus, we initiate another variable count(counter) to store the address value in the corresponding count in the address array for command lines.
3. During the first pass, We will iterate through each line of the input code and find every label. whenever a new label is discovered, and its address is added to the symbol table (label). If the length of the command is zero, which means it is an empty string, we will skip to the next line, and the navigator(nav) will be incremented by 1 unit. If a command has a label, we will detect it by using an **If statement**, and we will separate it from the body of the command and store it along with the addresses it corresponds to in a dictionary called label{ }, and we will store the rest of the command in a command[] array and the address will be appended to the address array, and if the command doesn't have a label, the whole command without any white spaces or blanks at the ends will be stored into the command[] array and the corresponding address will be appended to the address[] array and at last the navigator and the count will be incremented by 1, this process continues for n times which represents the number of lines in assembly code.

```
while nav < (n):
    lines[nav] = lines[nav].strip() # trimming the unnecessary white spaces
    if (len(lines[nav]) == 0): #eliminating the blank lines
        nav += 1 #skipping to the next line
        continue

    elif (":" in lines[nav]):
        current_line = lines[nav].split(':')
        address.append(address[count] + 4) #st
        current_line[1] = current_line[1].strip()
        command.append(current_line[1])
        label_name = current_line[0].strip()
        label[label_name] = address[count]
        # stores the address count of the particular for further access in the future
        nav += 1
        count += 1

    else: #a valid instruction
        current_line = lines[nav].strip()
        command.append(current_line) #adding it to the list of instructions
        address.append(address[count] + 4) #update the address
        count += 1
        nav += 1
```

PASS 2:

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

In our project, we have divided the SDLX instructions into three formats, R, I, and J, which are further divided into different types depending on their structure. Following are all the classifications that we have made to perform pass 2 as follows

R-type:

- Type 1: oper rd, rs1, rs2
- Type 2: oper rd, rs1, sa

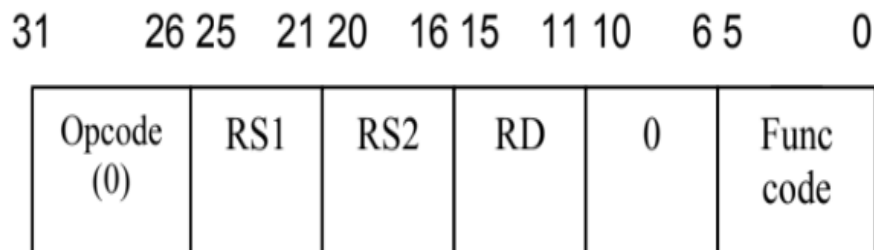
I-type

- Type 1: oper rd, rs1, imm
- Type 2: oper rd, rs1, sa
- Type 2: oper imm(rs1), imm
- Type 3: oper rd, imm(rs1)
- Type 4: oper rs1, label
- Type 5: oper rs1, imm (lhi \$rs1 0xffff00)

J-type:

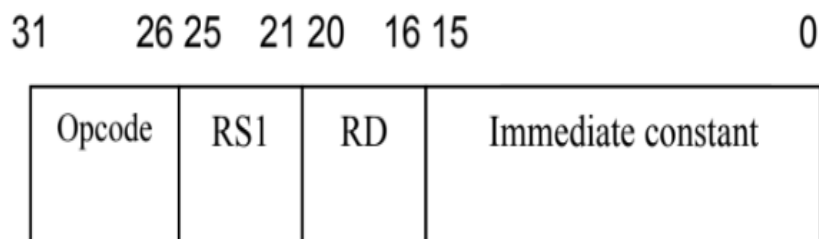
- Type 1: oper label
- As we stored the rest of the command in the command[] array, now each of the command arrays consists of 4 terms (Opcode, Rd, Rs1, and Rs2/ signed offset/ immediate constant) which are either separated by a comma or by space.
- Now by using the replace() function, we will replace all the commas with space, and later on, by using the split() function, we will split the command array into four parts and store them in a new array that is of length four. The first element in **a**, which is k (=a[0]) represents Opcode, and the second element in **a**, which is l(=a[1]), represents Rd, the third element in **a**, which is m(=a[2]) represents Rs1 and the fourth element in **a** which is n(=a[3]) represents Rs2 in R type triadic, represents an immediate constant in RI type triadic, represents the Signed offset in both R type dyadic and J type.

- As we created a dictionary of registers and Opcodes in the beginning, Opcode = opcodes[k], Rd = Registers[l], R1= Registers[m], R2 = Registers[n].
- As we want to convert the numbers of decimal form into binary form, we used the bin()[2:] function to do that, and as the registers are of a specific length, we used the zfill () function to fill the zeroes before the binary number if the length is less than required.
- The following notations were added to the code to simplify the program; these were kept out of the if the type of statements because these were not going to change in any type of format.
- R1_bina = bin(R1)[2:].zfill(5), opcode_bina= bin(opcode)[2:].zfill(6), Rd_bina = bin(Rd)[2:].zfill(5)
- Now there are 4 cases,
 - If Opcode is in range (1,21) it belongs to R type triadic
 - If Opcode is in range (21,42) it belongs to RI type triadic
 - If Opcode is in range (42,46) it belongs to R type dyadic
 - If Opcode is in the range (46,48) it belongs to J type
- The R-type triadic format is



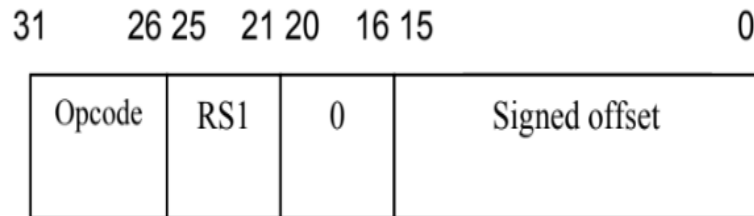
So R_type_triadic = '000000' + str(R1_bina) + str(R2_bina) + str(Rd_bina) + '00000' + str(opcode_bina)

- The RI-type triadic format is



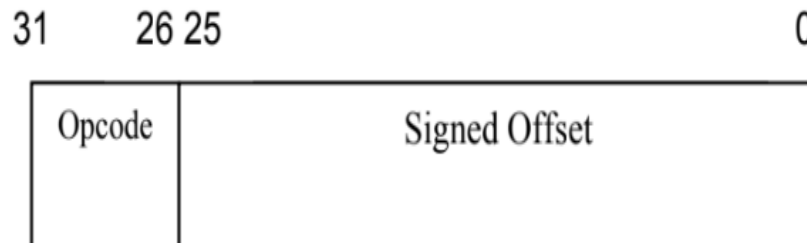
So `RI_type_triadic = str(opcode_bina) + str(R1_bina) + str(Rd_bina) + str(imm_bina)`,
 where `imm = int(a[3])` and `imm_bina = bin(imm)[2:].zfill(16)`

- The R-type dyadic format is



So `R_type_dyadic = str(opcode_bina) + str(R1_bina) + '00000' + str(imm_bina)`

- The J-type format is



So `J_type = str(opcode_bina) + str(signedoffset_bina)`, where `signedoffset = a[1]`,
`signedoffset_bina = bin(signedoffset)[2:].zfill(26)`.

- Now we will print out the final output, which is in the machine language nothing but in binary form of our given input.
- For further details, please look at the code in the given git hub link.

Advantages:

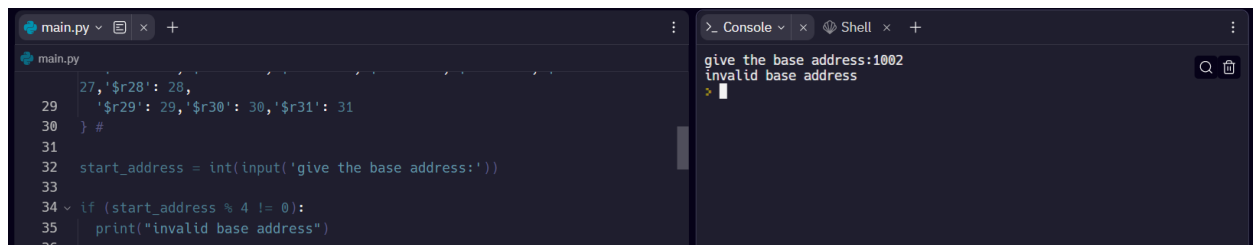
- This application will notify the user if there are any problems of any kind with the input provided. To accomplish this, try and except blocks that have been utilized throughout the code.

- This program will output the desired result for a variety of inputs. For example, there may be many commas or spaces between the registers, but because they replace and strip functions were used, the desired result will be created without displaying any problems.
- As we have used Python language, it is easier to understand for anyone who studies our program.
- We implemented all the concepts of SDLX processors and assembly language code, considering both text files and single command lines to the assembler as input
- We further encoded our assembler, considering the possibility of the programmer's mistakes. The assembler may be able to catch some of these errors. The effectiveness of assembler error signals significantly impacts how simple assembly language programming is.

Testing and Experiments:

1) Flags the error if the base address is not divisible by 4:

We gave the address as 1002 and produced an “invalid base address” as the output.



The screenshot shows a code editor with a file named `main.py` and a console window. The code in `main.py` is as follows:

```

27, '$r28': 28,
29, '$r29': 29, '$r30': 30, '$r31': 31
} #
31
32 start_address = int(input('give the base address:'))
33
34 ~ if (start_address % 4 != 0):
35     print("invalid base address")
36

```

The console window shows the input `give the base address:1002` and the output `invalid base address`.

2) Regardless of inputs that only differ in commas and spaces, produce the same outcome. For this, we first put commas between Rd and R1, then we took them out and looked at both of the outputs. Given that the only differences between the inputs are commas and blank spaces, it follows that this program yields identical results for both outputs.

```
give the base address:2000
give the file name:hi.txt
give the command line: addi $r22,,,$r21 100
010101101011011000000000001100100
> 
```

```
>_ Console x Shell x +
give the base address:2000
give the file name:hi.txt
give the command line: addi $r22 $r21 100
010101101011011000000000001100100
> 
```

3) As we previously stated, if there is an issue with the input, we want to let the user know. So we tried by giving the input \$r32 as Rd, which is an error.

```
>_ Console x Shell x test.txt x +
test.txt
1 addi $r32 $r2 100
```

```
>_ Console x Shell x test.txt x +
give the base address:1000
give the file name:test.txt
invalid command_line format in command line
> 
```

We can clearly see that the output produced is “ invalid command_line format in the command line, ” which proves our lemma.

4) Outputs are matched with the manual outcomes.

```
>_ Console x Shell x hi.txt x +
hi.txt
1 add,$r1, $r2, $r3
2 done:add $r2, $r4, $r6
3 bnez $r10 | done
4 loop:addi $r1 $r2 4
5 j loop
```

```
>_ Console x Shell x hi.txt x +
give the base address:1000
give the file name:hi.txt
00000000010000110000100000000001
00000000010000110000100000000001
1010100101000000111111111111110
01010100010000010000000000000100
1011101111111111111111111111110
```

Conclusion:

As shown in the above testing and experiments, we have manually tried various written assembly language code lines that give the desired and actual machine code.

Ultimately, we have developed an assembler for an SDLX processor that will provide output for text files, single command lines, and a variety of instructions.

Work distribution:

We split up the job equally among ourselves. Everyone contributed equally to writing the codes for the assembler, report writing, testing the written program, and debugging.

Sudarshan-Writing codes for Assembler, report writing, testing the written program, debugging

Hima Sagar -Writing codes for Assembler, report writing, testing the written program, debugging

Keerthi - Writing codes for Assembler, report writing, testing the written program, debugging`

REFERENCES:

- [1]. References are taken from Design for a simplified DLX (SDLX) processor by Professor Rajat Moona
- [2]. The DLX Instruction Instruction Set Set Architecture Architecture DLX Architecture Overview DLX Architecture Overview.” Accessed: Apr. 04, 2023. [Online]. Available: <https://www.cs.utexas.edu/~pingali/CS378/2015sp/lectures/DLX.pdf>
- [3] Q. Explain the following with respect to the design specifications of an Assembler: A) Data Structures B) pass1 & pass2 Assembler flow chart - Solved Assignments,” *sites.google.com*. <https://sites.google.com/site/assignmentssolved/mca/semester3/mc0073/3> (accessed Apr. 04, 2023).
- [4]. “AssemblerOrganization,” *www.d.umn.edu*. <https://www.d.umn.edu/~gshute/asm/assembler-organization.html>
- [5]. [Online]. Available: <https://www.geeksforgeeks.org/>