

Indian Institute of Technology Gandhinagar



Assembler for SDLX Processor

ES 215 Project Report

Members

Aaryan Darad (21110001)
Abhay Kumar Upparwal (21110004)
Vaibhavi Sharma (21110231)

4th April, 2023

Under the guidance of

Prof. Sameer G Kulkarni and Prof. Rajat Moona

CONTENTS

- 1) Abstract
- 2) Introduction
- 3) Ideation
- 4) Implementation
- 5) Verification
- 6) Limitations, Advantages and Further Scope
- 7) Acknowledgements and References

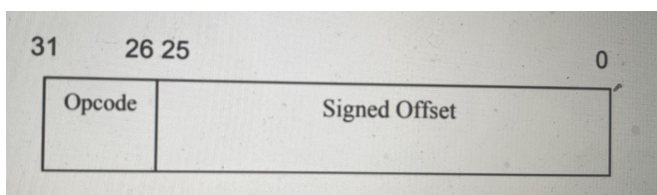
ABSTRACT

In the course Computer Organization and Architecture, we have studied the SDLX Processor and the different types of instructions it has. We have also studied about its datapath, internal circuiting and other functions related to it. In the previous assignments, we have been successfully able to make a completely functional SDLX processor on FPGA with machine code. However, for humans, understanding machine code is a very tedious task, and that's why we use assembly code to describe the instructions. This project dives deep into the same and consists of an assembler which converts assembly code to machine code (32-bit binary code).

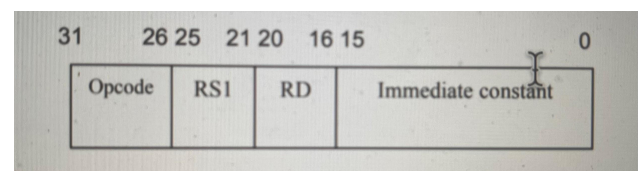
INTRODUCTION

The objective of this project is to design and implement an exhaustive SDLX Assembler which takes a set of instructions in assembly language and converts them into machine language code in binary format. We also aim to cover all types of instructions: R Type Triadic, R-I Type Triadic, R-I Type Dyadic, J Type and Load-Store instructions. This turns out to be an effective complement to the previously designed SDLX processor in Assignment 1,2 as the assembly language is easily understood by us humans, making it user friendly. Having made this project will enrich our understanding of the subject and will be great to apply the concepts taught in class and gain practical experience. This will turn out to be useful in the future as well when we would have to develop more complex systems.

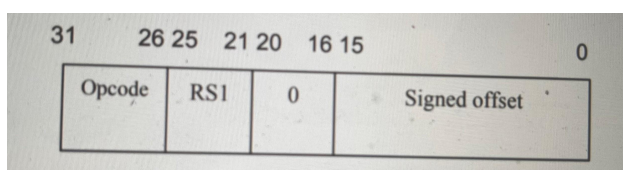
IDEATION



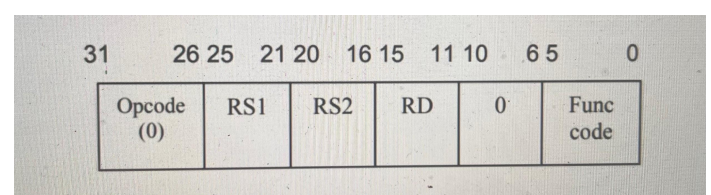
J Type



R-I Type Triadic



R-I Type Dyadic



R Type Triadic

We utilized the above instruction format for our machine code. We realized that the input Assembly code will be in string format which we would then convert to binary, hence we thought that using Python will be the best for our project as python is easy to deal with especially for strings. We have made the following data card which we referred to convert the assembly code into the required format:

R-I Type Triadic Instructions							R Type Triadic Instructions						
Assembly Format	Function	func code	opcode	op1	op2	stored in/from	Assembly Format	Function	func code	opcode	op1	op2	stored in
ADDI RS1 RD Imm	RS1+Imm=RD	-	000000	RS1	Imm	RD	ADD RS1 RS2 RD	RS1+RS2=RD	000000	-	RS1	RS2	RD
SUBI RS1 RD Imm	RS1-Imm=RD	-	000001	RS1	Imm	RD	SUB RS1 RS2 RD	RS1-RS2=RD	000001	-	RS1	RS2	RD
ANDI RS1 RD Imm	RS1&Imm=RD	-	000010	RS1	Imm	RD	AND RS1 RS2 RD	RS1&RS2=RD	000010	-	RS1	RS2	RD
ORI RS1 RD Imm	RS1 Imm=RD	-	000011	RS1	Imm	RD	OR RS1 RS2 RD	RS1 RS2=RD	000011	-	RS1	RS2	RD
XORI RS1 RD Imm	RS1^Imm=RD	-	000100	RS1	Imm	RD	XOR RS1 RS2 RD	RS1*RS2=RD	000100	-	RS1	RS2	RD
SLLI RS1 RD Imm	RS1<<Imm=RD	-	000101	RS1	Imm	RD	SLL RS1 RS2 RD	RS1<<RS2=RD	000101	-	RS1	RS2	RD
SRLI RS1 RD Imm	RS1>>Imm=RD	-	000110	RS1	Imm	RD	SRL RS1 RS2 RD	RS1>>RS2=RD	000110	-	RS1	RS2	RD
ROLI RS1 RD Imm	Rotate RS1 left by Imm	-	000111	RS1	Imm	RD	ROL RS1 RS2 RD	Rotate RS1 left by RS2	000111	-	RS1	RS2	RD
RORI RS1 RD Imm	Rotate RS1 right by Imm	-	001000	RS1	Imm	RD	ROR RS1 RS2 RD	Rotate RS1 right by RS2	001000	-	RS1	RS2	RD
SLTI RS1 RD Imm	Signed Less than Imm	-	001001	RS1	Imm	RD	SLT RS1 RS2 RD	Signed Less than	001001	-	RS1	RS2	RD
SGTI RS1 RD Imm	Signed Greater than Imm	-	001010	RS1	Imm	RD	SGT RS1 RS2 RD	Signed Greater than	001010	-	RS1	RS2	RD
SLEI RS1 RD Imm	Signed Less than or equal to Imm	-	001011	RS1	Imm	RD	SLE RS1 RS2 RD	Signed Less than or equal to	001011	-	RS1	RS2	RD
SGEI RS1 RD Imm	Signed greater than or equal to Imm	-	001100	RS1	Imm	RD	SGE RS1 RS2 RD	Signed greater than or equal to	001100	-	RS1	RS2	RD
UGTI RS1 RD Imm	Unsigned greater than Imm	-	001101	RS1	Imm	RD	UGT RS1 RS2 RD	Unsigned greater than	001101	-	RS1	RS2	RD
ULTI RS1 RD Imm	Unsigned greater than Imm	-	001110	RS1	Imm	RD	ULT RS1 RS2 RD	Unsigned greater than	001110	-	RS1	RS2	RD
ULEI RS1 RD Imm	Unsigned less than or equal to Imm	-	001111	RS1	Imm	RD	ULE RS1 RS2 RD	Unsigned less than or equal to	001111	-	RS1	RS2	RD
UGEI RS1 RD Imm	Unsigned greater than or equal to Imm	-	010000	RS1	Imm	RD	UGE RS1 RS2 RD	Unsigned greater than or equal to	010000	-	RS1	RS2	RD
SRAI RS1 RD Imm	RS1>>>Imm=RD	-	010001	RS1	Imm	RD	SRA RS1 RS2 RD	RS1>>>RS2=RD	010001	-	RS1	RS2	RD
LHI RS1 RD Imm	{Imm,RS1[31:16]}=RD	-	010010	RS1	Imm	RD	R Type Dyadic Instructions						
LB RD Imm(RS1)	Load Byte from RS1 address to RD	-	011010	RS1	Imm	RD	BEZ RS1 Imm	Branch PC by Imm if RS1=0	-	010101	RS1	Imm	-
LW RD Imm(RS1)	Load Word from RS1 address to RD	-	011011	RS1	Imm	RD	BNEZ RS1 Imm	Branch PC by Imm if RS1!=0	-	010110	RS1	Imm	-
LWH RD Imm(RS1)	Load Half Word from RS1 address to RD	-	011100	RS1	Imm	RD	JR RS1 Imm	Jump PC by RS1+ 4*Imm	-	011101	RS1	Imm	-
SB RD Imm(RS1)	Store Byte to RS1 address from RD	-	010111	RS1	Imm	RD(stored from)	JALR RS1 Imm	Jump PC by RS1+ 4*Imm and link to R31	-	011110	RS1	Imm	R31
SW RD Imm(RS1)	Store Word to RS1 address from RD	-	011000	RS1	Imm	RD(stored from)	J Dyadic Instructions						
SWH RD Imm(RS1)	Store Half Word to RS1 address from RD	-	011001	RS1	Imm	RD(stored from)	Assembly Format	Function	func code	opcode	op1	op2	stored in
							J Imm	Branch PC by Imm if RS1=0	-	010011	-	Imm	-
							JAL Imm	Branch PC by Imm if RS1!=0	-	010100	-	Imm	R31

Assembly code reference: [\[2\]](#)

Instruction reference: [\[3\]](#)

- We have used the same opcodes and func codes as we did in our Assignment 1 and 2. Also, for the Jump type instructions, we have incorporated the tag functionality in which the jump is to the instruction which is given a tag. For example if the instruction is: **J Loop** then it jumps to the Loop instruction which is tagged like: **Loop: BEZ R1 Done**.
- To implement we have divided all the instruction sets separately. Then we check for assembly language string character by character to get the correct machine code output.
- We have also incorporated for the comments, i.e if the statement starts with #, it will be ignored by the processor and will go ahead to the next instruction.
- We have tried to be as precise with the code as possible so that the code runs efficiently

IMPLEMENTATION

The implementation is uploaded in the file `Assembler.py` in the Github repository:

<https://github.com/daradaaryan/SDLX-Processor-Assembler>

The uploaded input will in the form of string, we will then identify the type of input using the `.split()` function which broadly divides the input into 7 categories according to the data card :

- I. R type Triadic: `INS RS1 RS2 RD`
- II. R-I type Triadic: Arithmetic `INSI RS1 RD Imm`
- III. Load/Store: `L/S(B/W/WH) RD Imm(RS1)`
- IV. R-I type Dyadic: `BEZ/BNEZ/J/JAL RS1 Imm`
- V. J type Jump `J/JAL Imm`
- VI. Tagged Instruction: Tag: `INS...`
- VII. `#This is a comment, will not be considered by the assembler.`

These will return a 32 bit machine code output.

- Any input apart from these types will be considered invalid by the assembler.
- Also, before entering the input string, in the first line we would have to enter the number of instructions we want to write so that the assembler knows when the assembly code ends.
- In the jump and branch instructions, we have put a tag string which will give the immediate offset according to the number of the tagged instruction.
- The opcodes and func codes are mapped to their corresponding values according to the data card. The registers and offsets get converted to the respective binary codes.
- Eventually the binary codes are combined to make 32 bit binary instruction according to the one mentioned in the ideation column.
- This is repeated for all instructions till the end of the assembly code is reached.

VERIFICATION

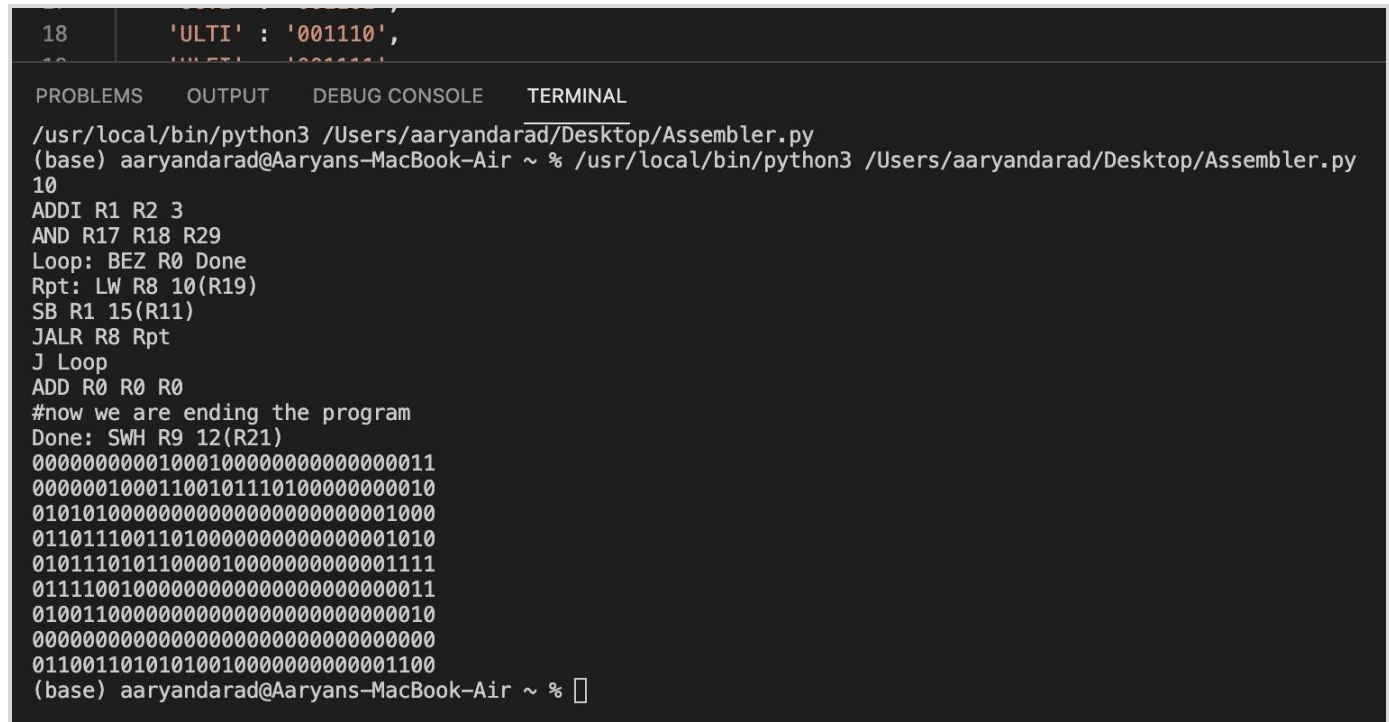
To verify the program written by us for the assembler, we ran a testcase consisting of the all instruction types, and cross verify them from the references [\[1\]](#), [\[3\]](#) and the datacard which have the instructions corresponding their machine code, assembly code and the format.

The testcase:

10

```
ADDI R1 R2 3 (expected output 000000 00001 00010 00000000000000011)
AND R17 R18 R29 (expected output 000000 10001 10010 11101 00000 000010)
Loop: BEZ R0 Done (expected output 010101 00000 00000 00000000000001000)
Rpt: LW R8 10(R19) (expected output 011011 10011 01000 00000000000001010)
SB R1 15(R11) (expected output 010111 01011 00001 00000000000001111)
JALR R8 Rpt (expected output 011110 01000 00000 00000000000000011)
J Loop (expected output 010011 00000000000000000000000000010)
ADD R0 R0 R0 (expected output 000000 00000 00000 00000 00000 000000) (NOPS)
#now we are ending the program (shouldn't give error)
Done: SWH R9 12(R21) (expected output 011001 10101 01001 00000000000001100)
```

Actual output:



```
18      'ULTI' : '001110',
19      'ULTI' : '000111'

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
/usr/local/bin/python3 /Users/aaryandarad/Desktop/Assembler.py
(base) aaryandarad@Aaryans-MacBook-Air ~ % /usr/local/bin/python3 /Users/aaryandarad/Desktop/Assembler.py
10
ADDI R1 R2 3
AND R17 R18 R29
Loop: BEZ R0 Done
Rpt: LW R8 10(R19)
SB R1 15(R11)
JALR R8 Rpt
J Loop
ADD R0 R0 R0
#now we are ending the program
Done: SWH R9 12(R21)
00000000001000100000000000000011
00000010001100101110100000000010
0101010000000000000000000001000
0110111001101000000000000001010
01011101011000010000000000001111
0111100100000000000000000000011
0100110000000000000000000000010
0000000000000000000000000000000
0110011010100100000000000001100
(base) aaryandarad@Aaryans-MacBook-Air ~ %
```

As shown in the terminal output, the expected output matches the actual output from the assembler from all types of instructions, hence we have verified that the assembler works correctly.

LIMITATIONS, ADVANTAGES AND FURTHER SCOPE

Advantages:

- The assembler accounts for all instructions of the SDLX processor and hence it is an complete assembler.
- We have incorporated the Tag functionality, so that user can directly jump to the tag without worrying much about the position of the instruction.
- We have incorporated the comment functionality so that user can add comments to keep track of the code.

Limitations:

- Although we have considered the comments for an output, but the comments only work if the hash is the first character of the line, else it shows error
- The instructions should be entered exactly according to the syntax even without any extra spaces etc. or else the program won't be able to run and will throw an error

Further Scope:

- We can expand the comment functionality so that we can add comments in the same line as the assembly code.
- We can make the code flexible to the type of syntax used, eg. interchangeable using commas/spaces without worrying about the output.
- Standard machine language is binary but have it in the form of hexadecimal code can be easier to understand and to debug for the user.

ACKNOWLEDGEMENTS

The members of our group would like to thank Prof. Sameer G Kulkarni and Prof. Rajat Moona as we learned a lot about the SDLX Processor and how we can implement it ourselves by coding it up in FPGA. We were also able to make an assembler for the processor on Python which made our understanding of the ES 215 course more concrete. It is always great to implement something from scratch and see it in front of our eyes run as we studied in the class. We hope that this helps us in making more complex projects in the future.

REFERENCES

[1]: SDLX Processor design by Prof. Rajat Moona

<https://drive.google.com/file/d/1dN9sZt886xOZsq7g0u54Eb1hvmtWWxRP/view>

[2]: Assembly code reference from ES 215 Tutorial 6 Solutions by Prof. Sameer Kulkarni

https://drive.google.com/file/d/1hnbw6Q9pHB_9JLfz_BPzFueguXEXyjP8/view

[3] Opcode reference from ES 215 Assignment 1 and 2 by Aaryan, Abhay, Vaibhavi

<https://github.com/daradaaryan/ES215-COA-Aaryan-Abhay-Vaibhavi>