

ShuPU User Manual

Sarang Hadagali
Dev Patel

I pledge my honor that I have abided by the Stevens honor system. - Sarang and Dev

Job Description

Sarang was in charge of building the assembler that would read Shu++ from a .txt file and assemble it into binary machine code that the CPU could understand. The assembler was built using Python3. It starts by fetching a .txt file that the user wants to run. The assembler will use dictionaries to find a mapping between the commands and the related binary encoding and then write it out into another .txt file called machine_code.txt, which will then be loaded into the instruction memory by the Logisim CPU. He also completed the manual, which included writing the job descriptions, describing how to use Shu++ as well as compile and run instructions, the overall architecture of the CPU, what functions the CPU can compute and finally a detailed description of how each instruction can be implemented.

Dev was in charge of the hardware portion of the project. He specialized in the design and execution of the CPU. He began this process by discussing the general layout of each major component such as the program counter, RAM, register file and ALU. He proceeded by figuring out what commands required what type of binary encoding ex. (ADD R1, R2, R2 -> 00 01 10 10), which he then relayed to Sarang so that they were able to build the hardware and software simultaneously. He also was in charge of testing the CPU, he tried breaking both the code and CPU, by trying various edge cases that would ensure that the CPU covered all areas of error. If the code or CPU had an issue, he either worked on fixing the hardware or told Sarang about the various code issues there were so that they could be fixed.

How to Use Shu++

1. Download the Project_Two project
2. Navigate into the Script folder Project_Two->Script
3. Create a .txt file naming it whatever you would like, make sure that the file is within the same directory as the generate_img.py script otherwise the instruction set will not work
 - a. In this example a sample instructions.txt will be used to explain the process
4. Within the instructions.txt file first create and allocate the data that will be stored in the memory section, it should be noted that the data memory can only hold 4 bits of memory, meaning 0-15 is bounds that the data segment goes until
 - a. The format is as follows write out "data"
 - b. To set a specific location in data memory to a value type
 - i. 0 = 12
 - ii. This will set the first spot in memory as the value 12
 - iii. You can continue this process from 0-15 setting whatever values within an 8 bit range

```

instructions.txt — Edited
data
0 = 0
1 = 1
2 = 2
3 = 3
4 = 78
5 = 90
6 = 122
7 = 12
8 = 2
9 = 1
10 = 98
11 = 34
12 = 90
13 = 29
14 = 78
15 = 34

```

iv.

5. The next component is to create an instruction set that will be computed
 - a. Start by loading in data into registers following the format
 - i. load r0 4
 - ii. Looking at the sample instructions.txt, the previous instruction would result in loading 78 into register 0
 - b. Continue this process until the necessary registers are loaded with the desired values

```

instructions.txt — Edited
load r0 4
load r1 1
load r2 2
load r3 3
data
0 = 0
1 = 1
2 = 2
3 = 3
4 = 78
5 = 90
6 = 122
7 = 12
8 = 2
9 = 1
10 = 98
11 = 34
12 = 90
13 = 29
14 = 78
15 = 34

```

c.

- d. Next, start the computing, to choose there is sum and diff
 - i. Placing sum r0 r1 r2 right below load r3 3 will result in r0 = 3
 - ii. Placing diff r0 r0 r2 below sum r0 r1 r2 will result in r0 = 1

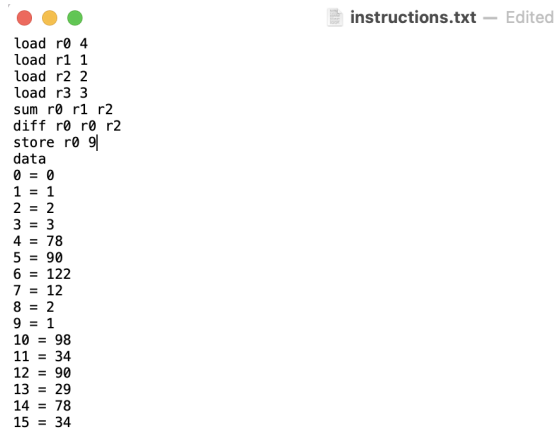
```

instructions.txt — Edited
load r0 4
load r1 1
load r2 2
load r3 3
sum r0 r1 r2
diff r0 r0 r2
data
0 = 0
1 = 1
2 = 2
3 = 3
4 = 78
5 = 90
6 = 122
7 = 12
8 = 2
9 = 1
10 = 98
11 = 34
12 = 90
13 = 29
14 = 78
15 = 34

```

- e. Finally, to store data back into the data memory follow the format
 - i. store r0 9

- ii. Placing this instruction underneath `diff r0 r0 r2` will result in storing `r0 = 1` into memory slot 9



The screenshot shows a text editor window titled "instructions.txt - Edited". The content is as follows:

```
load r0 4
load r1 1
load r2 2
load r3 3
sum r0 r1 r2
diff r0 r0 r2
store r0 9
data
0 = 0
1 = 1
2 = 2
3 = 3
4 = 78
5 = 90
6 = 122
7 = 12
8 = 2
9 = 1
10 = 98
11 = 34
12 = 90
13 = 29
14 = 78
15 = 34
```

- iii.
- iv. It should be noted that between each instruction there should be a new line, there should not be any extra new lines between instructions and there should be a space between each word
- f. With the instruction set done, it is time to assemble `instructions.txt`
 - i. Navigate back to `generate_img.py`
 - ii. Run the script
 1. The script takes in the command line argument `->` enter `instructions.txt`
 2. Make sure that the file is spelled correctly or the file will not compile
 - iii. Once the script has been run, notice that two files were created within the `scripts` directory
 1. `machine_code.txt`
 2. `data_code.txt`
- 6. Once the two `.txt` files have been created it is time to move to Logisim
 - a. Move back into the `Project_Two` folder and open the `cpu.circ` file in Logisim
 - b. In the CPU navigate to the instruction memory and right click it, an option to load image, select the `machine_code.txt` which should load in the hex format that the instruction memory segment takes in
 - c. Next navigate to the data memory and right click it, once again right click it and an option to load image will be there, select `data_code.txt` which should load in the the hex format that the data segment takes in
 - d. Finally, press the clock that connects to the PC to manually run it
 - e. Or to run the CPU automatically, select the `simulate` tab in Logisim then select the desired clock rate in `Auto-Tick frequency`, then press `auto-tick enabled`
 - f. This should run the CPU and the desired output of the `instructions.txt` should compute
 - g. Congratulations!!! The process has been complete and the output should have been computed

CPU Architecture

The architecture of the CPU consists of 4 general purpose registers, a RAM with 256 bytes storing the instruction memory, and another RAM with 16 bytes that is used for the Data memory. We can refer to these registers by calling r0-r3. The instructions that the CPU can complete are, loading data from memory, storing memory to data, adding using registers and subtracting using registers. It should be noted that adding and subtracting cannot be done with immediate numbers and must have data loaded into the registers.

Instruction Set

Instruction	Opcode	Binary Encoding	Instruction Breakdown
diff rt rn rm (diff r0 r1 r2)	00	00000110	The first two bits are 00 and this will represent that the subtraction function will be computed. Since there are 4 different instructions two bits are required to represent all different instructions. The next two bits represent the target register, 00, that the result of the difference will be stored in. Since there are 4 different registers that need to be represented, two bits are enough. The next two bits, 01, represent the leading number that is to be subtracted. The last two bits, 10, represent the trailing number that is to be subtracted.
sum rt rn rm (sum r1 r2 r3)	01	01011011	The first two bits are 01 and will represent that the addition function will be computed. The binary encoding follows the same structure as the diff function for the last 6 bits.
load rt imm4 (load r3 9)	10	10111001	The first two bits are 10 and will represent the load instruction. The next two bits, 11, will represent what register you want the requested data loaded into. Since there are only 4 registers two bits are enough to target this. Finally there are 4 bits left to calculate the offset, therefore the range that the data can go to is 0-15 or 4 bits worth of data.

store rt imm4 (store r2 5)	11	11100101	The first two bits are 11 and will represent the store instruction. The store instruction follows the same pattern as the load instruction.
----------------------------	----	----------	---

To go above and beyond in this project, we had decided to implement 3 different additional features. The first was to allow users to generate their own data within the text file. The next thing was adding the store function which would allow the user to store their register values in the data memory. And finally we added LED output lights to show what the end result of the ALU and load instruction would be!