Name of CPU: Pizza'nTacoCPU
Name: Miguel Merlin, Alice Agnoletto
Pledge: I pledge my honor that I have abided by the Stevens Honor System.

Collective job description:
In this class project, Alice and Miguel designed and built their own CPU. Alice and Miguel's role is to work collaboratively to make sure the CPU can handle various tasks smoothly.
Alice and Miguel are responsible for defining the instruction set that the CPU will support. This involves carefully specifying the operations and their corresponding opcodes, which serve as the machine language for the CPU. They then assigned unique binary opcodes to each instruction, ensuring that there is no overlap or ambiguity. These opcodes are essential for instructing the CPU on which operation to execute.

Job description Miguel Merlin:
Miguel's main job was developing the assembler component, an integral part of our CPU design. The assembler serves as the bridge between human-readable assembly code and the machine code understood by our custom CPU architecture.

Job description Alice Agnoletto:
Alice's main job was to implement the CPU. Alice brought the CPU design to life using Logisim Evolution. She translates the design into a functional CPU model by configuring and interconnecting the various components. Extensive testing and debugging ensure the CPU's correctness. Moreover, Alice implemented an extra led display to show the calculation results, helped debug the assembler, and extensively tested each instruction combination.

Documentation

1. **How to assemble {name} code.**

To compile the program, run the following command:
gcc -o assembler assembler.c

2. **How to load program instructions**

The program instructions will be written in a text file inside the same directory as the assembler.
To execute the program, specify the name of the text file as a variable in the following format:
./assembler <filename>
The text file will contain the program instructions in this format:

```
.data
   test: .quad 5
.text
   mov x0, 3
   mov x3, 10
   sub x1, x0, x3
   sub x2, x3, x0
   sub x2 , x2, 5
```

Once the assembler has been compiled and run, the program will generate two text files: *data.txt* and *instructions.txt*. These two text files will contain the data and the instructions translated into hex codes. Those instructions can be directly inputted into the Instruction Memory and will be effectively read and executed by the CPU.

3. How to load program to data memory

Architecture Description
The CPU has 4 general purpose registers labeled as X0, X1, X2, and X3. The supported functions are the following:

- LDR (Load from memory)
- STR (Store in memory)
- ADD (addition)
- SUB (subtraction)

Format of instructions
Load from Memory
LDR Target_Register, [Address, Offset]

LDR Rt, [Rn, Rm]
LDR Rt, [Rn, imm7]

Store in Memory
STR Target_Register [Address, Offset]

STR Rt, [Rn, imm7]
Note: STR Rt, [Rn, Rm] **Not Supported**

Add two numbers
ADD Destination_Register, Summand_1, Summand_2

ADD Rd, Rn, Rm
ADD Rd, Rn, imm7

Subtract two numbers
SUB Destination_Register, minuend, subtrahend

SUB Rd, Rn, Rm
SUB Rd, Rn, imm7

Binary Encodings
Encoding of arithmetic instructions with register operands

Instruction	opcode	Rm	00000	Rn	Rt
ADD Rd, Rn, Rm	01000				
SUB Rd, Rn, Rm	11000				

Encoding of arithmetic instructions with register operands and immediate

Instruction	opcode	imm7	Rn	Rt
ADD Rd, Rn, imm7	01100			
SUB Rd, Rn, imm7	11100			

Encoding of memory accessing instructions with offset in register

Instruction	opcode	Rm	00000	Rn	Rt
LDR Rt, [Rn, Rm]	01010				

Encoding of memory accessing instructions with offset as immediate

Instruction	opcode	imm7	Rn	Rt
LDR Rt, [Rn, imm7]	01110			
STR Rt, [Rn, imm7]	00101			

Description of Binary Encoding

opcode

The opcode is 5 bits that correspond to a control signal needed in the datapath. The control signals fetched from the opcode are the following:

Type of Operation (TO) [4]
Write to Register (WriteReg) [3]
Immediate (imm) [2]
Read from memory (ReadMem) [1]
Write to memory (WriteMem) [0]

Rm

Rm is 2 bits because the CPU contains 4 registers in total. To represent the 4 registers $log_2 4 = 2$ bits are needed.

Rn

Rm is 2 bits because the CPU contains 4 registers in total. To represent the 4 registers $log_2 4 = 2$ bits are needed.

Rt

Rm is 2 bits because the CPU contains 4 registers in total. To represent the 4 registers $log_2 4 = 2$ bits are needed.

imm7

Considering that all the instructions contain the same number of bits and need to be multiples of 8, then a 7-bit immediate is needed to make the size of the instruction encoding equal to 16 bits.

Assembler Extra Functions:

The assembler is coded in a way that the functions [MOV, ADR] can be written and can be uploaded into memory and can be recognized by the circuit.
Moreover, the operations [ADD, SUB, MOV] will display the result in a LED display.

DEMO PROGRAM

Make sure sample.txt and assembler.c are in the same directory.

sample.txt

```
.data
   test: .quad 5

.text
   ldr x0, [x1, 0]
   add x0, x0, 10
   mov x3, 8
   sub x0, x0, x3
   str x0, [x1, 0]
```

Compile the assembler
gcc -o assembler assembler.c

Run the assembler on the sample file
./assembler sample.txt

The assembler will output 2 files:

- instructions.txt
- data.txt

In Logisim, load the instructions.txt image to the instruction memory and the data.txt to the data memory.

Since the address of 5 in the data memory is 0x000, and the registers start with a value of 0x0000 then we do not need to change the address in register 1.

Run the simulation.