Instruction Syntax

The instructions follow a nearly identical syntax to ARM. Each instruction starts with a 3-letter mnemonic, followed by multiple parameters all separated by a comma. This comma is important because my assembler splits the instructions using them as pivot points. Registers are referred to as X0, X1, X2, and X3. One notable difference is that I dropped the usage of brackets for simplicity, meaning that LDR and STR would have similar syntax to something like ADD, with the offset being the last parameter. A byte offset must always be provided when addressing memory. If not using any offset you can use ", 0" following the register. See the provided instruction file for an example.

How to use the Assembler

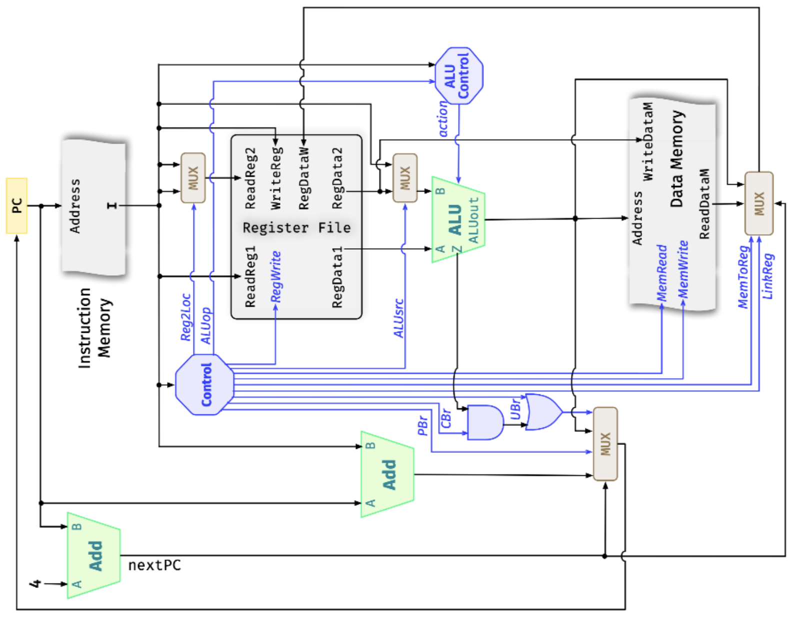
To assemble instructions such that they can be executed on the CPU they must be run through the python script. This script is labeled "assembler.py" and can be found in this directory. The assembler.py file takes in one command line argument: the name of the text file containing instructions. Place the file in the current directory with assembler.py and run:

- python assembler.py instructions.txt

where instruction.txt is the name of the file containing instructions. Afterwards a file labeled "image.txt" will be produced containing the machine code required for the CPU. This can be loaded into the instruction memory within Logisim to finally execute!

Architectural Description

This CPU contains 4 general purpose registers referred to as X0-X3 within the instructions. The CPU can perform addition and multiplication as well as move, load, and store data. It consists of 6 control signals which will be described in more detail in the instruction definition section. Each instruction is 16 bits, meaning my CPU has a 16-bit splitter coming from the 256 x 16 instruction memory. The data memory is composed of a 256 x 8 ram, or 256 addressable bytes. The data memory has asynchronous read turned to on, allowing data read during LDR instruction to be written back at the end of the same clock cycle. In general, my CPU follows a very similar structure to the below diagram, except it lacks branching and has an extra wire/multiplexor for MOV instruction:



Instruction Manual

As previously stated, each instruction is 16 bits. These 16 bits are split into the following categories:

- OPCODE:IMMEDIATE:REG_1:REG_DST

or

- OPCODE:REG_2:IRRELEVANT:REG1:REG_DST

Here is a brief description of each:

- <u>OPCODE</u>: This is a 6 bits opcode unique to each instruction. The bits within this code are directly converted to control signals using a splitter (1 is left-most bit):
  1. **MemRead** – to read to memory (0 = no read; 1 = read)
  2. **MemWrite** – to write to memory (0 = no write; 1 = write)
  3. **RegWrite** – to write to a register (0 = no write; 1 = write)
  4. **ALUop** – to add or multiply (0 = add; 1 = multiply)
  5. **ALUsrc** – to use an immediate (0 = register; 1 = immediate)
  6. **MOVctrl** - for MOV source to bypass ALU and memory (0 = disabled, 1 = bypass)

  Since there are 6 control signals this section must be 6 bits

- <u>IMMEDIATE</u>: This is a 6-bit immediate number to be used in specific instructions that use a constant number. When using a 6-bit opcode and two 2-bit registers this leaves 6 bits remaining, hence the size is 6 bits.

Note: The below registers are 2 bits since we have 4 registers, meaning we need to accommodate for 00, 01, 10, and 11.

- <u>REG_1</u>: 2-bits describing the first register used within an instruction.
- <u>REG_2</u>: 2-bits describing the second register used within an instruction.
- <u>RED_DST</u>: 2-bits describing the destination register for an instruction. More specifically, the register to be written back to in all cases except STR. With STR this will be the register to read and store.

- IRRELEVANT: Composed of 4 random bits to fill the excess space when not using an immediate number. 4-bits is because the immediate number is 6-bits and we are using the upper 2 for REG_2, meaning that we have 4 irrelevant bits.

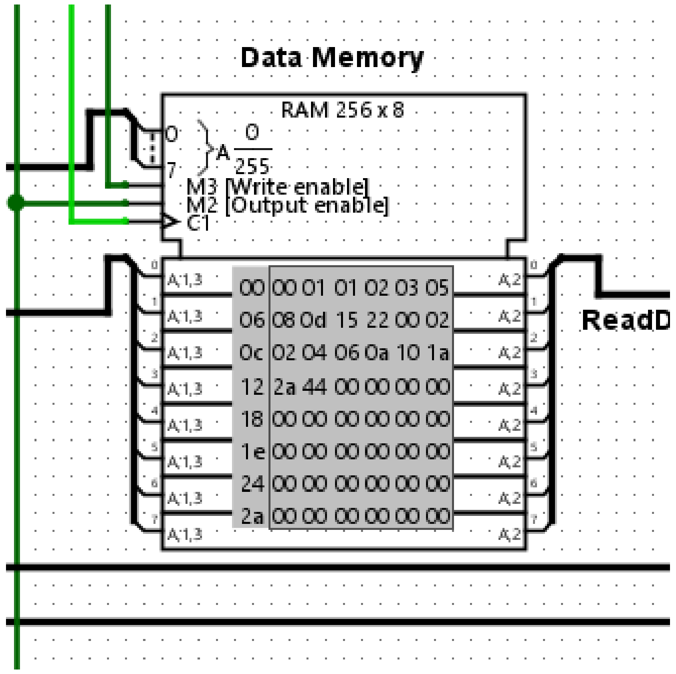
With the machine code composition described above and the number of bits required for each section stated, below is a table of all implemented instructions (9 total).

		IMMEDIATE (6)			
	OPCODE (6)	REG_2 (2)	IRRELEVANT (4)	REG_1 (2)	REG_DST (2)
ADD Xd, Xn, Xm	001000	Xm	0000	Xn	Xd
ADD Xd, Xn, imm6	001010	imm6		Xn	Xd
MUL Xd, Xn, Xm	001100	Xm	0000	Xn	Xd
MUL Xd, Xn, imm6	001110	imm6		Xn	Xd
LDR Xd, Xn, Xm	101000	Xm	0000	Xn	Xd
LDR Xd, Xn, imm6	101010	imm6		Xn	Xd
STR Xd, Xn, imm6	010010	imm6		Xn	Xd
MOV Xd, Xn	001001	Xn	0000	00	Xd
MOV Xd, imm6	001011	imm6		00	Xd

Note: for MOV we use REG_2 as the source register instead of REG_1, this is since RegData2's wire goes into the multiplexor with the immediate number. This syntax ensures that only one wire is needed to get either the immediate or register data moved past the ALU and memory with MOVctrl. Something else worth mentioning is that STR only has an immediate version. This is because we cannot read 3 registers in once cycle with only 2 read ports.

Sample Program

I have attached a sample program labeled instructions.txt with this project. This program calculates the first 10 Fibonacci numbers and stores them in memory addresses 0 – 9. Afterwards, the program loads each number one by one, doubles it, and then stores it back into addresses 10-19. The result has 20 numbers stored into memory, with the first 10 being the Fibonacci sequence and the next 10 being the Fibonacci sequence doubled. Below is a photo after the program is run:



```
00 00 01 01 02 03 05
06 08 0d 15 22 00 02
0c 02 04 06 0a 10 1a
12 2a 44 00 00 00 00
18 00 00 00 00 00 00
1e 00 00 00 00 00 00
24 00 00 00 00 00 00
2a 00 00 00 00 00 00
```