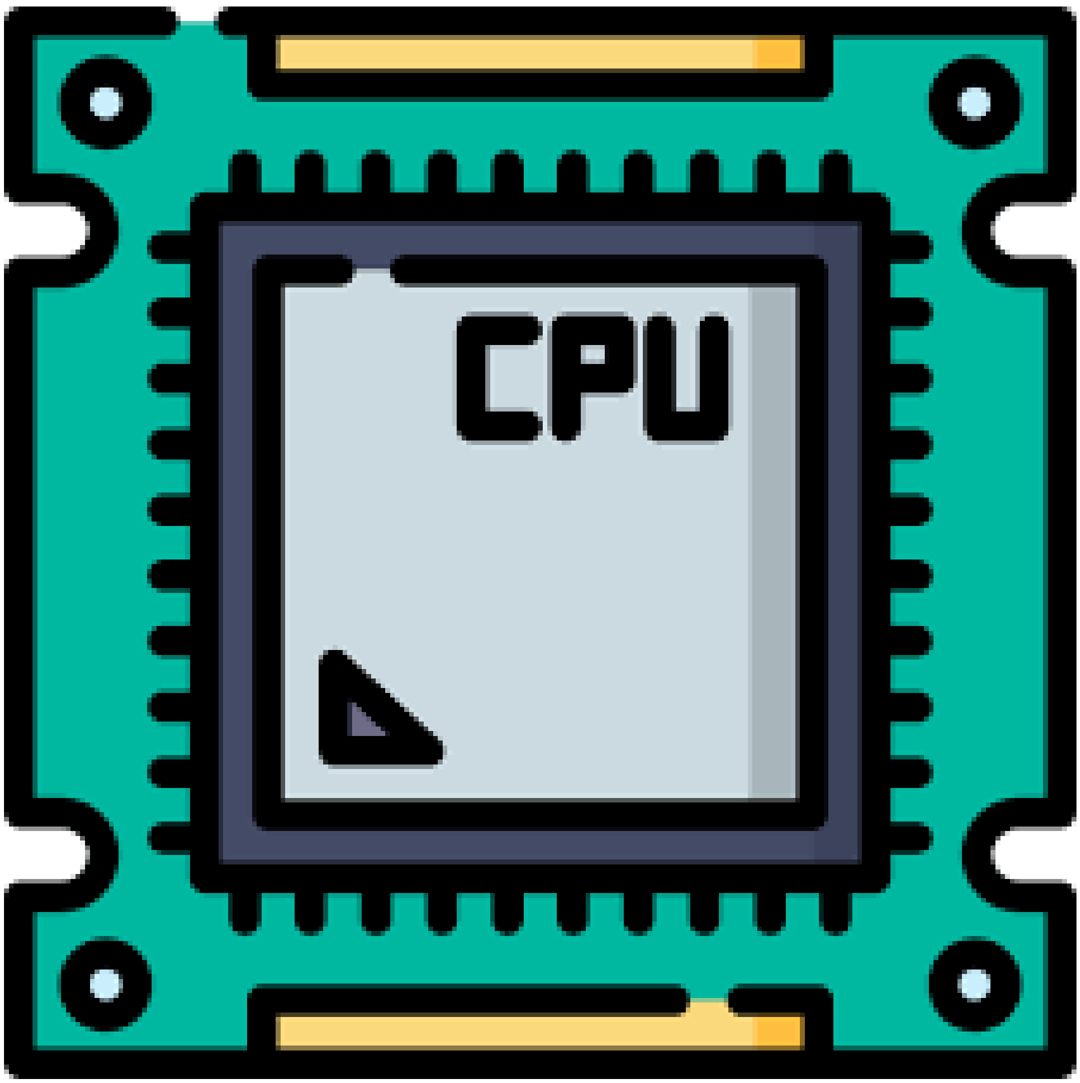
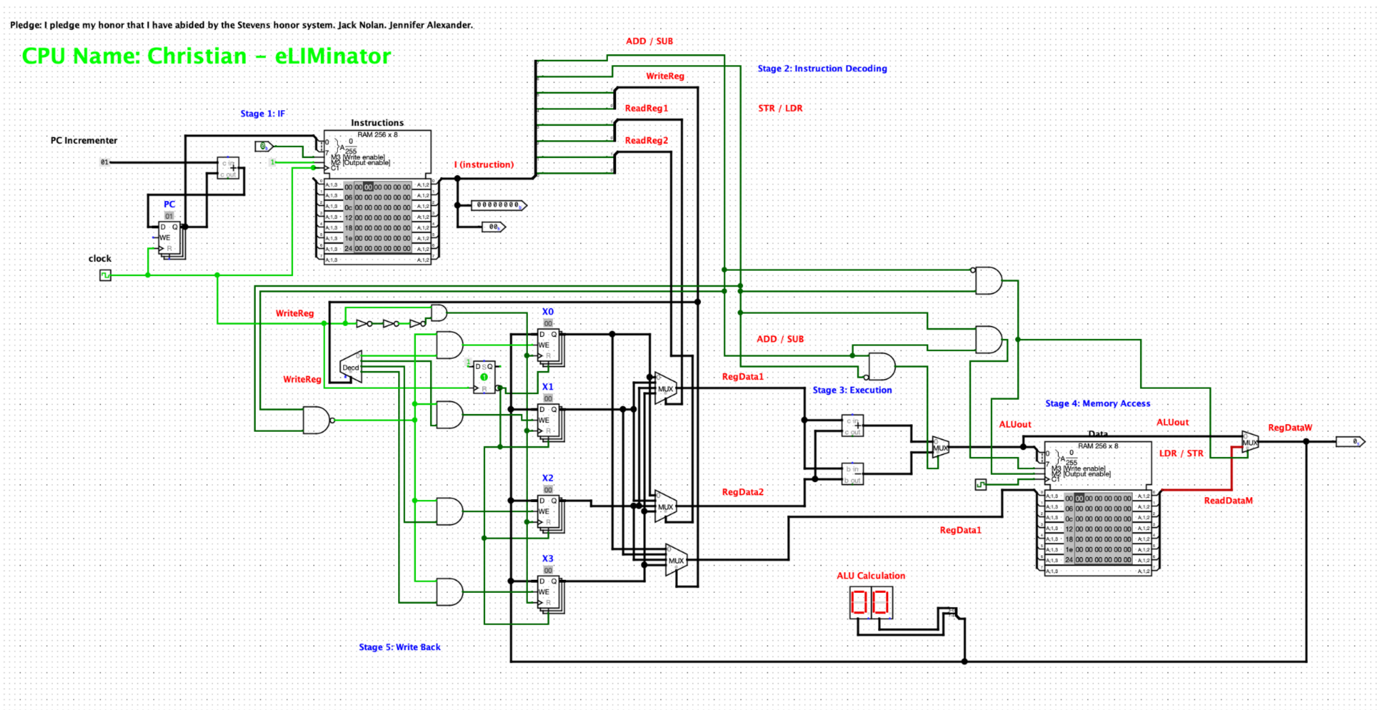
CPU User Manual



CPU Architecture Description



4 General Purpose Registers:

- Referred to as "X0", "X1", "X2", "X3" in .lim assembly programs

CPU Functions:

- *Addition* - add two values from registers and store the result into a register. Addition instruction is referred to as "ADD" in assembly programs.
- *Subtraction* - subtract one register value from another register value, and store this result into a register. Subtraction instruction is referred to as "SUB" in assembly programs.
- *Load* - load a value from data memory and store it into a register. Load instruction is referred to as "LDR" in assembly programs.
- *Store* - store a value from a register into data memory. Store instruction is referred to as "STR" in assembly programs.

Instruction Set Architecture for the lim Programming Language

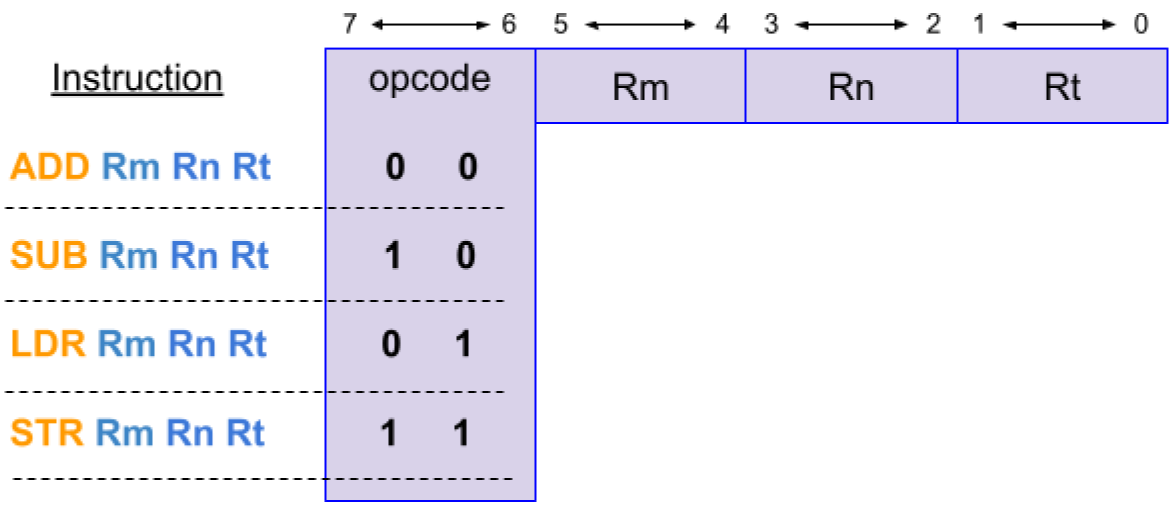


Figure: Encodings of all CPU instructions with the register operands.

General Description of Binary Encodings:

The binary encoding of every instruction is represented by 8 bits:

- Bits 7 and 6 are used to represent the opcode for the instruction. I am using two bits to represent the opcode, since there are 4 instructions, so there is a unique bit combination for each instruction.
- Bits 5 and 4 represent the register Rm. I used 2 bits for this register because the CPU has 4 registers, so each register can uniquely be identified using 2 bits.
- Bits 3 and 2 represent the register Rn. I used 2 bits for this register because the CPU has 4 registers, so each register can uniquely be identified using 2 bits.

- Bits 1 and 0 represent the register Rt. I used 2 bits for this register because the CPU has 4 registers, so each register can uniquely be identified using 2 bits.

Instruction #1: ADD Rm Rn Rt

- The "ADD" instruction performs the addition ('+') operation
- "ADD" will take the values stored in the registers Rn and Rt, add them together, and store the sum into the register Rm
- *Example:* ADD X0 X1 X2
    - ADD X0 X1 X2 will add the values stored in X1 and X2 together, then store this sum into the register X0
- Example Encoding: ADD X0 X1 X2 → binary encoding → 00000110b

Instruction #2: SUB Rm Rn Rt

- The "SUB" instruction performs the subtraction ('-') operation
- "SUB" will take the values stored in the registers Rn and Rt, then subtract the value stored in Rt from the values stored in Rn, and store this difference into the register Rm
- *Example:* SUB X3 X0 X2
    - SUB X3 X0 X2 will subtract the value stored in X2 from the value stored in X0, then store this difference into the register X3
- Example Encoding: SUB X3 X0 X2 → binary encoding → 10110010b

Instruction #3: LDR Rm Rn Rt

- The "LDR" instruction load a value from data memory, and store it into a register

- “LDR” will take the value stored in data memory, at the address from Rn (offset by the value stored in Rt), and store this data into the register Rm
- *Example:* LDR X0 X1 X2
    - LDR X0 X1 X2 will load the value from data memory that is stored at the memory address X1 (address) + X2 (offset), and store it into the register X0
- Example Encoding: LDR X0 X1 X2 → binary encoding → 01000110b

<u>Instruction #4:</u> STR Rm Rn Rt

- The “STR” instruction stores a value from a register into data memory
- “STR” will store the value from Rm, into data memory at the address stored in Rn (offset by the value stored in Rt)
- *Example:* STR X2 X0 X3
    - STR X2 X0 X3 will store the value from X2, into data memory at the address X0 (address) + X3 (offset)
- Example Encoding: LDR X0 X1 X2 → binary encoding → 01000110b

Example ‘.lim’ Written in the Lim Programming Language:

hello_world.lim

```
/* I pledge my honor that I have abided by the Stevens honor system. Jennifer Alexander. Jack Nolan. */
.text
.global _start

_start:
    ADD X0 X0 X0
    ADD X1 X0 X3
    SUB X1 X2 X2
    STR X1 X2 X3
    LDR X0 X2 X1
    ADD X3 X0 X0
    ADD X1 X3 X0
    SUB X1 X2 X3
    STR X0 X2 X0
    LDR X2 X2 X3
    ADD X0 X2 X0
    ADD X1 X0 X1
    SUB X2 X0 X2
    STR X0 X2 X2
    LDR X0 X1 X1
    ADD X2 X0 X0
    ADD X1 X0 X3
    SUB X1 X3 X3
    STR X3 X2 X3
    LDR X0 X0 X3
    ADD X0 X0 X1
    ADD X2 X0 X3
    SUB X1 X0 X1
    STR X0 X2 X2
    LDR X1 X3 X3
    ADD X2 X0 X3
    ADD X1 X2 X2
    SUB X2 X2 X1
    STR X2 X1 X0
    LDR X0 X2 X0



.data
    20
    200
    1
    0
    10
    20
    32
    0
    232
    154
```

How to Use the Assembler Program assembler.py

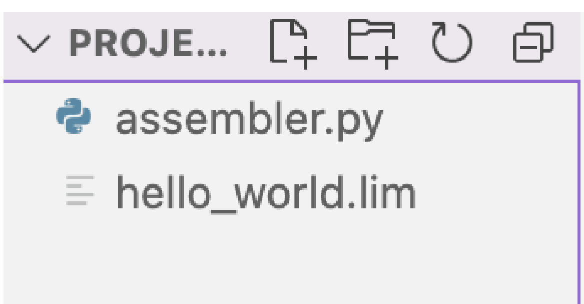
The assembler for the Christian-eLIMinator CPU is written in Python and named `assembler.py`. Its purpose is to process a text file containing an assembly program and produce two output files in image format. These files are designed to be loaded into Logisim's RAM for execution on the CPU.

The assembler begins by reading the assembly file, separating the instructions into one list and the data elements (declared in the `.data` segment) into another. It then translates the instructions using predefined opcodes and register codes, converting them first into binary and then into hexadecimal values. Similarly, the data elements, which are numerical values, are also converted into their hexadecimal equivalents.

Finally, the assembler generates two image files named `instructions` and `data`, encoding the hexadecimal values into a format compatible with the Logisim CPU. These files enable the assembly program to be loaded into RAM and executed on the Christian-eLIMinator CPU.

To use the assembler, follow these steps:

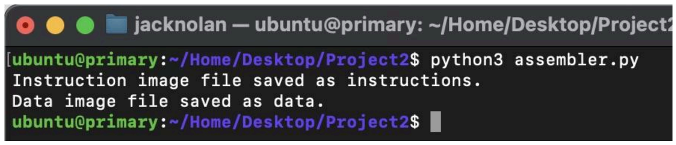
1. Create ythe code in a `.lim` file, including all instructions and data.
2. Place the `.lim` file and `assembler.py` in the same directory.

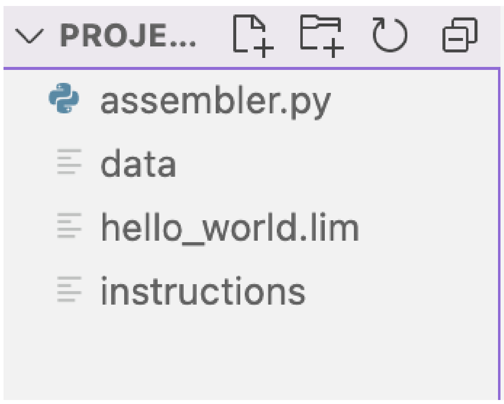


3. Open a terminal and navigate to the directory containing both the `assembler.py` and `.lim` files.
4. In the terminal, run the following command:

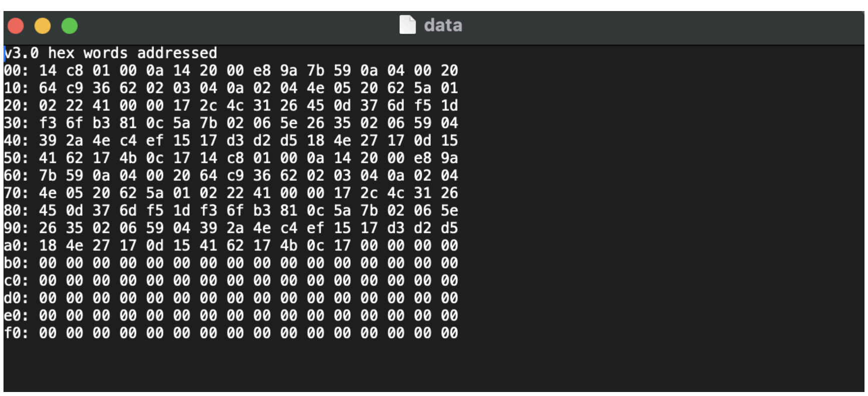
python3 assembler.py

5. If the assembler runs successfully, you should see two new image files generated in the same directory: `data` and `instructions`.





6. Next, open the CPU model in Logisim. Click on the memory, then navigate to the directory where the image files were generated. The image files should look similar to these below:

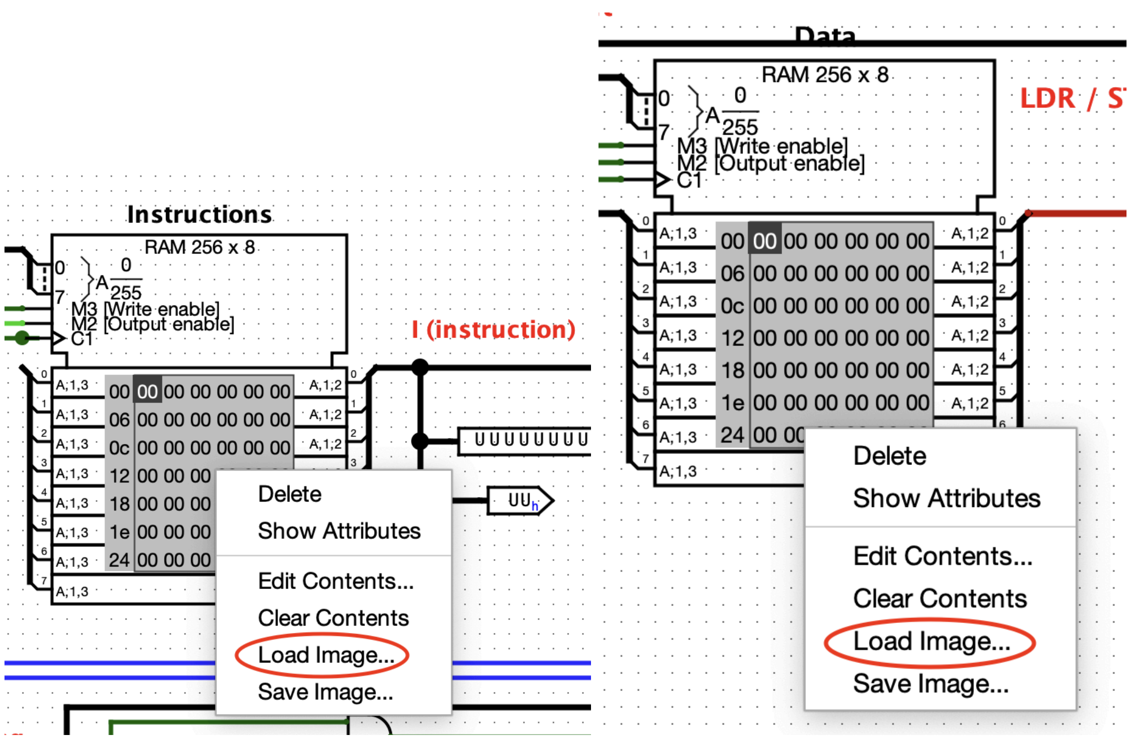



```
v3.0 hex words addressed
00: 14 c8 01 00 0a 14 20 00 e8 9a 7b 59 0a 04 00 20
10: 64 c9 36 62 02 03 04 0a 02 04 4e 05 20 62 5a 01
20: 02 22 41 00 00 17 2c 4c 31 26 45 0d 37 6d f5 1d
30: f3 6f b3 81 0c 5a 7b 02 06 5e 26 35 02 06 59 04
40: 39 2a 4e c4 ef 15 17 d3 d2 d5 18 4e 27 17 0d 15
50: 41 62 17 4b 0c 17 14 c8 01 00 0a 14 20 00 e8 9a
60: 7b 59 0a 04 00 20 64 c9 36 62 02 03 04 0a 02 04
70: 4e 05 20 62 5a 01 02 22 41 00 00 17 2c 4c 31 26
80: 45 0d 37 6d f5 1d f3 6f b3 81 0c 5a 7b 02 06 5e
90: 26 35 02 06 59 04 39 2a 4e c4 ef 15 17 d3 d2 d5
a0: 18 4e 27 17 0d 15 41 62 17 4b 0c 17 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
```

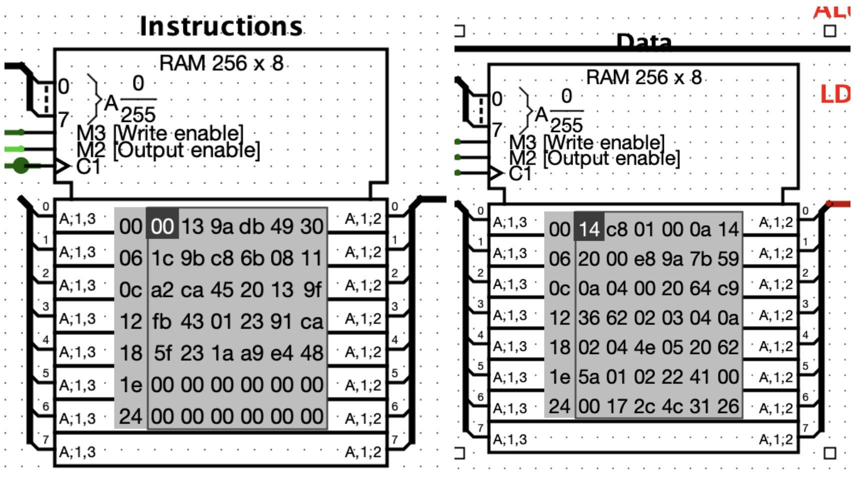

instructions

```
v3.0 hex words addressed
00: 00 13 9a db 49 30 1c 9b c8 6b 08 11 a2 ca 45 20
10: 13 9f fb 43 01 23 91 ca 5f 23 1a a9 e4 48 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
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
```

7. Load the `data` and `instructions` image files into the memory in Logisim. To do this, open up the "ChristianeLIMinator.circ" file. Locate the "Instructions" and "Data" memories, left-click on them, and select "Load Image…". Then navigate to the directory where the data and instructions image files were generated, and load them into the respective memories. Now you are ready to simulate ythe .lim program using the Christian eLIMinator CPU in Logisim!



This is what the memory should look like once you load the generated image files correctly.



By following these steps, you'll successfully assemble the code and load into the CPU model.