CPUseless!™

(cpu name ^)

Featuring……

NSL: No`Space`Lang™

(language name ^)

The world’s first programming language that doesn’t require spaces!
The official instruction manual (version 1.0.0)

Jesal`Gandhi
Dhihan`Ahmed

I`pledge`my`honor`that`I`have`abided`by`the`Stevens`Honor`System`.

Syntax/Available Instructions

PL instruction

- PL instruction performs addition
- Stores the result of sum of registers T1 and T2 into destination register T0
- Opcode for this instruction is always 01.
- *Format*: PL`Td`Tm`Tn

Example instruction: PL`T0`T1`T2

01	00	01	10
Opcode	destination register Td	1st register Tm	2nd register Tn

Machine code: 01000110b = 0x46

MI instruction

- MI instruction performs subtraction
- Stores the result of difference of registers T1 and T2 into destination register T0

- Opcode for this instruction is always 00.
- *Format*: MI`Td`Tm`Tn

Example instruction: MI`T0`T1`T2

00	00	01	10
Opcode	destination register Td	1st register Tm	2nd register Tn

Machine code: 00000110b = 0x06

LD instruction

- LD instruction loads data into
- Stores the data at the address of register Tm + offset (Tn) in data memory inside of register Td
- Opcode for this instruction is always 10.
- *Format*: LD`Td`Tm`Tn

Example instruction: LD`T0`T1`T2

10	00	01	10
Opcode	destination register Td	Address of register Tm	offset (register Tn)

Machine code: 10000110b = 0x86

Language Specification

- The file format for this language is a **.nsl file** extension
- The file consists of two sections:
    - The first section is **.text** (comes first in the file), which contains all instructions for the program
        - Each instruction is separated by a newline
    - Directly after the instructions section is **.data**, which contains all the data for the program.
        - Each line in the .data section contains one byte of data, with each byte being separated by a newline.

        - All values are encoded as their respective ascii values, when being written to data memory

Example of an .nsl file:

```
.text
PL`T0`T1`T2
MI`T2`T1`T3
PL`T1`T2`T0
LD`T2`T1`T3
.data
1
0
20
50
100
150
200
250
255
c
m
A
B
```

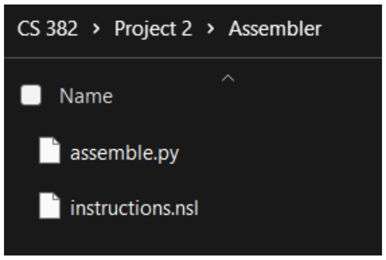
Running the Assembler

- To run the assembler, create a new directory and place assemble.py and an .nsl file in the same directory
    - You can use the provided .nsl file (instructions.nsl) to test, or create your own .nsl; just make sure it is in the same directory as assemble.py!
- Write the instructions and data in the .nsl file. Please see the language specification section if you need a refresher on how to write the .nsl file
- Open a terminal in the directory and run python3 assemble.py. You will know the process succeeded if there are two new files in the directory and you receive this message: “Successfully wrote instructions to image file!”
- The two files generated by the assembler are: **instruction_mem_img** and **data_mem_img**

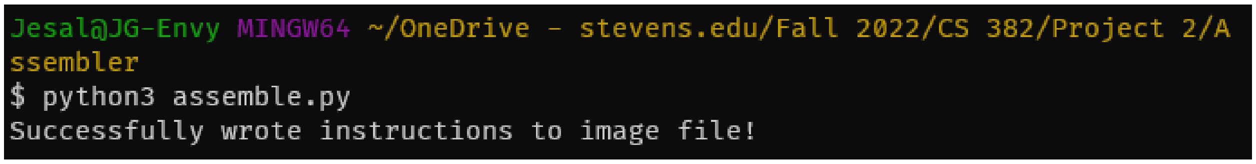
- **Instruction_mem_img** is the image file that contains the instructions for the program, and will be loaded into instruction memory
- **data_mem_img** is the image file that contains the data for the program, and will be loaded into data memory

Example of running the assembler on an .nsl file and loading the image files into logisim:

The assemble.py and .nsl file are both in the same directory

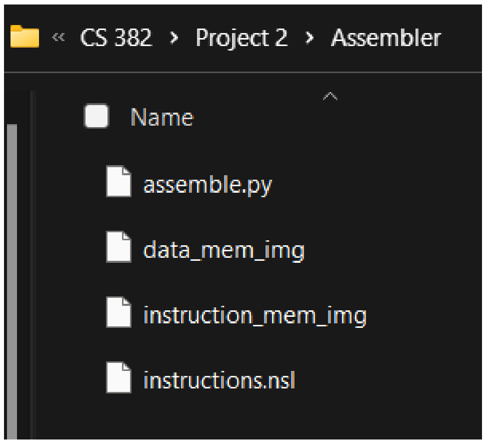


We open the terminal in this directory, and run python3 assemble.py. We receive the success message, so we know the process worked.

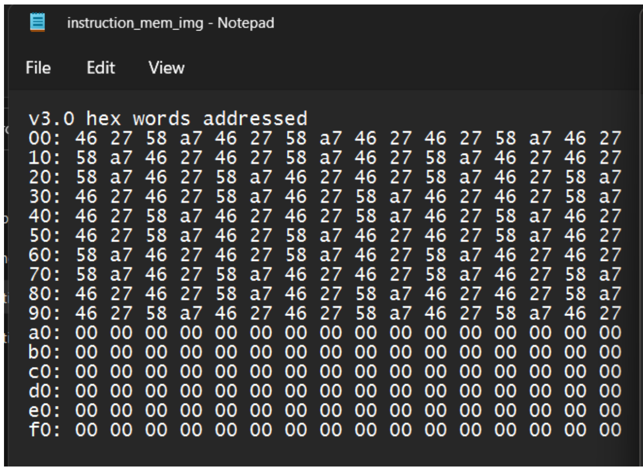


```
Jesal@JG-Envy MINGW64 ~/OneDrive - stevens.edu/Fall 2022/CS 382/Project 2/A
ssembler
$ python3 assemble.py
Successfully wrote instructions to image file!
```

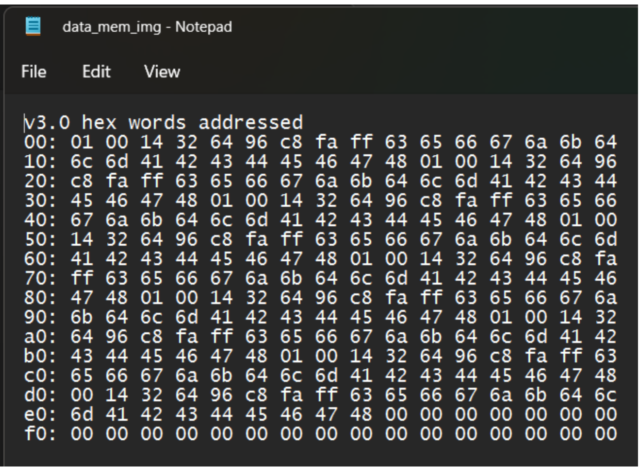
Indeed, if we head back to the directory, there are two new files – instruction_mem_img and data_mem_img!



Below shows the general format of instruction_mem_img (left) and data_mem_img (right). Of course, the values of the bytes will change depending on the data in your .nsl file.

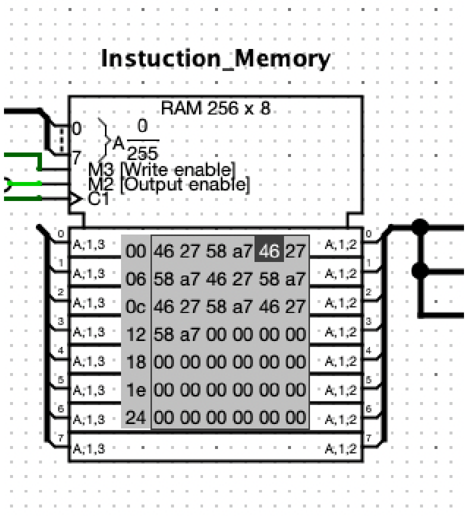


```
v3.0 hex words addressed
00: 46 27 58 a7 46 27 58 a7 46 27 46 27 58 a7 46 27
10: 58 a7 46 27 46 27 58 a7 46 27 58 a7 46 27 46 27
20: 58 a7 46 27 58 a7 46 27 46 27 58 a7 46 27 58 a7
30: 46 27 46 27 58 a7 46 27 58 a7 46 27 46 27 58 a7
40: 46 27 58 a7 46 27 46 27 58 a7 46 27 58 a7 46 27
50: 46 27 58 a7 46 27 58 a7 46 27 46 27 58 a7 46 27
60: 58 a7 46 27 46 27 58 a7 46 27 58 a7 46 27 46 27
70: 58 a7 46 27 58 a7 46 27 46 27 58 a7 46 27 58 a7
80: 46 27 46 27 58 a7 46 27 58 a7 46 27 46 27 58 a7
90: 46 27 58 a7 46 27 46 27 58 a7 46 27 58 a7 46 27
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

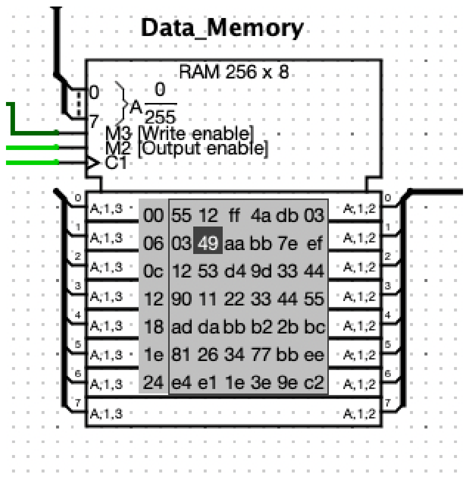


```
v3.0 hex words addressed
00: 01 00 14 32 64 96 c8 fa ff 63 65 66 67 6a 6b 64
10: 6c 6d 41 42 43 44 45 46 47 48 01 00 14 32 64 96
20: c8 fa ff 63 65 66 67 6a 6b 64 6c 6d 41 42 43 44
30: 45 46 47 48 01 00 14 32 64 96 c8 fa ff 63 65 66
40: 67 6a 6b 64 6c 6d 41 42 43 44 45 46 47 48 01 00
50: 14 32 64 96 c8 fa ff 63 65 66 67 6a 6b 64 6c 6d
60: 41 42 43 44 45 46 47 48 01 00 14 32 64 96 c8 fa
70: ff 63 65 66 67 6a 6b 64 6c 6d 41 42 43 44 45 46
80: 47 48 01 00 14 32 64 96 c8 fa ff 63 65 66 67 6a
90: 6b 64 6c 6d 41 42 43 44 45 46 47 48 01 00 14 32
a0: 64 96 c8 fa ff 63 65 66 67 6a 6b 64 6c 6d 41 42
b0: 43 44 45 46 47 48 01 00 14 32 64 96 c8 fa ff 63
c0: 65 66 67 6a 6b 64 6c 6d 41 42 43 44 45 46 47 48
d0: 00 14 32 64 96 c8 fa ff 63 65 66 67 6a 6b 64 6c
e0: 6d 41 42 43 44 45 46 47 48 00 00 00 00 00 00 00
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

In terms of Logism itself, make sure to load the instruction_mem_img file into the RAM on the left, the one that has the label Instruction_Memory. (The hexadecimal digits in the below photo are random, when you actually load the image file it should match the bytes in your own file).

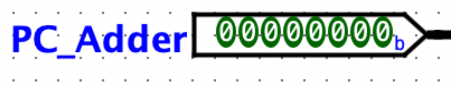


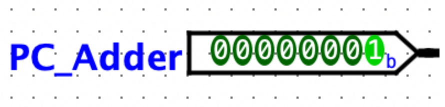
Load the data_mem_img file into the RAM on the right, the one that has the label Data_Memory.



```
00 55 12 ff 4a db 03
06 03 49 aa bb 7e ef
0c 12 53 d4 9d 33 44
12 90 11 22 33 44 55
18 ad da bb b2 2b bc
1e 81 26 34 77 bb ee
24 e4 e1 1e 3e 9e c2
```

All the pins in Logisim are what they should be **EXCEPT** the pin at the very top left, labeled PC_Adder. When normally opening a Logism file, the PC_Adder pin starts at 0. The PC_Adder has to be set to 00000001 in order to properly execute your instructions in Instruction_Memory in order.





Before -> After

Architectural Description:

- Our CPU was designed with 8 bits, as 8 bits provides enough space for 4 general purpose registers (T0 – T3) and 4 instructions
    - This is enough for our CPU, as the main instructions we needed were two arithmetic instructions (addition and subtraction) and a memory instruction (loading). Notice, we do not have a 4th instruction, as it was not necessary.
- Our program uses T registers, and there are 4 general purpose T registers ranging from T0 – T3
- Each register has a 2 bit capacity, thus giving 4 possible combinations with 8 bits.
    - They can be T0, T1, T2, or T3.
- For all instructions, we use all 8 bits. Bits 0-1 of every instruction are the opcode, and bits 2-3 of every instruction are the destination register Td.
    - For PL/MI: bits 4-5 are for register 1 Rm, and bits 6-7 are for register 2 Rn
    - For LD: bits 4-5 are for address of register Tm, and bits 6-7 are for register Tn, which contains the offset

Job Descriptions

Dhihan:

- Made the CPU in Logisim – specifically tailored to line up with the formatting of the instructions that Jesal and I previously decided on, making sure the right bits controlled what they were supposed to
- Tested edge cases and specific instruction inputs to ensure CPU outputted the desired result, along with ensuring the files properly loaded into their respective RAMs
- Made a separate circuit for the ALU to make the hardware look neater
- Wrote the User Manual PDF alongside Jesal
- **Came up with the dope CPU name****

Jesal:

- Wrote the assembler in python – works with a special extension we created called .nsl and generates both an instruction memory image file and a data memory image file
    - Both of these image files are to be loaded into the CPU into their respective memories
- Designed the assembler such that *any* file in the directory can be assembled, as long as it has the .nsl extension. An error is thrown if no .nsl is found in the same directory as the assembler.
- Assisted with the CPU design, specifically the ALU and how it performs the basic arithmetic operations
- Wrote the User Manual PDF alongside Dhihan