The Add Bass CPU User Manual

Architecture of the CPU:
The Add Bass CPU (ABCPU) contains four 8-bit general-purpose registers, which can be referred to in an assembly program by “R0”, “R1”, “R2”, and “R3”, with R0 being the topmost register and R3 being the bottommost.

ABCPU can perform addition and subtraction, store the value of registers into memory, and load data from memory into registers. The following table details the CPU’s instruction set architecture:

	Bits															
Instruction	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
LDR Rt Rn imm	1	0	1	1	imm								Rn		Rt	
LDR loads data stored in RAM at memory address Rn+imm into register Rt.																
STR Rt Rn imm	0	1	1	0	imm								Rn		Rt	
STR stores data in register Rt into RAM at memory address Rn+imm.																
ADD Rd Rn imm	0	0	1	1	imm								Rn		Rd	
ADD computes Rn+imm and stores the sum in register Rd.																
SUB Rd Rn imm	0	0	0	1	imm								Rn		Rd	
SUB computes Rn-imm and stores the difference in register Rd.																

* All uses of “Rn” in this table refer to the value stored in register Rn, not the register itself.

- `I[15]` *(a.k.a. bit 15 of the binary encoding)* represents whether the instruction is reading from memory. This bit is `1` when the instruction is LDR (since it’s the only instruction that reads from memory) and `0` for all other instructions.

- `I[14]` represents whether the instruction is writing to memory. This bit is `1` when the instruction is STR (since it's the only instruction that writes to memory) and `0` for all other instructions.
- `I[13]` represents what arithmetic operation is performed by the ALU. A value of `0` means subtraction and a value of `1` means addition. When the instruction is LDR, STR, or ADD, this bit is `1` (LDR performs addition to calculate the memory address to retrieve data from, STR performs addition to calculate the memory address to write data into, and ADD performs addition for obvious reasons). When the instruction is SUB, this bit is `0`.
- `I[12]` represents whether the instruction writes to a register. A value of `0` means the instruction (STR) does not write to a register and a value of `1` means the instruction (LDR, ADD, SUB) does write to a register.
- `I[11:4]` represents the immediate number. ADD and SUB use these bits for the second operand in its arithmetic operation. LDR and STR use these bits for the memory address offset. `imm` takes up 8 bits of the binary encoding because the registers are also 8 bits wide. Since there are only 8 bits, it can represent numbers 0 to 255. Please don't input negative numbers because I didn't program the assembler to deal with a minus sign😢.
- `I[3:2]` represents `Rn`, or the read register. ADD and SUB use the value of this register as the first operand in its arithmetic operation. LDR and STR use the value of this register to get the memory address. `Rn` is 2 bits wide since that is the minimum number of bits required to represent all four registers.
- `I[1:0]` represents `Rt/Rd`, or the write register. `Rt/Rd` is 2 bits wide since that is the minimum number of bits required to represent all four registers.

How to use the Assembler:

Write the assembly program in a text file. In order for the assembler to translate it correctly, the following requirements must be met:

- The name of the text file is “program.txt”.
- There are no extra, empty lines.
- All lines begin with a complete instruction. *The assembler splits each line by spaces and only parses the first 4 strings, so any text that comes one space after the final operand is ignored. This means you can write comments immediately following an instruction, but comments cannot be written on a line by themselves.*
- Immediate numbers are non-negative and between the range 0 - 255.

To translate the text file into an image file and to execute the program:

1) Put the Python assembler file and the text file containing your assembly program into the same folder.
2) Open the folder in the terminal.
3) Type “python main.py” or “python3 main.py” to execute the assembler. A new text file should appear in the folder named “image.txt”.
4) Go to the RAM labeled “INSTRUCTION MEMORY” in the circuit, right click it, press ‘Load Image’ and select the newly created “image.txt” file.
5) Now you can run the program!

Sample program (included in the .zip file) :

```
ADD R0 R0 5 // R0 = 5 (hex value 0x05)
STR R0 R3 0 // Store R0 at 0x00
LDR R1 R3 0 // Load data at 0x00 into R1
ADD R1 R1 5 // R1 = 10 (hex value 0x0a)
STR R1 R3 1 // Store R1 at 0x01
LDR R2 R3 1 // Load data at 0x01 into R2
ADD R2 R2 5 // R2 = 15 (hex value 0x0f)
STR R2 R3 2 // Store R2 at 0x02
LDR R3 R3 2 // Load data at 0x02 into R3
ADD R3 R3 5 // R3 = 20 (hex value 0x14)
SUB R3 R3 1 // R3 = 19 (hex value 0x13)
SUB R2 R2 1 // R2 = 14 (hex value 0x0e)
SUB R1 R1 1 // R1 = 9  (hex value 0x09)
SUB R0 R0 1 // R0 = 4  (hex value 0x04)
```

When this program finishes executing:

- Registers R0, R1, R2, and R3 should have values of 0x04, 0x09, 0x0e, and 0x13, respectively.
- The data RAM should contain the values 0x05, 0x0a, and 0x0f, at addresses 0x00, 0x01, and 0x02, respectively.